
Spoofax Documentation

Release 2.5.16

MetaBorg

May 09, 2022

Contents

1	Examples	1
1.1	MetaBorg: Bootstrapping Spoofax	1
1.2	Spoofax in Production	1
1.3	Spoofax in Education	2
1.4	MetaBorgCube	2
2	Publications	3
3	Installing Spoofax	5
3.1	Requirements	5
3.2	Installing the Spoofax Eclipse Plugin	5
4	Creating a Language Project	9
4.1	Installation	9
4.2	Hello World Language	9
4.3	How to proceed?	10
5	Using the API	11
5.1	Requirements	11
5.2	Project Setup	11
5.3	Using the API	12
5.4	How to proceed?	14
6	Getting Support	17
6.1	Documentation	17
6.2	Bug Reports and Feature Requests	17
6.3	Getting Help	18
7	Language Definition with Spoofax	19
7.1	Syntax Definition	20
7.2	Abstract Syntax	24
7.3	Transformation	25
7.4	Static Semantics	26
7.5	Dynamic Semantics	27
7.6	Editor Services	29
7.7	Testing	31
7.8	Exercises	31

8	Abstract Syntax with ATerms	35
8.1	Annotated Terms	35
8.2	Signatures	37
8.3	Term Rewriting	38
8.4	Term API	38
9	Syntax Definition with SDF3	39
9.1	SDF3 Overview	39
9.2	SDF3 Reference Manual	41
9.3	SDF3 Examples	58
9.4	SDF3 Configuration	58
9.5	Migrating SDF2 grammars to SDF3 grammars	60
9.6	SDF3 Bibliography	60
10	Static Semantics Definition with NaBL2	63
10.1	Introduction	63
10.2	Language Reference	63
10.3	Stratego API	77
10.4	Configuration	85
10.5	Examples	87
10.6	Bibliography	87
10.7	NaBL	87
11	Static Semantics Definition with Statix	93
11.1	Getting Started	93
11.2	Debugging	94
11.3	Language Reference	103
11.4	Stratego API	110
11.5	Signature Generator	110
11.6	Rename Refactoring	114
11.7	NaBL2 Migration Guide	115
11.8	Migrating to the Concurrent Solver	119
12	Data Flow Analysis Definition with FlowSpec	123
12.1	Introduction	123
12.2	Language Reference	124
12.3	Stratego API	130
12.4	Configuration	137
12.5	Examples (under construction)	139
12.6	Bibliography	139
13	Transformation with Stratego	141
13.1	Stratego Tutorial/Reference	141
13.2	The Stratego Library	219
13.3	Concrete Syntax in Stratego Transformations	233
13.4	Incremental Compilation for Stratego	236
14	DynSem	239
14.1	DynSem Language Reference	239
14.2	DynSem tutorials	254
14.3	Support	275
15	ESV	277
15.1	Structure	277
15.2	Configuration Sections	278

15.3	Main File	285
16	Language Testing with SPT	287
16.1	Test suites	287
16.2	Test Expectations	290
16.3	Running SPT Tests	295
16.4	The SPT Framework	298
17	Build and Develop Language Projects	299
17.1	Maven Builds	299
17.2	Continuous Integration	302
17.3	IntelliJ IDEA Environment	305
18	Configuring Language Projects	319
18.1	Configuration Reference	319
18.2	Language Extension	328
19	Sunshine: the Spoofox Command-Line Interface	331
19.1	Installation	331
19.2	Usage	331
19.3	Nailgun	331
20	Core API Manual	333
20.1	Introduction	333
20.2	Architecture	335
20.3	Concepts	337
20.4	Design	339
20.5	Dependency Injection	339
20.6	Services	339
20.7	Language Processing	343
20.8	Core Extensions	347
21	Spoofox Development Manual	353
21.1	Introduction	353
21.2	Requirements	354
21.3	Maven	355
21.4	Building	359
21.5	Developing	361
21.6	Naming Conventions	364
21.7	Documentation	365
21.8	Releasing Spoofox	367
21.9	IntelliJ IDEA Plugin Internals	371
22	Latest Stable Release	381
22.1	Eclipse plugin	381
22.2	IntelliJ plugin	382
22.3	Command-line utilities	382
22.4	Core API	382
22.5	StrategoXT	382
22.6	Maven artifacts	382
23	Current Development Release	383
23.1	Eclipse plugin	383
23.2	IntelliJ plugin	384
23.3	Command-line utilities	384

23.4	Core API	384
23.5	StrategoXT	384
23.6	Maven artifacts	384
23.7	Build farm	384
24	All Releases	385
24.1	Spoofax 2.5.16	385
24.2	Spoofax 2.5.15	387
24.3	Spoofax 2.5.14	389
24.4	Spoofax 2.5.13	390
24.5	Spoofax 2.5.12	392
24.6	Spoofax 2.5.11	394
24.7	Spoofax 2.5.10	395
24.8	Spoofax 2.5.9	398
24.9	Spoofax 2.5.8	400
24.10	Spoofax 2.5.7	402
24.11	Spoofax 2.5.6	404
24.12	Spoofax 2.5.5	405
24.13	Spoofax 2.5.4	407
24.14	Spoofax 2.5.3	409
24.15	Spoofax 2.5.2	411
24.16	Spoofax 2.5.1	413
24.17	Spoofax 2.5.0	414
24.18	Spoofax 2.4.1	418
24.19	Spoofax 2.4.0	419
24.20	Spoofax 2.3.0	421
24.21	Spoofax 2.2.1	424
24.22	Spoofax 2.2.0	425
24.23	Spoofax 2.1.0	428
24.24	Spoofax 2.0.0	430
24.25	Spoofax 2.0.0-beta1 (07-04-2016)	433
24.26	Spoofax 1.5.0 (18-12-2015)	434
24.27	Spoofax 1.4.0 (06-03-2015)	436
24.28	Spoofax 1.3.1 (09-12-2014)	437
24.29	Spoofax 1.3.0 (12-11-2014)	438
24.30	Spoofax 1.2.0 (13-08-2014)	440
24.31	Spoofax 1.1.0 (25-03-2013)	443
24.32	Spoofax 1.0.2 (15-02-2012)	444
24.33	Spoofax 1.0.0 (28-12-2011)	445
24.34	Spoofax vNext	445
25	Migration Guides	447
25.1	Spoofax 2.5.15 Migration Guide	447
25.2	Spoofax 2.5.10 Migration Guide	447
25.3	Spoofax 2.5.5 Migration Guide	449
25.4	Spoofax 2.2.0 Migration Guide	450
25.5	Spoofax 2.1.0 Migration Guide	451
25.6	Spoofax 2.0.0 Migration Guide	453
25.7	New Completions Framework	460
25.8	Directory structure migration	466
25.9	Spoofax vNext Migration Guide	468
26	Contributions	469
26.1	Universities	469

26.2	Funding	469
26.3	Tooling	470
26.4	Contributors	470
27	Developing Software Languages	473
28	Creating Full-Featured Editors	475
29	Declare Your Language	477
30	A Platform for Language Engineering	479
31	A Platform for Research	481
	Bibliography	483

Here we collect a list of (pointers to) languages developed with Spoofax. Not all projects have publicly available sources. Let us know if you know of more projects to list here.

1.1 MetaBorg: Bootstrapping Spoofax

Spoofax is bootstrapped; all meta-languages used in Spoofax are developed in Spoofax. You can find the repositories of these languages along with the Spoofax run-time system on the [MetaBorg](#) organization on github. The *introduction to Spoofax development* section provides an overview of these repositories.

1.2 Spoofax in Production

Spoofax is used for the development of several languages that are used in the production of software systems.

WebDSL A web programming language <<http://webdsl.org/>>

IceDust A data modeling language supporting relations, multiplicities, derived values and configurable strategies for calculation of derived values <<https://github.com/MetaBorgCube/IceDust>>

Green-Marl A DSL for graph analysis developed at Oracle Labs for [running graph algorithms in PGX](#)

PGQL Property Graph Query Language A DSL for graph querying developed at Oracle Labs as part of [Parallel Graph AnalytiX](#) framework. The Spoofax definition is part of the open source implementation <<https://github.com/oracle/pgql-lang>>.

LeQuest A DSL for modeling medical equipment interfaces for development of training software. This is a proprietary language.

1.3 Spoofax in Education

We use Spoofax in education at TU Delft in a master's course [compiler construction course](#) and in a bachelor course on concepts of programming languages. The two main example languages used in that course are Mini Java and Tiger:

Mini Java A subset of the object-oriented Java language defined as assignment in Appel's book. The sources are not published since our students need to produce this. But you can follow [the assignments](#).

Jasmin A Spoofax editor for Jasmin, an assembler for the Java Virtual Machine, which is used as target language for the Mini Java compiler. [<https://github.com/MetaBorgCube/spoofax-jasmin>](https://github.com/MetaBorgCube/spoofax-jasmin)

Tiger A functional language used as example in the ML edition of Appel's book. The [metaborg-tiger](#) repository provides a complete definition of syntax in SDF3, static semantics in NaBL2, and dynamic semantics in DynSem.

PAPLJ Another subset of Java based on an assignment for Shriram Krishnamurthi's book on Programming and Programming Languages. [<https://github.com/MetaBorgCube/metaborg-papl>](https://github.com/MetaBorgCube/metaborg-papl)

1.4 MetaBorgCube

The [MetaBorgCube](#) github organization is a collection of language projects developed with Spoofax. These are often research or demonstration projects exploring (some aspect of) language definition with Spoofax. The projects are (stuck) at different stages of development, not necessarily using the latest version of Spoofax, and at different stages of maintenance (decay). Consider these projects as inspiration rather than as a source of working code. Some repositories are private; let us know if you are interested. Here is a selection:

Simpl A small imperative language to demonstrate DynSem [<https://github.com/MetaBorgCube/simpl>](https://github.com/MetaBorgCube/simpl)

QL/QLS Questionnaire language used as example language in the language workbench challenge [<https://github.com/MetaBorgCube/metaborg-ql>](https://github.com/MetaBorgCube/metaborg-ql)

Grace A programming language designed for programming education. The Spoofax project defines a syntax definition, desugaring, and operational semantics for the language. [<https://github.com/MetaBorgCube/metaborg-grace>](https://github.com/MetaBorgCube/metaborg-grace)

Go A subset of the Go programming language with a translation to JavaScript (Private Repository)

MetaC A version of the C programming language with modules and domain-specific language extensions [<https://github.com/MetaBorgCube/metac>](https://github.com/MetaBorgCube/metac)

Pascal A syntax and static semantics of the Pascal programming language [<https://github.com/MetaBorgCube/metaborg-pascal>](https://github.com/MetaBorgCube/metaborg-pascal)

CHAPTER 2

Publications

The concepts and techniques underlying the design and implementation of Spoofax are described in scientific publications in conference proceedings and journals. While those publications are typically (somewhat) behind when it comes to technical details, this documentation cannot replace that body of work when it comes to exposition of concepts and ideas. We recommend students of Spoofax to explore the literature.

The work on Spoofax has its origins in the ASF+SDF MetaEnvironment [\[R4\]](#) and work on the SDF2 syntax definition formalism [\[R5\]](#). Experience with rewriting in ASF lead to the development of the Stratego transformation language [\[R6\]](#).

The Spoofax language workbench was first developed as an IDE extension of Stratego/XT [\[R1\]](#), a tool set for program transformation based on SDF2 and Stratego. The main publication about Spoofax is [\[R2\]](#), which was based on Spoofax 1.0 and develops the requirements and architecture for a language workbench supporting agile language development.

In [\[R3\]](#) we develop a vision and first prototype to take Spoofax to the web; realizing that vision is still work in progress. The vision for a *language designer's workbench* is outlined in [\[R7\]](#). That vision drives the current (May 2017) ongoing development to enable higher-level definition of static and dynamic semantics in Spoofax.

We maintain a complete [bibliography](#) of research results on [researchr](#). In the chapters on specific components, we discuss their history and related publications.

CHAPTER 3

Installing Spoofax

Spoofax is distributed as an Eclipse plugin. This guide shows how to download, install, and run Spoofax in Eclipse.

3.1 Requirements

Spoofax runs on the major operating systems:

- Windows (32 and 64 bits)
- Linux (32 and 64 bits)
- macOS (Intel only)

Spoofax requires a working internet connection to download several libraries when it is first started. These libraries are cached afterwards, and only need to be re-downloaded when you update Spoofax.

3.2 Installing the Spoofax Eclipse Plugin

3.2.1 Using Homebrew (macOS)

On *macOS* Spoofax can be installed easily using [Homebrew](#). For other platforms, or manual installation, follow the *Download (all platforms)* instructions below.

Install the *latest release* of Spoofax Eclipse as follows:

```
brew tap metaborg/metaborg
brew cask install spoofax
```

The optional command-line tools are installed with:

```
brew install strategoxt
```

Continue at [Running Eclipse](#).

Warning: Upgrading the Spoofax cask using `brew cask upgrade --greedy` will lose all manually installed plugins. It is recommended to use Eclipse update sites to keep Spoofax up-to-date.

3.2.2 Download (all platforms)

To get started with Spoofax, download an Eclipse Oxygen installation with Spoofax preinstalled for your platform:

- [Windows 32-bit](#) with embedded JRE
- [Windows 64-bit](#) with embedded JRE
- [Linux 64-bit](#) with embedded JRE
- [macOS Intel](#) with embedded JRE

These are bundled with an embedded Java Runtime Environment (JRE) version 8, such that a JRE on your system is not required. If your system has a JRE of version 8 or higher installed, and would rather use that, use the following download links instead:

- [Windows 32-bit](#)
- [Windows 64-bit](#)
- [Linux 64-bit](#)
- [macOS Intel](#)

3.2.3 Unpack

Unpack the downloaded archive to a location with write access, since Eclipse requires write access to the unpacked Eclipse installation.

Warning: On *Windows*, do **not** unpack the Eclipse installation into `Program Files`, because no write access is granted there, breaking both Eclipse and Spoofax.

Warning: On *macOS Sierra (10.12)* and above, you must move the unpacked `spoofax.app` file to a different location (such as `Applications`) after unpacking, to prevent [App Translocation](#) from moving the app into a read-only filesystem, breaking Eclipse and Spoofax.

Alternatively, you can prevent App Translocation by clearing attributes from the application. To do this, open the Terminal, navigate to the directory where the `spoofax.app` is located, and execute:

```
xattr -rc spoofax.app
```

3.2.4 Running Eclipse

Start up Eclipse, depending on your operating system:

- Windows: open `spoofax/eclipse.exe`
- Linux: open `spoofax/eclipse`

- macOS: open **spoofax.app**

Warning: Do not update Eclipse with *Help → Check For Updates*, as it will update Eclipse to newer major versions which are not always backwards compatible, and which require a JRE of version 11 or higher which we do not bundle (we bundle JRE8) with Eclipse.

Note: On *macOS*, if Eclipse cannot be opened because it is from an *unidentified developer*, right click `spoofax.app` and choose *Open* to grant permission to open Eclipse.

If Eclipse cannot be opened because it is *damaged*, open the Terminal, navigate to the directory where `spoofax.app` is located, and execute:

```
xattr -rc spoofax.app
```

This will clear the attributes that Eclipse has been downloaded from the internet, and grant permission to open Eclipse.

Note: If you downloaded Spoofax Eclipse without an embedded JRE, it may not start or give an error *A Java Runtime Environment (JRE) or Java Development Kit (JDK) must be available in order to run Eclipse* or *To open “spoofax” you need to install the legacy Java SE 6 runtime*. To fix this, specify the path to a valid JDK using the `-vm` option at the start of `eclipse.ini` (`spoofax.app/Contents/Eclipse/eclipse.ini` on macOS).

For example, to specify the current JDK installed by [Sdkman](#), add this at the top of your `eclipse.ini`:

```
-vm
/Users/myusername/.sdkman/candidates/java/current/lib/jli/libjli.dylib
```

See [this link](#) for more information.

Note: On *Ubuntu 16.04*, Eclipse is known to have problems with GTK+ 3. To work around this issue, add the following to `eclipse.ini`:

```
--launcher.GTK_version
2
```

before the line:

```
--launcher.appendVmargs
```

After starting up, choose where your workspace will be stored. The Eclipse workspace will contain all of your settings, and is the default location for new projects.

Some Eclipse settings unfortunately have sub-optimal defaults. After you have chosen a workspace and Eclipse has completely started up, go to the Eclipse preferences and set these options:

- *General → Startup and Shutdown*
 - Enable: *Refresh workspace on startup*
- *General → Workspace*
 - Enable: *Refresh using native hooks or polling*
- *Maven → Annotation Processing*

- Enable: *Automatically configure JDT APT*

3.2.5 Changing Eclipse Memory Allocation

By default a plain Eclipse has a maximum heap size of 1 GB. You may want to increase this limit. The default for the Eclipse application produced by Spoofax is 2 GB.

To run Eclipse once with a different memory limit, call it from the command-line like this:

```
eclipse [normal arguments] -vmargs -Xmx2G
```

If this works, you can permanently apply this limit in the `eclipse.ini` file in the Eclipse installation directory (macOS: `Contents/Eclipse` in the Eclipse package) by changing the `-Xmx` argument. For example:

```
-vmargs
[...]
-XstartOnFirstThread
-Xss16M
-Xms2G
-Xmx2G
-Dosgi.requiredJavaVersion=1.8
-server
```

-Xss the size of the thread stack

-Xms the initial size of the heap

-Xmx the maximum size of the heap

3.2.6 Further Instructions

Follow the *Getting Started guide* to get started with Spoofax in Eclipse.

Creating a Language Project

This guide will get you started with language development in Spoofax, within an Eclipse environment.

4.1 Installation

First follow the [Installation Guide](#) for instructions on how to download, install, and run Spoofax in Eclipse.

4.2 Hello World Language

To get you started, let's do the 'hello world' of language development; the hello world language. In Eclipse, open the new project dialog by choosing *File* → *New* → *Project* from the main menu. In the new project dialog, select *Spoofax* → *Spoofax language project* and press *Next* to open the wizard for creating a Spoofax language specification project. As project name, choose `helloworld`, which will automatically fill in the identifier, name, and extension of the language. Keep the defaults for the other fields and press *Finish* to create the project. Once the project has been created, open and expand it in the package or project explorer view.

The syntax for the language is specified in the `syntax/helloworld.sdf3` SDF3 file. SDF3 is our syntax definition language, from which we derive a parser, pretty-printer, and syntactic completions from your language. Currently, the syntax contains a single start symbol `Start`, and a production that accepts an empty program: `Start.Empty = <>`. Remove that production and replace it with the following productions:

```
Start.Program = <<Word> <Word>>
Word.Hello = <hello>
Word.World = <world>
```

This grammar accepts a program consisting of 2 words, where the words can be `hello` or `world`, with any number of layout characters (whitespace, tabs, empty lines, comments, etc.) in between.

To observe our changes to the grammar, we must first rebuild the project by selecting *Project* → *Build Project*. If this is greyed out, make sure that the project is selected in the project explorer.

Create a new file by choosing *File* → *New* → *File*, put the file at the root of the helloworld project and name it `test.hel`. Open that file and try out the parser by typing `hello world`, any combinations of the 2 words, and with or without layout between words.

If everything went well, the syntax highlighter will highlight the words in purple, which is the default highlighting color for keywords. To see the abstract syntax tree that the parser derives from your program, select *Spoofax* → *Syntax* → *Show parsed AST*. If you make an error in the program, for example `hello worl`, an error message will show up indicating where the error is.

4.3 How to proceed?

Guides for developing a language with Spoofax:

- [Declare Your Language](#) - This book has not been updated for Spoofax 2.0 yet, but most content still applies.

Reference manuals for our meta-languages:

- *SDF3*
- *Stratego*
- *NaBL*
- *NaBL2*
- *Statix*
- *Flowspec*
- *DynSem*
- *SPT*

Example language specifications:

- [paplj language](#)

This guide will get you started with the Spoofax Core API, within an Eclipse environment.

5.1 Requirements

Spoofax is written in Java, and thus runs on the major operating systems:

- Windows (32 and 64 bits)
- Linux (32 and 64 bits)
- Mac OSX (Intel only)

The Spoofax Core API is written in Java 8, and can be compiled with a Java Development Kit (JDK) of version 8 up to version 11. Higher JDK versions have not been tested, and may result in errors about unresolved Java base classes (e.g., `java.lang.Object`). You can download and install JDK 8 or JDK 11 from [AdoptOpenJDK](#), or get a proprietary release from [Oracle](#).

The Spoofax Core API is deployed as a set of Maven artifacts. We do not (yet) publish these artifacts to Maven Central, but rather to repositories on our own artifact server. To get access to our artifacts, read the [Using MetaBorg Maven artifacts](#) section. Adding our Maven repositories gives access to our artifacts.

In this guide, we will be using Eclipse to use the Core API, but any environment that works with Maven artifacts (e.g. IntelliJ, NetBeans, command-line Maven builds) will work. Download and install the Eclipse IDE for Java Developers from the [Eclipse website](#).

5.2 Project Setup

In Eclipse, open the new project dialog by choosing *File -> New -> Project* from the main menu. In the new project dialog, select *Maven -> Maven project* and press *Next* to open the wizard for creating a Maven project. Enable *Create a simple project (skip archetype selection)* and press *Next*.

Fill in the artifact details to your liking (see [Maven Naming Conventions](#)) for some info on these names), and press *Finish* to create the project. Once the project has been created, open and expand it in the package or project explorer view.

Open the `pom.xml` file and click the `pom.xml` tab to edit the source code of the POM file. Add the following snippet to the POM file:

```
<dependencies>
  <dependency>
    <groupId>org.metaborg</groupId>
    <artifactId>org.metaborg.spoofax.core</artifactId>
    <version>2.0.0</version>
  </dependency>
  <dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.1.2</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.2</version>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

This declares a dependency on version 2.0.0 of Spoofax Core, and a dependency on a logging framework so we get logging output from Spoofax Core. It also instructs Maven that this project requires a Java 7 compiler (instead of the default; Java 5).

Since the `pom.xml` file has changed, we need to update our Eclipse project. Right click the project in the package or project explorer view, select *Maven -> Update Project...*, and press *Ok*.

Now we can start using the Core API.

5.3 Using the API

5.3.1 Setup

To get started, we will download a language component, load it into Spoofax Core, and parse a file of that language.

First, let's create a main class as an entry point to the application. Right click `src/main/java` in the project, and select *New -> Class*. Call the class *Main* and press *Finish*. Add a main method to the class:

```
public static void main(String[] args) {
}
}
```

Second, let's download a language component that we can load into Spoofax Core. Download the [NaBL language](#) and store it in the `src/main/resources` directory of the project. Any resources stored in `src/main/resources`

are packaged into the JAR file of your application and are available at runtime.

To initialize Spoofax Core, create an instance of the `org.metaborg.spoofax.core.Spoofax` facade:

```
try(final Spoofax spoofax = new Spoofax()) {
    // Use Spoofax here
} catch (MetaborgException e) {
    e.printStackTrace();
}
```

We use the `try-with-resources` statement to initialize the Spoofax facade, such that it can clean up any temporary resources when the application shuts down. All code that uses Spoofax must go inside the statement, where the comment is.

Note: Use *Source -> Organize Imports* or `Ctrl+Shift+O` (`Cmd+Shift+O` on Mac OSX) to automatically add required imports when needed.

5.3.2 Loading a language

Now we can load the NaBL language into Spoofax Core. Spoofax Core uses [Apache VFS](#) as a file system abstraction, to be able to interact with different file systems. This means we must first get a `FileObject` (Apache VFS counterpart of `File`) that points to the NaBL language file we downloaded earlier. First get a URL to the NaBL language file which is on the classpath:

```
URL nablUrl = Main.class.getClassLoader().getResource(
    "org.metaborg.meta.lang.nabl-2.0.0.spoofax-language");
```

Then we resolve that to a `FileObject`, which points to the contents of the NaBL language implementation archive (which is actually a regular Zip file):

```
FileObject nablLocation = spoofax.resourceService.resolve("zip:" + nablUrl + "!/");
```

The `org.metaborg.core.resource.IResourceService` class is a service in Spoofax Core that provides functionality to retrieve `FileObjects`. In this case, we resolve to the contents inside the zip file. The `zip:` part indicates that we're using the `zip file system`, and the `! /` part indicates that we refer to the root path **inside** the zip file.

Spoofax Core has many services that provide small pieces of functionality. The `org.metaborg.core.language.ILanguageDiscoveryService` class is a service that discovers and loads languages, which we will use now to load the NaBL language:

```
Iterable<ILanguageDiscoveryRequest> requests =
    spoofax.languageDiscoveryService.request(nablLocation);
Iterable<ILanguageComponent> components =
    spoofax.languageDiscoveryService.discover(requests);
```

Since multiple languages can be requested from a single location, and multiple language components can be discovered from a single file, both methods return multiple values. However, we know that the NaBL language file only contains one language implementation, we can retrieve it with a couple of utility methods:

```
Set<ILanguageImpl> implementations = LanguageUtils.toImpls(components);
ILanguageImpl nabl = LanguageUtils.active(implementations);

if (nabl == null) {
    System.out.println("No language implementation was found");
}
```

(continues on next page)

(continued from previous page)

```
    return;
}
System.out.println("Loaded " + nabl);
```

Run the program by selecting *Run -> Debug As -> Java Application*. If all went well, `Loaded language impl. org.metaborg:org.metaborg.meta.lang.nabl:2.0.0` should appear in the log output.

5.3.3 Parsing a file

Now that the NaBL language is loaded into Spoofax Core, we can parse NaBL programs.

Right click `src/main/resources` and select *New -> File*, name the file `test.nabl` and press *Finish*. Open the file and fill it with the following content:

```
module test

namespaces Test1 Test2
```

To parse a file, we must first create a `org.metaborg.spoofax.core.unit.ISpoofaxInputUnit` which contains all information required to parse a file:

```
FileObject nablFile = spoofax.resourceService.resolve("res:test.nabl");
String nablContents = spoofax.sourceTextService.text(nablFile);
ISpoofaxInputUnit input = spoofax.unitService.inputUnit(nablFile, nablContents, nabl,
↳ null);
```

The `res` file system can be used to resolve files on the classpath. The catch clause must also be extended with `IOException` to handle the case where the text for the NaBL file cannot be retrieved:

```
} catch (MetaborgException | IOException e) {
    e.printStackTrace();
}
```

Then we pass the input to the `org.metaborg.core.syntax.ISyntaxService` for parsing:

```
ISpoofaxParseUnit output = spoofax.syntaxService.parse(input);
if (!output.valid()) {
    System.out.println("Could not parse " + nablFile);
    return;
}
System.out.println("Parsed: " + output.ast());
```

Run the program, `Parsed: Module("test", [Namespaces([NamespaceDef("Test1"), NamespaceDef("Test2")])])` should appear in the log output. Now you can optionally experiment a bit by making an error in the program, and printing the error messages from the opout.

5.4 How to proceed?

Todo: We are currently in the process of writing documentation, this section will be updated once we have more material.

The following manuals describe parts of the Spoofax Core API:

- *Services* - full list of available services in the Spoofox Core API
- *Core Extensions* - how to extend Spoofox Core

Spoofax is an open source project. We welcome contributions from the community.

Spoofax is developed by the TU Delft Programming Languages group. We do our best to make Spoofax usable by everyone and to help you when you encounter issues. However, our resources are limited, so please be patient.

6.1 Documentation

This documentation is the intended to be the first source of information on all things Spoofax. So, please read carefully.

6.2 Bug Reports and Feature Requests

To report bugs or request new features use our [YellowGrass](http://yellowgrass.org/project/Spoofax) issue trackers:

- Spoofax 2.0 and higher: [<http://yellowgrass.org/project/Spoofax>](http://yellowgrass.org/project/Spoofax)
- Spoofax 1.5 and lower: [<http://yellowgrass.org/project/SpoofaxLegacy>](http://yellowgrass.org/project/SpoofaxLegacy)

In the Spoofax 2.0 Eclipse plugin, use the main menu [Spoofax \(meta\) → Report issue](#) to report issues, it provides a link to the [Spoofax](#) project and the environment versions which you can copy paste.

When reporting bugs, please provide the following information such that we can easily reproduce and fix the bug: * Environment details (can be copy pasted from [Spoofax \(meta\) → Report](#) main menu)

- Eclipse/IntelliJ package and version
- Spoofax version
- Operating system name, architecture, and version
- High-level description of the bug
- How to reproduce the bug

- What the expected behavior is
- Any logs, exceptions, and stack traces related to the bug

If the documentation does not provide the answer you are looking for, you may find that it concerns an issue that was already reported by someone else.

6.3 Getting Help

If the documentation and issues do not provide the answer, and you are not sure you have encountered a bug, you can ask a question on our mailing list:

- [Spoofax mailing list](#)

These days we do most of our online communication within the development team on a private Slack organization. For communication with users we have created a public slack organization:

- [Spoofax Slack organization](#)

Please drop us a line to get an invitation.

If you are an experienced Spoofax user, please pitch in and help new users on these fora.

Language Definition with Spoofax

In this chapter we make a complete tour of language definition in Spoofax using Calc, a small calculator language, as example. In the sections of this chapter we define all aspects of the Calc language, including its concrete syntax, static semantics, dynamic semantics, testing, and configuration of the IDE. The source code of the language definition is available on [github](#). You can follow along by forking [the project](#) and building it in Spoofax.

The Calc language supports the following features

- Arithmetic with integer and floating point numbers with arbitrary precision
- Boolean values and boolean operators
- First-class (but non-recursive) polymorphic functions
- Variable bindings through top-level definitions and let bindings in expressions
- Type inference provides static type checking without explicit type annotations

The following Calc program calculates the monthly payments for a mortgage (according to [Wikipedia](#)):

```
monthly = \r Y P.  
  let N = Y * 12 in // number of months  
    if(r == 0) // no interest  
      P / N  
    else  
      let rM = r / (100 * 12) in // monthly interest rate  
      let f = (1 + rM) ^ N in  
        (rM * P * f) / (f - 1);  
  
r = 1.7;      // yearly interest rate (percentage)  
Y = 30;      // number of years  
P = 385,000; // principal  
  
monthly r Y P;
```

At the end of the chapter we give a list of exercises with ideas for extending the language. Those may be a good way to further explore the capabilities of Spoofax before venturing out on your own. If you do make one of those extensions (or another one for that matter), we would welcome pull requests for branches of the project.

Note that this is not an introduction to formal language theory, compiler construction, concepts of programming languages. Nor is this chapter an exhaustive introduction to all features of Spoofax.

7.1 Syntax Definition

We start with the definition of the concrete syntax of Calc using the syntax definition formalism SDF3. A syntax definition formalism is a language for defining all aspects of the syntax of a programming language. That goes well beyond a language for defining just the grammar; SDF3 also covers the definition of lexical syntax, AST constructors, formatting directives, and more.

7.1.1 Modules

To start with, syntax definitions consist of modules that can import other syntax modules. This is useful for dividing a large grammar into parts, but also for reusing a standard language component (e.g. expressions) between language definitions, or for composing the syntax of different languages. We first examine the `main syntax module` for Calc, which is named after the language and which imports module `CalcLexical` which defines the lexical syntax of the language:

```
module Calc
imports CalcLexical
```

The module defines the sorts `Program` and `Exp` as start symbols, which means that parsing starts with these sorts:

```
context-free start-symbols Program Exp
```

7.1.2 Context-Free Syntax

Syntactically, a language is a set of well-formed sentences. (In programming languages, sentences are typically known as programs.) Sentences are typically formed by composing different kinds of phrases, such as identifiers, constants, expressions, statements, functions, and modules. In grammar terminology there are two broad categories of phrases, *terminals* and *non-terminals*. In SDF3 we use *sorts* to identify both categories. For Calc we start with defining the `Program` and `Stat` sorts:

```
sorts Program Stat
```

A grammar consists of rules (known as productions) for composing phrases from sub-phrases. A Calc program consists of a list of statements that are either bindings that bind the value of an expression to a variable (identifier) or expression statements. This is defined by the following productions:

```
context-free syntax

Program.Program = <<{Stat "\n"}+>>

Stat.Bind = <<ID> = <Exp>;>
Stat.Exp = <<Exp>;>
```

If we take a closer look at the `Stat.Bind` production we see the following ingredients:

- The production defines one of two alternatives for the `Stat` sort. The alternatives of a sort are defined by separate productions. This makes it possible to introduce productions in an order that makes sense for presenting a language definition. Instead of defining all productions for a sort in one block, it is rather possible to define the

productions for different sorts that together define a language concept together. Furthermore, it enables *modular* definition of syntax.

- The body a production defines the composition of sub-phrases that it corresponds to. Thus, the body `<<ID> = <Exp>;>` defines a bind statement as the composition of an identifier, followed by an equal sign, followed by an expression, terminated by a semicolon.
- The body is known as a *template* and uses inverse quotation. The template makes everything inside literal elements of the text to be parsed, except for the quasi-quoted sorts (`<ID>` and `<Exp>`).
- The sub-phrases are implicitly separated by layout (whitespace and comments). The definition of layout is not built-in. We will see the definition of the layout for Calc when we discuss lexical syntax below.
- The constructor is used to construct abstract syntax tree nodes. Thus, the `Bind` constructor creates trees with two arguments trees for the identifier (`ID`) and expression (`Exp`) subtrees; in abstract syntax we leave out the literals and layout.

Note that a program is defined as a list of one or more statements, which could be expressed with the regular expression operator `+` as `Stat+`. The SDF3 notation `{Sort sep}+` denotes a list of `Sort` phrases *separated* by `sep`. For example, `{Exp " , "}` is a list of one or more expressions separated by commas. In the definition of statement lists we use a newline as separator. However, this does not imply that statements should be separated by newlines, but rather that newlines are inserted when formatting a program, as we will discuss below.

7.1.3 Expressions

Sorts and productions give us the basic concepts for defining syntax. Calc programs essentially consist of a sequence of expressions. So, the bulk of the its syntax definition consists of productions for various expression forms denoted by the `Exp` sort:

```
sorts Exp
context-free syntax
  Exp = <(<Exp>)> {bracket}
```

The *bracket* production defines that we can enclose an expression in parentheses. The `bracket` annotation states that we can ignore this production when constructing abstract syntax trees. That is, the abstract syntax tree for `(x + 1)` is the same as the abstract syntax tree for `x + 1`.

7.1.4 Operator Syntax

Operators are the workhorse of a language such as Calc. They capture the domain-specific operations that the language is built around. We start with the syntax of arithmetic operators:

```
context-free syntax // numbers
  Exp.Num = NUM
  Exp.Min = <-<Exp>>
  Exp.Pow = <<Exp> ^ <Exp>> {right}
  Exp.Mul = <<Exp> * <Exp>> {left}
  Exp.Div = <<Exp> / <Exp>> {left}
  Exp.Sub = <<Exp> - <Exp>> {left, prefer}
  Exp.Add = <<Exp> + <Exp>> {left}
```

Note that the concrete syntax is directly aligned with the abstract syntax. An addition is represented as the composition of two expression and the `+` symbol. This is best illustrated using term notation for abstract syntax trees. The term `C(t1, ..., tn)` denotes the abstract syntax tree for a production with constructor `C` and `n` sub-trees. For example, the term `Add(Num("1"), Var("x"))` represents the abstract syntax tree for the expression `1 + x`.

The consequence of this direct alignment is that the grammar is ambiguous. According to the `Exp.Add` production there are two ways to parse the expression `1 + x + y`, i.e. as `Add (Add (Num ("1"), Var ("x")), Var ("y"))` or as `Add (Num ("1"), Add (Var ("x"), Var ("y")))`.

A common approach to disambiguate the grammar for an expression language is by encoding the associativity and precedence of operators in the productions using additional sorts to represent precedence levels. However, that leads to grammars that are hard to understand and maintain and that do not have a one-to-one correspondence to the desired abstract syntax.

In SDF3, ambiguous expression syntax can be *declaratively* disambiguated using separate associativity and priority declarations. For example, the `Exp.Add` production above defines that addition is left associative. That is, the expression `1 + x + y` should be interpreted as `Add (Add (Num ("1"), Var ("x")), Var ("y"))`, i.e. $(1 + x) + y$. The other operators are disambiguated similarly according to [standard mathematical conventions](#). Note that power (exponentiation) is *right* associative, i.e. $x \wedge y \wedge z$ is equivalent to $x \wedge (y \wedge z)$.

comparison operators:

```
context-free syntax // numbers
Exp.Eq  = <<Exp> == <Exp>> {non-assoc}
Exp.Neq = <<Exp> != <Exp>> {non-assoc}
Exp.Gt  = [<Exp> > <Exp>] {non-assoc}
Exp.Lt  = [<Exp> < <Exp>] {non-assoc}
```

Non-assoc means that a phrase such as `a < b == true` is not syntactically well-formed. One should use parentheses, for example `(a < b) == true`, to explicitly indicate the disambiguation.

booleans:

```
context-free syntax // booleans

Exp.True  = <true>
Exp.False = <false>
Exp.Not   = <!  
<Exp>>
Exp.And   = <<Exp> & <Exp>> {left}
Exp.Or    = <<Exp> | <Exp>> {left}

Exp.If = <
  if (<Exp>)
    <Exp>
  else
    <Exp>
>
```

variables:

```
context-free syntax // variables and functions

Exp.Var = ID
Exp.Let = <
  let <ID> = <Exp> in
  <Exp>
>
Exp.Fun = <\\ <ID+> . <Exp>>
Exp.App = <<Exp> <Exp>> {left}
```

7.1.5 Disambiguation

priorities:

context-free priorities

```
Exp.Min
> Exp.App
> Exp.Pow
> {left: Exp.Mul Exp.Div}
> {left: Exp.Add Exp.Sub}
> {non-assoc: Exp.Eq Exp.Neq Exp.Gt Exp.Lt}
> Exp.Not
> Exp.And
> Exp.Or
> Exp.If
> Exp.Let
> Exp.Fun
```

sorts Type

context-free syntax

```
Type.NumT = <Num>
Type.BoolT = <Bool>
Type.FunT = [[Exp] -> [Exp]] {right}
Type      = <(<Type>)> {bracket}
```

template options

```
ID = keyword {reject}
```

7.1.6 Lexical Syntax

lexical syntax:

```
module CalcLexical
```

identifiers:

lexical syntax

```
ID = [a-zA-Z] [a-zA-Z0-9]*
```

lexical restrictions

```
ID -/- [a-zA-Z0-9\_]
```

numbers:

lexical syntax // numbers

```
INT      = "-"? [0-9]+
IntGroup = [0-9][0-9][0-9]
IntPref  = ([0-9] | ([0-9][0-9])) ", "
INT      = IntPref? {IntGroup ", ")+
FLOAT    = INT "." [0-9]+
NUM      = INT | FLOAT
```

lexical restrictions

```
INT    -/- [0-9]
FLOAT  -/- [0-9]
NUM    -/- [0-9]
```

strings:

lexical syntax

```
STRING      = "\"" StringChar* "\""
StringChar  = ~[\"\\n]
```

(continues on next page)

(continued from previous page)

```
StringChar      = "\\\"
StringChar      = BackSlashChar
BackSlashChar   = "\"
lexical restrictions
// Backslash chars in strings may not be followed by "
BackSlashChar -/- ["
```

layout:

```
lexical syntax // layout: whitespace and comments
LAYOUT         = [\ \t\n\r]
CommentChar     = [\*]
LAYOUT         = "/*" InsideComment* "*/"
InsideComment   = ~[\*]
InsideComment   = CommentChar
LAYOUT         = "//" ~[\n\r]* NewLineEOF
NewLineEOF      = [\n\r]
NewLineEOF      = EOF
EOF             =
lexical restrictions
CommentChar -/- [\/]
// EOF may not be followed by any char
EOF           -/- ~[]
context-free restrictions
// Ensure greedy matching for comments
LAYOUT? -/- [\ \t\n\r]
LAYOUT? -/- [\/] . [\/]
LAYOUT? -/- [\/] . [\*]
```

7.1.7 Grammar Interpretations

A grammar can be interpreted for (at least) the following operations:

Parsing Recognizing a well-formed sentence and constructing an abstract syntax tree

Signature Derive schema that defines well-formed abstract syntax trees

Formatting Map an abstract syntax tree to a well-formed sentence

Parse error recovery When editing programs, the program text is often in a syntactically incorrect state. Since all editor services depend on an AST representation of the program, getting stuck on syntax errors would reduce the utility of an editor. To get a better editing experience, a parser with error recovery does a best effort job to parse as much as possible and still produce an AST.

Syntactic completion Using a new language

7.2 Abstract Syntax

signature:

```
module signatures/Calc-sig

imports signatures/CalcLexical-sig
```

(continues on next page)

(continued from previous page)

```

signature
  sorts Program Exp Type
  constructors
    Program      : List(Stat) -> Program
    Exp          : Exp -> Stat
    Bind         : ID * Exp -> Stat
    Program-Plhdr : Program
    Stat-Plhdr   : Stat
    ID-Plhdr     : ID
    Exp-Plhdr    : Exp
    Num         : NUM -> Exp
    Min         : Exp -> Exp
    Pow         : Exp * Exp -> Exp
    Mul         : Exp * Exp -> Exp
    Div         : Exp * Exp -> Exp
    Sub         : Exp * Exp -> Exp
    Add         : Exp * Exp -> Exp
    Eq          : Exp * Exp -> Exp
    Neq         : Exp * Exp -> Exp
    Gt          : Exp * Exp -> Exp
    Lt          : Exp * Exp -> Exp
    NUM-Plhdr   : NUM
    Exp-Plhdr   : Exp
    True        : Exp
    False       : Exp
    Not         : Exp -> Exp
    And         : Exp * Exp -> Exp
    Or          : Exp * Exp -> Exp
    If          : Exp * Exp * Exp -> Exp
    Exp-Plhdr   : Exp
    Var         : ID -> Exp
    Let         : ID * Exp * Exp -> Exp
    Fun         : List(ID) * Exp -> Exp
    App         : Exp * Exp -> Exp
    ID-Plhdr    : ID
    Exp-Plhdr   : Exp
    NumT        : Type
    BoolT       : Type
    FunT        : Exp * Exp -> Type
    Exp-Plhdr   : Exp

```

7.3 Transformation

transformation:

```

module desugar

imports signatures/-

strategies

  desugar-calc = topdown(try(desugar))

rules

```

(continues on next page)

(continued from previous page)

```

desugar : Min(e) -> Sub(Num("0"), e)

desugar : Neq(e1, e2) -> Not(Eq(e1, e2))
desugar : Gt(e1, e2) -> Lt(e2, e1)

desugar : Not(e) -> If(e, False(), True())
desugar : And(e1, e2) -> If(e1, e2, False())
desugar : Or(e1, e2) -> If(e1, True(), e2)

desugar : Fun([x | xs@[_|_]], e) -> Fun([x], Fun(xs, e))

desugar : Num(i) -> Num(j)
  where <explode-string; filter(not(?44)); implode-string> i => j

```

7.4 Static Semantics

foobar:

```

module statics/calc

imports signatures/-

rules

  init ^ (s) := new s.

  [[ Program(stats) ^ (s) ]] :=
    new s', s' ---> s,
    Stats[[ stats ^ (s') ]].

  Stats[[ [] ^ (s) ]].

  Stats[[ [ stat | stats ] ^ (s) ]] :=
    Stat[[ stat ^ (s, s_nxt) ]],
    Stats[[ stats ^ (s_nxt) ]].

  Stat[[ Bind(x, e) ^ (s, s') ]] :=
    s' == s_nxt,
    new s_nxt, s_nxt ---> s,
    {x} <- s_nxt, {x} : ty_gen,
    ty_gen genOf ty,
    [[ e ^ (s) : ty ]].

  Stat[[ Exp(e) ^ (s, s_nxt) ]] :=
    s == s_nxt,
    [[ e ^ (s) : ty ]].

rules // numbers

  [[ Num(x) ^ (s) : NumT() ]].

  [[ Pow(e1, e2) ^ (s) : NumT() ]] :=
    [[ e1 ^ (s) : NumT() ]],

```

(continues on next page)

(continued from previous page)

```

    [[ e2 ^ (s) : NumT() ]].
[[ Mul(e1, e2) ^ (s) : NumT() ]] :=
    [[ e1 ^ (s) : NumT() ]],
    [[ e2 ^ (s) : NumT() ]].
[[ Add(e1, e2) ^ (s) : NumT() ]] :=
    [[ e1 ^ (s) : NumT() ]],
    [[ e2 ^ (s) : NumT() ]].
[[ Sub(e1, e2) ^ (s) : NumT() ]] :=
    [[ e1 ^ (s) : NumT() ]],
    [[ e2 ^ (s) : NumT() ]].
[[ Div(e1, e2) ^ (s) : NumT() ]] :=
    [[ e1 ^ (s) : NumT() ]],
    [[ e2 ^ (s) : NumT() ]].

[[ Eq(e1, e2) ^ (s) : BoolT() ]] :=
    [[ e1 ^ (s) : NumT() ]],
    [[ e2 ^ (s) : NumT() ]].
[[ Lt(e1, e2) ^ (s) : BoolT() ]] :=
    [[ e1 ^ (s) : NumT() ]],
    [[ e2 ^ (s) : NumT() ]].

rules // booleans

[[ True() ^ (s) : BoolT() ]].
[[ False() ^ (s) : BoolT() ]].

[[ If(e1, e2, e3) ^ (s) : ty2 ]] :=
    [[ e1 ^ (s) : BoolT() ]],
    [[ e2 ^ (s) : ty2 ]],
    [[ e3 ^ (s) : ty3 ]],
    ty2 == ty3 | error $[branches should have same type] @ e2.

rules // variables and functions

[[ Var(x) ^ (s) : ty ]] :=
    {x} -> s, {x} |-> d, d : ty_gen, ty instOf ty_gen.

[[ Let(x, e1, e2) ^ (s) : ty2 ]] :=
    new s', {x} <- s', {x} : ty, s' -P-> s,
    [[ e1 ^ (s) : ty ]],
    [[ e2 ^ (s') : ty2 ]].

[[ Fun([x], e) ^ (s) : FunT(ty1, ty2) ]] :=
    new s', {x} <- s', {x} : ty1, s' -P-> s,
    [[ e ^ (s') : ty2 ]].

[[ App(e1, e2) ^ (s) : ty2 ]] :=
    [[ e1 ^ (s) : FunT(ty1, ty2) ]],
    [[ e2 ^ (s) : ty1 ]].
    
```

7.5 Dynamic Semantics

run-time values:

```

module dynamics/eval
imports src-gen/ds-signatures/Calc-sig dynamics/bigdecimal
signature
  sorts Value
  sort aliases
    Env = Map(ID, Value)
  constructors
    NumV  : BigDecimal -> Value
    BoolV : Bool -> Value
    ClosV : ID * Exp * Env -> Value
  variables
    v : Value
  components
    E : Env

```

evaluation of program:

```

signature
  arrows
    Program -init-> Value
rules
  Program(e) -init-> v
  where E {} |- e --> v

```

evaluation of expressions:

```

signature
  arrows
    Exp --> Value
rules // numbers

  Num(n) --> NumV(parseB(n))

  Pow(NumV(i), NumV(j)) --> NumV(powB(i, j))
  Mul(NumV(i), NumV(j)) --> NumV(mulB(i, j))
  Div(NumV(i), NumV(j)) --> NumV(divB(i, j))
  Sub(NumV(i), NumV(j)) --> NumV(subB(i, j))
  Add(NumV(i), NumV(j)) --> NumV(addB(i, j))

  Lt(NumV(i), NumV(j)) --> BoolV(ltB(i, j))
  Eq(NumV(i), NumV(j)) --> BoolV(eqB(i, j))

signature
  arrows
    ift(Value, Exp, Exp) --> Value
rules // booleans

  False() --> BoolV(false)
  True() --> BoolV(true)

  If(v1, e2, e3) --> ift(v1, e2, e3)

  ift(BoolV(true), e2, _) --> e2
  ift(BoolV(false), _, e3) --> e3

rules // variables and functions

```

(continues on next page)

(continued from previous page)

```

E |- Var(x) --> E[x]

E |- Fun([x], e) --> ClosV(x, e, E)

E |- Let(x, v1, e2) --> v
where E {x |--> v1, E} |- e2 --> v

App(ClosV(x, e, E), v_arg) --> v
where E {x |--> v_arg, E} |- e --> v

signature
arrows
  List(Stat) --> Value
  Stat --> (Value * Env)
rules // top-level statements

[stat] : List(Stat) --> v
where stat --> (v, _)

[stat | stats@[_|_]] : List(Stat) --> v
where stat --> (_, E); E |- stats --> v

E |- Bind(x, v) --> (v, {x |--> v, E})

E |- Exp(v) --> (v, E)

```

7.5.1 Native Operators

BigDecimal library

declaration:

```

module dynamics/bigdecimal
signature
  native datatypes
    "java.math.BigDecimal" as BigDecimal { }
  native operators
    parseB : String -> BigDecimal
    addB : BigDecimal * BigDecimal -> BigDecimal
    powB : BigDecimal * BigDecimal -> BigDecimal
    subB : BigDecimal * BigDecimal -> BigDecimal
    mulB : BigDecimal * BigDecimal -> BigDecimal
    divB : BigDecimal * BigDecimal -> BigDecimal
    ltB : BigDecimal * BigDecimal -> Bool
    eqB : BigDecimal * BigDecimal -> Bool

```

7.5.2 Java Implementation

7.6 Editor Services

7.6.1 Main.esv

main:

```

module Main

imports Syntax Analysis

language

  extensions : calc

  provider : target/metaborg/stratego.ctree
  // provider : target/metaborg/stratego.jar
  provider : target/metaborg/stratego-javastrat.jar

```

7.6.2 Syntax.esv

syntax configuration:

```

module Syntax

imports

  libspoofax/color/default
  completion/colorer/Calc-cc-esv

language

  table          : target/metaborg/sdf-new.tbl
  start symbols : Program

  line comment  : "//"
  block comment : "/*" * "*/"
  fences        : [ ] ( ) { }

menus

  menu: "Syntax" (openeditor)

    action: "Format"          = editor-format (source)
    action: "Show parsed AST" = debug-show-aterm (source)

views

  outline view: editor-outline (source)
  expand to level: 3

```

7.6.3 Transformation.esv

transformation configuration:

```

module Transformation

menus

  menu: "Desugar" (openeditor)

```

(continues on next page)

(continued from previous page)

```

action: "Desugar" = desugar-pp (source)
action: "Desugar (AST)" = desugar-aterm (source)

```

7.6.4 Analysis.esv

analysis configuration:

```

module Analysis

imports

  nabl2/Menus
  nabl2/References

language

  observer : editor-analyze (constraint)

```

7.7 Testing

7.8 Exercises

In the previous sections you have seen all aspects of language definition with Spoofax. Before going out on your own and designing / implementing your own language using Spoofax, it is probably a good idea to first experiment a bit with working language implementation to get a better working understanding of the workbench. A good way to do that is to take the existing Calc project and extend / adapt it in various ways, perhaps in the direction of your own ideas for a language. This section provides some ideas for extending the Calc language. Fork the project from [github](#) and make a branch. Let us know by sending us a link to your fork or by submitting a pull request for a branch and we'll add your version to the list.

For many of these exercises you will need to dive deeper into the documentation of Spoofax.

7.8.1 Alternative Syntax

Experiment with designing an alternative notation for parts of the language. Here are some ideas:

- Introduce a keyword such as `def` to introduce a (top-level) variable declaration
- Currently, Calc function expressions use lambda notation `\ _._`. Try out alternatives such as Scala's `_ => _` notation or Javascripts `function (_){ _ }` notation. Can you do this without changing the abstract syntax?

7.8.2 Type Annotations

Calc relies on type inference; programmers do not declare the types of variables or return types of functions. That is fine for a calculator language; the types are often obvious from context. However, it is good programming practice to document the types of functions and variables. Extend Calc with optional type annotations for variable declarations. When present the inferred types should correspond to the type annotations, otherwise an error should be displayed.

example:

```
max : Num -> Num -> Num = \ x y . if (x > y) x else y
```

7.8.3 Type Aliases

When type expressions become more complicated it can be useful to assign them a name

example:

```
type foo = Num -> Num
```

7.8.4 Rich Arithmetic

The dynamic semantics of numbers is based on the BigDecimal Java library. Enrich the language to make better use of the library.

7.8.5 If-Then

Extend the language with an if-then statement *if(e1) e2*, i.e. without *else* branch. There are some challenges:

- The combination of the *if(e1) e2 else e3* expression with the *if(e1) e2* expression leads to the dangling else disambiguation problem. That is, how should the expression *if(e1) if(e2) e3 else e4* be parsed? Does the *else* belong to the inner or to the outer *if*? The usual convention is that the *else* belongs to the closest *if*. Can you express this in SDF3?
- Calc is a functional language. That is, the if-else form can be used as an expression that (always) yields a value. The *if* form only yields a value if the condition evaluates to *true*. Define a desugaring that transform the *if* form to the *if-else* form by producing a default value for the missing *else* branch. The challenge is that the default value depends on the type of *then* branch.

7.8.6 Nullary Functions

Calc does not support n-ary functions *x y z*, which are desugared to curried unary functions. However, nullary functions are not supported. Adapt the language definition to support nullary functions.

7.8.7 State

Variables in Calc are immutable. Add mutable variables to the language.

This requires threading a store through evaluation.

7.8.8 Recursion

Calc functions are currently not recursive since there is no way for a function to refer to itself. Extend the language with a *letrec* binding construct that allows recursive bindings.

7.8.9 Lazy Evaluation

create your own control constructs

instead of eagerly evaluating expressions, only evaluate an expression when it is required for a computation

7.8.10 Lists and Tuples

Extend the language with

7.8.11 Algebraic Data Types

Extend the language with algebraic data types

7.8.12 Units

7.8.13 Exceptions

Calculations may go wrong. For example, division by zero does not work. Extend the language with defined exceptions (possibly raised by native operators) and a *try-catch* form to handle exceptions.

7.8.14 Multiplicities

7.8.15 Translation to Java Bytecode

Abstract Syntax with ATerms

programs are represented as abstract syntax trees

in memory representation

different tools use different internal representations and data structures to represent trees

common interface

term format to provide a textual notation for ASTs

exchange between tools

persistent format

8.1 Annotated Terms

Terms in Stratego are terms in the *Annotated Term Format*, or *ATerms* for short. The ATerm format provides a set of constructs for representing trees, comparable to XML or algebraic data types in functional programming languages. For example, the expression $4 + f(5 * x)$ might be represented in a term as:

```
Plus(Int("4"), Call("f", [Mul(Int("5"), Var("x"))]))
```

ATerms are constructed from the following elements:

Integer An integer constant, that is a list of decimal digits, is an ATerm.

Examples: 1, 12343.

String A string constant, that is a list of characters between double quotes is an ATerm. Special characters such as double quotes and newlines should be escaped using a backslash. The backslash character itself should be escaped as well.

Examples: "foobar", "string with quotes\"", "escaped escape character\\ and a newline\n".

Constructor application A constructor is an identifier, that is an alphanumeric string starting with a letter, or a double quoted string.

A constructor application $c(t_1, \dots, t_n)$ creates a term by applying a constructor to a list of zero or more terms. For example, the term `Plus(Int("4"), Var("x"))` uses the constructors `Plus`, `Int`, and `Var` to create a nested term from the strings "4" and "x".

When a constructor application has no subterms (a “nullary constructor”) the parentheses can be omitted. However, this syntax is discouraged because this notation conflicts with variable names in some parts of the language, leading to confusing failures when you meant a variable but accidentally matched against a constructor.

List A list is a term of the form $[t_1, \dots, t_n]$, that is a list of zero or more terms between square brackets. While all applications of a specific constructor typically have the same number of subterms, lists can have a variable number of subterms. The elements of a list are typically of the same type, while the subterms of a constructor application can vary in type.

Example: The second argument of the call to "f" in the term `Call("f", [Int("5"), Var("x")])` is a list of expressions.

Tuple A tuple (t_1, \dots, t_n) is a constructor application without a constructor.

Example: `(Var("x"), Type("int"))`

Annotation The elements defined above are used to create the structural part of terms. Optionally, a term can be annotated with a list terms. These annotations typically carry additional semantic information about the term. An annotated term has the form $t\{t_1, \dots, t_n\}$.

Example: `Lt(Var("n"), Int("1")){Type("bool")}`. The contents of annotations is up to the application.

8.1.1 Exchanging Terms

The term format described above is used in Stratego programs to denote terms, but is also used to exchange terms between programs. Thus, the internal format and the external format exactly coincide. Of course, internally a Stratego program uses a data-structure in memory with pointers rather than manipulating a textual representation of terms. But this is completely hidden from the Stratego programmer. There are a few facts that are useful to be aware of, though.

When writing a term to a file in order to exchange it with another tool there are several representations to choose from. The textual format described above is the canonical *_meaning_* of terms, but does not preserve maximal sharing. Therefore, there is also a *_Binary ATerm Format (BAF)_* that preserves sharing in terms. The program *_baffle_* can be used to convert between the textual and binary representations.

8.1.2 Inspecting Terms

> TODO: Does *pp-aterm* still exist?

As a Stratego programmer you will be looking a lot at raw ATerms. Stratego pioneers would do this by opening an ATerm file in *_emacs_* and trying to get a sense of the structure by parenthesis highlighting and inserting newlines here and there. These days your life is much more pleasant through the tool *pp-aterm*, which adds layout to a term to make it readable. For example, parsing the following program:

```
let function fact(n : int) : int =
    if n < 1 then 1 else (n * fact(n - 1))
in printint(fact(10))
end
```

produces the following ATerm (say in file *fac.trm*):

```
Let ([FunDecs ([FunDec ("fact", [FArg("n", Tp(Tid("int"))]), Tp(Tid("int")),
If(Lt(Var("n"), Int("1")), Int("1"), Seq([Times(Var("n"), Call(Var("fact"),
[Minus(Var("n"), Int("1"))])))]), [Call(Var("printint"), [Call(Var(
"fact"), [Int("10")])])])])])
```

By pretty-printing the term using *pp-aterm* as

> TODO: Is this syntax still correct?

```
$ pp-aterm -i fac.trm -o fac-pp.trm --max-term-size 20
```

we get a much more readable term:

```
Let (
  [ FunDecs (
    [ FunDec (
      "fact"
      , [FArg("n", Tp(Tid("int"))])
      , Tp(Tid("int"))
      , If (
        Lt(Var("n"), Int("1"))
        , Int("1")
        , Seq([ Times(Var("n"), Call(Var("fact"), [Minus(Var("n"), Int("1"))]))
          ])
        )
      )
    ]
  )
, [ Call(Var("printint"), [Call(Var("fact"), [Int("10")])])
]
)
```

8.2 Signatures

The ATerm format does not restrict terms describing terms schemas with signatures

8.2.1 Stratego Signatures

Sort a type of terms

Constructor declaration name and types of arguments:

```
C : S1 * ... * Sn -> S0
```

To use terms in Stratego programs, their constructors should be declared in a signature. A signature declares a number of sorts and a number of constructors for these sorts. For each constructor, a signature declares the number and types of its arguments. For example, the following signature declares some typical constructors for constructing abstract syntax trees of expressions in a programming language:

```
module Expressions-sig
imports Common
signature
```

(continues on next page)

(continued from previous page)

```

sorts Exp
constructors
  Var    : ID -> Exp
  Int    : INT -> Exp
  Plus   : Exp * Exp -> Exp
  Mul    : Exp * Exp -> Exp
  Call   : ID * List(Exp) -> Exp

```

Currently, the Stratego compiler only checks the arity of constructor applications against the signature. Still, it is considered good style to also declare the types of constructors in a sensible manner for the purpose of documentation. Also, a later version of the language may introduce type checking.

8.2.2 SDF3 Syntax Definitions

signature derived from syntax definition

for example:

```

module Expressions
imports Common
sorts Exp
context-free syntax
  Exp.Var = <<ID>>
  Exp.Int = <<INT>>
  Exp.Plus = <<Exp> + <Exp>> {left}
  Exp.Mul = <<Exp> * <Exp>> {left}
  Exp.Call = <<ID> (<{Exp " " }*>)>
context-free priorities
  Exp.Mul > Exp.Plus

```

8.3 Term Rewriting

8.4 Term API

programming with terms in Java

Syntax Definition with SDF3

The definition of a textual (programming) language starts with its syntax. A grammar describes the well-formed sentences of a language. When written in the grammar language of a parser generator, such a grammar does not just provide such a description as documentation, but serves to generate an implementation of a parser that recognizes sentences in the language and constructs a parse tree or abstract syntax tree for each valid text in the language.

Syntax Definition Formalism 3 (SDF3) goes much further than the typical grammar languages. It covers all syntactic concerns of language definitions, including the following features: support for the full class of context-free grammars by means of generalized LR parsing; integration of lexical and context-free syntax through scannerless parsing; safe and complete disambiguation using priority and associativity declarations; an automatic mapping from parse trees to abstract syntax trees through integrated constructor declarations; automatic generation of formatters based on template productions; and syntactic completion proposals in editors.

9.1 SDF3 Overview

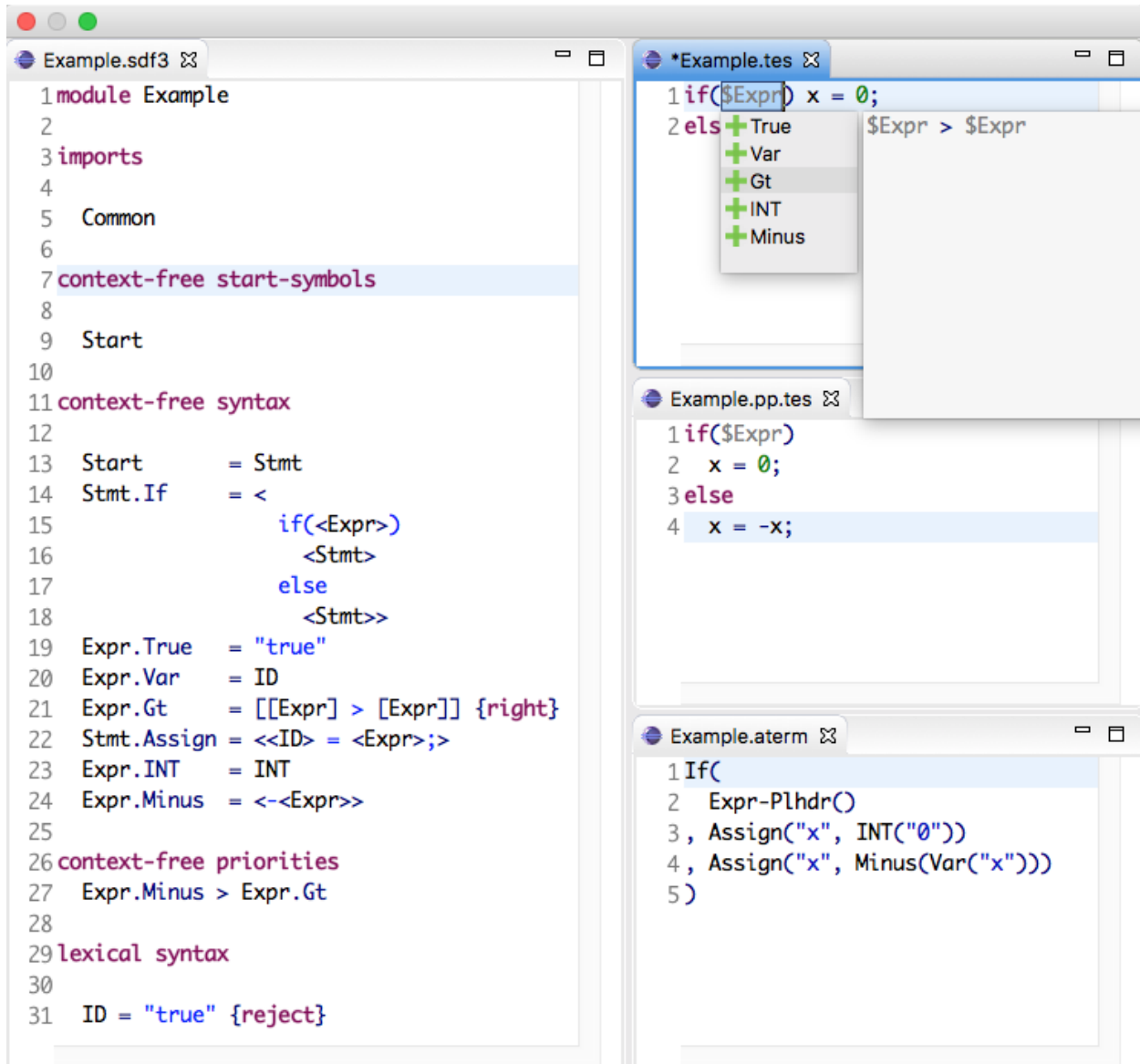
The SDF family of Syntax Definition Formalisms provides support for declarative definition of the syntax of programming languages and domain-specific languages. The key principle underlying the design of SDF is declarative syntax definition, so that the user does not need to understand underlying parsing algorithm.

SDF has evolved into SDF3 to serve the needs of modern language workbenches and at the same time improve various issues of its predecessor, SDF2. With SDF3, it is possible to modularly describe a language's syntax, generating a parser, a pretty printer, and basic editor features such as syntactic code completion and syntax highlighting. SDF3 also supports safe operator precedence and associativity disambiguation by means of priority declarations, and lexical ambiguities by means of reject rules and follow restrictions.

The screenshot below from Spoofax illustrates an excerpt of a grammar written in SDF3, a program of that language being edited, its pretty-printed version and its abstract syntax tree.

To list the main improvements of SDF3 with respect to SDF2:

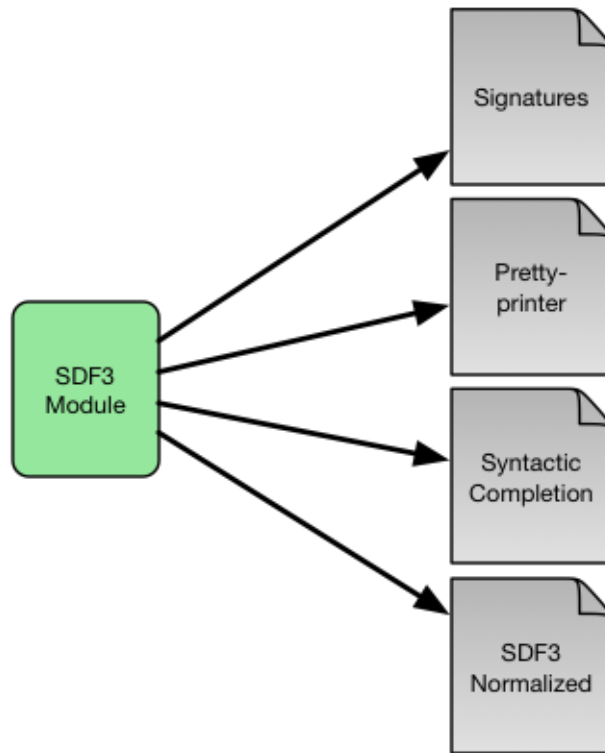
- SDF3 incorporates constructors into the syntax definition, providing a direct correspondence between abstract syntax trees and grammar rules.
- Grammar rules follow the productive form, improving readability and consistency with main-stream grammar formalisms.



- Grammar rules can be used next to template productions, which considers the whitespace surrounding symbols when deriving a pretty-printer.
- Finally, grammar rules can be identified by its sort and constructor, and do not need to be duplicated in the priorities section.

SDF3 is in constant evolution, and this documentation provides an up-to-date overview of its current features.

Even though the primary goal of SDF3 is syntax definition, it is used as input to generate many artifacts that are used in Spoofax. The figure below illustrates the artifacts produced when compiling an SDF3 module.



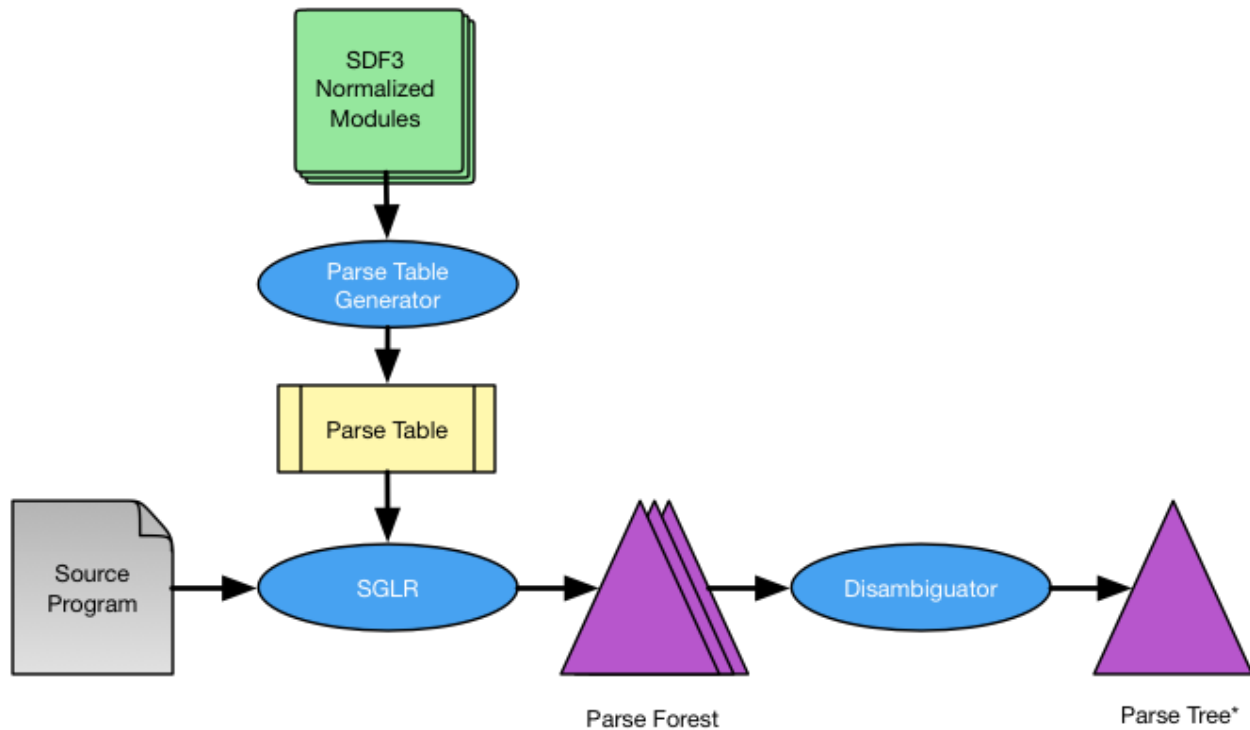
The most important of the artifacts generated from SDF3 is a parse table, which is used by the Scannerless Generalized LR parser to parse programs, producing an abstract syntax tree (AST). Note that SGLR supports ambiguous grammars, outputting a parse forest as result. SDF3 disambiguation mechanisms operate at parse table generation time, at parse time, and after parsing. Ideally, a single tree is produced at the end. The whole process of parsing a source program is described in the figure below.

In the remainder of this documentation we present the elements of an SDF3 definition.

9.2 SDF3 Reference Manual

9.2.1 Modules

An SDF3 specification consists of a number of module declarations. Each module may import other modules and define sections that include sort, start symbols, syntax, restrictions, priorities, and template options.



Imports

Modules may import other modules for reuse or separation of concerns. A module may extend the definition of a non-terminal in another module. A module may compose the definition of a language by importing the parts of the language. The structure of a module is as follows:

```

module <ModuleName>

<ImportSection>*

<Section>*
  
```

The `module` keyword is followed by the module name, then a series of imports can be made, followed by sections that contain the actual definition of the syntax. An import section is structured as follows:

```

imports <ModuleName>*
  
```

Note that SDF3 does not support parameterized modules.

9.2.2 Symbols

The building block of SDF3 productions is a symbol. SDF3 symbols can be compared to terminals and non-terminals in other grammar formalisms. The elementary symbols are literals, sorts and character classes.

Intrinsically, only character classes are real terminal symbols. All other symbols represent non-terminals. SDF3 also support symbols that capture BNF-like notation such as lists and optionals. Note that these symbols are also non-terminals, and are just shorthands for common structures present in context-free grammars.

Character classes

Character classes occur only in lexical syntax and are enclosed by `[` and `]`. A character class consists of a list of zero or more characters (which stand for themselves) such as `[x]` to represent the character `x`, or character ranges, as an abbreviation for all the characters in the range such as `[0-9]` representing 0, 1, ..., 9. A valid range consists of `[c1-c2]`, where the character `c2` has a higher ASCII code than `c1`. Note that nested character classes can also be concatenated within the same character class symbol, for example `[c1c2-c3c4-c5]` includes the characters `c1` and the ranges `c2-c3`, `c4-c5`. In this case, the nested character classes do not need to be ordered, as SDF3 orders them when performing a normalization step.

Escaped Characters: SDF3 uses a backslash (`\`) as a escape for the quoting of special characters. One should use `\c` whenever `c` is not a digit or a letter in a character class.

Arbitrary Unicode code points can be included in a character class by writing an escaped integer, which is particularly useful for representing characters outside the printable ASCII range. The integer can be a binary, octal, decimal, or hexadecimal number, for example: `\0b101010`, `\052`, `\42`, and `\0x2A` all represent the code point 42, or the `'*'` character.

Additionally, special ASCII characters are represented by:

- `\t` : horizontal tabulation
- `\n` : newline character
- `\v` : vertical tabulation
- `\f` : form feed
- `\r` : carriage return

Character Class Operators: SDF3 provides the following operators for character classes:

- (complement) `~` : Accepts all the characters that are *not* in the original class.
- (difference) `/` : Accepts all the characters in the first class unless they are in a second class.
- (union) `\|` : Accepts all the characters in either character classes.
- (intersection) `/\` : Accepts all the characters that are accepted by both character classes.

Note that the first operator is unary and the other ones are left associative binary operators. Furthermore, such operators are not applicable to other symbols in general.

Literals

A literal symbol defines a fixed length word. This usually corresponds to a terminal symbol in ordinary context-free grammars, for example `"true"` or `"+"`. Literals must always be quoted and consist of (possibly escaped) ASCII characters.

As literals are also regular non-terminals, SDF3 automatically generates productions for them in terms of terminal symbols.

```
"definition" = [d][e][f][i][n][i][t][i][o][n]
```

Note that the production above defines a case-sensitive implementation of the defined literal. Case-insensitive literals are defined using single-quoted strings as in `'true'` or `'else'`. SDF3 generates a different production for case-insensitive literals as

```
'definition' = [dD][eE][fF][iI][nN][iI][tT][iI][oO][nN]
```

The literal above accepts case-insensitive inputs such as `definition`, `DEFINITION`, `DeFiNiTiOn` or `definition`.

Sorts

A sort correspond to a plain non-terminal, for example, `Statement` or `Exp`. Sort names start with a capital letter and may be follow by letters, digits, hyphens, or underscores. Note that unlike SDF2, SDF3 does not support parameterized sorts (yet!).

Sorts must be declared before they can be used. Context-free sorts are declared in a `context-free sorts` block:

```
context-free sorts
  Statement
  Exp
```

Lexical sorts must be declared in a `lexical sorts` block:

```
lexical sorts
  ID
  INT
  STRING
```

Optionals

SDF3 provides a shorthand for describing zero or exactly one occurrence of a sort by appending the sort with `?`. For example, the sort `Extends?` can be parsed as `Extends` or without consuming any input. Internally, SDF3 generates the following productions after normalizing the grammar:

```
Extends?.None =
Extends?.Some = Extends
```

Note that using `?` adds the constructors `None` and `Some` to the final abstract syntax tree.

Lists

Lists symbols as the name says, indicate that a symbol should occur several times. In this way, it is also possible to construct flat structures to represent them. SDF3 provides support for two types of lists, with and without separators. Furthermore, it is also possible to indicate whether a list can be empty (`*`) or should have at least one element (`+`). For example, a list `Statement*` indicates zero or more `Statement`, whereas a list with separator `{ ID " , " }+` indicates one or more `ID` separated by `,`. Note that SDF3 only supports literal symbols as separators.

Again, SDF3 generates the following productions to represent lists, when normalizing the grammar

```
Statement* =
Statement* = Statement+
Statement+ = Statement+ Statement
Statement+ = Statement

{ ID " , " }* =
{ ID " , " }* = { ID " , " }+
{ ID " , " }+ = { ID " , " }+ " , " { ID " , " }
{ ID " , " }+ = { ID " , " }
```

When parsing a context-free list, SDF3 produces a flattened list as an AST node such as `[Statement, ..., Statement]` or `[ID, ..., ID]`. Note that because the separator is a literal, it does not appear in the AST.

Alternative

Alternative symbols express the choice between two symbols, for example, `ID | INT`. That is, the symbol `ID | INT` can be parsed as either `ID` or `INT`. For that reason, SDF3 normalizes alternatives by generating the following productions:

```
ID | INT = ID
ID | INT = INT
```

Note that SDF3 only allow alternative symbols to occur in lexical syntax. Furthermore, note that the alternative operator is right associative and binds stronger than any operator. That is, `ID ", " | ID ";"` expresses `ID (", " | ID) ";"`. To express `(ID ", ") | (ID ";")`, we can use a sequence symbol.

Sequence

A sequence operator allows grouping of two or more symbols. Sequences are useful when combined with other symbols such, lists or optionals, for example `("e" [0-9]+) ?`. Like alternative symbols, sequences can only occur in lexical syntax. A sequence symbol is normalized as:

```
("e" [0-9]+) = "e" [0-9]+
```

Labeled symbols

SDF3 supports decorating symbols with labels, such as `myList : {elem : Stmt " ; " } *`. The labels have no semantics but can be used by other tools that use SDF3 grammars as input.

LAYOUT

The `LAYOUT` symbol is a reserved sort name. It is used to indicate the whitespace that can appear in between context-free symbols. The user must define the symbol `LAYOUT` such as:

```
LAYOUT = [\ \t\n]
```

Note that the production above should be defined in the lexical syntax.

9.2.3 Syntax

As seen before, a SDF3 module may constitute of zero or more sections. All sections contribute to the final grammar that defines a language. Sections can define production rules, priorities, restrictions, or simply specify some characteristics of the syntax definition.

Sort declarations

Sorts are declared by listing their name in the appropriate sorts section, which have the following forms. For context-free sorts:

```
context-free sorts
```

```
<Sort>*
```

For lexical sorts:

```
lexical sorts
```

```
<Sort>*
```

SDF3 also supports kernel sorts:

```
sorts
```

```
<Sort>*
```

Note: Kernel sorts should be suffixed with `-CF` or `-LEX`, depending on whether they are context-free sorts or lexical sorts. When a sort in a `sorts` block does not have a suffix, it is treated as a context-free sort.

Writing a sort in these sections only indicates that a sort has been declared, even if it does not have any explicit production visible.

Start symbols

The lexical or context-free start symbols sections explicitly define the symbols which will serve as start symbols when parsing terms. If no start symbols are defined it is not possible to recognize terms. This has the effect that input sentences corresponding to these symbols can be parsed. So, if we want to recognize boolean terms we have to define explicitly the sort `Boolean` as a start symbol in the module `Booleans`. Any symbol and also lists, optionals, etc., can serve as a start-symbol. A definition of lexical start symbols looks like

```
lexical start-symbols
```

```
<Symbol>*
```

while context-free start symbols are defined as:

```
context-free start-symbols
```

```
<Symbol>*
```

SDF3 also supports kernel start-symbols:

```
start-symbols
```

```
<Symbol>*
```

In contrast to lexical and kernel start-symbols, context-free start symbols can be surrounded by optional layout. A lexical start-symbol should have been defined by a production in the lexical syntax; a context-free symbol should have been defined in the context-free syntax. Both symbols can also be defined in kernel syntax using the suffix `-LEX` or `-CF`.

Lexical syntax

The lexical syntax usually describes the low level structure of programs (often referred to as lexical tokens.) However, in SDF3 the token concept is not really relevant, since only character classes are terminals. The lexical syntax sections in SDF3 are simply a convenient notation for the low level syntax of a language. The `LAYOUT` symbol should also be defined in a lexical syntax section. A lexical syntax consists of a list of productions.

Lexical syntax is described as follows:

lexical syntax

```
<Production>*
```

An example of a production in lexical syntax:

lexical syntax

```
BinaryConst = [0-1]+
```

Context-free syntax

The context-free syntax describes the more high-level syntactic structure of sentences in a language. A context-free syntax contains a list of productions. Elements of the right-hand side of a context-free production are pre-processed in a normalization step before parser generation that adds the `LAYOUT?` symbol between any two symbols. Context-free syntax has the form:

context-free syntax

```
<Production>*
```

An example production rule:

context-free syntax

```
Block.Block = "{" Statement* "}"
```

SDF3 automatically allows for layout to be present between the symbols of a rule. This means that a fragment such as:

```
{
}
```

will still be recognized as a block (assuming that the newline and line-feed characters are defined as layout).

Kernel syntax

The rules from context-free and lexical syntax are translated into kernel syntax by the SDF3 normalizer. When writing kernel syntax, one has more control over the layout between symbols of a production.

As part of normalization, among other things, SDF3 renames each symbol in the lexical syntax to include the suffix `-LEX` and each symbol in the context-free syntax to include the suffix `-CF`. For example, the two productions above written in kernel syntax look like

syntax

```
Block-CF.Block = "{" LAYOUT?-CF Statement*-CF LAYOUT?-CF "}"
BinaryConst-LEX = [0-1]+
```

Literals and character-classes are lexical by definition, thus they do not need any suffix. Note that each symbol in kernel syntax is uniquely identified by its full name including `-CF` and `-LEX`. That is, two symbols named `Block-CF` and `Block` are different, if both occur in kernel syntax. However, `Block-CF` is the same symbol as `Block` if the latter appears in a context-free syntax section.

As mentioned before, layout can only occur in between symbols if explicitly specified. For example, the production

```
syntax

Block-CF.Block = "{" Statement*-CF LAYOUT?-CF "}"
```

does not allow layout to occur in between the opening bracket and the list of statements. This means that a fragment such as:

```
{
  x = 1;
}
```

would not be recognized as a block.

Productions

The basic building block of syntax sections is the production. The left-hand side of a regular production rule can be either just a symbol or a symbol followed by `.` and a constructor name. The right-hand side consists of zero or more symbols. Both sides are separated by `=`:

```
<Symbol> = <Symbol>*
<Symbol>.<Constructor> = <Symbol>*
```

A production is read as the definition. The symbol on the left-hand side is defined by the right-hand side of the production.

Productions are used to describe lexical as well as context-free syntax. Productions may also occur in priority sections, but might also be referred to by its `<Symbol>.<Constructor>`. All productions with the same symbol together define the alternatives for that symbol.

Attributes

The definition of lexical and context-free productions may be followed by attributes that define additional (syntactic or semantic) properties of that production. The attributes are written between curly brackets after the right-hand side of a production. If a production has more than one attribute they are separated by commas. Attributes have thus the following form:

```
<Sort> = <Symbol>* { <Attribute1>, <Attribute2>, ... }
<Sort>.<Constructor> = <Symbol>* { <Attribute1>, <Attribute2>, ... }
```

The following syntax-related attributes exist:

- `bracket` is an important attribute in combination with priorities. For example, the *sdf2parenthesize* tool uses the `bracket` attribute to find productions to add to a parse tree before pretty printing (when the tree violates priority constraints). Note that most of these tools demand the production with a `bracket` attribute to have the shape: `X = " (" X ")" {bracket}` with any kind of bracket syntax but the `X` being the same symbol on the left-hand side and the right-hand side. The connection with priorities and associativity is that when a non-terminal is disambiguated using either of them, a production rule with the `bracket` attribute is probably also needed.
- `left`, `right`, `non-assoc`, `assoc` are disambiguation constructs used to define the associativity of productions. See *associativity*.
- `prefer` and `avoid` are **deprecated** disambiguation constructs to define preference of one derivation over others. See *preferences*.

- `reject` is a disambiguation construct that implements language difference. It is used for keyword reservation. See [rejections](#).

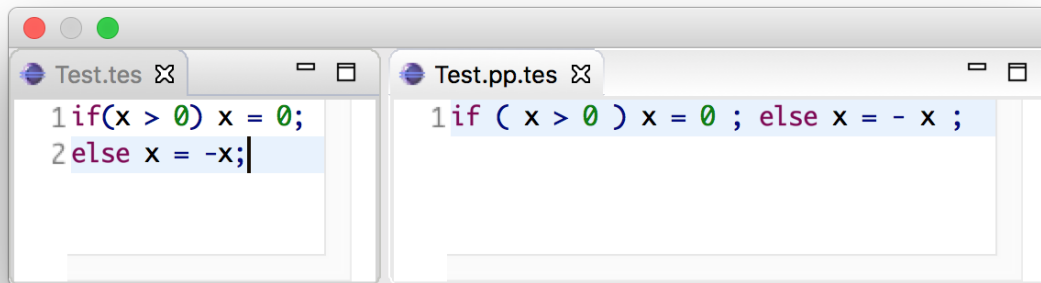
9.2.4 Templates

Templates are a major change in SDF3 when comparing to SDF2. They are essential when aiming to generate a nice pretty printer or generate proper syntactic code completion templates. When generating such artifacts, a general production simply introduces a whitespace in between symbols.

For example, when writing a grammar rule

```
Statement.If = "if" "(" Exp ")" Exp "else" Exp
```

and pretty printing a valid program, we would get the text in a single line separated by spaces, as:



Furthermore, code completion would consider the same indentation when inserting code snippets.

However, when using template productions such as

```
Statement.If = <
  if (<Exp>)
    <Exp>
  else
    <Exp>>
```

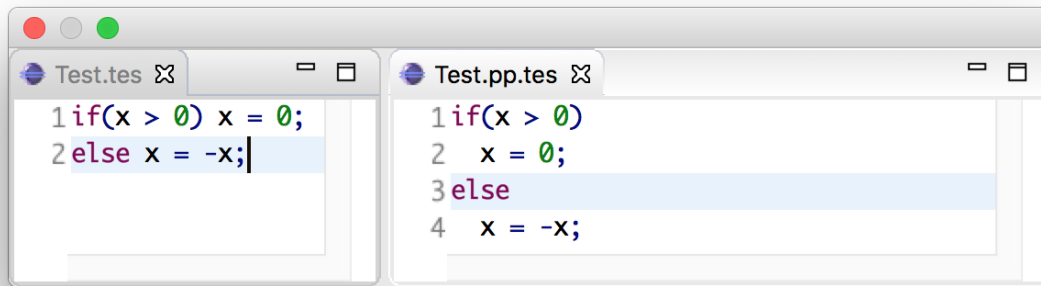
We would get the following program.

Again, code completion would also consider this indentation for proposals.

That is, in template productions, the surrounding layout is used to nicely pretty print programs and its code completion suggestions.

Template Productions

Template productions are an alternative way of defining productions. Similarly, they consist of a left-hand side and a right-hand side separated by `=`. The left-hand side is the same as for productive rules. The right-hand side is a template delimited by `<` and `>`. The template can contain zero or more symbols:



```
<Sort>                = < <Symbol>* >
<Sort>.<Constructor> = < <Symbol>* >
```

Alternatively, square brackets can be used to delimit a template:

```
<Sort>                = [ <Symbol>* ]
<Sort>.<Constructor> = [ <Symbol>* ]
```

The symbols in a template can either be placeholders or literal strings. It is worth noting that:

- placeholders need to be enclosed within the same delimiters (either `< . . >` or `[. .]`) as the template ;
- literal strings need not be enclosed within quotation marks;
- literal strings are tokenized on space characters (whitespace, tab);
- additionally, literal strings are tokenized on boundaries between characters from the set given by the `tokenize` option, see the `tokenize` template option;
- placeholders translate literally. If a separator containing any layout characters is given, the placeholder maps to a list with separator that strips the layout.

An example of a template rule:

```
Exp.Addition = < <Exp> + <Exp> >
```

Here, the `+` symbol is a literal string and `<Exp>` is a placeholder for sort `Exp`.

Placeholders are of the form:

- `<Sort?>`: optional placeholder
- `<Sort*>`: repetition (0...n)
- `<Sort+>`: repetition (1...n)
- `<{Sort " , " }*>`: repetition with separator

Case-insensitive Literals

As we showed before, SDF3 allows defining case-insensitive literals as single-quoted strings in regular productions. For example:

```
Exp.If = 'if' "(" Exp ")" Exp 'else' Exp
```

accepts case-insensitive keywords for `if` and `else` such as `if`, `IF`, `If`, `else`, `ELSE` or `ELsE`. However, to generate case-insensitive literals from template productions, it is necessary to add annotate these productions as case-insensitive. For example, a template production:

```
Exp.If = <
  if(<Exp>)
    <Exp>
  else
    <Exp>
> {case-insensitive}
```

accepts the same input as the regular production mentioned before.

Moreover, lexical symbols can also be annotated as case-insensitive to parse as such. The constructed abstract syntax tree contains lower-case symbols, but the original term is preserved via origin-tracking. For example:

```
ID = [a-zA-z][a-zA-Z0-9]* {case-insensitive}
```

can parse `f00`, `F00`, `F0o`, `f0o`, `f0O`, `f0O` or `FOO`. Whichever option generates a node `"f00"` in the abstract syntax tree. By consulting the origin information on this node, it is possible to know which term was used as input to the parser.

Template options

Template options are options that are applied to the current file. A template options section is structured as follows:

```
template options
  <TemplateOption*>
```

Multiple template option sections are not supported. If multiple template option sections are specified, the last one is used.

There are three kinds of template options.

keyword Convenient way for setting up lexical follow restrictions for keywords. See the section on follow restrictions for more information. The structure of the keyword option is as follows:

```
keyword -/- <Pattern>
```

This will add a follow restriction on the pattern for each keyword in the language. Keywords are automatically detected, any terminal that ends with an alphanumeric character is considered a keyword.

Multiple keyword options are not supported. If multiple keyword options are specified, the last one is used.

Note that this only sets up follow restrictions, rejection of keywords as identifiers still needs to be written manually.

tokenize Specifies which characters may have layout around them. The structure of a tokenize option is as follows:

```
tokenize : "<Character*>"
```

Consider the following grammar specification:

```
template options

  tokenize : "("

context-free syntax

Exp.Call = <<ID> () ;>
```

Because layout is allowed around the (and) characters, there may be layout between (and ; in the template rule. If no tokenize option is specified, it defaults to the default value of ().

Multiple tokenize options are not supported. If multiple tokenize options are specified, the last one is used.

reject Convenient way for setting up reject rules for keywords. See the section on [rejections](#) for more information. The structure of the reject option is as follows:

```
Symbol = keyword {attrs}
```

where Symbol is the symbol to generate the rules for. Note that attrs can be include any attribute, but by using reject, reject rules such as ID = "true" {reject} are generated for all keywords that appear in the templates.

Multiple reject template options are not supported. If multiple reject template options are specified, the last one is used.

9.2.5 Disambiguation

As we showed before, the semantics of SDF3 can be seen as two-staged. First, the grammar generates all possible derivations. Second, the disambiguation constructs remove a number of derivations that are not valid. Note that SDF3 actually performs some disambiguation when generating the parse table or during parsing.

Rejections

Rejections filter derivations. The semantics of a rejection is that the set of valid derivations for the left-hand side of the production will not contain the construction described on the right-hand side. In other words, the language defined by the sort on the left-hand side has become smaller, removing all the constructions generated by the rule on the right-hand side. Disambiguation by reject occurs at parse time (mostly).

A rule can be marked as rejected by using the attribute {reject} after the rule:

```
<Sort> = ... {reject}
```

The {reject} attribute works well for lexical rejections, especially keyword reservation in the form of productions like:

```
ID = "keyword" {reject}
```

Preferences

The preferences mechanism is another disambiguation filter that provides a post parse filter to parse forests. The attributes prefer and avoid are the only disambiguation constructs that compare alternative derivations after parsing.

Warning: `prefer` and `avoid` are deprecated and will be removed in a future version of Spoofax.

The following definition assumes that derivations are represented using parse forests with “packaged ambiguity nodes”. This means that whenever in a derivation there is a choice for several sub-derivations, at that point a special choice node (ambiguity constructor) is placed with all alternatives as children. We assume here that the ambiguity constructor is always placed at the location where a choice is needed, and not higher (i.e. a minimal parse forest representation). The preference mechanism compares the top nodes of each alternative:

- All alternative derivations that have `avoid` at the top node will be removed, but only if other alternatives derivations are there that do not have `avoid` at the top node.
- If there are derivations that have `prefer` at the top node, all other derivations that do not have `prefer` at the top node will be removed.

The preference attribute can be used to handle the case when two productions can parse the same input. Here is an example:

```
Exp.FunctionApp = <<Expr> <Expr*>>
Exp.Constructor = <<ID> <Expr*>> {prefer}
```

Priorities

Priorities are one of SDF3’s most often used disambiguation constructs. A priority section defines the relative priorities between productions. Priorities are a powerful disambiguation construct because it occurs at parse generation time. The idea behind the semantics of priorities is that productions with a higher priority “bind stronger” than productions with a lower priority. The essence of the priority disambiguation construct is that certain parse trees are removed from the ‘forest’ (the set of all possible parse trees that can be derived from a segment of code). The basic priority syntax looks like this:

context-free priorities

```
<ProductionRef> > <ProductionRef>
```

Where `<ProductionRef>` can either be `<Sort>`, `<Cons>` or the entire production itself.

Several priorities in a priority grammar are separated by commas. If more productions have the same priority they may be grouped between curly braces on each side of the `>` sign.

context-free priorities

```
{<ProductionRef> <ProductionRef>}
  > <ProductionRef>,
<ProductionRef>
  > <ProductionRef>
```

By default, the priority relation is automatically transitively closed (i.e. if $A > B$ and $B > C$ then $A > C$). To specify a non-transitive priority relation it is necessary to include a dot before the `>` sign (`. >`).

SDF3 provides *safe* disambiguation, meaning that priority relations only remove ambiguous derivations. Furthermore, SDF3 also allows tree filtering by means of indexed priorities such as:

context-free priorities

```
<ProductionRef> <idx> > <ProductionRef>
```

where the symbol at position `idx` (starting with 0) in the first production should not derive the second production.

An example defining priorities for the addition, subtraction and multiplication operators is listed below. Because addition and subtraction have the same priority, they are grouped together between brackets.

context-free priorities

```
{Exp.Times} >
{Exp.Plus Exp.Minus}
```

Associativity

Like with priorities, the essence of the associativity attribute is that certain parse trees are removed from the ‘forest’.

- The `left` associativity attribute on a production `P` filters all occurrences of `P` as a direct child of `P` in the right-most argument. This implies that `left` is only effective on productions that are recursive on the right (as in `A B C -> C`).
- The `right` associativity attribute on a production `P` filters all occurrences of `P` as a direct child of `P` in the left-most argument. This implies that `right` is only effective on productions that are recursive on the left (as in `C A B -> C`).
- The `non-assoc` associativity attribute on a production `P` filters all occurrences of `P` as a direct child of `P` in any argument. This implements that `non-assoc` is only effective if a production is indeed recursive (as in `A C B -> C`).
- The `assoc` attribute means the same as `left`

Associativity declarations occur in two places in SDF3. The first is as production attributes. The second is as associativity declarations in priority groups.

An example on how to mention associativity as a production attribute is given below:

```
Exp.Plus = <<Exp> + <Exp>> {left}
```

In priority groups, the associativity has the same semantics as the associativity attributes, except that the filter refers to more nested productions instead of a recursive nesting of one production. The group associativity attribute works pairwise and commutative on all combinations of productions in the group. If there is only one element in the group the attribute is reflexive, otherwise it is not reflexive.

context-free priorities

```
{left: Exp.Times} >
{left: Exp.Plus Exp.Minus}
```

Restrictions

The notion of restrictions enables the formulation of lexical disambiguation strategies. Examples are “shift before reduce” and “longest match”. A restriction filters applications of productions for certain non-terminals if the following character (lookahead) is in a certain class. The result is that specific symbols may not be followed by a character from a given character class. A lookahead may consist of more than one character class (multiple lookahead). Restrictions come in two flavors:

- lexical restrictions that apply to lexical non-terminals
- context-free restrictions that apply to context-free non-terminals.

The general form of a restriction is:

```
<Symbol>+ -/- <Lookaheads>
```

The semantics of a restriction is to remove all derivations that produce a certain `<Symbol>`. The condition for this removal is that the derivation tree for that symbol is followed immediately by something that matches the lookahead declaration. Note that to be able to check this condition, one must look past derivations that produce the empty language, until the characters to the right of the filtered symbol are found. Also, for finding multiple lookahead matches, one must ignore nullable sub-trees that may occur in the middle of the matched lookahead.

In case of lexical restrictions `<Symbol>` may be either a literal or sort. In case of context-free restrictions only a sort or symbol is allowed. The restriction operator `-/-` should be read as may not be followed by. Before the restriction operator `-/-` a list of symbols is given for which the restriction holds.

As an example, the following restriction rule implements the “longest match” policy: an identifier can not be followed by an alpha-numeric character.

```
ID -/- [a-zA-Z0-9\_]
```

Layout-sensitive parsing

SDF3 supports definition of layout sensitive syntax by means of layout constraints. While we haven’t covered this feature in this documentation, the paper [\[S2\]](#) describes the concepts.

Layout Declarations

In the paper [\[S2\]](#), the authors describe layout constraints in terms of restrictions involving the position of the subtree involved in the constraint (0, 1, ...), token selectors (`first`, `left`, `last` and `right`), and position selectors as lines and columns (`line` and `col`). This mechanism allows writing layout constraints to express alignment, offside and indentation rules, but writing such constraints is rather cumbersome and error prone. Alternatively, one may write layout constraints using **layout declarations**, which are more declarative specifications and abstract over lines, columns and token selectors as the original layout constraints from [\[S2\]](#). **Note:** if you want to use layout declarations, you should specify the `jsglr-version: layout-sensitive` parameter for SDF3, see [JSGLR configuration](#).

- **tree selectors**

To specify which trees should be subject to a layout constraint, one may use: tree positions, SDF3 labeled non-terminals, or unique literals that occurs in the production. For example:

context-free syntax

```
Stmt.IfElse = "if" Exp "then" Stmt "else" else:Stmts {layout(
  indent "if" 3, else &&
  align 3 else &&
  align "if" "else"
)}
```

In the layout constraint for the production above, `else` refers to the tree for the labeled non-terminal `else:Stmts`, `"if"` refers to the tree corresponding to the `"if"` literal and the number 3 correspond to the tree at *position 3* in the parse tree (starting at 0, ignoring trees for LAYOUT?).

- **align**

The layout constraint `layout(align x y1, ..., yn)` specifies that the trees indicated by the tree selectors `yi` should be aligned with the tree indicated by the tree selector `x`, i.e., all these trees should start in the same column. For example, if we consider the production above, the following program is correct according to the **align** constraints:

```
if x < 0 then
  ..x = 0
```

(continues on next page)

(continued from previous page)

```
else
  ..y = 1
```

Whereas, the following program is incorrect because neither the if and else keyword align (`align "if" "else"`), nor the statements in the branches (`align 3 else`):

```
if x < 0 then
  ..x = 0
  ..else
    ...y = 1
```

- **align-list**

The constraint **align-list** can be used to indicate that all subtrees within a list should be aligned. That is, a constraint `layout (align-list x)`, where `x` is a tree selector for a list subtree, can be used to enforce such constraint. For example, consider the following production and its layout constraint:

```
context-free syntax

  Stmt.If = "if" Exp "then" then:Stmt* {layout (
    align-list then
  ) }
```

This constraint indicates that statements inside the list should be aligned. Therefore, the following program is correct according to this constraint:

```
if x < 0 then
  ..x = 0
  ..y = 4
  ..z = 2
```

And the following program is invalid, as the second statement is misaligned:

```
if x < 0 then
  ..x = 0
  ...y = 4
  ..z = 2
```

- **offside**

The offside rule is very common in layout-sensitive languages. It states that all lines after the first one should be further to the right compared to the first line. For a description of how the offside rule can be modelled with layout constraints, refer to [S2]. An example of a declarative specification of the offside rule can be seen in the production below:

```
context-free syntax

  Stmt.Assign = <<ID> = <Exp>> {layout (offside 3) }
```

The layout constraint specifies that when the expression in the statement spans multiple lines, all following lines should be indented with respect to the column where the expression started. For example, the following program is valid according to this constraint:

```
x = 4 * 10
    ....+ 2
```


However, the following program is not valid, as the second line of the expression starts at the same column as the first line:

```
x = 4 * 10
...+ 2
```

Note that if the expression is written on a single line, the constraint is also verified. That is, the following program successfully parses:

```
x = 4 * 10 + 2
```

It is also possible to use the offside relation on different trees. For example, consider the constraint in the following production:

```
context-free syntax

Stmt.If = "if" Exp "then" then:Stmt* {layout(
    offside "if" then
)}
```

This constraint states that all lines (except the first) of the statements in the `then` branch should be indented with respect to the `if` literal. Thus, the following program is invalid according to this layout constraint, because the statement `x = 2` should be indented with relation to the topmost `if`.

```
if x < 0 then
..if y < 0 then
x = 2
```

In general, an **offside** constraint involving more than a single tree is combined with **indent** constraint to enforce that the column of the first and all subsequent lines should be indented.

- **indent**

An indent constraint indicates that the column of the first line of a certain tree should be further to the right with respect to another tree. For example, consider the following production:

```
context-free syntax

Stmt.If = "if" Exp "then" then:Stmt* {layout(
    indent "if" then
)}
```

This constraint indicates that the first line of the list of statements should be indented with respect to the `if` literal. Thus, according to this constraint the following program is valid:

```
if x < 0 then
..x = 2
```

Note that if the list of statements in the `then` branch spans multiple lines, the constraint does not apply to its subsequent lines. For example, consider the following program:

```
if x < 0 then
..x = 2 + 10
* 4
y = 3
```

This program is still valid, since the column of the first line of the first assignment is indented with respect to the `if` literal. To indicate that the first and all subsequent lines should be indented, an offside constraint should also be included.

context-free syntax

```

    Stmt.If = "if" Exp "then" then: Stmt* {layout (
        indent "if" then &&
        offside "if" then
    )}

```

With this constraint, the remainder of the expression `* 4` should also be further to the right compared to the “if” literal. The following program is correct according to these two constraints, since the second line of the first assignment and the second assignment are also indented with respect to the `if` literal:

```

if x < 0 then
  · · x = 2 + 10
  · * 4
  · y = 3

```

Finally, all these layout declarations can be ignored by the parser and used only when generating the pretty-printer. To do that, prefix the constraint with **pp-** writing, for example, **pp-offside** or **pp-align**.

Todo: Part of this documentation is not yet written.

9.3 SDF3 Examples

Todo: This part part of the documentation is not yet written.

9.4 SDF3 Configuration

When using SDF3 inside Spoofax, it is possible to specify different configuration options that. They allow using the new parser generator, specifying the shape of completion placeholders, or disable SDF altogether. These options should be specified in the `metaborg.yaml` file.

For example, to disable SDF for the current project, use:

```

language:
  sdf:
    enabled: false

```

This configuration should be present when defining language components for a language that has SDF enabled.

SDF3 allows generating placeholders for code completion. The default “shape” of placeholders is `[[Symbol]]`. However, it is possible to tweak this shape using the configuration below (the configuration for suffix is optional):

```

language:
  sdf:
    placeholder:
      prefix: "$"
      suffix: "$"

```

Currently, the path to the parse table is specified in the `Syntax.esv` file, commonly as `table: target/metaborg/sdf.tbl`. When the ESV file does not contain this entry, it is also possible to specify the path to

the parse table in the `metaborg.yaml` file. This is useful when testing an external parse table, or using a parse table different from the one being generated in the project. In the example below, the table is loaded from the path `tables/sdf.tbl`. The same can be applied to the parse table used for code completion.

```
language:
  sdf:
    parse-table: "tables/sdf.tbl"
    completion-parse-table: "tables/sdf-completions.tbl"
```

In a Spoofax project, it is also possible to use SDF2 instead of SDF3. This enables SDF2 tools such as the SDF2 parenthesizer, signature generator, etc. For example:

```
language:
  sdf:
    version: sdf2
```

By default SDF3 compilation works by generating SDF2 files, and depending on the SDF2 toolchain. However, a new (and experimental) parse table generator can be selected by writing:

```
language:
  sdf:
    sdf2table: java
```

This configuration disables the SDF2 generation, and may cause problems when defining grammars to use concrete syntax, since this feature is not supported yet by SDF3. However, the `java` parse table generator supports Unicode, whereas SDF2 generation does not. Furthermore, `dynamic` can be used instead of `java`, to enable lazy parse table generation, where the parse table is generated while the program is parsed.

A namespaced grammar can be generated automatically from an SDF3 grammar. This namespacing is done by adding the language name to all module names and sort names. The generated grammar is put in `src-gen/syntax`. The configuration to enable this is:

```
language:
  sdf:
    generate-namespaced: true
```

Note that namespacing doesn't not handle imports of grammar files from other projects very well.

9.4.1 JSGLR version

An experimental new version of the SGLR parser implementation is available: JSGLR2. It supports parsing, imploding and syntax highlighting. Error reporting, recovery and completions are currently not supported. It can be enabled with:

```
language:
  sdf:
    jsglr-version: v2
```

There are some extensions of JSGLR2 available. To use them, change the `jsglr-version` by replacing `v2` with one of the following:

data-dependent Data-dependent JSGLR2 solves deep priority conflicts using data-dependent parsing, which does not require duplicating the grammar productions.

incremental Incremental JSGLR2 reuses previous parse results to speed up parsing.

layout-sensitive Layout-sensitive JSGLR2 is documented in the [reference manual of SDF3](#).

recovery JSGLR2 with recovery tries to recover from parse errors. This extension is experimental.

recovery-incremental Incremental JSGLR2 with recovery. This extension is experimental.

9.4.2 JSGLR2 logging

Logging is available for JSGLR2. It can be enabled with:

```
language:
  sdf:
    jsclr2-logging: all
```

Since logging all parsing events is quite verbose, several other scopes are available in addition to the `all` option:

none Log nothing (default).

minimal Only log the start and end of a parse, including a measurement of total parse time (including imploding and tokenization).

parsing Log all standard parsing events (such as stack and parse forest operations, action execution, etc.) but no variant-specific events (e.g. related to recovery).

recovery Log the recovery iterations and the recovery productions that are applied.

Warning: Whenever changing any of these configurations, clean the project before rebuilding.

9.5 Migrating SDF2 grammars to SDF3 grammars

The conversion of SDF2 (.sdf) or template language (.tmpl) files into SDF3 can be done (semi) automatically.

For SDF2 files, it is possible to apply the Spoofax builder Lift to SDF3 to get a SDF3 file that corresponds to the SDF2 grammar. Another way of doing that is to apply the same builder to a definition (.def) file (in the include directory), that contains all SDF2 modules of your language. The result is a list of SDF3 files corresponding to all modules of your grammar. All SDF3 files are generated in the `src-gen/sdf3-syntax` directory.

For template language files with deprecated constructors, you can also apply the Lift to SDF3 builder, to convert the grammar into a SDF3 grammar in the `src-gen/formatted` directory.

Lift to SDF3 has two different versions: it can lift productions into templates or it can lift it into productive productions. In the case of wanting to have productive productions out of templates, the Extract productions builder can be used.

9.6 SDF3 Bibliography

SDF3 is the successor of SDF2, which is itself the successor of SDF.

- SDF [\[S3\]](#)
- SDF2 [\[S7\]](#)
- Disambiguation filters [\[S4\]](#)
- Scannerless generalized-LR parsing [\[S6\]](#)
- Template productions [\[S9\]](#) [\[S8\]](#)
- Layout sensitive syntax [\[S2\]](#)
- Syntactic completions [\[S1\]](#)

- Safe and complete disambiguation

Static Semantics Definition with NaBL2

Programs that are syntactically well-formed are not necessarily valid programs. Programming languages typically impose additional *context-sensitive* requirements on programs that cannot be captured in a syntax definition. Languages use names to identify reusable units that can be invoked at multiple parts in a program. In addition, statically typed languages require that expressions are consistently typed. The NaBL2 ‘Name Binding Language’ supports the specification of name binding and type checking rules of a language. NaBL2 uses a constraint-based approach, and uses scope graphs for name resolution.

10.1 Introduction

10.1.1 Name Resolution with Scope Graphs

10.2 Language Reference

This section gives a systematic overview of the NaBL2 language.

10.2.1 Lexical matters

Identifiers

Most identifiers in NaBL2 fall into one of two categories, which we will refer to as:

- *Lowercase identifiers*, that start with a lowercase character, and must match the regular expression `[a-z][a-zA-Z0-9_]*`.
- *Uppercase identifiers*, that start with an uppercase character, and must match the regular expression `[A-Z][a-zA-Z0-9_]*`.

Comments

Comments in NaBL2 follow the C-style:

- `// ... single line ...` for single-line comments
- `/* ... multiple lines ... */` for multi-line comments

Multi-line comments can be nested, and run until the end of the file when the closing `*/` is omitted.

10.2.2 Terms and patterns

```
term = ctor-id "(" {term ","}* ")"
      | "(" {term ","}* ")"
      | "[" {term ","}* "]"
      | "[" {term ","}* "|" term "]"
      | namespace-id? "{" term ("@" var-id)? "}"

pattern = ctor-id "(" {pattern ","}* ")"
          | "(" {pattern ","}* ")"
          | "[" {pattern ","}* "]"
          | "[" {pattern ","}* "|" pattern "]"
          | "_"
          | var-id
```

10.2.3 Modules

```
module module-id

    section*
```

NaBL2 specifications are organized in modules. A module is identified by a module identifier. Module identifiers consist of one or more names separated by slashes, as in `{name "/"}`+. The names must match the regular expression `[a-zA-Z0-9_][a-zA-Z0-9_\.\\-]*`.

Every module is defined in its own file, with the extensions `.nabl2`. The module name and the file paths must coincide.

Example. An empty module `analysis/main`, defined in a file `.../analysis/main.nabl2`.

```
module analysis/main

// work on this
```

Modules consist of sections for imports, signatures, and rule definitions. The rest of this section describes imports, and subsequent sections deal with signatures and rules.

Imports

```
imports

    module-ref*
```

A module can import definitions from other modules by importing the other module. Imports are specified in an `imports` section, which lists the modules being imported. A module reference can be:

- A module identifier, which imports a single module with that name.
- A wildcard, which imports all modules with a given prefix. A wildcard is like a module identifier, but with a dash as the last part, as in `{name "/" }+ "/" -`.

A wildcard import does not work recursively. For example, `analysis/-` would imports `analysis/functions`, and `analysis/classes`, but not `analysis/lets/recursive`.

Example. A main module importing several submodules.

```
module main
imports
  builtins
  functions/-
  classes/-
  types
```

10.2.4 Signatures

```
signature
signature*
```

Signatures contain definitions and parameters used in the specification. In the rest of this section, signatures for terms, name binding, functions and relations, and constraint rules are described.

Terms

Terms in NaBL2 are multi-sorted, and are defined in the `sorts` and `constructors` signatures.

Sorts

```
sorts
sort-id*
```

Available since version 2.3.0

The `sorts` signature lists the sorts that are available. Sort are identified by uppercase identifiers.

Example. Module declaring a single sort `Type`.

```
module example
signature
  sorts Type
```

Constructors

```
constructors
```

```
ctor-def*
```

Constructors are defined in a `constructors` signature, and identified by uppercase identifiers. Constructor definitions are written as follows:

- *Nullary constructors* are defined using `ctor-id ":" sort-id`.
- *N-ary constructors* are defined using `ctor-id ":" {sort-ref "*" }+ "->" sort-id`.

Sort references can refer to sorts defined in the signature, or to several builtin sorts. One can refer to the following sorts:

- *User-defined sorts* using its `sort-id`.
- *Tuples* using `"(" {sort-ref "*" }* ")"`.
- *Lists* using `"list(" sort-ref ")"`.
- *Generic terms* using the `"term"` keyword. The term sort contains all possible terms, and can be seen as a supertype of all other sorts.
- *Strings* using the `"string"` keyword.
- *Scopes* using the `"scope"` keyword.
- *Occurrences* using the `"occurrence"` keyword.
- *Sort variables* are written using lowercase identifiers.

For example, a module specifying the types for a language with numbers, functions, and records identified by scopes, might look like this:

```
module example

signature

  sorts Type

  constructors
    NumT : Type
    FunT : Type * Type -> Type
    RecT : scope -> Type
```

Name binding

Two signatures are relevant for name binding. One describes namespaces, that are used for occurrences, and one describes the parameters for name resolution.

Namespaces

```
namespaces
```

```
namespace-def*
```

Namespaces are defined in the `namespaces` signature. Namespaces are identified by uppercase identifiers. A namespace definition has the following form: `namespace-id (":" sort-ref)? properties?`. The optional `:" sort-ref` indicates the sort used for the types of occurrences in this namespace.

Other properties of occurrences in this namespace, are specified as a block of the form `"{" {(prop-id ":" sort-ref) ", "}* "}"`. Properties are identified by lowercase identifiers, and `type` is a reserved property keyword that cannot be used.

The following example defines three namespaces: 1) for modules, without a type or properties, 2) for classes, which has a property to record the body of the class, and 3) for variables, which has a type property, of sort `Type`. For completeness the sort declaration for `Type` is shown as well.

```
module example

signature

  sorts Type

  namespaces
    Module
    Class { body : term }
    Var : Type
```

Name resolution

```
name resolution
  labels
    label-id*
  order
    {label-order ", "}*
  well-formedness
    label-regexp
```

Name resolution parameters are specified in a `name-resolution` signature. Note that this block can only be specified once per project.

Edge labels are specified using the `labels` keyword, followed by a list of uppercase label identifiers. The label `"D"` is reserved and signifies a declaration in the same scope.

The specificity order on labels is specified using the `order` keyword, and a comma-separated list of `label-ref` `"<" label-ref` pairs. Label references refer to a label identifier, or the special label `D`.

Finally, the well-formedness predicate for paths is specified as a regular expression over edge labels, after the `well-formedness` keyword. The regular expression has the following syntax:

- A *literal* label using its `label-id`.
- *Empty* sequence using `"e"`.
- *Concatenation* with `regexp regexp`.
- *Optional* (zero or one) with `regexp "?"`.
- *Closure* (zero or more) with `regexp "*"`.
- *Non-empty* (one or more) with `regexp "+"`.
- Logical *or* with `regexp "|" regexp`.
- Logical *and* with `regexp "&" regexp`.

- *Empty* language using `"0"`, i.e., this will not match on anything.
- Parenthesis, written as `"(regexp)"`, can be used to group complex expressions.

The following example shows the default parameters, that are used if no parameters are specified:

```
name resolution
labels
  P I

order
  D < P,
  D < I,
  I < P

well-formedness
  P* I*
```

Functions and relations

Functions

```
functions

( function-id (":" sort-ref "->" sort-ref )?
  ("{" {function-case ","}* "}")? )*
```

```
function-case = pattern "->" term
```

Functions available at constraint time are defined in a `functions` signature. A function is identified by a name, followed by a type and the function cases. The cases are rewrite rules from the match in the left, to the term on the right. The function cases need to be linear, which all the variables mentioned in the right-hand side term have to be bound in the left-hand side pattern.

The type is currently not checked, but can be used to document to sorts of the elements in the function.

Example. A module that defines the `left` and `right` projection functions for pairs.

```
module example

signature

functions
  left : (Type * Type) -> Type {
    (x, y) -> x
  }
  right : (Type * Type) -> Type {
    (x, y) -> y
  }
```

Relations

```
relations
```

(continues on next page)

(continued from previous page)

```
( relation-option* relation-id
  (":" sort-ref "*" sort-ref)?
  ("{" {variance-pattern ","}* "}")? )*
```

```
relation-option = "reflexive" | "irreflexive"
                 | "symmetric" | "anti-symmetric"
                 | "transitive" | "anti-transitive"

variance-pattern = ctor-id "(" {variance ","}* ")"
                  | "[" variance "]"
                  | "(" {variance ","}* ")"

variance = "="
          | "+"relation-id?
          | "-"relation-id?
```

The relations that are available are defined in a `relations` signature. A relation is identified by a name, possibly preceded by properties of the relation, and followed by an optional type and special cases for specific constructors.

The properties that are specified are enforced at runtime. The positive properties (`reflexive`, `symmetric`, and `transitive`) ensure that all pairs that were not explicitly added to the relation are inferred. The negative properties (`irreflexive`, `anti-symmetric`, and `anti-transitive`) are checked when adding a pair to the relation, and result in an error in the program if violated. The positive and negative properties are mutually exclusive. For example, it is not allowed to specify both `reflexive` and `irreflexive` at the same time.

The type specified for the relation is currently not checked, but can be used to document the sorts of the elements in the relation.

Variance patterns are used to specify general cases for certain constructors. This can be used, for example, to add support for lists, that are checked pair-wise.

Example. Module below defines a reflexive, transitive, anti-symmetric subtype relation `sub`, with the common variance on function types, and covariant type lists.

```
module example

signature

  relations
    reflexive, transitive, anti-symmetric sub : Type * Type {
      FunT(-sub, +sub),
      [+sub]
    }
```

Rules

```
constraint generator

  rule-def*
```

The type signatures for constraint generation rules are defined in a `constraint generator` signature. Rule signatures describe the sort being matched, the sorts of any parameters, and optionally the sort of the type. A rule signature is written as `rule-id? "[" [sort-ref "^" "(" {sort-ref ","}* ")" (":" sort-ref)? "]"`. Rules are identified by uppercase identifiers.

The following example shows a module that defines a default rule for expressions, and rules for recursive and parallel bindings. The rule for expressions has one scope parameter, and expressions are assigned a type of sort `Type`. The bind rules are named, and match on the same AST sort `Bind`. They take two scope parameters, and do not assign any type to the bind construct.

```
module example

signature

  constraint generator
    [[ Expr ^ (scope) : Type ]]
    BindPar[[ Bind ^ (scope, scope) ]]
    BindRec[[ Bind ^ (scope, scope) ]]
```

NaBL2 supports higher-order rules. In those cases, the `rule-id` is extended with a list of parameters, written as `rule-id "(" {rule-id ","}* ")"`.

For example, the rule that applies some rule, given as a parameter `X`, to the elements of a list has signature `Map1 (X) [[a ^ (b)]]`. Note that we use variables `a` and `b` for the AST and parameter sort respectively, since the map rule is polymorphic.

10.2.5 Rules

```
rules

  rule*
```

The rules section of a module defines syntax-directed constraint generation rules.

Init rule

```
init ^ ( {parameter ","}* ) (":" type)? := {clause ","}* .
init ^ ( {parameter ","}* ) (":" type)? .
```

Constraint generation starts by applying the default rule to the top-level constructor. The `init` rule, which must be specified exactly once, provides the initial values to the parameters of the default rule.

Init rules come in two variants. The first variant outputs rule clauses. These can create new scopes, or defined constraints on top-level declarations. If the rule has no clauses, the rule can be closed without a clause definition. For example, `init ^ () .` is shorthand for `init ^ () := true`.

In the example module below, the default rule takes one scope parameter. The init rule creates a new scope, which will be used as the initial value for constraint generation.

```
module example

rules

  init ^ (s) := new s.

  [[ t ^ (s) ]].
```

Generation rules

```
rule-def? [[ pattern ^ ( {parameter ","}* ) (":" type)? ]] := {clause ","}* .
rule-def? [[ pattern ^ ( {parameter ","}* ) (":" type)? ]] .
```

```
rule-def = rule-id "(" {rule-id ","}* ")"?
```

Constraint generation rules are defined for the different syntactic constructs in the object language. Rules can accept a number of parameters and an optional type. The parameters are often used to pass around scopes, but can be used for other parameters as well. The rule clause consists of a comma-separated list of constraints.

Rules can be named to distinguish different versions of a rule for the same syntactic construct. Named rules can also accept rule parameters, which makes it possible to write higher-order rules. For example, the `Map(X) [[list(a) ^ (b)]]` rule accepts as argument the rule that will be applied to the elements in the list. Note that only a single rule with a certain name can be defined per AST pattern.

Rules are distinguished by name and arity, so `Map1` is different from `Map1(X)`. There is no overloading based on the number of parameters, or the presence or absence of a type.

All variables in the rule's clauses that are not bound in the pattern, the parameters, the type, or a new directive, are automatically inferred to be unification variables.

The rule form without clauses is equal to a rule that simply return `true`. For example, `[[Int(_) ^ (s) : IntT()]]` is shorthand for `[[Int(_) ^ (s) : IntT()]] := true..`

Recursive calls

```
clause = rule-ref "[" var "^" "(" {var ","}* ")" (":" term)? "]"
rule-ref = rule-id "(" {rule-ref ","}* ")"?
           | "default"
```

Recursive calls are used to invoke constraint generation for subterms of the current term. Recursive calls can only be made on parts of the program AST, therefore the term argument needs to be a variable that is bound in the current match pattern.

If no rule name is specified, the default rule will be called. Rules that are applied are selected based on the name and the term argument. To pass the default rule as an argument to a higher-order rule, the `default` keyword is used.

There is no overloading on the number of parameters or the presence or absence of a type. Calling a rule with the wrong number of parameters will result in errors during constraint collection.

Delegating to other rules is *only* supported if the delegate has the same parameters and type as the rule that is delegating.

Example. A module defining and calling different rules.

```
module example

rules

  [[ Int(_) ^ (s) : IntT() ]].

  Map1[[ [x|xs] ^ (s) : [ty|tys] ]] :=
    [[ x ^ (s) : ty ], // call default rule on head
     Map1[[ xs ^ (s) : tys ]]. // recurse on tail

  Map1(X)[[ [x|xs] ^ (s) : [ty|tys] ]] :=
```

(continues on next page)

(continued from previous page)

```
X[[ x ^ (s) : ty ]],          // call rule X on head
Map1(X)[[ xs ^ (s) : tys ]]. // recurse on tail, passing on X
```

The rule `Map1` could also be defined in terms of `Map1 (X)` as follows:

```
module example

rules

Map1[[ xs ^ (s) : tys ]] :=
  Map1(default)[[ xs ^ (s) : tys ]].
```

10.2.6 Constraints

This section gives an overview of the different constraints that can be used in clauses of constraint rules.

Base constraints & error messages

```
clause = "true"
        | "false" message?

message = "|" message-kind message-content message-position?

message-kind      = "error" | "warning" | "note"
message-content   = "\"\" chars "\"\"
                  | "$[" (chars | "[" term "]" ) * "]"
message-position  = "@" var-id
```

The two basic constraints are `true` and `false`. The constraint `true` is always satisfied, while `false` is never satisfied.

The message argument to a constraint specifies the error message that is displayed if the constraint is not satisfied. The severity of the error can be specified to be `error`, `warning` or `note`. The message itself can either be a simple string, or an interpolated string that can match terms and variables used in the rule. By default the error will appear on the match term of the rule, but using the `@t` syntax the location can be changed to `t`. The variable `t` needs to be bound in the AST pattern of the rule.

Example. Some constraints with different ways of specifying error messages.

```
false | error @t1          // generic error on whole term
false | note "Simple note"  // specific note on whole term
false | warning $[Consider [t2]] @t1 // formatted warning on first subterm
```

Term equality

```
clause = term "==" term message?
        | term "!=" term message?
```

Equality of terms is specified using equality and inequality constraints. An equality constraint `t1 == t2` specifies that the two terms need to be equal. If the terms contain variables, the solver infers values for them using unification. If unification leads to any conflicts, an error will be reported.

Inequality is specified with a constraint of the form $t1 \neq t2$. Inequality constraints cannot always be solved if both sides contain variables. The inequality between two variables depends on the values that will be inferred for them. Only after a value is assigned, will the inequality be tested. If the constraint cannot be solved, because some variables have remained free, an error is reported as well.

Example. A few constraints for term (in)equality.

```
ty == FunT(ty1, ty2) | error $[Expected function type, but got [ty]]
ty != NilT()
```

Name binding

Name binding is concerned with constraints for building a scope graph, constraints for resolving references and accessing properties of declarations, and name sets that talk about sets of declarations or references in the scope graph.

Occurrences

```
occurrence = namespace-id? "{" term position? "}"
position    = "@" var-id
```

References and declarations in the scope graph are not simply names, but have a richer representation called an occurrence. An occurrence consists of the name, the namespace, and a position.

The name can be any term, although usually it is a term from the AST. Names are not restricted to strings, and can contain terms with subterms if necessary. However, it is required that the name contains only literals, or variables that are bound in the match pattern.

Namespaces allow us to separate different kinds of names, so that type names do not resolve to variables or vice versa. If there is only one namespace in a language, it can be omitted and the default namespace will be used.

The position is necessary to differentiate different occurrences of the same name in the program, and is the connection between the AST and the scope graph. The position can usually be omitted, in which case the position of the name is taken if it is an AST term, or the position of the match term, if the name is a literal.

Example. Several of occurrences, with explicit and implicit components.

```
Var{x}           // variable x, x must be bound in match
{y}             // no namespace, y must be bound in match
Type{a}         // type a, a must be bound in match
Var{"this" @c}  // this variable with explicit position
Var{This()}     // this variable using a constructor instead of a string
Type{"Object"}  // literal type occurrence
```

Scope graph

```
clause = occurrence "<-" scope
        | occurrence "->" scope
        | scope "-label-id->" scope
        | occurrence "=label-id=>" scope
        | occurrence "<=label-id=" scope
        | "new" var-id*
```

Scope graph constraints construct a scope graph. Names in the graph are represented by occurrences. Scopes in the graph are abstract, but can be created using the `new` directive. Rules usually receive scope parameters that allows them to extend and connect to the surrounding scope graph.

The following scope graph constraints are available:

- Declarations are introduced with `d <- s`. The arrow points from the scope to the declaration, which indicates that the occurrence is reachable from scope `s`.
- References are introduced with `r -> s`. The arrow points from the reference to the scope, indicating that the reference should be resolved in scope `s`. Note that a reference in the scope graph is not automatically required to resolve to a declaration. To check that, a resolution constraint needs to be specified.
- A direct edge from one scope to another is written as `s1 -l-> s2`. This indicates that the scope `s2` is reachable from `s1`. If the label is omitted, as in `s1 --> s2`, the label `P` is used implicitly.
- An associated scope, written as `d =l=> s`, exports the declarations visible in `s` via the declaration `d`. If the label is omitted, as in `d ===> s`, the label `I` is used implicitly.
- Declarations from an associated scope can be imported from reference `r` with an import edge, written as `r <=l= s`. Given that the reference `r` resolves to a declaration `d`, and `d` has an associated scope edge labeled `l` to a scope `s'`, the declarations visible in `s'` will become visible in `s`. Similarly to the associated scope edge, if the label on an import edge is omitted, as in `r <=== s`, the label `I` is used implicitly. Note that the import edge does not specify where the imported reference has to be resolved. Make sure that the reference itself is also added to the graph with a reference edge.
- Scopes need to be explicitly created. This is done with the `new s` constraint, which creates a new scope and binds it to the variable `s`.

A few restrictions apply to the scope graph. An occurrence can only be a reference in one scope. Similarly, an occurrence can only be a declaration in one scope. However, it is possible to use an occurrence as both a declaration and a reference.

It is important to note that all scope graph constraints *define* a scope graph. This means that the scopes or occurrences generally should not contain constraint variables. The exception is the target scope in a direct edge, or the reference in an import edge. This allows modeling type-dependent name resolution.

Example. Rules that build a scope graph.

```
[[ Module(x,imps,defs) ^ (s) ]] :=
  Mod{x} <- s,           // module declaration
  new ms,                // new scope for the module
  Mod{x} ==> ms,          // associate module scope with the declaration
  Map2[[imps ^ (ms, s) ]], // recurse on imports
  Map1[[defs ^ (ms) ]].   // recurse on statements

[[ Import(x) ^ (ms, s) ]] :=
  Mod{x} -> s,           // module reference
  Mod{x} <=== ms.        // import in the module scope

[[ TypeDef(x,_) ^ (s) ]] :=
  Type{x} <- s.          // type declaration

[[ TypeRef(x) ^ (s) ]] :=
  Type{x} -> s.          // type reference

[[ VarDef(x, t) ^ (s) ]] :=
  Var{x} <- s,           // variable declaration
  [[ t ^ (s) ]].         // recurse on type annotation
```

(continues on next page)

(continued from previous page)

```
[[ VarRef(x) ^ (s) ]] :=
  Var{x} -> s.           // variable reference
```

Name resolution

```
clause = occurrence "|->" occurrence message?
        | occurrence "?="label-id">" scope message?
        | occurrence"."prop-id ":"=" term priority? message?
        | occurrence ":" type priority? message?

priority = "!"*
```

The following constraints are available to resolve references to declarations, and specify properties of declarations:

- Resolution of a reference occurrence r to a declaration occurrence d is written as $r \mid\rightarrow d$. Resolution constraints can infer the declaration given a reference, however not the other way around. A resolution constraint requires that a reference resolves to exactly one declaration, or else it will fail.
- A lookup of the associated scope s of a declaration d is done with $d \text{ ?}=\text{label}\Rightarrow s$, where label is the edge label. If the label is omitted, as in $d \text{ ?}==\Rightarrow s$, the label I is used implicitly.
- A property p of a declaration d is specified with $d.p := t$. If multiple constraints for the same property of the same declaration exist, their values will be unified. Priorities can be used to guide where errors will be reported. If two constraints for the same property of the same declaration are in conflict, the error will more likely be reported on the constraint with the least priority annotations.
- A special form exists for the type property, which is preferably written as $d : t$.

Example. Rules that include resolution constraints and declaration types.

```
[[ TypeDef(x,t) ^ (s) ]] :=
  Type{x} <- s,           // type declaration
  Type{x} : t !.          // semantics type of the type declaration

[[ TypeRef(x) ^ (s) : ty ]] :=
  Type{x} -> s.           // type reference
  Type{x} \mid\rightarrow d,    // resolve reference
  d : ty.                 // semantic type of the resolved declaration

[[ VarDef(x, t) ^ (s) ]] :=
  Var{x} <- s,            // variable declaration
  [[ t ^ (s) : ty ]],     // recurse on type annotation
  Var{x} : ty !.         // type of the variable declaration

[[ VarRef(x) ^ (s) : ty ]] :=
  Var{x} -> s,            // variable reference
  Var{x} \mid\rightarrow d,      // resolve variable reference
  d : ty.                // type of resolved declaration
```

Name sets

```
name-set = "D(" scope ")"("/namespace-id)?
           | "R(" scope ")"("/namespace-id)?
```

(continues on next page)

(continued from previous page)

```
| "V(" scope ") "("/"namespace-id)?
| "W(" scope ") "("/"namespace-id)?

set-proj = "name"

message-position = "@NAMES"
```

Name sets are set expressions (see [Sets](#)) that are based on the scope graph. They can be used in set constraints to test for properties such as duplicate names, shadowing, or complete coverage.

The expression $D(s)$ represents the set of all declarations in scope s , and similarly $R(s)$ refers to all references in scope s . The set of all visible declarations in s is represented by $V(s)$, and the set of all reachable declarations in s is represented by $W(s)$. The difference between visible and reachable declarations is that the former takes the label order into account to do shadowing, while the latter only considers path well-formedness but does not shadow.

All the name sets can also be restricted to declarations or references in a particular namespace by appending a forward-slash and the namespace. For example, all variable declarations in scope s would be represented by $D(s)/\text{Var}$.

Name sets support the `name` set projection that makes it possible to compare occurrences by name only, ignoring the namespace and position.

When using set constraints on name sets, there are some options to relate the error messages to the elements in the set, instead of the term where the constraint was created. First, the position in the message can be set to `@NAMES`. This makes error messages appear on the names in the set. For example, `distinct/name D(s)/Var | error @NAMES` will report errors on all the names that are duplicate. If you want to refer to the name in the error message, use the `NAME` keyword. For example, in the message `error ${Duplicate name [NAME]} @NAMES`, the keyword will be replaced by the correct name.

Sets

```
clause = "distinct" ("/"set-proj)? set-expr message?
| set-expr "subsetq" ("/"set-proj)? set-expr message?
| set-expr "seteq" ("/"set-proj)? set-expr message?

set-expr = "0"
| "(" set-expr "union" set-expr ")"
| "(" set-expr "isect" ("/"set-proj)? set-expr ")"
| "(" set-expr "minus" ("/"set-proj)? set-expr ")"
| name-set
```

Set constraints are used to test for distinct elements in a set, or subset relations between sets. Set expressions allow the usual set operations such as union and intersection.

The constraint `distinct S` check whether any elements in the set S appears multiple times. Note that this works because the sets behave like multisets, so every element has a count associated with it as well. If the set S supports projections, it is possible to test whether the set contains any duplicates after projection. Which projections are available depends on the sets involved. For example, when working with sets of occurrences (see [Name sets](#)), the `name` projection can be used.

Constraints `S1 subsetq S2` and `S1 seteq S2` test whether $S1$ is a subset of $S2$, or if the sets are equal, respectively. Both constraints also support projections, written as `S1 subsetq/proj S2` and `S1 seteq/proj S2`.

The basic set expressions that are supported are `0` for the empty set, `(S1 union S2)` for set union, `(S1 isect S2)` for intersection, and `(S1 minus S2)` for set difference. Set intersection and difference can also be performed under projection, written as `(S1 isect/proj S2)` and `(S1 minus/proj S2)`. This means the comparison

to check if elements from the two sets match is done after projecting the elements. However, the resulting set will still contain the original elements, not the projections. For example, this can be used to compute sets of occurrences where a comparison by name is necessary.

Example. Set constraints over name sets.

```
distinct/name D(s)/Var // variable names in s must be unique
```

Functions and relations

```
clause = term "<"relation-id?"!" term message?
        | term "<"relation-id?"?" term message?
        | term "is" function-ref "of" term message?

function-ref = function-id
              | relation-id".lub"
              | relation-id".glb"
```

Symbolic

```
clause = "?-" term
        | "!" term
```

10.3 Stratego API

The Stratego API to NaBL2 allows the customization of certain parts of the analysis process, and access to analysis result during after analysis.

The full definition of the API can be found in the [nabl2/api](#) module.

10.3.1 Setup

Using the Stratego API requires a dependency on the NaBL2 runtime, and an import of `nabl2/api`.

Example. A Stratego module importing the NaBL2 API.

```
module example

imports

  nabl2/api
```

10.3.2 Customizing analysis

Several aspects of the analysis process can be customized by implementing hooks in Stratego.

Custom pretty-printing

By default the object language terms that are mentioned in error messages are printed as generic terms. This may lead to error messages like `Expected number, but got FunT(IntT(), Int())`. By implementing the `nabl2-prettyprint-hook`, the term might be pretty-printed, resulting in a message like `Expected number, but got 'int -> int'`. Care needs to be taken though, because the terms might contain constraint variables, that are not part of the object language and would break pretty-printing. This can be fixed by injecting the `nabl2-prettyprint-term` strategy into the object language pretty-printing rule.

Example. A module that implements the pretty-printing hooks to format types from the object language. This assumes that types are defined in an SDF3 file using the sort `Type`.

```
module example

imports

  nabl2/api

rules

  nabl2-prettyprint-hook    = prettyprint-YOURLANG-Type
  prettyprint-YOURLANG-Type = nabl2-prettyprint-term
```

Custom analysis

Analysis in NaBL2 proceeds in three phases. An initial phase is used to create initial scopes and parameters. A second phase is used to collect, and partially solve, constraints per compilation unit. A final phase solves constraints. The constraints in the final phase are those of all compilation units combined, if multi-file analysis is enabled.

It is possible to run custom code after each of these phases, by implementing `nabl2-custom-analysis-init-hook`, `nabl2-custom-analysis-unit-hook`, and `nabl2-custom-analysis-final-hook(|a)`.

The initial hook receives a tuple of a resource string and AST node as its argument. The result type of the initial hook is free.

The unit hook receives a tuple of a resource string, and AST node, and the initial result if the initial hook was implemented. If the initial hook is not implemented, an empty tuple `()` is passed as the initial result. The result type of the unit hook is free.

The final hook receives a tuple of a resource string, the result if the initial hook, and a list of results of the unit hook. The initial result will again be `()` if the initial hook is not implemented. The unit results might be an empty list if the unit hook is not implemented. The final hook also receives an term argument for the analysis result, that can be passed to the strategies to access the analysis result. The final hook should return a tuple of errors, warnings, notes, and a custom result. Messages should be tuples `(origin-term, message)` of the origin term from the AST, and the message to report.

Custom analysis hooks are advised to use the strategies `nabl2-custom-analysis-info-msg(|msg)` and `nabl2-custom-analysis-info(|msg)` to report to the user. The first version just logs the message term, while the second version also logs the current term. If formatting the info messages is expensive, the strategy `nabl2-is-custom-analysis-info-enabled` to check if logging is actually enabled. The advantage of using these logging strategies is that they are influenced by the logging settings in the project configuration.

Example. A Stratego module that shows how to set up custom analysis strategies.

```
module example
```

(continues on next page)

(continued from previous page)

```
imports

  nabl2/api

rules

  nabl2-custom-analysis-init-hook:
    (resource, ast) -> custom-initial-result
    with nabl2-custom-analysis-info-msg(|"Custom initial analysis step");
    custom-initial-result := ...

  nabl2-custom-analysis-unit-hook:
    (resource, ast, custom-initial-result) -> custom-unit-result
    with <nabl2-custom-analysis-info(|"Custom unit analysis step")> resource;
    custom-unit-result := ...

  nabl2-custom-analysis-final-hook(|a):
    (resource, custom-initial-result, custom-unit-results) -> (errors, warnings,
↳notes, custom-final-result)
    with nabl2-custom-analysis-info-msg(|"Custom final analysis step");
    custom-final-result := ... ;
    errors      := ... ;
    warnings    := ... ;
    notes       := ...
```

10.3.3 Querying analysis

The analysis API gives access to the result of analysis. The analysis result is available during the final custom analysis step, or in post-analysis transformations.

The API defines several strategies to get an analysis term by resource name or from an AST node. This analysis term can then be passed to the querying strategies that give access to the scope graph, name resolution, etc.

Getting the analysis result

```
/**
 * Get analysis for the given AST node
 *
 * @type node:Term -> Analysis
 */
nabl2-get-ast-analysis

/**
 * Get analysis for the given resource
 *
 * @type filename:String -> Analysis
 */
nabl2-get-resource-analysis

/**
 * Test if analysis has errors
 *
 * Fails if there are no errors, succeeds otherwise.
 */
```

(continues on next page)

(continued from previous page)

```
* @type Analysis -> _
*/
nabl2-analysis-has-errors
```

There are two ways to get the result of analysis. The first is calling `nabl2-get-ast-analysis` on a node if the analyzed AST. The second is to call `nabl2-get-resource-analysis` with a resource name. The resulting term can be passed as a term argument to the different query strategies.

To check if analysis was successful, the strategy `nabl2-analysis-has-errors` can be used. This strategy will succeed if any errors were encountered, and fail otherwise.

Example. Builder that only runs if analysis has no errors.

```
module example

imports

  nabl2/api

rules

  example-builder:
    (_, _, ast, path, project-path) -> (output-file, result)
    where analysis := <nabl2-get-resource-analysis> $[[project-path]/[path]];
           <not (nabl2-analysis-has-errors)> analysis
    with output-file := ... ;
         result      := ...
```

AST properties

```
/**
 * @param a : Analysis
 * @type node:Term -> Term
 */
nabl2-get-ast-params(|a)

/**
 * @param a : Analysis
 * @type node:Term -> Type
 */
nabl2-get-ast-type(|a)
```

AST nodes are associated with the parameters and (optionally) the type mentioned in the rule that was applied to the node. For example, if a rule like `[[e ^ (s) : ty]]` was applied to an expression in the AST, it is possible to query the analysis for the scope `s` and the type `ty`. The strategy `nabl2-get-ast-params(|a)` expects an AST node, and returns a tuple with the parameters. Similarly `nabl2-get-ast-type(|a)` expects an AST node and returns the type. If no type was specified, for example in a rule such as `[[e ^ (s1, s2)]]`, the call will fail. The term argument `a` should be an analysis result.

Nodes in the AST are indexed to make the connection between the AST and the analysis result. The following strategies can be used to preserve or manipulate AST indices. Note that this has no effect on the result of analysis, so whether such manipulation is sound is up to the user.

```
/**
 * Get AST index. Fails if term has no index.
```

(continues on next page)

(continued from previous page)

```

*
* @type Term -> TermIndex
*/
nabl2-get-ast-index

/**
* Set AST index on a term. Throws an exception if the index argument
* is not a valid index.
*
* @param index : TermIndex
* @type Term -> Term
*/
nabl2-set-ast-index(|index)

/**
* Copy AST index from one term to another. Fails if the source has no
* index.
*
* @param from : TermIndex
* @type Term -> Term
*/
nabl2-copy-ast-index(|from)

/**
* Execute a strategy and copy the index of the input term to the output
* term. If the original term has no index, the result of applying s is
* returned unchanged. Thus, failure behaviour of s is preserved.
*
* @type Term -> Term
*/
nabl2-preserve-ast-index(s)

/**
* Erase AST indices from a term, preserving other annotations and
* attachments.
*
* @type Term -> Term
*/
nabl2-erase-ast-indices

```

Scope graph & name resolution

The strategies concerning scope graphs and name resolution are organized in three groups. The first group are strategies to create and query occurrences in the scope graph. The second group gives access to the structure of the scope graph. The third group exposes the result of name resolution, as well as types and properties that are set on declarations.

Working with occurrences

```

/**
* Make an occurrence in the default namespace
*
* NaBL2 equivalent: {node}
*

```

(continues on next page)

(continued from previous page)

```

* @type node:Term -> Occurrence
*/
nabl2-mk-occurrence

/**
* Make an occurrence in the specified namespace
*
* NaBL2 equivalent: ns{node}
*
* @param ns : String
* @type node:Term -> Occurrence
*/
nabl2-mk-occurrence(|ns)

/**
* Make an occurrence in the specified namespace, using an origin term
*
* NaBL2 equivalent: ns{node @t}
*
* @param ns : String
* @param t : Term
* @type node:Term -> Occurrence
*/
nabl2-mk-occurrence(|ns,t)

/**
* Get namespace of an occurrence
*
* @type Occurrence -> ns:String
*/
nabl2-get-occurrence-ns

/**
* Get name of an occurrence
*
* @type Occurrence -> Term
*/
nabl2-get-occurrence-name

```

Querying the scope graph

```

/**
* Get all declarations in the scope graph
*
* @param a : Analysis
* @type _ -> List(Occurrences)
*/
nabl2-get-all-decls(|a)

/**
* Get all references in the scope graph
*
* @param a : Analysis
* @type _ -> List(Occurrences)

```

(continues on next page)

(continued from previous page)

```

*/
nabl2-get-all-refs(|a)

/**
 * Get all scopes in the scope graph
 *
 * @param a : Analysis
 * @type _ -> List(Scope)
 */
nabl2-get-all-scopes(|a)

/**
 * Get the scope of a reference
 *
 * @param a : Analysis
 * @type ref:Occurrence -> Scope
 */
nabl2-get-ref-scope(|a)

/**
 * Get the scope of a declaration
 *
 * @param a : Analysis
 * @type decl:Occurrence -> Scope
 */
nabl2-get-decl-scope(|a)

/**
 * Get declarations in a scope
 *
 * @param a : Analysis
 * @type Scope -> List(Occurrence)
 */
nabl2-get-scope-decls(|a)

/**
 * Get references in a scope
 *
 * @param a : Analysis
 * @type Scope -> List(ref:Occurrence)
 */
nabl2-get-scope-refs(|a)

/**
 * Get direct edges from a scope
 *
 * @param a : Analysis
 * @type Scope -> List((Label, Scope))
 * @type (Scope, Label) -> List(Scope)
 */
nabl2-get-direct-edges(|a)

/**
 * Get inverse direct edges from a scope
 *
 * @param a : Analysis
 * @type Scope -> List((Label, Scope))

```

(continues on next page)

(continued from previous page)

```
* @type (Scope,Label) -> List(Scope)
*/
nabl2-get-direct-edges-inv(|a)

/**
 * Get import edges from a scope
 *
 * @param a : Analysis
 * @type Scope -> List((Label,ref:Occurrence))
 * @type (Scope,Label) -> List(ref:Occurrence)
 */
nabl2-get-import-edges(|a)

/**
 * Get inverse import edges from a reference
 *
 * @param a : Analysis
 * @type ref:Occurrence -> List((Label,Scope))
 * @type (ref:Occurrence,Label) -> List(Scope)
 */
nabl2-get-import-edges-inv(|a)

/**
 * Get associated scopes of a declaration
 *
 * @param a : Analysis
 * @type decl:Occurrence -> List((Label,Scope))
 * @type (decl:Occurrence,Label) -> List(Scope)
 */
nabl2-get-assoc-edges(|a)

/**
 * Get associated declarations of a scope
 *
 * @param a : Analysis
 * @type Scope -> List((Label,decl:Occurrence))
 * @type (Scope,Label) -> List(decl:Occurrence)
 */
nabl2-get-assoc-edges-inv(|a)
```

Querying name resolution

```
/**
 * @param a : Analysis
 * @type decl:Occurrence -> Type
 */
nabl2-get-type(|a)

/**
 * @param a : Analysis
 * @param prop : String
 * @type decl:Occurrence -> Term
 */
nabl2-get-property(|a,prop)
```

(continues on next page)

(continued from previous page)

```
/**
 * @param a : Analysis
 * @type ref:Occurrence -> (decl:Occurrence, Path)
 */
nabl2-get-resolved-name(|a)

/**
 * @param a : Analysis
 * @type ref:Occurrence -> List((decl:Occurrence, Path))
 */
nabl2-get-resolved-names(|a)

/**
 * Get visible declarations in scope
 *
 * @param a : Analysis
 * @type Scope -> List(Occurrence)
 */
nabl2-get-visible-decls(|a)

/**
 * Get reachable declarations in scope
 *
 * @param a : Analysis
 * @type Scope -> List(Occurrence)
 */
nabl2-get-reachable-decls(|a)
```

10.4 Configuration

We will show you how to prepare your project for use with NaBL2, and write your first small specification.

10.4.1 Prepare your project

You can start using NaBL2 by creating a new project, or by modifying an existing project. See below for the steps for your case.

Start a new project

If you have not done this already, install Spoofax Eclipse, by following the [installation instructions](#).

Create a new project by selecting New > Project... from the menu. Selecting Spoofax > Spoofax language project from the list, and click Next. After filling in a project name, an identifier, name etc will be automatically suggested. To use NaBL2 in the project, select NaBL2 as the analysis type. Click Finish to create the project.

Convert an existing project

If you have an existing project, and you want to start using NaBL2, there are a few changes you need to make.

First of all, make sure the `metaborg.yaml` file contains at least the following dependencies.

```
---
# ...
dependencies:
  compile:
    - org.metaborg:org.metaborg.meta.nabl2.lang:${metaborgVersion}
  source:
    - org.metaborg:org.metaborg.meta.nabl2.shared:${metaborgVersion}
    - org.metaborg:org.metaborg.meta.nabl2.runtime:${metaborgVersion}
```

We will set things up, such that analysis rules will be grouped together in the directory `trans/analysis`. Create a file `trans/analysis/main.str` that contains the following.

```
module analysis/main

imports

  nabl2shared
  nabl2runtime
  analysis/-
```

Add the following lines to your main `trans/LANGUAGE.str`.

```
module LANGUAGE

imports

  analysis/main

rules

  editor-analyze = analyze(desugar-pre,desugar-post)
```

If your language does not have a desugaring step, use `analyze(id, id)` instead.

Finally, we will add reference resolution and menus to access the result of analysis, by adding the following lines to `editor/Main.esv`.

```
module Main

imports

  nabl2/References
  nabl2/Menus
```

You can now continue to the [example specification here](#), or directly to the [language reference](#).

10.4.2 Runtime settings

Multi-file analysis

By default, files are analyzed independently of each other. Files can also be analyzed in the project context. This allows cross-file references, imports, et cetera. This is called `multifile` mode, and is configured the ESV files of a language definition. To enable multi-file mode, add the `(multifile)` option to the `observer`:

```
observer = editor-analyze (multifile)
```

Logging

The log output of NaBL2 analysis can be controlled by setting the `runtime.nabl2.debug` option in a projects `metaborg.yaml`.

The following debug flags are recognized:

- `analysis` enables summary output about the analysis; number of files analyzed and overall runtime.
- `files` enables output about individual files; which files are being analyzed.
- `collection` enables output about constraint collection; a trace of the rules are applied during collection.
- `timing` enables output about the runtimes of different parts of the analysis.
- `all` enables all possible output.

For example, to enable summary output about the analysis, add the following to a projects `metaborg.yaml`:

```
runtime:
  nabl2:
    debug: analysis
```

10.4.3 Inspecting analysis results

You can debug your specification by inspecting the result of analysis, and by logging a trace of the rules that get applied during constraint generation.

The result of analysis can be inspected, by selecting elements from the Spoofax > NaBL2 Analysis the menu. For multifile projects, use the `Project` results, or the `File` results for singlefile projects.

10.5 Examples

10.6 Bibliography

Relevant papers.

10.7 NaBL

This is the NaBL reference manual. In Spoofax, name binding is specified in NaBL. NaBL stands for *Name Binding Language* and the acronym is pronounced ‘enable’. Name binding is specified in terms of:

- namespaces
- binding instances (name declarations)
- bound instances (name references)
- scopes
- imports

10.7.1 Namespaces

A namespace is a collection of names and is not necessarily connected to a specific language concept. Different concepts can contribute names to a single namespace. For example, in Java, classes and interfaces contribute to the same namespace, as do variables and parameters. Namespaces are declared in the `namespaces` section of a language definition:

```
namespaces

Module Entity Property Function Variable
```

Note: Some languages such as C# provide namespaces as a language concept to scope the names of declarations such as classes. It is important to distinguish these namespaces as a language concept from NaBL namespaces as a language definition concept. The two are *not* related.

10.7.2 Name Binding Rules

Name bindings are specified in name binding rules. Each rule consists of a term pattern (a term that may contain variables and wildcards `_`) and a list of name binding clauses about the language construct matched by the pattern. There are four different kinds of clauses for definition sites, use sites, scopes, and imports.

Definition Sites

The following rules declare definition sites for module and entity names:

```
rules

Module(m, _): defines non-unique Module m
Entity(e, _): defines unique Entity e
```

The patterns in these rules match module and entity declarations, binding variables `m` and `e` to module and entity names, respectively. These variables are then used in the clauses on the right-hand sides. In the first rule, the clause specifies any term matched by `Module(m, _)` to define a name `m` in the `Module` namespace. Similarly, the second rule specifies any term matched by `Entity(e, _)` to define a name `e` in the `Entity` namespace.

Consider the following example module:

```
module shopping

entity Item {
  name : String
}
```

The parser turns this into an abstract syntax tree, represented as a term:

```
Module(
  "shopping"
, [ Entity(
    "Item"
  , [ Property("name", EntityType("String")) ]
  )
]
)
```


The patterns in name binding rules match subterms of this term, indicating definition and use sites. The whole term is a definition site of the module name `shopping`. The first name binding rule specifies this binding. Its pattern matches the term and binds `m` to `"shopping"`. Similarly, the subterm `Entity("Item", ...)` is a definition site of the entity name `Item`. The pattern of the second name binding rule matches this term and binds `e` to `"Item"`.

While entity declarations are `unique` definition sites, module declarations are `non-unique` definition sites. That is, multiple module declarations can share the same name. This allows language users to spread the content of a module over several files, similar to Java packages. Definition sites are by default `unique`, so the `unique` keyword is only optional and can be omitted. For example, the following rules declare `unique` definition sites for property and variable names:

```
Property(p, _): defines Property p
Param(p, _)   : defines Variable p
Declare(v, _)  : defines Variable v
```

Note: Spoofax distinguishes the name of a namespace from the sort and the constructor of a program element: in the last rule above, the sort of the program element is `Statement`, its constructor is `Declare`, and it defines a name in the `Variable` namespace. By distinguishing these three things, it becomes easy to add or exclude program elements from a namespace. For example, return statements are also of syntactic sort `Statement`, but they do not correspond to any namespace. On the other hand, function parameters also define names in the `Variable` namespace, even though (in contrast to variable declarations) they do not belong to the syntactic sort `Statement`.

Use Sites

Use sites refer to definition sites of names. They can be declared similarly to definition sites. The following rule declares use sites for entity names:

```
Type(t): refers to Entity t
```

Use sites might refer to different names from different namespaces. For example, a variable might refer either to a `Variable` or a `Property`. This can be specified by exclusive resolution options:

```
Var(x):
  refers to Variable x
  otherwise refers to Property x
```

The `otherwise` keyword signals ordered alternatives: only if the reference cannot be resolved to a variable, Spoofax will try to resolve it to a property. As a consequence, variable declarations shadow property definitions. If this is not intended, constraints can be defined to report corresponding errors.

Scopes

Scopes restrict the visibility of definition sites. For example, an entity declaration scopes property declarations that are not visible from outside the entity.

```
entity Customer {
  name : String // Customer.name
}

entity Product {
  name : String // Product.name
}
```

In this example, the two `name` properties both reside in the `Property` namespace, but can still be distinguished: if `name` is referred in a function inside `Customer`, it refers the one in `Customer`, not the one in `Product`.

Simple Scopes

Scopes can be specified in scope clauses. They can be nested and name resolution typically looks for definition sites from inner to outer scopes. In the running example, modules scope entities, entities scope properties and functions, and functions scope local variables.

```
Module(m, _):
  defines Module m
  scopes Entity
Entity(e, _):
  defines Entity e
  scopes Property, Function
Function(f, _):
  defines Function f
  scopes Variable
```

As these examples illustrate, scopes are often also definition sites. However, this is not a requirement. For example, a block statement has no name, but scopes variables:

```
Block(_): scopes Variable
```

Definition Sites with Limited Scope

Many definitions are visible in their enclosing scope: entities are visible in the enclosing module, properties and functions are visible in the enclosing entity, and parameters are visible in the enclosing function. However, this does not hold for variables declared inside a function. Their visibility is limited to statements after the declaration. The following rule restricts the visibility to the subsequent scope:

```
Declare(v, _):
  defines Variable v in subsequent scope
```

Similarly, the iterator variable in a for loop is only visible in its condition, the update, and the loop's body, but not in the initializing expression. This can be declared as follows:

```
For(v, t, init, cond, update, body):
  defines Variable v in cond, update, body
```

Scoped References

Typically, use sites refer to names which are declared in its surrounding scopes. But a use site might also refer to definition sites which reside outside its own scope. For example, a property name in an expression might refer to a property in another entity:

```
entity Customer {
  name : String
}

entity Order {
  customer : Customer
```

(continues on next page)

(continued from previous page)

```
function getCustomerName(): String {
    return customer.name;
}
```

Here, `name` in `customer.name` refers to the property in entity `Customer`. The following name binding rule is a first attempt to specify this:

```
PropAccess(exp, p):
    refers to Property p in Entity e
```

But this rule does not specify which entity `e` is the right one. Interaction with the type system is required in this case:

```
PropAccess(exp, p):
    refers to Property p in Entity e
    where exp has type EntityType(e)
```

This rule extracts `e` from the type of the expression `exp`. We will later discuss interactions with the type system in more detail.

Imports

Many languages offer import facilities to include definitions from another scope into the current scope. For example, a module can import other modules, making entities from the imported modules available in the importing module:

```
module order

import banking

entity Customer {
    name    : String
    account: BankAccount
}
```

Here, `BankAccount` is not declared in the scope of module `order`. However, module `banking` declares an entity `BankAccount` which is imported into module `order`. The type of property `account` should refer to this entity. This can be specified by the following name binding rule:

```
Import(m):
    imports Entity from Module m
```

This rule has two effects. First, `Import(m)` declares use sites for module names. Second, entities declared in these modules become visible in the current scope.

10.7.3 Interaction with Type System

We can associate names with type information. This type information is specified at definition sites, and accessed at use sites. The following name binding rules involve type information at definition sites:

```
Property(p, t):
    defines Property p of type t
Param(p, t):
    defines Variable p of type t
```

These rules match property and parameter declarations, binding their name to p and their type to t . Spoofax remembers t as type information about the property or parameter name p . In this example, the type is explicitly declared in property and parameter declarations. But the type of a definition site is not always explicitly declared but needs to be inferred by the type system. For example, variable declarations might come with an initial expression, but without an explicit type.

```
var x = 42;
```

The type of x is the type of its initial expression 42 . To make the type of x explicit, the type of the initial expression needs to be inferred by the type system. The following name binding rule makes this connection between name binding and type system:

```
Declare(v, e) :  
  defines Variable v  
    of type t  
    in subsequent scope  
    where e has type t
```

Note: The predecessor of NaBL2, the NaBL/TS name binding and type analysis meta-language is deprecated.

Static Semantics Definition with Statix

Programs that are syntactically well-formed are not necessarily valid programs. Programming languages typically impose additional *context-sensitive* requirements on programs that cannot be captured in a syntax definition. Languages use names to identify reusable units that can be invoked at multiple parts in a program. In addition, statically typed languages require that expressions are consistently typed. The Statix language supports the specification of name binding and type checking rules of a language. The rules of the static semantics are written as logic rules, and solved using a constraint-based approach, and uses scope graphs for name resolution.

11.1 Getting Started

Note: Generate a language project that uses Statix by following [this guide](#) and selecting Statix for Analysis type.

The best way to get started on Statix is to use the the lectures from the TU Delft compiler construction course. These lectures explain the concepts that are used in Statix, and discuss scope graph patterns and Statix rules for several language features. In particular, these are the relevant lectures:

- [Type Checking and Type Constraints](#) introduces type checking concepts and the basics of Statix specifications.
- [Name Binding and Name Resolution](#) introduces scope graphs in depth, discusses many name binding patterns in terms of scope graphs and queries, and shows example Statix rules for many of these patterns.
- [Constraint Semantics and Constraint Resolution](#) Discusses in the semantics of Statix and some of the algorithms that are used in its implementation.

The lecture pages also link to other presentations and tutorials on Statix and scope graphs.

Example projects using Statix can be found in [this repository](#). The **‘STL-Crec**<https://github.com/metaborg/nabl/tree/master/statix.integrationtest/lang.stlcrec>’ project is the simplest, and shows a simply-typed lambda calculus extended with structural records. The **‘Units**<https://github.com/metaborg/nabl/tree/master/statix.integrationtest/lang.units>’ project contains definitions for a language that supports various module and package features. The **‘FGJ**<https://github.com/metaborg/nabl/tree/master/statix.integrationtest/lang.fgj>’ project shows a more

advanced specification for Featherweight Generic Java, showing scopes as types, complex scope graph patterns, and lazy-substitution-based generics.

11.2 Debugging

This section describes several techniques that can be used to debug your Statix specification if it does not work as you expect.

Note: The single most useful thing you can do when debugging is to **make the problem as small as possible!** All the techniques you can use for debugging are more effective when the problem is as small as possible. Try to find the smallest example program and Statix specification that still exhibits your problem, and focus on those.

There are three main categories of problems you may encounter:

1. Errors reported in your Statix files. These may come from syntax errors, unresolved names for modules or predicates, type errors, or problems with illegal scope extension. When these errors are unexpected and not mentioned in *Some Common Problems*, follow the steps in *Basic Checklist* and *Creating Minimal Examples*. If that does not help out, please follow the steps in *Getting Help and Reporting Issues*.
2. Unexpected behavior when running Statix on files of your language. This is the most common kind of problems. All debugging techniques after the *Basic Checklist* are focused on finding and debugging problems in the definitions in your specification. **Note that it is useless to try to solve problems of this kind when your specification still has errors!**
3. Analysis fails altogether. This usually results in `Analysis failed` errors at the top of files or on the project in the *Package Explorer*. Check the *Console* and *Error Log* for reported Java exceptions that can be included in a bug report.

11.2.1 Basic Checklist

These are some basic things that should always be checked when you encounter a problem:

- See if the disappears after a clean build (run `Project > Clean...` and then `Project > Build Project` on your project). If the problem disappears after a clean build, but then consistently reappears after subsequent editing or building, it should be reported as a potential bug.
- Are there errors on any Statix files? The behavior of running Statix on your language files is undefined when the specification itself has errors. Check the *Package Explorer* as well as the *Problems* and *Console* views to make sure none of the Statix files have errors on them. Fix such errors before debugging any other issues!
- Check for errors in the *Package Explorer* and in open editors, as well as in the *Console*, *Problems*, and *Error Log* views.
- See whether the section on *Some Common Problems* or the remainder of the documentation answers your question already.

11.2.2 Checking AST Traversal

Ensure that your Statix rules are applied to the AST nodes. It is easy to forget to apply a predicate to a subterm, especially for bigger languages. If you are not sure if the rules are applied to a certain AST node, add a forced note (e.g. `try { false } | note "text"`) to that AST node as follows:

```
extendsOk(s_lex, Extends(m), s_mod) :- {s_mod'}
  try { false } | note "extendsOK applied",
  resolveMod(s_lex, m) == s_mod',
  s_mod -EXT-> s_mod'.
```

Build your project and check if the note appears where you expect it. If it does not appear, find the places where those AST nodes may appear and ensure the predicate is applied.

11.2.3 Checking Reference Resolution

Setting `ref` attributes on references allows you to check reference resolution interactively in example programs of your language. The following rule shows how to do that using `@x.ref := x'`:

```
resolveVar(s, x) = T :- {x'}
  query typeOfDecl
    filter P* and { x' :- x' == x }
    min $ < P and true
    in s |-> [(_ , (x', T))],
  @x.ref := x'.
```

This requires that `x` and `x'` are both names from the AST. Now write some example programs and check if references resolve to the definitions you expect, by `Ctrl + Click` / `Cmd + Right Click` on the reference.

Note that `statix/References` must be included in one of your ESV files for this to work. This is by default the case for generated projects that use Statix.

11.2.4 Interpreting Error Messages

Statix can be configured to contain a trace as part of the error messages for failing constraints, which can be a great help for debugging. Two parameters control message formatting. The first, `message-trace-length` controls whether a trace is included in the message, and how long it will be. A value of 0 means no trace (if no custom message is provided, the failing constraint itself is still shown), -1 means the full trace. The default is set to 0, as showing traces is mostly helpful for debugging when writing the spec and it slows down message formatting. The second, `message-term-depth` controls the depth up to which terms in messages are formatted. A value of -1 means full terms. The default is set to 3, which is usually enough to understand the terms involved, without choking up Eclipse or the console with large error messages. It is not recommended to set both settings to -1, because then every message will contain the full AST.

The configuration settings are part of the `metaborg.yaml` file of the project containing the language files (not the project containing the specification!), and look as follows:

```
runtime:
  statix:
    message-trace-length: 5 # default: 0, full trace: -1
    message-term-depth: 3 # -1 = max
```

A typical error message including a trace may look as follows:

```
[ (?q.unit-wld61-10, (?q.unit-x'-11, ?q.unit-T-5)) ] == []
> query filter ((Label("units/name-resolution/interface!EXT")) * Label("units/name-
→ resolution/default-impl!var")) and { (?x', _) :- ?x' == "q" } min irrefl trans anti-
→ sym { <edge>Label("units/name-resolution/default-impl!var") < <edge>Label("units/
→ name-resolution/interface!EXT"); } and { _, _ :- true } in #p.unit-s_mod_4-4 |-> [(?
→ q.unit-wld61-10, (?q.unit-x'-11, ?q.unit-T-5))]
```

(continues on next page)

(continued from previous page)

```
> units/name-resolution/interface!resolveVar(#q.unit-s_mod_2-4,
↪QDefRef(QModInModRef(ModRef("P"), "B"), "q"), ?q.unit-T-5)
> units/statics!typeOfExpr(#q.unit-s_mod_2-4, VarRef(QDefRef(QModInModRef(ModRef(...),
↪"B"), "q")), ?q.unit-T-5)
> units/statics!defOk(#q.unit-s_mod_2-4, VarDef("e", VarRef(QDefRef(QModInModRef(...,
↪...), "q"))), #q.unit-s_mod_2-4)
> ... trace truncated ...
```

As this looks daunting at first, we break it down. At the top is the constraint that failed; in this case an equality constraint. Below that are several lines prefixed with `>` that show where the constraint above it originated. We see that the equality originated from a `query`, which itself originated from one of the rules of `resolveVar`, which was applied in one of the rules of `typeOfExpr` etc. As these traces can get very long, they are truncated to five entries.

Now we explain some more details of what we can see here:

- Errors may contain unification variables of the form `?FILENAME-VARNAME-NUM` or `?VARNAME-NUM`. These are instantiations of the meta-variables in the specification. The variable name `VARNAME` corresponds to the name of the meta-variable that was instantiated, and can be helpful in reasoning about the origin of a unification variable. When the name corresponds to a functional predicate name, it is a return value from that predicate. The file name is the file that was being checked when the unification variable was created. Due to Statix's operation, this can sometimes be the project root instead of the actual file.
- Scope values are shown as `#FILENAME-VARNAME-NUM` or `#VARNAME-NUM`. (Rarely they appear in the exploded form `Scope("FILENAME", "VARNAME-NUM")`).
- Predicate names are prefixed with the name of the module they are defined in. For example, `defOk` is defined in `units/statics` and therefore appears as `units/statics!defOk` in the trace. Note that the predicate name is prefixed with the Statix module that *defines* the predicate. (The rules for the predicate may be defined in other modules.)
- The trace shows which predicates were applied, and to which arguments. It does not show which predicate rule was chosen! This can often be deduced from the line above it in the trace, but if unsure, use a forced note (see [Inspecting Variables](#)) to check your expectation.
- Error messages are fully instantiated with the *final* result. This means that variables that appear in error messages are free in the final result of this Statix execution. Therefore, we do *not* have to consider the order of execution or the moment when the error message was generated when interpreting error messages!

The section on [Some Common Problems](#) contains tips on how to deal with many error messages.

11.2.5 Inspecting Variables

Inspecting the values assigned to meta-variables can be very helpful to debug a specification. Variables cannot be automatically inspected, but we can show their values by forcing a note in the rule where the variable appears. The following rule shows how to do this for the intermediate type `T` of the assigned variable:

```
stmtOk(s, Assign(x, e)) :- {T U}
  T == resolveVar(s, x),
  try { false } | note $[assignee has type [T]],
  U == typeOfExp(s, e),
  subtype(U, T).
```


11.2.6 Inspecting the Scope Graph

Inspecting the scope graph that is constructed by Statix can be very helpful in debugging problems with scoping and name resolution queries. After type checking, view the scope graph of a file using the Spoofax > Statix > Show scope graph menu. Note that in multi-file mode, the scope graph is always the graph of the whole project. Therefore, creating a small example project with only a few files can be very helpful (see also [Creating Minimal Examples](#)).

Here is an example of such a scope graph:

```
scope graph
  #q.unit-s_mod_2-4 {
    relations {
      units/name-resolution/default-impl!var : ("e", UNIT())
    }
    edges {
      units/name-resolution/interface!LEX : #s_1-1
    }
  }
  #p.unit-s_mod_4-4 {
    relations {
      units/name-resolution/default-impl!var : ("b", UNIT())
    }
    edges {
      units/name-resolution/interface!LEX : #p.unit-s_mod_2-6
    }
  }
  #p.unit-s_mod_2-6 {
    relations {
      units/name-resolution/default-impl!mod : ("B", #p.unit-s_mod_4-4)
    }
    edges {
      units/name-resolution/interface!LEX : #s_1-1
    }
  }
  #s_1-1 {
    relations {
      units/name-resolution/default-impl!mod : ("E", #q.unit-s_mod_2-4)
                                          ("P", #p.unit-s_mod_2-6)
    }
  }
}
```

The scope graph is presented as a list of scopes, with the relation entries and outgoing edges from that scope. Remember that the names of the scopes match the names of the meta-variables in the specification! For example, #p.unit-s_mod_4-4 originated from a meta-variable s_mod. Paying attention to this is very helpful in figuring out the structure of the graph.

Some useful questions you can ask yourself when inspecting the scope graph for debugging:

- Does the graph have the structure I expect from the current example program? Are all the scopes that I expect there, and are all the scopes that are there expected? Do all scopes have the expected relations in them? Do they have the expected outgoing edges?
- When you are debugging a certain query, consider the scope in which the query starts, and execute the query in the given graph. Are the necessary edges present? Does the regular expression allow those edges to be traversed? Are you querying the correct relation, and is the filter predicate correct for the data you want to match?

When considering these questions, it can be helpful to use the ideas from [Inspecting Variables](#) to verify the scope a query is executed in, or to show the scope that is created for a definition, and match those with what you see in the

scope graph.

11.2.7 Creating Minimal Examples

Creating a minimal example is one of the most useful things you can do when debugging. It helps you to get to the core of the problem, but it also benefits all of the other techniques we have discussed so far. Having a smaller example makes it easier to inspect the scope graph, makes it easier to inspect variables as there are fewer, and reduced the number of error messages to review.

An example is a file, or set of files, in your language, where Statix does not behave as you expect. A minimal example is usually created by starting from a big example that exhibits the problem. Try to eliminate files and simplify the example program while keeping the unexpected behavior. The smaller the program and the fewer rules in your specification are used for this program, the easier it is to debug.

11.2.8 Testing Predicates

Sometimes creating a minimal example program in your language is not enough to fix a problem. In such cases writing Statix tests is a great way to test your definitions in even more detail. In a Statix test you can specify a constraint and evaluate it to see how it behaves. For example, if you suspect a bug in the definition of the `subtype` predicate, you could test it as follows:

```
// file: debug.stxtest
resolve {T}
  T == typeOfExp(Int("42")),
  subtype(T, LONG())
imports
  statics
```

The `.stxtest` file starts with `resolve` and a constraint, which can be anything that can appear in a rule body. After that, the test may specify `imports`, `signature` and `rules` sections like a regular Statix module. A test is executed using the Spoofox > Evaluate > Evaluate Test menu. Evaluation outputs a `.stxresult` file, which looks as follows:

```
substitution
  T |-> INT()

analysis
  scope graph

errors
  *   INT() == LONG()
    > statics!subtype(INT(), LONG())
    > ... trace truncated ...

warnings

notes
```

The test result shows the value of top-level variables from the `resolve` block (in this case `T`), the scope graph that was constructed (in this case empty), and any messages that were generated (in this case one error).

These tests are a great way to verify that the predicate definitions work as you expect. Apply your predicates to different arguments to check their behavior. Even more complicated mechanisms such as queries can be debugged this way. Simply construct a scope graph in the `resolve` block (using `new`, `edges`, and `declarations`), and execute your querying predicate on the scopes you have created. As a starting point, you can take the AST of your example

program (using the Spoofox > Syntax > Show parse AST menu), and use that as an argument to your top-level predicate.

Creating a *self-contained* Statix test is a good way to isolate a problem. Instead of importing all your definitions, copy the relevant definitions to the test (in a `rules` section), and try to create the smallest set of rules and predicate arguments that still exhibit the problem you are debugging. A self-contained test is also very helpful when asking others for help, as it is much easier to review and run than having to setup and build a complete language project.

11.2.9 Some Common Problems

- Predicates fail with `amb (. . .)` terms as arguments. These terms indicate parsing ambiguities, which should be fixed in the grammar (SDF3) files.
- Errors in your specification appear at incorrect places (e.g. sort or constructor declarations). In such cases, the declaration is referenced from an invalid position anywhere in your specification, but due to the non-deterministic order of constraint solving the error appears at the observed position. The best approach to solve these issues is to comment away all usages, until the error disappears. Then, in the last commented position, the declaration is used incorrectly.
- One or both of the `fileOk (. . .)` or `projectOk (. . .)` predicates fail immediately, for example with the error messages:

```
statics!fileOk(#s_1-1,Test([Prog("A.mod",Decls(...)),Prog("B.mod",Decls(...)),
↪Prog("C.mod",Decls(...))])) (no origin information)
statics!projectOk(#s_1-1) (no origin information)
```

In such cases, you have probably renamed the top-level file, or moved the declarations of these predicates to another file that is imported. Assuming the predicates are now defined in the module `statics/mylang` as follows:

```
// file: trans/statics/mylang.stx
module statics/mylang
imports statics/mylang/program

rules

  projectOk : scope
  projectOk(s) .

  fileOk : scope * Start
  fileOk(s, p) :- programOk(s, p) .
```

If this module is the top-level module of your specification, then you have to change the call to `stx-editor-analyze` in `trans/analysis.str` such that the first term argument (which specifies the module to use, by default `"statics"`) is the new module name (in this case `statics/mylang`).

On the otherhand, if you kept `statics` as the top-level module and have it import the module `statics/mylang`, then you have to change the call to `stx-editor-analyze` in `trans/analysis.str` such that the second and third term argument (which specify the predicates to apply to projects and files, respectively) are qualified by the module name (in this case `"statics/mylang!projectOk"` and `"statics/mylang!fileOk"`, respectively).

- Files of your language are only analyzed by Statix after they are opened in an editor in Eclipse. There are several reasons why this may be happening:
 - The project containing the files is not a Spoofox project. A spoofox project must contain a `metaborg.yaml`. If it is a Maven project, the `packaging` must be one of `spoofox-{language,test,project}`.

- The project containing the files does not have a dependency on your language. Spoofax only analyzes files of your language if the *metaborg.yaml* configuration contains a *compile* dependency on the language definition. This should look similar to the following:

```
dependencies:
  compile:
    - org.example:your-language:1.0-SNAPSHOT
```

- The language is missing. If a language dependency is missing, this is reported with errors on the console. Make sure your language definition project is open in Eclipse and that it is successfully built.
- Eclipse is not configured to automatically build files. This can be enabled by selecting *Project > Build automatically* from the Eclipse menu.
- The project in Eclipse did not get the Spoofax nature. Imported Maven projects with one of the *spoofax*-* packagings normally get the Spoofax nature automatically, but sometimes this doesn't work correctly. Non-Maven projects always have to be assigned the Spoofax nature manually. This can be done with *Spoofax > Add Spoofax nature* in the context-menu of the project containing the files.
- A lot of errors are reported. It happens that a single problem in the type checked program leads to the failure of other constraints (cascading errors). For example, an unresolved name might lead to errors about subtype checks that cannot be solved, import edges that cannot be created, etc. Here are some tips to help you find the root cause of the problem:

- Differentiate between failed and unsolved constraints. The cause of a problem is usually found best by looking at the failed constraints. For example, an unresolved name might result in an error on the equality constraint between the expected and actual query result. Errors on unsolved constraints are marked as *_Unsolved_*. Unsolved errors are often the result of uninstantiated logical variables.

Predicates remain unsolved if the uninstantiated variable prevents the selection of an applicable rule for the predicate. For example, an unsolved error `subtype(INT(), ?T-1)` is caused by the free variable `?T-1` which prevents selecting the appropriate rule of the `subtype` predicate.

Queries remain unsolved if the query scope is not instantiated, or if variables used in the matching predicate (such as the name to resolve) remained free. For example, an unsolved error `query filter (e Label("typeOfDecl")) and { (?x',_) :- ?x' == ?x-5 } min irrefl trans anti-sym { <edge>Label("typeOfDecl") < <edge>Label("P"); } and { _, _ :- true } in ?s-3 |-> [(?wld0-1, (?x'-2, ?T-4))]` cannot be resolved because the scope variable `?s-3` is free, and the free variable `?x-5` would prevent matching declarations. Use of the variables `?x'-2` and `?T-4` might cause more unsolved constraints, since these also remain free when the query cannot be solved.

Edge and declaration assertions remain unsolved if the scopes are not instantiated. For example, the edge assertion `#s_2-1 -Label("P")-> ?s'-5` cannot be solved because the variable for the target scope `?s'-5` is not instantiated. Unsolved edge constraints in particular can lead to lots of cascading errors, as they block all queries going through the source scope of the edge.

- If it is not immediately clear which error is the root of a problem, it helps to figure out the free variable dependencies between reported errors. Consider the following small example of three reported errors:

```
subtype(?T-5, LONG)
#s_3-1 -Label("P")-> ?s'-6
query filter ((Label("P")) * Label("typeOfDecl")) and { (?x',_) :- ?x' == "v" }
→ min irrefl trans anti-sym { <edge>Label("typeOfDecl") < <edge>Label("P"); }
→ and { _, _ :- true } in #s_3-1 |-> [(?wld4-1, (?x'-2, ?T-5))]
```

For each of these we can see which variables are necessary for the constraint to be solved, and which they might instantiate when solved. The `subtype` predicate is blocked on the variable `?T-5`. The edge assertion is blocked on the scope variable `?s'-6`. The query does not seem blocked on a variable (both

the scope and the filter predicate are instantiated), but would instantiate the variables `?x'-2` and `?T-5` when solved.

We can conclude that the `subtype` constraint depends on solving the query, so we focus our attention on the query. Now we realize that we query in the scope of the unsolved edge assertion. So, the query depends on the edge assertion, and our task is to figure out why the scope variable in the edge target is not instantiated.

11.2.10 Getting Help and Reporting Issues

If the techniques above did not help to solve your problem, you can ask us for help or report the issue you found. To make this process as smooth as possible, we ask you to follow the following template when asking a Statix related question:

1. Single sentence description of the issue.
2. Spoofax version. See *About Eclipse; Installation Details; Features*, and search for *Spoofax*.
3. Statix configuration: single-file or multi-file mode. Multi-file mode is enabled when the `observer` setting in your ESV looks like `observer: editor-analyze (constraint) (multifile)`.
4. Steps to reproduce. Best is to include a small, self-contained test (see *Testing Predicates* above) so that others can easily run the test and reproduce the issue! If that is not possible, provide a (link to) a project, including an example file, that shows the problem. Keep the project and the example as small as possible, and be specific about the relevant parts of your program and of your specification.
5. Description of the observed behavior. Also mention if the problem occurs consistently, or only sometimes? If only sometimes, does it occur always/never after a clean build, or does it occur always/never after editing and/or building without cleaning?
6. Description of the expected behavior.
7. Extra information that you think is relevant to the problem. For example, things you have tried already, pointers to the part of the rules you think are relevant to the problem etc. If you tried other examples that show some light on the issue, this is a good place to put those. Again, it is best if these also come as self-contained tests!

Example. An example bug report described using the format above:

```
Issue:
Spoofax version: 2.6.0.20210208-173259-master
Statix setup: multi-file

Steps to reproduce:
Execute the test in ``example1.stxtest``.

Observed behavior:
Sometimes an error is reported that the ``query`` failed.
The problem does not occur consistently. On some runs, the error appears, but not on
↳ others. This
does not seem related to cleaning or building the project.

Expected behavior:
The test is executed and no errors are reported. Scope ``s1`` is reachable from
↳ ``s2``, so the
query return a single result, and ``ps != []`` should therefore hold.

Extra information:
The test in ``example2.stxtest`` is very similar. The only difference is that the
↳ predicate
```

(continues on next page)

(continued from previous page)

``nonempty`` has an extra rule for the singleton list. The predicate is semantically ↵
 ↵the same, as
 the extra rule fails, just as the general rule would do on the singleton list. ↵
 ↵However, this
 example never gives the unexpected error.

The bug report is accompanied by two self-contained tests. One illustrates the problem, while the other shows a very similar variant that does not exhibit the problem.

```
// example1.stxttest
resolve {s1 s2}
  new s1, new s2, s2 -I-> s1,
  reachable(s1, s2)

signature
  name-resolution
  labels
  I

rules

  reachable : scope * scope
  reachable(s1, s2) :- {ps}
    query () filter I*
      and { s1' :- s1' == s1 }
      min and true
      in s2 |-> ps,
    nonempty(ps).

  nonempty : list((path * scope))
  nonempty(ps) :- ps != [].
```

```
// example2.stxttest
resolve {s1 s2}
  new s1, new s2, s2 -I-> s1,
  reachable(s1, s2)

signature
  name-resolution
  labels
  I

rules

  reachable : scope * scope
  reachable(s1, s2) :- {ps}
    query () filter I*
      and { s1' :- s1' == s1 }
      min and true
      in s2 |-> ps,
    nonempty(ps).

  nonempty : list((path * scope))
  nonempty(ps) :- ps != [].
  nonempty([_]) :- false.
```

11.3 Language Reference

This section gives a systematic overview of the Statix language. Statix specifications are organized in modules, consisting of signatures and predicate rules.

Warning: This section is currently incomplete. The information that is there is up-to-date, but many constructs are not yet documented.

11.3.1 Lexical matters

Identifiers

```
id      = [A-Za-z][a-zA-Z0-9\_]*
lc-id   = [a-z][a-zA-Z0-9\_]*
uc-id   = [A-Z][a-zA-Z0-9\_]*
```

Most identifiers in Statix fall into one of the following categories:

- *Identifiers*, that start with a character, and must match the regular expression `[A-Za-z][a-zA-Z0-9_]*`.
- *Lowercase identifiers*, that start with a lowercase character, and must match the regular expression `[a-z][a-zA-Z0-9_]*`.
- *Uppercase identifiers*, that start with an uppercase character, and must match the regular expression `[A-Z][a-zA-Z0-9_]*`.

Comments

Comments in Statix follow the C-style:

- `// ... single line ...` for single-line comments
- `/* ... multiple lines ... */` for multi-line comments

Multi-line comments can be nested, and run until the end of the file when the closing `*/` is omitted.

11.3.2 Terms

```
term = uc-id "(" {term ","}* ")"
      | "(" {term ","}* ")"
      | "[" {term ","}* "]"
      | "[" {term ","}* "]" term "]"
      | numeral
      | string
      | id
      | "_"

numeral = "-"? [1-9][0-9]*

string = "\"" ([^"\\]|"\\"[nrt\\])* "\""
```

11.3.3 Modules and Tests

Statix specifications are organized in named modules and test files. Modules and test files can import named modules to extend and use the predicates from the imported modules.

Modules

```
module module-id

    section*
```

Statix specifications are organized in modules. A module is identified by a module identifier. Module identifiers consist of one or more names separated by slashes, as in {name "/" }+. The names must match the regular expression `[a-zA-Z0-9_][a-zA-Z0-9_\.\\-]*`.

Every module is defined in its own file, with the extensions `.stx`. The module name and the last components of the file path must coincide.

Example. An empty module `analysis/main`, defined in a file `.../analysis/main.stx`.

```
module analysis/main

// work on this
```

Modules consist of sections for imports, signatures, and rule definitions. The rest of this section describes imports, and subsequent sections deal with signatures and rules.

Imports

```
imports

    module-ref*
```

A module can import definitions from other modules by importing the other module. Imports are specified in an `imports` section, which lists the modules being imported.

Imports make predicates defined in the imported module visible. The importing module can use the imported predicates, and extend the predicates with new rules. Imports are not transitive, and locally defined elements (e.g., sorts or predicates) shadow imported elements of the same kind and the same name.

Example. A main module importing several submodules.

```
module main

imports

    signatures/MyLanguage-sig

    types
```

Tests


```
resolve constraint

section*
```

Apart from named modules, stand-alone test can be defined in `.stxttest` files. All sections that are allowed in named modules are allowed in tests as well. This means tests can have signatures, rules, and import named modules.

Example. A test using the predicate `concat` imported from a named module.

```
resolve {xs} concat([1,2,3], [4,5,6], xs)

imports

  lists
```

Statix tests can be executed in Eclipse with the Spoofax > Evaluate > Evaluate Test menu action. The test output contains the values of top-level variables in the test constraint (i.e., `xs` in this example), as well as any errors from failed constraints.

11.3.4 Signatures

```
signatures

  signature*
```

Terms

Sorts

```
signature = ...
  | "sorts" sort-decl*

sort-decl = uc-id
  | uc-id "=" sort

sort = "string"
  | "int"
  | "list" "(" sort ")"
  | "(" {sort "*" }* ")"
  | "scope"
  | "occurrence"
  | "path"
  | "label"
  | "astId"
  | uc-id
```

Statix uses algebraic data types to validate term well-formedness. First, Statix has several built-in scalar data types, such as `int`, `string` and `scope`. In addition, Statix also has two built-in composite data types: `tuples` and `lists`. Next to the built-in sorts, custom syntactic categories can be defined by adding a name to a `sorts` subsection of the `signatures` section.

Example. Declaration of a custom sort `Exp` and a sort alias for identifiers.

```
signature
  sorts
    Exp
    ID = string
```

Constructors

```
signature = ...
  | "constructors" cons-decl*

cons-decl = uc-id ":" uc-id
  | uc-id ":" {sort "*" }* "->" uc-id
```

In order to construct or match actual data terms, constructors for these terms need to be declared. Constructors without arguments are declared by stating the constructor name and its sort, separated by a colon. For constructors with arguments, the argument sorts are separated by an asterisk, followed by an arrow operator and the target sort.

Example. Declaration of various constructors for the `Exp` sort.

```
signature
  sorts
    ID = string
    Exp

  constructors
    True : Exp
    Var  : ID -> Exp
    Plus : Exp * Exp -> Exp
```

The example above states three constructors for the `Exp` sort. The `True` constructor has no arguments, the `Var` constructor has a single name as argument, while the `Plus` constructor takes two subexpressions as arguments.

Name binding

Relations

```
signature = ...
  | "relations" rel-decl*

rel-decl = uc-id ":" {sort "*" }*
  | uc-id ":" {sort "*" }* "->" sort
```

In Statix, relations associate data with a scope. All used relations and the type of their data must be declared in the `relations` section. Each relation declaration consist of the name of the relation and the arguments it accepts.

Relation declarations come in two flavors. There is a *predicative* variant and a *functional* variant. The functional variant has the last two arguments separated with a `->`, which indicates that the relation is intended to map the first terms to the last term (as in the regular notion of functions).

Apart from intended semantics, the difference between the variants has to do with the formulation of the predicates of queries. Please refer to the [Queries](#) section for more information on querying relations. Otherwise, both ways of declaring relations are equivalent. In fact, during compilation, the functional variant is normalized to the predicate variant.

Example. Declaration of a predicative relation `this` and a functional relation `var`.

```
signature
  sorts
    TYPE
    ID = string

  relations
    this : TYPE
    var  : ID -> TYPE
```

Namespaces

Warning: Usage of namespaces is strongly discouraged and will be removed or revised in a future version of Statix.

Name resolution

Warning: Usage of namespaces is strongly discouraged and will be removed or revised in a future version of Statix.

11.3.5 Predicates and Rules

```
rules

  rule-def*
```

Predicates and their rules make up the main part of a Statix specification.

Predicate rules

```
lc-id : {sort " * "}*

rule-name? lc-id(<{term " , "}*>).
rule-name? lc-id(<{term " , "}*>) :- constraint.
```

```
rule-name = "[" id "]"
```

Predicates are defined with the sorts of their arguments. Rules define the meaning of the predicate for different cases of the arguments. Rule patterns can be non-linear, i.e., variables can appear multiple times, in which case the terms in those positions must be equal.

Committed choice rule selection

Statix has a committed-choice semantics. This means that once a rule is selected, the solver does never backtrack on that choice. That is different from logic languages like Prolog, where rules are optimistically selected and the solver backtracks when the rule does not work out.

Committed choice evaluation has consequences for inference during constraint solving. If a predicate has multiple rules, a rule is only selected once the constraint arguments are sufficiently instantiated.

Rule order

The order in which the rules of a predicate apply is determined by the patterns it matches on, not by the order in which the rules appear in the specification. Most specific rules apply before more general rules. The parameter patterns are considered from left to right when determining this order. It is an error to have rules with overlapping patterns, where neither is more general than the other. These rules are marked with an error.

Example. An `or` predicate that computes a logical or, with its last argument the result.

```
or : Bool * Bool * Bool

or(True(), _, b) :- b == True().
or(_, True(), b) :- b == True().
or(_, _, b) :- b == False().
```

In the example above, the rules are considered in the order they are presented above. Beware that changing the rule order would not change the specifications behaviour. The last rule is the most general, and therefore comes last, as it matches any arguments. The first rule is more specific than the second because of the left-to-right nature of the ordering.

Non-linear patterns

Non-linear patterns are patterns in which at least one pattern variable occurs multiple times. Such patterns only match on terms that have equal subterms at the positions where such a variable occurs.

Example. An `xor` predicate that computes a logical exclusive or, with its last argument the result.

```
xor: Bool * Bool * Bool

xor(B, B, b) :- b == False().
xor(_, _, b) :- b == True().
```

In the example above, the first rule for `xor` has a non-linear pattern, because the variable `B` occurs both at the first and at the second position. In this way, the first rule only matches on equal input terms (either `True(), True(), b` or `False(), False(), b`).

Regarding the ordering of rules by specificity, it holds that an occurrence of a variable that is seen earlier is regarded as more specific than a free variable. Therefore, the first rule of `xor` takes precedence over the second rule. Bound variables are as specific as concrete constructors.

Ordering rules with non-linear patterns

Careful attention to rule order needs to be paid when non-linear patterns and concrete constructor patterns are mixed. For example, consider a `subtype` predicate with rules for record types and equal types:

```
subtype: TYPE * TYPE

subtype(REC(s_rec1), REC(s_rec2)) :- /* omitted */.
subtype(T, T).
```

In this example, equal record types match on both rules. Because of the left to right nature of the rule application, the first rule will be chosen, because for the first argument, the `REC` constructor is regarded as more specific than the (at that position free) `T` variable.

If that behavior is not desired, an explicit rule for the intersection of the domains of the pair of rules in question needs to be added. This rule is more specific than both of the other rules, and is therefore selected for any matching input. For example, consider this augmented `subtype` predicate with an additional rule for equal record types:

```
subtype: TYPE * TYPE

subtype(REC(s_rec), REC(s_rec)).
subtype(REC(s_rec1), REC(s_rec2)) :- /* omitted */.
subtype(T, T).
```

In this example, we added a rule that declares that a record type is a subtype of itself. This rule ensures that equal record types are regarded as subtypes without verifying additional constraints. So, while it seems that the first and the third rules are equivalent, and the first one superfluous, this is not the case because the rule ordering will choose the second rule when the behavior of the third rule is desired.

Functional rules

```
lc-id : {sort " * "}* -> sort

rule-name? lc-id(<{term " , "}*>) = term.
rule-name? lc-id(<{term " , "}*>) = term :- constraint.
```

Predicates can be defined in functional style as well. Functional predicates can be understood in terms of regular predicates. For example, the `or` predicate can be written in functional style as follows:

```
or : Bool * Bool -> Bool

or(True(), _) = True().
or(_, True()) = True().
or(_, _) = False().
```

This form is equivalent to the definition given above, however its use in the specification is slightly different. Function predicates are used in term positions, where they behave as a term of the output type.

Example. Rule for a functional predicate to type check expressions. The functional predicate `typeOfExp` is used in two term positions: as the result of a functional rule, and in an equality constraint.

```
typeOfExp : scope * Exp -> TYPE

typeOfExp(s, Int()) = INT().

typeOfExp(s, IfThen(c, e)) = typeOfExp(s, e) :-
    typeOfExp(s, c) == BOOL().
```

Every specification with functional predicates is normalized to a form with only regular predicates. To show the normal form of a specification in Eclipse, use the Spoofax > Syntax > Format normalized AST menu action.

Mapping rules

11.3.6 Constraints

Base constraints

Term equality

Name binding

Scope graph

Queries

Occurrences

Arithmetic

11.4 Stratego API

The Stratego API to Statix ...

Warning: This section has not yet been written.

11.5 Signature Generator

It is quite cumbersome to write Statix signatures. Thankfully, the `sdf3.ext.statix` project can generate these signatures for you.

11.5.1 Well-Formed SDF3 Requirements

For the generator to work correctly, your SDF3 must be well formed. In particular, you must:

- explicitly declare each sort *exactly once* in your project
- declare lexical sorts in a `lexical sorts` block
- declare context-free sorts in a `context-free sorts` block
- for every use of a sort: either have a local declaration of a sort, or an import of a file that declares the sort
- not declare sorts that are not used in any rules
- not use any implicitly declared sorts
- not use complex injections, such as `Pair = Expr Expr`. However, list injections without terminal syntax, such as `List = Elem*`, are allowed.
- constructors must start with an upper-case letter
- not use `sdf2table: c`

The generator generates strategies and signatures for each explicit declaration of a sort in SDF3, which is why each sort must be declared exactly once. SDF3 does not generate Stratego signatures for placeholders for sorts that have no corresponding rules, causing errors in the generated Statix injection explication strategies. Complex injections are not supported across Spoofax. Optional sorts cannot be represented in Statix.

11.5.2 Applying the Generator in Spoofax 2

In your language project's `metaborg.yaml` file, change your compile dependencies to include `org.metaborg:sdf3.ext.statix`. For example:

```
dependencies:
  compile:
    - org.metaborg:org.metaborg.meta.lang.esv:${metaborgVersion}
    - org.metaborg:org.metaborg.meta.lang.template:${metaborgVersion}
    - org.metaborg:sdf3.ext.statix:${metaborgVersion}
```

Note: Clean the project and restart Eclipse when changing the `metaborg.yaml` file.

Once you clean your project, the extension automatically generates the following:

- Statix signatures declarations (in `src-gen/statix/signatures/`)
- Stratego strategies for explicating and removing injections (in `src-gen/injections/`)

11.5.3 Using the Generated Injection strategies

The generator generates strategies for explicating and removing injections. This is unfortunately needed since Statix does not support injections directly. To use these strategies, import `injections/-` and call the `explicate-injections-MyLang-Start` and `implicate-injections-MyLang-Start` strategies for the analysis pre-processing and post-processing respectively, where `MyLang` is the name of your language and `Start` is your language's start symbol (as specified in `Syntax.esv`). For example, in `trans/analysis.str`:

```
module analysis

imports

  libspoofax/sdf/pp

  statixruntime
  statix/api

  injections/-
  libspoofax/term/origin

rules

  editor-analyze = stx-editor-analyze(pre-analyze, post-analyze|"static-semantics",
  ↪"programOk")
  pre-analyze   = origin-track-forced(explicate-injections-MyLang-Start)
  post-analyze  = origin-track-forced(implicate-injections-MyLang-Start)
```

11.5.4 Using the Generated Signatures

Using the generated Statix signatures is quite simple: just import them into your Statix specification. Each SDF3 file gets an associated Statix file with the signatures. For example, if your syntax is defined across two files named `MyLang.sdf3` and `Common.sdf3`, then in Statix you should add the following imports:

```
imports
  signatures/MyLang-sig
  signatures/Common-sig
```

Because Statix does not support injections, you have to use explicit constructor names for injections. For example, the following SDF3 syntax:

```
context-free sorts
  Stmt VarName

lexical sorts
  ID

context-free syntax
  Stmt.VarDecl = <var <VarName>;>
  VarName.Wildcard = <_>
  VarName = ID

lexical syntax
  ID = [a-zA-Z] [a-zA-Z0-9\_]*

lexical restrictions
  ID -/- [a-zA-Z0-9\_]
```

would approximately produce the following signatures:

```
module signatures/Test-sig

imports

signature
  sorts
    Stmt
    VarName
    ID = string
  constructors
    Stmt-Plhdr : Stmt
    VarName-Plhdr : VarName

signature
  constructors
    VarDecl : VarName -> Stmt
    Wildcard : VarName
    ID2VarName : ID -> VarName
```

Now, in Statix if you just want to capture the term of sort `VarName` in the `VarDecl` constructor, this would suffice:

```
VarDecl (x)
```

But if you want to match the term only if it has the sort `ID`, then you have to use the explicit injection constructor name `ID2VarName`:

```
VarDecl (ID2VarName (x))
```

In this example, `ID` is a lexical sort, so it is an alias for `string` in the Statix specification.

11.5.5 Troubleshooting

Calls non-existing

Build fails with errors such as this:


```
[ strj | error ] *** ("is-MyLang-MySort-or-inj",0,0) calls non-existing ("is-MyLang-
↪ID-or-inj",0,0)
[ strj | error ] *** ("explicate-injections-MyLang-MySort",0,0) calls non-existing (
↪"explicate-injections-MyLang-ID",0,0)
[ strj | error ] *** ("implicate-injections-MyLang-MySort",0,0) calls non-existing (
↪"implicate-injections-MyLang-ID",0,0)
Executing strj failed: {}
Failing builder was required by "Generate sources".
BUILD FAILED
```

To solve this, ensure you have declared ID (in this example) as a lexical sort in your syntax, and make sure that the syntax file with rules for MySort that reference ID import the syntax file that declares ID.

Transformation failed unexpectedly

Clean or build fails with an error such as this:

```
ERROR: Optional sorts are not supported by Statix: Opt(Sort("MySort"))
Transformation failed unexpectedly for eclipse:///mylang/syntax/mysyntax.sdf3
org.metaborg.core.transform.TransformException: Invoking Stratego strategy generate-
↪statix failed at term:
  CfSignature("MySort", Some("MyCons"), [ Param(Opt(Sort("MySort")), "mySort") ])
Stratego trace:
  generate_statix_0_0
  generate_statix_abstract_0_0
  geninj_generate_statix_0_0
  geninj_module_to_sig_0_0
  with_1_1
  flatfilter_1_0
  filter_1_0
  with_1_1 <==
  map_1_0
  geninj_symbol_to_stxsig_0_0
Internal error: 'with' clause failed unexpectedly in 'geninj-sig-to-stxsig'
```

Note the first line with ERROR, it tells you that something is not supported. In this case, the use of optional sorts such as MySort? is not supported by Statix and the Statix signature generator.

To solve this, rewrite a syntax rule with an optional sort such as:

```
Stmt.VarDecl    = <<Type?> <ID> = <Exp>>
```

Into a rule with an explicit sort:

```
Stmt.VarDecl    = <<Type-OPT> <ID> = <Exp>>
Type-OPT.NoType = <>
Type-OPT        = Type
```

Note that the -OPT suffix has no special meaning. You can name the sort differently, such as OptionalType.

Constructor MySort-Plhdr/0 not declared

Build fails with an error such as this:

```
[ strj | error ] in rule explicate-injections-MyLang-MySort(0|0): constructor MySort-
↳ Plhdr/0 not declared
-   MySort-Plhdr()
Executing strj failed: {}
BUILD FAILED
```

You have declared a sort for which you don't have any syntax rules. Remove the sort from the `context-free` `sorts` or `sorts` block.

No pp entry found, cannot rewrite to box

Clean fails with an error such as this:

```
[ identity crisis | error ] No pp entry found for: (1,["declSortLex"])
- [ identity crisis | error ] Cannot rewrite to box:
-   declSortLex("MySort")
```

You are using the old `sdf2table`: `c`. Change this in `metaborg.yaml` into `sdf2table`: `java`.

SPT analysis tests calling Stratego strategies fail

An SPT test can run an arbitrary Stratego strategy on an analyzed AST and compare the results with the expected AST. If the origin of the is not tracked properly, the root constructor of the resulting analyzed AST will be missing and the comparison will fail.

To fix this, ensure the `pre-analyze` and `post-analyze` strategies in `analysis.str` call `origin-track-forced`:

```
imports libspoofox/term/origin

rules
  pre-analyze = origin-track-forced(explicate-injections-MyLang-Start)
  post-analyze = origin-track-forced(implicate-injections-MyLang-Start)
```

11.6 Rename Refactoring

Spoofax provides an automated Rename refactoring as an editor service for every language developed with it that has the static semantics defined with Statix or NaBL2. The renaming algorithm is implemented as a Stratego strategy and can be imported from the `statixruntime` library. When called, the strategy needs to be parameterized with the layout-preserving pretty-printing strategy `construct-textual-change`, the `editor-analyze` strategy and an indicator strategy for multi-file mode.

When creating a new Spoofax language project, such a strategy is automatically generated and placed in the `analysis` module. But it can also easily be added to existing projects, for example with a module like this:

```
module renaming

imports
  statixruntime
  statix/runtime/renaming

pp
```

(continues on next page)

(continued from previous page)

```
analysis

rules
  rename-menu-action = rename-action (construct-textual-change,
                                       editor-analyze, id)
```

The renaming is triggered from an entry in the Spoofax menu. For new projects this is automatically created. To add it to an existing project a menu like the following can be implemented in ESV:

```
module Refactoring

menus
  menu: "Refactoring"

  action: "Rename" = rename-menu-action
```

For the renaming to work correctly in all cases, terms that represent a declaration of a program entity, such as a function or a variable, need to set the `decl` property on the name of the entity. This is an example when declaring a type:

```
declareType(scope, name, T) :-
  scope -> Type{name} with typeOfDecl T,
  @name.decl := name,
  typeOfDecl of Type{name} in scope |-> [(_, (_, T))].
```

11.6.1 Renaming in NaBL2

There also exists a version of the Rename refactoring that works with languages using NaBL2. It can be added with a Stratego module like this:

```
module renaming

imports
  nabl2/runtime

  pp
  analysis

rules
  rename-menu-action = nabl2-rename-action (construct-textual-change,
                                             editor-analyze, id)
```

11.7 NaBL2 Migration Guide

11.7.1 Terms

All sorts and constructors must be explicitly defined in Statix in `sorts` and `constructors` signatures. Sorts in Statix are mostly similar to terms in NaBL2. Notable differences:

- There is no catch-all sort `term` in Statix.
- There are not sort variables in Statix.
- List sorts in Statix are written as `list (X)` for some sort `X`.

Statix signatures for language syntax can be generated from SDF3 definitions with the *signature generator*.

Signature

11.7.2 Name Resolution

Name resolution in NaBL2 heavily relies on occurrences and their unique identity. In Statix, the notion of a stand-alone reference is replaced by the notion of a query. Therefore, the use of occurrences is now discouraged in favour of regular terms, relations, and and predicates for the different namespaces.

```
signature

  namespaces
    Var

  name resolution
    labels P
    well-formedness P*
    order D < P

  rules

    [[ Def(x, T) ^ (s) ]] :=
      Var{x} <- s,
      Var{x} : T.

    [[ Var(x) ^ (s) : T ]] :=
      Var{x} -> s,
      Var{x} |-> d,
      d : T.
```

```
signature

  relations
    var : string * TYPE

  name-resolution
    labels P

  rules

    declareVar : scope * string * TYPE

    declareVar(s, x, T) :-
      !var[x, T] in s.

    resolveVar : scope * string -> TYPE

    resolveVar(s, x) = T :-
      {x'}
      query var
        filter P* and { x' :- x' == x }
        min $ < P and true
        in s |-> [(_ , (x, T))],
      @x.ref := x'.
```

(continues on next page)

(continued from previous page)

```

rules

stmtOk : scope * Stmt

stmtOk(s, Def(x, T)) :-
  declareVar(s, x, T);

typeOfExp : scope * Exp -> TYPE

typeOfExp(s, Var(x)) = T :-
  T == resolveVar(s, x).

```

Things to note:

- Each namespace gets its own relation, and set of predicates to declare and resolve in that namespace (declareXXX and resolveXXX).
- The regular expression and order on labels is not global anymore, but part of the query in the resolveXXX rules.
- If a declaration should have a type associated with it, it is now part of the relation. The fact that it appears after the arrow \rightarrow indicates that each declaration has a single type. As a result, declareXXX combines the constraints $\text{XXX}\{\dots\} \leftarrow s, \text{XXX}\{\dots\} : T$. Similarly, resolveXXX combines the constraints $\text{XXX}\{\dots\} \rightarrow s, \text{XXX}\{\dots\} \mid \rightarrow d, d : T$.
- The end-of-path label, called D in NaBL2, now has a special symbol \$, instead of the reserved name.

11.7.3 Functions

NaBL2 functions can be translated to Statix predicates in a straight-forward manner. Note that if the function was used overloaded, it is necessary to defined different predicates for the different argument types.

```

signature

functions

plusType : (Type * Type) -> Type {
  (IntTy(), IntTy()) -> IntTy(),
  (StrTy(), _) -> StrTy(),
  (ListTy(a), a) -> ListTy(a),
  (ListTy(a), ListTy(a)) -> ListTy(a)
}

```

```

plusType : Type * Type -> Type

plusType(IntTy(), IntTy()) = IntTy().
plusType(StrTy(), _) = StrTy().
plusType(ListTy(a), a) = ListTy(a).
plusType(ListTy(a), ListTy(a)) = ListTy(a).

```

11.7.4 Relations

Relations as they exist in NaBL2 are not supported in Statix.

An example of a subtyping relation in NaBL2 would translate as follows:

```
signature

relations
  reflexive, transitive, anti-symmetric sub : Type * Type {
    FunT(-sub, +sub),
    ListT(+sub)
  }

rules

[[ Class(x, superX, _) ^ (s) ]] :=
  ... more constraints ...,
  ClassT(x) <sub! ClassT(superX).

[[ Def(x, T, e) ^ (s) ]] :=
  [[ e ^ (s) : T' ]],
  T1 <sub? T2.
```

```
rules

subType : TYPE * TYPE

subType(FunT(T1, T2), FunT(U1, U2)) :-
  subType(U1, T1),
  subType(T2, T1).

subType(ListT(T), ListT(U)) :-
  subType(T, U).

subType(ClassT(s1), ClassT(s2)) :-
  ... check connectivity of s1 and s2 in the scope graph ...
```

In this case implementing the `subType` rule for `ClassT` requires changing the encoding of class types. Instead of using names, we use the class scope to identify the class type. This pattern is known as `_Scopes as Types_`. Subtyping between class scopes can be checked by checking if one scope is reachable from the other.

Rules

NaBL2 constraint generation rules must be translated to Statix predicates and corresponding rules. Predicates in Statix are explicitly typed, and a predicate has to be defined for each sort for which constraint generation rules are defined.

Here are some example rules for expressions in NaBL2:

```
[[ Let(binds, body) ^ (s) : T ]] :=
  new s_rec, s_rec -P-> s,
  Map1[[ binds ^ (s) ]],
  [[ body ^ (s) : T ]].

[[ Bind(x, e) ^ (s, s_let) ]] :-
  [[ e ^ (s) : T ]],
  Var{x} <- s_let,
  Var{x} : T.
```

In Statix these would be encoded as:

```

typeOfExp : scope * Exp -> TYPE

typeOfExp(s, e@Let(binds, body)) = T :-
{s_rec}
  new s_rec, s_rec -P-> s,
  bindsOk(s, binds, s_let),
  T == typeOfExp(s_rec, body),
  @e.type := T.

bindOk : scope * Bind * scope
bindsOk maps bindOk(*, list(*))

bindOk(s, Bind(x, e), s_let) :-
  declareVar(x, typeOfExp(s, e), s_let).

```

11.8 Migrating to the Concurrent Solver

11.8.1 Enabling the Concurrent Solver

In order to enable the concurrent solver, either one of the following approaches can be taken.

For a Language

To enable the concurrent solver for a language, set the `language.statix.concurrent` property in the `metaborg.yaml` file to `true`. This ensures that the concurrent solver is used for all sources in the language.

Example

```

id: org.example:mylang:0.1.0-SNAPSHOT
name: mylang
language:
  statix:
    concurrent: true

```

For an Example Project

To enable the concurrent solver for a particular project only, set the `runtime.statix.concurrent` property in the `metaborg.yaml` file to a list that contains all names of the languages for which you want to use the concurrent solver. The name of the language should correspond to the `name` property in the `metaborg.yaml` of the language definition project.

Example

```

id: org.example:mylang.example:0.1.0-SNAPSHOT
runtime:
  statix:
    concurrent:
      - mylang

```

Warning: Please be aware that changes in the `metaborg.yaml` file may require a restart of Eclipse.

Note: The concurrent solver can only be used in `multifile` mode.

11.8.2 Indirect Type Declaration

Type checking with the concurrent solver might result in deadlock when type-checkers have mutual dependencies on their declarations. This problem can be solved by adding an intermediate declaration that splits the part of the declaration data that is filtered on (usually the declaration *name*), and the part that is processed further by the querying unit (usually the *type*). This pattern is best explained with an example.

Example. Suppose you have the following specification to model type declarations.

```
signature
  relations
    type : ID -> TYPE

rules
  declareType : scope * ID * TYPE
  resolveType : scope * ID -> TYPE

  declareType(s, x, T) :-
    !type[x, T] in s.

  resolveType(s, x) = T :-
    query type
      filter P* I* and { x' :- x' == x }
      in s |-> [(_, (x, T))].
```

This specification needs to be changed in the following:

```
signature
  relations
    type : ID -> scope
    typeOf : TYPE

rules
  declareType : scope * ID * TYPE
  resolveType : scope * ID -> TYPE

  declareType(s, x, T) :-
    !type[x, withType(T)] in s.

  resolveType(s, x) = typeOf(T) :-
    query type
      filter P* I* and { x' :- x' == x }
      in s |-> [(_, (x, T))].

rules
  withType : TYPE -> scope
  typeOf : scope -> TYPE

  withType(T) = s :-
```

(continues on next page)

(continued from previous page)

```
new s, !typeOf[T] in s.

typeOf(s) = T :-
  query typeOf filter e in s |-> [(_ , T)].
```

We now discuss the changes one-by-one. First, the signature of relation `type` is be changed to `ID -> scope`. In this scope, we store the type using the newly introduced `typeOf` relation. This relation only carries a single `TYPE` term. In this way, the original term is still indirectly present in the outer declaration.

The `withType` and `typeOf` rules allow to convert between these representations. The `withType` rule creates a scope with a `typeOf` declaration that contains the type. In the adapted `declareType` rule, this constraint is used to convert the `T` argument to the representation that the `type` relation accepts. Likewise, the `typeOf` rule queries the `typeOf` declaration to extract the type from a scope. This rule is used in the `resolveType` rule to convert back to the term representation of a type.

Performing this change should resolve potential deadlocks when executing your specifications. Because the signatures of the rules in the original specification did not change, and the new specification should have identical semantics, the remainder of the specification should not be affected.

11.8.3 Using new Solver Features

The concurrent solver also comes with some new features that were not present in the traditional solver. This sections explains these features, and shows how to use them.

Grouping

Warning: Not yet written

Libraries

Warning: Not yet written

Data Flow Analysis Definition with FlowSpec

Programs that are syntactically well-formed are not necessarily valid programs. Programming languages typically impose additional *context-sensitive* requirements on programs that cannot be captured in a syntax definition. Languages use data and control flow to check certain extra properties that fall outside of names and type systems. The FlowSpec ‘Flow Analysis Specification Language’ supports the specification of rules to define the static control flow of a language, and data flow analysis over that control flow. FlowSpec supports flow-sensitive intra-procedural data flow analysis.

12.1 Introduction

FlowSpec is a domain-specific meta-language for the specification of static control-flow and data-flow analysis over that control flow. We briefly explain these basic concepts.

12.1.1 Control Flow Graphs

Control-flow represents the execution order of a program. Depending on the input given to the program, or other things the program may observe of its execution environment (e.g. network communication, or a source of noise used to generate pseudo-random numbers), a program may execute a different trace of instructions. Since in general programs may not terminate at all, and humans are not very adapt at reasoning about possible infinities, we use a finite representation of possibly infinite program traces using control-flow graphs.

Control-flow graphs are similarly finite as program text and are usually very similar, giving rise to a visual representation of the program. Loops in the program are represented as cycles in the control-flow graph, conditional code is represented by a split in control-flow which is merged again automatically after the conditional code.

12.1.2 Data Flow Analysis over Control Flow Graphs

Data-flow analysis propagates information either forward or backward along the control-flow graph. This can be information that approximates the data that is handled by the program, or the way in which the program interacts with memory, or something else altogether.

Examples of data-flow analysis include [constant analysis](#) which checks when variables that are used in the program are guaranteed to have the same value regardless of the execution circumstances of the program, or [live variables analysis](#) which identifies if values in variables are actually observable by the program.

12.2 Language Reference

This section gives a systematic overview of the FlowSpec language.

12.2.1 Lexical matters

Identifiers

Most identifiers in FlowSpec fall into one of two categories, which we will refer to as:

- *Lowercase identifiers*, that start with a lowercase character, and must match the regular expression `[a-z][a-zA-Z0-9]*`.
- *Uppercase identifiers*, that start with an uppercase character, and must match the regular expression `[A-Z][a-zA-Z0-9]*`.

Comments

Comments in FlowSpec follow C-style comments:

- `// ... single line ...` for single-line comments
- `/* ... multiple lines ... */` for multi-line comments

Multi-line comments can be nested, and run until the end of the file when the closing `*/` is omitted.

12.2.2 Terms and patterns

Terms can be constructed terms from either the abstract syntax tree or a user-defined algebraic data type. Tuples are built-in, as are sets and maps. The latter two have special construction and comprehension syntax.

```
term = ctor-id "(" {term ","}* ")"
      | "(" {term ","}* ")"
      | "{" {term ","}* "}"
      | "{" term "|" {term ","}* "}"
      | "{" {(term "|->" term) ","}* "}"
      | "{" term "|->" term "|" {(term "|->" term) ","}* "}"
```

Pattern matching is done with patterns for constructed terms and tuples. Parts can also be bound to a variable or ignored with a wildcard.

```
pattern = ctor-id "(" {pattern ","}* ")"
          "(" {pattern ","}* ")"
          var-id "@" pattern
          "_"
          var-id
```

12.2.3 Modules

```
module module-id

    section*
```

FlowSpec specifications are organized in modules. A module has a name. This name consists of one or more names separated by slashes {name `"/"`}+. The names must match the regular expression `[a-zA-Z0-9_][a-zA-Z0-9_\.\\-]*`.

Every module is defined in its own file, with the extension `.flo`. The module name and the file paths must match.

Example. An empty module `analysis/flow/control`, defined in a file `.../analysis/flow/control.flo`.

```
module analysis/flow/control

// TODO: add control-flow rules
```

Modules consist of sections for imports, control-flow rules, data-flow properties and rules, lattices and functions.

Imports

```
imports

    module-ref*

    external

        module-ref*
```

A module can import definitions from other modules by importing the other module. Imports are specified in an `imports` section, which lists the modules being imported. A module reference can be:

- A module identifier, which imports a single module with that name.
- A wildcard, which imports all modules with a given prefix. A wildcard is like a module identifier, but with a dash as the last part, as in {name `"/"`}+ `"/-`.

A wildcard import does not work recursively. For example, `analysis/-` would imports `analysis/functions`, and `analysis/classes`, but not `analysis/lets/recursive`.

External imports allow you to import module of for example `Stratego`, to import the signatures of the abstract syntax you wish to match on.

Example. A main module importing several submodules.

```
module liveness

imports
    control

external
    signatures/-
```

12.2.4 Control Flow

```
control-flow rules

control-flow-rule*
```

The first step of analysis in FlowSpec is to define the control-flow through a program. This connection is established with rules that match patterns of abstract syntax and providing the control-flow of that pattern.

Rules

A normal control-flow rule maps an abstract syntax pattern to a list of control-flow edges.

```
pattern* = {cfg-edges " " " "}
```

These edges can start from the special `entry` and `exit` control-flow nodes that are provided to connect the pattern to the wider control-flow graph. Subtrees matched in the abstract syntax pattern are usually used directly at one side of an edge to connect their corresponding sub-control-flow graph. They can also be inserted as direct control-flow nodes using the `node` keyword. This is rarely used. More likely, you may want to insert the whole matched pattern as a node. The `this` keyword can be used for that.

```
cfg-edges = {cfg-edge-end "->"}+

cfg-edge-end = "entry"
              | "exit"
              | variable
              | "node" variable
              | "this"
```

A common case exists where you merely wish to register a pattern as a control-flow graph node. Rather than write out `[pattern] = entry -> this -> exit`, you can write `node [pattern]` for this.

Example. Module that defines control-flow for some expressions

```
module control

control-flow rules

  node Int(_)
  Add(l, r) = entry -> l -> r -> this -> exit
```

Root rules

A root of the control-flow defines the `start` and `end` nodes of a control-flow graph. You can have multiple control-flow graphs in the same AST, but not nested ones. Each control-flow graph has a unique `start` and `end` node. A `root` control-flow rule introduces the `start` and `end` node. In other control-flow rules these nodes can be referred to for abrupt termination.

```
cfg-edge-end = ...
              | "start"
              | "end"
```

Example. Module that defines control-flow for a procedure, and the return statement that goes straight to the end of the procedure.

```

module control

control-flow rules

  root Procedure(args, _, body) = start -> args -> body -> end
  Return(_) = entry -> this -> end

```

12.2.5 Data Flow

Data-flow analysis in FlowSpec is based on named *properties*. Data-flow properties are defined in a property definition section, their rules are defined in a property rules section. Properties have an associated lattices, whose operations take care of merging data at merge points in the control-flow.

```

properties

  property-definition*

```

```

property rules

  property-rule*

```

Definitions

A property definition consists only of the property name, with a lowercase start and otherwise camelcase for multiple words. The lattice looks like a type expression but uses a lattice name. This lattice instance is used internally for the data-flow computation.

```

property-definition = name ":" lattice

```

Rules

A property rule consists of the name of the property, a pattern within round brackets and an expression after the equals sign. The pattern matches a control-flow graph node by its originating AST and another control-flow graph node before or after it by name. The expression describes the effect of the matched control-flow graph node, in terms of a change to the value from the adjacent control-flow graph node matched. All rules of a property need to propagate the information in the same way, either forward or backward.

Each property needs to have at least one rule with the *start* or *end* pattern. This is pattern matches the extremal node of a control-flow graph and defines the initial value there. With set-based analyses this is usually the empty set, as usually nothing is known at that point. The extremal node needs to match the direction of the rules, with *start* for a forward analysis and *end* of a backward analysis.

```

property-rule = name "(" prop-pattern ")" "=" expr
prop-pattern = name "->" pattern
               | pattern "->" name
               | pattern "." "start"
               | pattern "." "end"

```

12.2.6 Lattices

Lattices definitions are defined in their own section.

```
lattices

    lattice-definition*
```

Each lattice definition consists of a name and a number of components: the underlying type, the least-upper-bound or join operation between two lattice values (the less-than-or-equal comparison is derived from this operation), the top value and the bottom value. The type is usually defined by the user in another section as an algebraic data type, and operated on with pattern matching.

```
name where
    type = type
    lub([name], name) = expr
    top = expr
    bottom = expr
```

12.2.7 Types

Algebraic data types can be defined in a types block.

```
types

    type-definition*
```

Each type has a name and one or more option preceded by a vertical bar. Each option has a constructor and zero or more arguments in round brackets.

```
name =
    ("|" ctor-id "(" {type ","}* ")" ) +
```

12.2.8 Expressions

There are many expressions supported to express (abstract) semantics of control-flow nodes.

Integers

Integer literals are as usual: an optional minus sign followed by one or more decimals.

Supported integer operations are

1. addition [+],
2. subtraction [-],
3. multiplication [*],
4. division [/],
5. modulo [%],
6. negate [-],
7. comparison [<, <=, >, >=, ==, !=].

Booleans

Boolean literals `true` and `false` are available as well as the usual boolean operations:

1. `and` [`&&`]
2. `or` [`||`]
3. `not` [`!`]

Sets and Maps

Set and map literals are both denoted with curly braces. A set literal contains a comma-separated list of elements: `{elem1, elem2, elem3}`. A map literal contains a comma-separated list of bindings of the form `key |-> value`.

Operations on sets and maps include

1. `union` [`\`],
2. `intersection` [`/`],
3. `set/map minus` [`\`],
4. `containment/lookup` [`in`].

There are also comprehensions of the form `{ new | old <- set, conditions }` or `{ newkey |-> newvalue | oldkey |-> oldvalue <- map, condition }`, where new elements or bindings are gathered based on old ones from a set or map, as long as the boolean condition expressions hold. Such a condition expression may also be a match expression without a body for the arms. This is commonly used to filter maps or sets.

Match expressions

Pattern matching can be done with a match expression: `match expr with | pattern1 => expr2 | pattern2 => expr2`, where `expr` are expressions and `pattern` are patterns. Terms and patterns are defined at the start of the reference.

Variables and references

Pattern matching can introduce variables. Other references include values in the lattice, such as `MaySet.bottom` or `MustSet.top`.

Function application

User defined functions are available to be called with `functionname(arg1, arg2)`. Lattice operations can also be called with for example `MaySet.lub(s1, s2)`.

Property lookup

Property lookup is similar to a function call, although property lookup only ever has a single argument.

12.2.9 Functions

Functions are defined in their own section.

```
functions

  function-definition*
```

A function definition consists of a name, typed arguments and a function body expression.

```
name ([{ (name ":" type) ", " }+]) =
  expr
```

12.3 Stratego API

The Stratego API to FlowSpec allows access to analysis result during and after analysis.

The full definition of the API can be found in the [flowspec/api](#) module.

12.3.1 Setup

Using the Stratego API requires a dependency on the FlowSpec Stratego code (source dependency), and an import of `flowspec/api`.

Example. A Stratego module importing the FlowSpec API.

```
module example

imports

  flowspec/api
```

12.3.2 Running the analysis

There are strategies to integrate FlowSpec analysis in the NaBL2 analysis, and strategies for doing both NaBL2 analysis and FlowSpec analysis on an AST.

Integrated into NaBL2 analysis

These can be used in the final phase of the NaBL2 analysis process using the *Stratego hooks*.

```
/**
 * Analyze the given AST with FlowSpec.
 * The FlowSpec analysis is added to given NaBL2 analysis result and returned.
 *
 * @param analysis:Analysis
 * @param propnames:String or List(String)
 * @type ast:Term -> Analysis
 */
flowspec-analyze(|analysis)
```

(continues on next page)

(continued from previous page)

```
/**
 * Analyze the given AST with FlowSpec, but only the given FlowSpec properties.
 * The FlowSpec analysis is added to given NaBL2 analysis result and returned.
 *
 * @param analysis:Analysis
 * @param propnames:String or List(String)
 * @type ast:Term -> Analysis
 */
flowspec-analyze(|analysis, propnames)
```

The analysis results are also usable at that point for generating editor messages. Integration with NaBL2 is done by giving the FlowSpec analysis result as the “custom final analysis result”:

```
nabl2-custom-analysis-unit-hook:
    (resource, ast, custom-initial-result) -> (resource, ast)

nabl2-custom-analysis-final-hook(|a):
    (resource, custom-initial-result, custom-unit-results) -> (errors, warnings,
↳notes, custom-final-result)
    with asts      := <map(\(ast-resource, ast) -> <nabl2--index-ast(|ast-resource)>↳
↳ast\)> custom-unit-results ; // workaround for https://yellowgrass.org/issue/NaBL2/
↳54
    custom-final-result := <flowspec-analyze(|a)> asts ;
    errors      := ... ;
    warnings    := ... ;
    notes       := ...
```

This propagates the AST of each unit from the unit phase, and analyzes all of them together in the final phase. The custom-final-result is returned so that NaBL2 preserves it for later usage. FlowSpec provides convenience functions that request the custom final result again later:

```
/**
 * Get analysis for the given AST node. Includes flowspec analysis if custom final_
↳hook is set up
 * correctly.
 *
 * @type node:Term -> Analysis
 */
flowspec-get-ast-analysis

/**
 * Get analysis for the given resource. Includes flowspec analysis if custom final_
↳hook is set up
 * correctly.
 *
 * @type filename:String -> Analysis
 */
flowspec-get-resource-analysis
```

Running the analysis manually

Sometimes you need data-flow analysis between transformations which change the program. That means you need to run the analysis just before a transformation to have analysis results corresponding to the current program.

The following strategies execute the analysis and help with consuming the resulting tuple.

```

/**
 * Analyze the given AST with NaBL2 and FlowSpec
 *
 * @param resource:String
 * @type ast:Term -> (ast:Term, Analysis, errors:List(EditorMessage),
↳warnings:List(EditorMessage), notes:List(EditorMessage))
 */
flowspec-analyze-ast(|resource)

/**
 * Analyze the given AST with NaBL2 and FlowSpec.
 * Transform the AST with pre before the FlowSpec analysis, and with post after the
↳FlowSpec analysis.
 *
 * @param pre:Term -> Term
 * @param post:Term -> Term
 * @param resource:String
 * @type ast:Term -> (ast:Term, Analysis, errors:List(EditorMessage),
↳warnings:List(EditorMessage), notes:List(EditorMessage))
 */
flowspec-analyze-ast(pre,post|resource)

/**
 * Analyze the given AST with NaBL2 and FlowSpec, but only the given FlowSpec
↳properties.
 *
 * @param resource:String
 * @param propnames:String or List(String)
 * @type ast:Term -> (ast:Term, Analysis, errors:List(EditorMessage),
↳warnings:List(EditorMessage), notes:List(EditorMessage))
 */
flowspec-analyze-ast(|resource, propname)

/**
 * Analyze the given AST with NaBL2 and FlowSpec, but only the given FlowSpec
↳properties.
 * Transform the AST with pre before the FlowSpec analysis, and with post after the
↳FlowSpec analysis.
 *
 * @param pre:Term -> Term
 * @param post:Term -> Term
 * @param resource:String
 * @param propnames:String or List(String)
 * @type ast:Term -> (ast:Term, Analysis, errors:List(EditorMessage),
↳warnings:List(EditorMessage), notes:List(EditorMessage))
 */
flowspec-analyze-ast(pre,post|resource, propnames)

/**
 * Take the analyze-ast 5-tuple output and return the result of applying the given
↳strategy to the AST.
 * Note that the strategy takes the analysis object as a term argument.
 *
 * @param s(/Analysis): Term -> Term
 * @type ast: (ast:Term, Analysis, errors:List(EditorMessage),
↳warnings:List(EditorMessage), notes:List(EditorMessage)) -> Term
 */

```

(continues on next page)

(continued from previous page)

```
flowspec-then(s)

/**
 * Analyze the given AST with NaBL2 and FlowSpec, but only the given FlowSpec
 * → properties.
 * Then return the result of applying the given strategy to the AST.
 * Note that the strategy takes the analysis object as a term argument.
 *
 * @param s(|Analysis): Term -> Term
 * @param resource:String
 * @param propnames:String or List(String)
 * @type ast:Term -> Term
 */
flowspec-analyze-ast-then(s|resource, propnames)

/**
 * Analyze the given AST with NaBL2 and FlowSpec, but only the given FlowSpec
 * → properties.
 * Transform the AST with pre before the FlowSpec analysis, and with post after the
 * → FlowSpec analysis.
 * Then return the result of applying the given strategy to the AST.
 * Note that the strategy takes the analysis object as a term argument.
 *
 * @param pre:Term -> Term
 * @param post:Term -> Term
 * @param s(|Analysis): Term -> Term
 * @param resource:String
 * @param propnames:String or List(String)
 * @type ast:Term -> Term
 */
flowspec-analyze-ast-then(pre, post, s|resource, propnames)
```

12.3.3 Querying analysis

The NaBL2 API defines several *strategies to get an analysis term by resource name or from an AST node*. This analysis term can then be passed to the querying strategies that give access to the data flow properties, *if* you hooked FlowSpec into the NaBL2 analysis process.

The other way to get the analysis term is to execute the analysis with the *flowspec-analyze-ast** variants.

Control-flow graph

There are a number of strategies to get the control-flow graph nodes associated with an AST fragment, as well as control-flow graph navigation strategies and AST search strategies to get back to the AST from a control-flow graph node. Note that querying the control-flow graph is cheap but finding the way back from the control-flow graph to the AST is more expensive.

```
/**
 * Get the control flow graph node associated with the given term.
 *
 * @param a : Analysis
 * @type term:Term -> CFGNode
 */
flowspec-get-cfg-node(|a)
```

(continues on next page)

(continued from previous page)

```

/**
 * Get the control flow graph start node associated with the given term.
 *
 * @param a : Analysis
 * @type term:Term -> CFGNode
 */
flowspec-get-cfg-start-node(|a)

/**
 * Get the control flow graph start node associated with the given term.
 *
 * @param a : Analysis
 * @type term:Term -> CFGNode
 */
flowspec-get-cfg-end-node(|a)

/**
 * Get the control flow graph start node associated with the given term.
 *
 * @param a : Analysis
 * @type term:Term -> CFGNode
 */
flowspec-get-cfg-entry-node(|a)

/**
 * Get the control flow graph start node associated with the given term.
 *
 * @param a : Analysis
 * @type term:Term -> CFGNode
 */
flowspec-get-cfg-exit-node(|a)

/**
 * Get the control flow graph start node associated with the given term.
 *
 * @param a : Analysis
 * @type term:Term -> CFGNode
 */
flowspec-get-cfg-prev-nodes(|a)

/**
 * Get the control flow graph start node associated with the given term.
 *
 * @param a : Analysis
 * @type term:Term -> CFGNode
 */
flowspec-get-cfg-next-nodes(|a)

/**
 * Find AST node corresponding to the CFGNode back again
 *
 * @param ast : Term
 * @type node:CFGNode -> Term
 */
flowspec-cfg-node-ast(|ast)

```

(continues on next page)

(continued from previous page)

```
/**
 * Find AST node corresponding to the CFGNode back again
 *
 * @param ast : Term
 * @type pos:Position -> Term
 */
flowspec-pos-ast(|ast)

/**
 * Find parent of AST node corresponding to the CFGNode back again by matching the
 * →parent with
 * →the parent argument and giving back the child that is likely to be a match to
 * →the CFG node.
 *
 * @param parent : Term -> Term
 * @param ast : Term
 * @type node:CFGNode -> Term
 */
flowspec-cfg-node-ast(parent|ast)

/**
 * Find parent of AST node corresponding to the CFGNode back again by matching the
 * →parent with
 * →the parent argument and giving back the child that is likely to be a match to
 * →the CFG node.
 *
 * @param parent : Term -> Term
 * @param ast : Term
 * @type pos:Position -> Term
 */
flowspec-pos-ast(parent|ast)

/**
 * Get the position of an AST node.
 *
 * @type Term -> Position
 */
flowspec-get-position
```

Data flow properties

FlowSpec properties can be read in two versions, *pre* and *post*. These indicate whether the effect of the cfg node has been applied yet. Whether or not it is applied depends on the direction of the analysis. *pre* for a forward analysis is without the effect of the node, but *pre* for a backward analysis includes the effect of the node.

Note that each strategy can simply take the term that's associated with the control-flow graph node. But the control-flow graph node itself is also an accepted input.

```
/**
 * Get the property of the control flow graph node associated with
 * the given term. The value returned is the value of the property
 * →_before_ the effect of the control flow graph node.
 *
 * @param a : Analysis
 * @param prop : String
```

(continues on next page)

(continued from previous page)

```

* @type term:Term -> Term
*/
flowspec-get-property-pre(|a, propname)

/**
 * Get the property of the control flow graph node associated with
 * the given term. The value returned is the value of the property
 * _after_ the effect of the control flow graph node.
 *
 * @param a : Analysis
 * @param prop : String
 * @type term:Term -> Term
 */
flowspec-get-property-post(|a, propname)

/**
 * Get the property of the control flow graph node associated with
 * the given term. The value returned is the value of the property
 * _after_ the effect of the control flow graph node. If no node
 * is found the exit control flow graph node of the AST node is
 * queried for its post-effect property value.
 *
 * @param a : Analysis
 * @param prop : String
 * @type term:Term -> Term
 */
flowspec-get-property-post-or-exit-post(|analysis-result, analysis-name)

```

FlowSpec data helpers

FlowSpec sets and maps are passed back to Stratego as lists wrapped in Set and Map constructors. As a convenience, the most common operations are lifted and added to the flowspec API:

```

/**
 * Check if a FlowSpec Set contains an element. Succeeds if the given strategy_
↳succeeds for at
 * least one element.
 *
 * @param s: Term -?>
 * @type FlowSpecSet -?> FlowSpecSet
 */
flowspec-set-contains(s)

/**
 * Look up elements in a FlowSpec Set of pairs. Returns the right elements of all_
↳pairs where
 * the given strategy succeeds on the left element.
 *
 * @param s: Term -?>
 * @type FlowSpecSet -?> List(Term)
 */
flowspec-set-lookup(s)

/**
 * Look up a key in a FlowSpec Map. Returns the element if the given key exists in_
↳the map.

```

(continues on next page)

(continued from previous page)

```

*
* @param k: Term
* @type FlowSpecMap -?> Term
*/
flowspec-map-lookup(|k)

```

12.3.4 Hover text

For a hover implementation that displays name, type and FlowSpec properties use:

```

/**
 * Provides a strategy for a hover message with as much information as possible,
 * about name, type
 * (from NaBl2) and FlowSpec properties.
 */
flowspec-editor-hover(language-pp)

```

12.3.5 Profiling information

```

/**
 * If flowspec-debug-profile is extended to succeed, some timing information will be
 * printed in
 * stderr when using flowspec-analyze*.
 */
flowspec-debug-profile

```

12.4 Configuration

We will show you how to prepare your project for use with FlowSpec, and write your first small specification.

12.4.1 Prepare your project

You can start using FlowSpec by creating a new project, or by modifying an existing project. See below for the steps for your case.

Start a new project

If you have not done this already, install Spoofax Eclipse, by following the *installation instructions*.

Create a new project by selecting New > Project... from the menu. Selecting Spoofax > Spoofax language project from the list, and click Next. After filling in a project name, an identifier, name etc will be automatically suggested. Select NaBL2 as the analysis type, FlowSpec builds on top of NaBL2's analysis infrastructure. Click Finish to create the project.

Add the following dependencies in the `metaborg.yaml` file:

```
---
# ...
dependencies:
  compile:
    - org.metaborg:flowspec.lang:${metaborgVersion}
  source:
    - org.metaborg:flowspec.lang:${metaborgVersion}
```

Add menus to access the result of analysis, by adding the following import to `editor/Main.esv`.

```
module Main

imports

  flowspec/Menus
```

Convert an existing project

If you have an existing project, and you want to start using FlowSpec, there are a few changes you need to make.

First of all, make sure the `metaborg.yaml` file contains at least the following dependencies.

```
---
# ...
dependencies:
  compile:
    - org.metaborg:org.metaborg.meta.nabl2.lang:${metaborgVersion}
    - org.metaborg:flowspec.lang:${metaborgVersion}
  source:
    - org.metaborg:org.metaborg.meta.nabl2.shared:${metaborgVersion}
    - org.metaborg:org.metaborg.meta.nabl2.runtime:${metaborgVersion}
    - org.metaborg:flowspec.lang:${metaborgVersion}
```

We will set things up, such that analysis rules will be grouped together in the directory `trans/analysis`. Create a file `trans/analysis/main.str` that contains the following.

```
module analysis/main

imports

  nabl2shared
  nabl2runtime
  analysis/-
```

Add the following lines to your main `trans/LANGUAGE.str`.

```
module LANGUAGE

imports

  analysis/main

rules

  editor-analyze = nabl2-analyze(desugar-pre)
```

If your language does not have a desugaring step, use `nabl2-analyze(id)` instead.

Add an NaBL2 specification. The most minimal one is the following.

```
module analysis/minimal

rules

init.

[[ _ ]].
```

Running and integrating the FlowSpec analysis is explained on the [Stratego API page](#).

Finally, we will add reference resolution and menus to access the result of analysis, by adding the following lines to editor/Main.esv.

```
module Main

imports

  nabl2/References
  nabl2/Menus
  flowspec/Menus
```

You can now continue to the [example specification here](#), or directly to the [language reference](#).

12.4.2 Inspecting analysis results

You can debug your specification by inspecting the result of analysis.

The result of analysis can be inspected, by selecting elements from the Spoofax > FlowSpec Analysis the menu. For multi-file projects, use the Project results, or the File results for single-file projects. The result is given as a control-flow graph annotated with data-flow properties in the DOT format used by GraphViz. If you have GraphViz installed, you can set the dot executable in the settings of the graphviz editor to allow you to jump straight from Eclipse to the rendered graph.

12.5 Examples (under construction)

12.6 Bibliography

- FlowSpec [\[S1\]](#)

Transformation with Stratego

Parsing a program text results in an abstract syntax tree. Stratego is a language for defining transformations on such trees. Stratego provides a term notation to construct and deconstruct trees and uses *term rewriting* to define transformations. Instead of applying all rewrite rules to all sub-terms, Stratego supports programmable *rewriting strategies* that control the application of rewrite rules.

13.1 Stratego Tutorial/Reference

This documentation is closely based on the Stratego Tutorial and Reference Manual that was developed for Stratego/XT 0.17. While the basic explanation of the language is up-to-date, this manual makes liberal use of a tool called the “Stratego Shell,” which is not currently provided as part of Spoofax. However, in many cases there are guidelines for running similar examples in the Spoofax Eclipse interface, and pointers to a working project that contains many of the examples.

13.1.1 1. Introduction

Program transformation is the mechanical manipulation of a program in order to improve it relative to some cost function C , such that $C(P) > C(tr(P))$, i.e., the cost decreases as a result of applying the transformation. The cost of a program can be measured in different dimensions such as performance, memory usage, understandability, flexibility, maintainability, portability, correctness, or satisfaction of requirements. Related to these goals, program transformations are applied in different settings; e.g. compiler optimizations improve performance and refactoring tools aim at improving understandability.

While transformations can be achieved by manual manipulation of programs, in general, the aim of program transformation is to increase programmer productivity by automating programming tasks, thus enabling programming at a higher level of abstraction, and increasing maintainability and re-usability of programs. Automatic application of program transformations requires their implementation in a programming language. In order to make the implementation of transformations productive, such a programming language should support abstractions for the domain of program transformation.

Stratego is a language designed for this purpose. It is a language based on the paradigm of rewriting with programmable rewriting strategies and dynamic rules.

1.1. Transformation by Rewriting

Term rewriting is an attractive formalism for expressing basic program transformations. A rewrite rule $p1 \rightarrow p2$ expresses that a program fragment matching the left-hand side pattern $p1$ can be replaced by the instantiation of the right-hand side pattern $p2$. For instance, the rewrite rule

```
| [ i + j ] | -> | [ k ] | where sum(i, j) => k
```

expresses constant folding for addition, i.e., replacing an addition of two constants by their sum. Similarly, the rule

```
| [ if 0 then e1 else e2 ] | -> | [ e2 ] |
```

defines unreachable code elimination by reducing a conditional statement to its right branch since the left branch can never be executed. Thus, rewrite rules can directly express laws derived from the semantics of the programming language, making the verification of their correctness straightforward. A correct rule can be safely applied anywhere in a program. A set of rewrite rules can be directly operationalized by rewriting to normal form, i.e. exhaustive application of the rules to a term representing a program. If the rules are confluent and terminating, the order in which they are applied is irrelevant.

1.2. Limitations of Pure Rewriting

However, there are two limitations to the application of standard term rewriting techniques to program transformation: the need to intertwine rules and strategies in order to control the application of rewrite rules and the context-free nature of rewrite rules.

1.3. Transformation Strategies

Exhaustive application of all rules to the entire abstract syntax tree of a program is not adequate for most transformation problems. The system of rewrite rules expressing basic transformations is often non-confluent and/or non-terminating. An ad hoc solution that is often used is to encode control over the application of rules into the rules themselves by introducing additional function symbols. This intertwining of rules and strategies obscures the underlying program equalities, incurs a programming penalty in the form of rules that define a traversal through the abstract syntax tree, and disables the reuse of rules in different transformations.

Stratego solves the problem of control over the application of rules while maintaining the separation of rules and strategies. A strategy is a little program that makes a selection from the available rules and defines the order and position in the tree for applying the rules. Thus rules remain pure, are not intertwined with the strategy, and can be reused in multiple transformations.

1.4. Context-Sensitive Transformation

The second problem of rewriting is the context-free nature of rewrite rules. A rule has access only to the term it is transforming. However, transformation problems are often context-sensitive. For example, when inlining a function at a call site, the call is replaced by the body of the function in which the actual parameters have been substituted for the formal parameters. This requires that the formal parameters and the body of the function are known at the call site, but these are only available higher up in the syntax tree. There are many similar problems in program transformation, including bound variable renaming, typechecking, data flow transformations such as constant propagation, common-subexpression elimination, and dead code elimination. Although the basic transformations in all these applications can be expressed by means of rewrite rules, these require contextual information.

The following chapters give a tutorial for the Stratego language in which these ideas are explained and illustrated.

13.1.2 2. Terms

Stratego programs transform terms. When using Stratego for program transformation, terms typically represent the abstract syntax tree of a program. But Stratego does not much care what a term represents. Terms can just as well represent structured documents, software models, or anything else that can be rendered in a structured format.

Generally program text is transformed into a term by means of parsing, and turned back into program text by means of pretty-printing. One way to achieve this is by using [SDF3](#). For most of the examples, we will just assume that we have terms that should be transformed and ignore parsing and pretty-printing. However, when we turn to running examples in the Spoofax environment in the Eclipse IDE, we will rely on SDF3 as that is the primary way to produce terms in Spoofax/Eclipse.

2.1. Annotated Term Format

Terms in Stratego are terms in the *Annotated Term Format*, or *ATerms* for short. The ATerm format provides a set of constructs for representing trees, comparable to XML or abstract data types in functional programming languages. For example, the code `4 + f(5 * x)` might be represented in a term as:

```
Plus(Int("4"), Call("f", [Mul(Int("5"), Var("x"))]))
```

ATerms are constructed from the following elements:

- **Integer:** An integer constant, that is a list of decimal digits, is an ATerm.

Examples: 1, 12343.

- **String:** A string constant, that is a list of characters between double quotes is an ATerm. Special characters such as double quotes and newlines should be escaped using a backslash. The backslash character itself should be escaped as well.

Examples: "foobar", "string with quotes\"", "escaped escape character\\ and a newline\n".

- **Constructor application:** A constructor is an identifier, that is an alphanumeric string starting with a letter, or a double quoted string.

A constructor application `c(t1, ..., tn)` creates a term by applying a constructor to a list of zero or more terms. For example, the term `Plus(Int("4"), Var("x"))` uses the constructors `Plus`, `Int`, and `Var` to create a nested term from the strings "4" and "x".

- **List:** A list is a term of the form `[t1, ..., tn]`, that is a list of zero or more terms between square brackets. While all applications of a specific constructor typically have the same number of subterms, lists can have a variable number of subterms. The elements of a list are typically of the same type, while the subterms of a constructor application can vary in type.

Example: The second argument of the call to "f" in the term `Call("f", [Int("5"), Var("x")])` is a list of expressions.

- **Tuple:** A tuple `(t1, ..., tn)` is a constructor application without a constructor.

Example: `(Var("x"), Type("int"))`

- **Annotation:** The elements defined above are used to create the structural part of terms. Optionally, a term can be annotated with a list of terms. These annotations typically carry additional semantic information about the term. An annotated term has the form `t{t1, ..., tn}`.

Example: `Lt(Var("n"), Int("1")){Type("bool")}`. The contents of annotations is up to the application.

2.2. Exchanging Terms

The term format described above is used in Stratego programs to denote terms, but is also used to exchange terms between programs. Thus, the internal format and the external format exactly coincide. Of course, internally a Stratego program uses a data-structure in memory with pointers rather than manipulating a textual representation of terms. But this is completely hidden from the Stratego programmer.

2.3. Inspecting Terms

As a Stratego programmer you will be looking a lot at raw ATerms. Stratego pioneers did this by opening an ATerm file in *emacs* and trying to get a sense of the structure by parenthesis highlighting and inserting newlines here and there. These days your life is much more pleasant through pretty-printing ATerms, which adds layout to a term to make it readable. For example, parsing the following program

```
let function fact(n : int) : int =
    if n < 1 then 1 else (n * fact(n - 1))
in printint(fact(10))
end
```

produces the following ATerm:

```
Let ([FunDecs ([FunDec ("fact", [FArg ("n", Tp (Tid ("int"))]), Tp (Tid ("int")),
If (Lt (Var ("n"), Int ("1")), Int ("1"), Seq ([Times (Var ("n"), Call (Var ("fact"),
[Minus (Var ("n"), Int ("1"))]))]))]), [Call (Var ("printint"), [Call (Var (
"fact"), [Int ("10")])])])])
```

By pretty-printing the term we get a much more readable term:

```
Let (
  [ FunDecs (
    [ FunDec (
      "fact"
      , [FArg ("n", Tp (Tid ("int")))]
      , Tp (Tid ("int"))
      , If (
        Lt (Var ("n"), Int ("1"))
        , Int ("1")
        , Seq ([ Times (Var ("n"), Call (Var ("fact"), [Minus (Var ("n"), Int ("1"))]))
          ])
        )
      )
    ]
  )
, [ Call (Var ("printint"), [Call (Var ("fact"), [Int ("10")])])
]
)
```

In Spoofax/Eclipse, you will find that in some contexts ATerms are automatically pretty-printed, whereas in others they are simply printed linearly. However, you can obtain assistance with perceiving the structure of any ATerm by writing it into a file with the “.aterm” extension and opening it in the Spoofax Editor in Eclipse. On the right there will be a convenient Outline Navigator which allows you to select any node in the ATerm and see the entire subtree below it highlighted in the editor.

2.4. Signatures

To use terms in Stratego programs, their constructors should be declared in a signature. A signature declares a number of sorts and a number of constructors for these sorts. For each constructor, a signature declares the number and types of its arguments. For example, the following signature declares some typical constructors for constructing abstract syntax trees of expressions in a programming language:

```
signature
  sorts Id Exp
  constructors
    : String -> Id
  Var   : Id -> Exp
  Int   : Int -> Exp
  Plus  : Exp * Exp -> Exp
  Mul   : Exp * Exp -> Exp
  Call  : Id * List(Exp) -> Exp
```

Currently, the Stratego compiler only checks the arity of constructor applications against the signature. Still, it is considered good style to also declare the types of constructors in a sensible manner for the purpose of documentation.

The situation in Spoofax/Eclipse is even more convenient; if you have an SDF3 language specification, Spoofax will automatically generate a corresponding signature definition that you can import into Stratego.

13.1.3 3. Running Stratego Programs

Note: This entire chapter refers to tools that are a part of Stratego/XT. We will see examples of how to run similar programs in Spoofax/Eclipse toward the end of the next chapter. However, even if you plan to use Spoofax/Eclipse, it's worth at least skimming through this chapter to see Stratego in action and understand how most of the examples throughout this manual will be presented.

Now let's see how we can actually transform terms using Stratego programs. In the rest of this chapter we will first look at the structure of Stratego programs, and how to compile and run them. In the next chapters we will then see how to define transformations.

3.1. Compiling Stratego Programs

The simplest program you can write in Stratego is the following `identity.str` program:

```
module identity
imports list-cons
strategies
  main = id
```

It features the following elements: each Stratego file is a module, which has the same name as the file it is stored in without the `.str` extension. A module may import other modules in order to use the definitions in those modules. A module may contain one or more `strategies` sections that introduce new strategy definitions. It will become clear later what strategies and strategy definitions are. Each Stratego program has *one main definition*, which indicates the strategy to be executed on invocation of the program. In the example, the body of this program's main definition is the `identity` strategy `id`.

Now let's see what this program means. To find that out, we first need to compile it, which we do using the Stratego compiler `strc` as follows:

```
$ strc -i identity.str
[ strc | info ] Compiling 'identity.str'
[ strc | info ] Front-end succeeded           : [user/system] = [0.59s/0.56s]
[ strc | info ] Back-end succeeded            : [user/system] = [0.46s/0.16s]
[ strc | info ] C compilation succeeded       : [user/system] = [0.28s/0.23s]
[ strc | info ] Compilation succeeded         : [user/system] = [1.35s/0.95s]
```

The `-i` option of `strc` indicates the module to compile. The compiler also reads all imported modules, in this case the `list-cons.str` module that is part of the Stratego library and that `strc` magically knows how to find. The compiler prints some information about what it is doing, i.e., the stages of compilation that it goes through and the times for those stages. You can turn this off using the argument `--verbose 0`. However, since the compiler is not very fast, it may be satisfying to see something going on.

The result of compilation is an executable named `identity` after the name of the main module of the program. Just to satisfy our curiosity we inspect the file system to see what the compiler has done:

```
$ ls -l identity*
-rwxrwxr-x  1 7182 Sep  7 14:54 identity*
-rw-----  1 1362 Sep  7 14:54 identity.c
-rw-rw-r--  1  200 Sep  7 14:54 identity.dep
-rw-rw-r--  1 2472 Sep  7 14:54 identity.o
-rw-rw-r--  1  57 Sep  7 13:03 identity.str
```

Here we see that in addition to the executable the compiler has produced a couple of other files. First of all the `identity.c` file gives away the fact that the compiler first translates a Stratego program to C and then uses the C compiler to compile to machine code. The `identity.o` file is the result of compiling the generated C program. Finally, the contents of the `identity.dep` file will look somewhat like this:

```
identity: \
    /usr/local/share/stratego-lib/collection/list/cons.rtree \
    /usr/local/share/stratego-lib/list-cons.rtree \
    ./identity.str
```

It is a rule in the Make language that declares the dependencies of the `identity` program. You can include this file in a Makefile to automate its compilation. For example, the following Makefile automates the compilation of the `identity` program:

```
include identity.dep

identity : identity.str
    strc -i identity.str
```

Just invoke `make` on the command-line whenever you change something in the program.

Ok, we were digressing a bit. Let's turn back to finding out what the `identity` program does. When we execute the program with some arbitrary arguments on the command-line, this is what happens:

```
$ ./identity foo bar
["./identity", "foo", "bar"]
```

The program writes to `stdout` the list of command-line arguments as a list of strings in the ATerm format. So what we have learned is that a Stratego program applies its main strategy to the list of command-line arguments, and writes the resulting term to `stdout`. Since the strategy in the `identity` program is the identity transformation it just writes the original command-line arguments (as a term).

3.2. Basic Input and Output

That was instructive, but not very useful. We are not interested in transforming lists of strings, but rather programs represented as terms. So we want to read a term from a file, transform it, and write it to another file. Let's open the bag of tricks. The `identity-io` program improves the previous program:

```
module identity-io
imports libstrategolib
strategies
  main = io-wrap(id)
```

The program looks similar to the previous one, but there are a couple of differences. First, instead of importing module `list-cons`, this module imports `libstrategolib`, which is the interface to the separately compiled Stratego library. This library provides a host of useful strategies that are needed in implementing program transformations. Part IV gives an overview of the Stratego library, and we will every now and then use some useful strategies from the library before we get there.

Right now we are interested in the `io-wrap` strategy used above. It implements a wrapper strategy that takes care of input and output for our program. To compile the program we need to link it with the `stratego-lib` library using the `-la` option:

```
$ strc -i identity-io.str -la stratego-lib
```

What the relation is between `libstrategolib` and `stratego-lib` will become clear later; knowing that it is needed to compile programs using `libstrategolib` suffices for now.

If we run the compiled `identity-io` program with its `--help` option we see the standard interface supported by the `io-wrap` strategy:

```
$ ./identity-io --help
Options:
  -i f|--input f    Read input from f
  -o f|--output f   Write output to f
  -b               Write binary output
  -S|--silent       Silent execution (same as --verbose 0)
  --verbose i       Verbosity level i (default 1)
                    ( i as a number or as a verbosity descriptor:
                      emergency, alert, critical, error,
                      warning, notice, info, debug, vomit )
  -k i | --keep i   Keep intermediates (default 0)
  --statistics i    Print statistics (default 0 = none)
  -h|-?|--help      Display usage information
  --about           Display information about this program
  --version         Same as --about
```

The most relevant options are the `-i` option for the input file and the `-o` option for the output file. For instance, if we have some file `foo-bar.trm` containing an `ATerm` we can apply the program to it:

```
$ echo "Foo(Bar())" > foo-bar.trm
$ ./identity-io -i foo-bar.trm -o foo-bar2.trm
$ cat foo-bar2.trm
Foo(Bar)
```

If we leave out the `-i` and/or `-o` options, input is read from `stdin` and output is written to `stdout`. Thus, we can also invoke the program in a pipe:

```
$ echo "Foo(Bar())" | ./identity-io
Foo(Bar)
```

Now it might seem that the `identity-io` program just copies its input file to the output file. In fact, the `identity-io` does not just accept any input. If we try to apply the program to a text file that is not an ATerm, it protests and fails:

```
$ echo "+ foo bar" | ./identity-io
readFromTextFile: parse error at line 0, col 0
not a valid term
./identity: rewriting failed
```

So we have written a program to check if a file represents an ATerm.

3.3. Combining Transformations

A Stratego program based on `io-wrap` defines a transformation from terms to terms. Such transformations can be combined into more complex transformations, by creating a chain of tool invocations. For example, if we have a Stratego program `trafo-a` applying some undefined `transformation-a` to the input term of the program

```
module trafo-a
imports libstrategolib
strategies
  main = io-wrap(transformation-a)
  transformation-a = ...
```

and we have another similar program `trafo-b` applying a `transformation-b`:

```
module tool-b
imports libstrategolib
strategies
  main = io-wrap(transformation-b)
  transformation-b = ...
```

then we can combine the transformations to transform an input file to an output file using a Unix pipe, as in

```
$ tool-a -i input | tool-b -o output
```

or using an intermediate file:

```
$ tool-a -i input -o intermediate
$ tool-b -i intermediate -o output
```

3.4. Running Programs Interactively with the Stratego Shell

We have just learned how to write, compile, and execute Stratego programs. This is the normal mode for development of transformation systems with Stratego. Indeed, we usually do not invoke the compiler from the command-line *by hand*, but have an automated build system based on (auto)make to build all programs in a project at once. For learning to use the language this can be rather laborious, however. Therefore, we have also developed the [Stratego Shell](#), an interactive interpreter for the Stratego language. The shell allows you to type in transformation strategies on the command-line and directly seeing their effect on the current term. While this does not scale to developing large programs, it can be instructive to experiment while learning the language. In the following chapters we will use the `stratego-shell` to illustrate various features of the language.

Here is a short session with the Stratego Shell that shows the basics of using it:

```
$ stratego-shell
stratego> :show
()
stratego> !Foo(Bar())
Foo(Bar)
stratego> id
Foo(Bar)
stratego> fail
command failed
stratego> :show
Foo(Bar)
stratego> :quit
Foo(Bar)
$
```

The shell is invoked by calling the command `stratego-shell` on the regular command-line. The `stratego>` prompt then indicates that you have entered the Stratego Shell. After the prompt you can enter strategies or special shell commands.

Strategies are the statements and functions of the Stratego language. A strategy transforms a term into a new term, or fails. The term to which a strategy is applied, is called the *current term*. In the Stratego Shell you can see the current term with `:show`. In the session above we see that the current term is the empty tuple if you have just started the Stratego Shell. At the prompt of the shell you can enter strategies. If the strategy succeeds, then the shell will show the transformed term, which is now the new current term. For example, in the session above the strategy `!Foo(Bar())` replaces the current term with the term `Foo(Bar())`, which is echoed directly after applying the strategy. The next strategy that is applied is the identity strategy `id` that we saw before. Here it becomes clear that it just returns the term to which it is applied. Thus, we have the following general scheme of applying a strategy to the current term:

```
current term
stratego> strategy expression
transformed current
stratego>
```

Strategies can also fail. For example, the application of the `fail` strategy always fails. In the case of failure, the shell will print a message and leave the current term untouched:

```
current term
stratego> strategy expression
command failed
stratego> :show
current term
```

Finally, you can leave the shell using the `:quit` command.

The Stratego Shell has a number of non-strategy commands to operate the shell configuration. These commands are recognizable by the `:` prefix. The `:help` command tells you what commands are available in the shell:

```
$ stratego-shell
stratego> :help

Rewriting
  strategy          rewrite the current subject term with strategy

Defining Strategies
  id = strategy      define a strategy  (doesn't change the current subject term)
  id : rule          define a rule      (doesn't change the current subject term)
  import modname     import strategy definitions from 'modname' (file system or xtc)
```

(continues on next page)

(continued from previous page)

<code>:undef id</code>	delete definitions of all strategies 'id'/(s,t)
<code>:undef id(s,t)</code>	delete definition of strategy 'id'/(s,t)
<code>:reset</code>	delete all term bindings, all strategies, reset syntax.
Debugging	
<code>:show</code>	show the current subject term
<code>:autoshow on off</code>	show the current subject term after each rewrite
<code>:binding id</code>	show term binding of id
<code>:bindings</code>	show all term bindings
<code>:showdef id</code>	show definitions of all strategies 'id'/(s,t)
<code>:showdef id(s,t)</code>	show definition of strategy 'id'/(s,t)
<code>:showast id(s,t)</code>	show ast of definition of strategy 'id'/(s,t)
Concrete Syntax	
<code>:syntax defname</code>	set the syntax to the sdf definition in 'defname'.
XTC	
<code>:xtc import pathname</code>	
Misc	
<code>:include file</code>	execute command in the script of 'file'
<code>:verbose int</code>	set the verbosity level (0-9)
<code>:clear</code>	clear the screen
<code>:exit</code>	exit the Stratego Shell
<code>:quit</code>	same as <code>:exit</code>
<code>:q</code>	same as <code>:exit</code>
<code>:about</code>	information about the Stratego Shell
<code>:help</code>	show this help information
stratego>	

3.5. Summary

Let's summarize what we have learned so far about Stratego programming.

First, a Stratego program is divided into modules, which reside in files with extension `.str` and have the following general form:

```

module mod0
imports libstrategolib mod1 mod2
signature
  sorts A B C
  constructors
    Foo : A -> B
    Bar : A
strategies
  main = io-wrap(foo)
  foo = id

```

Modules can import other modules and can define signatures for declaring the structure of terms and can define strategies, which we not really know much about yet. However, the `io-wrap` strategy can be used to handle the input and output of a Stratego program. This strategy is defined in the `libstrategolib` module, which provides an interface to the Stratego Library. The main module of a Stratego program should have a `main` strategy that defines the entry point of the program.

Next, a Stratego program is compiled to an executable program using the Stratego Compiler `strc`.

```
$ strc -i mod0 -la stratego-lib
```

The resulting executable applies the `main` strategy to command-line arguments turned into a list-of-strings term. The `io-wrap` strategy interprets these command-line arguments to handle input and output using standard command-line options.

Finally, the Stratego Shell can be used to invoke strategies interactively.

```
$ stratego-shell
stratego> id
()
stratego>
```

13.1.4 4. Term Rewriting

In this chapter we show how to implement term transformations using *term rewriting* in Stratego. In term rewriting a term is transformed by repeated application of *rewrite rules*.

4.1. Transformation with Rewrite Rules

To see how this works we take as example the language of propositional formulae, also known as Boolean expressions:

```
module prop
signature
  sorts Prop
  constructors
    False : Prop
    True  : Prop
    Atom  : String -> Prop
    Not   : Prop -> Prop
    And   : Prop * Prop -> Prop
    Or    : Prop * Prop -> Prop
    Impl  : Prop * Prop -> Prop
    Eq    : Prop * Prop -> Prop
```

Given this signature we can write terms such as `And(Impl(True(),False()),False())`, and `And(Atom("p"),False())`. Atoms are also known as proposition letters; they are the variables in propositional formulae. That is, the truth value of an atom should be provided in order to fully evaluate an expression. Here we will evaluate expressions as far as possible, a transformation also known as *constant folding*. We will do this using *rewrite rules* that define how to simplify a single operator application.

A *term pattern* is a term with *meta variables*, which are identifiers that are not declared as (nullary) constructors. For example, `And(x, True())` is a term pattern with variable `x`. Variables in term patterns are sometimes called *meta variables*, to distinguish them from variables in the source language being processed. For example, while atoms in the proposition expressions are variables from the point of view of the language, they are not variables from the perspective of a Stratego program.

A term pattern `p` *matches* with a term `t`, if there is a *substitution* that replaces the variables in `p` such that it becomes equal to `t`. For example, the pattern `And(x, True())` matches the term `And(Impl(True(),Atom("p")), True())` because replacing the variable `x` in the pattern by `Impl(True(),Atom("p"))` makes the pattern equal to the term. Note that `And(Atom("x"),True())` does *not* match the term `And(Impl(True(),Atom("p")), True())`, since the subterms `Atom("x")` and `Impl(True(),Atom("p"))` do not match.

An *unconditional rewrite rule* has the form `L : p1 -> p2`, where `L` is the name of the rule, `p1` is the left-hand side and `p2` the right-hand side term pattern. A rewrite rule `L : p1 -> p2` applies to a term `t` when the pattern `p1`

matches `t`. The result is the instantiation of `p2` with the variable bindings found during matching. For example, the rewrite rule

```
E : Eq(x, False()) -> Not(x)
```

rewrites the term `Eq(Atom("q"), False())` to `Not(Atom("q"))`, since the variable `x` is bound to the subterm `Atom("q")`.

Now we can create similar evaluation rules for all constructors of sort `Prop`:

```
module prop-eval-rules
imports prop
rules
  E : Not(True())      -> False()
  E : Not(False())     -> True()
  E : And(True(), x)   -> x
  E : And(x, True())   -> x
  E : And(False(), x)  -> False()
  E : And(x, False())  -> False()
  E : Or(True(), x)    -> True()
  E : Or(x, True())    -> True()
  E : Or(False(), x)   -> x
  E : Or(x, False())   -> x
  E : Impl(True(), x)  -> x
  E : Impl(x, True())  -> True()
  E : Impl(False(), x) -> True()
  E : Impl(x, False()) -> Not(x)
  E : Eq(False(), x)   -> Not(x)
  E : Eq(x, False())   -> Not(x)
  E : Eq(True(), x)    -> x
  E : Eq(x, True())    -> x
```

Note that all rules have the same name, which is allowed in Stratego.

Next we want to *normalize* terms with respect to a collection of rewrite rules. This entails applying all rules to all subterms until no more rules can be applied. The following module defines a rewrite system based on the rules for propositions above:

```
module prop-eval
imports libstrategolib prop-eval-rules
strategies
  main = io-wrap(eval)
  eval = innermost(E)
```

The module imports the Stratego Library (`libstrategolib`) and the module with the evaluation rules, and then defines the main strategy to apply `innermost(E)` to the input term. The `innermost` strategy from the library exhaustively applies its argument transformation to the term it is applied to, starting with *inner* subterms.

As an aside, we have now seen Stratego modules with `rules` and `strategies` sections. It's worth noting that a module can have any number of sections of either type, and that there is no actual semantic difference between the two section headings. In fact, either rewrite rules and/or strategy definitions can occur in either kind of section. Nevertheless, it often helps with making your transformations clearer to generally segregate rules and strategy definitions, and so both headings are allowed so you can punctuate your Stratego modules with them to improve readability.

In any case, we can now compile the above program:

```
$ strc -i prop-eval.str -la stratego-lib
```


This results in an executable `prop-eval` that can be used to evaluate Boolean expressions. For example, here are some applications of the program:

```
$ cat test1.prop
And(Impl(True(), And(False(), True())), True())

$ ./prop-eval -i test1.prop
False

$ cat test2.prop
And(Impl(True(), And(Atom("p"), Atom("q"))), Atom("p"))

$ ./prop-eval -i test2.prop
And(And(Atom("p"), Atom("q")), Atom("p"))
```

We can also import these definitions in the Stratego Shell, as illustrated by the following session:

```
$ stratego-shell
stratego> import prop-eval

stratego> !And(Impl(True(), And(False(), True())), True())
And(Impl(True(), And(False, True)), True)

stratego> eval
False

stratego> !And(Impl(True(), And(Atom("p"), Atom("q"))), Atom("p"))
And(Impl(True, And(Atom("p"), Atom("q"))), Atom("p"))

stratego> eval
And(And(Atom("p"), Atom("q")), Atom("p"))

stratego> :quit
And(And(Atom("p"), Atom("q")), Atom("p"))
$
```

The first command imports the `prop-eval` module, which recursively loads the evaluation rules and the library, thus making its definitions available in the shell. The `!` commands replace the current term with a new term. (This *build* strategy will be properly introduced in [Chapter 8](#).)

The next commands apply the `eval` strategy to various terms.

4.2. Running `prop-eval` in Spoofax/Eclipse

If you'd like to try out some of these Stratego examples in Spoofax/Eclipse, the first step is to define a *concrete syntax* for Boolean expressions that will parse to the sorts of *ATerms* that we have been working with. The [SDF3 Manual](#) provides the best introduction to how one might go about doing that, but here is the bulk of an SDF3 syntax definition that will allow us to represent any of the *ATerms* above:

```
context-free syntax

Prop.True  = <1>
Prop.False = <0>
Prop.Atom  = String
Prop.Not   = <!  
<Prop>>
Prop.And   = <<Prop> & <Prop>> {left}
Prop.Or    = <<Prop> | <Prop>> {left}
```

(continues on next page)

(continued from previous page)

```

Prop.Impl = [[Prop] -> [Prop]] {right}
Prop.Eq   = <<Prop> = <Prop>> {non-assoc}
Prop      = <(<Prop>)> {bracket}

String = ID

context-free priorities

Prop.Not
> Prop.And
> Prop.Or
> Prop.Impl
> Prop.Eq

```

With this grammar in place, the first two examples at the beginning of this chapter (just below the `prop` signature) can be expressed by `(1 -> 0) & 0` and `p & 0`, respectively. So you can see that the concrete syntax will actually make it much easier to construct the example expressions used throughout this manual.

If you'd like to see this in action in Spoofax/Eclipse, you can set up a language with the above grammar. Or you can clone the publicly-available [repository](#) containing most of the `prop` language examples from this manual.

Either way, you can place either of the above expressions in a file (`syntax/examples/sec4.1_A.spl` or `..._B.spl` in the repository) and visit it in Eclipse. Then if you select “Syntax > Show parsed AST” from the Spoofax menu, the parsed AST matching our first expressions above will pop up in the editor.

4.2.1 Using Editor Services to run a Stratego transformation

Naturally, we'd now like to run `prop-eval` in Spoofax/Eclipse. So we can take the `prop-eval-rules` module above and save it as `trans/prop-eval-rules.str`, with just one small change. Instead of `import prop` in the second line, we can say `import signatures/-`, since Spoofax has written out the signature implied by the grammar for us.

We're also going to have a small module `trans/prop-eval.str` to call the `prop-eval-rules`. It starts out rather similarly to the `prop-eval` for Stratego/XT; here's the first four lines:

```

module prop-eval
imports libstrategolib prop-eval-rules
strategies
  eval = innermost(E)

```

Note that we don't have the `main = io-wrap(eval)` line. For Stratego/XT, that was the sort of “glue” we needed to connect the execution environment with the basic `eval` strategy we've defined in Stratego. Similarly, a “glue” expression is needed in Spoofax/Eclipse as well. Because the Eclipse environment is more flexible, the necessary glue is rather more complicated; for now we needn't worry much about its details:

```

// Interface eval strategy with editor services and file system
do-eval: (selected, _, _, path, project-path) -> (filename, result)
  with filename := <guarantee-extension(|"eval.aterm")> path
  ; result      := <eval> selected

```

How do we now invoke this interface? That's where the Spoofax [Editor Services](#) (ESV) comes in. ESV is responsible, among other things, for the “Spoofax” menu item on the top bar of Eclipse. And you can add a new submenu, which we'll call “Manual”, to that menu with a little module `editor/Manual.esv` like this:

```
module Manual
menus
  menu: "Manual" (openeditor) (source)
    action: "prop-eval" = do-eval
```

Finally, we have to get Spoofax to see our new Stratego and ESV modules. We do this by importing them in the main Stratego and ESV files of the project. In the repository these are in `trans/spoofax_propositional_language.str` and `editor/Main.esv`, respectively. Their beginnings end up looking like:

```
module spoofax_propositional_language

imports

  completion/completion
  pp
  outline
  analysis
  prop-eval
rules // Debugging
```

and

```
module Main
imports
  Syntax
  Analysis
  Manual
language
```

Now at last we're ready to invoke the `eval` transformation. Make sure you have your project rebuilt cleanly. Visit a `.spl` file that has the expression you'd like to evaluate, such as `syntax/examples/sec4.1_test1.spl` containing `(1 -> 0 & 1) & 1`. Then select "Spoofax > Manual > prop-eval" from the menu bar to see the value (in this case `False()`). (There's also a `..._test2.spl` with `(1 -> p & q) & p` for the other example, and you can create your own files for some of the expressions in the Stratego Shell session shown above, if you like.)

4.3. Adding Rules to a Rewrite System

Next we extend the rewrite rules above to rewrite a Boolean expression to disjunctive normal form. A Boolean expression is in disjunctive normal form if it conforms to the following signature:

```
signature
  sorts Or And NAtom Atom
  constructors
    Or    : Or * Or -> Or
          : And -> Or
    And  : And * And -> And
          : NAtom -> And
    Not  : Atom -> NAtom
          : Atom -> NAtom
    Atom : String -> Atom
```

We use this signature only to describe what a disjunctive normal form is, not in an the actual Stratego program. This is not necessary, since terms conforming to the DNF signature are also `Prop` terms as defined before. For example, the disjunctive normal form of

```
And(Impl(Atom("r"), And(Atom("p"), Atom("q"))), Atom("p"))
```

is

```
Or(And(Not(Atom("r")), Atom("p")),
   And(And(Atom("p"), Atom("q")), Atom("p")))
```

Module `prop-dnf-rules` extends the rules defined in `prop-eval-rules` with rules to achieve disjunctive normal forms:

```
module prop-dnf-rules
imports prop-eval-rules
rules
  E : Impl(x, y) -> Or(Not(x), y)
  E : Eq(x, y)   -> And(Impl(x, y), Impl(y, x))

  E : Not(Not(x)) -> x

  E : Not(And(x, y)) -> Or(Not(x), Not(y))
  E : Not(Or(x, y))  -> And(Not(x), Not(y))

  E : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
  E : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
```

The first two rules rewrite implication (`Impl`) and equivalence (`Eq`) to combinations of `And`, `Or`, and `Not`. The third rule removes double negation. The fifth and sixth rules implement the well known DeMorgan laws. The last two rules define distribution of conjunction over disjunction.

We turn this set of rewrite rules into a compilable Stratego program in the same way as before:

```
module prop-dnf
imports libstrategolib prop-dnf-rules
strategies
  main = io-wrap(dnf)
  dnf = innermost(E)
```

compile it in the usual way

```
$ strc -i prop-dnf.str -la stratego-lib
```

so that we can use it to transform terms:

```
$ cat test3.prop
And(Impl(Atom("r"), And(Atom("p"), Atom("q"))), Atom("p"))
$ ./prop-dnf -i test3.prop
Or(And(Not(Atom("r")), Atom("p")), And(And(Atom("p"), Atom("q")), Atom("p")))
```

4.4. Using Spoofax Testing Language to run a Stratego Transformation

We can of course run `prop-dnf` in Spoofax/Eclipse in the same way as before. The `prop-dnf-rules` module is saved verbatim in `trans/prop-dnf-rules.str`, and the `prop-dnf` module becomes the following `trans/prop-dnf.str`:

```
module prop-dnf
imports libstrategolib prop-dnf-rules
```

(continues on next page)

(continued from previous page)

```

strategies
  dnf = innermost (E)

  // Interface dnf strategy with editor services and file system
  do-dnf: (selected, _, _, path, project-path) -> (filename, result)
    with filename := <guarantee-extension("dnf.aterm")> path
    ; result      := <dnf> selected
    
```

If you add a “prop-dnf” action to `editor/Manual.esv` calling `do-dnf` and rebuild the project, then you can visit, say, `syntax/examples.sec4.2_test3.spl` containing `(r -> p & q) & p` to produce exactly the DNF shown above.

However, we also want use this example to show another method of running Stratego strategies from the Eclipse IDE.

The [Spoofox Testing Language](#) (SPT) is a declarative language that provides for a full range of tests for a Spoofox language project. As such, it includes the ability to run an arbitrary Stratego strategy on the results of parsing an arbitrary piece of the language you’re working with.

So, we can just take our `test3` expression above and make it a part of an SPT test suite, which we will call `test/manual-suite.spt`:

```

module manual-suite
  language Spoofox-Propositional-Language

  test sec4_2_test3 [[
    (r -> p & q) & p
  ]] run dnf
    
```

Once we have saved this file, the tests run automatically. What does this mean? The file seems to be just “sitting there;” there’s no indication that anything is happening. That’s because this test we’ve just written succeeds. All we asked is that Spoofox run the `dnf` transformation on the results of parsing the test expression. It did that, and the transformation succeeded. So all is well, and no output is generated.

But of course, we want to check the result of the transformation as well. Fortunately, we know what we expect it to be. So we can change the test like so:

```

test sec4_2_test3_ex [[
  (r -> p & q) & p
]] run dnf to Or (And (Not (Atom ("r")), Atom ("p")),
                  And (And (Atom ("p"), Atom ("q")), Atom ("p")))
    
```

Now if there is no error or warning on this test then you know the `dnf` strategy produced the result shown in the `to` clause, and otherwise, the actual result will be shown in the error popup.

What if you *don’t* know what the expression is going to produce? Then you can just put a dummy expression like `Atom ("x")` in the `to` clause, and you will be sure to get an error. The error popup will show the actual transformation results. But beware! The results will be hard to read because of the annotations that Spoofox adds to track where in the source code each part of the AST originates. (For example, in the example above we get

```

Got: Or (And (Not (Atom ("r" {TermIndex ("test/manual-suite.spt", 1)})) {TermIndex
 ("test/manual-suite.spt", 2)}), Atom ("p" {TermIndex ("test/manual-suite.spt", 9)}))
 {TermIndex ("test/manual-suite.spt", 10)}), And (And (Atom ("p" {TermIndex ("test/manual-
 suite.spt", 3)})) {TermIndex ("test/manual-suite.spt", 4)}, Atom ("q" {TermIndex
 ("test/manual-suite.spt", 5)})) {TermIndex ("test/manual-suite.spt", 6)})) {TermIndex
 ("test/manual-suite.spt", 7)}, Atom ("p" {TermIndex ("test/manual-suite.spt", 9)}))
 {TermIndex ("test/manual-suite.spt", 10)}))
    
```

Nevertheless, with the editor outliner you can puzzle out what your transformation has done. The fact remains that it is most practical to put the actual expected result of the transformation in the `to` clause.

13.1.5 5. Rewriting Strategies

5.1. Limitations of Term Rewriting

In [Chapter 4](#) we saw how term rewriting can be used to implement transformations on programs represented by means of terms. Term rewriting involves exhaustively applying rules to subterms until no more rules apply. This requires a *strategy* for selecting the order in which subterms are rewritten. The `innermost` strategy introduced in [Chapter 4](#) applies rules automatically throughout a term from inner to outer terms, starting with the leaves. The nice thing about term rewriting is that there is no need to define traversals over the syntax tree; the rules express basic transformation steps and the strategy takes care of applying it everywhere. However, the complete normalization approach of rewriting turns out not to be adequate for program transformation, because rewrite systems for programming languages will often be non-terminating and/or non-confluent. In general, it is not desirable to apply all rules at the same time or to apply all rules under all circumstances.

Consider for example, the following extension of `prop-dnf-rules` with distribution rules to achieve conjunctive normal forms:

```
module prop-cnf
imports prop-dnf-rules
rules
  E : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
  E : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
strategies
  main = io-wrap(cnf)
  cnf = innermost(E)
```

This rewrite system is non-terminating because after applying one of the and-over-or distribution rules, the or-over-and distribution rules introduced here can be applied, and vice versa.

```
And(Or(Atom("p"), Atom("q")), Atom("r"))
->
Or(And(Atom("p"), Atom("r")), And(Atom("q"), Atom("r")))
->
And(Or(Atom("p"), And(Atom("q"), Atom("r"))),
    Or(Atom("r"), And(Atom("q"), Atom("r"))))
->
...
```

There are a number of solutions to this problem. We'll first discuss a couple of solutions within pure rewriting, and then show how programmable rewriting strategies can overcome the problems of these solutions.

5.1.1. Attempt 1: Remodularization

The non-termination of `prop-cnf` is due to the fact that the and-over-or and or-over-and distribution rules interfere with each other. This can be prevented by refactoring the module structure such that the two sets of rules are not present in the same rewrite system. For example, we could split module `prop-dnf-rules` into `prop-simplify` and `prop-dnf2` as follows:

```
module prop-simplify
imports prop-eval-rules
rules
```

(continues on next page)

(continued from previous page)

```
E : Impl(x, y) -> Or(Not(x), y)
E : Eq(x, y)   -> And(Impl(x, y), Impl(y, x))

E : Not(Not(x)) -> x

E : Not(And(x, y)) -> Or(Not(x), Not(y))
E : Not(Or(x, y))  -> And(Not(x), Not(y))
```

```
module prop-dnf2
imports prop-simplify
rules
  E : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
  E : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
strategies
  main = io-wrap(dnf)
  dnf  = innermost(E)
```

Now we can reuse the rules from `prop-simplify` without the `and-over-or` distribution rules to create a `prop-cnf2` for normalizing to conjunctive normal form:

```
module prop-cnf2
imports prop-simplify
rules
  E : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
  E : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
strategies
  main = io-wrap(cnf)
  cnf  = innermost(E)
```

Although this solves the non-termination problem, it is not an ideal solution. In the first place it is not possible to apply the two transformations in the same program. In the second place, extrapolating the approach to fine-grained selection of rules might require definition of a single rule per module.

5.1.2. Attempt 2: Functionalization

Another common solution to this kind of problem is to introduce additional constructors that achieve normalization under a restricted set of rules. That is, the original set of rules $p_1 \rightarrow p_2$ is transformed into rules of the form $f(p_1) \rightarrow p_2'$, where f is some new constructor symbol and the right-hand side of the rule also contains such new constructors. In this style of programming, constructors such as f are called *functions* and are distinguished from constructors. Normal forms over such rewrite systems are assumed to be free of these *function* symbols; otherwise the function would have an incomplete definition.

To illustrate the approach we adapt the DNF rules by introducing the function symbols `Dnf` and `DnfR`. (We ignore the evaluation rules in this example.)

```
module prop-dnf3
imports libstrategolib prop
signature
  constructors
    Dnf  : Prop -> Prop
    DnfR : Prop -> Prop
rules
  E : Dnf(Atom(x)) -> Atom(x)
```

(continues on next page)

(continued from previous page)

```

E : Dnf(Not(x))      -> DnfR(Not(Dnf(x)))
E : Dnf(And(x, y))   -> DnfR(And(Dnf(x), Dnf(y)))
E : Dnf(Or(x, y))    -> Or(Dnf(x), Dnf(y))
E : Dnf(Impl(x, y))  -> Dnf(Or(Not(x), y))
E : Dnf(Eq(x, y))    -> Dnf(And(Impl(x, y), Impl(y, x)))

E : DnfR(Not(Not(x))) -> x
E : DnfR(Not(And(x, y))) -> Or(Dnf(Not(x)), Dnf(Not(y)))
E : DnfR(Not(Or(x, y))) -> Dnf(And(Not(x), Not(y)))
D : DnfR(Not(x))      -> Not(x)

E : DnfR(And(Or(x, y), z)) -> Or(Dnf(And(x, z)), Dnf(And(y, z)))
E : DnfR(And(z, Or(x, y))) -> Or(Dnf(And(z, x)), Dnf(And(z, y)))
D : DnfR(And(x, y))      -> And(x, y)
strategies
main = io-wrap(dnf)
dnf  = innermost(E <+ D)

```

The `Dnf` function mimics the innermost normalization strategy by recursively traversing terms. The auxiliary transformation function `DnfR` is used to encode the distribution and negation rules. The `D` rules are *default* rules that are only applied if none of the `E` rules apply, as specified by the strategy expression `E <\+ D`.

In order to compute the disjunctive normal form of a term, we have to *apply* the `Dnf` function to it, as illustrated in the following application of the `prop-dnf3` program:

```

$ cat test1.dnf
Dnf(And(Impl(Atom("r"), And(Atom("p"), Atom("q"))), Atom("p")))

$ ./prop-dnf3 -i test1.dnf
Or(And(Not(Atom("r")), Atom("p")),
    And(And(Atom("p"), Atom("q")), Atom("p")))

```

If you're going to try to run this example in Spoofax/Eclipse, a few words of caution. First, it's easiest to just accumulate all of the different test modules as imports in your main language “.str” file. But if you do that, all of their rules will be in the same namespace. So you're going to want to use different identifiers (say “E3” and “D3”) in place of “E” and “D” in your `prop-dnf3.str` file. Also, the concrete syntax has no way to represent the “extra” function symbol `Dnf` that is used here, so you'll want to use alternate triggering strategies like

```

make-nf = innermost(E3 <+ D3)
dnf3 : x -> <make-nf> Dnf(x)

```

that wrap the input in `Dnf (...)` themselves.

For conjunctive normal form we can create a similar definition, which can now co-exist with the definition of DNF. Indeed, we could then simultaneously rewrite one subterm to DNF and the other to CNF.

```

E : DC(x) -> (Dnf(x), Cnf(x))

```

In the solution above, the original rules have been completely intertwined with the `Dnf` transformation. The rules for negation cannot be reused in the definition of normalization to conjunctive normal form. For each new transformation a new traversal function and new transformation functions have to be defined. Many additional rules had to be added to traverse the term to find the places to apply the rules. In the modular solution we had 5 basic rules and 2 additional rules for DNF and 2 rules for CNF, 9 in total. In the functionalized version we needed 13 rules *for each transformation*, that is 26 rules in total.

In the [example repository](#), you can find both the `dnf` and `cnf` strategies in `trans/prop-dnf3.str` and `trans/prop-cnf3.str`.

5.1.3 Running Stratego Transformations with the Spoofax Command-line Utilities

The Spoofax project offers an executable jar called Sunshine that allows several different Spoofax actions to be invoked from the command line. Let's say you have downloaded it to the path `SUNSHINE_JAR`. For convenience, let's say your project resides in the `SPOOFAX_PROJECT` directory, and Eclipse is installed in the `ECLIPSE_INSTALL` directory.

To use Sunshine to run a Stratego strategy, you must have a Spoofax menu item already set up to run it. But then you can invoke it on an arbitrary file from the command line like so:

```
java -jar $SUNSHINE_JAR transform -i <file> -n <menu-item> -p $SPOOFAX_PROJECT -l
↳$SPOOFAX_PROJECT -l "${ECLIPSE_INSTALL}/plugins"
```

(Note in this command, `<file>` and `<menu-item>` are placeholders that show you where to put in the actual filename and label of the menu item, respectively.) So the Sunshine jar doesn't really give us any new capabilities, but it could make running examples more convenient for you.

There is a similar executable jar that allows any SPT test file to be run from the command line. We won't go into the details here, because again, it doesn't do anything that you can't already do from inside Eclipse. (But it can be convenient for automated integration testing, for example.)

With this section, we've summarized all the ways that you can execute Stratego strategies in the Spoofax/Eclipse environment. Of them, placing tests in an SPT file generally seems to be the most convenient, because it involves editing the fewest files and places the expression and the expected results of the transformation together. Indeed, in the sample repository all of the remaining examples are implemented by means of `test/manual-suite.spt`. And that wraps up the Spoofax/Eclipse-specific information in this manual; there are more notes in the documentation of the example repository about specific implementation details of some of the later examples.

5.2. Programmable Rewriting Strategies

In general, there are two problems with the functional approach to encoding the control over the application of rewrite rules, when comparing it to the original term rewriting approach: traversal overhead and loss of separation of rules and strategies.

In the first place, the functional encoding incurs a large *overhead* due to the explicit specification of *traversal*. In pure term rewriting, the strategy takes care of traversing the term in search of subterms to rewrite. In the functional approach traversal is spelled out in the definition of the function, requiring the specification of many additional rules. A traversal rule needs to be defined for each constructor in the signature and for each transformation. The overhead for transformation systems for real languages can be inferred from the number of constructors for some typical languages:

```
language : constructors
Tiger    : 65
C        : 140
Java     : 140
COBOL    : 300 - 1200
```

In the second place, rewrite rules and the strategy that defines their application are completely *intertwined*. Another advantage of pure term rewriting is the separation of the specification of the rules and the strategy that controls their application. Intertwining these specifications makes it more difficult to *understand* the specification, since rules cannot be distinguished from the transformation they are part of. Furthermore, intertwining makes it impossible to *reuse* the rules in a different transformation.

Stratego introduced the paradigm of *programmable rewriting strategies with generic traversals*, a unifying solution in which application of rules can be carefully controlled, while incurring minimal traversal overhead and preserving separation of rules and strategies.

The following are the design criteria for strategies in Stratego:

- **Separation of rules and strategy:** Basic transformation rules can be defined separately from the strategy that applies them, such that they can be understood independently.
- **Rule selection:** A transformation can select the necessary set of rules from a collection (library) of rules.
- **Control:** A transformation can exercise complete control over the application of rules. This control may be fine-grained or course-grained depending on the application.
- **No traversal overhead:** Transformations can be defined without overhead for the definition of traversals.
- **Reuse of rules:** Rules can be reused in different transformations.
- **Reuse of traversal schemas:** Traversal schemas can be defined generically and reused in different transformations.

5.3. Idioms of Strategic Rewriting

In the next chapters we will examine the language constructs that Stratego provides for programming with strategies, starting with the low-level actions of building and matching terms. To get a feeling for the purpose of these constructs, we first look at a couple of typical idioms of strategic rewriting.

5.3.1. Cascading Transformations

The basic idiom of program transformation achieved with term rewriting is that of *cascading transformations*. Instead of applying a single complex transformation algorithm to a program, a number of small, independent transformations are applied in combination throughout a program or program unit to achieve the desired effect. Although each individual transformation step achieves little, the cumulative effect can be significant, since each transformation feeds on the results of the ones that came before it.

One common cascading of transformations is accomplished by exhaustively applying rewrite rules to a subject term. In Stratego the definition of a cascading normalization strategy with respect to rules R_1, \dots, R_n can be formalized using the `innermost` strategy that we saw before:

```
simplify = innermost (R1 <+ ... <+ Rn)
```

The argument strategy of `innermost` is a *selection* of rules. By giving *different* names to rules, we can control the selection used in each transformation. There can be multiple applications of `innermost` to different sets of rules, such that different transformations can co-exist in the same module without interference. Thus, it is now possible to develop a large library of transformation rules that can be called upon when necessary, without having to compose a rewrite system by cutting and pasting. For example, the following module defines the normalization of proposition formulae to both disjunctive and to conjunctive normal form:

```
module prop-laws
imports libstrategolib prop
rules

DefI : Impl(x, y) -> Or(Not(x), y)
DefE : Eq(x, y)   -> And(Impl(x, y), Impl(y, x))

DN   : Not(Not(x)) -> x

DMA  : Not(And(x, y)) -> Or(Not(x), Not(y))
DMO  : Not(Or(x, y))  -> And(Not(x), Not(y))

DAOL : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
DAOR : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
```

(continues on next page)

(continued from previous page)

```
DOAL : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
DOAR : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))

strategies

dnf = innermost (DefI <+ DefE <+ DAOL <+ DAOR <+ DN <+ DMA <+ DMO)
cnf = innermost (DefI <+ DefE <+ DOAL <+ DOAR <+ DN <+ DMA <+ DMO)

main-dnf = io-wrap(dnf)
main-cnf = io-wrap(cnf)
```

The rules are named, and for each strategy different selections from the ruleset are made.

The module even defines two main strategies, which allows us to use one module for deriving multiple programs. Using the `--main` option of `strc` we declare which strategy to invoke as main strategy in a particular program. Using the `-o` option we can give a different name to each derived program.

```
$ strc -i prop-laws.str -la stratego-lib --main main-dnf -o prop-dnf4
```

5.3.2. One-pass Traversals

Cascading transformations can be defined with other strategies as well, and these strategies need not be exhaustive, but can be simpler *one-pass traversals*. For example, constant folding of Boolean expressions only requires a simple one-pass bottom-up traversal. This can be achieved using the `bottomup` strategy according to the following scheme:

```
simplify = bottomup(repeat(R1 <+ ... <+ Rn))
```

The `bottomup` strategy applies its argument strategy to each subterm in a bottom-to-top traversal. The `repeat` strategy applies its argument strategy repeatedly to a term.

Module `prop-eval2` defines the evaluation rules for Boolean expressions and a strategy for applying them using this approach:

```
module prop-eval2
imports libstrategolib prop
rules
  Eval : Not(True())      -> False()
  Eval : Not(False())     -> True()
  Eval : And(True(), x)   -> x
  Eval : And(x, True())   -> x
  Eval : And(False(), x)  -> False()
  Eval : And(x, False())  -> False()
  Eval : Or(True(), x)    -> True()
  Eval : Or(x, True())    -> True()
  Eval : Or(False(), x)   -> x
  Eval : Or(x, False())   -> x
  Eval : Impl(True(), x)  -> x
  Eval : Impl(x, True())  -> True()
  Eval : Impl(False(), x) -> True()
  Eval : Impl(x, False()) -> Not(x)
  Eval : Eq(False(), x)   -> Not(x)
  Eval : Eq(x, False())   -> Not(x)
  Eval : Eq(True(), x)    -> x
  Eval : Eq(x, True())    -> x
```

(continues on next page)

(continued from previous page)

```
strategies
  main = io-wrap(eval)
  eval = bottomup(repeat(Eval))
```

The strategy `eval` applies these rules in a bottom-up traversal over a term, using the `bottomup(s)` strategy. At each sub-term, the rules are applied repeatedly until no more rule applies using the `repeat(s)` strategy. This is sufficient for the `Eval` rules, since the rules never construct a term with subterms that can be rewritten.

Another typical example of the use of one-pass traversals is *desugaring*, that is rewriting language constructs to more basic language constructs. Simple desugarings can usually be expressed using a single top-to-bottom traversal according to the scheme

```
simplify = topdown(try(R1 <+ ... <+ Rn))
```

The `topdown` strategy applies its argument strategy to a term and then traverses the resulting term. The `try` strategy tries to apply its argument strategy once to a term.

Module `prop-desugar` defines a number of desugaring rules for Boolean expressions, defining propositional operators in terms of others. For example, rule `DefN` defines `Not` in terms of `Impl`, and rule `DefI` defines `Impl` in terms of `Or` and `Not`. So not all rules should be applied in the same transformation or non-termination would result.

```
module prop-desugar
imports prop libstrategolib

rules

  DefN  : Not(x)      -> Impl(x, False())
  DefI  : Impl(x, y) -> Or(Not(x), y)
  DefE  : Eq(x, y)    -> And(Impl(x, y), Impl(y, x))
  DefO1 : Or(x, y)    -> Impl(Not(x), y)
  DefO2 : Or(x, y)    -> Not(And(Not(x), Not(y)))
  DefA1 : And(x, y)   -> Not(Or(Not(x), Not(y)))
  DefA2 : And(x, y)   -> Not(Impl(x, Not(y)))

  IDefI : Or(Not(x), y) -> Impl(x, y)

  IDefE : And(Impl(x, y), Impl(y, x)) -> Eq(x, y)

strategies

  desugar =
    topdown(try(DefI <+ DefE))

  impl-nf =
    topdown(repeat(DefN <+ DefA2 <+ DefO1 <+ DefE))

  main-desugar =
    io-wrap(desugar)

  main-inf =
    io-wrap(impl-nf)
```

The strategies `desugar` and `impl-nf` define two different desugaring transformation based on these rules. The `desugar` strategy gets rid of the implication and equivalence operators, while the `impl-nf` strategy reduces an expression to implicative normal-form, a format in which *only* implication (`Impl`) and `False()` are used.

A final example of a one-pass traversal is the `downup` strategy, which applies its argument transformation during a

traversal on the way down, and again on the way up:

```
simplify = downup(repeat(R1 <+ ... <+ Rn))
```

An application of this strategy is a more efficient implementation of constant folding for Boolean expressions:

```
eval = downup(repeat(Eval))
```

This strategy reduces terms such as

```
And(... big expression ..., False)
```

in one step (to `False()` in this case), while the `bottomup` strategy defined above would first evaluate the big expression.

5.3.3. Staged Transformations

Cascading transformations apply a number of rules one after another to an entire tree. But in some cases this is not appropriate. For instance, two transformations may be inverses of one another, so that repeatedly applying one and then the other would lead to non-termination. To remedy this difficulty, Stratego supports the idiom of *staged transformation*.

In staged computation, transformations are not applied to a subject term all at once, but rather in stages. In each stage, only rules from some particular subset of the entire set of available rules are applied. In the TAMPR program transformation system this idiom is called *sequence of normal forms*, since a program tree is transformed in a sequence of steps, each of which performs a normalization with respect to a specified set of rules. In Stratego this idiom can be expressed directly according to the following scheme:

```
strategies

simplify =
    innermost(A1 <+ ... <+ Ak)
    ; innermost(B1 <+ ... <+ B1)
    ; ...
    ; innermost(C1 <+ ... <+ Cm)
```

5.3.4. Local Transformations

In conventional program optimization, transformations are applied throughout a program. In optimizing imperative programs, for example, complex transformations are applied to entire programs. In GHC-style compilation-by-transformation, small transformation steps are applied throughout programs. Another style of transformation is a mixture of these ideas. Instead of applying a complex transformation algorithm to a program we use staged, cascading transformations to accumulate small transformation steps for large effect. However, instead of applying transformations throughout the subject program, we often wish to apply them locally, i.e., only to selected parts of the subject program. This allows us to use transformations rules that would not be beneficial if applied everywhere.

One example of a strategy which achieves such a transformation is

```
strategies

transformation =
    alltd(
        trigger-transformation
        ; innermost(A1 <+ ... <+ An)
    )
```

The strategy `alltd(s)` descends into a term until a subterm is encountered for which the transformation `s` succeeds. In this case the strategy `trigger-transformation` recognizes a program fragment that should be transformed. Thus, cascading transformations are applied locally to terms for which the transformation is triggered. Of course more sophisticated strategies can be used for finding application locations, as well as for applying the rules locally. Nevertheless, the key observation underlying this idiom remains: Because the transformations to be applied are local, special knowledge about the subject program at the point of application can be used. This allows the application of rules that would not be otherwise applicable.

13.1.6 6. Rules and Strategies

Pure term rewriting is not adequate for program transformation because of the lack of control over the application of rules. Attempts to encode such control within the pure rewriting paradigm lead to functionalized control by means of extra rules and constructors, at the expense of traversal overhead and at the loss of the separation of rules and strategies. By selecting the appropriate rules and strategy for a transformation, Stratego programmers can control the application of rules, while maintaining the separation of rules and strategies and keeping traversal overhead to a minimum.

We saw that many transformation problems can be solved by alternative strategies such as a one-pass bottom-up or top-down traversals. Others can be solved by selecting the rules that are applied in an innermost normalization, rather than all the rules in a specification. However, no fixed set of such alternative strategies will be sufficient for dealing with all transformation problems. Rather than providing one or a few fixed collection of rewriting strategies, Stratego supports the *composition* of strategies from basic building blocks with a few fundamental operators.

On this page we define the basic notions of rules and strategies, and we will see how new strategies and strategy combinators can be defined.

6.1. What is a rule?

A *rule* defines a transformation on terms. A *named rewrite rule* is a declaration of the form:

```
L : p1 -> p2
```

where `L` is the rule name, `p1` the left-hand side term pattern, and `p2` the right-hand side term pattern. A rule can be applied *through its name* to a term. It will transform the term if the term matches with `p1`, and will replace the term with `p2` instantiated with the variables bound during the match to `p1`. The application *fails* if the term does not match `p1`. Thus, a *transformation* is a *partial function from terms to terms*.

Let's look at an example. The `SwapArgs` rule swaps the sub-terms of the `Plus` constructor. Note that it is possible to introduce rules on the fly in the Stratego Shell.

```
stratego> SwapArgs : Plus(e1,e2) -> Plus(e2,e1)
```

Now we create a new term, and apply the `SwapArgs` rule to it by calling its name at the prompt. (The build `!t` of a term replaces the current term by `t`.)

```
stratego> !Plus(Var("a"),Int("3"))
Plus(Var("a"),Int("3"))
stratego> SwapArgs
Plus(Int("3"),Var("a"))
```

The application of `SwapArgs` fails when applied to a term to which the left-hand side does not match. For example, since the pattern `Plus(e1,e2)` does not match with a term constructed with `Times` the following application fails:

```
stratego> !Times(Int("4"),Var("x"))
Times(Int("4"),Var("x"))
```

(continues on next page)

(continued from previous page)

```
stratego> SwapArgs
command failed
```

A rule is applied at the *root* of a term, not at one of its subterms. Thus, the following application fails even though the term *contains* a `Plus` subterm:

```
stratego> !Times (Plus (Var ("a"), Int ("3")), Var ("x"))
Times (Plus (Var ("a"), Int ("3")), Var ("x"))
stratego> SwapArgs
command failed
```

Likewise, the following application only transforms the outermost occurrence of `Plus`, not the inner occurrence:

```
stratego> !Plus (Var ("a"), Plus (Var ("x"), Int ("42")))
Plus (Var ("a"), Plus (Var ("x"), Int ("42")))
stratego> SwapArgs
Plus (Plus (Var ("x"), Int ("42")), Var ("a"))
```

Finally, there may be multiple rules with the same name. This has the effect that all rules with that name will be tried in turn until one succeeds, or all fail. The rules are tried in some undefined order. This means that it only makes sense to define rules with the same name if they are mutually exclusive, that is, do not have overlapping left-hand sides. For example, we can extend the definition of `SwapArgs` with a rule for the `Times` constructor, as follows:

```
stratego> SwapArgs : Times (e1, e2) -> Times (e2, e1)
```

Now the rule can be applied to terms with a `Plus` *and* a `Times` constructor, as illustrated by the following applications:

```
stratego> !Times (Int ("4"), Var ("x"))
Times (Int ("4"), Var ("x"))
stratego> SwapArgs
Times (Var ("x"), Int ("4"))

stratego> !Plus (Var ("a"), Int ("3"))
Plus (Var ("a"), Int ("3"))
stratego> SwapArgs
Plus (Int ("3"), Var ("a"))
```

Later we will see that a rule is nothing more than a syntactical convention for a strategy definition.

6.2. What is a Strategy?

A rule defines a transformation, that is, a partial function from terms to terms. A *strategy expressions* is a combination of one or more transformations into a new transformation. So, a strategy expressions also defines a transformation, i.e., a partial function from terms to terms. Strategy *operators* are functions from transformations to transformations.

In the previous chapter we saw some examples of strategy expressions. Lets examine these examples in the light of our new definition. First of all, *rule names* are basic strategy expressions. If we import module `prop-laws`, we have at our disposal all rules it defines as basic strategies:

```
stratego> import prop-laws
stratego> !Impl (True (), Atom ("p"))
Impl (True, Atom ("p"))
stratego> DefI
Or (Not (True), Atom ("p"))
```

Next, given a collection of rules we can create more complex transformations by means of strategy operators. For example, the `innermost` strategy creates from a collection of rules a new transformation that exhaustively applies those rules.

```
stratego> !Eq(Atom("p"), Atom("q"))
Eq(Atom("p"), Atom("q"))

stratego> innermost(DefI <+ DefE <+ DAOL <+ DAOR <+ DN <+ DMA <+ DMO)

Or(Or(And(Not(Atom("p")), Not(Atom("q"))),
      And(Not(Atom("p")), Atom("p"))),
   Or(And(Atom("q"), Not(Atom("q"))),
      And(Atom("q"), Atom("p"))))
```

(Exercise: add rules to this composition that remove tautologies or false propositions.)

Here we see that the rules are first combined using the choice operator `<+` into a composite transformation, which is the argument of the `innermost` strategy.

The `innermost` strategy always succeeds (but may not terminate), but this is not the case for all strategies. For example `bottomup(DefI)` will not succeed, since it attempts to apply rule `DefI` to all sub-terms, which is clearly not possible. Thus, strategies extend the property of rules that they are *partial* functions from terms to terms.

Observe that in the composition `innermost(...)`, the term to which the transformation is applied is never mentioned. The ‘current term’, to which a transformation is applied is often implicit in the definition of a strategy. That is, there is no variable that is bound to the current term and then passed to an argument strategy. Thus, a strategy operator such as `innermost` is a function from transformations to transformations.

While strategies are functions, they are not necessarily *pure* functions. Strategies in Stratego may have side effects such as performing input/output operations. This is of course necessary in the implementation of basic tool interaction such as provided by `io-wrap`, but is also useful for debugging. For example, the `debug` strategy prints the current term, but does not transform it. We can use it to visualize the way that `innermost` transforms a term.

```
stratego> !Not(Impl(Atom("p"), Atom("q")))
Not(Impl(Atom("p"), Atom("q")))
stratego> innermost(debug(!"in:  "); (DefI <+ DefE <+ DAOL <+ DAOR <+ DN <+ DMA <+
↪DMO); debug(!"out: "))
in:  p
in:  Atom("p")
in:  q
in:  Atom("q")
in:  Impl(Atom("p"), Atom("q"))
out: Or(Not(Atom("p")), Atom("q"))
in:  p
in:  Atom("p")
in:  Not(Atom("p"))
in:  q
in:  Atom("q")
in:  Or(Not(Atom("p")), Atom("q"))
in:  Not(Or(Not(Atom("p")), Atom("q")))
out: And(Not(Not(Atom("p"))), Not(Atom("q")))
in:  p
in:  Atom("p")
in:  Not(Atom("p"))
in:  Not(Not(Atom("p")))
out: Atom("p")
in:  p
in:  Atom("p")
```

(continues on next page)

(continued from previous page)

```
in:  q
in:  Atom("q")
in:  Not(Atom("q"))
in:  And(Atom("p"), Not(Atom("q")))
And(Atom("p"), Not(Atom("q")))
```

This session nicely shows how innermost traverses the term it transforms. The `in:` lines show terms to which it attempts to apply a rule, the `out:` lines indicate when this was successful and what the result of applying the rule was. Thus, innermost performs a post-order traversal applying rules after transforming the sub-terms of a term. (Note that when applying `debug` to a string constant, the quotes are not printed.)

6.3. Strategy Definitions

Stratego programs are about defining transformations in the form of rules and strategy expressions that combine them. Just defining strategy *expressions* does not scale, however. Strategy *definitions* are the abstraction mechanism of Stratego and allow naming and parametrization of strategy expressions for reuse.

6.3.1. Simple Strategy Definition and Call

A simple strategy definition names a strategy expression. For instance, the following module defines a combination of rules (`dnf-rules`), and some strategies based on it:

```
module dnf-strategies
imports libstrategolib prop-dnf-rules
strategies

dnf-rules =
  DefI <+ DefE <+ DAOL <+ DAOR <+ DN <+ DMA <+ DMO

dnf =
  innermost(dnf-rules)

dnf-debug =
  innermost(debug(!"in: "); dnf-rules; debug(!"out: "))

main =
  io-wrap(dnf)
```

Note how `dnf-rules` is used in the definition of `dnf`, and `dnf` itself in the definition of `main`.

In general, a definition of the form

```
f = s
```

introduces a new transformation f , which can be invoked by calling f in a strategy expression, with the effect of executing strategy expression s . The expression should have no free variables. That is, all strategies called in s should be defined strategies. Simple strategy definitions just introduce names for strategy expressions. Still, such strategies have an argument, namely the implicit current term.

6.3.2. Parametrized Definitions

Strategy definitions with strategy and/or term parameters can be used to define transformation *schemas* that can be instantiated for various situations.

A parametrized strategy definition of the form

```
f(x1, ..., xn | y1, ..., ym) = s
```

introduces a user-defined operator f with n *strategy parameters* and m *term parameters*. Such a user-defined strategy operator can be called as $f(s_1, \dots, s_n | t_1, \dots, t_m)$ by providing it n strategy arguments and m term arguments. The meaning of such a call is the body s of the definition in which the actual arguments have been substituted for the formal parameters. Strategy arguments and term arguments can be left out of calls and definitions. That is, a call $f()$ without strategy and term arguments can be written as $f()$, or even just f . A call $f(s_1, \dots, s_n |)$ without term arguments can be written as $f(s_1, \dots, s_n)$. The same holds for strategy definitions.

In most cases the term parameters are simple variable names to which the argument will be bound when the strategy is called. However, it is also possible to use a *term pattern* in place of a term parameter, to which the argument will be matched. The strategy will fail when one or more term arguments could not be matched to their corresponding term pattern. These term patterns have the same freedoms as those used at left-hand side of a rule. For example, the following strategies act like a switch-case:

```
strategies
  to-english(|1) = !"one"
  to-english(|2) = !"two"
  to-english(|_) = !"many"
```

Or a more complex example:

```
strategies
  get-type(|<is-list>) = !"A list"
  get-type(|<not(is-list)>) = !"Not a list"
```

As we will see, strategies such as `innermost`, `topdown`, and `bottomup` are *not built into the language*, but are defined using strategy definitions in the language itself using more basic combinators, as illustrated by the following definitions (without going into the exact meaning of these definitions):

```
strategies
  try(s) = s <+ id
  repeat(s) = try(s; repeat(s))
  topdown(s) = s; all(topdown(s))
  bottomup(s) = all(bottomup(s)); s
```

Such parametrized strategy operators are invoked by providing arguments for the parameters. Specifically, strategy arguments are instantiated by means of strategy expressions. Wherever the argument is invoked in the body of the definition, the strategy expression is invoked. For example, in the previous chapter we saw the following instantiations of the `topdown`, `try`, and `repeat` strategies:

```
module prop-desugar
// ...
strategies
  desugar =
    topdown(try(DefI <+ DefE))

  impl-nf =
    topdown(repeat(DefN <+ DefA2 <+ DefO1 <+ DefE))
```

Multiple definitions with the same name but with a *different* numbers of parameters are treated as *different* strategy operators.

6.3.3. Local Definitions

Strategy definitions at top-level are visible everywhere. Sometimes it is useful to define a *local* strategy operator. This can be done using a `let` expression of the form `let d* in s end`, where `d*` is a list of definitions. For example, in the following strategy expression, the definition of `dnf-rules` is only visible in the instantiation of `innermost`:

```
let dnf-rules = DefI <+ DefE <+ DAOL <+ DAOR <+ DN <+ DMA <+ DMO
in innermost(dnf-rules)
end
```

The current version of Stratego does not support hidden strategy definitions at the module level. Such a feature is under consideration for a future version.

6.3.4. Extending Definitions

A Stratego program can introduce several rules with the same name. It is even possible to extend rules across modules. This is also possible for strategy definitions. If two strategy definitions have the same name and the same number of parameters, they effectively define a single strategy that tries to apply the bodies of the definitions in some undefined order. Thus, a definition of the form

```
strategies
  f = s1
  f = s2
```

entails that a call to `f` has the effect of first trying to apply `s1`, and if that fails applies `s2`, or the other way around. Thus, the definition above is either translated to

```
strategies
  f = s1 <+ s2
```

or to

```
strategies
  f = s2 <+ s1
```

6.4. Calling Primitives

Stratego provides combinators for composing transformations and basic operators for analyzing, creating and traversing terms. However, it does not provide built-in support for other types of computation such as input/output and process control. In order to make such functionality available to Stratego programmers, the language provides access to user-definable *primitive* strategies through the `prim` construct. For example, the following call to `prim` invokes the `SSL_printnl` native function:

```
stratego> prim("SSL_printnl", stdout(), ["foo", "bar"])
foobar
"
```

In general, a call `prim("f", s* | t*)` represents a call to a *primitive function* `f` with strategy arguments `s*` and term arguments `t*`. Note that the ‘current’ term is not passed automatically as argument.

This mechanism allows the incorporation of mundane tasks such as arithmetic, I/O, and other tasks not directly related to transformation, but necessary for the integration of transformations with the other parts of a transformation system.

6.4.1. Implementing Primitives

The Stratego Library provides all the primitives for I/O, arithmetic, string processing, and process control that are usually needed in Stratego programs. However, it is possible to add new primitives to a program as well. That requires linking with the compiled program a library that implements the function.

6.5. External Definitions

The Stratego Compiler is a *whole program compiler*. That is, the compiler includes all definitions from imported modules (transitively) into the program defined by the main module (the one being compiled). This is the reason that the compiler takes its time to compile a program. To reduce the compilation effort and the size of the resulting programs it is possible to create separately compiled *libraries* of Stratego definitions. The strategies that such a library provides are declared as *external* definitions. A declaration of the form

```
external f(s1 ... sn | x1 ... xm)
```

states that there is an externally defined strategy operator f with n strategy parameters and m term parameters. When compiling a program with external definitions, a library should be provided that implements these definitions.

The Stratego Library is provided as a separately compiled library. The `libstrategolib` module that we have been using in the example programs contains external definitions for all strategies in the library, as illustrated by the following excerpt:

```
module libstrategolib
// ...
strategies
  // ...
  external io-wrap(s)
  external bottomup(s)
  external topdown(s)
  // ...
```

When compiling a program using the library we used the `-la stratego-lib` option to provide the implementation of those definitions.

6.5.1. External Definitions cannot be Extended

Unlike definitions imported in the normal way, external definitions cannot be extended. If we try to compile a module extending an external definition, such as

```
module wrong
imports libstrategolib
strategies
  bottomup(s) = fail
```

compilation fails:

```
$ strc -i wrong.str
[ strc | info ] Compiling 'wrong.str'
error: redefining external definition: bottomup/1-0
[ strc | error ] Compilation failed (3.66 secs)
```

6.5.2. Creating Libraries

It is possible to create your own Stratego libraries as well. Currently that exposes you to a bit of compilation gibberish; in the future this may be incorporated in the Stratego compiler. Lets create a library for the rules and strategy definitions in the `prop-laws` module. We do this using the `--library` option to indicate that a library is being built, and the `-c` option to indicate that we are only interested in the generated C code.

```
$ strc -i prop-laws.str -c -o libproplib.rtree --library
[ strc | info ] Compiling 'proplib.str'
[ strc | info ] Front-end succeeded      : [user/system] = [4.71s/0.77s]
[ strc | info ] Abstract syntax in 'libproplib.rtree'
[ strc | info ] Concrete syntax in 'libproplib.str'
[ strc | info ] Export of externals succeeded : [user/system] = [2.02s/0.11s]
[ strc | info ] Back-end succeeded        : [user/system] = [6.66s/0.19s]
[ strc | info ] Compilation succeeded     : [user/system] = [13.4s/1.08s]
$ rm libproplib.str
```

The result of this compilation is a module `libproplib` that contains the external definitions of the module *and* those inherited from `libstrategolib`. (This module comes in to versions; one in concrete syntax `libproplib.str` and one in abstract syntax `libproplib.rtree`; for some obscure reason you should throw away the `.str` file.) Furthermore, the Stratego Compiler produces a C program `libproplib.c` with the implementation of the library. This C program should be turned into an object library using `libtool`, as follows:

```
$ libtool --mode=compile gcc -g -O -c libproplib.c -o libproplib.o -I <path/to/aterm-
→stratego/include>
...
$ libtool --mode=link gcc -g -O -o libproplib.la libproplib.lo
...
```

The result is a shared library `libproplib.la` that can be used in other Stratego programs. (TODO: the production of the shared library should really be incorporated into `strc`.)

6.5.3. Using Libraries

Programmers that want to use your library can now import the module with external definitions, instead of the original module.

```
module dnf-tool
imports libproplib
strategies
  main = main-dnf
```

This program can be compiled in the usual way, adding the new library to the libraries that should be linked against:

```
$ strc -i dnf-tool.str -la stratego-lib -la ./libproplib.la

$ cat test3.prop
And(Impl(Atom("r"), And(Atom("p"), Atom("q"))), ATom("p"))

$ ./dnf-tool -i test3.prop
Or(And(Not(Atom("r")), ATom("p")), And(And(Atom("p"), Atom("q")), ATom("p")))
```

To correctly deploy programs based on shared libraries requires some additional effort.

6.6. Dynamic Calls

Strategies can be called dynamically by name, i.e., where the name of the strategy is specified as a string. Such calls can be made using the `call` construct, which has the form:

```
call(f | s1, ..., sn | t1, ..., tn)
```

where `f` is a term that should evaluate to a string, which indicates the name of the strategy to be called, followed by a list of strategy arguments, and a list of term arguments.

Dynamic calls allow the name of the strategy to be computed at run-time. This is a rather recent feature of Stratego that was motivated by the need for call-backs from a separately compiled Stratego library combined with the computation of dynamic rule names. Otherwise, there is not yet much experience with the feature.

In the current version of Stratego it is necessary to ‘register’ a strategy to be able to call it dynamically. (In order to avoid deletion in case it is not called explicitly somewhere in the program.) Strategies are registered by means of a dummy strategy definition `DYNAMIC-CALLS` with calls to the strategies that should be called dynamically.

```
DYNAMICAL-CALLS = foo(id)
```

6.7. Summary

We have learned that rules and strategies define *transformations*, that is, functions from terms to terms that can fail, i.e., partial functions. Rule and strategy definitions introduce names for transformations. Parametrized strategy definitions introduce new strategy *operators*, functions that construct transformations from transformations, and support term patterns as parameters.

Primitive strategies are transformations that are implemented in some language other than Stratego (usually Java), and are called through the `prim` construct. External definitions define an interface to a separately compiled library of Stratego definitions. Dynamic calls allow the name of the strategy to be called to be computed as a string.

13.1.7 7. Strategy Combinators

We have seen the use of strategies to combine rules into complex transformations. Rather than providing a fixed set of high-level strategy operators such as `bottomup`, `topdown`, and `innermost`, Stratego provides a small set of basic combinators, that can be used to create a wide variety of strategies. In [Chapter 7](#) until [Chapter 10](#) we will introduce these combinators. In this chapter we start with a set of combinators for sequential composition and choice of strategies.

7.1. Identity and Failure

The most basic operations in Stratego are `id` and `fail`. The identity strategy `id` always succeeds and behaves as the identity function on terms. The failure strategy `fail` always fails. The operations have no side effects.

```
stratego> !Foo(Bar())
Foo(Bar)
stratego> id
Foo(Bar)
stratego> fail
command failed
```

7.2. Sequential composition

The sequential composition $s_1 ; s_2$ of the strategies s_1 and s_2 first applies the strategy s_1 to the subject term and then s_2 to the result of that first application. The strategy fails if either s_1 or s_2 fails.

Properties. Sequential composition is associative. Identity is a left and right unit for sequential composition; since `id` always succeeds and leaves the term alone, it has no additional effect to the strategy that it is composed with. Failure is a left zero for sequential composition; since `fail` always fails the next strategy will never be reached.

```
(s1; s2) ; s3 = s1; (s2; s3)

id; s = s

s; id = s

fail; s = fail
```

However, not for all strategies s we have that failure is a right zero for sequential composition:

```
s ; fail = fail    (is not a law)
```

Although the composition $s ; \text{fail}$ will always fail, the execution of s may have side effects that are not performed by `fail`. For example, consider printing a term in s .

Examples. As an example of the use of sequential composition consider the following rewrite rules.

```
stratego> A : P(Z(), x) -> x
stratego> B : P(S(x), y) -> P(x, S(y))
```

The following session shows the effect of first applying B and then A :

```
stratego> !P(S(Z()), Z())
P(S(Z), Z)
stratego> B
P(Z, S(Z))
stratego> A
S(Z)
```

Using the sequential composition of the two rules, this effect can be achieved ‘in one step’:

```
stratego> !P(S(Z()), Z())
P(S(Z), Z)
stratego> B; A
S(Z)
```

The following session shows that the application of a composition fails if the second strategy in the composition fails to apply to the result of the first:

```
stratego> !P(S(Z()), Z())
P(S(Z), Z)
stratego> B; B
command failed
```

Choosing between rules to apply is achieved using one of several *choice* combinators, all of which are based on the guarded choice combinator. The common approach is that failure to apply one strategy leads to backtracking to an alternative strategy.

Deterministic Choice (Left Choice)

The left choice or deterministic choice $s1 \text{ <+ } s2$ tries to apply $s1$ and $s2$ in that order. That is, it first tries to apply $s1$, and if that succeeds the choice succeeds. However, if the application of $s1$ fails, $s2$ is applied to *the original term*.

Properties. Left choice is associative. Identity is a left zero for left choice; since `id` always succeeds, the alternative strategy will never be tried. Failure is a left and right unit for left choice; since `fail` always fails, the choice will always backtrack to the alternative strategy, and use of `fail` as alternative strategy is pointless.

```
(s1 <+ s2) <+ s3 = s1 <+ (s2 <+ s3)

id <+ s = id

fail <+ s = s

s <+ fail = s
```

However, identity is not a right zero for left choice. That is, not for all strategies s we have that

```
s <+ id = s      (is not a law)
```

The expression $s \text{ <+ } id$ always succeeds, even (especially) in the case that s fails, in which case the right-hand side of the equation fails of course.

Local Backtracking. The left choice combinator is a *local backtracking* combinator. That is, the choice is committed once the left-hand side strategy has succeeded, even if the continuation strategy fails. This is expressed by the fact that the property

```
(s1 <+ s2); s3 = (s1; s3) <+ (s2; s3)      (is not a law)
```

does *not* hold for all $s1$, $s2$, and $s3$. The difference is illustrated by the following applications:

```
stratego> !P(S(Z()), Z())
P(S(Z), Z)
stratego> (B <+ id); B
command failed

stratego> !P(S(Z()), Z())
P(S(Z), Z)
stratego> (B <+ id)
P(Z, S(Z))
stratego> B
command failed

stratego> (B; B) <+ (id; B)
P(Z, S(Z))
```

In the application of $(B \text{ <+ } id); B$, the first application of B succeeds after which the choice is committed. The subsequent application of B then fails. This equivalent to first applying $(B \text{ <+ } id)$ and then applying B to the result. The application of $(B; B) \text{ <+ } (id; B)$, however, is successful; the application of $B; B$ fails, after which the choice backtracks to $id; B$, which succeeds.

Choosing between Transformations. The typical use of left choice is to create a composite strategy trying one from several possible transformations. If the strategies that are composed are mutually exclusive, that is, don't succeed for the same terms, their sum is a transformation that (deterministically) covers a larger set of terms. For example, consider the following two rewrite rules:


```
stratego> PlusAssoc : Plus(Plus(e1, e2), e3) -> Plus(e1, Plus(e2, e3))
stratego> PlusZero  : Plus(Int("0"), e) -> e
```

These rules are mutually exclusive, since there is no term that matches the left-hand sides of both rules. Combining the rules with left choice into `PlusAssoc <+ PlusZero` creates a strategy that transforms terms matching both rules as illustrated by the following applications:

```
stratego> !Plus(Int("0"), Int("3"))
Plus(Int("0"), Int("3"))

stratego> PlusAssoc
command failed
stratego> PlusAssoc <+ PlusZero
Int("3")

stratego> !Plus(Plus(Var("x"), Int("42")), Int("3"))
Plus(Plus(Var("x"), Int("42")), Int("3"))

stratego> PlusZero
command failed
stratego> PlusAssoc <+ PlusZero
Plus(Var("x"), Plus(Int("42"), Int("3")))
```

Ordering Overlapping Rules. When two rules or strategies are mutually exclusive the order of applying them does not matter. In cases where strategies are overlapping, that is, succeed for the same terms, the order becomes crucial to determining the semantics of the composition. For example, consider the following rewrite rules reducing applications of `Mem`:

```
stratego> Mem1 : Mem(x, []) -> False()
stratego> Mem2 : Mem(x, [x|xs]) -> True()
stratego> Mem3 : Mem(x, [y|ys]) -> Mem(x, ys)
```

Rules `Mem2` and `Mem3` have overlapping left-hand sides. Rule `Mem2` only applies if the first argument is equal to the head element of the list in the second argument. Rule `Mem3` applies always if the list in the second argument is non-empty.

```
stratego> !Mem(1, [1, 2, 3])
Mem(1, [1, 2, 3])
stratego> Mem2
True
stratego> !Mem(1, [1, 2, 3])
Mem(1, [1, 2, 3])
stratego> Mem3
Mem(1, [2, 3])
```

In such situations, depending on the order of the rules, different results are produced. (The rules form a non-confluent rewriting system.) By ordering the rules as `Mem2 <\+ Mem3`, rule `Mem2` is tried before `Mem3`, and we have a deterministic transformation strategy.

Try: A useful application of `<+` in combination with `id` is the reflexive closure of a strategy `s`:

```
try(s) = s <+ id
```

The user-defined strategy combinator `try` tries to apply its argument strategy `s`, but if that fails, just succeeds using `id`.

Sometimes it is not desirable to backtrack to the alternative specified in a choice. Rather, after passing a *guard*, the choice should be committed. This can be expressed using the *guarded left choice* operator `s1 < s2 + s3`. If `s1`

succeeds s_2 is applied, else s_3 is applied. If s_2 fails, the complete expression fails; no backtracking to s_3 takes place.

Properties. This combinator is a generalization of the left choice combinator $<+$.

```
s1 <+ s2 = s1 < id + s2
```

The following laws make clear that the ‘branches’ of the choice are selected by the success or failure of the guard:

```
id < s2 + s3 = s2
fail < s2 + s3 = s3
```

If the right branch always fails, the construct reduces to the sequential composition of the guard and the left branch.

```
s1 < s2 + fail = s1; s2
```

Guarded choice is not associative:

```
(s1 < s2 + s3) < s4 + s5 = s1 < s2 + (s3 < s4 + s5)    (not a law)
```

To see why consider the possible traces of these expressions. For example, when s_1 and s_2 succeed subsequently, the left-hand side expression calls s_4 , while the right-hand side expression does not.

However, sequential composition distributes over guarded choice from left *and* right:

```
(s1 < s2 + s3); s4 = s1 < (s2; s4) + (s3; s4)
s0; (s1 < s2 + s3) = (s0; s1) < s2 + s3
```

Examples. The guarded left choice operator is most useful for the implementation of higher-level control-flow strategies. For example, the *negation* $\text{not}(s)$ of a strategy s , succeeds if s fails, and fails when it succeeds:

```
not(s) = s < fail + id
```

Since failure discards the effect of a (successful) transformation, this has the effect of testing whether s succeeds. So we have the following laws for not :

```
not(id) = fail
not(fail) = id
```

However, side effects performed by s are not undone, of course. Therefore, the following equation does *not* hold:

```
not(not(s)) = s    (not a law)
```

Another example of the use of guarded choice is the `restore-always` combinator:

```
restore-always(s, r) = s < r + (r; fail)
```

It applies a ‘restore’ strategy r after applying a strategy s , even if s fails, and preserves the success/failure behavior of s . Since `fail` discards the transformation effect of r , this is mostly useful for ensuring that some side-effecting operation is done (or undone) after applying s .

For other applications of guarded choice, Stratego provides syntactic sugar.

The guarded choice combinator is similar to the traditional if-then-else construct of programming languages. The difference is that the ‘then’ branch applies to the result of the application of the condition. Stratego’s `if s1 then s2 else s3 end` construct is more like the traditional construct since both branches apply to the original term. The condition strategy is only used to test if it succeeds or fails, but its transformation effect is undone. However,

the condition strategy `s1` is still applied to the current term. The `if s1 then s2 end` strategy is similar; if the condition fails, the strategy succeeds.

Properties. The `if-then-else-end` strategy is just syntactic sugar for a combination of guarded choice and the `where` combinator:

```
if s1 then s2 else s3 end = where(s1) < s2 + s3
```

The strategy `where(s)` succeeds if `s` succeeds, but returns the original subject term. The implementation of the `where` combinator will be discussed in [Chapter 8](#). The following laws show that the branches are selected by success or failure of the condition:

```
if id then s2 else s3 end = s2
if fail then s2 else s3 end = s3
```

The `if-then-end` strategy is an abbreviation for the `if-then-else-end` with the identity strategy as right branch:

```
if s1 then s2 end = where(s1) < s2 + id
```

Examples. The *inclusive or* `or(s1, s2)` succeeds if one of the strategies `s1` or `s2` succeeds, but guarantees that both are applied, in the order `s1` first, then `s2`:

```
or(s1, s2) =
  if s1 then try(where(s2)) else where(s2) end
```

This ensures that any side effects are always performed, in contrast to `s1 <\+ s2`, where `s2` is only executed if `s1` fails. (Thus, left choice implements a short circuit Boolean or.)

Similarly, the following `and(s1, s2)` combinator is the non-short circuit version of Boolean conjunction:

```
and(s1, s2) =
  if s1 then where(s2) else where(s2); fail end
```

The `switch` construct is an *n*-ary branching construct similar to its counter parts in other programming languages. It is defined in terms of guarded choice. The `switch` construct has the following form:

```
switch s0
  case s1 : s1'
  case s2 : s2'
  ...
  otherwise : sdef
end
```

The `switch` first applies the `s0` strategy to the current term `t` resulting in a term `t'`. Then it tries the cases in turn applying each `si` to `t'`. As soon as this succeeds the corresponding case is selected and `si'` is applied to the `t`, the term to which the `switch` was applied. If none of the cases applies, the default strategy `sdef` from the `otherwise` is applied.

Properties. The `switch` construct is syntactic sugar for a nested `if-then-else`:

```
{x : where(s0 => x);
  if <s1> x
  then s1'
  else if <s2> x
    then s2'
    else if ...
```

(continues on next page)

(continued from previous page)

```

        then ...
        else sdef
        end
    end
end}

```

This translation uses a couple of Stratego constructs that we haven't discussed so far.

Examples. TODO

7.3. Non-deterministic Choice

The deterministic left choice operator prescribes that the left alternative should be tried before the right alternative, and that the latter is only used if the first fails. There are applications where it is not necessary to define the order of the alternatives. In those cases non-deterministic choice can be used.

The non-deterministic choice operator $s1 + s2$ chooses one of the two strategies $s1$ or $s2$ to apply, such that the one it chooses succeeds. If both strategies fail, then the choice fails as well. Operationally the choice operator first tries one strategy, and, if that fails, tries the other. The order in which this is done is undefined, i.e., arbitrarily chosen by the compiler.

The $+$ combinator is used to model modular composition of rewrite rules and strategies with the same name. Multiple definitions with the same name, are merged into a single definition with that name, where the bodies are composed with $+$. The following transformation illustrates this:

```
f = s1  f = s2    ==>    f = s1 + s2
```

This transformation is somewhat simplified; the complete transformation needs to take care of local variables and parameters.

While the $+$ combinator is used internally by the compiler for this purpose, programmers are advised *not* to use this combinator, but rather use the left choice combinator $<+$ to avoid surprises.

Repeated application of a strategy can be achieved with recursion. There are two styles for doing this; with a recursive definition or using the fixpoint operator `rec`. A recursive definition is a normal strategy definition with a recursive call in its body.

```
f(s) = ... f(s) ...
```

Another way to define recursion is using the fixpoint operator `rec x(s)`, which recurses on applications of x within s . For example, the definition

```
f(s) = rec x(... x ...)
```

is equivalent to the one above. The advantage of the `rec` operator is that it allows the definition of an unnamed strategy expression to be recursive. For example, in the definition

```
g(s) = foo; rec x(... x ...); bar
```

the strategy between `foo` and `bar` is a recursive strategy that does *not* recurse to $g(s)$.

Originally, the `rec` operator was the only way to define recursive strategies. It is still in the language in the first place because it is widely used in many existing programs, and in the second place because it can be a concise expression of a recursive strategy, since call parameters are not included in the call. Furthermore, all free variables remain in scope.

Examples. The `repeat` strategy applies a transformation s until it fails. It is defined as a recursive definition using `try` as follows:

```
try(s)      = s <+ id
repeat(s)   = try(s; repeat(s))
```

An equivalent definition using `rec` is:

```
repeat(s) = rec x(try(s; x))
```

The following Stratego Shell session illustrates how it works. We first define the strategies:

```
stratego> try(s) = s <+ id
stratego> repeat(s) = try(s; repeat(s))
stratego> A : P(Z(), x) -> x
stratego> B : P(S(x), y) -> P(x, S(y))
```

Then we observe that the repeated application of the individual rules A and B reduces terms:

```
stratego> !P(S(Z()), Z())
P(S(Z), Z)
stratego> B
P(Z, S(Z))
stratego> A
S(Z)
```

We can automate this using the `repeat` strategy, which will repeat the rules as often as possible:

```
stratego> !P(S(Z()), Z())
P(S(Z), Z)
stratego> repeat(A <+ B)
S(Z)

stratego> !P(S(S(S(Z()))), Z())
P(S(S(S(Z))), Z)
stratego> repeat(A <+ B)
S(S(S(Z)))
```

To illustrate the intermediate steps of the transformation we can use `debug` from the Stratego Library.

```
stratego> import libstratego-lib
stratego> !P(S(S(S(Z()))), Z())
P(S(S(S(Z))), Z)
stratego> repeat(debug; (A <+ B))
P(S(S(S(Z))), Z)
P(S(S(Z)), S(Z))
P(S(Z), S(S(Z)))
P(Z, S(S(S(Z))))
S(S(S(Z)))
S(S(S(Z)))
```

A Library of Iteration Strategies. Using sequential composition, choice, and recursion a large variety of iteration strategies can be defined. The following definitions are part of the Stratego Library (in module `strategy/iteration`).

```
repeat(s) =
  rec x(try(s; x))

repeat(s, c) =
  (s; repeat(s, c)) <+ c
```

(continues on next page)

(continued from previous page)

```
repeat1(s, c) =
  s; (repeat1(s, c) <+ c)

repeat1(s) =
  repeat1(s, id)

repeat-until(s, c) =
  s; if c then id else repeat-until(s, c) end

while(c, s) =
  if c then s; while(c, s) end

do-while(s, c) =
  s; if c then do-while(s, c) end
```

The following equations describe some relations between these strategies:

```
do-while(s, c) = repeat-until(s, not(c))

do-while(s, c) = s; while(c, s)
```

13.1.8 8. Creating and Analyzing Terms

In previous chapters we have presented rewrite rules as basic transformation steps. However, rules are not really atomic transformation actions. To see this, consider what happens when the rewrite rule

is applied. First it matches the subject term against the pattern `And(Or(x, y), z)` in the left-hand side. This means that a substitution for the variables `x`, `y`, and `z` is sought, that makes the pattern equal to the subject term. If the match fails, the rule fails. If the match succeeds, the pattern `Or(And(x, z), And(y, z))` on the right-hand side is instantiated with the bindings found during the match of the left-hand side. The instantiated term then replaces the original subject term. Furthermore, the rule limits the scope of the variables occurring in the rule. That is, the variables `x`, `y`, `z` are local to this rule. After the rule is applied the bindings to these variables are invisible again.

Thus, rather than considering rules as the atomic actions of transformation programs, Stratego provides their constituents, that is building terms from patterns and matching terms against patterns, as atomic actions, and makes these available to the programmer. In this chapter, you will learn these basic actions and their use in the composition of more complex operations such as various flavors of rewrite rules.

8.1. Building Terms

The build operation `!p` replaces the subject term with the instantiation of the pattern `p` using the bindings from the environment to the variables occurring in `p`. For example, the strategy `!Or(And(x, z), And(y, z))` replaces the subject term with the instantiation of `Or(And(x, z), And(y, z))` using bindings to variables `x`, `y` and `z`.

```
stratego> !Int("10")
Int("10")
stratego> !Plus(Var("a"), Int("10"))
Plus(Var("a"), Int("10"))
```

It is possible to build terms with variables. We call this building a term pattern. A pattern is a term with *meta-variables*. The current term is replaced by an instantiation of pattern `p`.

```
stratego> :binding e
e is bound to Var("b")
stratego> !Plus(Var("a"), e)
Plus(Var("a"), Var("b"))
stratego> !e
Var("b")
```

8.2. Matching Terms

Pattern matching allows the analysis of terms. The simplest case is matching against a literal term. The match operation `?t` matches the subject term against the term `t`.

```
Plus(Var("a"), Int("3"))
stratego> ?Plus(Var("a"), Int("3"))
stratego> ?Plus(Int("3"), Var("b"))
command failed
```

Matching against a term *pattern* with variables binds those variables to (parts of) the current term. The match strategy `?x` compares the current term (`t`) to variable `x`. It binds variable `x` to term `t` in the environment. A variable can only be bound once, or to the same term.

```
Plus(Var("a"), Int("3"))
stratego> ?e
stratego> :binding e
e is bound to Plus(Var("a"), Int("3"))
stratego> !Int("17")
stratego> ?e
command failed
```

The general case is matching against an arbitrary term pattern. The match strategy `?p` compares the current term to a pattern `p`. It will add bindings for the variables in pattern `p` to the environment. The wildcard `_` in a match will match any term.

```
Plus(Var("a"), Int("3"))
stratego> ?Plus(e, _)
stratego> :binding e
e is bound to Var("a")
Plus(Var("a"), Int("3"))
```

Patterns may be non-linear. Multiple occurrences of the same variable can occur and each occurrence matches the same term.

```
Plus(Var("a"), Int("3"))
stratego> ?Plus(e, e)
command failed
stratego> !Plus(Var("a"), Var("a"))
stratego> ?Plus(e, e)
stratego> :binding e
e is bound to Var("a")
```

Non-linear pattern matching is a way to test equality of terms. Indeed the equality predicates from the Stratego Library are defined using non-linear pattern matching:

```
equal = ?(x, x)
equal(|x) = ?x
```

The `equal` strategy tests whether the current term is a pair of the same terms. The `equal(|x)` strategy tests whether the current term is equal to the argument term.

```
stratego> equal = ?(x, x)
stratego> !("a", "a")
("a", "a")
stratego> equal
("a", "a")
stratego> !("a", "b")
("a", "b")
stratego> equal
command failed

stratego> equal(|x) = ?x
stratego> !Foo(Bar())
Foo(Bar)
stratego> equal(|Foo(Baz()))
command failed
stratego> equal(|Foo(Bar()))
Foo(Bar)
```

8.3. Implementing Rewrite Rules

Match and build are first-class citizens in Stratego, which means that they can be used and combined just like any other strategy expressions. In particular, we can implement rewrite rules using these operations, since a rewrite rule is basically a match followed by a build. For example, consider the following combination of match and build:

```
Plus(Var("a"), Int("3"))
stratego> ?Plus(e1, e2); !Plus(e2, e1)
Plus(Int("3"), Var("a"))
```

This combination first recognizes a term, binds variables to the pattern in the match, and then replaces the current term with the instantiation of the build pattern. Note that the variable bindings are propagated from the match to the build.

Stratego provides syntactic sugar for various combinations of match and build. We'll explore these in the rest of this chapter.

8.3.1. Anonymous Rewrite Rule

An *anonymous rewrite rule* $(p1 \rightarrow p2)$ transforms a term matching $p1$ into an instantiation of $p2$. Such a rule is equivalent to the sequence `?p1; !p2`.

```
Plus(Var("a"), Int("3"))
stratego> (Plus(e1, e2) -> Plus(e2, e1))
Plus(Int("3"), Var("a"))
```

8.3.2. Term variable scope

Once a variable is bound it cannot be rebound to a different term. Thus, when we have applied an anonymous rule once, its variables are bound and the next time it is applied it only succeeds for the same term. For example, in the next session the second application of the rule fails, because `e2` is bound to `Int("3")` and does not match with `Var("b")`.


```
stratego> !Plus(Var("a"), Int("3"))
Plus(Var("a"), Int("3"))
stratego> (Plus(e1, e2) -> Plus(e2, e1))
Plus(Int("3"), Var("a"))

stratego> :binding e1
e1 is bound to Var("a")
stratego> :binding e2
e2 is bound to Int("3")

stratego> !Plus(Var("a"), Var("b"))
Plus(Var("a"), Var("b"))
stratego> (Plus(e1, e2) -> Plus(e2, e1))
command failed
```

To use a variable name more than once Stratego provides *term variable scope*. A scope $\{x_1, \dots, x_n : s\}$ locally undefines the variables x_i . That is, the binding to a variable x_i outside the scope is not visible inside it, nor is the binding to x_i inside the scope visible outside it. For example, to continue the session above, if we wrap the anonymous swap rule in a scope for its variables, it can be applied multiple times.

```
stratego> !Plus(Var("a"), Int("3"))
Plus(Var("a"), Int("3"))
stratego> {e3, e4 : (Plus(e3, e4) -> Plus(e4, e3))}
Plus(Var("a"), Int("3"))
stratego> :binding e3
e3 is not bound to a term

stratego> !Plus(Var("a"), Var("b"))
Plus(Var("a"), Var("b"))
stratego> {e3, e4 : (Plus(e3, e4) -> Plus(e4, e3))}
Plus(Var("b"), Var("a"))
```

Of course we can name such a scoped rule using a strategy definition, and then invoke it by its name:

```
stratego> SwapArgs = {e1, e2 : (Plus(e1, e2) -> Plus(e2, e1))}
stratego> !Plus(Var("a"), Int("3"))
Plus(Var("a"), Int("3"))
stratego> SwapArgs
Plus(Int("3"), Var("a"))
```

8.3.3. Implicit Variable Scope

When using match and build directly in a strategy definition, rather than in the form of a rule, the definition contains free variables. Strictly speaking such variables should be declared using a scope, that is one should write

```
SwapArgs = {e1, e2 : (Plus(e1, e2) -> Plus(e2, e1))}
```

However, since declaring all variables at the top of a definition is distracting and does not add much to the definition, such a scope declaration can be left out. Thus, one can write

```
SwapArgs = (Plus(e1, e2) -> Plus(e2, e1))
```

instead. The scope is automatically inserted by the compiler. This implies that there is no global scope for term variables. Of course, variables in inner scopes should be declared where necessary. In particular, note that variable scope is *not* inserted for strategy definitions in a let binding, such as

```
let SwapArgs = (Plus(e1,e2) -> Plus(e2,e1)) in ... end
```

While the variables are bound in the enclosing definition, they are not restricted to `SwapArgs` in this case, since in a `let` you typically want to use bindings to variables in the enclosing code.

8.3.4. Where

Often it is useful to apply a strategy only to test whether some property holds or to compute some auxiliary result. For this purpose, Stratego provides the `where(s)` combinator, which applies `s` to the current term, but restores that term afterwards. Any bindings to variables are kept, however.

```
Plus(Int("14"),Int("3"))
stratego> where(?Plus(Int(i),Int(j)); <addS>(i,j) => k)
Plus(Int("14"),Int("3"))
stratego> :binding i
i is bound to "14"
stratego> :binding k
k is bound to "17"
```

With the match and build constructs `where(s)` is in fact just syntactic sugar for $\{x: ?x; s; !x\}$ with `x` a fresh variable not occurring in `s`. Thus, the current subject term is *saved* by binding it to a new variable `x`, then the strategy `s` is applied, and finally, the original term is *restored* by building `x`.

We saw the use of `where` in the definition of `if-then-else` in [Chapter 7](#).

8.3.5. Conditional rewrite rule

A simple rewrite rule succeeds if the match of the left-hand side succeeds. Sometimes it is useful to place additional requirements on the application of a rule, or to compute some value for use in the right-hand side of the rule. This can be achieved with *conditional rewrite rules*. A conditional rule `L: p1 -> p2 where s` is a simple rule extended with an additional computation `s` which should succeed in order for the rule to apply. The condition can be used to test properties of terms in the left-hand side, or to compute terms to be used in the right-hand side. The latter is done by binding such new terms to variables used in the right-hand side.

For example, the `EvalPlus` rule in the following session uses a condition to compute the sum of `i` and `j`:

```
stratego> EvalPlus: Plus(Int(i),Int(j)) -> Int(k) where !(i,j); addS; ?k
stratego> !Plus(Int("14"),Int("3"))
Plus(Int("14"),Int("3"))
stratego> EvalPlus
Int("17")
```

A conditional rule can be desugared similarly to an unconditional rule. That is, a conditional rule of the form

```
L : p1 -> p2 where s
```

is syntactic sugar for

```
L = ?p1; where(s); !p2
```

Thus, after the match with `p1` succeeds the strategy `s` is applied to the subject term. Only if the application of `s` succeeds, is the right-hand side `p2` built. Note that since `s` is applied within a `where`, the build `!p2` is applied to the original subject term; only *variable bindings* computed within `s` can be used in `p2`.

As an example, consider the following constant folding rule, which reduces an addition of two integer constants to the constant obtained by computing the addition.

```
EvalPlus : Add(Int(i), Int(j)) -> Int(k) where ! (i, j); addS; ?k
```

The addition is computed by applying the primitive strategy `addS` to the pair of integers `(i, j)` and matching the result against the variable `k`, which is then used in the right-hand side. This rule is desugared to

```
EvalPlus = ?Add(Int(i), Int(j)); where (! (i, j); addS; ?k); !Int(k)
```

8.3.6. Lambda Rules

Sometimes it is useful to define a rule anonymously within a strategy expression. The syntax for anonymous rules with scopes is a bit much since it requires enumerating all variables. A *lambda* rule of the form

```
\ p1 -> p2 where s \
```

is an anonymous rewrite rule for which the variables in the left-hand side `p1` are local to the rule, that is, it is equivalent to an expression of the form

```
{x1, ..., xn : (p1 -> p2 where s)}
```

with `x1, ..., xn` the variables of `p1`. This means that any variables used in `s` and `p2` that do *not* occur in `p1` are bound in the context of the rule.

A typical example of the use of an anonymous rule is

```
stratego> ! [(1,2), (3,4), (5,6)]
[(1,2), (3,4), (5,6)]
stratego> map (\ (x, y) -> x \ )
[1,3,5]
```

8.4. Apply and Match

One frequently occurring scenario is that of applying a strategy to a term and then matching the result against a pattern. This typically occurs in the condition of a rule. In the constant folding example above we saw this scenario:

```
EvalPlus : Add(Int(i), Int(j)) -> Int(k) where ! (i, j); addS; ?k
```

In the condition, first the term `(i, j)` is built, then the strategy `addS` is applied to it, and finally the result is matched against the pattern `k`.

To improve the readability of such expressions, the following two constructs are provided. The operation `<s> p` captures the notion of *applying* a strategy to a term, i.e., the scenario `!p; s`. The operation `s => p` capture the notion of applying a strategy to the current subject term and then matching the result against the pattern `p`, i.e., `s; ?p`. The combined operation `<s> p1 => p2` thus captures the notion of applying a strategy to a term `p1` and matching the result against `p2`, i.e., `!p1; s; ?p2`. Using this notation we can improve the constant folding rule above as

```
EvalPlus : Add(Int(i), Int(j)) -> Int(k) where <addS>(i, j) => k
```

Applying Strategies in Build. Sometimes it useful to apply a strategy directly to a subterm of a pattern, for example in the right-hand side of a rule, instead of computing a value in a condition, binding the result to a variable, and then using the variable in the build pattern. The constant folding rule above, for example, could be further simplified by directly applying the addition in the right-hand side:

```
EvalPlus : Add(Int(i), Int(j)) -> Int(<addS>(i, j))
```

This abbreviates the conditional rule above. In general, a strategy application in a build pattern can always be expressed by computing the application before the build and binding the result to a new variable, which then replaces the application in the build pattern.

Another example is the following definition of the `map(s)` strategy, which applies a strategy to each term in a list:

```
map(s) : [] -> []
map(s) : [x | xs] -> [<s> x | <map(s)> xs]
```

8.5 Auxiliary values and assignment

As mentioned above, it can be convenient to apply a strategy only to compute some auxiliary result. Although the `where` construct created to constrain when a rule or strategy may apply (as covered in Sections 8.3.4 and 8.3.5 above) can be used for this purpose, often it is better to use the `with` strategy specifically designed with computing auxiliaries in mind.

Specifically, if `s` is any strategy, the strategy `with(s)` executes `s` on the current subject term and then restores the current subject term. In other words, `s` is executed solely for its side effects, such as binding variables. In this respect, `with` is like `where`. However, `with(s)` differs in a key way: if the strategy `s` fails, Stratego immediately stops with an error, reporting the strategy that failed. Thus, if `with(s)` is used for auxiliary computations that really should not fail if the transformation is proceeding properly, there is no opportunity for Stratego to backtrack and/or continue applying other strategies, potentially creating an error at a point far removed from the place that things actually went awry. In short, using `with(s)` instead of `where(s)` any time the intention is not to constrain the applicability of a rule or strategy generally makes debugging your Stratego program significantly easier.

Also as with `where`, we can add a `with` clause to a rewrite rule in exactly the same way. In other words,

```
L : p1 -> p2 with s
```

is syntactic sugar for

```
L = ?p1; with(s); !p2
```

So as an example, the `where` version of `EvalPlus` from Section 8.4 would be better cast as

```
EvalPlus : Add(Int(i), Int(j)) -> Int(k) with <addS>(i, j) => k
```

because after all, there is no chance that Stratego will be unable to add two integers, and so if the contents of the `with` clause fails it means something has gone wrong – perhaps an `Int` term somehow ended up with a parameter that does not actually represent an integer – and Stratego should quit now.

Furthermore, in setting auxiliary variables often the full power of Stratego strategies is not used, but rather new terms are simply built as needed. Stratego provides an `:=` operator for this purpose; the above rule can be written probably more clearly as

```
EvalPlus : Add(Int(i), Int(j)) -> Int(k) with k := <addS>(i, j)
```

Technically, `p1 := p2` (which can be used anywhere a strategy is called for, although it is primarily useful in `with` and `where` clauses) is just syntactic sugar for `!p2; ?p1`. In other words, it builds the value `p2`, and then matches it with `p1`. In the typical case that `p1` is just a variable, this ends up assigning the result of building the expression `p2` to that variable.

To sum up, we have actually already seen an example of both `with` and `:=` in the “glue” strategy used to run a Stratego transformation via Editor Services:

```
do-eval: (selected, _, _, path, project-path) -> (filename, result)
  with filename := <guarantee-extension(|"eval.aterm"|)> path
    ; result    := <eval> selected
```

To make the operation of this rule clearer, the two components of the outcome are separated into auxiliary computations in the `with` clause, and these two auxiliaries are implemented as assignments with the `:=` operator. Moreover, if either the `eval` strategy fails or if Stratego is unable to compute the proper output filename, there is no point in continuing. So Stratego will simply terminate immediately and report the error.

8.6 Wrap and Project

Term wrapping and projection are concise idioms for constructing terms that wrap the current term and for extracting subterms from the current term.

8.6.1. Term Wrap

One often write rules of the form `x -> Foo (Bar (x))`, i.e. wrapping a term pattern around the current term. Using rule syntax this is quite verbose. The syntactic abstraction of *term wraps*, allows the concise specification of such little transformations as `!Foo (Bar (<id>))`.

In general, a term wrap is a build strategy `!p [<s>]` containing one or more strategy applications `<s>` that are *not applied to a term*. When executing the the build operation, each occurrence of such a strategy application `<s>` is replaced with the term resulting from applying `s` to the current subject term, i.e., the one that is being replaced by the build. The following sessions illustrate some uses of term wraps:

```
3
stratego> !(<id>,<id>)
(3,3)
stratego> !(<Fst; inc>,<Snd>)
(4,3)
stratego> !"foobar"
"foobar"
stratego> !Call(<id>,<[]>)
Call("foobar",<[]>)
stratego> mod2 = <mod>(<id>,<2>)
stratego> !6
6
stratego> mod2
0
```

As should now be a common pattern, term projects are implemented by translation to a combination of match and build expressions. Thus, a term wrap `!p [<s>]` is translated to a strategy expression

```
{x: where (s => x); !p[x]}
```

where `x` is a fresh variable not occurring in `s`. In other words, the strategy `s` is applied to the *current subject term*, i.e., the term to which the build is applied.

As an example, the term wrap `!Foo (Bar (<id>))` is desugared to the strategy

```
{x: where (id => x); !Foo(Bar(x))}
```

which after simplification is equivalent to `{x: ?x; !Foo (Bar (x))}`, i.e., exactly the original lambda rule `x -> Foo (Bar (x))`.

8.6.2. Term Project

Term projections are the match dual of term wraps. Term projections can be used to *project* a subterm from a term pattern. For example, the expression `?And(<id>, x)` matches terms of the form `And(t1, t2)` and reduces them to the first subterm `t1`. Another example is the strategy

```
map(?FunDec(<id>, _, _))
```

which reduces a list of function declarations to a list of the names of the functions, i.e., the first arguments of the `FunDec` constructor. Here are some more examples:

```
[1, 2, 3]
stratego> ?[_|<id>]
[2, 3]
stratego> !Call("foobar", [])
Call("foobar", [])
stratego> ?Call(<id>, [])
"foobar"
```

Term projections can also be used to apply additional constraints to subterms in a match pattern. For example, `?Call(x, <?args; length => 3>)` matches only with function calls with three arguments.

A match expression `?p[<s>]` is desugared as

```
{x: ?p[x]; <s> x}
```

That is, after the pattern `p[x]` matches, it is reduced to the subterm bound to `x` to which `s` is applied. The result is also the result of the projection. When multiple projects are used within a match the outcome is undefined, i.e., the order in which the projects will be performed can not be counted on.

13.1.9 9. Traversal Strategies

In [Chapter 5](#) we saw a number of idioms of strategic rewriting, which all involved *tree traversal*. In the previous chapters we saw how strategies can be used to control transformations and how rules can be broken down into the primitive actions match, build and scope. The missing ingredient are combinators for defining traversals.

There are many ways to traverse a tree. For example, a bottom-up traversal, visits the subterms of a node before it visits the node itself, while a top-down traversal visits nodes before it visits children. One-pass traversals traverse the tree one time, while fixed-point traversals, such as `innermost`, repeatedly traverse a term until a normal form is reached.

It is not desirable to provide built-in implementations for all traversals needed in transformations, since such a collection would necessarily be incomplete. Rather we would like to define traversals in terms of the primitive ingredients of traversal. For example, a top-down, one-pass traversal strategy will first visit a node, and then descend to the children of a node in order to *recursively* traverse all subterms. Similarly, the bottom-up, fixed-point traversal strategy *innermost*, will first descend to the children of a node in order to *recursively* traverse all subterms, then visit the node itself, and possibly recursively reapply the strategy.

Traversal in Stratego is based on the observation that a full term traversal is a recursive closure of a one-step descent, that is, an operation that applies a strategy to one or more direct subterms of the subject term. By separating this one-step descent operator from recursion, and making it a first-class operation, many different traversals can be defined.

In this chapter we explore the ways in which Stratego supports the definition of *traversal strategies*. We start with explicitly programmed traversals using recursive traversal rules. Next, *congruences operators* provide a more concise notation for such data-type specific traversal rules. Finally, *generic traversal operators* support data type independent definitions of traversals, which can be reused for any data type. Given these basic mechanisms, we conclude with an exploration of idioms for traversal and standard traversal strategies in the Stratego Library.

In [Chapter 8](#) we saw the following definition of the `map` strategy, which applies a strategy to each element of a list:

```
map(s) : [] -> []
map(s) : [x | xs] -> [<s> x | <map(s)> xs]
```

The definition uses explicit recursive calls to the strategy in the right-hand side of the second rule. What `map` does is to *traverse* the list in order to apply the argument strategy to all elements. We can use the same technique to other term structures as well.

We will explore the definition of traversals using the propositional formulae from [Chapter 5](#), where we introduced the following rewrite rules:

```
module prop-rules
imports libstrategolib prop
rules
  DefI : Impl(x, y)      -> Or(Not(x), y)
  DefE : Eq(x, y)        -> And(Impl(x, y), Impl(y, x))
  DN   : Not(Not(x))     -> x
  DMA  : Not(And(x, y))  -> Or(Not(x), Not(y))
  DMO  : Not(Or(x, y))   -> And(Not(x), Not(y))
  DAOL : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
  DAOR : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
  DOAL : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
  DOAR : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
```

In [Chapter 5](#) we saw how a functional style of rewriting could be encoded using extra constructors. In Stratego we can achieve a similar approach by using rule names, instead of extra constructors. Thus, one way to achieve normalization to disjunctive normal form, is the use of an explicitly programmed traversal, implemented using recursive rules, similarly to the `map` example above:

```
module prop-dnf4
imports libstrategolib prop-rules
strategies
  main = io-wrap(dnf)
rules
  dnf : True()      -> True()
  dnf : False()     -> False()
  dnf : Atom(x)     -> Atom(x)
  dnf : Not(x)      -> <dnfred> Not (<dnf>x)
  dnf : And(x, y)   -> <dnfred> And (<dnf>x, <dnf>y)
  dnf : Or(x, y)    -> Or (<dnf>x, <dnf>y)
  dnf : Impl(x, y)  -> <dnfred> Impl(<dnf>x, <dnf>y)
  dnf : Eq(x, y)    -> <dnfred> Eq (<dnf>x, <dnf>y)
strategies
  dnfred = try(DN <+ (DefI <+ DefE <+ DMA <+ DMO <+ DAOL <+ DAOR); dnf)
```

The `dnf` rules recursively apply themselves to the direct subterms and then apply `dnfred` to actually apply the rewrite rules.

We can reduce this program by abstracting over the base cases. Since there is no traversal into `True`, `False`, and `Atoms`, these rules can be left out.

```
module prop-dnf5
imports libstrategolib prop-rules
strategies
  main = io-wrap(dnf)
rules
  dnft : Not(x)      -> <dnfred> Not (<dnf>x)
```

(continues on next page)

(continued from previous page)

```

dnft : And(x, y)  -> <dnfred> And (<dnf>x, <dnf>y)
dnft : Or(x, y)   ->                Or (<dnf>x, <dnf>y)
dnft : Impl(x, y) -> <dnfred> Impl(<dnf>x, <dnf>y)
dnft : Eq(x, y)   -> <dnfred> Eq (<dnf>x, <dnf>y)
strategies
dnf    = try(dnft)
dnfred = try(DN <+ (DefI <+ DefE <+ DMA <+ DMO <+ DAOL <+ DAOR); dnf)

```

The `dnf` strategy is now defined in terms of the `dnft` rules, which implement traversal over the constructors. By using `try(dnft)`, terms for which no traversal rule has been specified are not transformed.

We can further simplify the definition by observing that the application of `dnfred` does not necessarily have to take place in the right-hand side of the traversal rules.

```

module prop-dnf6
imports libstrategolib prop-rules
strategies
  main = io-wrap(dnf)
rules
  dnft : Not(x)      -> Not (<dnf>x)
  dnft : And(x, y)   -> And (<dnf>x, <dnf>y)
  dnft : Or(x, y)    -> Or (<dnf>x, <dnf>y)
  dnft : Impl(x, y)  -> Impl(<dnf>x, <dnf>y)
  dnft : Eq(x, y)    -> Eq (<dnf>x, <dnf>y)
strategies
  dnf    = try(dnft); dnfred
  dnfred = try(DN <+ (DefI <+ DefE <+ DMA <+ DMO <+ DAOL <+ DAOR); dnf)

```

In this program `dnf` first calls `dnft` to transform the subterms of the subject term, and then calls `dnfred` to apply the transformation rules (and possibly a recursive invocation of `dnf`).

The program above has two problems. First, the traversal behavior is mostly uniform, so we would like to specify that more concisely. We will address that concern below. Second, the traversal is not reusable, for example, to define a conjunctive normal form transformation. This last concern can be addressed by factoring out the recursive call to `dnf` and making it a parameter of the traversal rules.

```

module prop-dnf7
imports libstrategolib prop-rules
strategies
  main = io-wrap(dnf)
rules
  proptr(s) : Not(x)      -> Not (<s>x)
  proptr(s) : And(x, y)   -> And (<s>x, <s>y)
  proptr(s) : Or(x, y)    -> Or (<s>x, <s>y)
  proptr(s) : Impl(x, y)  -> Impl(<s>x, <s>y)
  proptr(s) : Eq(x, y)    -> Eq (<s>x, <s>y)
strategies
  dnf    = try(proptr(dnf)); dnfred
  dnfred = try(DN <+ (DefI <+ DefE <+ DMA <+ DMO <+ DAOL <+ DAOR); dnf)
  cnf    = try(proptr(cnf)); cnfred
  cnfred = try(DN <+ (DefI <+ DefE <+ DMA <+ DMO <+ DOAL <+ DOAR); cnf)

```

Now the traversal rules are reusable and used in two different transformations, by instantiation with a call to the particular strategy in which they are used (`dnf` or `cnf`).

But we can do better, and also make the *composition* of this strategy reusable.


```

module prop-dnf8
imports libstrategolib prop-rules
strategies
  main = io-wrap(dnf)
rules
  proptr(s) : Not(x)      -> Not (<s>x)
  proptr(s) : And(x, y)   -> And (<s>x, <s>y)
  proptr(s) : Or(x, y)    -> Or  (<s>x, <s>y)
  proptr(s) : Impl(x, y)  -> Impl(<s>x, <s>y)
  proptr(s) : Eq(x, y)    -> Eq  (<s>x, <s>y)
strategies
  propbu(s) = try(proptr(propbu(s))); s
strategies
  dnf = propbu(dnfred)
  dnfred = try(DN <+ (DefI <+ DefE <+ DMA <+ DMO <+ DAOL <+ DAOR); dnf)
  cnf = propbu(cnfred)
  cnfred = try(DN <+ (DefI <+ DefE <+ DMA <+ DMO <+ DOAL <+ DOAR); cnf)
    
```

That is, the `propbu(s)` strategy defines a complete bottom-up traversal over proposition terms, applying the strategy `s` to a term after transforming its subterms. The strategy is completely independent of the `dnf` and `cnf` transformations, which instantiate the strategy using the `dnfred` and `cnfred` strategies.

Come to think of it, `dnfred` and `cnfred` are somewhat useless now and can be inlined directly in the instantiation of the `propbu(s)` strategy:

```

module prop-dnf9
imports libstrategolib prop-rules
strategies
  main = io-wrap(dnf)
rules
  proptr(s) : Not(x)      -> Not (<s>x)
  proptr(s) : And(x, y)   -> And (<s>x, <s>y)
  proptr(s) : Or(x, y)    -> Or  (<s>x, <s>y)
  proptr(s) : Impl(x, y)  -> Impl(<s>x, <s>y)
  proptr(s) : Eq(x, y)    -> Eq  (<s>x, <s>y)
strategies
  propbu(s) = try(proptr(propbu(s))); s
strategies
  dnf = propbu(try(DN <+ (DefI <+ DefE <+ DMA <+ DMO <+ DAOL <+ DAOR); dnf))
  cnf = propbu(try(DN <+ (DefI <+ DefE <+ DMA <+ DMO <+ DOAL <+ DOAR); cnf))
    
```

Now we have defined a *transformation independent* traversal strategy that is *specific* for proposition terms.

Next we consider cheaper ways for defining the traversal rules, and then ways to get completely rid of them.

9.1. Congruence Operators

The definition of the traversal rules above frequently occurs in the definition of transformation strategies. Congruence operators provide a convenient abbreviation of precisely this operation. A congruence operator applies a strategy to each direct subterm of a specific constructor. For each n -ary constructor c declared in a signature, there is a corresponding *congruence operator* $c(s_1, \dots, s_n)$, which applies to terms of the form $c(t_1, \dots, t_n)$ by applying the argument strategies to the corresponding argument terms. A congruence fails if the application of one the argument strategies fails or if constructor of the operator and that of the term do not match.

Example. For example, consider the following signature of expressions:

```
module expressions
signature
  sorts Exp
  constructors
    Int    : String -> Exp
    Var    : String -> Exp
    Plus   : Exp * Exp -> Exp
    Times  : Exp * Exp -> Exp
```

The following Stratego Shell session applies the congruence operators `Plus` and `Times` to a term:

```
stratego> import expressions
stratego> !Plus(Int("14"), Int("3"))
Plus(Int("14"), Int("3"))
stratego> Plus(!Var("a"), id)
Plus(Var("a"), Int("3"))
stratego> Times(id, !Int("42"))
command failed
```

The first application shows how a congruence transforms a specific subterm, that is the strategy applied can be different for each subterm. The second application shows that a congruence only succeeds for terms constructed with the same constructor.

The `import` at the start of the session is necessary to declare the constructors used; the definitions of congruences are derived from constructor declarations. Forgetting this import would lead to a complaint about an undeclared operator:

```
stratego> !Plus(Int("14"), Int("3"))
Plus(Int("14"), Int("3"))
stratego> Plus(!Var("a"), id)
operator Plus/(2,0) not defined
command failed
```

Defining Traversals with Congruences. Now we return to our `dnf/cnf` example, to see how congruence operators can help in their implementation. Since congruence operators basically define a one-step traversal for a specific constructor, they capture the traversal rules defined above. That is, a traversal rule such as

```
proptr(s) : And(x, y) -> And(<s>x, <s>y)
```

can be written by the congruence `And(s, s)`. Applying this to the `prop-dnf` program we can replace the traversal rules by congruences as follows:

```
module prop-dnf10
imports libstrategolib prop-rules
strategies
  main = io-wrap(dnf)
strategies
  proptr(s) = Not(s) <+ And(s, s) <+ Or(s, s) <+ Impl(s, s) <+ Eq(s, s)
  propbu(s) = try(proptr(propbu(s))); s
strategies
  dnf = propbu(try(DN <+ (DefI <+ DefE <+ DMA <+ DMO <+ DAOL <+ DAOR); dnf))
  cnf = propbu(try(DN <+ (DefI <+ DefE <+ DMA <+ DMO <+ DOAL <+ DOAR); cnf))
```

Observe how the five traversal rules have been reduced to five congruences which fit on a single line.

Traversing Tuples and Lists. Congruences can also be applied to tuples, (s_1, s_2, \dots, s_n) , and lists, $[s_1, s_2, \dots, s_n]$. A special list congruence is `[]` which ‘visits’ the empty list. As an example, consider again the definition of `map(s)` using recursive traversal rules:

```
map(s) : [] -> []
map(s) : [x | xs] -> [<s> x | <map(s)> xs]
```

Using list congruences we can define this strategy as:

```
map(s) = [] <+ [s | map(s)]
```

The `[]` congruence matches an empty list. The `[s | map(s)]` congruence matches a non-empty list, and applies `s` to the head of the list and `map(s)` to the tail. Thus, `map(s)` applies `s` to each element of a list:

```
stratego> import libstratego-lib
stratego> ![1,2,3]
[1,2,3]
stratego> map(inc)
[2,3,4]
```

Note that `map(s)` only succeeds if `s` succeeds for each element of the list. The `fetch` and `filter` strategies are variations on `map` that use the failure of `s` to list elements.

```
fetch(s) = [s | id] <+ [id | fetch(s)]
```

The `fetch` strategy traverses a list *until* it finds a element for which `s` succeeds and then stops. That element is the only one that is transformed.

```
filter(s) = [] + ([s | filter(s)] <+ ?[ |<id>]; filter(s))
```

The `filter` strategy applies `s` to each element of a list, but only keeps the elements for which it succeeds.

```
stratego> import libstratego-lib
stratego> even = where(<eq>(<mod>(<id>,2),0))
stratego> ![1,2,3,4,5,6,7,8]
[1,2,3,4,5,6,7,8]
stratego> filter(even)
[2,4,6,8]
```

Format Checking. Another application of congruences is in the definition of format checkers. A format checker describes a subset of a term language using a recursive pattern. This can be used to verify input or output of a transformation, and for documentation purposes. Format checkers defined with congruences can check subsets of signatures or regular tree grammars. For example, the subset of terms of a signature in a some normal form.

As an example, consider checking the output of the `dnf` and `cnf` transformations.

```
conj(s) = And(conj(s), conj(s)) <+ s
disj(s) = Or (disj(s), disj(s)) <+ s

// Conjunctive normal form
conj-nf = conj(disj(Not(Atom(id))) <+ Atom(id)))

// Disjunctive normal form
disj-nf = disj(conj(Not(Atom(id))) <+ Atom(id)))
```

The strategies `conj(s)` and `disj(s)` check that the subject term is a conjunct or a disjunct, respectively, with terms satisfying `s` at the leaves. The strategies `conj-nf` and `disj-nf` check that the subject term is in conjunctive or disjunctive normal form, respectively.

Using congruence operators we constructed a generic, i.e. transformation independent, bottom-up traversal for proposition terms. The same can be done for other data types. However, since the sets of constructors of abstract syntax trees of typical programming languages can be quite large, this may still amount to quite a bit of work that is not

reusable *across* data types; even though a strategy such as *bottom-up traversal*, is basically data-type independent. Thus, Stratego provides generic traversal by means of several *generic one-step descent operators*. The operator `all`, applies a strategy to all direct subterms. The operator `one`, applies a strategy to one direct subterm, and the operator `some`, applies a strategy to as many direct subterms as possible, and at least one.

9.1.1. Visiting All Subterms

The `all(s)` strategy transforms a constructor application by applying the parameter strategy `s` to each direct subterm. An application of `all(s)` fails if the application to one of the subterms fails. The following example shows how `all(1)` applies to any term, and (2) applies its argument strategy uniformly to all direct subterms. That is, it is not possible to do something special for a particular subterm (that's what congruences are for).

```
stratego> !Plus(Int("14"), Int("3"))
Plus(Int("14"), Int("3"))
stratego> all(!Var("a"))
Plus(Var("a"), Var("a"))
stratego> !Times(Var("b"), Int("3"))
Times(Var("b"), Int("3"))
stratego> all(!Var("z"))
Times(Var("z"), Var("z"))
```

The `all(s)` operator is really the ultimate replacement for the traversal rules that we saw above. Instead of specifying a rule or congruence for each constructor, the single application of the `all` operator takes care of traversing all constructors. Thus, we can replace the `propbu` strategy by a completely generic definition of bottom-up traversal. Consider again the last definition of `propbu`:

```
proptr(s) = Not(s) <+ And(s, s) <+ Or(s, s) <+ Impl(s, s) <+ Eq(s, s)
propbu(s) = try(proptr(propbu(s))); s
```

The role of `proptr(s)` in this definition can be replaced by `all(s)`, since that achieves exactly the same, namely applying `s` to the direct subterms of constructors:

```
propbu(s) = all(propbu(s)); s
```

Moreover, `all` succeeds on any constructor in any signature, so we can also as you see above drop the `try` as well, which was there only because `proptr` fails on the `Atom(...)`, `True()`, and `False()` nodes at the leaves.

However, the strategy now is completely generic, i.e. independent of the particular structure it is applied to. In the Stratego Library this strategy is called `bottomup(s)`, and defined as follows:

```
bottomup(s) = all(bottomup(s)); s
```

It first recursively transforms the subterms of the subject term and then applies `s` to the result. Using this definition, the normalization of propositions now reduces to the following module, which is only concerned with the selection and composition of rewrite rules:

```
module prop-dnf11
imports libstrategolib prop-rules
strategies
  main = io-wrap(dnf)
strategies
  dnf = bottomup(try(DN <+ (DefI <+ DefE <+ DMA <+ DMO <+ DAOL <+ DAOR); dnf))
  cnf = bottomup(try(DN <+ (DefI <+ DefE <+ DMA <+ DMO <+ DOAL <+ DOAR); cnf))
```

In fact, these definitions still contain a reusable pattern. With a little squinting we see that the definitions match the following pattern:

```
dnf = bottomup(try(dnf-rules; dnf))
cnf = bottomup(try(cnf-rules; cnf))
```

In which we can recognize the definition of *innermost* reduction, which the Stratego Library defines as:

```
innermost(s) = bottomup(try(s; innermost(s)))
```

The *innermost* strategy performs a bottom-up traversal of a term. After transforming the subterms of a term it tries to apply the transformation *s*. If successful the result is recursively transformed with an application of *innermost*. This brings us to the final form for the proposition normalizations:

```
module prop-dnf12
imports libstrategolib prop-rules
strategies
  main = io-wrap(dnf)
strategies
  dnf = innermost(DN <+ DefI <+ DefE <+ DMA <+ DMO <+ DAOL <+ DAOR)
  cnf = innermost(DN <+ DefI <+ DefE <+ DMA <+ DMO <+ DOAL <+ DOAR)
```

Different transformations can be achieved by using a selection of rules and a strategy, which is generic, yet defined in Stratego itself using strategy combinators.

9.1.2. Visiting One Subterm

The *one(s)* strategy transforms a constructor application by applying the parameter strategy *s* to exactly one direct subterm. An application of *one(s)* fails if the application to all of the subterms fails. The following Stratego Shell session illustrates the behavior of the combinator:

```
stratego> !Plus(Int("14"), Int("3"))
Plus(Int("14"), Int("3"))
stratego> one(!Var("a"))
Plus(Var("a"), Int("3"))
stratego> one(\ Int(x) -> Int(<addS>(x, "1")) \ )
Plus(Var("a"), Int("4"))
stratego> one(?Plus(_, _))
command failed
```

A frequently used application of *one* is the *onced(s)* traversal, which performs a left to right depth first search/transformation that stops as soon as *s* has been successfully applied.

```
onced(s) = s <+ one(onced(s))
```

Thus, *s* is first applied to the root of the subject term. If that fails, its direct subterms are searched one by one (from left to right), with a recursive call to *onced(s)*.

An application of *onced* is the *contains(|t)* strategy, which checks whether the subject term contains a subterm that is equal to *t*.

```
contains(|t) = onced(?t)
```

Through the depth first search of *onced*, either an occurrence of *t* is found, or all subterms are verified to be unequal to *t*.

Here are some other one-pass traversals using the *one* combinator:

```
oncebu(s) = one(oncebu(s)) <+ s
spinetd(s) = s; try(one(spinetd(s)))
spinebu(s) = try(one(spinebu(s))); s
```

Exercise: figure out what these strategies do.

Here are some fixe-point traversals, i.e., traversals that apply their argument transformation exhaustively to the subject term.

```
reduce(s) = repeat(rec x(one(x) + s))
outermost(s) = repeat(oncebd(s))
innermostI(s) = repeat(oncebu(s))
```

The difference is the subterm selection strategy. Exercise: create rewrite rules and terms that demonstrate the differences between these strategies.

9.1.3. Visiting Some Subterms

The `some(s)` strategy transforms a constructor application by applying the parameter strategy `s` to as many direct subterms as possible and at least one. An application of `some(s)` fails if the application to all of the subterms fails.

Some one-pass traversals based on `some`:

```
sometd(s) = s <+ some(sometd(s))
somebu(s) = some(somebu(s)) <+ s
```

A fixed-point traversal with `some`:

```
reduce-par(s) = repeat(rec x(some(x) + s))
```

9.2. Idioms and Library Strategies for Traversal

Above we have seen the basic mechanisms for defining traversals in Stratego: custom traversal rules, data-type specific congruence operators, and generic traversal operators. Term traversals can be categorized into classes according to how much of the term they traverse and to which parts of the term they modify. We will consider a number of idioms and standard strategies from the Stratego Library that are useful in the definition of traversals.

One class of traversal strategies performs a *full traversal*, that is visits and transforms every subterm of the subject term. We already saw the `bottomup` strategy defined as

```
bottomup(s) = all(bottomup(s)); s
```

It first visits the subterms of the subject term, recursively transforming *its* subterms, and then applies the transformation `s` to the result.

A related strategy is `topdown`, which is defined as

```
topdown(s) = s; all(topdown(s))
```

It *first* transforms the subject term and *then* visits the subterms of the result.

A combination of `topdown` and `bottomup` is `downup`, defined as

```
downup(s) = s; all(downup(s)); s
```

It applies `s` on the way down the tree, and again on the way up. A variation is `downup(2, 0)`

```
downup(s1, s2) = s1; all(downup(s1, s2)); s2
```

which applies one strategy on the way down and another on the way up.

Since the parameter strategy is applied at every subterm, these traversals only succeed if it succeeds everywhere. Therefore, these traversals are typically applied in combination with `try` or `repeat`.

```
topdown(try(R1 <+ R2 <+ ...))
```

This has the effect that the rules are tried at each subterm. If none of the rules apply the term is left as it was and traversal continues with its subterms.

Choosing a Strategy. The strategy to be used for a particular transformation depends on the rules and the goal to be achieved.

For example, a constant folding transformation for proposition formulae can be defined as a bottom-up traversal that tries to apply one of the truth-rules `T` at each subterm:

```
T : And(True(), x) -> x
T : And(x, True()) -> x
T : And(False(), x) -> False()
T : And(x, False()) -> False()
T : Or(True(), x) -> True()
T : Or(x, True()) -> True()
T : Or(False(), x) -> x
T : Or(x, False()) -> x
T : Not(False()) -> True()
T : Not(True()) -> False()

eval = bottomup(try(T))
```

Bottomup is the strategy of choice here because it evaluates subterms before attempting to rewrite a term. An evaluation strategy using `topdown`

```
eval2 = topdown(try(T)) // bad strategy
```

does not work as well, since it attempts to rewrite terms before their subterms have been reduced, thus missing rewriting opportunities. The following Stratego Shell session illustrates this:

```
stratego> !And(True(), Not(Or(False(), True())))
And(True,Not(Or(False,True)))
stratego> eval
False
stratego> !And(True(), Not(Or(False(), True())))
And(True,Not(Or(False,True)))
stratego> eval2
Not(True)
```

Exercise: find other terms that show the difference between these strategies.

On the other hand, a desugaring transformation for propositions, which defines implication and equivalence in terms of other connectives is best defined as a `topdown` traversal which tries to apply one of the rules `DefI` or `DefE` at every subterm.

```
DefI : Impl(x, y) -> Or(Not(x), y)
DefE : Eq(x, y) -> And(Impl(x, y), Impl(y, x))

desugar = topdown(try(DefI <+ DefE))
```

Since `DefE` rewrites `Eq` terms to terms involving `Impl`, a strategy with `bottomup` does not work.

```
desugar2 = bottomup(try(DefI <+ DefE)) // bad strategy
```

Since the subterms of a node are traversed *before* the node itself is visited, this transformation misses the desugaring of the implications (`Impl`) originating from the application of the `DefE` rule. The following Shell session illustrates this:

```
stratego> !Eq(Atom("p"), Atom("q"))
Eq(Atom("p"), Atom("q"))
stratego> desugar
And(Or(Not(Atom("p")), Atom("q")), Or(Not(Atom("q")), Atom("p")))
stratego> !Eq(Atom("p"), Atom("q"))
Eq(Atom("p"), Atom("q"))
stratego> desugar2
And(Impl(Atom("p"), Atom("q")), Impl(Atom("q"), Atom("p")))
```

Repeated Application. In case one rule produces a term to which another desugaring rule can be applied, the desugaring strategy should repeat the application of rules to each subterm. Consider the following rules and strategy for desugaring propositional formulae to implicative normal form (using only implication and `False`).

```
DefT : True() -> Impl(False(), False())
DefN : Not(x) -> Impl(x, False())
DefA2 : And(x, y) -> Not(Impl(x, Not(y)))
DefO1 : Or(x, y) -> Impl(Not(x), y)
DefE : Eq(x, y) -> And(Impl(x, y), Impl(y, x))

impl-nf = topdown(repeat(DefT <+ DefN <+ DefA2 <+ DefO1 <+ DefE))
```

Application of the rules with `try` instead of `repeat`

```
impl-nf2 = topdown(try(DefT <+ DefN <+ DefA2 <+ DefO1 <+ DefE)) // bad strategy
```

is not sufficient, as shown by the following Shell session:

```
stratego> !And(Atom("p"), Atom("q"))
And(Atom("p"), Atom("q"))
stratego> impl-nf
Impl(Impl(Atom("p"), Impl(Atom("q"), False())), False)
stratego> !And(Atom("p"), Atom("q"))
And(Atom("p"), Atom("q"))
stratego> impl-nf2
Not(Impl(Atom("p"), Impl(Atom("q"), False)))
```

Note that the `Not` is not desugared with `impl-nf2`.

Paramorphism. A variation on `bottomup` is a traversal that also provides the original term as well as the term in which the direct subterms have been transformed. (Also known as a paramorphism?)

```
bottomup-para(s) = <s>(<id>, <all(bottomup-para(s))>)
```

This is most useful in a bottom-up traversal; the original term is always available in a top-down traversal.

Exercise: give an example application of this strategy

9.2.1. Cascading Transformations

Cascading transformations are transformations upon transformations. While the full traversals discussed above walk over the tree once, cascading transformations apply multiple *waves* of transformations to the nodes in the tree. The prototypical example is the `innermost` strategy, which exhaustively applies a transformation, typically a set of rules, to a tree.

```
simplify =
  innermost (R1 <+ ... <+ Rn)
```

The basis of `innermost` is a `bottomup` traversal that tries to apply the transformation at each node after visiting its subterms.

```
innermost(s) = bottomup(try(s; innermost(s)))
```

If the transformation `s` succeeds, the result term is transformed again with a recursive call to `innermost`.

Application of `innermost` exhaustively applies *one* set of rules to a tree. Using sequential composition we can apply several *stages* of reductions. A special case of such a *staged transformation*, is known as *sequence of normal forms* (in the TAMPR system):

```
simplify =
  innermost (A1 <+ ... <+ Ak)
; innermost (B1 <+ ... <+ B1)
; ...
; innermost (C1 <+ ... <+ Cm)
```

At each stage the term is reduced with respect to a different set of rules.

Of course it is possible to mix different types of transformations in such a stage pipeline, for example.

```
simplify =
  topdown(try (A1 <+ ... <+ Ak) )
; innermost (B1 <+ ... <+ B1)
; ...
; bottomup(repeat (C1 <+ ... <+ Cm))
```

At each stage a different strategy and different set of rules can be used. (Of course one may use the same strategy several times, and some of the rule sets may overlap.)

9.2.2. Mixing Generic and Specific Traversals

While completely generic strategies such as `bottomup` and `innermost` are often useful, there are also situations where a mixture of generic and data-type specific traversal is necessary. Fortunately, Stratego allows you to mix generic traversal operators, congruences, your own traversal and regular rules, any way you see fit.

A typical pattern for such strategies first tries a number of special cases that deal with traversal themselves. If none of the special cases apply, a generic traversal is used, followed by application of some rules applicable in the general case.

```
transformation =
  special-case1
<+ special-case2
<+ special-case3
<+ all(transformation); reduce

reduce = ...
```

Constant Propagation. A typical example is the following constant propagation strategy. It uses the exceptions to the basic generic traversal to traverse the tree in the order of the control-flow of the program that is represented by the term. This program makes use of *dynamic rewrite rules*, which are used to propagate context-sensitive information through a program. In this case, the context-sensitive information concerns the constant values of some variables in the program, which should be propagated to the uses of those variables. Dynamic rules will be explained in [Chapter 12](#); for now we are mainly concerned with the traversal strategy.

```
module propconst
imports
  libstratego-lib

signature
  constructors
    Var      : String -> Exp
    Plus     : Exp * Exp -> Exp
    Assign   : String * Exp -> Stat
    If       : Exp * Stat * Stat -> Stat
    While    : Exp * Stat -> Stat

strategies

propconst =
  PropConst
  <+ propconst-assign
  <+ propconst-if
  <+ propconst-while
  <+ all(propconst); try (EvalBinOp)

EvalBinOp :
  Plus(Int(i), Int(j)) -> Int(k) where <addS>(i,j) ==> k

EvalIf :
  If(Int("0"), s1, s2) -> s2

EvalIf :
  If(Int(i), s1, s2) -> s1 where <not(eq)>(i, "0")

propconst-assign =
  Assign(?x, propconst ==> e)
  ; if <is-value> e then
    rules( PropConst : Var(x) -> e )
  else
    rules( PropConst :- Var(x) )
  end

propconst-if =
  If(propconst, id, id)
  ; (EvalIf; propconst
    <+ (If(id, propconst, id) /PropConst\ If(id,id,propconst)))

propconst-while =
  While(id,id)
  ; (/PropConst\* While(propconst, propconst))

is-value = Int(id)
```

The main strategy of the constant propagation transformation, follows the pattern described above; a number of special case alternatives followed by a generic traversal alternative. The special cases are defined in their own definitions.

Generic traversal is followed by the constant folding rule `EvalBinOp`.

The first special case is an application of the dynamic rule `PropConst`, which replaces a constant valued variable by its constant value. This rule is defined by the second special case strategy, `propconst-assign`. It first traverses the right-hand side of an assignment with an `Assign` congruence operator, and a recursive call to `propconst`. Then, if the expression evaluated to a constant value, a new `PropConst` rule is defined. Otherwise, any old instance of `PropConst` for the left-hand side variable is undefined.

The third special case for `If` uses congruence operators to order the application of `propconst` to its subterms. The first congruence applies `propconst` to the condition expression. Then an application of the rule `EvalIf` attempts to eliminate one of the branches of the statement, in case the condition evaluated to a constant value. If that is not possible the branches are visited by two more congruence operator applications joined by a dynamic rule intersection operator, which distributes the constant propagation rules over the branches and merges the rules afterwards, keeping only the consistent ones. Something similar happens in the case of `While` statements. For details concerning dynamic rules, see [Chapter 12](#).

To see what `propconst` achieves, consider the following abstract syntax tree (say in file `foo.prg`).

```
Block([
  Assign("x", Int("1")),
  Assign("y", Int("42")),
  Assign("z", Plus(Var("x"), Var("y"))),
  If(Plux(Var("a"), Var("z")),
    Assign("b", Plus(Var("x"), Int("1"))),
    Block([
      Assign("z", Int("17")),
      Assign("b", Int("2"))
    ])),
  Assign("c", Plus(Var("b"), Plus(Var("z"), Var("y"))))
])
```

We import the module in the Stratego Shell, read the abstract syntax tree from file, and apply the `propconst` transformation to it:

```
stratego> import libstrategolib
stratego> import propconst
stratego> <ReadFromFile> "foo.prg"
...
stratego> propconst
Block([Assign("x",Int("1")),Assign("y",Int("42")),Assign("z",Int("43")),
If(Plux(Var("a"),Int("43")),Assign("b",Int("2")),Block([Assign("z",
Int("17")),Assign("b",Int("2"))])),Assign("c",Plus(Int("2"),Plus(
Var("z"),Int("42"))))]])
```

Since the Stratego Shell does not (yet) pretty-print terms, the result is rather unreadable. We can remedy this by writing the result of the transformation to a file, and pretty-printing it on the regular command-line with `pp-aterm`.

```
stratego> <ReadFromFile> "foo.prg"
...
stratego> propconst; <WriteToTextFile> ("foo-pc.prg", <id>)
...
stratego> :quit
...
$ pp-aterm -i foo-pc.prg
Block(
  [ Assign("x", Int("1"))
    , Assign("y", Int("42"))
    , Assign("z", Int("43"))
```

(continues on next page)

(continued from previous page)

```
, If(
  Plus(Var("a"), Int("43"))
, Assign("b", Int("2"))
, Block(
  [Assign("z", Int("17")), Assign("b", Int("2"))]
)
)
, Assign(
  "c"
, Plus(Int("2"), Plus(Var("z"), Int("42")))
)
]
)
```

Compare the result to the original program and try to figure out what has happened and why that is correct. (Assuming the *usual* semantics for this type of imperative language.)

Generic Strategies with Exceptional Cases. Patterns for mixing specific and generic traversal can be captured in parameterized strategies such as the following. They are parameterized with the usual transformation parameter *s* and with a higher-order strategy operator *stop*, which implements the special cases.

```
topdownS(s, stop: (a -> a) * b -> b) =
  rec x(s; (stop(x) <+ all(x)))

bottomupS(s, stop: (a -> a) * b -> b) =
  rec x((stop(x) <+ all(x)); s)

downupS(s, stop: (a -> a) * b -> b) =
  rec x(s; (stop(x) <+ all(x)); s)

downupS(s1, s2, stop: (a -> a) * b -> b) =
  rec x(s1; (stop(x) <+ all(x)); s2)
```

While normal strategies (parameters) are functions from terms to terms, the *stop* parameter is a function from strategies to strategies. Such exceptions to the default have to be declared explicitly using a type annotation. Note that the *bottomupS* strategy is slightly different from the pattern of the *propconst* strategy; instead of applying *s* *only* after the generic traversal case, it is here applied in all cases.

However, the added value of these strategies is not very high. The payoff in the use of generic strategies is provided by the basic generic traversal operators, which provide generic behavior for all constructors. The *stop* callback can make it harder to understand the control-flow structure of a strategy; use with care and don't overdo it.

9.2.3. Separate rules and strategies

While it is possible to construct your own strategies by mixing traversal elements and rules, in general, it is a good idea to try to get a clean separation between pure rewrite rules and a (simple) strategy that applies them.

9.2.4. Partial Traversals

The full traversals introduced above mostly visit all nodes in the tree. Now we consider traversals that visit only some of the nodes of a tree.

The *oncet* and *oncebu* strategies apply the argument strategy *s* at one position in the tree. That is, application is tried at every node along the traversal until it succeeds.

```
oncetd(s) = s <+ one(oncetd(s))
oncebu(s) = one(oncebu(s)) <+ s
```

The `sometd` and `somebu` strategies are variations on `oncet` and `oncebu` that apply `s` at least once at some positions, but possibly many times. As soon as one is found, searching is stopped, i.e., in the top-down case searching in subtrees is stopped, in bottom-up case, searching in upper spine is stopped.

```
sometd(s) = s <+ some(sometd(s))
somebu(s) = some(somebu(s)) <+ s
```

Similar strategies that find as many applications as possible, but at least one, can be built using `some`:

```
manybu(s) = rec x(some(x); try(s) <+ s)
manytd(s) = rec x(s; all(try(x)) <+ some(x))
```

```
somedownup(s) = rec x(s; all(x); try(s) <+ some(x); try(s))
```

The `alltd(s)` strategy stops as soon as it has found a subterm to which `s` can be successfully applied.

```
alltd(s) = s <+ all(alltd(s))
```

If `s` does not succeed, the strategy is applied recursively at all direct subterms. This means that `s` is applied along a frontier of the subject term. This strategy is typically used in substitution operations in which subterms are replaced by other terms. For example, the strategy `alltd(?Var(x); !e)` replaces all occurrences of `Var(x)` by `e`. Note that `alltd(try(s))` is not a useful strategy. Since `try(s)` succeeds at the root of the term, no traversal is done.

A typical application of `alltd` is the definition of local transformations, that only apply to some specific subterm.

```
transformation =
  alltd(
    trigger-transformation
    ; innermost(A1 <+ ... <+ An)
  )
```

Some relatives of `alltd` that add a strategy to apply on the way up.

```
alldownup2(s1, s2) = rec x((s1 <+ all(x)); s2)
alltd-fold(s1, s2) = rec x(s1 <+ all(x); s2)
```

Finally, the following strategies select the *leaves* of a tree, where the determination of what is a leaf is up to a parameter strategy.

```
leaves(s, is-leaf, skip: a * (a -> a) -> a) =
  rec x((is-leaf; s) <+ skip(x) <+ all(x))

leaves(s, is-leaf) =
  rec x((is-leaf; s) <+ all(x))
```

A spine of a term is a chain of nodes from the root to some subterm. `spinetd` goes down one spine and applies `s` along the way to each node on the spine. The traversal stops when `s` fails for all children of a node.

```
spinetd(s) = s; try(one(spinetd(s)))
spinebu(s) = try(one(spinebu(s))); s
spinetd'(s) = s; (one(spinetd'(s)) + all(fail))
spinebu'(s) = (one(spinebu'(s)) + all(fail)); s
```

Apply `s` everywhere along all spines where `s` applies.

```
somespinetd(s) = rec x(s; try(some(x)))
somespinebu(s) = rec x(try(some(x)); s)
spinetd'(s)    = rec x(s; (one(x) + all(fail)))
spinebu'(s)    = rec x((one(x) + all(fail)); s)
```

While these strategies define the notion of applying along a spine, they are rarely used. In practice one would use more specific traversals with that determine which subterm to include in the search for a path.

TODO: examples

9.2.5. Recursive Patterns (*)

TODO: format checking

TODO: matching of complex patterns

TODO: contextual rules (local traversal)

9.2.6. Dynamic programming (*)

TODO (probably move to dynamic rules chapter)

13.1.10 10. Type Unifying Strategies

In [Chapter 5](#) we have seen combinators for composing *type preserving* strategies. That is, structural transformations in which basic transformation rules don't change the type of a term. Such strategies are typically applied in transformations, which change the structure of a term, but not its type. Examples are simplification and optimization. In this chapter we consider the class of *type unifying* strategies, in which terms of different types are mapped onto one type. The application area for this type of strategy is analysis of expressions with examples such as free variables collection and call-graph extraction.

We consider the following example problems:

- *term-size*: Count the number of nodes in a term
- *occurrences*: Count number of occurrences of a subterm in a term
- *collect-vars*: Collect all variables in expression
- *free-vars*: Collect all *free* variables in expression

These problems have in common that they reduce a structure to a single value or to a collection of derived values. The structure of the original term is usually lost.

We start with examining these problems in the context of lists, and then generalize the solutions we find there to arbitrary terms using generic term deconstruction, which allows concise implementation of generic type unifying strategies, similarly to the generic traversal strategies of [Chapter 5](#).

10.1. Type Unifying List Transformations

We start with considering type-unifying operations on lists.

Sum. Reducing a list to a value can be conveniently expressed by means of a fold, which has as parameters operations for reducing the list constructors. The `foldr/2` strategy reduces a list by replacing each `Cons` by an application of `s2`, and the empty list by `s1`.

```
foldr(s1, s2) =
[]; s1 <+ \ [y|ys] -> <s2>(y, <foldr(s1, s2)> ys) \
```

Thus, when applied to a list with three terms the result is

```
<foldr(s1,s2)> [t1,t2,t3] => <s2>(t1, <s2>(t2, <s2>(t3, <s1> [])))
```

A typical application of `foldr/2` is `sum`, which reduces a list to the sum of its elements. It sums the elements of a list of integers, using 0 for the empty list and `add` to combine the head of a list and the result of folding the tail.

```
sum = foldr(!0, add)
```

The effect of `sum` is illustrated by the following application:

```
<foldr(!0,add)> [1,2,3] => <add>(1, <add>(2, <add>(3, <!0> []))) => 6
```

Note the build operator for replacing the empty list with 0; writing `foldr(0, add)` would be wrong, since 0 by itself is a congruence operator, which basically *matches* the subject term with the term 0 (rather than replacing it).

Size. The `foldr/2` strategy does not touch the elements of a list. The `foldr/3` strategy is a combination of `fold` and `map` that extends `foldr/2` with a parameter that is applied to the elements of the list.

```
foldr(s1, s2, f) =
[]; s1 <+ \ [y|ys] -> <s2>(<f>y, <foldr(s1,s2,f)>ys) \
```

Thus, when applying it to a list with three elements, we get:

```
<foldr(s1,s2)> [t1,t2,t3] => <s2>(<f>t1, <s2>(<f>t2, <s2>(<f>t3, <s1> [])))
```

Now we can solve our first example problem `term-size`. The size of a list is its *length*, which corresponds to the sum of the list with the elements replaced by 1.

```
length = foldr(!0, add, !1)
```

Number of occurrences. The number of occurrences in a list of terms that satisfy some predicate, entails only counting those elements in the list for which the predicate succeeds. (Where a predicate is implemented with a strategy that succeeds only for the elements in the domain of the predicate.) This follows the same pattern as counting the length of a list, but now only counting the elements for which `s` succeeds.

```
list-occurrences(s) = foldr(!0, add, s < !1 + !0)
```

Using `list-occurrences` and a match strategy we can count the number of variables in a list:

```
list-occurrences(?Var(_))
```

Collect. The next problem is to *collect* all terms for which a strategy succeeds. We have already seen how to do this for lists. The `filter` strategy reduces a list to the elements for which its argument strategy succeeds.

```
filter(s) = [] <+ [s | filter(s)] <+ ?[ |<filter(s)>]
```

Collecting the variables in a list is a matter of filtering with the `?Var(_)` match.

```
filter(?Var(_))
```

The final problem, collecting the free variables in a term, does not really have a counter part in lists, but we can mimick this if we consider having two lists; where the second list is the one with the bound variables that should be excluded.

```
(filter(?Var(_)), id); diff
```

This collects the variables in the first list and subtracts the variables in the second list.

10.2. Extending Fold to Expressions

We have seen how to do typical analysis transformations on lists. How can we generalize this to arbitrary terms? The general idea of a folding operator is that it replaces the constructors of a data-type by applying a function to combine the reduced arguments of constructor applications. For example, the following definition is a sketch for a fold over abstract syntax trees:

```
fold-exp(binop, assign, if, ...) = rec f(
  fold-binop(f, binop)
  <+ fold-assign(f, assign)
  <+ fold-if(f, if)
  <+ ... )

fold-binop(f, s) : BinOp(op, e1, e2) -> <s>(op, <f>e1, <f>e2)
fold-assign(f, s) : Assign(e1, e2)    -> <s>(<f>e1, <f>e2)
fold-if(f, s)      : If(e1, e2, e3)    -> <s>(<f>e1, <f>e2, <f>e3)
```

For each constructor of the data-type the fold has an argument strategy and a rule that matches applications of the constructor, which it replaces with an application of the strategy to the tuple of subterms reduced by a recursive invocation of the fold.

Instantiation of this strategy requires a rule for each constructor of the data-type. For instance, the following instantiation defines `term-size` using `fold-exp` by providing rules that sum up the sizes of the subterms and add one (`inc`) to account for the node itself.

```
term-size = fold-exp(BinOpSize, AssignSize, IfSize, ...)

BinOpSize : (Plus(), e1, e2) -> <add; inc>(e1, e2)
AssignSize : (e1, e2)        -> <add; inc>(e1, e2)
IfSize     : (e1, e2, e3)     -> <add; inc>(e1, <add>(e2, e3))
```

This looks suspiciously like the traversal rules in [Chapter 5](#). Defining folds in this manner has several limitations. In the definition of `fold`, one parameter for each constructor is provided and traversal is defined explicitly for each constructor. Furthermore, in the instantiation of `fold`, one rule for each constructor is needed, and the default behaviour is not generically specified.

One solution would be to use the generic traversal strategy `bottomup` to deal with `fold`:

```
fold-exp(s) = bottomup(s)

term-size = fold-exp(BinOpSize <+ AssignSize <+ IfSize <+ ...)

BinOpSize : BinOp(Plus(), e1, e2) -> <add>(1, <add>(e1, e2))
AssignSize : Assign(e1, e2)        -> <add>(e1, e2)
IfSize     : If(e1, e2, e3)        -> <add>(e1, <add>(e2, e3))
```

Although the recursive application to subterms is now defined generically, one still has to specify rules for the default behavior.

10.3. Generic Term Deconstruction

Instead of having folding rules that are specific to a data type, such as


```
BinOpSize : BinOp(op, e1, e2) -> <add>(1, <add>(e1, e2))
AssignSize : Assign(e1, e2) -> <add>(1, <add>(e1, e2))
```

we would like to have a generic definition of the form

```
CSize : c(e1, e2, ...) -> <add>(e1, <add>(e2, ...))
```

This requires generic decomposition of a constructor application into its constructor and the list with children. This can be done using the # operator. The match strategy `?c#(p2)` decomposes a constructor application into its constructor name and the list of direct subterms. Matching such a pattern against a term of the form `C(t1, ..., tn)` results in a match of "C" against `p1` and a match of `[t1, ..., tn]` against `p2`.

```
Plus(Int("1"), Var("2"))
stratego> ?c#(xs)
stratego> :binding c
variable c bound to "Plus"
stratego> :binding xs
variable xs bound to [Int("1"), Var("2")]
```

Crush. Using generic term deconstruction we can now generalize the type unifying operations on lists to arbitrary terms. In analogy with the generic traversal operators we need a generic one-level reduction operator. The `crush/3` strategy reduces a constructor application by folding the list of its subterms using `foldr/3`.

```
crush(nul, sum, s) : c#(xs) -> <foldr(nul, sum, s)> xs
```

Thus, `crush` performs a fold-map over the direct subterms of a term. The following application illustrates what

```
<crush(s1, s2, f)> C(t1, t2) => <s2>(<f>t1, <s2>(<f>t2, <s1>[]))
```

The following Shell session instantiates this application in two ways:

```
stratego> import libstrategolib
stratego> !Plus(Int("1"), Var("2"))
Plus(Int("1"),Var("2"))

stratego> crush(id, id, id)
(Int("1"),(Var("2"),[]))

stratego> !Plus(Int("1"), Var("2"))
Plus(Int("1"),Var("2"))

stratego> crush(!Tail(<id>), !Sum(<Fst>,<Snd>), !Arg(<id>))
Sum(Arg(Int("1")),Sum(Arg(Var("2")),Tail([])))
```

The `crush` strategy is the tool we need to implement solutions for the example problems above.

Size. Counting the number of direct subterms of a term is similar to counting the number of elements of a list. The definition of `node-size` is the same as the definition of `length`, except that it uses `crush` instead of `foldr`:

```
node-size = crush(!0, add, !1)
```

Counting the number of subterms (nodes) in a term is a similar problem. But, instead of counting each direct subterm as 1, we need to count *its* subterms.

```
term-size = crush(!1, add, term-size)
```

The `term-size` strategy achieves this simply with a recursive call to itself.

```
stratego> <node-size> Plus(Int("1"), Var("2"))
2
stratego> <term-size> Plus(Int("1"), Var("2"))
5
```

Occurrences. Counting the number of occurrences of a certain term in another term, or more generally, counting the number of subterms that satisfy some predicate is similar to counting the term size. However, only those terms satisfying the predicate should be counted. The solution is again similar to the solution for lists, but now using `crush`.

```
om-occurrences(s) = s < !1 + crush(!0, add, om-occurrences(s))
```

The `om-occurrences` strategy counts the *outermost* subterms satisfying `s`. That is, the strategy stops counting as soon as it finds a subterm for which `s` succeeds.

The following strategy counts *all* occurrences:

```
occurrences(s) = <add><(s < !1 + !0>, <crush(!0, add, occurrences(s))>>
```

It counts the current term if it satisfies `s` and adds that to the occurrences in the subterms.

```
stratego> <om-occurrences(?Int(_))> Plus(Int("1"), Plus(Int("34"), Var("2")))
2
stratego> <om-occurrences(?Plus(_,_))> Plus(Int("1"), Plus(Int("34"), Var("2")))
1
stratego> <occurrences(?Plus(_,_))> Plus(Int("1"), Plus(Int("34"), Var("2")))
2
```

Collect. *Collecting* the subterms that satisfy a predicate is similar to counting, but now a *list* of subterms is produced. The `collect(s)` strategy collects all *outermost* occurrences satisfying `s`.

```
collect(s) = ! [<s>] <+ crush(![], union, collect(s))
```

When encountering a subterm for which `s` succeeds, a singleton list is produced. For other terms, the matching subterms are collected for each direct subterm, and the resulting lists are combined with `union` to remove duplicates.

A typical application of `collect` is the collection of all variables in an expression, which can be defined as follows:

```
get-vars = collect(?Var(_))
```

Applying `get-vars` to an expression AST produces the list of all subterms matching `Var(_)`.

The `collect-all(s)` strategy collects *all* occurrences satisfying `s`.

```
collect-all(s) =
! [<s> | <crush(![], union, collect(s))>] <+ crush(![], union, collect(s))
```

If `s` succeeds for the subject term combines the subject term with the collected terms from the subterms.

Free Variables. Collecting the variables in an expression is easy, as we saw above. However, when dealing with languages with variable bindings, a common operation is to extract only the *free* variables in an expression or block of statements. That is, the occurrences of variables that are not bound by a variable declaration. For example, in the expression

```
x + let var y := x + 1 in f(y, a + x + b) end
```

the free variables are `{x, a, b}`, but not `y`, since it is bound by the declaration in the `let`. Similarly, in the function definition

```
function f(x : int) = let var y := h(x) in x + g(z) * y end
```

the only free variable is `z` since `x` and `y` are declared.

Here is a free variable extraction strategy for Tiger expressions. It follows a similar pattern of mixing generic and data-type specific operations as we saw in [Chapter 5](#). The `crush` alternative takes care of the non-special constructors, while `ExpVars` and `FreeVars` deal with the special cases, i.e. variables and variable binding constructs:

```
free-vars =
  ExpVars
  <+ FreeVars(free-vars)
  <+ crush(![], union, free-vars)

ExpVars :
  Var(x) -> [x]

FreeVars(fv) :
  Let([VarDec(x, t, e1)], e2) -> <union>(<fv> e1, <diff>(<fv> e2, [x]))

FreeVars(fv) :
  Let([FunctionDec(fdecs)], e2) -> <diff>(<union>(<fv> fdecs, <fv>e2), fs)
  where <map>(?FunDec(<id>,_,_,_))> fdecs => fs

FreeVars(fv) :
  FunDec(f, xs, t, e) -> <diff>(<fv>e, xs)
  where <map>(Fst)> xs => xs
```

The `FreeVars` rules for binding constructs use their `fv` parameter to recursively get the free variables from subterms, and they subtract the bound variables from any free variables found using `diff`.

We can even capture the pattern exhibited here in a generic collection algorithm with support for special cases:

```
collect-exc(base, special : (a -> b) * a -> b) =
  base
  <+ special(collect-exc(base, special))
  <+ crush(![], union, collect-exc(base, special))
```

The `special` parameter is a strategy parameterized with a recursive call to the collection strategy. The original definition of `free-vars` above, can now be replaced with

```
free-vars = collect-exc(ExpVars, FreeVars)
```

10.4. Generic Term Construction

It can also be useful to *construct* terms generically. For example, in parse tree implosion, application nodes should be reduced to constructor applications. Hence build operators can also use the `#` operator. In a strategy `!p1#(p2)`, the current subject term is replaced by a constructor application, where the constructor name is provided by `p1` and the list of subterms by `p2`. So, if `p1` evaluates to `"C"` and `p2` evaluates to `[t1, ..., tn]`, the expression `!p1#(p2)` build the term `C(t1, ..., tn)`.

Imploding Parse Trees. A typical application of generic term construction is the implosion of parse trees to abstract syntax trees performed by `implode-asfix`. Parse trees produced by `sglr` have the form:

```
appl(prod(sorts, sort, attrs([cons("C")])), [t1, ..., tn])
```

That is, a node in a parse tree consists of an encoding of the original production from the syntax definition, and a list with subtrees. The production includes a constructor annotation `cons("C")` with the name of the abstract syntax tree

constructor. Such a tree node should be imploded to an abstract syntax tree node of the form $C(t_1, \dots, t_n)$. Thus, this requires the construction of a term with constructor C given the string with its name. The following implosion strategy achieves this using generic term construction:

```
implode = appl(id, map(implode)); Implode
Implode : appl(prod(sorts, sort, attrs([cons(c)])), ts) -> c#(ts)
```

The `Implode` rule rewrites an `appl` term to a constructor application, by extracting the constructor name from the production and then using generic term construction to apply the constructor.

Note that this is a gross over simplification of the actual implementation of `implode-asfix`. See the source code for the full strategy.

Generic term construction and deconstruction support the definition of generic analysis and generic translation problems. The generic solutions for the example problems term size, number of occurrences, and subterm collection demonstrate the general approach to solving these types of problems.

13.1.11 11. Concrete Object Syntax

Stratego programs can be used to analyze, generate, and transform object programs. In this process object programs are structured data represented by terms. Terms support the easy composition and decomposition of abstract syntax trees. For applications such as compilers, programming with abstract syntax is adequate; only small fragments, i.e., a few constructors per pattern, are manipulated at a time. Often, object programs are reduced to a core language that only contains the essential constructs. The abstract syntax can then be used as an intermediate language, such that multiple languages can be expressed in it, and meta-programs can be reused for several source languages.

However, there are many applications of program transformation in which the use of abstract syntax is not adequate since the distance between the concrete programs that we understand and the abstract syntax trees used in specifications is too large. Even with pattern matching on algebraic data types, the construction of large code fragments in a program generator can become painful. For example, even the following tiny program pattern is easier to read in the concrete variant

```
let d*
  in let var x ta := (e1*) in e2* end
end
```

than the abstract variant

```
Let(d*, [Let([VarDec(x, ta, Seq(e1*))], e2*)])
```

While abstract syntax is manageable for fragments of this size (and sometimes even more concise!), it becomes unpleasant to use when larger fragments need to be specified.

Besides the problems of understandability and complexity, there are other reasons why the use of abstract syntax may be undesirable. Desugaring to a core language is not always possible. For example, in the renovation of legacy code the goal is to repair the bugs in a program, but leave it intact otherwise. This entails that a much larger abstract syntax needs to be dealt with. Another occasion that calls for the use of concrete syntax is the definition of transformation or generation rules by users (programmers) rather than by compiler writers (meta-programmers). Other application areas that require concrete syntax are application generation and structured document (XML) processing.

Hence, it is desirable to have a meta-language that lets us write object-program fragments in the concrete syntax of the object language. This requires the extension of the meta-language with the syntax of the object language of interest, such that expressions in that language are interpreted as terms. In this chapter it is shown how the Stratego language based on abstract syntax terms is extended to support the use of concrete object syntax for terms.

11.1. Instrumenting Programs

To appreciate the need for concrete syntax in program transformation, it is illuminating to contrast the use of concrete syntax with the traditional use of abstract syntax in a larger example. Program *instrumentation* is the extension of a program in a systematic way in order to obtain measurements during run-time. Instrumentation is used, for example, in debugging to get information about the run-time behavior of a program, and in profiling to collect statistics about about run-time and call frequency of program elements. Here we consider a simple instrumentation scheme that instruments Tiger functions with calls to trace functions.

The following Stratego fragment shows rewrite rules that instrument a function `f` such that it prints `f entry` on entry of the function and `f exit` at the exit. The actual printing is delegated to the functions `enterfun` and `exitfun`. Functions are instrumented differently than procedures, since the body of a function is an expression statement and the return value is the value of the expression. It is not possible to just glue a print statement or function call at the end of the body. Therefore, a `let` expression is introduced, which introduces a temporary variable to which the body expression of the function is assigned. The code for the functions `enterfun` and `exitfun` is generated by rule `IntroducePrinters`. Note that the declarations of the `Let` generated by that rule have been omitted.

```
instrument =
  topdown(try(TraceProcedure + TraceFunction))
; IntroducePrinters
; simplify

TraceProcedure :
  FunDec(f, x*, NoTp(), e) ->
  FunDec(f, x*, NoTp(),
    Seq([Call(Var("enterfun"), [String(f)]), e,
        Call(Var("exitfun"), [String(f)])]))

TraceFunction :
  FunDec(f, x*, Tp(tid), e) ->
  FunDec(f, x*, Tp(tid),
    Seq([Call(Var("enterfun"), [String(f)]),
        Let([VarDec(x, Tp(tid), NilExp)],
            [Assign(Var(x), e),
              Call(Var("exitfun"), [String(f)]),
              Var(x)])]))

IntroducePrinters :
  e -> Let(..., e)
```

The next program fragment implements the same instrumentation transformation, but now it uses *concrete syntax*.

```
TraceProcedure :
  |[ function f(x*) = e ]| ->
  |[ function f(x*) = (enterfun(s); e; exitfun(s)) ]|
  where !f => s

TraceFunction :
  |[ function f(x*) : tid = e ]| ->
  |[ function f(x*) : tid =
    (enterfun(s);
      let var x : tid
      in x := e; exitfun(s); x
    end) ]|
  where new => x ; !f => s
```

(continues on next page)

(continued from previous page)

```
IntroducePrinters :
  e -> |[ let var ind := 0
        function enterfun(name : string) = (
          ind := +(ind, 1);
          for i := 2 to ind do print(" ");
          print(name); print(" entry\\n")
        )
        function exitfun(name : string) = (
          for i := 2 to ind do print(" ");
          ind := -(ind, 1);
          print(name); print(" exit\\n")
        )
      in e end ]|
```

It is clear that the concrete syntax version is much more concise and easier to read. This is partly due to the fact that the concrete version is shorter than the abstract version: 225 bytes vs 320 bytes after eliminating all non-significant whitespace. However, the concrete version does not use much fewer lines. A more important reason for the increased understandability is that in order to read the concrete version it is not necessary to mentally translate the abstract syntax constructors into concrete ones. The implementation of `IntroducePrinters` is only shown in concrete syntax since its encoding in abstract syntax leads to unreadable code for code fragments of this size.

Note that these rewrite rules cannot be applied as such using simple innermost rewriting. After instrumenting a function declaration, it is still a function declaration and can thus be instrumented again. Therefore, we use a single pass top-down strategy for applying the rules.

11.2. Observations about Concrete Syntax Specifications

The example gives rise to several observations. The concrete syntax version can be read without knowledge of the abstract syntax. On the other hand, the abstract syntax version makes the tree structure of the expressions explicit. The abstract syntax version is much more verbose and is harder to read and write. Especially the definition of large code fragments such as in rule `IntroducePrinters` is unattractive in abstract syntax.

The abstract syntax version *implements* the concrete syntax version. The concrete syntax version has all properties of the abstract syntax version: pattern matching, term structure, can be traversed, etc.. In short, the concrete syntax is just *syntactic sugar* for the abstract syntax.

Extension of the Meta Language. The instrumentation rules make use of the concrete syntax of Tiger. However, program transformation should not be restricted to transformation of Tiger programs. Rather we would like to be able to handle arbitrary object languages. Thus, the object language or object languages that are used in a module should be a parameter to the compiler. The specification of instrumentation is based on the real syntax of Tiger, not on some combinators or infix expressions. This entails that the syntax of Stratego should be extended with the syntax of Tiger.

Meta-Variables. The patterns in the transformation rules are not just fragments of Tiger programs. Rather some elements of these fragments are considered as meta-variables. For example in the term

```
|[ function f(x*) = e ]|
```

the identifiers `f`, `x*`, and `e` are not intended to be Tiger variables, but rather meta-variables, i.e., variables at the level of the Stratego specification, ranging over identifiers, lists of function arguments, and expressions, respectively.

Antiquotation. Instead of indicating meta-variables implicitly we could opt for an antiquotation mechanism that lets us splice in meta-level expressions into a concrete syntax fragment. For example, using `~` and `~*` as antiquotation operators, a variant of rule `TraceProcedure` becomes:

```
TraceProcedure :
  |[ function ~f(~* x*) = ~e ]| ->
  |[ function ~f(~* x*) =
    (enterfun(~String(f)); ~e; exitfun(~String(f))) ]|
```

With such antiquotation operators it becomes possible to directly embed meta-level computations that produce a piece of code within a syntax fragment.

In the previous section we have seen how the extension of Stratego with concrete syntax for terms improves the readability of meta-programs. In this section we describe the techniques used to achieve this extension.

11.2.1. Extending the Meta Language

To embed the syntax of an object language in the meta language the syntax definitions of the two languages should be combined and the object language sorts should be injected into the appropriate meta language sorts. In the Stratego setting this is achieved as follows. The syntax of a Stratego module `m` is declared in the `m.meta` file, which declares the name of an SDF module. For instance, for modules using Tiger concrete syntax, i.e., using the extension of Stratego with Tiger, the `.meta` would contain

```
Meta([Syntax("StrategoTiger")])
```

thus declaring SDF module `StrategoTiger.sdf` as defining the extension.

The SDF module combines the syntax of Stratego and the syntax of the object language(s) by importing the appropriate SDF modules. The syntax definition of Stratego is provided by the compiler. The syntax definition of the object language is provided by the user. For example, the following SDF module shows a fragment of the syntax of Stratego:

```
module Stratego-Terms
exports
  context-free syntax
    Int          -> Term {cons("Int")}
    String       -> Term {cons("Str")}
    Var          -> Term {cons("Var")}
    Id "(" {Term ","}* ")" -> Term {cons("Op")}
    Term "->" Term -> Rule {cons("RuleNoCond")}
    Term "->" Term "where" Strategy -> Rule {cons("Rule")}
```

The following SDF module `StrategoTiger`, defines the extension of Stratego with Tiger as object language.

```
module StrategoTiger
imports Stratego [ Term => StrategoTerm
                  Var  => StrategoVar
                  Id   => StrategoId
                  StrChar => StrategoStrChar ]
imports Tiger Tiger-Variables
exports
  context-free syntax
    "|[" Dec      "]" -> StrategoTerm {cons("ToTerm"),prefer}
    "|[" TypeDec  "]" -> StrategoTerm {cons("ToTerm"),prefer}
    "|[" FunDec   "]" -> StrategoTerm {cons("ToTerm"),prefer}
    "|[" Exp      "]" -> StrategoTerm {cons("ToTerm"),prefer}
    "~" StrategoTerm -> Exp {cons("FromTerm"),prefer}
    "~*" StrategoTerm -> {Exp " " }+ {cons("FromTerm")}
    "~*" StrategoTerm -> {Exp ";" }+ {cons("FromTerm")}
    "~int:" StrategoTerm -> IntConst {cons("FromTerm")}
```

The module illustrates several remarkable aspects of the embedding of object languages in meta languages using SDF.

A combined syntax definition is created by just importing appropriate syntax definitions. This is possible since SDF is a modular syntax definition formalism. This is a rather unique feature of SDF and essential to this kind of language extension. Since only the full class of context-free grammars, and not any of its subclasses such as LL or LR, are

closed under composition, modularity of syntax definitions requires support from a generalized parsing technique. SDF2 employs scannerless generalized-LR parsing.

The syntax definitions for two languages may partially overlap, e.g., define the same sorts. SDF2 supports renaming of sorts to avoid name clashes and ambiguities resulting from them. In the `StrategoTiger` module several sorts from the `Stratego` syntax definition (`Term`, `Id`, `Var`, and `StrChar`) are renamed since the `Tiger` definition also defines these names. In practice, instead of doing this renaming for each language extension, module `StrategoRenamed` provides a syntax definition of `Stratego` in which all sorts have been renamed.

The embedding of object language expressions in the meta-language is implemented by adding appropriate injections to the combined syntax definition. For example, the production

```
"|[" Exp "]"|"-> StrategoTerm {cons("ToTerm"),prefer}
```

declares that a `Tiger` expression (`Exp`) between `|` `[` and `]` `|` can be used everywhere where a `StrategoTerm` can be used. Furthermore, abstract syntax expressions (including meta-level computations) can be spliced into concrete syntax expressions using the `~` splice operators. To distinguish a term that should be interpreted as a list from a term that should be interpreted as a list *element*, the convention is to use a `~*` operator for splicing a list.

The declaration of these injections can be automated by generating an appropriate production for each sort as a transformation on the SDF definition of the object language. It is, however, useful that the embedding can be programmed by the meta-programmer to have full control over the selection of the sorts to be injected, and the syntax used for the injections.

Using the injection of meta-language `StrategoTerms` into object language `Expressions` it is possible to distinguish meta-variables from object language identifiers. Thus, in the term `| [var ~x := ~e] |`, the expressions `~x` and `~e` indicate meta-level terms, and hence `x` and `e` are meta-level variables.

However, it is attractive to write object patterns with as few squiggles as possible. This can be achieved through another feature of SDF, i.e., variable declarations. The following SDF module declares syntax schemata for meta variables.

```
module Tiger-Variables
exports
  variables
    [s] [0-9]* -> StrConst {prefer}
    [xyzfgh] [0-9]* -> Id {prefer}
    [e] [0-9]* -> Exp {prefer}
    "ta" [0-9]* -> TypeAn {prefer}
    "x" [0-9]* "*" -> {FArg "",""}+ {prefer}
    "d" [0-9]* "*" -> Dec+ {prefer}
    "e" [0-9]* "*" -> {Exp ";" }+ {prefer}
```

According to this declaration `x`, `y`, and `g10` are meta-variables for identifiers and `e`, `e1`, and `e1023` are meta-variables of sort `Exp`. The last three productions declare variables over lists using the convention that these are distinguished from other variables with an asterisk. Thus, `x*` and `x1*` range over lists of function arguments. The `prefer` attribute ensures that these identifiers are preferred over normal `Tiger` identifiers.

Parsing a module according to the combined syntax and mapping the parse tree to abstract syntax results in an abstract syntax tree that contains a mixture of meta- and object-language abstract syntax. Since the meta-language compiler only deals with meta-language abstract syntax, the embedded object-language abstract syntax needs to be expressed in terms of meta abstract syntax. For example, parsing the following `Stratego` rule

```
| [ x := let d* in ~* e* end ] | -> | [ let d* in x := (~* e*) end ] |
```

with embedded `Tiger` expressions, results in the abstract syntax tree


```
Rule (ToTerm (Assign (Var (meta-var ("x")),
    Let (meta-var ("d*"), FromTerm (Var ("e*")))),
    ToTerm (Let (meta-var ("d*"),
        [Assign (Var (meta-var ("x")),
            Seq (FromTerm (Var ("e*"))))]))))
```

containing Tiger abstract syntax constructors (e.g., `Let`, `Var`, `Assign`) and meta-variables (`meta-var`). The transition from meta language to object language is marked by the `ToTerm` constructor, while the transition from meta-language to object-language is marked by the constructor `FromTerm`.

Such mixed abstract syntax trees can be normalized by *exploding* all embedded abstract syntax to meta-language abstract syntax. Thus, the above tree should be exploded to the following pure Stratego abstract syntax:

```
Rule (Op ("Assign", [Op ("Var", [Var ("x")]),
    Op ("Let", [Var ("d*"), Var ("e*")])]),
    Op ("Let", [Var ("d*"),
        Op ("Cons", [Op ("Assign", [Op ("Var", [Var ("x")]),
            Op ("Seq", [Var ("e*")])]),
            Op ("Nil", [])])])])
```

Observe that in this explosion all embedded constructor applications have been translated to the form `Op (C, [t1, ..., tn])`. For example, the Tiger *variable* constructor `Var (x)` becomes `Op ("Var", [x])`, while the Stratego meta-variable `Var ("e*")` remains untouched, and meta-vars become Stratego Vars. Also note how the list in the second argument of the second `Let` is exploded to a `Cons/Nil` list.

The resulting term corresponds to the abstract syntax for the rule

```
Assign (Var (x), Let (d*, e*)) -> Let (d*, [Assign (Var (x), Seq (e*))])
```

written with abstract syntax notations for terms.

The explosion of embedded abstract syntax does not depend on the object language; it can be expressed generically, provided that embeddings are indicated with the `FromTerm` constructor.

Disambiguating Quotations. Sometimes the fragments used within quotations are too small for the parser to be able to disambiguate them. In those cases it is useful to have alternative versions of the quotation operators that make the sort of the fragment explicit. A useful, although somewhat verbose, convention is to use the sort of the fragment as operator:

```
"exp" " | [ " Exp " ] | " -> StrategoTerm {cons ("ToTerm") }
```

Other Quotation Conventions. The convention of using `| [...] |` and `~` as quotation and anti-quotation delimiters is inspired by the notation used in texts about semantics. It really depends on the application, the languages involved, and the *audience* what kind of delimiters are most appropriate.

The following notation was inspired by active web pages is developed. For instance, the following quotation `%> . . . <%` and antiquotation `<% . . . %>` delimiters are defined for use of XML in Stratego programs:

```
context-free syntax
"%>" Content "<%" -> StrategoTerm {cons ("ToTerm"), prefer}
"<%" StrategoTerm "%>" -> Content {cons ("FromTerm") }
"<%" StrategoStrategy "%>" -> Content {cons ("FromApp") }
```

Desugaring Patterns. Some meta-programs first desugar a program before transforming it further. This reduces the number of constructs and shapes a program can have. For example, the Tiger binary operators are desugared to prefix form:

```
DefTimes : |[ e1 * e2 ]| -> |[ *(e1, e2) ]|
DefPlus  : |[ e1 + e2 ]| -> |[ +(e1, e2) ]|
```

or in abstract syntax

```
DefPlus : Plus(e1, e2) -> BinOp(PLUS, e1, e2)
```

This makes it easy to write generic transformations for binary operators. However, all subsequent transformations on binary operators should then be done on these prefix forms, instead of on the usual infix form. However, users/meta-programmers think in terms of the infix operators and would like to write rules such as

```
Simplify : |[ e + 0 ]| -> |[ e ]|
```

However, this rule will not match since the term to which it is applied has been desugared. Thus, it might be desirable to desugar embedded abstract syntax with the same rules with which programs are desugared. This phenomenon occurs in many forms ranging from removing parentheses and generalizing binary operators as above, to decorating abstract syntax trees with information resulting from static analysis such as type checking.

We have seen how the use of concrete object syntax can make the definition of transformation rules more readable.

13.1.12 12. Dynamic Rules

In the previous chapters we have shown how programmable rewriting strategies can provide control over the application of transformation rules, thus addressing the problems of confluence and termination of rewrite systems. Another problem of pure rewriting is the context-free nature of rewrite rules. A rule has access only to the term it is transforming. However, transformation problems are often context-sensitive. For example, when inlining a function at a call site, the call is replaced by the body of the function in which the actual parameters have been substituted for the formal parameters. This requires that the formal parameters and the body of the function are known at the call site, but these are only available higher-up in the syntax tree. There are many similar problems in program transformation, including bound variable renaming, typechecking, data flow transformations such as constant propagation, common-subexpression elimination, and dead code elimination. Although the basic transformations in all these applications can be expressed by means of rewrite rules, these require contextual information.

In Stratego context-sensitive rewriting can be achieved without the added complexity of local traversals and without complex data structures, by the extension of rewriting strategies with scoped dynamic rewrite rules. Dynamic rules are otherwise normal rewrite rules that are defined at run-time and that inherit information from their definition context. As an example, consider the following strategy definition as part of an inlining transformation:

```
DefineUnfoldCall =
  ?|[ function f(x) = e1 ]|
  ; rules(
    UnfoldCall : |[ f(e2) ]| -> |[ let var x := e2 in e1 end ]|
  )
```

The strategy `DefineUnfoldCall` matches a function definition and defines the rewrite rule `UnfoldCall`, which rewrites a call to the specific function `f`, as encountered in the definition, to a `let` expression binding the formal parameter `x` to the actual parameter `e2` in the body of the function `e1`. Note that the variables `f`, `x`, and `e1` are bound in the definition context of `UnfoldCall`. The `UnfoldCall` rule thus defined at the function definition site, can be used at all function call sites. The storage and retrieval of the context information is handled transparently by the underlying language implementation and is of no concern to the programmer.

An overview with semantics and examples of dynamic rewrite rules in Stratego is available in the following publications:

- M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundamenta Informaticae*, 69:1–56, 2005.

An extended version is available as [technical report UU-CS-2005-005](#).

- K. Olmos and E. Visser. Composing Source-to-Source Data-Flow Transformations with Rewriting Strategies and Dependent Dynamic Rewrite Rules. In R. Bodik, editor, 14th International Conference on Compiler Construction (CC'05), volume 3443 of Lecture Notes in Computer Science, pages 204–220. Springer-Verlag, April 2005.

An extended version is available as [technical report UU-CS-2005-006](#)

Since these publications provide a fairly complete and up-to-date picture of dynamic rules, incorporation into this manual is not as urgent as other parts of the language.

13.2 The Stratego Library

13.2.1 1. Introduction

The Stratego Library was designed with one goal in mind: it should contain be a good collection of strategies, rules and data types for manipulating programs. In the previous part of this tutorial, we have already introduced you some of the specific features in the library for doing program manipulation. However, the library also contains abstract data types which are found in almost any library, such as lists, strings, hash tables, sets, file and console I/O, directory manipulation and more. In this chapter, we aim to complete your basic Stratego education by introducing you to how these bread-and-butter data types have been implemented for Stratego.

Beware that the online documentation will display strategies on the form `apply-and-fail(Strategy s, ATerm name, ATerm in-term, ATerm out)`, whereas we adopt the more conventional format in this manual: `apply-and-fail(s | name, in-term, out)`

1.1. Anatomy of the Stratego Library

The organization of the [Stratego library](#) is hierarchical. At the coarsest level of organization, it is divided into packages, whose named as on a path-like form, e.g. `collection/list`. Each package in turn consists of one or several modules. A module is a leaf in the hierarchy. It maps to one Stratego (`.str`) file, and contains definitions for strategies, rules, constructors and overlays. The available packages in the library is listed below.

- `collection/hash-table`
- `collection/list`
- `collection/set`
- `collection/tuple`
- `lang`
- `strategy`
- `strategy/general`
- `strategy/pack`
- `strategy/traversal`
- `system/io`
- `system/posix`
- `term`
- `util`

- `util/config`

As an example, the `collection/list` package consists of the modules `common`, `cons`, `filter`, `index`, `integer`, `lookup`, `set`, `sort`, `zip`. Inside the `sort` module, we find the `qsort` strategy, for sorting lists.

In the remainder of this part of the tutorial, we will present the most important parts of the library, and show their typical usage patterns and idioms. If anything seems unclear, you are encouraged to consult the [online documentation](#) for further details.

13.2.2 2. Arithmetic Operations

In this chapter we introduce strategies for working with numbers. The Stratego runtime provides two kinds of numbers: real numbers and integers. They are both terms, but cannot be used interchangeably. The library strategies described in this chapter also maintain the distinction between real numbers and integers, but many may also be applied to strings which contain numbers.

2.1. Basic Operations

Stratego does not have the normal mathematical syntax for arithmetic operators, such as `+`, `-`, `/` and `*`. These operators are used for other purposes. Instead, the library provides the operators as the strategies, namely `add`, `subt`, `div` and `mul`. Further, there is convenience strategy for integer increment, `inc` and decrement, `dec`.

While the Stratego language operates exclusively on terms, there are different kinds of primitive terms. The runtime maintains a distinction between real numbers and integer numbers. The library mirrors this distinction by providing a family of strategies for arithmetic operations. Arithmetic strategies which work on real numbers end in an `r`, e.g. `addr`, and strategies working on integers end in an `i`, e.g. `subti`. For each arithmetic operator, there is also a type-promoting variant, e.g. `mul`, which will type-promote from integer to real, when necessary. Finally, there are convenience strategies for working on strings containing numbers. For each arithmetic operation, there is a string variant, e.g. `divS`.

The full set of arithmetic operations in Stratego:

```
add,  addr,  addi,  addS
div,  divr,  divi,  divS
mul,  mulr,  muli,  mulS
subt, subtr, subti, subts
```

Using these strategies is straightforward.

```
stratego> <addr> (1.5, 1.5)
3.0000000000000000e+00
stratego> <subti> (5, 2)
3
stratego> <mul> (1.5, 2)
3.0000000000000000e+00
stratego> <inc> 2
3
```

As we can see, the `mul` operator can be applied to a pair which consists of different terms (real and integer). In this case, type promotion from integer to real happens automatically.

Working on Strings

The string variants, e.g. `addS` and `divS` work on strings containing integers. The result in strings containing integers.

```
stratego> <addS> ("40", "2")
"42"
stratego> <divS> ("9", "3")
"3"
```

2.2. Number comparisons

The strategies found in the library for comparing two numbers correspond to the usual mathematical operators for less-than (`lt`), less-than-equal (`leq`), equal (`eq`), greater-than (`gt`), greater-than-or-equal (`geq`). As with the arithmetic strategies, each of these operators comes in an integer variant, suffixed with `i`, a real variant (suffixed by `r`), a string variant (suffixed by `S`) and a type promoting variant without suffix. The full matrix of comparison functions thus looks like:

```
lt,   ltr,  lti,  ltS
gt,   gtr,  gti,  gtS
leq,  leqr, leqi, leqS
geq,  geqr, geqi, geqS
```

A few examples:

```
stratego> <lt> (1.0, 2)
(1.0000000000000000e+00,2)
stratego> <ltS> ("1", "2")
("1", "2")
stratego> <geqS> ("2", "2")
("2", "2")
stratego> <gtr> (0.9, 1.0)
command failed
```

The maximum and minimum of a two-element tuple of numbers can be found with the `max` and `min` strategies, respectively. These do not distinguish between real and integers. However, they do distinguish between numbers and strings; `maxS` and `minS` are applicable to strings.

```
stratego> <max> (0.9, 1.0)
1.0
stratego> <min> (99, 22)
22
stratego> <minS> ("99", "22")
"22"
```

Some other properties of numbers, such as whether a number is even, negative or positive, can be tested with the strategies `even`, `neg` and `pos`, respectively.

2.3. Other Operations

The modulus (remainder) of dividing an integer by another is provided by the `mod` strategy. `gcd` gives the greatest common divisor of two numbers. Both `mod` and `gcd` work on a two-element tuple of integers. The `log2` strategy can be used to find the binary logarithm of a number. It will only succeed if the provided number is an integer and that number has an integer binary logarithm.

```
stratego> <mod> (412, 123)
43
stratego> <gcd> (412, 123)
1
```

(continues on next page)

(continued from previous page)

```
stratego> <log2> 16
4
```

2.4. Random Numbers

The library provides a strategy for generating random numbers, called `next-random`. The algorithm powering this random generator requires an initial “seed” to be provided. This seed is just a first random number. You can pick any integer you want, but it’s advisable to pick a different seed on each program execution. A popular choice (though not actually random) is the number of seconds since epoch, provided by `time`. The seed is initialized by the `set-random-seed` strategy. The following code shows the normal idiom for getting a random number in Stratego:

```
stratego> time ; set-random-seed
[]
stratego> next-random
1543988747
```

The random number generator needs only be initialized with a seed once for every program invocation.

2.5. Summary

In this chapter, we saw that Stratego is different from many other languages in that it does not provide the normal arithmetic operators. We saw that instead, strategies such as `add` and `mul` are used to add and multiply numbers. We also saw which strategies to use for comparing numbers and generating random numbers.

The module `term/integer` contains strategies for working with numbers. Refer to the [library reference documentation](#) for more information.

13.2.3 3. Lists

This chapter will introduce you to the basic strategies for working with lists. The strategies provide functionality for composing and decomposing, sorting, filtering, merging as well as constructing new abstractions from basic lists, such as associative lists.

Every value in Stratego is a term. This is also the case for lists. You can write the list 1, 2, 3 as `Cons (1, Cons (2, Cons (3, Nil)))`, which is clearly a term. Fortunately, Stratego also provides some convenient syntactic sugar that makes lists more readable and easy to work with. We can write the same list as `[1, 2, 3]`, which will be desugared internally in the term above.

3.1. Making heads and tails of it

The most fundamental operations on lists is the ability compose and decompose lists. In Stratego, list composition on “sugared” lists, that is, lists written in the sugared form, has some sugar of its own. Assume `xs` is the list `[1, 2, 3]`. The code `[0|xs]` will prepend a 0 to it, yielding `[0,1,2,3]`. List decomposition is done using the match operator. The code `![0,1,2,3] ; ?[y|ys]` will bind `y` to the head of the list, 0, and `ys` to the tail of the list, `[1, 2, 3]`.

The module `collection/list` contains a lot of convenience functions for dealing with lists. (`collection/list` is contained in the `libstratego-lib` library.) For example, the strategy `elem` will check if a given value is in a list. If it is, the identity of the list will be returned.

```
stratego> import libstratego-lib
stratego> <elem> (1, [2,3,1,4])
[2,3,1,4]
```

Continuing on the above Stratego Shell session, we can exercise some of the other strategies:

```
stratego> <length> [1,2,3,4,5]
5
stratego> <last> [5,6,7,8,9]
9
stratego> <reverse> [1,2,3,4,5]
[5,4,3,2,1]
```

There are two strategies for concatenating lists. If the lists are given as a tuple, use `conc`. If you have a list of lists, use `concat`:

```
stratego> <conc> ([1,2,3], [4,5,6], [7,8,9])
[1,2,3,4,5,6,7,8,9]
stratego> <concat> [[1,2,3], [4,5,6], [7,8,9]]
[1,2,3,4,5,6,7,8,9]
```

The sublist of the first n elements can be picked out with the `take(|n)` strategy:

```
stratego> <take(|3)> [1,2,3,4,5]
[1,2,3]
```

Finally, the `fetch(s)` strategy can be used to find the first element for which s succeeds:

```
stratego> <fetch(?2)> [1,2,3]
2
```

The Stratego library contains many other convenient functions, which are documented in the API documentation.

3.2. Sorting

The list sorting function is called `qsort(s)`, and implements the Quicksort algorithm. The strategy parameter s is the comparator function.

```
stratego> <qsort(gt)> [2,3,5,1,9,7]
[9,7,5,3,2,1]
```

3.3. Associative Lists

Stratego also has library support for associative lists, sometimes known as assoc lists. There are essentially lists of (key, value) pairs, and work like a poor man's hash table. The primary strategy for working with these lists is `lookup`. This strategy looks up the first value associated with a particular key, and returns it.

```
stratego> <lookup> (2, [(1, "a"), (2, "b"), (3, "c")]) => "b"
```

3.4. Pairing Lists

The library also contains some useful strategies for combining multiple lists. The `cart(s)` strategy makes a Cartesian product of two lists. For each pair, the parameter strategy s will be called. In the second case below, each pair will be summed by `add`.

```
stratego> <cart(id)> ([1,2,3],[4,5,6])
[(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]
stratego> <cart(add)> ([1,2,3],[4,5,6])
[5,6,7,6,7,8,7,8,9]
```

Two lists can be paired using `zip(s)`. It takes a tuple of two lists, and will successively pick out the head of the lists and pair them into a tuple, and apply `s` to the tuple. `zip` is equivalent to `zip(id)`.

```
stratego> <zip> ([1,2,3],[4,5,6])
[(1,4),(2,5),(3,6)]
stratego> <zip(add)> ([1,2,3],[4,5,6])
[5,6,7]
```

The inverse function of `zip` is `unzip`.

```
stratego> <unzip> [(1,4),(2,5),(3,6)]
([1,2,3],[4,5,6])
```

There is also `unzip(s)` which like `unzip` takes a list of two-element tuples, and applies `s` to each tuple before unzipping them into two lists.

3.5. Lightweight Sets

In Stratego, lightweight sets are implemented as lists. A set differs from a list in that a given element (value) can only occur once. The strategy `nub` (also known as `make-set`) can be used to make a list into a set. It will remove duplicate elements. The normal functions on sets are provided, among them `union`, `intersection`, `difference` and `equality`:

```
stratego> <nub> [1,1,2,2,3,4,5,6,6]
[1,2,3,4,5,6]
stratego> <union> ([1,2,3],[3,4,5])
[1,2,3,4,5]
stratego> <diff> ([1,2,3],[3,4,5])
[1,2]
stratego> <isect> ([1,2,3],[3,4,5])
[3]
stratego> <set-eq> ([1,2,3],[1,2,3])
([1,2,3],[1,2,3])
```

3.6. Transforming Lists

Element-wise transformation of a list is normally done with the `map(s)` strategy. It must be applied to a list. When used, it will apply the strategy `s` to each element in the list, as shown here. It will return a list of equal length to the input. If the application of `s` fails on one of the elements `map(s)` fails.

```
stratego> <map(inc)> [1,2,3,4]
[2,3,4,5]
```

`mapconcat(s)` is another variant of the element-wise strategy application, equivalent to `map(s)` ; `concat`. It takes a strategy `s` which will be applied to each element. The strategy `s` must always result in a list, thus giving a list of lists, which will be concatenated. A slightly more convoluted version of the above mapping.

If we want to remove elements from the list, we can use `filter(s)`. The `filter` strategy will apply `s` to each element of a list, and keep whichever elements it succeeds on:


```
stratego> <filter(?2 ; !6)> [1,2,3,2]
[6,6]
```

```
stratego> <mapconcat(\ x -> [ <inc> x ] \)> [1,2,3,4]
```

3.7. Folding from the Left and Right

List folding is a somewhat flexible technique primarily intended for reducing a list of elements to a single value. Think of it as applying an operator between any two elements in the list, e.g. going from `[1, 2, 3, 4]` to the result of `1 + 2 + 3 + 4`. If the operator is not commutative, that is `x <op> y` is not the same as `y <op> x`, folding from the left will not be the same as folding from the right, hence the need for both `foldl` and `foldr`.

The `foldr(init, oper)` strategy takes a list of elements and starts folding them from the right. It starts after the rightmost element of the list. This means that if we use the `+` operator with `foldr` on the list `[1, 2, 3, 4]`, we get the expression `1 + 2 + 3 + 4 +`, which obviously has a dangling `+`. The strategy argument `init` is used to supply the missing argument on the right hand side of the last plus. If the `init` supplied is `id`, `[]` will be supplied by default. We can see this from the this trial run:

```
stratego> <foldr>(id, debug)
(4, [])
(3, (4, []))
(2, (3, (4, [])))
(1, (2, (3, (4, []))))
(1, (2, (3, (4, []))))
```

With this in mind, it should be obvious how we can sum a list of numbers using `foldr`:

```
stratego> <foldr(!0, add)> [1,2,3,4]
10
```

The related strategy `foldl(s)` works similarly to `foldr`. It takes a two-element tuple with a list and a single element, i.e. `([x | xs], elem)`. The folding will start in the left end of the list. The first application is `s` on `(elem, x)`, as we can see from the following trial run:

```
stratego> <foldl(debug)> ([1,2,3,4], 0)
(1, 0)
(2, (1, 0))
(3, (2, (1, 0)))
(4, (3, (2, (1, 0))))
(4, (3, (2, (1, 0))))
```

Again, summing the elements of the list is be pretty easy:

```
stratego> <foldl(add)> ([1,2,3,4], 0)
10
```

3.8. Summary

In this chapter we got a glimpse of the most important strategies for manipulating lists. We saw how to construct and deconstruct lists, using `build` and `match`. We also saw how we can sort, merge, split and otherwise transform lists. The strategies for associative lists and sets gave an impression of how we can construct new abstractions from basic lists.

More information about list strategies available can be found in the `collections/list` module, in the [library reference documentation](#).

13.2.4 4 Strings

4.1. Basic String Operations

Strings, like all other primitive data types in Stratego, are terms. They are built with the `build (!)` operator and matched with the `match (?)` operator. Additional operations on and with strings are realized through strategies provided by the Stratego library. The most basic operations on strings provided by the library are concatenation, length computation and splitting. We will discuss operation each in turn.

The library provides two variants of the string concatenation operation. The first, `concat-strings`, takes a list of strings and returns the concatenated result. The second, `conc-strings` takes a two-element tuple of strings and returns the concatenated result:

```
stratego> <concat-strings> ["foo", "bar", "baz"]
"foobarbaz"
stratego> <conc-strings> ("foo", "bar")
"foobar"
```

Once you have a string, you may want to know its length, i.e. the number of characters it contains. There are two equivalent strategies for determining the length of a string. If you come from a C background, you may favor the `strlen` strategy. If not, the `string-length` strategy may offer a clearer name.

The final basic operation on strings is splitting. There is a small family of closely related strategies for this, which all do simple string tokenization. The simplest of them is `string-tokenize(|sepchars)`. It takes a list of characters as its term argument, and must of course be applied to a string.

```
stratego> <string-tokenize(|[' '])> "foo bar baz"
["foo", "bar", "baz"]
```

Another strategy in the tokenizer family is `string-tokenize-keep-all(|sepchars)`. It works exactly like `string-tokenize(|sepchars)`, except that it also keeps the separators that were matched:

```
stratego> <string-tokenize-keep-all(|[' '])> "foo bar baz"
["foo", " ", "bar", " ", "baz"]
```

4.2. Sorting Strings

Even if you don't maintain a phone directory, sorting lists of strings may come in handy in many other enterprises. The strategies `string-sort` and `string-sort-desc` sort a list of strings in ascending and descending order, respectively.

```
stratego> !["worf", "picard", "data", "riker"]
["worf", "picard", "data", "riker"]
stratego> string-sort
["data", "picard", "riker", "worf"]
stratego> string-sort-desc
["worf", "riker", "picard", "data"]
```

If you only have two strings to sort, it may be more intuitive to use the string comparison strategies instead. Both `string-gt` and `string-lt` take a two-element tuple of strings, and return 1 if the first string is lexicographically bigger (resp. smaller) than the second, otherwise they fail.

```
stratego> <string-gt> ("picard", "data")
1
stratego> <string-lt> ("worf", "data")
command failed
```

Not directly a sorting operation, `string-starts-with(|pre)` is a strategy used to compare prefixes of strings. It takes a string as the term argument `pre` and must be applied to a string. It will succeed if `pre` is a prefix of the string it was applied to:

```
stratego> <strings-starts-with(|"wes")> "wesley"
"wesley"
```

4.3. Strings and Terms

We already said that strings are terms. As with terms, we can also deconstruct strings, but we cannot use normal term deconstruction for this. Taking apart a string with `explode-string` will decompose a string into a list of characters. We can then manipulate this character list using normal list operations and term matching on the elements. Once finished, we can construct a new string by calling `implode-string`. Consider the following code, which reverses a string:

```
stratego> !"evil olive"
"evil olive"
stratego> explode-string
[101,118,105,108,32,111,108,105,118,101]
stratego> reverse
[101,118,105,108,111,32,108,105,118,101]
stratego> implode-string
"evilo live"
```

This `explode-string`, `strategy`, `implode-string` idiom is useful enough to warrant its own library strategy, namely `string-as-chars(s)`. The code above may be written more succinctly:

```
stratego> <string-as-chars(reverse)> "evil olive"
"evilo live"
```

Sometimes, in the heat of battle, it is difficult to keep track of your primitive types. This is where `is-string` and `is-char` come in handy. As you might imagine, they will succeed when applied to a string and a character, respectively. A minor note about characters is in order. The Stratego runtime does not separate between characters and integers. The `is-char` must therefore be applied to an integer, and will verify that the value is within the printable range for ASCII characters, that is between 32 and 126, inclusive.

Finally, it may be useful to turn arbitrary terms into strings, and vice versa. This is done by `write-to-string` and `read-from-string`, which may be considered string I/O strategies.

```
stratego> <write-to-string> Foo(Bar())
"Foo(Bar) "
stratego> read-from-string
Foo(Bar)
```

4.4. Strings and Numbers

Another interplay between primitive types in Stratego is between numbers and strings. Converting numbers to strings and strings to numbers is frequently useful when dealing with program transformation, perhaps particularly partial evaluation and interpretation. Going from numbers to strings is done by `int-to-string` and `real-to-string`. Both will accept reals and integers, but will treat its input as indicated by the name.

```
stratego> <int-to-string> 42.9
"42"
```

(continues on next page)

(continued from previous page)

```
stratego> <real-to-string> 42.9
"42.8999999999999999"
```

The resulting number will be pretty-printed as best as possible. In the second example above, the imprecision of floating point numbers results in a somewhat non-intuitive result.

Going the other way, from strings to numbers, is a bit more convoluted, due to the multiple bases available in the string notation. The simplest strategies, `string-to-real` and `string-to-int`, assume the input string is in decimal.

```
stratego> <string-to-real> "123.123"
1.2312300000000000e+02
stratego> <string-to-int> "123"
123
```

For integers, the strategies `hex-string-to-int`, `dec-string-to-int`, `oct-string-to-int` and `bin-string-to-int` can be used to parse strings with numbers in the most popular bases.

13.2.5 5 Hashtables and Sets

5.1. Hashtables

The rewriting paradigm of Stratego is functional in nature, which is somewhat contradictory to the imperative nature of hashtables. Normally, this doesn't present any practical problems, but remember that changes to hashtables "stick", i.e. they are changed by side-effect.

The Stratego hashtable API is pretty straightforward. Hashtables are created by `new-hashtable` and destroyed by `hashtable-destroy`.

```
stratego> import lib
stratego> new-hashtable => h
Hashtable(136604296)
```

The result `Hashtable(136604296)` here is a handle to the actual hashtable. Consider it a pointer, if you will. The content of the hashtable must be retrieved with the `hashtable-*` strategies, which we introduce here. The strategy `hashtable-copy` can be used to copy a hashtable.

Adding a key with value to the table is done with `hashtable-put(|k,v)`, where `k` is the key, `v` is the value. Retrieving the value again can be done by `hashtable-get(|k)`.

```
stratego> <hashtable-put(|"one", 1)> h
Hashtable(136604296)
stratego> <hashtable-get(|"one")
1
```

The contents of the hashtable can be inspected with `hashtable-values` and `hashtable-keys`.

Nesting is also supported by the Stratego hashtables. Using `hashtable-push(|k,v)`, a new "layer" can be added to an existing key (or an initial layer can be added to a non-existing key). Removing a layer for a key can be done with `hashtable-pop(|k)`.

```
stratego> <hashtable-push("one",2)> h
Hashtable(136604296)
stratego> <hashtable-get("one")> h
[2,1]
stratego> <hashtable-pop(|"one")> h
```

(continues on next page)

(continued from previous page)

```
Hashtable(136604296)
stratego> <hashtable-get(|"one")> h
[1]
stratego> <hashtable-remove(|"one")> h
Hashtable(136604296)
stratego> <hashtable-values> h
[]
```

5.2. Indexed Sets

The library provides a rather feature complete implementation of indexed sets, based on hashtables. A lightweight implementation of sets, based on lists, is explained in the [chapter on lists](#).

Similar to hashtables, indexed sets are created with the `new-iset` strategy, copied with `iset-copy` and destroyed with `iset-destroy`.

```
stratego> new-iset => i
IndexedSet(136662256)
```

The resulting term, `IndexedSet(136662256)`, is a handle to the actual indexed set, which can only be manipulated through the `iset-*` strategies.

Adding a single element to a set is done with `iset-add(|e)`, whereas an entire list can be added with the `iset-addlist(|es)`. Its elements can be returned as a list using `iset-elements`.

```
stratego> <iset-addlist(|[1,2,3,4,4])> i
IndexedSet(136662256)
stratego> iset-elements
[1,2,3,4]
```

Notice that the result is indeed a set: every value is only represented once.

Using `iset-get-index(|e)`, the index of a given element `e` can be found. Similarly, `iset-get-elem(|i)` is used to get the value for a particular index.

```
stratego> <iset-get-index(|3)> i
2
stratego> <iset-get-elem(|3)> i
4
```

Note that the indexes start at 0.

The set intersection between two sets can be computed with the `iset-isect(|set2)` strategy. The strategy `iset-union(|set2)` calculates the union of two sets, whereas `iset-subset(|set2)` checks if one set is a subset of another. Equality between two sets is checked by `iset-eq(|set2)`. These strategies are all used in a similar way:

```
stratego> <iset-eq(|set2)> set1
```

A single element can be removed from the set with `iset-remove(|e)`. `iset-clear` will remove all elements in a set, thus emptying it.

13.2.6 6 I/O

This chapter explains the strategies available in the library for controlling file and console I/O.

The need for traditionally file I/O is somewhat diminished for typical applications of Stratego. Normally, Stratego programs are designed to work together connected by Unix pipes. The programs employ `io-wrap` (or similar strategies) that automatically take care of the input and output.

The primitive layer of Stratego I/O inherits its characteristics from Unix. The basic I/O strategies recognize the special files `stdout`, `stdin` and `stderr`. Streams are opened by `fopen` and closed with `fclose`. On top of this, a collection of more convenient strategies have been built.

6.1. Console I/O

The basic strategies for console I/O `print` and `println` are used to write terms to `stdout` or `stderr` (or any other opened file). They both take a tuple. The first element of the tuple is the file to write to, the second is a list of terms. Each term in the list be converted to a string, and these strings will be concatenated together to form the resulting output. The `println` will also append a newline to the end of the resulting string.

The following module should be compiled with `strc`, as usual.

```
module example
imports libstratego-lib
strategies
  main =
    <print> (stdout, ["baz"])
    ; <println> (stdout, [ "foo", 0, "bar" ])
```

After compiling this file, running it will give the following result:

```
$ ./example
bazfoo0bar
$
```

Notice how the string `baz` will be written without a newline (or other space). Also, notice how the terms in the list argument were concatenated.

When using these strategies in the Stratego Shell, some care must be taken when using the `std*` files, as the following example shows.

```
stratego> <println> (stdout(), [ "foo", 0, "bar" ])
foo0bar
```

The shell requires that you put an extra parenthesis after the `stdout`.

The `debug` and `error` are convenience wrappers around `println`. They will always write their result to `stderr`. The `error` strategy is defined as:

```
error =
  where (<println> (stderr, <id>))
```

It is used similarly to the `println` strategy:

```
stratego> <error> ["foo", 0, "bar"]
foo0bar
```

The `debug` strategy accepts any term, i.e. not only lists of terms. The term will be written verbatim:

```
stratego> <debug> [ "foo", 0, "bar" ]
["foo",0,"bar"]
```

6.2. Path and Directory Operations

The library provides a small set of simple file and directory manipulation operations. Assume the directory `/tmp` only contains the files `foo`, `bar`, `baz`. Elementary directory operations can be done as illustrated below:

```
stratego> <readdir> "/tmp"
["foo","bar","baz"]
stratego> <rename-file> ("/tmp/foo", "/tmp/bax")
"/tmp/bax"
stratego> <remove-file> "/tmp/baz"
[]
stratego> <link-file> ("/tmp/bar", "/tmp/foo")
"/tmp/foo"
stratego> <link-file> ("/tmp/bar", "/tmp/foo")
"/tmp/foo"
stratego> <new-temp-dir> "/tmp"
"/tmp/StrategoXTnsGplS"
```

The library contains a family of strategies which must be applied to a `File`, and will return information about it. these include `isdir`, `isatty`, `isfifo` and `islnk` which are predicates checking if a file is a directory, TTY, FIFO or a symbolic link, respectively. To obtain a `File` object in the first place, we should call `file-exists` followed by `filemode`. Thus, checking if `/etc` is a directory is done as follows:

```
stratego> <file-exists ; filemode ; isdir> "/etc"
```

The library also has another family of strategies for getting information about files. These must be applied to a string containing the filename. The family includes `is-executable`, `is-readable` and `is-writeable`.

```
stratego> <is-executable> "/bin/bash"
"/bin/bash"
```

Finally, the directory strategies also include the usual suspects for dealing with paths.

```
stratego> <is-abspath> "../foo"
command failed
stratego> <dirname> "/foo/bar/baz"
"/foo/bar"
stratego> <base-filename> "/foo/bar/baz"
"baz"
stratego> <get-extension> "/tmp/foo.trm"
"trm"
stratego> <abspath> "../foo"
/home/karltk/source/oss/stratego/strategox-manual/trunk/../../foo
```

There are also a few strategies for finding files. We shall describe `find-file(s)`. The other variants of `find-file` are described in the library documentation. The strategy `find-file(s)` finds one file with a specific file extension in a list of directories. It takes a two-element tuple. The first element is a file name as a string, then second element is a list of paths, i.e. `(f, [d*])`. The extension of `f` will be replaced by what is produced by `s`, and the directories given in `[d*]`. Consider the code below.

```
stratego> <find-file(!"rtree")> ("file.str", ["."])
```

This snippet will consider the filename `file.str`, replace its extension with `rtree` and look through the directories in the list `["."]`. Effectively, it will search for `file.rtree` in the current directory.

6.3. File and Text I/O

Opening a file is done with the `fopen` strategy. It takes a two-element tuple, the first element is the filename as a string, the second is the open mode, which is also a string. The most important modes are read (`r`); write (`w`) which opens and empty file for writing, truncating any existing file with the same name; and append (`a`) which appends to the file if it already exists. After all file operations stream have been finished, it should be closed with `fclose`, which will flush and close the file. Explicit flushing can also be done with `fflush`.

It should be pointed out that reading and writing text files with Stratego is rather rare. Normally, text files are read with a parser generated from an SDF description and written using a pretty-printer defined in the Box formalism. In rare cases, this may turn out be too heavy handed, especially if the file format is simplistic and line-based. In this instance, we can come up with an easier solution using `read-text-file` and `read-text-line`.

Assume the file `/tmp/foo` contains the following lines:

```
one
two
three
```

We can read this file in one big chunk into a string with the `read-text-file` strategy, which must be applied to a filename:

```
stratego> <read-text-file> "/tmp/foo"
"one\ntwo\nthree\n"
```

Alternatively, for example if the file is large, we can read it line by line. In this scenario, we must open the file and get a handle to a stream.

```
stratego> <fopen> ("foo.txt", "r") => inp
Stream(136788400)
stratego> <read-text-line> inp
"one"
```

6.4. Term I/O

The primary form of file I/O you will be using in Stratego is reading and writing terms. As explained earlier, the terms are stored on disk as either binary, compressed text or plain text ATerms. Reading a term, no matter which storage format, is done with the `ReadFromFile` strategy. It is applied to a filename.

```
stratego> <ReadFromFile> "/tmp/foo.trm"
Foo(Bar)
```

To write a term to file, you can use `WriteToTextFile` or `WriteToBinaryFile`. The binary format is approximately eight times more space-efficient on average. Both strategies take a two-element tuple where the first element is the filename and second is the term to write. Writing the current term requires a minor twist, which is shown here:

```
stratego> <WriteToBinaryFile> ("/tmp/bar.trm", <id>)
Foo(Bar)
```

It is also possible to read and write terms from and to strings, using `read-from-string` and `write-to-string`. [The chapter on Strings](#) contains explanation of how these strategies work.

6.5. Logging

The strategies for logging are used pervasively throughout the Stratego toolchain. They are easy to use in your own applications, too. The logging system is built on top of the `log(|severity, msg)` and `log(|severity, msg, term)` strategies. It is possible to use these directly, as the following example demonstrates.

```
stratego> import util/log
stratego> log(|Error(), "my error")
```

However, it is preferable to use the high-level wrapper strategies `fatal-err-msg(|msg)`, `err-msg(|msg)`, `warn-msg(|msg)` and `notice-msg(|msg)`. Except for `fatal-err-msg`, these strategies will return with the current term untouched, and write the message as a side effect. The `fatal-err-msg` strategy will also terminate the program with error code 1, after writing the message.

13.3 Concrete Syntax in Stratego Transformations

When writing language-to-language transformations in Stratego, it is possible to use different approaches, for example, writing AST-to-AST transformations. However, to write such transformations, the language engineer needs to know the constructors of both languages. Moreover, AST nodes in rules that define such transformations may contain many nested children, making the work of writing such rules cumbersome and error-prone. Note that Stratego only statically checks the name and arity of constructors, thus, errors would only be detected when pretty-printing the generated AST to the target language.

As an example of this approach, the rule below specifies a transformation of a Calc program to a Java program.

```
program-to-java :
  Program(stats) -> CompilationUnit(None()
    , []
    , [ ClassDeclaration(
        [Public()]
        , Id("Program")
        , None()
        , None()
        , None()
        , [ MethodDecl(
            [Public(), Static()]
            , MethodHeader(
                java-type
                , Id("eval")
                , NoParams()
                , []
                , None()
            )
            , Block(
                [ java-stats*
                ]
            )
          )
        ]
      )
    ]
  )

with
  java-type   := ...
  java-stats* := ...
```

An alternative approach consists of using string interpolation. Instead of generating abstract terms of the target language, transformations generate source code directly, interpolating strings with boilerplate code of the target language and variables defined in the transformation itself. The problem with this approach is that syntax errors in the string fragments of the target language are not detected statically.

Consider the rule shown previously, rewritten below using string interpolation (the code between `$[` and `]`). Note that if the fragment would contain a typo, the syntax error would only be detected after the code had been generated. Note also that one can interpolate Stratego variables with the fragment of the target language by escaping them between `[` and `]`.

```
program-to-java :
  Program(stats) -> $[public class Program {
                        public static [java-type] eval() {
                          [java-stats]
                        }
                      }
  ]
  with
    java-type      := ...
    java-stats     := ...
```

The third option is to use concrete syntax. When using concrete syntax, the transformation is still AST-to-AST but the AST of the target language is abstracted over using the concrete syntax of the language instead. That is, the concrete syntax fragment is parsed internally producing an AST, and that AST is resulted from the transformation.

The same rule defined using concrete syntax is shown below. Note that any syntax error in the fragment would in fact, be detected by the editor, as the fragment is being parsed internally. Moreover, the fragment also has the syntax highlighting of the target language when shown by the editor.

```
program-to-java :
  Program(stats) -> compilation-unit
    [| public class Program {
      public static ~type:java-type eval() {
        ~bstm*:java-stats*
      }
    }
  ]|
  with
    java-type      := ...
    java-stats*    := ...
```

There are two aspects to consider when enabling concrete syntax inside Spoofax. The first one is being able to write Stratego transformations with fragments of a target (or source) language. In other words, the first aspect consists of generating a mixed parse table that embeds the desired target language inside Stratego. The second aspect consists of including the parse table inside an Spoofax project, adding an additional `.meta` file to then enable concrete syntax for a specific Stratego file. Below, we describe both aspects with more detail.

13.3.1 Generating Mixed Parse Tables for a new Language

To generate a mixed parse table that embeds a language inside Stratego, it is necessary to modify the original Stratego grammar, extending it with the desired language. One problem that may occur when combining the grammars of two different languages is name clashing, i.e., non-terminals that have the same name in Stratego and the embedded language. For that reason, the embedding occurs using a modified Stratego grammar, which renames all Stratego context-free non-terminals using parameterized sorts, avoiding name clashes. Since parameterized symbols are not supported in SDF3, this grammar is written in SDF2. The `StrategoMix` grammar can be found [here](#). Inside

Spoofox, though, it is not necessary to have this grammar copied to a project, as it is automatically imported when referenced by another grammar file.

Having the `StrategoMix` grammar as starting point, the next step consists of defining the embedding grammar, the grammar that actually mixes the two languages. A grammar that embeds one language into another may contain three types of productions: productions that define *quotations* for elements of the target language in the host language, productions that define *anti-quotations* back to the host language from the target language, and *variables*, which are shortcuts to anti-quotations, and may appear inside the target language fragments.

When embedding a language into Stratego, it is common to allow fragments of the host language as Stratego terms. For that reason, quotation productions are injected into Stratego terms. For example, the productions below, written in SDF2, indicates that a Java compilation unit can occur in Stratego in a place where a Stratego term can occur.

```
"compilation-unit" "|[" CompilationUnit "]"| " -> Term {cons("ToMetaExpr")}
                  "|[" CompilationUnit "]"| " -> Term {cons("ToMetaExpr")}
```

Note that the first production with constructor `ToMetaExpr` explicitly specifies that the inserted fragment consists of a `compilation-unit`. That is necessary when defining multiple unnamed fragments (second production) for different symbols, which might result in ambiguities.

Due to the renaming that occurs in the `StrategoMix` syntax, we also parameterize the module of the embedding grammar (module `EmbeddingGrammar[E]`), instantiating the symbol `E` later on, according to how `StrategoMix` is imported. That is, instead of writing the rules above using the symbol `Term`, we use the parameter `E` instead. Therefore, the embedding grammar does not depend on `StrategoMix` and should only depend on the grammar of the target language.

```
"compilation-unit" "|[" CompilationUnit "]"| " -> E {cons("ToMetaExpr")}
                  "|[" CompilationUnit "]"| " -> E {cons("ToMetaExpr")}
```

Anti-quotation productions define points to insert elements of the host language inside fragments of the target language. For example, with the production below, we allow Stratego terms to occur in a Java fragment whenever a non-terminal `Type` can occur.

```
"~"          E -> Type {cons("FromMetaExpr")}
"~type:"      E -> Type {cons("FromMetaExpr")}
```

Note that the constructor `FromMetaExpr` indicates that productions represent anti-quotations. Furthermore, note that anti-quotations may also be named after the non-terminal being referenced (e.g., `~type:`).

Using anti-quotations might make the fragment of the target language quite verbose. Therefore, it is also possible to define variables as shortcuts to anti-quotations. For example, the productions below define variables to reference anti-quotations to `Type` fragments. That is, instead of reference to a Stratego variable `X` by using `~type:X`, one may name this variable `t_1` which corresponds to a variable for a non-terminal `Type`.

```
variables
"t_" [0-9\']* -> Type {prefer}
```

The `prefer` annotation indicates that in case of an ambiguity, the variable production should be preferred.

Using the three types of productions above, it is possible to specify which fragments one wants to write using concrete syntax and which symbols may appear inside these fragments as Stratego variables (using anti-quotation or variables with a specific name).

Finally, it is necessary to define third module `Stratego-<LanguageName>` that should import the `StrategoMix` grammar and the embedding grammar, instantiating their parameters accordingly. This module should be defined in a file named `Stratego-<LanguageName>.sdf` and put in the `syntax` folder so that Spoofox can locate it and build the mixed table. That is, if we define the third module for our `Stratego-Java` mixed grammar:

```
module Stratego-Java

imports StrategoMix[StrategoHost]
        EmbeddedJava[Term[[StrategoHost]]]

exports
    context-free start-symbols
        Module[[StrategoHost]]
```

Importing the `StrategoMix` grammar as `StrategoMix[StrategoHost]`, renames all its context-free symbols to `S[[StrategoHost]]`. That is, if we want the quotation, anti-quotation and variable productions to work with `Stratego` terms, we import the embedding grammar as `EmbeddedJava[Term[[StrategoHost]]]`. Note that it is also necessary to redefine the start symbol of the mixed grammar as the new parameterized symbol `Module[[StrategoHost]]`.

After defining the embedding grammar and the `Stratego-<LanguageName>` module, Spoofax generates the mixed table inside the `trans` folder when rebuilding the project.

13.3.2 Using Mixed Parse Tables to Allow Concrete Syntax

Assuming a mixed parse table has been successfully generated or already exists, the next step is to allow concrete syntax in transformations using that table. Thus, it is necessary to first copy the mixed table to the project which will contain `Stratego` files with concrete syntax. The table needs to be in a folder that can be discovered by the `Stratego` compiler, i.e., ideally the `trans` folder of the project that contains `Stratego` files with concrete syntax.

Next, together with the file in which we would like to enable concrete syntax, it is necessary to create a `.meta` file with the same name. That is, to enable concrete syntax in a file `generate.str`, it is necessary to create, in the same directory, an addition file `generate.meta`. This file should indicate which mixed table should be used to parse `generate.str`. For that reason it should contain:

```
Meta([Syntax("<ParseTableName>")])
```

where `ParseTableName` is the filename of the parse table without extension.

With the configuration above, Spoofax automatically detects that the file contains concrete syntax and use that table to parse it. In that file, one may write rules containing concrete syntax as defined by the productions in the mixed grammar.

13.4 Incremental Compilation for Stratego

Warning: This feature has become very unstable in recent releases (2.5.10 and later) and will be replaced by a better tested rewrite.

Note: This feature is fairly new, please [open issues](#) when your build fails or your program has different behaviour under this build setting.

The `Stratego` compiler is usually the slowest part in the build of a Spoofax project. To improve on the development experience, we have added an incremental compilation option for `Stratego`. This can be opted into by editing the `metaborg.yaml` file:

```
dependencies:
  source:
    - org.metaborg:org.metaborg.meta.lang.stratego:${metaborgVersion}
language:
  stratego:
    build: incremental
    format: jar
```

Your file most likely said nothing of the build, meaning it was on the batch setting. The format was probably on ctree. If that is the case you will also need to find the provider setting in your ESV files, likely in editor/Main.esv. Find the ctree provider setting, it should now be:

```
provider: target/metaborg/stratego.jar
```

Note that a clean build using this setting is necessary at first. It will likely take significantly longer than a clean build using the ctree format. All subsequent builds should be faster.

As of Spoofax 2.5.7, there are no known limitations to the incremental compilation setting.

13.4.1 Troubleshooting

Eclipse specific “strategy is undefined” issue: If your project was generated by a Spoofax 2.5.11 or older, the project build path may not be up to date any more. This will result in errors where the compiled strategies are not found. These errors may look like so:

```
Caused by: org.metaborg.core.MetaborgException: Invoking Stratego strategy editor-
↳outline failed, strategy is undefined
```

This error would show up once you open an editor for your language once you’ve built your language with the jar format instead of the ctree format. The solution to the problem is to add the src-gen/java directory to the build path in Eclipse. You can do this in the Package Explorer by right-clicking the src-gen/java directory, selecting “Build Path”, then selecting “Use as Source Folder”. Then clean, rebuild, and strategies should resolve again.

DynSem is a domain specific language for the concise specification of dynamic semantics of programming languages.

14.1 DynSem Language Reference

This is the root of DynSem’s language reference. The documentation is organized by concept/concern rather than by syntactic construct.

14.1.1 Modules

DynSem specifications can be split over multiple files. Every file constituted a **module**. Every module begins with a module declaration and optionally provides subsections:

```
module trans/semantics/mymodule

IMPORTS*

SIGNATURE*

RULES*
```

The name of the module must consist of the path to its file relative to the root of the project, followed by the file name without the *.ds* extension. The example above declares the module `mymodule` in the file *mymodule.ds* which is located at `trans/semantics/` relative to the project root.

Importing modules

Modules can import other modules using one or more **imports** sections. The module below imports modules A, B and C, and shows that multiple modules can be imported at once and that an **imports** section can appear anywhere in the module:

```

module trans/semantics/mymodule

imports
  trans/semantics/A
  trans/semantics/B

rules
  ...

imports
  trans/semantics/C

```

The semantics of imports are those of the `C include directive`, i.e. imports are includes and they are transitive. All signatures and rules defined in (transitively) imported modules are visible for the importing module. Cyclic imports are automatically resolved.

Signatures

A module may have any number of **signature** sections. Signatures are used to declare sorts, constructors, components and arrows. See *Term signatures* for defining sorts and constructors, and *Signatures of arrows, components and meta-functions* for defining components, arrows and other operations.

Rules

Multiple **rules** sections are permitted in a DynSem module. Each rule section declares arbitrarily many reduction rules. See *Reduction rules* about defining reduction rules and the constructs supported.

14.1.2 Term signatures

The **signature** section of a DynSem module provides definitions for program terms and for additional entities used in the specification of a language’s dynamic semantics. We discuss here the subsections of **signature** which deal with representation of terms.

Note: For a discussion of **signature** subsections dealing with operational concepts see *Signatures of arrows, components and meta-functions*.

sorts

Define sorts of program and value terms, separated by white space. For example:

```

sorts Exprs Stmts

```

DynSem has a number of built-in sorts and operations on them: `String`, `Int`, `List`, etc. For a complete list and description of operations available see *Built-in data types*.

constructors

Define constructors for program and value terms. There are two constructor variants:

regular constructors Define regular constructors. Definitions take the form *NAME: {SORT “*”}* -> SORT*, where *NAME* is the name of the constructor, followed by the sorts of the children of the constructor, and where the last *SORT* is the sort of the constructor. Example:


```
constructors
  Plus: Exprs * Exprs -> Exprs
```

implicit constructors Define unary constructors which can be implicitly constructed/deconstructed in pattern matches and term constructions. For example, the constructor:

```
constructors
  OkV: V -> O {implicit}
```

declares the **OkV** unary constructor. In term constructions where a term of sort **O** is expected but a term *t* of sort **V** is provided, the constructor **OkV** is automatically constructed to surround term *t* to become *Ok(t)*. In pattern matches where a term of sort **O** is provided but a term of sort **V** is expected, a pattern match for the term **OkV** is automatically inserted.

sort aliases

Declare sort synonyms. Sort aliases are useful to define shorthands for composed sorts such as for Maps and Lists. For example:

```
sort aliases
  Env = Map(String, Value)
  SciNum = (Float * Int)
```

declares *Env* as a sort alias for *Map(String, Value)*. Wherever the sort *Map(String, Value)* is used, the alias *Env* can be used instead. The example also declares *SciNum* as a sort alias for the pair of a *Float* and an *Int*.

Note: sort-aliases are only syntactic sugar for their aliased sorts and sorts can therefore not be distinguished based on name. For example if two sort aliases *Env1* and *Env2* are defined for *Map(String, Value)* they all become equal and there is no type difference between *Env1* and *Env2*. One can now see *Env1* = *Env2* = *Map(String, Value)*.

variables

Defines variable prefix schemes. Variable schemes take the form *ID = S* and express the expectation that all variables prefixed with *ID* are of the sort *S*. A variable is part of the scheme *X* if it's name begins with *X* and is either followed only by numbers and/or apostrophes, or is followed by *_* followed by any valid identifier. For example given the scheme:

```
variables
  v : Value
```

the following are valid variable names: **v1**, **v2**, **v'**, **v''**, **v1'**, **v_foo**.

Note: Variable schemes can be useful in combination with dynsem implicit reductions to concisely express the expected sort.

native datatypes

These define datatypes implemented natively (in Java) which can be used inside DynSem specifications.

Error: Not documented

14.1.3 Reduction rules

Reduction rules are the principal DynSem mechanism to specify relations (reductions) from program terms to values. Defining rules consists of declaring them as **arrows** and then giving them implementations as rules.

Signatures of arrows, components and meta-functions

components

Define semantic components. A semantic component has a label and a term type. All uses of the component will have a term of that type. All semantic components must be declared before use:

```
signature
components
  E : Env
  H : Heap
```

declares the components *E* and *H* of types *Env* and *Heap*, respectively. The declared components can now be used in arrow declarations and rules. Each semantic component declaration implicitly introduces a variable scheme for the component name and type. The example above implicitly introduces the following variable schemes:

```
signature
variables
  E : Env
  H : Heap
```

for ease of use.

arrows

Declare named reduction relations. Relations in DynSem have to be declared before they are used to define reductions over them. Declarations take the form *S1 -ID-> S2*. Such a declaration makes the relation *-ID->* (where ID is the relation name) available to reduce terms of sort *S1* (input sort) to terms of sort *S2* (output sort). For example, the relation declaration:

```
signature
arrows
  RO* |- Exprs :: RW-IN* -eval-> Values :: RW-IN*
```

declares relation **eval** to relate terms of the **Exprs** sort to terms of the **Values** sort. The declared relation has read-only components **RO*** and read-write components **RW***. Component declarations are optional but they are obeyed. Components associated with arrows are determined by merging the declaration components with those gathered from use sites of the arrows.

Multiple relations with the same name may be declared as long as their input sorts are different. Relations cannot be distinguished by their output sort; it is invalid to define two relations with the same input sort, same name but different output sorts.

Note: It is valid to have multiple identical arrow declarations.

The name-part of the relation declaration may be omitted, such that the following is legal:

```
signature
arrows
  Exprs --> Values
```

meta-functions

Define singleton reductions:

```
signature
arrows
concat(String, String) --> String
```

which can be read as “define meta-function **concat** which reduces two terms of sort **String** to a term of sort **String**”. Semantic components may be declared on meta-functions identically to arrow declarations.

native operators

These are natively defined (in Java) operators.

Error: Not documented

Rules

The rules section of a DynSem module is used to specify inductive definitions for reduction relations of program terms. A rule follows the following scheme:

```
RO* |- PAT :: RW-IN* --> T :: RW-OUT*
where
PREM+
```

For example:

```
E |- Box(e) :: H h --> BoxV(addr) :: H h'
where
E |- e :: H h --> v :: H h';
E |- allocate(v) :: H h' --> addr :: H h'
```

where *PAT* is a pattern match on the input term of the rule. If the pattern match succeeds the rule applies to the term and the variables in the pattern *PAT* are bound in the scope of the rule. *RO** and *RW-IN** are optional comma-separated lists of input semantic components, read-only and read-write, respectively. *PREM+* is a semicolon-separated list of premises that the rule uses to compute the result term *T*. *RW-OUT** is an optional comma-separated list of the read-write semantic components that are outputted from the rule.

premises

Premises are constructs in a rule used by a rule to reduce the input term to the output term.

relation premises Relation premises apply a reduction of a term to a resulting term. They take the form:

```
RO* |- T :: RW-IN* --> PAT :: RW-OUT*
```

*RO** is an optional comma-separated list of read-only semantic components that are propagated into the target relation. *T* is a term construction that builds the input term for the target reduction. Examples of valid term constructions are: variable reference, constructor application, list construction. *RW-IN** is an optional comma-separated list of read-write semantic components that are propagated into the target relation. The elements of *RO** and *RW-IN**, and *T* are all term constructions, i.e. may not contain match symbols or unbound variables. *PAT* is a match pattern applied to the term resulting after the application of the arrow *-->* to the term *T*. *RW-OUT** is an optional comma-separated list of match patterns applied to the read-write semantic components emitted by the applied relation.

A concrete example of a relation premise is:

```
E |- e :: H h --> v :: H h'
```

where the term which variable `e` holds is reduced over the relation `-->` to a term which is stored in variable `v`. The term `E` is a read-only component passed into the reduction. Terms `h`` and ```h'` pass and match, respectively the read-write semantic component labeled `H`.

term equality premise The term equality premise allows checks for equality of two terms. The premise takes the following form:

```
T1 == T2
```

where `T1` and `T2` are the constructions of the two terms whose equality is asserted. The primary use of the equality premise is to determine whether whether two bound variables contain terms that match, but can be used for general purpose equality comparison:

```
a == b;
l == [];
"hello" == s1;
i1 = 42;
b1 == true;
```

pattern-match premise A pattern matching premise is used to perform pattern matching on terms and to bind new variables. The syntax of a premise follows the following form:

```
T => PAT
```

Where `T` is a term construction (e.g. variable reference or constructor application), and `PAT` is the pattern to match against (such as a constructor, term literal, list). All variables in `T` must be bound and none of the variables in `PAT` may be bound. Examples of valid pattern matching premises are:

```
a => b;
a => Plus(e1, e2);
l => [x|xs];
b => Ifz(ec, _, _);
x => 42;
s => "Hello";
```

The pattern matching premise can also be used to bind variables to constructed terms:

```
42 => x;
Plus(a, b) => plusexp;
"hello" => s1;
["hello", "world"] => s2;
```

A special `@` notation allows variables to be bound in nested pattern matches. For example the following premise:

```
exp => Plus(c@Num(_), e@Plus(_, _))
```

both pattern matches the first and second subterms of `Plus` and binds variables `c` and `e`. More precisely the variables `c` and `e` will be bound to `Num` and `Plus` terms, respectively.

Warning: Non-linear pattern matches are not permitted. For example the following are invalid pattern match premises:

```
exp => Plus(e, e);
```

because the pattern on the right hand side contains a variable that is already bound (the second occurrence of `e` is bound by the first occurrence). One can express the behavior intended above using the term equality premise:

```
exp => Plus(e1, e2);
e1 == e2;
```

case pattern matching premise The case pattern matching premise allows behavior to be associated with multiple patterns. It takes the following form:

```
case T of {
  CASE+
}
```

where `T` is a term construction and `CASE+` is a list of cases which may take one the following forms:

```
PAT =>
  PREM*

otherwise=>
  PREM*
```

The first form is for regular pattern matching cases. An example is:

```
case fs of {
  [f | fs'] =>
    f -load-> _;
    fs' -load-> _
  [] =>
}
```

where there are two cases for `fs`, one handling a non-empty list and the other handling an empty list. An example of the `otherwise` case is:

```
Ifz(NumV(ci), e1, e2) --> v
where
  case ci of {
    0 =>
      e1 --> v
    otherwise =>
      e2 --> v
  }
```

where the `otherwise` case is handled if none of patterns of the other cases match. A rule may only have one `otherwise` case and it must be the last case.

Explication of semantic components

Semantic components are used to carry contextual information through reduction rules. They are typically used to model variable environments and heaps, but can be used for anything. A cache store can be propagated as a semantic component for example. There are two kinds of components: read-only (RO) and read-write (RW). The two kinds differ in how they are propagated through the rules. Reduction rules which do not need a particular component can omit mentioning it, it will automatically be propagated to/through rules that need it.

We informally describe how components are implicitly propagated. Let the following (quite abstract) module:

```

module myspec

signature
  sorts
    X
    P1 P2 P3

  components
    A : X
    B : X
    C : X

  constructors
    foo1: P1
    foo2: P2
    bar: P2
    baz: P3

  arrows
    P1 --> Int
    P2 --> Int
    P3 --> Int

rules
  C |- foo1() --> i
  where
    bar() :: C --> i :: C'

  foo2() :: B --> 99 :: B

  bar() --> baz()

  A |- baz() --> 42

```

Semantic components are explicated in the following phases:

1. Gather explicitly used semantic components per arrow
2. Gather inter-arrow dependencies
3. Explicate components in arrow signatures
4. Explicate components in each rule

Whenever a rule explicitly mentions a semantic component it implicitly declares that the arrow which the rule belongs to uses that component. That is the case with rule `foo2()`: it introduces the dependency on RW component B of arrow `P1 --> Int`. Rule `baz()` introduces the dependency of rule `P3 --> Int` on RO component A. The premise of rule `foo1()` introduces a dependency of arrow `P1 --> Int` on RO component C, and a dependency of arrow `P2 --> Int` on RW component C. Note here that one arrow may use a component as RO and (implicitly) pass it to other rules as RW, and viceversa.

Looking only at explicit uses we can explicate the arrow declarations:

```

signature
  ...
  arrows
    C |- P1 :: B --> Int :: B
    P2 :: C --> Int :: C

```

(continues on next page)

(continued from previous page)

```
A |- P3 --> Int
...
```

For ease of referecing, let us label each arrow with an explicit name:

```
signature
...
arrows
  C |- P1 :: B -e1-> Int :: B
  P2 :: C -e2-> Int :: C
  A |- P3 -e3-> Int
...
```

We gather the dependencies between arrows: $-e1-> \Rightarrow -e2->$, $-e2-> \Rightarrow -e3->$. Whenever an arrow $-e1->$ depends on another arrow $-e2->$, the former must provide the required components to the latter, at every invocation. This means that components are inherited from the arrow it depends on. Looking at inter-arrow dependencies, we can propagate the components in the arrow declarations:

```
signature
...
arrows
  C, A |- P1 :: B -e1-> Int :: B
  A |- P2 :: C -e2-> Int :: C
  A |- P3 -e3-> Int
...
```

After explication of components in arrow declarations is complete, we can explicate their uses in every rule, independently:

```
...
rules
  C, A |- fool() :: B --> i :: B
  where
    A |- bar() :: C --> i :: C'

  C, A |- foo2() :: B --> 99 :: B

  A |- bar() :: C --> baz() :: C

  A |- baz() --> 42
```

14.1.4 Built-in data types

DynSem has built-in support for the following types:

- *Numbers*
- *Bool*
- *String*
- *List*
- *Map*
- *Tuple*

Numbers

Premises on numbers

a == b Equality check. Fails if the two numbers are not equal. *a* and *b* may be bound variables or number literals.

a != b Inequality check. Fails if the two numbers are equal. *a* and *b* may be bound variables or number literals.

a => b Equivalent to == if *b* is a number literal. Binds new variable *b* to the value of *a* if *b* is an unbound variable. Invalid if *b* is a bound variable.

a !=> b Equivalent to != if *b* is a number literal. Invalid if *b* is a variable (bound or unbound).

case a of { b => ... } Switch-like premise for multiple cases. *a* must either be a number literal or a bound variable. *b* must either be a number literal or an unbound variable.

One can write reduction rules directly on numbers, for example:

```
module nums

signature
  arrows
    Int -inc-> Int

rules
  3 -inc-> 4
```

Int

The sort *Int* denotes positive and negative whole numbers, e.g. 3, 99, -49. The range of is the same as Java's `int` range: -2,147,483,648 to 2,147,483,647.

Integers can be typed literally, read or written from variables and matched against. *Int* terms are coercible to *Float* where needed.

Long

Similar to *Int*, except for the range being the same as Java's `long` range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

Long values are coercible to *Real* where needed.

Float

The sort *Float* denotes positive and negative decimal numbers, e.g. 3.14, 9.99, -42.42. The range is the same as Java's `float` range. *Float* inherits the precision of Java's `float`.

All *Int* operations are supported on *Float*. Note that due to precision issues equality/match checks between decimal values may unexpectedly fail.

Float values are coercible to *Real* where needed.

Real

The sort `Real` denotes large positive and negative decimal numbers. The range is the same as Java's `double`.

All *Int* operations are supported on `Real`. Note that due to precision issues equality/match checks between decimal vaues may unexpectedly fail.

Bool

The sort `Bool` denotes logical values. Values are either `true` or `false`.

Premises on booleans

a == b Equality check. Fails if the two booleans are not equal. *a* and *b* may be bound variables or boolean literals.

a != b Inequality check. Fails if the two boolean are equal. *a* and *b* may be bound variables or boolean literals.

a => b Equivalent to `==` if *b* is a boolean literal. Binds new variable *b* to the value of *a* if *b* is an unbound variable. Invalid if *b* is a bound variable.

a !=> b Equivalent to `!=` if *b* is a boolean literal. Invalid if *b* is a variable (bound or unbound).

case a of { b => ... } Switch-like premise for multiple cases. *a* must either be a boolean literal or a bound variable. *b* must either be a boolean literal or an unbound variable.

There are no built-in logical operations on the sort `Bool`. One can define meta-functions for these operations, for example:

```
module booleans

signature
  arrows
    not (Bool) --> Bool

rules
  not (true) --> false
  not (false) --> true
```

One can write reduction rules directly on boolean literals, for example:

```
module booleans

signature
  arrows
    Bool -not-> Bool

rules
  true -not-> false
  false -not-> true
```

String

The `String` sort denotes ASCII strings. The maximum length of a string is the maximum size of an `Int`.

string operations

building: “hello”, “hello world” Builds a string from a literal

matching: $a \Rightarrow \text{"hello"}, a \neq \Rightarrow \text{"hello"}$ see $==$ and $!=$ below

$s1 == s2$ Compare strings $s1$ and $s2$ for equality. Fail if the strings are not identical.

$s1 != s2$ Compare strings $s1$ and $s2$ for equality. Fail if the strings are identical.

List

The `List` terms denote list terms of homogeneously typed terms. Use of a list sort must specify the sort of the contained elements. For example, the following declares a constructor `foo` having a list of integers as child:

```
module lists

signature
  sorts
    F
  constructors
    foo: List(Int) -> F
```

list operations

building: `[], [a, b], [a, b | xs]` Build an empty list, a list of two elements and a list of two or more elements, respectively. If a , b and/or xs are variables they must be bound variables. If a is of sort S then b has to be of sort S and xs must be of sort `List (S)` or an empty list literal. Empty list literals - `[]` - are coercible to any list sort.

Examples of list build premises:

```
[] => x
[1, 2, 3] => x
[1, 2 | []] => x
[1, 2 | [3, 4]] => x
[a, b | [a, b]] => x
```

matching: `[], [a, b], [a, b, c | xs], [_ , _ | xs]` Matches an empty list, a list of two elements, a list of three or more elements and a list of two or more elements, respectively. All variables in a list match must be unbound variables. Variables occurring in the pattern will be bound if the entire pattern match succeeds. If any of a , b , c , xs is a term pattern (i.e. not a variable) then a pattern match will be attempted for that pattern.

Examples of list pattern matching premises:

```
t => []
t => [_ , _]
t => [_ , _ | _]
t => [1, 2]
t => [a, b | [c, d | xs]]
```

$l1 == l2$ Check lists $l1$ and $l2$ for equality. Two lists are equal if they are of the same type, they have the same length and being element-wise equal. Premise fails if the two lists are not equal.

$l1 != l2$ Check lists $l1$ and $l2$ for inequality. See above for a definition of list equality. Premise fails if the two lists are equal.

indexed access: `l1[idx]` Retrieves the element at index `idx` in the list `l1`. Fails if the index is out of bounds.

concat: `l1 ++ l2` Concatenates two lists of the same type: if the type of $l1$ is `List (S)`, then the type of $l2$ has to be `List (S)`, unless $l2$ is an empty list literal. The elements in the $l2$ list will be appended, in order, to the elements of $l1$.

reverse: `!l` Reverses the list `l`. Example:

```
`[1, 2, 3] // => [3, 2, 1]
`[] // => []
`[1, 2, 3 | `[4, 5]] // => [4, 5, 3, 2, 1]
```

One may write reduction rules directly on list literals, but the type of the list has to be explicitly specified:

```
module lists

signature
  arrows
    List(Int) -empty-> Bool

rules
  [] : List(Int) -empty-> true

  [_|_] : List(Int) -empty-> false
```

Map

The sort `Map` denotes associative arrays, or dictionaries. A use of the map sort must declare the types of keys and the type of values. The following declares the sort `Env` as an alias to the sort mapping strings to integers:

```
signature
  sort aliases
    Env = Map(String, Int)
```

Note: Maps are immutable. Adding or removing entries from a map does not modify the existent map, instead it creates a new map.

map operations

building: `{}, {k1 l-> v1}, {k1 l-> v1, k2 l-> v2, ...}` Build an empty map, a map with one entry and a map with multiple entries, respectively. All appearing variables must be bound. All keys must be of the same type, and all values must be of the same type. Results in a new map, the old map is unmodified.

extending: `{k1 l-> v1, map}, {map, k1 l-> v1}` Extend the map represented by variable `map` with a new binding from key `k1` to value `v1`. Entries on the left map replace entries with the same key on the right. Multiple additions can be performed at once: `{k1 |--> v1, map, k2 |--> v2}`. This is equivalent to writing: `{k1 |--> v1, {map, k2 |--> v2}}`. Multiple maps can be merged into one: `{map1, map2, map3}`. Again, the left entries replace the right entries. It is equivalent to writing: `{map1, {map2, map3}}`. The result is always a new map, the old map(s) remain(s) unmodified.

removing: `map1 \ k1` Return a new map containing all entries in map `map1` except for the entry with key `k1`. Fails if `map1` has no entry for key `k1`. Map `map1` remains unmodified.

access: `map1[k1]` Return the value associated with key `k1` in map `map1`. Fails if `map1` does not have an entry for key `k1`.

contains: `map1[k1?]` Check whether map `map1` has an entry for key `k1`. Return `true` if an entry exists, `false` otherwise.

matching Pattern matching is **not** possible on maps

Defining rules directly on maps is possible, but the type of the map has to be explicitly specified in the rule:

```
signature
  arrows
    Map(String, Int) -global-> Int

rules
  m : Map(String, Int) -global-> m["global"]
```

Tuple

The `Tuple` sort denotes pairs of terms of arbitrary (higher than 0) arity. Tuple sort usages must be accompanied by declarations of the types of their elements. For example:

```
signature
  sort aliases
    T = (S1 * S2 * S3)
```

declares sort `T` to be an alias of the 3-tuple of terms of type `S1`, `S2` and `S3`, respectively.

tuple operations

building: (t1, t2, ...) Build a tuple literal. All variables appearing in the construction must be bound. If the types of children `t1`, `t2`, ... are `S1`, `S2`, ..., respectively, then the type of the resulting tuple is `(S1 * S2 * ...)`.

matching: (p1, p2, p3) Match a term against a 3-tuple pattern. The patterns `p1`, `p2`, `p3` are matched against the respective child of the incoming tuple. A tuple pattern match succeeds if the term matched is a tuple of equal arity and if the sub-pattern matches succeed. Variables appearing in the pattern must be unbound. Variables appearing in the pattern will be bound if the pattern match succeeds.

It is possible to define rules directly on tuple literals, but the type of the tuple has to be explicitly specified in the rule:

```
signature
  arrows
    (Bool * Bool) -or-> Bool

rules

  (true, _) : (Bool * Bool) -or-> true
  (_, true) : (Bool * Bool) -or-> true
  (false, false) : (Bool * Bool) -or-> false
```

14.1.5 Configuration file

The `dynsem.properties` file specifies configuration parameters for the DynSem interpreter and interpreter generator. Such a file is required for every project from which a DynSem-based interpreter will be derived. The `dynsem.properties` file should be present at the root of the language project. If no properties file can be found a warning will be reported in every DynSem editor for that language.

dynsem.properties

source.language = SIMPL Name of the language. May be any valid Java identifier

source.version = 0.1 Version of the language/semantics. Any valid version, e.g. 1.2.3 is permitted.

source.mimetype = application/x-simpl (optional) mime type for files of this language

source.table = target/metaborg/sdf.tbl (optional) path to parse table for programs in the language.

source.startsymbol = Prog Start symbol for parsing programs of this language.

source.initconstructor.name = Program Constructor name of the term where program reduction begins.

source.initconstructor.arity = 1 Arity of the reduction entry-point constructor.

interpreter.fullbacktracking = false (optional) Enable full backtracking support in the interpreter. If full backtracking is disabled, once the interpreter descends into a reduction premise it is committed to successfully applying one of the rules for that reduction. If full backtracking is enable, the interpreter treats the inability to apply successfully apply a reduction as a regular failure of a pattern match and bails out of the currently evaluated rule to attempt others. In this case, currently evaluated rules are peeled off until a succeeding alternative is found, or the top-level rule is peeled off and the interpreter halts.

interpreter.safecomponents = false (optional) Enables safe semantic components operations. If enabled, all semantic component operations that write or yield a *null* semantic component will cause the interpreter to halt immediately. The same is enforced for referencing a variable which has not been bound. Enabling safe components is a good way to catch bugs in rules with multiple branches.

interpreter.termcaching = false (optional) Enables inline caching of terms and pattern matching results. This can be make a performance difference for programs which are longer running or contain loops. Caching is disabled by default. When enabled every term construction whose subterms are constant will be fetched from a cache instead of recomputed.

interpreter.vargs = (optional) Customize arguments passed to the JVM. For example setting this option to `-ea` will enable assertions in the running JVM. The arguments passed should not be surrounded by quotes.

project.path = ../simpl.interpreter/ Path to the interpreter project. The path must be either relative to the language project or absolute.

project.groupid = org.metaborg Maven Group Identifier for the interpreter project.

project.artifactid = simpl.interpreter Maven Artifact Identifier for the interpreter project.

project.create = true (optional) Enable generation of an interpreter project and associated launch configuration. Defaults to false. When enabled, during generation of the interpreter a project will also be generated including all required directories. A `pom.xml` file will also be created. The project will not be automatically imported in the Eclipse workspace. The generator will also create a launch configuration which can be used in Eclipse.

project.clean = true (optional) Enable cleaning of the target project before writing files. Defaults to false.

project.javapackage = simpl.interpreter.generated (optional) Package to contain all generated Java classes. Defaults to `GROUPID.ARTIFACTID.interpreter.generated`.

project.nativepackage = simpl.interpreter.natives Package name for manually implemented interpreter nodes

project.preprocessor = org.metaborg.lang.sl.interpreter.natives.DesugarTransformer (optional) Fully qualified class name of a custom program pre-processor. The pre-processor will be invoked on the program AST prior to evaluation. Defaults to the identity transformation. See [IdentityTransformer](#) for an example.

project.ruleregistry = org.metaborg.lang.sl.interpreter.natives.SLRuleRegistry (optional) Fully qualified class name of a manually implemented rule registry. Languages which do not provide hand-written rules in Java need not have a custom rule registry. See [SLRuleRegistry](#) from the *SL* language for an example.

project.javapath = src/main/java (optional) Path relative to the interpreter project where Java code will reside.

project.specpath = src/main/resources/specification.aterm (optional) Path in interpreter project for the DynSem specification file.

project.tablepath = src/main/resources/parsetable.tbl (optional) Path in interpreter project for parse table

14.2 DynSem tutorials

The following DynSem tutorials are available:

14.2.1 Getting started

This guide will get you started with DynSem to specify the dynamic semantics of your language. You'll be guided through:

1. *Your first DynSem module*
2. *Context-free semantics*
3. *Context-sensitive semantics*
4. *Conditional semantics*
5. *Semantic libraries using meta-functions*
6. *Semantics of functions*
7. *Get ready for interpretation*
8. *Call into Java code*
9. *Run a program*

Note: DynSem is actively developed and this guide requires features only available in the bleeding edge DynSem. To follow along this tutorial it's best to have the latest Spoofax 2.3.0 nightly build installed.

The SIMPL language

This guide is centered around a very simple language we call *SIMPL*. The *SIMPL* code is kept in a [GitHub SIMPL repository](#). We start with a basic definition (in [SDF3](#)) of concrete syntax which covers arithmetic expressions:

```
1 context-free start-symbols
2   Exp
3
4 context-free syntax
5   Exp.Lit    = <<INT>>
6   Exp.Plus   = <<Exp> + <Exp>> { left }
7   Exp.Minus  = <<Exp> - <Exp>> { left }
8   Exp.Times  = <<Exp> * <Exp>> { left }
9   Exp       = <(<Exp>)> { bracket }
```

Note that terms of the sort `Exp` are start symbols for *SIMPL* programs.

Your first DynSem module

We create the main DynSem module named *trans/simpl* in the file *trans/simpl.ds*:

```
1 module trans/simpl
2
3 imports
4   src-gen/ds-signatures/simpl-sig
```

The module imports the abstract syntax definitions (term signatures) which is generated from the concrete syntax definition:

```

1 module ds-signatures/simpl-sig
2
3 imports ds-signatures/Common-sig
4
5 signature
6   sorts
7     Exp
8   constructors
9     Lit : INT -> Exp
10    Plus : Exp * Exp -> Exp
11    Minus : Exp * Exp -> Exp
12    Times : Exp * Exp -> Exp
    
```

Importing these makes the sorts and constructors available to the rest of the modules. We extend module *trans/simpl* with definitions for value sorts and for the main reduction relation:

```

1 module trans/simpl
2
3 imports
4   src-gen/ds-signatures/simpl-sig
5
6 signature
7   sorts
8     V
9   constructors
10    NumV: Int -> V
11   arrows
12    Exp --> V
13   variables
14    v : V
    
```

We declared constructor `NumV` which will be used to represent numerical value terms. We also declare reduction relation `Exp --> V` from `Exp` terms to values `V`, and a variable scheme for variables named `v`. For details about the signature section of DynSem specification see the [Term signatures](#) and [Signatures of arrows, components and meta-functions](#).

Context-free semantics

We specify reduction rules for *SIMPL* constructs that do not depend on the evaluation contexts (such as environments). These are *number literals*, and simple *arithmetic operations*. The reduction rules are given in a big-step style:

```

1 rules
2   Lit(s) --> NumV(parseI(s)) .
3
4   Plus(e1, e2) --> NumV(addI(i1, i2))
5   where
6     e1 --> NumV(i1) ;
7     e2 --> NumV(i2) .
    
```

The first rule specifies that literal terms such as 42 whose abstract syntax is of the form `Lit("42")` evaluate to `NumV` terms. The second rule specifies the semantics of the addition expressions of the form `Plus(e1, e2)` inductively on the default reduction relation. First the expression `e1` is reduced and the expectation is that it reduces to a `NumV` term. Variable `i1` is bound to the integer value surrounded by the resulting `NumV` term. This is captured in the first

premise of the reduction rule. Similarly, the reduction of the right expression of the addition is captured in the second premise. The conclusion of the rule composes the two integers to a `NumV` term.

In the rules above, `parseI` and `addI` are native operators which we provide the functionality of parsing a string into an integer, and of adding two integers, respectively. We provide the signatures for these when we look at [Call into Java code](#).

Note: Dissimilar to regular big-step style rules, premises in `DynSem` are ordered. The `Plus` rule above states that the left expression will be evaluated first and the right expression second.

The rules for subtraction and multiplication proceed similarly:

```

1 Minus(e1, e2) --> NumV(subI(i1, i2))
2 where
3   e1 --> NumV(i1);
4   e2 --> NumV(i2).
5
6 Times(e1, e2) --> NumV(mulI(i1, i2))
7 where
8   e1 --> NumV(i1);
9   e2 --> NumV(i2).
```

In all three rules seen so far (`Plus`, `Minus`, `Times`) the reductions for the subexpressions can be specified implicitly:

```

1 Plus(NumV(i1), NumV(i2)) --> NumV(addI(i1, i2)).
2 Minus(NumV(i1), NumV(i2)) --> NumV(subI(i1, i2)).
3 Times(NumV(i1), NumV(i2)) --> NumV(mulI(i1, i2)).
```

Specifying the reductions and term expectations implicitly allows rules to be written more concisely without creating ambiguities.

Note: Implicit reductions are applied in left-to-right order and expand to the explicit form of the rules.

Context-sensitive semantics

We define *SIMPL* language constructs whose semantics depend on the evaluation context. First we extend the syntax definition of *SIMPL* with *let*-expressions:

```

1 context-free syntax
2   Exp.Let = <let <ID> = <Exp> in <Exp>> {non-assoc}
3   Exp.Var = <<ID>>
```

This accepts expressions that bind and read variables. An example of a such a program is:

```
let x = 40 in x + 2
```

We expect the program above to evaluate to `NumV(42)` and extend the semantics of *SIMPL* with the following definitions:

```

1 signature
2   sort aliases
3   Env = Map(String, V)
4
```

(continues on next page)

(continued from previous page)

```

5  components
6    E : Env
7
8  rules
9    E |- Let(x, e1, e2) --> v2
10   where
11     E |- e1 --> v1;
12     E {x |--> v1, E} |- e2 --> v2.
13
14   E |- Var(x) --> E[x].

```

The signature `sort aliases` subsection defines `Env` as an alias for an associative array from `String` to `V`. We use this associative array as the evaluation context for variables - variable environment. The `signature components` subsection defines `E` as a semantic component of type `Env`. This labelled component (which in our case holds an environment) will be propagated downwards in the evaluation tree.

Looking at the first rule, it reduces a `Let` term to a value by first reducing the variable expression in the surrounding environment and then reducing the body expression in the updated environment. The variable environment `E` is received into the reduction rule together with the `Let` expression to be reduced, and it is propagated downwards in the evaluation tree of the premises. Updates to the environment are not visible upwards in the evaluation tree. The second rule reduces `Var` expressions to the value associated with the variable name in the variable environment.

Note: Terms left of the `| -` symbol are called *read-only semantic components*.

Although we have extended *SIMPL* with context-sensitive constructs we do not have to modify the reduction rules which are context-independent. `DynSem` reduction rules do not need to explicitly propagate semantic components that they do not depend on.

We illustrate the principle of implicit propagation by further extending *SIMPL* with mutable variable boxes:

```

1  context-free syntax
2    Exp.Box = <box(<Exp>)>
3    Exp.Unbox = <unbox(<Exp>)>
4    Exp.Setbox = <setbox(<Exp>, <Exp>)>

```

This accepts programs that use mutable variables. The `Box` expression allocates a new box on the heap and puts the result of the expression in the box, evaluating to a box value. The `Unbox` expression reads the value inside the box provided by the argument expression. The `Setbox` expression puts the value of the second expression inside the box provided by the first expression. For example, a valid program could be:

```
let b = box(40) in setbox(b, unbox(b) + 2)
```

We extend the `DynSem` specification with the following signature and reduction rules for box operations:

```

1  signature
2    constructors
3      BoxV: Int -> V
4    sort aliases
5      Heap = Map(Int, V)
6    components
7      H : Heap
8
9  rules
10   Box(e) :: H --> BoxV(addr) :: Heap {addr |--> v, H'}
11   where

```

(continues on next page)

(continued from previous page)

```

12  e :: H --> v :: H';
13  fresh => addr.
14
15  Unbox(e) :: H --> H'[addr] :: H'
16  where
17    e :: H --> BoxV(addr) :: H'.
18
19  Setbox(box, e) :: H --> v :: Heap {addr |--> v, H''}
20  where
21    box :: H --> BoxV(addr) :: H';
22    e :: H' --> v :: H''.

```

where `BoxV` is a new *SIMPL* value representing the address of a box in the heap *H*. The `Box` reduces to a `BoxV` value by reducing the subexpression to a value, obtaining a new unoccupied address using the `fresh` primitive. It extends the incoming `Heap` with a new entry for the evaluated expression at the new address. The `Unbox` rule reduces the subexpression to a box value and looks up the associated value in the *H*.

Note: Terms to the right side of `::` symbol are called *read-write semantic components*. They are woven through the evaluation tree and updates to them are made visible upwards in the evaluation tree.

Similarly to the addition of the *let*-expression, extending with a heap structure and mutable variables does not require changing the existing reduction rules. Rules do not have to explicitly mention (or handle) read-write components which they do not depend on.

Conditional semantics

We illustrate how to specify the semantics of a conditional language construct by introducing an `ifz` expression in *SIMPL*. Extend the syntax definition of *SIMPL* with the following:

```

1  context-free syntax
2  Exp.Ifz = <ifz <Exp> then <Exp> else <Exp>>

```

The `ifz` expression executes the `then` expression if the condition expression evaluates to 0, or executes the `else` expression otherwise. An example of a valid *SIMPL* program is:

```

1  let iszero = a -> ifz(a) then 1 else 0
2  in iszero(42)

```

We extend the semantics with the following DynSem rule:

```

1  rules
2  Ifz(NumV(ci), e1, e2) --> v
3  where
4    case ci of {
5      0 =>
6        e1 --> v
7      otherwise =>
8        e2 --> v
9    }.

```

The condition expression is first evaluated to a `NumV`. The two cases of interest are handled using the case pattern matching premise (*Rules*).

Semantic libraries using meta-functions

To keep reduction rules concise and simple it is useful to introduce layers of abstraction over common semantic operations. For example, in the case of *SIMPL* we can abstract away from much of the operations that depend on the variable environment and the heap. Instead of directly manipulating the heap and environment in the reduction rules of the *SIMPL* expressions one can define *meta-functions* to encapsulate heap and environment operations. The *meta-functions* introduced can be reused in all places where access to the environment or heap is required.

Note: *Meta-functions* declarations are 2-in-1 auxiliary constructors and relation declaration used for library abstractions. They benefit from implicit propagation of semantic components just like regular reduction rules. See [Signatures of arrows, components and meta-functions](#) for details on how they are declared.

To create the abstractions we first define a module to hold the sort declaration for *V* and the variable scheme *v*:

```

1 module trans/runtime/values
2
3 signature
4   sorts
5     V
6
7   variables
8     v : V

```

Note: Read about *variable schemes* in the `dynsem_reference_signatures` section.

These declarations can be imported in the rest of the specification. We define the environment meta-functions:

```

1 module trans/environment
2
3 imports
4   trans/runtime/values
5
6 signature
7   sort aliases
8     Env = Map(String, V)
9
10  components
11    E : Env
12
13  arrows
14    bindVar(String, V) --> Env
15    readVar(String) --> V
16
17  rules
18
19    E |- bindVar(x, v) --> {x |--> v, E}.
20
21    E |- readVar(x) --> E[x].

```

And declare the `bindVar` and `readVar` *meta-functions* which update the environment with a new binding and read the associated value, respectively. Note in the highlighted declaration lines the `-->` arrow marking the constructor declaration as *meta-functions*. Similarly, define meta-functions for heap operations:

```

1 module trans/runtime/store
2
3 imports
4   trans/runtime/values
5
6 signature
7   sort aliases
8     Heap = Map(Int, V)
9
10  components
11    H : Heap
12
13  arrows
14    read(Int) --> V
15    allocate(V) --> Int
16    write(Int, V) --> V
17
18  rules
19
20    read(addr) :: H --> H[addr] .
21
22    allocate(v) --> addr
23  where
24    fresh => addr;
25    write(addr, v) --> _ .
26
27    write(addr, v) :: H --> v :: H {addr |--> v, H} .

```

And declare *meta-functions* `allocate`, `read`, `write`, which create a box, read the contents of a box and update the contents of the box, respectively. Note that since the `allocate` rule does not access the `Heap` locally it can be left implicit. We can use the *meta-functions* to re-specify the semantics of the context-sensitive *SIMPL* constructs:

```

1 rules
2   Let(x, v1, e2) --> v2
3   where
4     E bindVar(x, v1) |- e2 --> v2.
5
6   Var(x) --> readVar(x) .

```

By using the semantic abstractions over the environment the rules become more concise and do not depend on specific implementations. Note that because the environment does not have to be explicitly propagated the rules can rely on *implicit reductions*. The rules above automatically expand to their fully explicated variants. During the expansion first the implicit reductions are lifted:

```

1 rules
2   Let(x, v1, e2) --> v2
3   where
4     bindVar(x, v1) --> E';
5     E' |- e2 --> v2.
6
7   Var(x) --> v
8   where
9     readVar(x) --> v.

```

Secondly the semantic components (read-only and read-write) are explicated:

```

1 rules
2   E |- Let(x, v1, e2) --> v2
3   where
4     E |- bindVar(x, v1) --> E';
5     E' |- e2 --> v2.
6
7   E |- Var(x) --> v
8   where
9     E |- readVar(x) --> v.

```

Note: The performance of derived interpreters is **not** adversely affected by the introduction and use of *meta-functions*.

Rules for boxes can be re-specified in a similar way to those for environments:

```

1 rules
2   Box(v) --> BoxV(allocate(v)).
3
4   Unbox(BoxV(addr)) --> read(addr).
5
6   Setbox(BoxV(addr), v) --> write(addr, v).

```

Semantics of functions

We grow *SIMPL* with functions. Functions will be first class citizens *SIMPL* but will only take a single argument (will be unary). We define syntax for function declaration and application:

```

1 context-free syntax
2   Exp.Fun = [[ID] -> [Exp]]
3   Exp.App = <<Exp> (<Exp>) > {left}

```

Now programs such as the following are syntactically correct in *SIMPL*:

```

let sum = a -> b -> a + b
in sum(40) (2)

```

From an execution perspective we expect the above program to evaluate to `NumV(42)` by first applying function `sum` to number 40 which evaluates to a function which is applied to number 2. Functions are only associated to names via the *let*-expression, so anonymous functions literals are allowed. The program below is equivalent to the program above:

```

(a -> b -> a + b) (40) (2)

```

From a dynamic semantics point of view we add a new type of value - `ClosV` - which closes a function body over its declaration environment. A function application reduces the function expression to a `ClosV` and the application of the closure body to the argument:

```

1 signature
2   constructors
3     ClosV: String * Exp * Env -> V
4
5   rules
6     E |- Fun(x, e) --> ClosV(x, e, E).
7

```

(continues on next page)

(continued from previous page)

```

8   App(ClosV(x, e, E), v1) --> v2
9   where
10    E  |- bindVar(x, v1) --> E';
11    E' |- e --> v2.

```

Get ready for interpretation

To get a functioning interpreter derived from a DynSem specification one has to go through the following steps:

1. *A reduction entry-point*
2. *Configure the interpreter generator*
3. *Generate interpreter components*

A reduction entry-point

The *SIMPL* interpreter must have a clearly defined entry point. The entry point is a reduction rule over a relation named `-init->`. The relation named `-init->` should satisfy all semantic components of the arrows it applies. By default `-init->` is the relation invoked by the interpreter at startup. First we extend the syntax definition with a constructor for the top-level of a program:

```

1   context-free start-symbols
2     Prog
3
4   context-free syntax
5     Prog.Program = Exp

```

Term of sort `Prog` are top-level terms in *SIMPL* and reduction of a program should start at the only one possible `Program`.

```

1   signature
2     arrows
3     Prog -init-> V
4
5   rules
6     Program(e) -init-> v
7     where
8     E {} |- e :: H {} --> v :: H _.

```

We extend the DynSem specification with a declaration of the arrow `-init->` reducing terms of sort `Prog` to a value. `Program` is the only term of sort `Prog` and we specify its reduction to value. This reduction rule introduces initial values for the variable environment `E` and for the heap `H`.

Configure the interpreter generator

To configure the interpreter generator with the specifics of *SIMPL* you will need a *dynsem.properties* file:

Note: The *dynsem.properties* file must be located in the root directory of the *SIMPL* language project, not the interpreter project

```

1  # the name of the language. may not contain hyphens
2  source.langname = simpl
3
4  # version of this language
5  source.version = 0.1
6
7  # start symbol to use for parsing programs in this language
8  source.startsymbol = Prog
9
10 # constructor name/arity of reduction entry point
11 source.initconstructor.name = Program
12 source.initconstructor.arity = 1
13
14 # path to interpreter project, absolute or relative to the language project
15 project.path = ../simpl.interpreter/
16
17 # (optional) enable/disable creation of the target project
18 project.create = true
19
20 # (optional) enable/disable cleaning of the target project before writing files
21 project.clean = true
22
23 # groupid & artifactid for the interpreter project
24 project.groupid = org.metaborg
25 project.artifactid = simpl.interpreter
26
27 # package name for manually implemented interpreter nodes
28 project.nativepackage = simpl.interpreter.natives

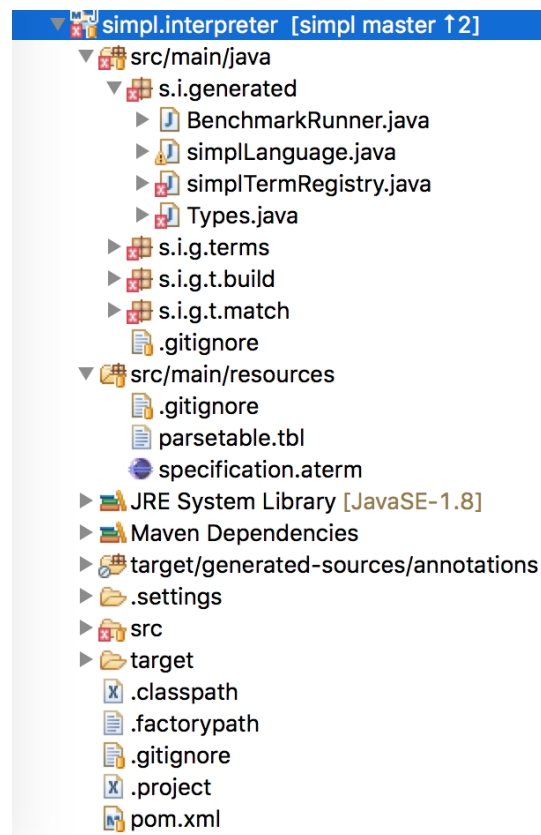
```

The first fragment (lines 1-3) configures the language name, a version identifier and the MIME-TYPE. Line 5 configures the path to the parse table for *SIMPL*, relative to the project, which will be copied into the interpreter project. Line 6 configures the start symbol used to parse *SIMPL* programs and it has to be one of the start symbols specified in the syntax definition. Lines 7-8 specify the constructor name and arity to be used as the entry point for the evaluation. It is expected that an `-init->` rule is declared for this term. For *SIMPL* the top-level term and rule are the ones defined in *A reduction entry-point*.

The third fragment (lines 10-15) sets parameters for the target interpreted project. `project.path` gives the path to the interpreter project. This must be a path relative to the language project, in this case to the *SIMPL* project. The `project.clean` flag indicates whether the target generation directory should be recursively removed (clean compilation target) before generation. If this property is not mentioned in *dynsem.properties*, it defaults to **false**. For a detailed explanation of all valid properties consult the *Configuration file* reference.

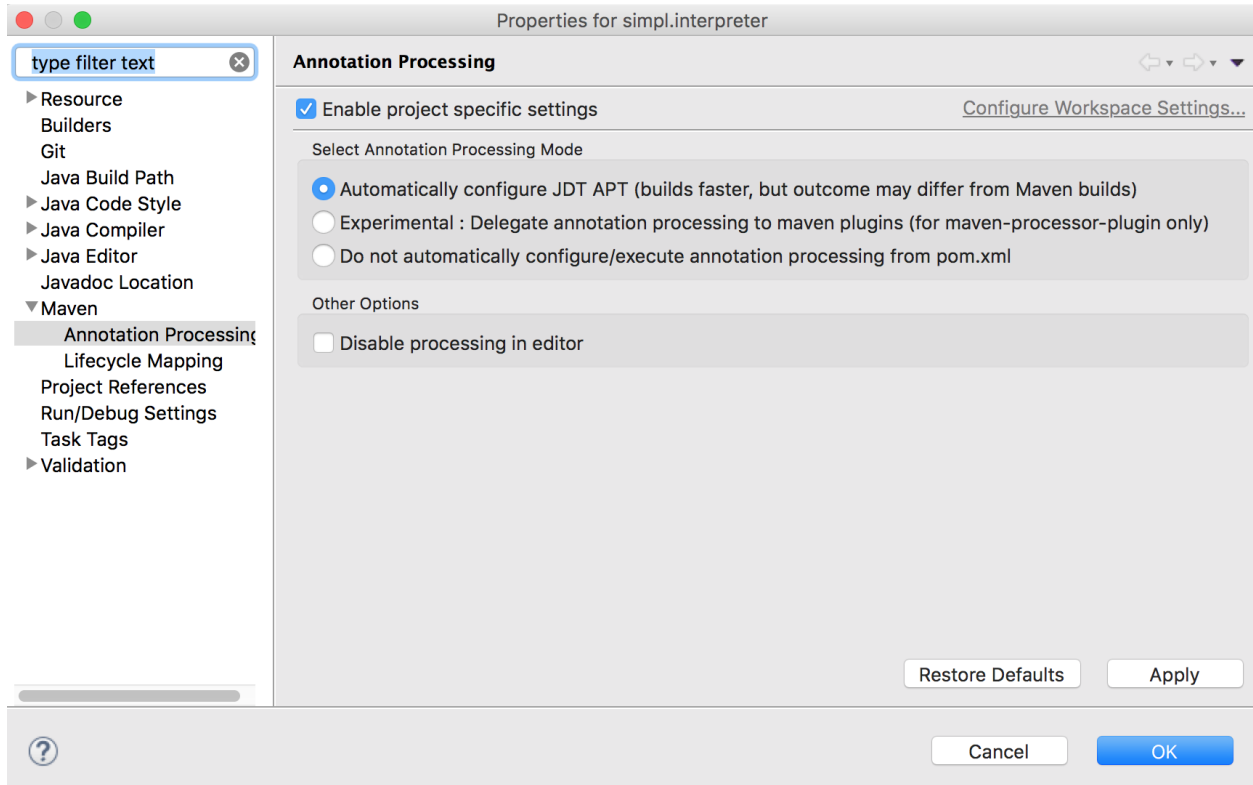
Generate interpreter components

An interpreter derived from a DynSem specification relies on components that are generated from the specification. This generation project happens on-demand. Ensure that the *SIMPL* language project is built and that you have the *SIMPL* interpreter project open in the Eclipse workspace. Open the top-level DynSem specification file - *simpl.ds* - and select `.`. Observe that files have been placed into the *SIMPL* interpreter project:



The *src/main/java* directory contains the *SIMPL*-specific generated term library. The *src/main/resources* directory contains the *SIMPL* parse table (*parsetable.tbl*) and an interpretable form of the DynSem specification (*specification.aterm*).

Note: At this stage it is normal that the project contains Java errors about the missing *simpl.interpreter:natives* package. We will populate this package with native operations (see [Call into Java code](#)). If other errors are reported make sure you have enabled and configured annotation processing:



Warning: If the entry is not available it means you probably do not have the [M2E-APT Eclipse plugin](#) installed. Install it from the Eclipse Marketplace and try again to configure the project by selecting in the window pane opened by selecting .

Call into Java code

Many times a semantics for a language will depend on operations whose specification/implementation will reside outside of the formal specification. In the case of the *SIMPL* language such operation are the conversion of a string representation of a number to a number literal, arithmetic operations, and the *fresh* address generator. More complex languages will require interactions with existent systems such as application of library functions. DynSem specifications can interact with specification-external (native) operations by means of *native operators*. Although we have used native operators for arithmetic operations in *SIMPL*, this guide has so far omitted their signature declaration:

```

1 signature
2   native operators
3     parseI: String -> Int
4     addI: Int * Int -> Int
5     subI: Int * Int -> Int
6     mulI: Int * Int -> Int

```

Line 3 declares the `parseI` native operator which takes one argument of type `String` and produces an `Int`. For a detailed explanation of the `native operators` signature section consult the [Term signatures](#) page.

We now provide an implementation for `parseI` and for `addI`. Create the package `simpl.interpreter.natives`. This package has to be same as the one specified in the `target.nativepackage` property in [Configure the interpreter generator](#). Inside the package create an abstract class named `parseI_1`:

```

1 package simpl.interpreter.natives;
2
3 import org.metaborg.meta.lang.dynsem.interpreter.nodes.building.TermBuild;
4
5 import com.oracle.truffle.api.dsl.NodeChild;
6 import com.oracle.truffle.api.dsl.Specialization;
7 import com.oracle.truffle.api.source.SourceSection;
8
9 @NodeChild(value = "stringbuild", type = TermBuild.class)
10 public abstract class parseI_1 extends TermBuild {
11
12     public parseI_1(SourceSection source) {
13         super(source);
14     }
15
16     @Specialization
17     public int doInt(String s) {
18         return Integer.parseInt(s);
19     }
20
21     public static TermBuild create(SourceSection source, TermBuild stringbuild) {
22         return parseI_1NodeGen.create(source, stringbuild);
23     }
24 }

```

The class name `parseI_1` is required: it's the name of the constructor (*parseI*) followed by `_` and its arity (*I*). The class extends DynSem's `TermBuild` class which corresponds to DynSem fragments that construct terms. The `@NodeChild` annotation is a Truffle annotation declaring a child to our class, named `stringbuild` of the `TermBuild` type. This child corresponds to the sole argument of the `parseI` constructor.

The class is abstract as we rely on Truffle's annotation processor to generate a concrete class named `parseI_1NodeGen`. The method declaration at line 17 implements the business logic of the `parseI` node. It receives one argument corresponding to the evaluated `stringbuild` child and relies on the Java standard library to parse the string to an integer.

The method declared on line 21 is a factory method instantiating the generated subclass of `parseI_1`. The generated language specific library uses this method to obtain instances of the `parseI_1` term build.

In a similar way create an implementation for the `addI` native operator with arity 2:

```

1 package simpl.interpreter.natives;
2
3 import org.metaborg.meta.lang.dynsem.interpreter.nodes.building.TermBuild;
4
5 import com.oracle.truffle.api.dsl.NodeChild;
6 import com.oracle.truffle.api.dsl.NodeChildren;
7 import com.oracle.truffle.api.dsl.Specialization;
8 import com.oracle.truffle.api.source.SourceSection;
9
10 @NodeChildren({ @NodeChild(value = "left", type = TermBuild.class),
11                @NodeChild(value = "right", type = TermBuild.class) })
12 public abstract class addI_2 extends TermBuild {
13
14     public addI_2(SourceSection source) {
15         super(source);
16     }
17
18     @Specialization

```

(continues on next page)

(continued from previous page)

```

19  public int doInt(int left, int right) {
20      return left + right;
21  }
22
23  public static TermBuild create(SourceSection source, TermBuild left,
24      TermBuild right) {
25      return addI_2NodeGen.create(source, left, right);
26  }
27
28  }

```

The significant difference to `parseI` is that `addI` has two children. Using the `@NodeChildren` Truffle annotation multiple child fields can be specified, in this case `left` and `right`. Both of the children are expected to evaluate to integers, expectation made explicit in the method declaration of line 19. The factory method of line 23 receives two children arguments, reflecting the arity of the `addI` constructor. The *SIMPL* interpreter project should have no errors once all required native operators are defined.

Note: The implementation for the other native operators used by *SIMPL* can be found in the repository at [tags/native-operators](#).

Run a program

After following through the previous steps the *SIMPL* interpreter is ready to be imported into the workspace and begin to evaluate programs. Before continuing import the generated interpreter project into the workspace by .

Create a simple program and save it as *simple/examples/ex1.smpl*:

```

let sum = a -> b -> a + b
in sum(40) (2)

```

A launch configuration for the interpreter project has automatically been generated. It can evaluate files that are selected or open in Eclipse. Either select the *ex1.smpl* file in the Package Explorer or open it in the editor. With focus on the example file, evaluate it by selecting , select in the left hand side pane and select the *smpl* launch configuration.

Press . Observe the result of evaluating the program in the Console view.

14.2.2 Run an interpreter as a daemon

Interpreters derived from DynSem specifications can be run as daemons. They run as a background process which accepts client requests for program evaluation. Running interpreters in the background significantly reduces the startup time of the interpreter.

Requirements

The daemon mode uses [Nailgun for Java](#) to launch background processes and make requests. On OS X you can install Nailgun using [Homebrew](#):

```
brew install nailgun
```

Launching an interpreter daemon

If you have enabled generation of the interpreter project in *dynsem.properties*:

```
project.create = true
```

upon generation of an interpreter three files will be generated in the root of the interpreter project. Assuming your language is named **simpl** these files will be:

- *simpl-server*
- *simpl-client*
- *simpl (Daemon).launch*

Launch an interpreter daemon by running the *simpl-server* shell script or by launching the *simpl (Daemon)* launch configuration from Eclipse. To quit the daemon simply quit the process or run the stop command from a shell:

```
ng ng-stop
```

To evaluate a program you can either use the *simpl-client*:

```
./simpl-client yourprogram.smpl
```

or invoke the *ng* command:

```
ng simpl yourprogram.smpl
```

Warning: Nailgun daemon are not secure. For more information see the [Nailgun website](#)

14.2.3 Testing & continuous integration

You have a Spoofax language which has a dynamic semantics in DynSem and you want to automate building and testing of the derived interpreter. This guide will take you through the necessary steps.

Enable JUnit test suite generation

DynSem can automatically generate a JUnit test harness which automatically runs all your tests. To enable generation you need to add a few extra properties in *dynsem.properties*:

```
1 # associated file extensions (comma-separated)
2 source.extensions = smpl
3 # (optional) Enable generation of a JUnit test suite
4 project.generatejunit = true
5 # (optional) Specify path to deposit the JUnit test suite into
6 project.testjavapath = src/test/java
7 # (optional) Specify the path to the test files
8 project.testspath = src/test/resources
```

Line 2 specifies the file extensions for the language. *SIMPL* only allows *smpl*. Your language can support multiple by giving a comma-separated list.

Line 4 enables automatic generation of the JUnit Test Suite. The package in which the test suite will be placed depends on the other properties in the configuration file. It is a concatenation of properties:

[project.groupid].[project.artifactid].generated.test. For *SIMPL* the fully qualified name of the generated JUnit Suite will be **org.metaborg.simpl.interpreter.generated.test**.

Line 6 specifies the directory in which the JUnit test suite will be placed. The directory structure corresponding to the package will automatically be created. If the property is not given the default is `project.testjavapath = src/test/java`.

Line 8 specifies the directory in which the test programs reside. If the property is not given the default is `project.testspath = src/test/resources`.

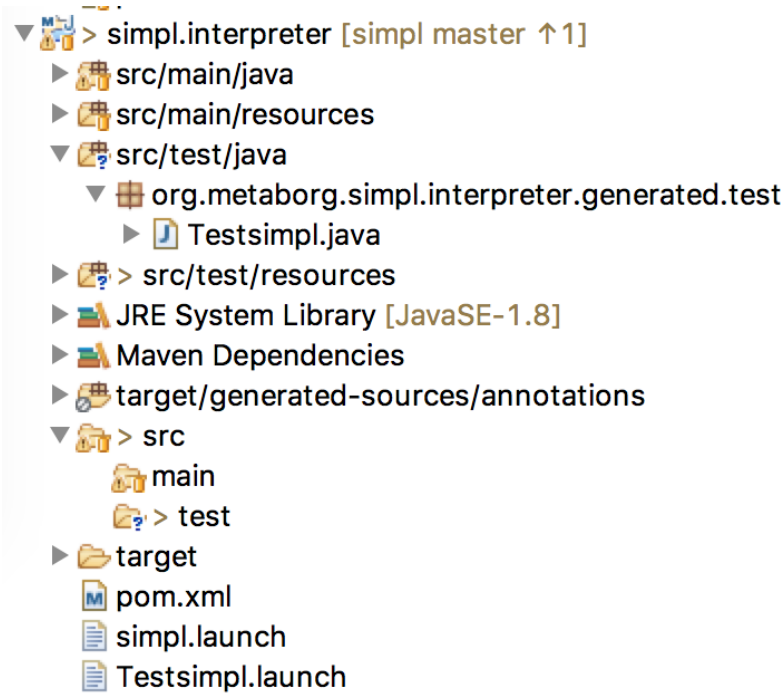
With the above additions the `dynsem.properties` file becomes:

```

1  # the name of the language. may not contain hyphens
2  source.langname = simpl
3  # version of this language
4  source.version = 0.1
5  # associated file extensions (comma-separated)
6  source.extensions = smpl
7
8  # start symbol to use for parsing programs in this language
9  source.startsymbol = Prog
10
11 # constructor name/arity of reduction entry point
12 source.initconstructor.name = Program
13 source.initconstructor.arity = 1
14
15 # path to interpreter project, absolute or relative to the language project
16 project.path = ../simpl.interpreter/
17
18 # (optional) enable/disable creation of the target project
19 project.create = true
20
21 # (optional) enable/disable cleaning of the target project before writing files
22 project.clean = true
23
24 # groupid & artifactid for the interpreter project
25 project.groupid = org.metaborg
26 project.artifactid = simpl.interpreter
27
28 # package name for manually implemented interpreter nodes
29 project.nativepackage = simpl.interpreter.natives
30
31 # (optional) Enable generation of a JUnit test suite
32 project.generatejunit = true
33 # (optional) Specify path to deposit the JUnit test suite into
34 project.testjavapath = src/test/java
35 # (optional) Specify the path to the test files
36 project.testspath = src/test/resources

```

We regenerate the interpreter project by invoking `.`. The *SIMPL* interpreter the project structure now is:



Note the generation of the **Testsimpl** Java class and the **Testsimpl.launch**. The latter is an Eclipse launch configuration for the test suite. We could already run the test suite but without any tests it would instantaneously succeed.

Create test programs

To create tests we create a 3-tuple for every program consisting of:

1. the program file
2. (optional) the input to the program
3. the expected output of the program

As an example consider the following program in *SIMPL*:

```
"hello world"
```

We save this program in a file named **helloworld.smpl** and place it in the tests directory. From the semantics of *SIMPL* expect this program to evaluate to `StringV(hello world)` so we create the expected output file:

```
StringV(hello world)
```

We name it **helloworld.output** (note the `.output` file extension) and save it next to the program file - **helloworld.smpl**.

We similarly add a test program which calculates the factorial of 7:

```
let
  recf = box(-1)
in {
  let
    f = box(
      n ->
        ifz(n)
```

(continues on next page)

(continued from previous page)

```

    then 1
    else {
      let
        newn = n - 1
      in
        (n * unbox(recf) (newn))
    }
  )
in {
  setbox(recf, unbox(f))
  ;unbox(f) (7)
}
}

```

And its expected output:

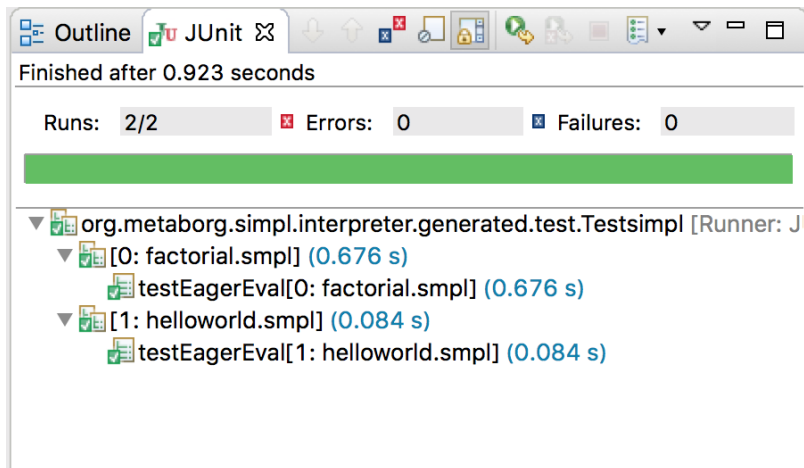
```
NumV(5040)
```

We save these files as **factorial.smpl** and **factorial.output**.

If your programs consumes user input you can create a `.input` file containing the input to be served to the program.

Run tests

We can now run the *SIMPL* tests. In Eclipse, select `JUnit` in the left hand side pane and select the *Testsimpl* launch configuration. Press `Run`. The JUnit view opens and we can observe the test results:



Tests can also be run from CLI using Maven. To do this, open a console and change into the interpreter project - **simpl.interpreter**. At the prompt run `mvn test`:

```

...
[INFO] --- maven-surefire-plugin:2.19.1:test (default-test) @ simpl.interpreter ---

-----
T E S T S
-----

Running org.metaborg.simpl.interpreter.generated.test.Testsimpl
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.278 sec - in org.
metaborg.simpl.interpreter.generated.test.Testsimpl

```

(continues on next page)

(continued from previous page)

```
Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

Build your language from CLI

We now have a language project and an interpreter project for *SIMPL*. We can run tests from CLI. But we cannot yet generate an interpreter from CLI. To achieve this we need to modify the language project build configuration.

We contribute the following goal to the **pom.xml** file of the *SIMPL* language:

```
1 <build>
2   <plugins>
3     <plugin>
4       <groupId>org.metaborg</groupId>
5       <artifactId>spoofax-maven-plugin</artifactId>
6       <version>${metaborg-version}</version>
7       <extensions>true</extensions>
8       <executions>
9         <execution>
10           <phase>verify</phase>
11           <goals>
12             <goal>transform</goal>
13           </goals>
14         </execution>
15       </executions>
16       <configuration>
17         <language>ds</language>
18         <goal>Generate interpreter</goal>
19         <fileSets>
20           <fileSet>
21             <directory>${basedir}/trans</directory>
22             <includes>
23               <include>simpl.ds</include>
24             </includes>
25           </fileSet>
26         </fileSets>
27       </configuration>
28     </plugin>
29   </plugins>
30 </build>
31 <dependencies>
32   <dependency>
33     <groupId>org.metaborg</groupId>
34     <artifactId>org.metaborg.meta.lang.esv</artifactId>
35     <type>spoofax-language</type>
36     <version>${metaborg-version}</version>
37   </dependency>
38   <dependency>
39     <groupId>org.metaborg</groupId>
```

(continues on next page)

(continued from previous page)

```

40     <artifactId>org.metaborg.meta.lang.template</artifactId>
41     <type>spoofox-language</type>
42     <version>${metaborg-version}</version>
43 </dependency>
44 <dependency>
45     <groupId>org.metaborg</groupId>
46     <artifactId>meta.lib.spoofax</artifactId>
47     <type>spoofox-language</type>
48     <version>${metaborg-version}</version>
49 </dependency>
50 <dependency>
51     <groupId>org.metaborg</groupId>
52     <artifactId>org.metaborg.meta.lib.analysis</artifactId>
53     <type>spoofox-language</type>
54     <version>${metaborg-version}</version>
55 </dependency>
56 <dependency>
57     <groupId>org.metaborg</groupId>
58     <artifactId>dynsem</artifactId>
59     <type>spoofox-language</type>
60     <version>${metaborg-version}</version>
61 </dependency>
62 </dependencies>

```

Line 23 is the only *SIMPL*-specific entry, it needs to point to the main DynSem file of the language. By convention this must always be **LANGNAME.ds**. This addition instructs Maven to run the *Generate Interpreter* transformation on **simpl.ds** during the *verify* phase of the build.

After this addition, issuing `mvn verify` in the *SIMPL* language project causes the interpreter project to be generated after the language is built. We can run now the *SIMPL* interpretation tests by issuing `mvn test` in the *SIMPL* interpreter project.

At this stage the language project can be built and the interpreter can be generated and tested, all from the command line using Maven.

Continuous integration with Travis CI

We now put wire everything together such that the *SIMPL* interpreter is built and tested on Travis CI for every pushed commit. The Travis build must take the following actions:

1. Checkout source from Git
2. Build the language project
3. Generate the interpreter project
4. Build the interpreter
5. Run the tests

Firstly, we create a Shell script (file name *travis-build.sh*) that orchestrates the build steps:

```

1  #!/bin/bash
2  set -ev
3  cd $TRAVIS_BUILD_DIR/simpl
4  mvn -Pstandalone install
5  cd $TRAVIS_BUILD_DIR/simpl.interpreter
6  mvn test

```

The `$TRAVIS_BUILD_DIR` variable is bound in the Travis CI build environment and points to the root of the Git repository. The script first builds, tests and installs the language project. During the *verify* phase the interpreter project will be generated. It then builds and tests the generated interpreter project. The `-Pstandalone` parameter instructs Maven to activate the *standalone* repository profile (yet to be created).

Secondly, we create a `.travis.yml` configuration file:

```

1 language: java
2 before_script:
3   - echo "MAVEN_OPTS='-server -Xms512m -Xmx1024m -Xss16m'" > ~/.mavenrc
4 script: ./travis-build.sh
5
6 cache:
7   directories:
8     - $HOME/.m2
9
10  jdk:
11    - oraclejdk8

```

There's nothing *SIMPL* specific here. We specify additional parameters (bigger heap, larger stack) for the building VM; instruct Travis that it should cache the Maven repository; and that the build should happen in an Oracle JDK 1.8. And of course we link the previously created build script.

Thirdly, if we were to try it out on Travis the build would be broken failing to download the Metaborg dependencies. We need to add an additional Maven repository profile (the *standalone* profile) to the language projects so that it can find the Metaborg dependencies during the build. We add the following to both `pom.xml` files:

```

1 <profiles>
2   <profile>
3     <id>standalone</id>
4     <repositories>
5       <repository>
6         <id>metaborg-release-repo</id>
7         <url>http://artifacts.metaborg.org/content/repositories/releases/</url>
8         <releases>
9           <enabled>true</enabled>
10        </releases>
11        <snapshots>
12          <enabled>>false</enabled>
13        </snapshots>
14      </repository>
15      <repository>
16        <id>metaborg-snapshot-repo</id>
17        <url>http://artifacts.metaborg.org/content/repositories/snapshots/</url>
18        <releases>
19          <enabled>>false</enabled>
20        </releases>
21        <snapshots>
22          <enabled>true</enabled>
23        </snapshots>
24      </repository>
25      <repository>
26        <id>spoofox-eclipse-repo</id>
27        <url>http://download.spoofox.org/update/nightly/</url>
28        <layout>p2</layout>
29        <releases>
30          <enabled>>false</enabled>
31        </releases>

```

(continues on next page)

(continued from previous page)

```

32     <snapshots>
33       <enabled>>false</enabled>
34     </snapshots>
35   </repository>
36 </repositories>
37 <pluginRepositories>
38   <pluginRepository>
39     <id>metaborg-release-repo</id>
40     <url>http://artifacts.metaborg.org/content/repositories/releases/</url>
41     <releases>
42       <enabled>>true</enabled>
43     </releases>
44     <snapshots>
45       <enabled>>false</enabled>
46     </snapshots>
47   </pluginRepository>
48   <pluginRepository>
49     <id>metaborg-snapshot-repo</id>
50     <url>http://artifacts.metaborg.org/content/repositories/snapshots/</url>
51     <releases>
52       <enabled>>false</enabled>
53     </releases>
54     <snapshots>
55       <enabled>true</enabled>
56     </snapshots>
57   </pluginRepository>
58 </pluginRepositories>
59 </profile>
60 </profiles>

```

This adds the Metaborg repositories (both releases and snapshots) to a Maven profile named *standalone*.

Fourthly, and finally we enable Travis CI builds for *SIMPL* using Travis’s dashboard. We can also include a buildstatus badge ().

If you have a tutorial or a usage scenario you would like to contribute, head over to the [DynSem issue tracker](#) and let us know.

14.3 Support

14.3.1 Getting help and reporting bugs

Thank you for using DynSem to specify your language’s dynamic semantics. If you need assistance using DynSem or you have discovered a bug please report it on [DynSem’s issue tracker](#). Reporting your issue will help us remedy the situation and get you going again quickly.

14.3.2 Feature requests

DynSem’s features and design are still evolving. Please write to us if you want a feature in DynSem, in DynSem’s tooling (IDE, generated interpreter) or you would like a particular topic to be covered in the documentation. The best place to address feature requests is in [DynSem’s issue tracker](#).

14.3.3 Contributing

All of DynSem is open source and licensed under Apache License 2.0 and lives in the [GitHub repository](#). If you would like to contribute to DynSem please fork the repository and follow the customary procedure to submit your contributions as pull requests.

The Editor SerVice (ESV) language is a declarative meta-language for configuring the editor services of a language. For example, the following ESV code fragment configures the syntax highlighting for a language, based on the types of tokens:

```
module color

colorer

  keyword      : 153 51 153
  identifier   : black
  string       : 177 47 2
  number       : 17 131 22
  operator     : black
  layout       : 63 127 95 italic
```

We now look at how an ESV file is structured.

15.1 Structure

Each ESV file starts by defining a module for the file. Module identifiers must be unique. Modules can be imported with an import statement. For example, the following code fragment defines an ESV file with module `Main` and imports module `Syntax` and `Analysis`:

```
module Main

imports

  Syntax
  Analysis
```

Importing modules means importing all the sections and definitions from that module.

The rest of the ESV file can contain sections for configuring editor services and other aspects of a language. Sections can be defined multiple times. For example, the following code fragment defines an ESV file with several sections:

```
module Main

language

  extensions : ent

colorer

  keyword      : 153 51 153
  identifier   : default

language

  line comment : "//"
```

It sets the extension in a `language` section, adds coloring rules in a `colorer` section, and sets the line comment to use in a `language` section again.

15.2 Configuration Sections

Since editor services are a cross-cutting concern, ESV has a lot of cross-cutting configuration for these concerns. We now describe each configuration section in detail.

15.2.1 File Extension

The file extension of your language are set with the `extensions` option under the `language` section:

```
language

  extensions : ent
```

Multiple extensions can be set with a comma-separated list:

```
language

  extensions : ent, entity, entities
```

This registers the given extensions with Spoofox, which can then identify files to your language based on extensions. This is used to open an editor for your language in an IDE setting, and to process files of your language in a command-line setting.

15.2.2 Syntax

The SDF parse table file, and which start symbols to use, are set as follows:

```
language

  table           : target/metaborg/sdf.tbl
  start symbols   : Start
```

Multiple start symbols can be set with a comma-separated list:

```
language
    start symbols : Start, Program
```

The parse table of your language is set with the `table` option. By default, the parse table of an SDF specification is always produced at `target/metaborg/sdf.tbl`. It is only necessary to change this configuration when a custom parse table is used. The `start symbols` option determine which start symbols to use when an editor is opened. This must be a subset of the start symbols defined in the SDF3 specification of your language.

The syntax for comments are set as follows:

```
language
    line comment : "//"
    block comment : "/*" "*/"
```

The `line comment` option determines how single-line comments are created. It is used by editors to toggle the comment for a single line. For example, in Eclipse, pressing `Ctrl-/` (`Cmd-/` on macOS), comments or uncomments the line. The `block comment` option determines how multi-line comments are created. It is used similarly, but when a whole block needs to be commented or uncommented. A block comment is determined by two strings denoting the start and end symbols of the block comment.

Fences for bracket matching are set as follows:

```
language
    fences : [ ] ( ) { }
```

The `fences` options determines which symbols to use and match for bracket matching. A single fence is defined by a starting and closing symbol. Multiple fences can be set with a space-separated list. Fences are used to do bracket matching in text editors.

Warning: Fences can contain multiple characters, but some implementations may not handle bracket matching with multiple fence characters. For example, Eclipse does not handle this case and ignores multi-character fences.

15.2.3 Syntax Highlighting

Token-based syntax highlighting is configured in a `colorer` section. Such a section can contain *style definitions* and *styling rules*.

Style definitions bind an identifier to a *style* with `syntax identifier = style` for reuse later. A style is a foreground (text) color, optional background color, and optional font attributes. For example, the following style definitions bind the `red`, `green`, and `blue` colors:

```
colorer
    red   = 255 0 0
    green = 0 255 0
    blue  = 0 0 255
```

A color is denoted by its RGB values, with values ranging from 0 to 255. An optional background color can be set by adding another RGB value:

```
colorer

redWithGreenBackground = 255 0 0 0 255 0
```

Optional font attributes can be used to make the font bold or italic:

```
colorer

redWithBold    = 255 0 0 bold
redWithItalic  = 255 0 0 italic
redWithGreenBackgroundWithBoldItalic = 255 0 0 0 255 0 bold italic
```

Style rules assign styles to matched tokens with `syntax matcher : styleOrRef`. The left hand side of style rules match a token, whereas the right hand side assigns a style by referring to a previously defined style definition, or by directly assigning a style. For example, the following matches a token type and references a style definition:

```
colorer

operator : black
```

whereas the following matches a token with a sort and constructor, and directly assigns a style:

```
colorer

ClassBodyDec.MethodDec : 0 255 0
```

The following matchers on the left-hand side are supported:

- Matching on built-in token types. The following types are supported:
 - `identifier` - matches identifiers, found by lexical non-terminals without numbers
 - `string` - matches strings, found by lexical non-terminals that include quotation marks
 - `number` - matches numbers, found by lexical non-terminals with numbers
 - `keyword` - matches keywords, found by terminals in the syntax definition
 - `operator` - matches operations, found by terminals that contain just symbols (no characters)
 - `layout` - matches layout, such as whitespace and comments, found by layout definition
 - `unknown` - matches tokens which the parser was unable to infer a type for

For example, the following code defines a simple highlighting with token types:

```
colorer

keyword      : 153 51 153
identifier   : black
string       : 177 47 2
number       : 17 131 22
operator     : black
layout       : 63 127 95 italic
```

- Matching on sorts of tokens. For example:

```
colorer

ID          : darkblue
```

(continues on next page)

(continued from previous page)

```
TYPEID    : blue
JQTYPEID  : blue
PQTYPEID  : blue
FUNCID    : 153 51 0
JFUNCID   : 153 51 0
STRING    : 177 47 2
```

- Matching on sorts of tokens, and the constructor of the term that was created using the token. This uses the `Sort.Constructor` syntax. For example:

```
colorer

  ClassBodyDec.MethodDec : yellow
  ClassBodyDec.FieldDec  : red
```

- Matching on the constructor only. This uses the `_.Constructor` syntax. For example:

```
colorer

  _.Str      : blue
  _.StrCong  : blue
  _.QStr     : blue
  _.QDollar  : blue
  _.QBr      : gray
```

15.2.4 Menus

Menus are used to bind actions of your language, such as transformations, to a menu in the IDE. Menus are defined under a `menus` section:

```
menus

  menu: "Generate"
```

This adds a submenu titled `Generate` to the menu of your language. Submenus can be nested under `menus`, and submenus can be nested as well:

```
menus

  menu: "Generate"
    submenu: "To Java"
      submenu: "Abstract"
    end
    submenu: "Concrete"
  end
end
```

Actions (sometimes called builders) are defined under a menu or submenu with syntax `action: "Name" = strategy modifiers:`

```
menus

  menu: "Generate"
    action: "To normal form" = to-normal-form (source)
```

(continues on next page)

(continued from previous page)

```
submenu: "To Java"
  action: "Abstract" = to-java-abstract (openeditor)
  action: "Concrete" = to-java-concrete
end
```

An action has a name which is displayed in the menu, an identifier to a Stratego strategy, and optional modifiers. The following modifiers are supported:

- (source) - indicates that the action is performed on the parsed AST, not the analyzed AST
- (openeditor) - indicates that the result of the action should be shown in a new text editor

The Stratego strategy that an action refers to has a defined signature. It must take as input a 5-tuple (`_`, `_`, `ast`, `path`, `projectPath`), and must produce either `None()` or `(filename, output)` when the action produces a file. Multiple files can be produced by returning a tuple `(filename*, output*)` of two equal length lists, one with the filenames and the other with the corresponding content. The 5-tuple has wildcards which are not used by Spoofax any more, but are kept in the signature for compatibility reasons. The following Stratego code is an example of a strategy that implements this signature:

```
j2m-action:
  (_, _, ast, path, projectPath) -> (outputFile, result)
  with
    outputFile := $[[projectPath]/[<remove-extension> path].mod]
  ; result      := <j2m-main> ast
```

The following Stratego code is an example of a menu action that returns multiple files:

```
gen-str:
  (_, _, ast, path, projectPath) -> result
  with
    sig-file    := $[[projectPath]/[<remove-extension> path]-sig.str]
  ; sig-str     := <ast-to-sig> ast
  ; rules-file  := $[[projectPath]/[<remove-extension> path]-rules.str]
  ; rules-str   := <ast-to-rules> ast
  ; result := <unzip> [
    (sig-file, sig-str),
    (rules-file, rules-str)
  ]
```

Modifiers can also be used on menus and submenus, which mean that all nested actions inherit those modifiers. For example, in:

```
menus

menu: "Generate" (source) (openeditor)
  action: "To normal form" = to-normal-form
  submenu: "To Java"
    action: "Abstract" = to-java-abstract
    action: "Concrete" = to-java-concrete
  end
```

all actions inherit the (source) and (openeditor) modifiers on the menu.

15.2.5 Outline

An outline is a summary of a file that is shown in a separate view next to a textual editor. An outline is created by a Stratego strategy, but is configured in ESV under the `views` section:

```
views

  outline view: editor-outline
    expand to level: 3
```

This configures the `editor-outline` Stratego strategy to be used to create outlines, and that outline nodes should be expanded 3 levels deep by default.

Todo: Describe input and output signature of the outline strategy.

15.2.6 Hover Tooltips

Hover tooltips show a textual tooltip with extra information, when hovering part of the text. Hover tooltips are created by a Stratego strategy, but are configured in ESV under the `references` section:

```
references

  hover _ : editor-hover
```

The identifier after the colon refers to the Stratego strategy that creates the hover tooltip. The Stratego strategy takes an AST node, and either fails if no tooltip should be produced, or returns a tooltip string.

The string may contain a few simple HTML tag to style the output. The following tags are supported:

- `
` - line break
- `text` - bold
- `<i>text</i>` - italics
- `<pre>code</pre>` - preformatted (code) text

15.2.7 Compiler

The compiler strategy (frequently called the on-save handler) is used to transform files when they are saved in an editor. In an IDE setting, when a new project is opened, the compiler strategy is also executed on each file in the project, as well as when files change in the background. In a command-line batch compiler setting, it is used to transform all files.

The compiler strategy is configured in ESV with the `on save` option:

```
language

  on save : compile-file
```

The identifier after the colon refers to the Stratego strategy that performs the transformation. This strategy must have the *exact same signature as the one for actions*.

15.2.8 Analyzer and Context

The analyzer strategy is used to perform static analyses such as name and type analysis, on the AST that a parser produces. An analysis context provides a project-wide store to facilitate multi-file analysis and incrementality. There are four ways to configure the analysis, which set the analyzer strategy with option `observer` and context with option `context`.

- No analysis. This disables analysis completely. Do not set an `observer` and set the `context` to `none`:

```
language
  context : none
```

- Stratego-based analysis. This allows you to implement your analysis in Stratego:

```
language
  context : legacy
  observer : editor-analyze
```

The identifier after the colon refers to the Stratego strategy that performs the analysis. It must take as input a 3-tuple (`ast`, `path`, `projectPath`). As output it must produce a 4-tuple (`ast`, `error*`, `warning*`, `note*`). The following Stratego code is an example of a strategy that implements this signature:

```
editor-analyze:
  (ast, path, projectPath) -> (ast', errors, warnings, notes)
  with
    ast'      := <analyze> ast
  ; errors    := <collect-all(check-error)> ast'
  ; warnings  := <collect-all(check-warning)> ast'
  ; notes     := <collect-all(check-note)> ast'
```

- NaBL/TS based analysis. This uses the *NaBL and TS* meta-languages for name and type analysis. Your project must have been generated with NaBL+TS as the analyzer. It will produce the following ESV configuration:

```
language
  context : taskengine
  observer : editor-analyze (multifile)
```

- NaBL2 based analysis. This uses the *NaBL2* meta-language for name and type analysis. Your project must have been generated with NaBL2 as the analyzer. It will produce the following ESV configuration:

```
language
  observer : editor-analyze (constraint)
```

By default, the NaBL2 analyzer works in single-file mode and does not consider multi-file name resolution. To enable that, add the `(multifile)` modifier:

```
language
  observer : editor-analyze (constraint) (multifile)
```

15.2.9 Reference Resolution

Reference resolution takes an AST node containing a reference, and tries to resolve it to its definition. The resolution is performed by a Stratego strategy, but is configured in ESV under the `references` section:

```
references

  reference _ : editor-resolve
```

The identifier after the colon refers to the Stratego strategy that performs the resolution. The Stratego strategy takes an AST node, and either fails if it could not be resolved, or returns an AST node that has an origin location pointing to the definition site.

If you use the *NaBL and TS* or *NaBL2* meta-language to implement name and type analysis, the provided `editor-resolve` strategy implements resolution generically.

15.2.10 Stratego

The JAR and CTree files that will be loaded into the Stratego runtime for your language can be configured with the `provider` option:

```
language

  provider : target/metaborg/stratego.ctree
```

The extension of the provider should match the format in the `metaborg.yaml` file of your language.

Multiple files can be set by setting the option multiple times:

```
language

  provider : target/metaborg/stratego.ctree
  provider : target/custom1.jar
  provider : target/custom2.ctree
```

15.3 Main File

ESV currently does not have a configurable main file. The main ESV file of your language must be located at `editor/Main.esv` or `editor/main.esv`. Every ESV file that is (transitively) imported from the main ESV file is used.

Language Testing with SPT

The SPOofax Testing language (SPT) allows language developers to test their language in a declarative way. It offers a language to express test cases for any textual language that you want to test, and a framework for executing those tests on language implementations created with Spoofax.

We will first describe the syntax and semantics of the SPT language. Then, we will discuss how you can execute your SPT test cases, and finally we conclude with an overview of the architecture of the SPT framework.

In this section we will describe the syntax and semantics of SPT.

If you want to write your own tests you can follow along as the different concepts are explained. We suggest using the Spoofax Eclipse plugins, as they contain an editor for SPT files. In an Eclipse with Spoofax installed, simply create a new file with the extension `.spt` and follow along to create your first SPT test suite.

16.1 Test suites

Test cases in SPT are grouped in test suites (or modules). Each SPT file contains exactly 1 test suite.

Test suites allow you to group together the test cases that test similar aspects of your language, making it easier to organize your tests. They also allow you to reduce the size of your test cases by using configuration options like *headers* and *test fixtures* that apply to all test cases in the test suite. We will describe those later on.

The syntax for a test suite is as follows:

```
TestSuite = <
  module <MODULENAME>
    <Header?>
    <TestFixture?>
    <TestCase*>
>
```

The modulename must start with a letter and can be followed by any number of letters, digits, underscores, and dashes.

If you are following along, you can create your first SPT test suite. Just create a file with the extension `.spt` and paste the following:

```
module my-first-test-suite

language MyLanguageName
```

Be sure to replace `MyLanguageName` with the name of the language you want to test. Also, feel free to use a more descriptive module name if you like.

16.1.1 Headers

Headers are a configuration option for all test cases in a test suite. The most important header is the language header. If you are following along with the example, you have already used it to specify which language you wanted to test. All test cases in the test suite will be ran against the language you specify with the language header. This language will be called the *language under test* (LUT):

```
Header.Language = <language <LANGNAME>>
```

For now, there is only one other header: the start symbol header. This optional header allows you to specify a nonterminal from your grammar from which the test cases will be parsed. Don't worry if that doesn't make any sense yet. We will look at an example later:

```
Header.StartSymbol = <start symbol <ID>>
```

Where the `ID` must start with a letter, and can be followed by any number of letters or digits.

As our example test suite already contains the language header, we will move on to writing our first test.

16.1.2 Test Cases

Test cases are the most important parts of SPT. Each test case is a behavioral test, or black-box test, for your language. A behavioral test consists of a component that should be tested, an initial state for that component, input for that component, and the expected output after running the component on the input.

First, let's look at the input of a test case. As we are testing languages, the input of a test case is always a program written in the language under test. Such a program written in the language under test is called a *fragment*, and it is embedded in the SPT test suite. In the future, we want to offer all editor services of your language under test (e.g., syntax highlighting) while you are writing such a fragment. However, for now this is not yet supported.

The component that should be tested and the expected output are captured in *test expectations*. We will discuss those in a later section.

Finally, the initial state of the test case can be specified in a *test fixture*, analogous to the JUnit `setUp` and `tearDown` methods. These fixtures will also be discussed in a later section.

The syntax of test case is as follows:

```
TestCase = <
  test <Description> <OpenMarker>
    <Fragment>
  <CloseMarker>
  <Expectation*>
>
```

Where `Description` is the name of the test and can contain any character that is not a newline or `[`. The `OpenMarker` marks the start of the fragment and can be one of `[`, `[[`, or `[[[`. The `CloseMarker` marks the end of the fragment and should be the counter part of the `OpenMarker`, so either `]`, `]]`, or `]]]`.

The `Fragment` is a piece of text written in the language under test, about which we want to reason in our test case. It may contain any sequence of characters that are not open- or closing markers. For example, if the `[[[` and `]]]` markers are used, the fragment may contain at most 2 consecutive open or closing square brackets. If the `[[[[` and `]]]]` markers are used, the fragment may contain at most 3 consecutive open or closing square brackets.

Parts of the fragment may be selected, by surrounding the text with the same open- and closing markers that were used to delimit the fragment. These *selections* can later be referred to in the test expectations to allow more precise reasoning about the fragment.

We will discuss selections and expectations later, but note that not supplying any test expectation is equivalent to supplying only a `parse succeeds` expectation, indicating that the fragment is expected to be valid syntax and parsing it with the language under test is expected to succeed without errors.

If you are following along with your own test suite, let's create our first test case. I will be using a simplified version of Java (called `MiniJava`) as my language under test, but it shouldn't be too hard to write a simple test for your own language. We will be writing a `MiniJava` program with valid syntax and test that it does indeed parse. The input for our test case will simply be a main class in `MiniJava`. The component that we will be testing is the parser, and the expected output is a successful parse result:

```
module simple-parse-tests

language MiniJava

test check if this simple program parses successfully [[
  class Main {
    public static void main(String[] args) {
      System.out.println(42);
    }
  }
]] parse succeeds
```

Change the language header to refer to the name of your language and change the fragment to be a valid specification in your language, and you should have your first working test case. Try messing up the syntax of your fragment and an error message should be displayed to indicate that the test failed, as the fragment failed to be parsed correctly. These error messages can be displayed directly inside the fragment you wrote, to make it easier for you to spot why the test failed. This is the power of SPT fragments!

Now that we know the basic structure of a test, we can already see how the start symbol header can be used to decrease the size of our test:

```
module statement-parse-tests

language MiniJava
start symbol Statement

test check if a println is a valid statement [[
  System.out.println(42);
]]
```

Note how the fragment is now no longer a proper `MiniJava` program. The test still passes, as the fragment is now parsed starting from the `Statement` nonterminal. Note that this only works if `Statement` is exported as a start symbol in the `MiniJava` language. These start symbols are a way of indicating what the initial state of the component under test should be. In this case, it influences the state of the parser and only allows it to successfully parse statements.

Before we move on to discuss the set of all supported test expectations, we will first look at another way to influence the initial state and reduce the size of our test cases: test fixtures.

16.1.3 Test Fixtures

A test fixture offers a template for all test cases in the test suite. Using test fixtures, you can factor out common boilerplate from your tests and write it only once.

The syntax for a test fixture is as follows:

```
TestFixture = <
  fixture <OpenMarker>
    <StringPart>
    <OpenMarker> ... <CloseMarker>
    <StringPart>
    <CloseMarker>
>
```

Where the `OpenMarker` is one of `[`, `[[`, or `[[[`, and the `CloseMarker` is one of `]`, `]`, or `]`. The `StringPart` can contain any sequence of characters that is not an open- or closing marker, just like a fragment from a test. However, unlike a fragment of a test, it can not contain selections.

For each test case, the fragment of the test will be inserted into the fixture at the marked location (`<OpenMarker> ... <CloseMarker>`), before the test expectations will be evaluated.

We can now use a test fixture to test the syntax of statements in MiniJava without the use of the start symbol header:

```
module statements-parse-test

language MiniJava

fixture [[
  class Main {
    public static void main(String[] args) {
      [...]]
  }
]]

test check if printing an integer is allowed [[
  System.out.println(42);
]]

test check if printing a String is allowed [[
  System.out.println("42");
]]
```

Note that test fixtures offer a fully language implementation agnostic way of factoring out boiler plate code, whereas the start symbol header requires knowledge of the non terminals of the language implementation.

16.2 Test Expectations

Test expectations allow you to specify which component of the language under test should be tested, and what the expected output of the test will be.

We have already seen the `parse succeeds` expectation in action, and briefly mentioned that a test case without any test expectations is the same as a test case with a `parse succeeds` expectation. We will now list the syntax and semantics of all the currently supported test expectations:

16.2.1 Parse Expectations

Parse expectations can be used to test the syntax of your language. They indicate that the input fragment of the test case should be parsed and allow you to specify what you expect the result to be. As parsing is the preliminary step to all other language components (e.g., analysis and transformations) they are treated differently from other expectations. If no parse expectation is present on a test case, even if another expectation (e.g. an analysis expectation) is present, a `parse succeeds` expectation will be added to the test case.

Expectation.ParseSucceeds = `<parse succeeds>` Parse the fragment and expect the parsing to succeed, with no parse errors and no ambiguities.

Expectation.ParseFails = `<parse fails>` Parse the fragment and expect the parsing to fail, with at least one parse error.

Expectation.ParseAmbiguous = `<parse ambiguous>` Parse the fragment and expect the parsing to succeed with one or more ambiguities.

Expectation.ParseToATerm = `<parse to <ATerm>>` Parse the fragment, expect parsing to succeed, and compare it to the given ATerm AST. When using test fixtures, the ATerm should only be the AST of the *fragment* of the test, not of the entire test fixture. Please note that if you want to specify a List in the ATerm, the square brackets of the list may interfere with the markers of a fragment. Therefore, to specify a list as the expected output, prepend it with the keyword `!ATerm`. For example:

```
parse to !ATerm ["5"]
```

Expectation.ParseToFragment = `<parse to <Language?> <OpenMarker> <Fragment> <CloseMarker>]`

Parse the fragment, expect parsing to succeed, and compare it to the result of parsing the given `Fragment` with the given `Language`. When the `Language` is omitted the language under test will be used to parse the given fragment. When using test fixtures, only the test's input fragment will be combined with the test fixture. The fragment in this expectation (i.e., the output fragment) will **not** be combined with it, even if the language under test is used to parse it. To counteract this, the entire AST (including the nodes from the fixture) will be compared to the expectation's fragment's AST.

This expectation can be useful to test disambiguations, or to test your language against a reference implementation. An example test case for disambiguation in MiniJava would be:

```
module disambiguation
language MiniJava
start symbol Exp

test plus is left associative [[
    1 + 2 + 3
]] parse to [[
    (1 + 2) + 3
]]
```

16.2.2 Analysis Expectations

Analysis expectations specify that the analyzer should be tested. We will first discuss the most generic analysis expectations: the message expectations. These expectations can be used to test the entire static semantic analysis process. Then we will look at test expectations that are more specific: the name analysis expectations.

16.2.3 Analysis Message Expectations

Analysis message expectations will cause the input fragment to be parsed and analyzed (e.g., name and type analysis and static error checking). Finally, the resulting messages (i.e. errors, warnings, or notes) will be compared to the

expectation. Note that messages of the expected type are **not** allowed to appear in the test *fixture*, if one is present. This is to prevent test from succeeding, when a message that would make it fail appears in an unexpected location. Only the messages within the test's *fragment* will be compared to the expectation.

Expectation.MessageExpectation = <<Operator?> <INT> <Severity>> These expectations will check if the number of messages of the given severity generated by the analysis matches the given number. Any other messages (of different severity or located in the test fixture) will be ignored by this expectation.

The optional operator can be used to loosen the strictness of the check. For example the expectation `> 2 errors` allows any number of errors bigger than 2. If no operator is specified, it will default to `=`. The allowed operators are `=`, `>`, `>=`, `<`, `<=`.

The allowed message severities are: `error` or `errors` for error messages, `warning` or `warnings` for warning messages, and `note` or `notes` for note messages. Please note that error messages in this expectation refer only to analysis errors (not parse errors). Also when you are testing for warnings, for example, any analysis messages of different severity (including errors) will not cause the test to fail. For example, the following test will succeed, regardless of the blatant type error, because we only test for warnings:

```
module test-for-warnings
language MiniJava

fixture [[
  class Main {
    public static void main(String[] args) {
      System.out.println([...]);
    }
  }
]]

test no warnings [[1 + new A()]] 0 warnings
```

These analysis message expectations may be followed by the `at` keyword and a list of one or more comma separated references to selections of the test's fragment: `#<INT>`, `#<INT>`, `#<INT>`, ... As this is the first time we encounter references to selections, let's look at an example:

```
module error-locations
language MiniJava

test duplicate classes [[
  class Main {
    public static void main(String[] args) {
      System.out.println(42);
    }
  }
  class [[A]]{}
  class [[A]]{}
]] 2 errors at #1, #2
```

This test will cause SPT to check if the specified messages appeared at the location of the given selection references. The selections are the classnames `A` that are selected by wrapping them in an open and close marker. Selections are referenced by the order in which they appear, starting at 1, from left to right and top to bottom.

It is allowed to give less selection references than the number of expected messages. In this case SPT assumes you don't care about the location of the other messages. If the same selection is referenced more than once, multiple messages will be expected at that location. For example `3 errors at #1` expects 3 errors, 2 of which should be at the location of selection number 1. The other error may be anywhere within the test fragment.

Expectation.MessageContent = <<Severity> like <STRING>> This expectation specifies that there should be at least 1 message of the given severity that contains the given String. For example `error`

like "duplicate class name" expects there to be at least 1 error in the fragment whose message contains duplicate class name.

This expectation can also be followed by the `at` keyword, and a single selection reference, to indicate where you expect the message with the given content.

Expectation.AnalysisSucceeds = `<analysis succeeds>` This expectation is syntactic sugar for 0 errors.

Expectation.AnalysisFails = `<analysis fails>` This expectation is syntactic sugar for > 0 errors.

16.2.4 Name Analysis Expectations

Name analysis expectations will check if use sites can be resolved and, if required, if they resolve to the correct definition. The fragment will be parsed and analyzed, but any number and severity of analysis messages are allowed.

Expectation.Resolve = `<resolve #<INT>>` Try to resolve the AST node at the given selection. Expect it to successfully resolve to any definition site.

Expectation.ResolveTo = `<resolve #<INT> to #<INT>>` Try to resolve the AST node at the first given selection. Expect it to successfully resolve to the location marked by the second given selection.

Note that selections can only occur in the test's *fragment*, not in the test *fixture*. So name analysis can only be tested within a test's fragment.

16.2.5 Transformation Expectations

A transformation transforms an AST to another AST. The idea within Spoofox is that a transformation has a name, and can be nested within a structure of menu's. Furthermore, it can have additional information about whether it transforms the raw AST (i.e. the parse result) or the analyzed AST (i.e. the result of desugaring and analysis). In languages created with Spoofox, transformations are Stratego strategies that are registered in the `Menus.esv` file.

Transformation expectations will first look up a given transformation using the name under which it was registered. Note that, for Spoofox languages, this is *not* necessarily the name of the Stratego strategy, but the name under which it is registered in the `Menus.esv` file. If this name is not unique, the menu structure can be used to look up the proper transformation.

Once the transformation is found, SPT will determine if it requires the raw AST, or the analyzed AST. If the raw AST is required, it will only parse the fragment. If the analyzed AST is required, it will also analyze the parse result. However, analysis is allowed to produce any number and severity of messages. Then, SPT will run the transformation on the entire AST, **including** the nodes from the test fixture, if there was one.

Expectation.Transform = `<transform <STRING>>` The `STRING` should be delimited by double quotes and contain the name of the transformation. If the name is not unique, the menu structure can be included as well, separated by `->`. For example: `transform "Menu name -> transformation name" to Some(Result())`. As long as the transformation returns a result, this expectation passes.

Expectation.TransformToAterm = `<transform <STRING> to <ATerm>>` Same as `Transform`, but the result of the transformation is compared to the given AST.

Expectation.TransformToFragment = `<transform <STRING> to <Language?> <OpenMarker> <Fragment>`
Does the same as `TransformToAterm`, but compares the result of the transformation to the AST of the given fragment. If the applied transformation required the raw AST, the given fragment will only be parsed with the given language. If no language is given, the language under test will be used. If the applied transformation required an analyzed AST, the given fragment will be parsed and analyzed.

16.2.6 Run Stratego Expectations

These test expectations are really only applicable to languages that use Stratego strategies in their implementation. They will parse and analyze the fragment and run a given Stratego strategy (with no arguments) and compare its output to the expectation.

Expectation.Run = `<run <STRATEGY>>` This expectation will lookup the given strategy name and run it on the AST node in the test's fragment. If the fragment contains multiple nodes (e.g., it's a list of Statements but some Statements were in the test fixture) the strategy will be run on each of these nodes. Either until it completes successfully, or until it failed on all these nodes. Note that it wil **not** be executed on the nodes in the test fixture, if there was one.

Expectation.RunWithArgs = `<run <STRATEGY>(|<TermArgs>)>` This expectation will run a strategy that expects term arguments. String literals, integer literals and selection references are permitted as term arguments.:

```
test rename variable without type [[
  let
    var msg := "Hello World"
  in
    print ([[msg]])
  end
]] run rename(|#1, "message", 0) to [[
  let
    var message := "Hello World"
  in
    print (message)
  end
]]
```

Expectation.RunFails = `<run <STRATEGY> fails>` This expectation checks if the given strategy fails.

Expectation.RunOn = `<run <STRATEGY> on #<INT>>` This expectation does the same as Run, except it runs the strategy on the nodes at the given selection instead of the nodes of the test's fragment.

Expectation.RunToAterm = `<run <STRATEGY> to <ATerm>>`

Expectation.RunToAtermOn = `<run <STRATEGY> on #<INT> to <ATerm>>` These expectations are similar to the first two, but they require the result of running the strategy to match the given AST.

Expectation.RunToFragment = `<run <STRATEGY> to <Language?> <OpenMarker> <Fragment> <CloseM`

Expectation.RunToFragmentOn = `<run <STRATEGY> on #<INT> to <Language?> <OpenMarker> <Fragment> <CloseMarker>>` These expectations are similar to the first two, but they require the result of running the strategy to match the result of analyzing the given fragment with the given language. If no language is given, the language under test is used.

16.2.7 Origin Location Expectations

Expectation.HasOrigins = `<has origin locations>` This expectation parses and analyzes the fragment. It then checks if all AST nodes in the test's fragment (except for Lists in Spoofox) have a source region (an origin) associated with them. It does **not** check the AST nodes in the test fixture.

When using Spoofox, there are some strategies that will break the origin information when used. This can lead to desugarings that create AST nodes without origin information, which can cause problems when trying to create messages at their location and with other services. This expectation can be used to check that your analysis is origin preserving.

16.3 Running SPT Tests

SPT tests can be run different ways. Each one corresponds to a different use case of SPT. In the end, they all use the common core SPT Java API for extracting and executing tests.

We will now briefly discuss the different ways to run an SPT test suite based on the use case. If you are new to SPT, the Interactive Test Design use case will be the one for you.

16.3.1 Interactive Test Design

SPT was originally designed for this use case. Its goal was to lower the threshold of writing tests for your language, by allowing you to concisely declare test inputs and outputs, offering editor services for the fragments that you write, and providing you with real time in-editor feedback on whether your test fails or passes.

For this use case you would be using Eclipse with the Spoofax plugin. When you open a test suite in the Eclipse editor, all failing test cases will have error markers on them. By turning the test results into message markers inside the editor, we can provide you with a detailed location on where it went wrong. Especially for parsing or analysis errors. However, to keep the error message readable, they can not contain full stack traces, which you might need to debug transformation or Stratego run tests. It is also impractical to check all your test suites this way if you have many of them in your project.

To solve this, we have created a JUnit style test runner in Eclipse. It is available through the Spoofax (meta) menu bar entry, and offers two ways to run tests.

The first one is called `Run tests of SPT files`. If you click this, it will check if you currently have an SPT file that is open and, if so, launch the test runner to run all tests in this file. This mode can be useful if one of your tests is failing and you would like to see a more detailed error message.

The second entry is called `Run all selected tests`. It will check what you selected in the package or project explorer. If you selected any SPT files, directories, or projects, it will scan **all** of the selected locations and run all of the SPT files it found within those selections. This method is useful for running regression tests.

The user interface of the test runner consists of 3 parts. The first part is the progress bar, which is followed by two numbers that indicate the progress of your current test runs. This part is displayed at the top of the runner. The second part is the overview of all the test suites and their test cases which are part of this test run. This part is displayed in the bottom left. The final part is a console window, which contains more detailed error messages about the test suite or test case you selected in the second part. This part is displayed in the bottom right:

```
!["SPT TestRunner Layout"] (images/SPTTestRunner.png)
```

The test runner will start displaying any test suites and test cases within those test suites as soon as it discovers them. Then, after they are all loaded, they will be executed one by one, and the progress bar at the top will increase. As long as the progress bar remains green, no tests failed yet. As soon as a single test fails, the progress bar will turn red to indicate so.

The numbers next to the progress bar also indicate the progress. For example, `5 / 7` means 5 tests passed already out of a total of 7 tests. This can mean that either 2 tests failed, or some tests have not been executed yet. Which case applies can be determined by looking at the progress bar.

Any SPT files that fail can't be parsed or from which we can't extract test cases for some other reason, will be included in the list on the bottom left side, along with the test suites that did manage to get extracted. The ones that did not extract properly will be displayed in red, as opposed to the default black color for test suites. By selecting a red test suite, the extraction errors will be displayed in the console on the bottom right. Any test suite can be double clicked to open the corresponding file in Eclipse. Test suites that got extracted successfully can be expanded if they contained any test cases. This will show all the test cases of that suite as child elements of the test suite in the bottom left view.

Test cases are displayed in the default black color if they have not been executed yet. Test cases that have finished will have their duration appended to their name. Failed test cases are displayed in red, and passing test cases are displayed in green. A red test case can be selected, doing so will show the messages about the test failure, including the exceptions that caused them (e.g. a `StrategoRuntimeException` with a stacktrace) in the console on the bottom right. Double clicking a test case will open the corresponding SPT file and jump to the location of the test case.

When a test case fails, the test suite that contained the failing test case will be appended with the number of failed tests in that test suite so far.

16.3.2 Run using the Command Line Runner

At <https://github.com/metaborg/spt/tree/master/org.metaborg.spt.cmd> there is a project that creates an executable jar with which you can run all the test suites in a given directory. It is more of a proof of concept and an example of how to use our core SPT Java API than a full fledged test runner.

For those interested in giving it a try:

1. Obtaining the test runner jar:

```
bash
$ git clone https://github.com/metaborg/spt.git
$ cd spt/org.metaborg.spt.cmd
$ mvn package
$ ls target/org.metaborg.spt.cmd*
target/org.metaborg.spt.cmd-2.0.0-SNAPSHOT.jar
```

This jar is the executable jar that contains the test runner. Next up, we want to run the tests for our language. To do so, we need:

1. the directory with tests to run (e.g., `path/to/test/project`)
 2. the language under test (e.g. `path/to/MiniJava/project`)
 3. the SPT language, to be able to extract the tests from the specification
 4. (Optionally) the Stratego language if we want to be able to execute the `run` or `transform` expectations
2. You should already have your tests and your language project, so next up is the SPT language. This is in the same repo as the command line runner:

```
$ cd spt/org.metaborg.meta.lang.spt
$ mvn verify
```

3. If you want to use the `run` and `transform` expectations, you also need the Stratego language:

```
$ git clone https://github.com/metaborg/stratego.git
$ cd stratego/org.metaborg.meta.lang.stratego
$ mvn verify
```

4. Now we can run the tests:

```
$ java -jar spt/org.metaborg.spt.cmd/target/org.metaborg.spt.cmd-2.0.0-SNAPSHOT.jar -h
Usage: <main class> [options]
Options:
  --help, -h
    Shows usage help
    Default: false
  --lang, -ol
```

(continues on next page)

(continued from previous page)

```

        Location of any other language that should be loaded
        Default: []
    * --lut, -l
        Location of the language under test
    * --spt, -s
        Location of the SPT language
    --start-symbol, -start
        Start Symbol for these tests
    * --tests, -t
        Location of test files
$ java -jar spt/org.metaborg.spt.cmd/target/org.metaborg.spt.cmd-2.0.0-SNAPSHOT.
↪ jar
    --lut /path/to/MiniJava/project
    --tests /path/to/test/project
    --spt spt/org.metaborg.meta.lang.spt
    --lang stratego/org.metaborg.meta.lang.stratego

```

16.3.3 Run using the SPT Framework

The SPT framework at <https://github.com/metaborg/spt> offers a Java API to run SPT test suites. The framework is split between the generic part (`org.metaborg.mbt.core` - MetaBorg Testing (MBT)) and the Spoofax specific part (`org.metaborg.spt.core` SPOofax Testing (SPT)).

The first step in running tests is to extract them from an SPT test suite. `org.metaborg.mbt.core` provides a Java object model to represent SPT test cases. To extract test cases from a test suite to the Java model, you can use the `ITestCaseExtractor`. You can either implement this for your own version of the SPT language, or use our SPT language (`org.metaborg.meta.lang.spt`) and our extractor (`ISpoofaxTestCaseExtractor`).

Now that you have the tests in Java objects, you can execute them with the `ITestCaseRunner`. If the language you are testing is not integrated with Metaborg Core, you will either have to do so and subclass the `TestCaseRunner`, or make your own implementation for the `ITestCaseRunner`. If your language under test *is* integrated with Metaborg Core (this is the case for all languages created with Spoofax), you can use our `ISpoofaxTestCaseRunner`.

For an example on how to use dependency injection to obtain the correct classes and extract and run SPT tests using the Java API, see the `TestRunner` class at (<https://github.com/metaborg/spt/tree/master/org.metaborg.spt.core>).

16.3.4 Run using Maven

For regression testing and continuous integration, it can be useful to be able to execute tests from a maven build. To do so, create a `pom.xml` file in your test project with the following content:

```

<?xml version="1.0" encoding="UTF-8"?>
<project
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
↪ maven-4.0.0.xsd"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
>
  <modelVersion>4.0.0</modelVersion>
  <groupId>your.group.id</groupId>
  <artifactId>your.test.project.name</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>spoofax-test</packaging>

```

(continues on next page)

(continued from previous page)

```
<parent>
  <groupId>org.metaborg</groupId>
  <artifactId>parent.language</artifactId>
  <version>2.1.0-SNAPSHOT</version>
</parent>

<dependencies>
  <dependency>
    <groupId>your.group.id</groupId>
    <artifactId>your.language.under.test.id</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <type>spoofax-language</type>
  </dependency>
  <dependency>
    <groupId>org.metaborg</groupId>
    <artifactId>org.metaborg.meta.lang.spt</artifactId>
    <version>${metaborg-version}</version>
    <type>spoofax-language</type>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.metaborg</groupId>
      <artifactId>spoofax-maven-plugin</artifactId>
      <version>${metaborg-version}</version>
      <configuration>
        <languageUnderTest>your.group.id:your.language.under.test.id:1.0.0-SNAPSHOT
↪</languageUnderTest>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

You should now be able to execute the tests with `mvn verify`.

16.4 The SPT Framework

Todo: This part of the documentation has not been written yet.

Build and Develop Language Projects

This is the reference manual on build and development tools for Spoofox language development.

17.1 Maven Builds

This page describes how to build Spoofox languages with Maven.

17.1.1 Requirements

JDK8 or higher

A recent version of JDK 8 or higher is required to build Spoofox languages with Maven. Old versions may not accept the LetsEncrypt certificate of our artifact server. We try to keep Spoofox compatible with newer JDKs (such as JDK13) as well, but if they do not work, please try JDK8.

Maven

We require Maven version 3.5.4 or higher, except Maven version 3.6.1 or 3.6.2 (due to bugs in those Maven versions). Maven can be downloaded and installed from <https://maven.apache.org/download.cgi>. On macOS, Maven can be easily installed with Homebrew by executing `brew install maven`.

Confirm the installation and version by running `mvn --version`.

By default, Maven does not assign a lot of memory to the JVM that it runs in, which may lead to out of memory exceptions during builds. To increase the allocated memory, set the `MAVEN_OPTS` environment variable:

```
export MAVEN_OPTS="-Xms512m -Xmx1024m -Xss16m"
```

To make this permanent, add this line to your `.bashrc/.profile` or equivalent for your operating system/shell.

17.1.2 Maven Build

The local Maven build starts from the generate new project wizard (you need the generated files). New Project > New Spoofox language project > select all generation options. This generates 6 projects in total. In this tutorial, our language name and ID is `entity`.

Move the six projects to a new parent directory, and create a `pom.xml` file in this directory with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<project
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
  ↪maven-4.0.0.xsd"
  xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
  ↪XMLSchema-instance">

  <modelVersion>4.0.0</modelVersion>
  <artifactId>entity.build</artifactId>
  <version>0.1.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <parent>
    <groupId>org.metaborg</groupId>
    <artifactId>parent</artifactId>
    <version>REPLACEME_SPOOFAX_VERSION</version>
    <relativePath />
  </parent>

  <modules>
    <module>entity</module>
    <module>entity.eclipse</module>
    <module>entity.eclipse.feature</module>
    <module>entity.eclipse.site</module>
    <module>entity.test</module>
  </modules>

  <repositories>
    <repository>
      <id>metaborg-release-repo</id>
      <url>https://artifacts.metaborg.org/content/repositories/releases/</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
    <repository>
      <id>metaborg-snapshot-repo</id>
      <url>https://artifacts.metaborg.org/content/repositories/snapshots/</url>
      <releases>
        <enabled>false</enabled>
      </releases>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    </repository>
    <repository>
      <id>spoofox-eclipse-repo</id>
```

(continues on next page)

(continued from previous page)

```

        <url>https://artifacts.metaborg.org/content/unzip/releases-unzipped/org/
        ↪metaborg/org.metaborg.spoofax.eclipse.update.site/|rel-version|/org.metaborg.spoofax.
        ↪eclipse.update.site-|rel-version|-assembly.zip-unzip/</url>
        <layout>p2</layout>
        <releases>
          <enabled>>false</enabled>
        </releases>
        <snapshots>
          <enabled>>false</enabled>
        </snapshots>
      </repository>
    </repositories>

    <pluginRepositories>
      <pluginRepository>
        <id>metaborg-release-repo</id>
        <url>https://artifacts.metaborg.org/content/repositories/releases/</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>>false</enabled>
        </snapshots>
      </pluginRepository>
      <pluginRepository>
        <id>metaborg-snapshot-repo</id>
        <url>https://artifacts.metaborg.org/content/repositories/snapshots/</url>
        <releases>
          <enabled>>false</enabled>
        </releases>
        <snapshots>
          <enabled>true</enabled>
        </snapshots>
      </pluginRepository>
    </pluginRepositories>
  </project>

```

and replace `REPLACEME_SPOOFAX_VERSION` with the version of Spoofax you are using.

Copy the `.mvn` folder from your language folder to the parent folder. e.g. `root/entity/.mvn -> root/.mvn`. (Error message otherwise: `[ERROR] Failed to execute goal org.metaborg:spoofax-maven-plugin:2.1.0-SNAPSHOT:clean (default-clean) on project entity: Building clean input failed unexpectedly: Language for dependency org.metaborg:org.metaborg.meta.lang.esv:2.1.0-SNAPSHOT does not exist -> [Help 1]`)

Fix the generated test yaml file (known issue) e.g. `root/entity.test/metaborg.yaml`. (Error message otherwise: `[ERROR] Field 'id' must be set`)

```

---
dependencies:
  compile:
    - org.example:entity:0.1.0-SNAPSHOT
    - org.metaborg:org.metaborg.meta.lang.spt:${metaborgVersion}

```

to

```
---
id: org.example:entity.test:0.1.0-SNAPSHOT
name: entity
dependencies:
  compile:
    - org.example:entity:0.1.0-SNAPSHOT
    - org.metaborg:org.metaborg.meta.lang.spt:${metaborgVersion}
```

Now you can build the language with `mvn clean verify`, with the final output succeeding with something like:

```
[INFO] -----
[INFO] Building entity.build 0.1.0-SNAPSHOT
[INFO] -----
[INFO] --- maven-clean-plugin:3.0.0:clean (default-clean) @ entity.build ---
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] entity ..... SUCCESS [ 31.033 s]
[INFO] entity.eclipse ..... SUCCESS [ 1.252 s]
[INFO] entity.eclipse.feature ..... SUCCESS [ 0.469 s]
[INFO] entity.eclipse.site ..... SUCCESS [ 3.776 s]
[INFO] entity.test ..... SUCCESS [ 0.140 s]
[INFO] entity.build ..... SUCCESS [ 0.013 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 51.354 s
[INFO] Finished at: 2016-09-16T16:58:25+02:00
[INFO] Final Memory: 280M/963M
[INFO] -----
```

17.2 Continuous Integration

This page describes how to build Spoofox languages on a Jenkins buildfarm or using an alternative platform such as GitHub Actions.

Full continuous integration includes:

- Building the language on the buildfarm
- Running SPT tests as part of the build
- Publishing an eclipse updatesite for the language
- Doing the above on every commit, and on every spoofox-master update

Setting up continuous integration is a two step process. The first step is to setup a local maven build for building the language, running the tests and creating an update site. The second step is configuring Jenkins to perform these maven builds and publish the artifacts.

17.2.1 Local Maven Build

Follow the [guide for getting a local Maven build](#) going first.

17.2.2 Build on Jenkins

(Note: can be skipped in GitHub organizations MetaBorg and MetaBorgCube)

New Item> enter a name and choose Multibranch Pipeline

Add the git repo Branch Sources>Add source>Git. Fill in project repository such as `https://github.com/MetaBorgCube/metaborg-entity.git`, select credentials, and save.

You should now get a message saying that the repository has branch but does not meet the criteria, as the Jenkinsfile is not setup yet.

17.2.3 Jenkins configuration

Create the file Jenkinsfile in the root of the repository containing (be sure to update the update site path, and to change the slack integration channel or comment out the slack integration):

```
properties([
  pipelineTriggers([
    upstream(
      threshold: hudson.model.Result.SUCCESS,
      upstreamProjects: '/metaborg/spoofax-releng/master' // build this project after
↳ Spoofax-master is built
    )
  ]),
  buildDiscarder(logRotator(artifactNumToKeepStr: '3')),
  disableConcurrentBuilds() //disables parallel builds
])

node{
  try{
    notifyBuild('Started')

    stage('Checkout') {
      checkout scm
      sh "git clean -fXd"
    }

    stage('Build and Test') {
      withMaven(
        //mavenLocalRepo: "${env.JENKINS_HOME}/m2repos/${env.EXECUTOR_NUMBER}", //
↳ http://yellowgrass.org/issue/SpoofaxWithCore/173
        mavenLocalRepo: ".repository",
        mavenOpts: '-Xmx1G -Xms1G -Xss16m'
      ){
        sh 'mvn -B -U clean verify -DforceContextQualifier=\$(date +%Y%m%d%H%M) '
      }
    }

    stage('Archive') {
      archiveArtifacts(
        artifacts: 'yourlanguage.eclipse.site/target/site/',
        excludes: null,
        onlyIfSuccessful: true
      )
    }
  }
```

(continues on next page)

(continued from previous page)

```

    stage('Cleanup') {
        sh "git clean -fXd"
    }

    notifyBuild('Succeeded')

} catch (e) {

    notifyBuild('Failed')
    throw e

}
}

def notifyBuild(String buildStatus) {
    def message = ""${buildStatus}: ${env.JOB_NAME} [${env.BUILD_NUMBER}] ${env.BUILD_
↪URL}""

    if (buildStatus == 'Succeeded') {
        color = 'good'
    } else if (buildStatus == 'Failed') {
        color = 'danger'
    } else {
        color = '#4183C4' // Slack blue
    }

    slackSend (color: color, message: message, channel: '#some-slack-channel')
}

```

Go to the Jenkins project > Branch Indexing > Run now. This should trigger the build of the master branch.

17.2.4 Trigger Jenkins on commit

(Note: can be skipped in GitHub organizations MetaBorg and MetaBorgCube)

In order to trigger Jenkins to build on every commit we need to install a GitHub service. In the GitHub repository go to Settings > Integrations & services > Add service > Jenkins (Git plugin) (not GitHub plugin) and provide the jenkins url (for example <http://buildfarm.metaborg.org/>)

17.2.5 Build badge on GitHub

For a GitHub build-badge add the following the the readme file:

```

[![Build status](http://buildfarm.metaborg.org/job/Entity/job/master/badge/
↪icon)](http://buildfarm.metaborg.org/job/Entity/job/master/)

```

TODO: figure out how to use Promoted Builds to promote spoofox-master only if language build succeeds.

17.2.6 CI using GitHub Actions

Using GitHub Actions is an alternative to Jenkins for setting up CI using Maven. Enable it by adding the file `.github/workflows/ci.yml` to your repository with the following contents:


```

name: CI

on: [push]

jobs:
  build:

    runs-on: ubuntu-latest
    container: maven:3.5.4-jdk-8

    steps:
    - uses: actions/checkout@v2
    - name: Maven build
      run: mvn clean verify

```

This configures your repository with a CI workflow that runs the build on every push. Publishing the language is not included in this configuration.

See the [Spoofax definition of MiniZinc](#) for an example (config and actions).

Similar to Jenkins, you can add a build status badge by adding the following to the readme file:

```

![Build status] (https://github.com/namespace/project/workflows/CI/badge.svg)

```

17.3 IntelliJ IDEA Environment

Spoofax for IntelliJ IDEA is an IntelliJ IDEA plugin for creating and building languages with the [Spoofax Language Workbench](#). Spoofax is a platform for developing textual domain-specific languages, and its language workbench includes tools and meta languages for defining syntax, name bindings, types and tree transformations, among others.

17.3.1 Installation

Recently we created a Spoofax plugin for IntelliJ IDEA.

To install the plugin, either:

- clone this repository, then execute `./gradlew runIde` (or `gradlew.bat runIde` on Windows) from the repository's `org.metaborg.intelliJ` subdirectory to start an instance of IntelliJ IDEA with the Spoofax plugin loaded; or
- ensure you have Git and a JDK installed, then execute this from the command line; or:

```

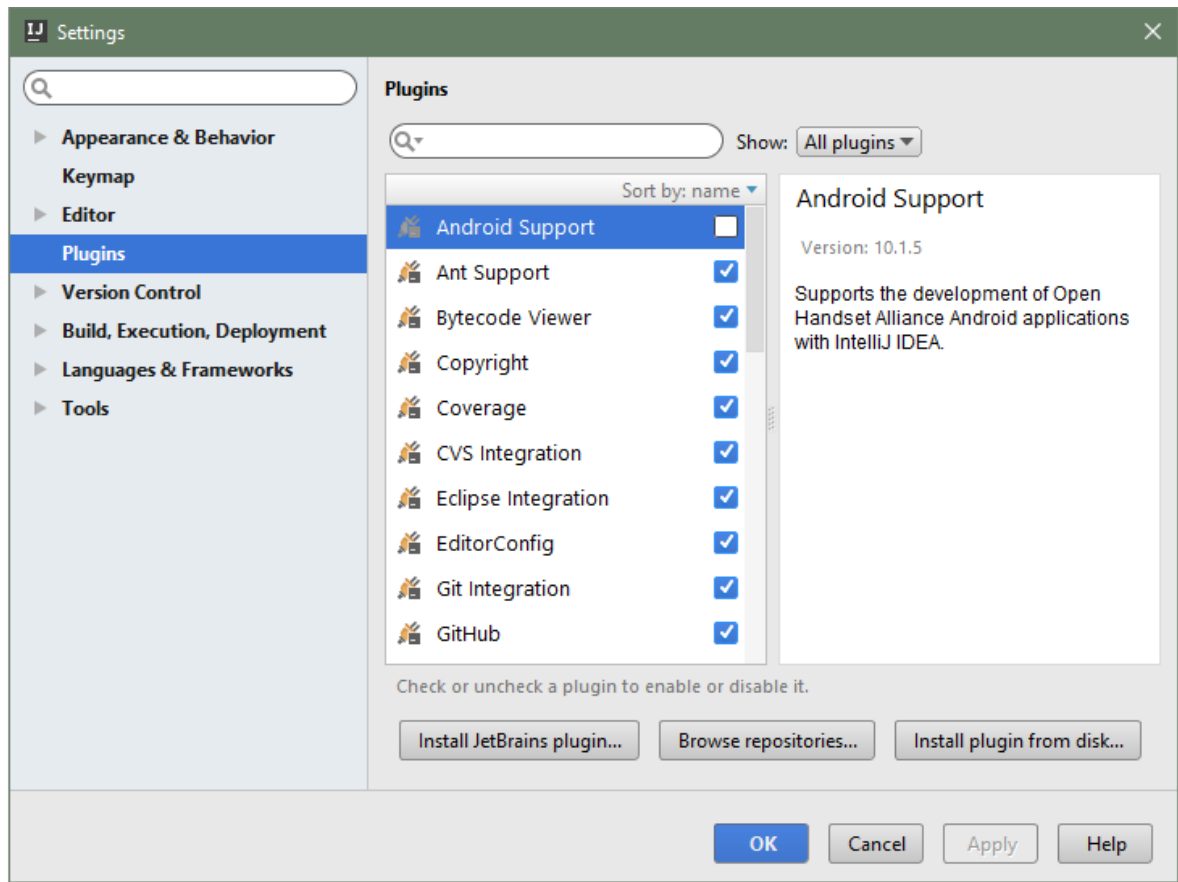
curl https://raw.githubusercontent.com/metaborg/spoofax-intellij/master/
↪ repository/install.sh -sSLf | bash

```

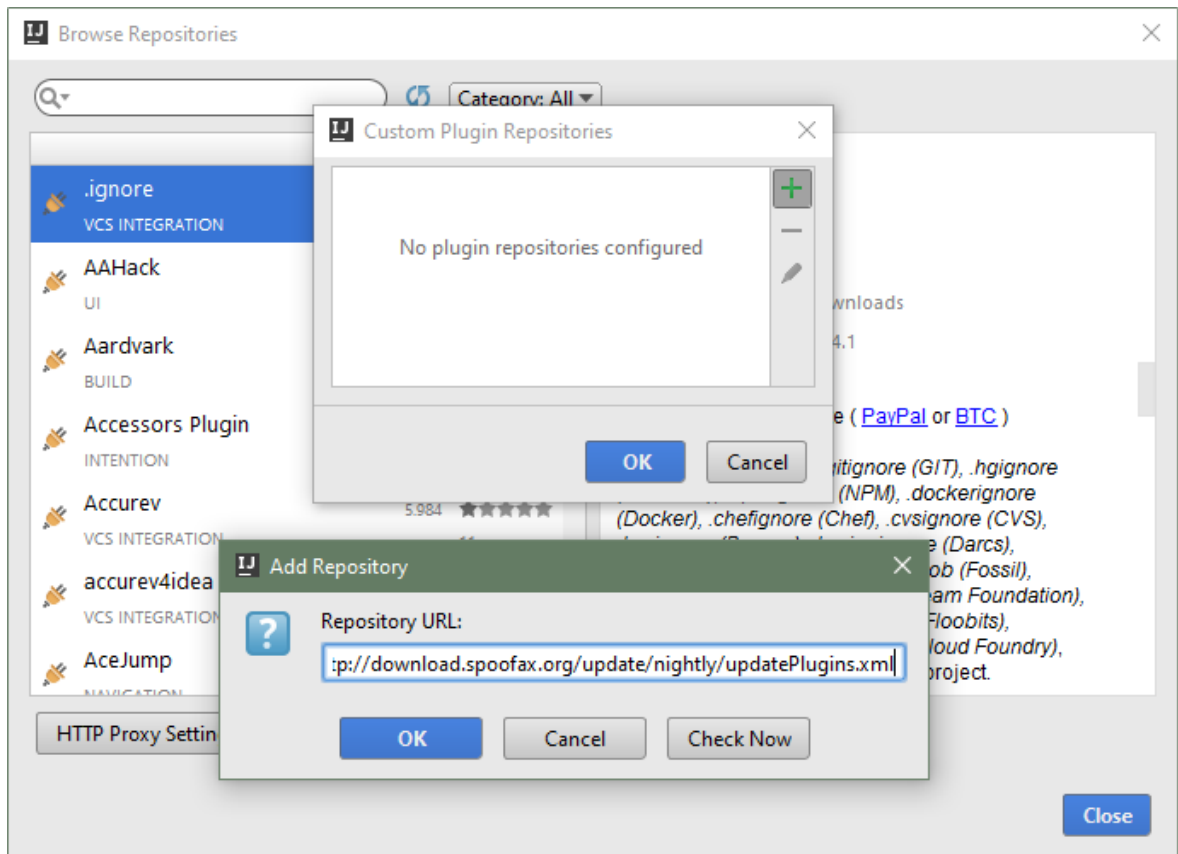
Installing the Spoofax plugin in IntelliJ IDEA

If you already have an IntelliJ IDEA installation or manually downloaded one, here are the detailed instructions for installing the Spoofax plugin:

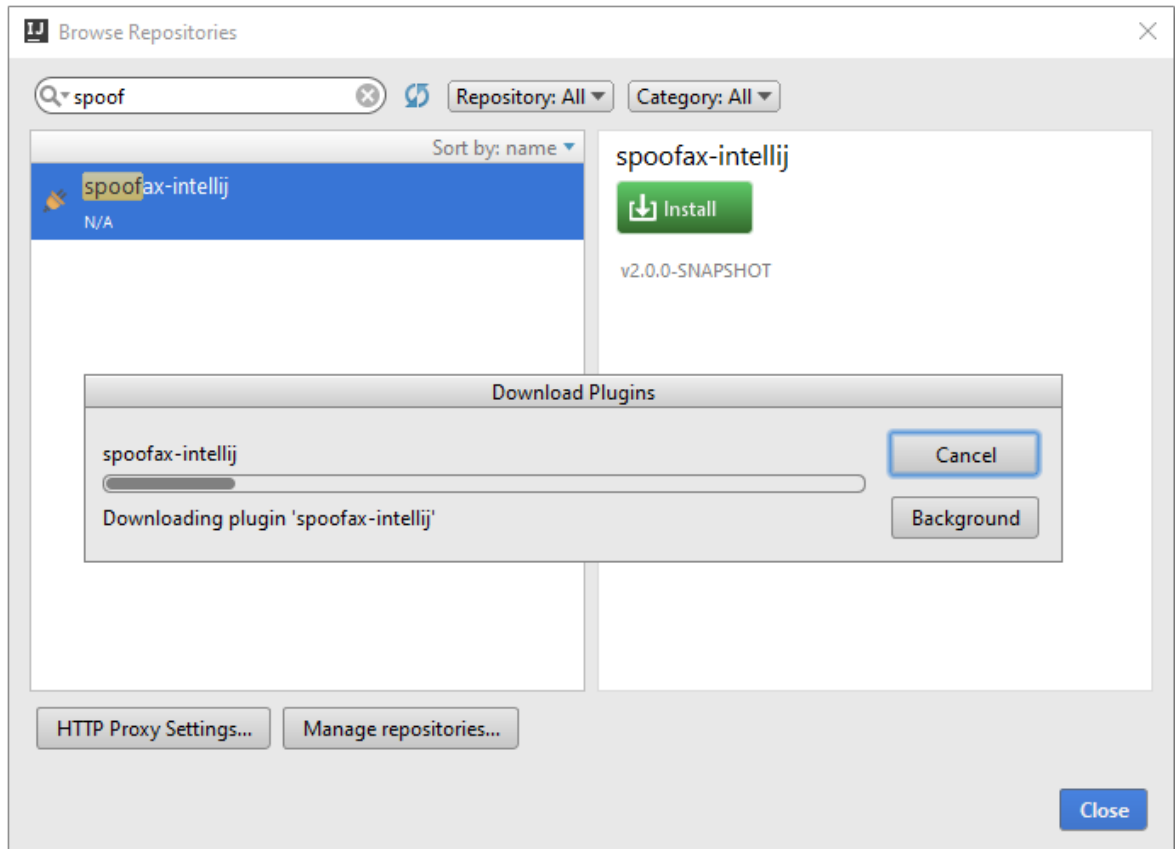
1. Go to the *File* menu, *Settings*, and click the *Plugins* tab. Or if you're on the welcome screen, click the *Configure* button at the bottom, then click *Plugins*. Here you can manage the plugins.



2. Click the *Browse repositories...* button, then in the new window click *Manage repositories...*. This window allows you to add and remove custom repositories.



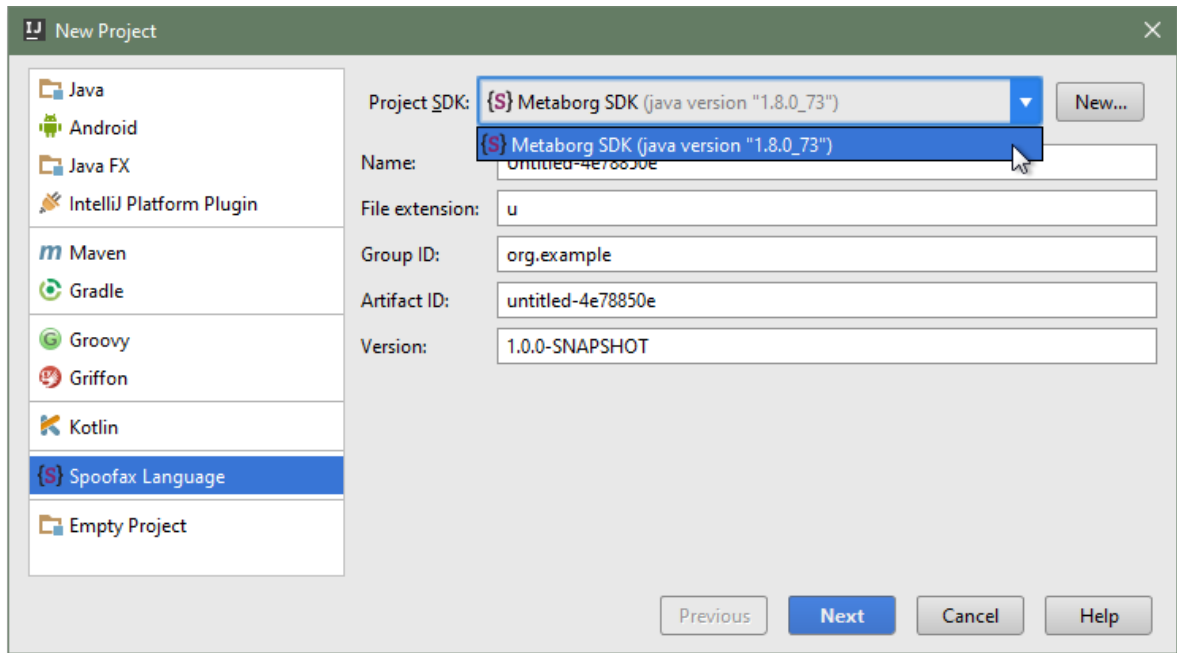
3. Add the above mentioned repository URL, and click *OK* to close the dialog.
4. In the *Browse repositories* window, find and select the *spoofax-intellij* plugin.
5. Click the green *Install* button, and restart IntelliJ after the plugin's installation.



17.3.2 Creating a new language specification

Follow this quick start guide to create a new Spoofax language specification project to define your own language.

1. Choose *Create New Project* from the welcome dialog, or the *File* → *New* → *Project...* menu item, to open the *New Project* dialog.
2. Select the *Spoofax Language* project type.
3. Select the *Metaborg SDK* as the project's SDK.

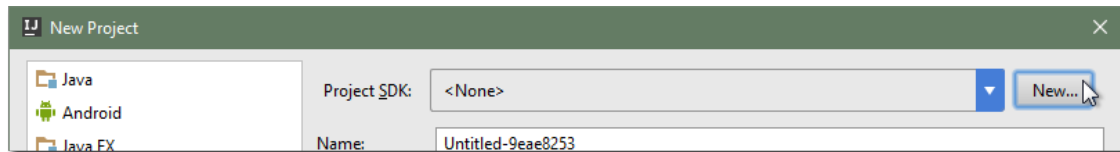


Select

the "Metaborg SDK"

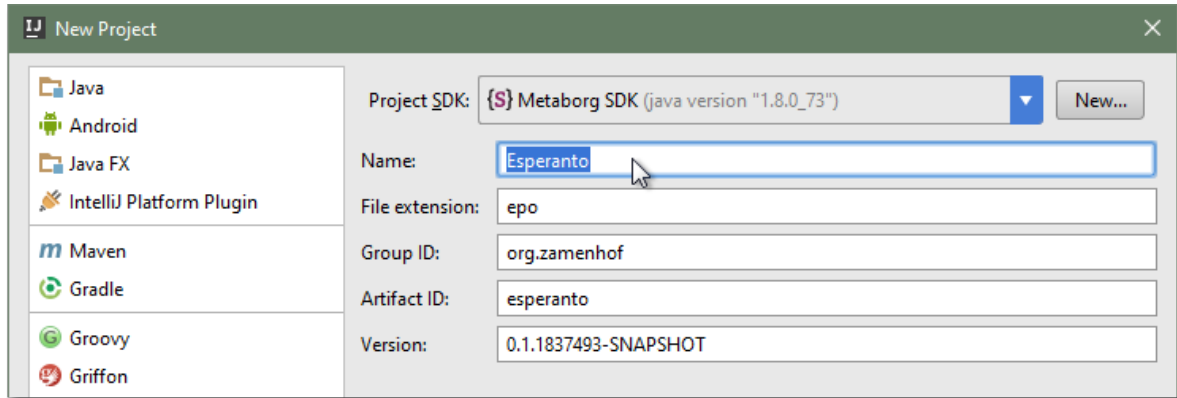
Note: If the *Project SDK* displays *<None>* and you can't select the Metaborg SDK, then you need to add it first.

- (a) Click the *New...* button next to the *Project SDK* field.



- (b) You may get a warning if you have no Java JDK configured. Click *OK* and configure the JDK's home location. The suggested home directory is usually correct. Click *OK*.
- (c) Select a home directory for the Metaborg SDK. The default is the Spoofox plugin's `lib/` folder, which is sufficient as it contains all the core dependencies.
- (d) Click *OK*.

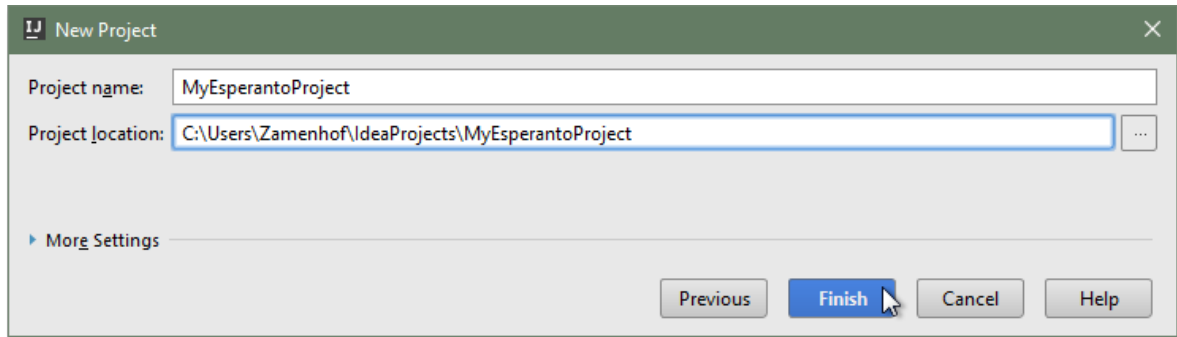
4. Change the fields to suit your needs.



Change

the fields

5. Click *Next*.
6. Pick a project name and location, and click *Finish*.



Pick

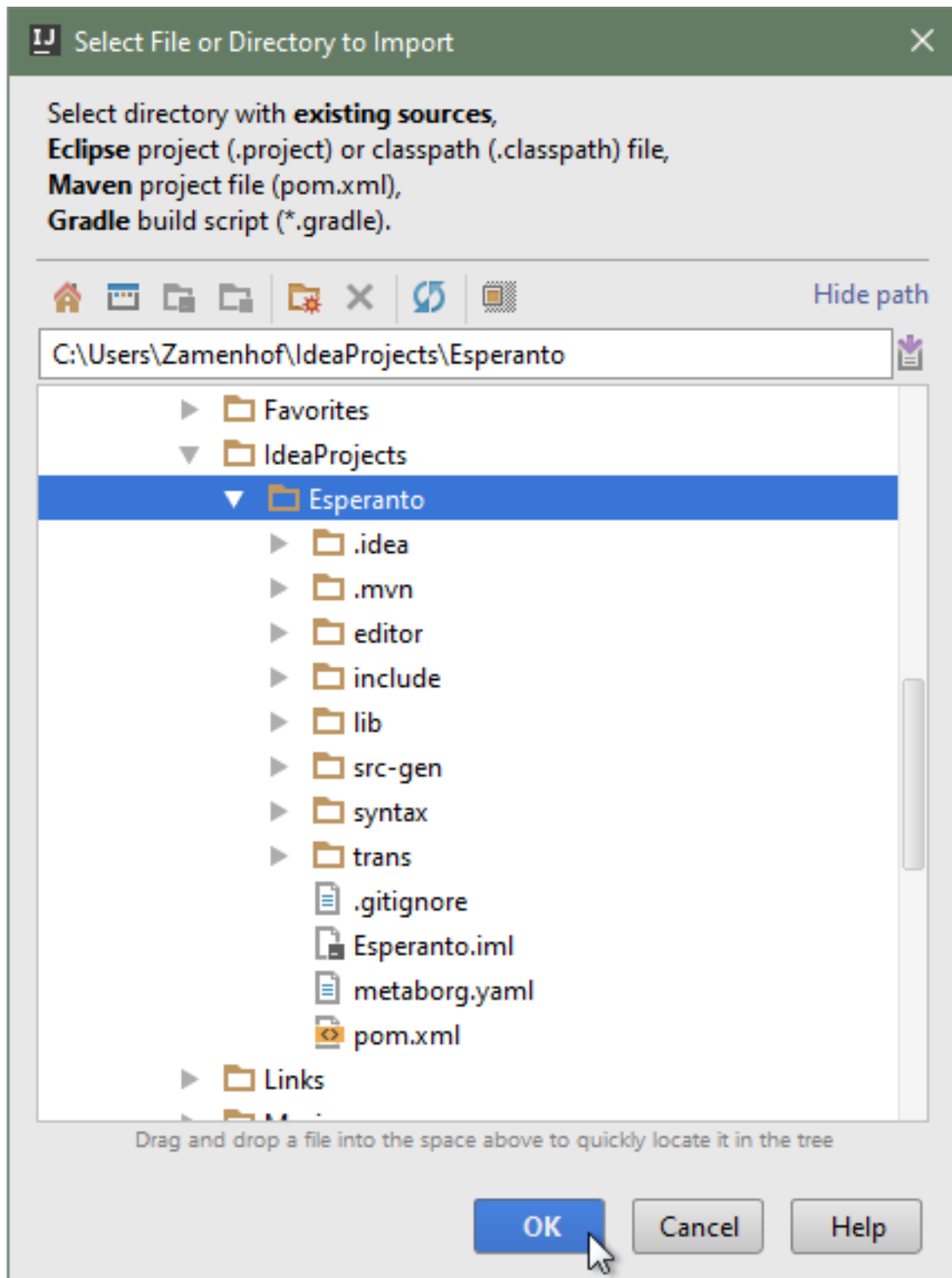
a project name

The created Spoofox language specification project will have a `metaborg.yaml` file, which specifies the configuration and language dependencies of the specification.

17.3.3 Importing an existing language specification

Follow this quick start guide to import an existing Spoofox language specification into IntelliJ.

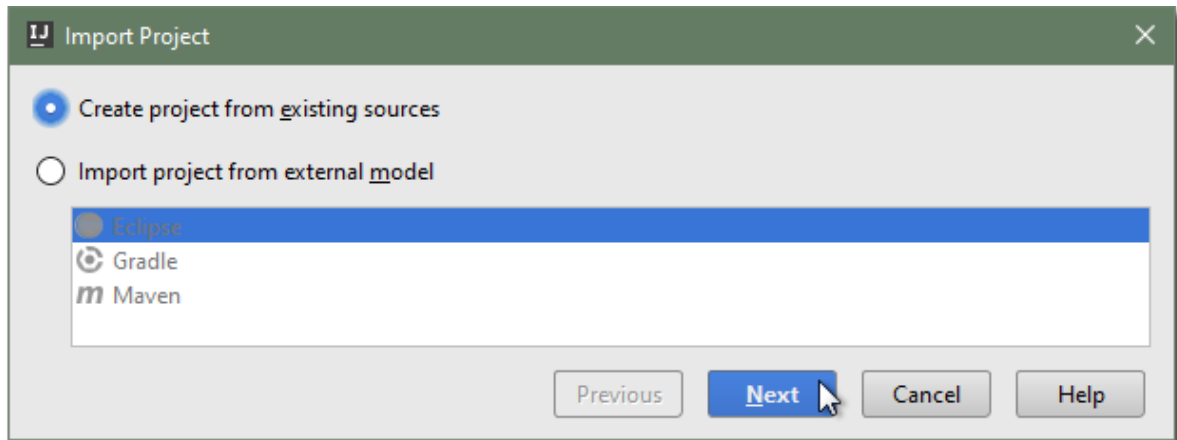
1. Choose *Import Project* from the welcome dialog, or the *File* → *New* → *Project from Existing Sources...* menu item, to open the *Import Project* dialog.
2. Browse to the root folder of the project, and click *OK*.



Browsing

to the project

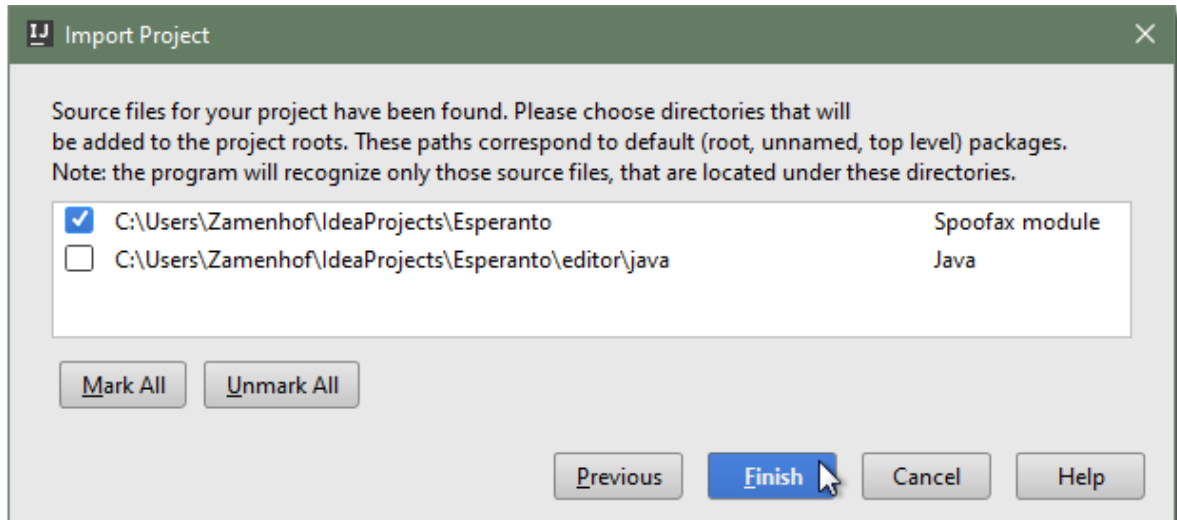
3. Select *Create project from existing sources* and click *Next*.



Browsing

to the project

4. Pick a project name, and ensure the location is correct. Click *Next*.
5. Ensure only the *Spoofax module* root is checked.



Check

the project roots

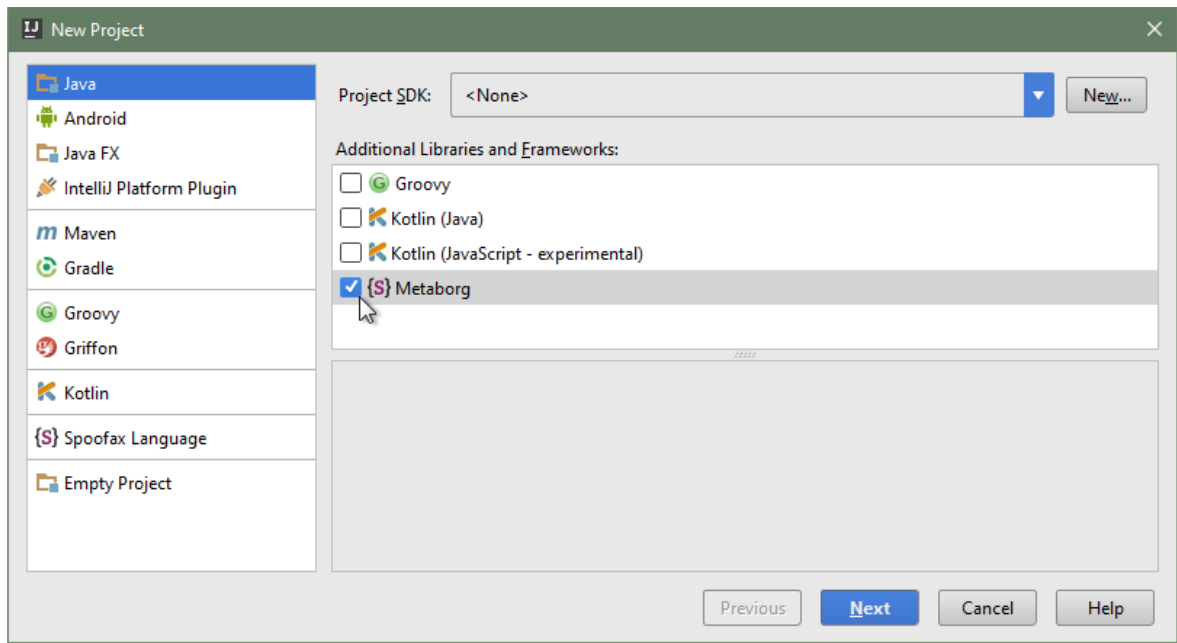
6. Click *Finish*.

The imported Spoofax language specification project has a `metaborg.yaml` file, which specifies the configuration and language dependencies of the specification.

17.3.4 Creating a new project

Follow this quick start guide to create a new Java project in which you can use Metaborg languages.

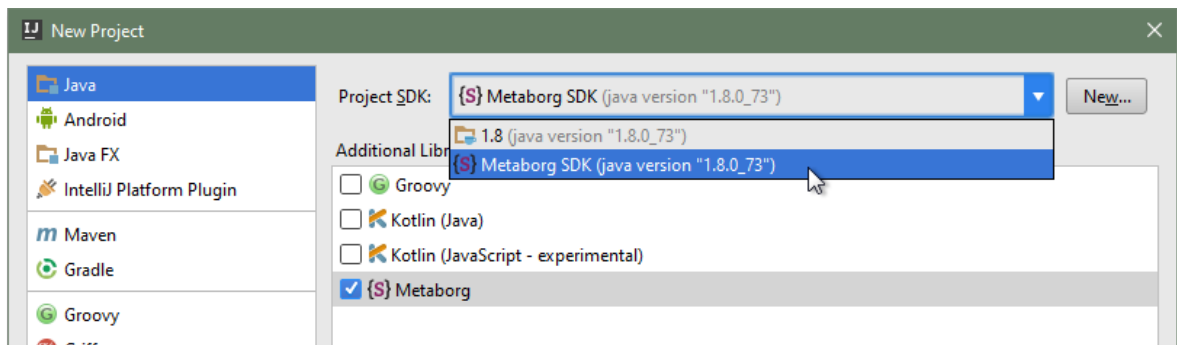
1. Choose *Create New Project* from the welcome dialog, or the *File* → *New* → *Project...* menu item, to open the *New Project* dialog.
2. Select the *Java* project type.
3. Check the *Metaborg* framework.



"New

Project" dialog

4. Select the *Metaborg SDK* as the project's SDK.

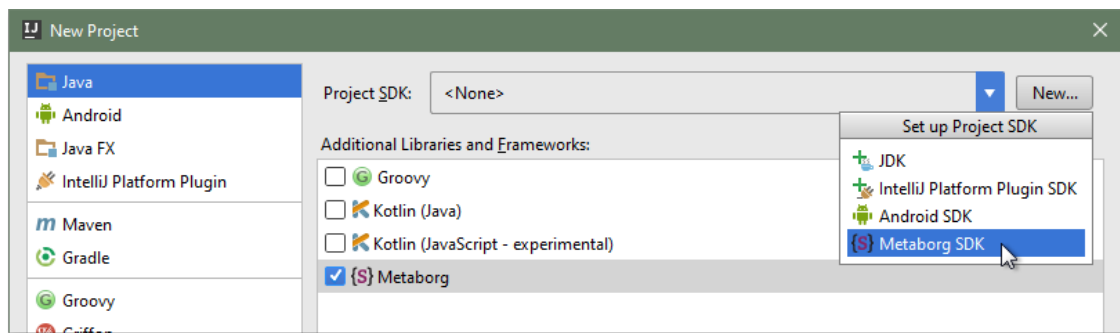


Select

the "Metaborg SDK"

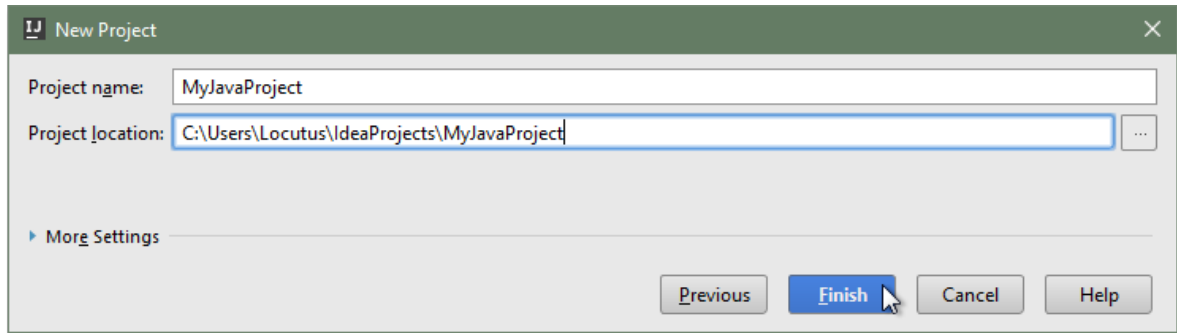
Note: If the *Project SDK* displays *<None>* and you can't select the Metaborg SDK, then you need to add it first.

- (a) Click the *New...* button next to the *Project SDK* field.
- (b) Select *Metaborg SDK*.



- (c) You may get a warning if you have no Java JDK configured. Click *OK* and configure the JDK's home location. The suggested home directory is usually correct. Click *OK*.
- (d) Select a home directory for the Metaborg SDK. The default is the Spoofax plugin's `lib/` folder, which is sufficient as it contains all the core dependencies.
- (e) Click *OK*.

5. Click *Next*.
6. Pick a project name and location, and click *Finish*.




Pick

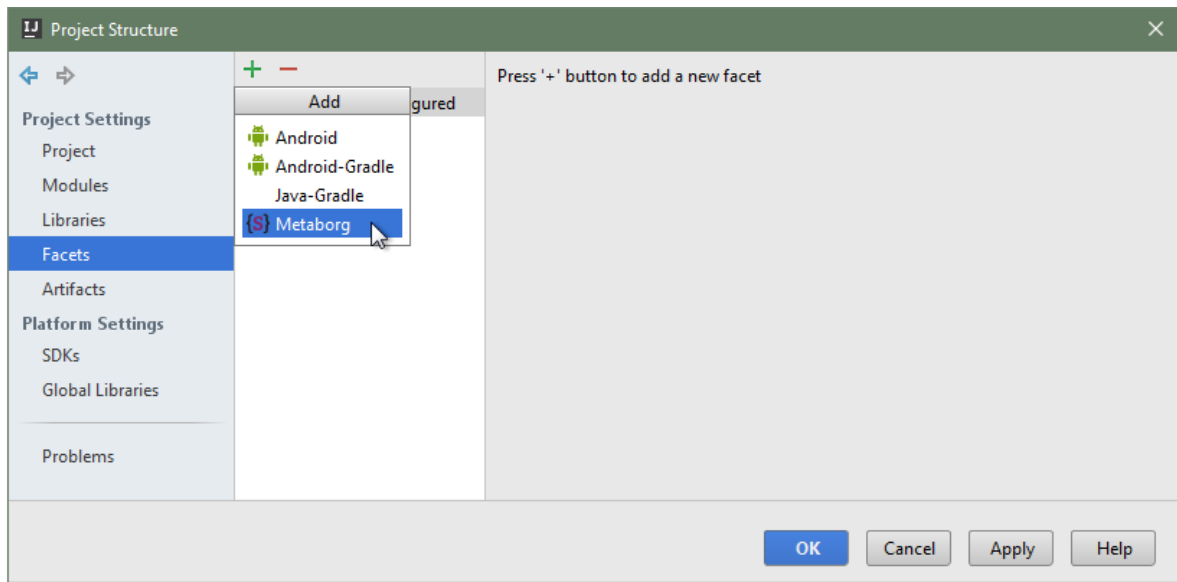
a project name

The created Java project will have the *Metaborg facet*, indicating that the project uses Metaborg languages. It will also have a `metaborg.yaml` file, which specifies the language dependencies of the project.

17.3.5 Importing an existing project

Follow this quick start guide to change an existing Java project such that you can use Metaborg languages in it.

1. Open the project in IntelliJ IDEA.
2. Go to the *Project Structure* dialog, either through the *File* → *Project Structure...* menu item, or by pressing `Ctrl+ Alt+Shift+S`.
3. Go to the *Facets* tab.
4. Click the  Plus button and select *Metaborg* from the drop-down menu.



"Project

Structure" dialog

5. Select the module to add the facet to, and click *OK*.
6. In the *Modules* tab, select the module.
7. Go to the *Dependencies* tab.
8. Select the *Metaborg SDK* as the module's SDK.

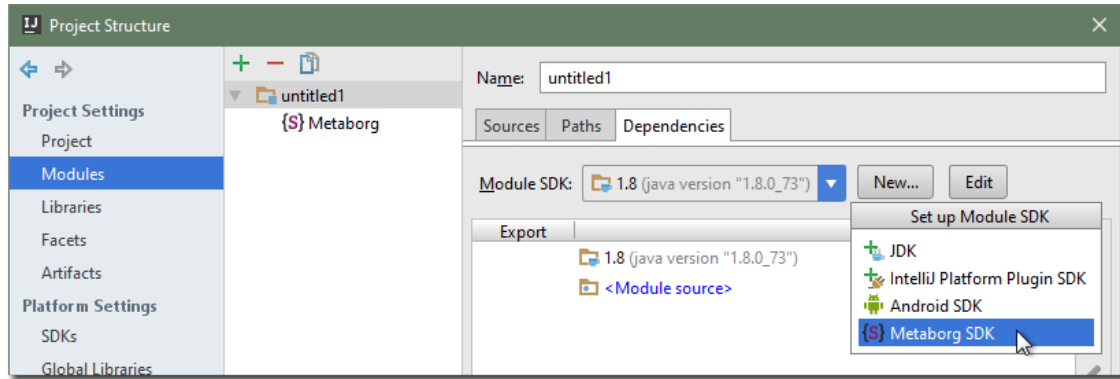


Select

the "Metaborg SDK"

Note: If the *Module SDK* displays *<None>* and you can't select the Metaborg SDK, then you need to add it first.

- (a) Click the *New...* button next to the *Module SDK* field.
- (b) Select *Metaborg SDK*.



- (c) You may get a warning if you have no Java JDK configured. Click *OK* and configure the JDK's home location. The suggested home directory is usually correct. Click *OK*.
- (d) Select a home directory for the Metaborg SDK. The default is the Spoofax plugin's `lib/` folder, which is sufficient as it contains all the core dependencies.
- (e) Click *OK*.

9. Click *OK* to apply the changes.

The Java project will have the *Metaborg facet*, indicating that the project uses Metaborg languages. It will also have a `metaborg.yaml` file, which specifies the language dependencies of the project.

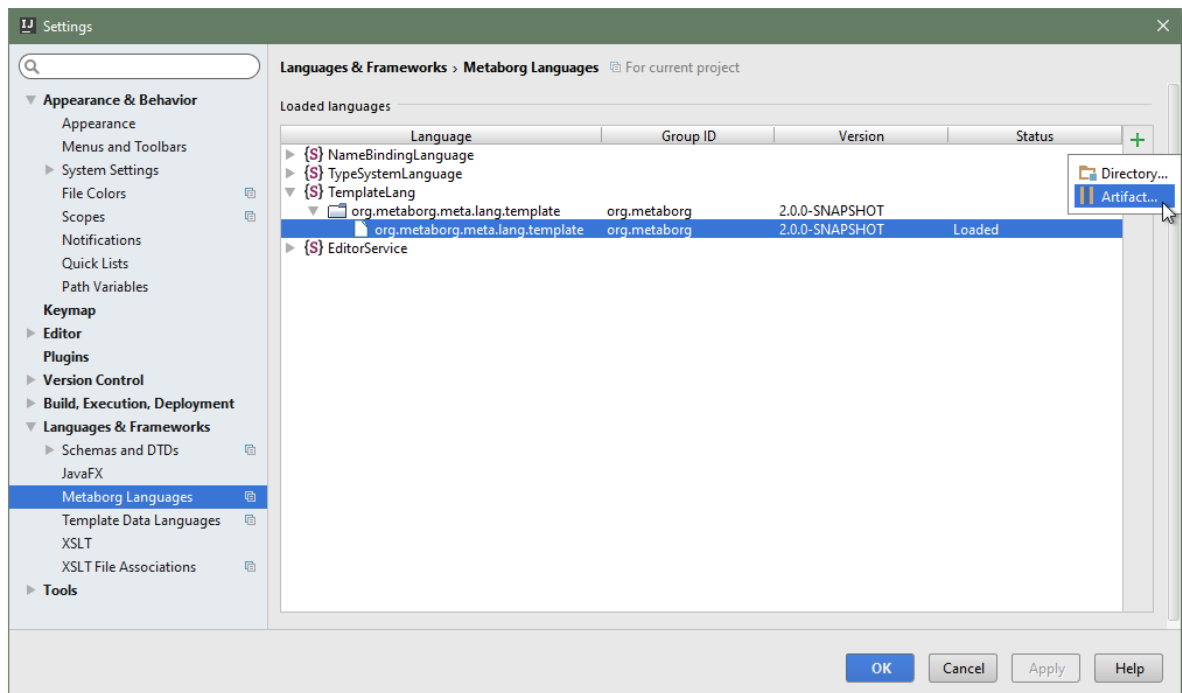
17.3.6 Load and unload languages

Spoofax for IntelliJ supports dynamic loading and unloading of languages. Follow this quick start guide to load or unload a language from the IDE. Note that alternatively you can edit the project's `metaborg.yaml` file to alter the dependencies, and reload the project. Languages loaded through the *Settings* dialog are loaded for every instance of the application.

1. Choose the *File* → *Settings* menu or press the `Ctrl+Alt+S` keyboard combination, to open the *Settings* dialog.
2. Go to the *Languages & Frameworks* → *Metaborg Languages* page.



3. Use the Plus and Minus buttons to load or unload a language.



Changing

the loaded languages

4. Click *OK* to apply the changes.

17.3.7 See also

- *IntelliJ IDEA Plugin Internals*

Configuring Language Projects

This is the reference manual on configuring Spoofax language projects.

18.1 Configuration Reference

Configuration is specified in the `metaborg.yaml` file at the root of the project.

18.1.1 Format

The configuration is written in [YAML](#), a human-friendly textual data format. YAML is indentation sensitive, be sure to properly indent nested elements with 2 spaces.

We use the [commons-configuration2](#) framework to process configuration, which supports variables, nesting, and lists.

Variables

Any existing configuration option can be used as a variable with the `${var}` syntax, for example:

```
id: org.metaborg:org.metaborg.meta.lang.sdf:${metaborgVersion}
metaborgVersion: 2.0.0-SNAPSHOT
```

Here, the `${metaborgVersion}` variable reference is replaced with `2.0.0-SNAPSHOT` when reading the `id` configuration option. Note that the order in which configuration options and variable references occur does not matter.

The `${path:root}` builtin property can be used to point to the root of the language specification. Paths in the configuration must be absolute unless stated otherwise. Use `${path:root}/` to make paths absolute, relative to the project root, where required.

Furthermore, environment variables can be used through `${env:}`, for example `${env:PATH}`. See the documentation on [variable interpolation](#) for more detailed informations on how variables work.

Nesting

Configuration options can be nested by nesting YAML objects, for example:

```
language:
  sdf:
    version: sdf2
```

results in the `language.sdf.version` option being set to `sdf2`, and can be referenced with a variable using `${language.sdf.version}`. The same option can be set in the following way:

```
language.sdf.version: sdf2
```

Lists

Lists are supported using the YAML list syntax, for example:

```
dependencies:
  compile:
    - org.metaborg:org.metaborg.meta.lang.esv:${metaborgVersion}
    - org.metaborg:org.metaborg.meta.lang.nabl:${metaborgVersion}
    - org.metaborg:org.metaborg.meta.lang.test:${metaborgVersion}
```

results in the `dependencies.compile` option being set to a list with elements:

- `org.metaborg:org.metaborg.meta.lang.esv:${metaborgVersion}`
- `org.metaborg:org.metaborg.meta.lang.nabl:${metaborgVersion}`
- `org.metaborg:org.metaborg.meta.lang.test:${metaborgVersion}`

18.1.2 Options

All supported configuration options for projects are listed here.

End-user project configuration

An end-user project is a project that contains programs of languages, intended to be developed by an end-user of those languages. The configuration for an end-user project specifies dependencies.

dependencies

Compile and source dependencies to other language components, and dependencies to Java artifacts.

compile

List of compile dependencies to language components. A compile dependency to a language component indicates that this project uses files of that language, and as such its compiler should be invoked.

- Format: List of language component identifiers (see `id` option below)
- Default: None
- Example:

```
dependencies:
  compile:
    - org.metaborg:org.metaborg.meta.lang.esv:${metaborgVersion}
```


source

List of source dependencies to language components. A source dependency to a language component indicates that this project uses exported files of that language or library.

- Format: List of language component identifiers (see `id` option below)
- Default: None
- Example:

```
dependencies:
  source:
    - org.metaborg:org.metaborg.meta.lib.analysis:${metaborgVersion}
```

java

List of dependencies to Java artifacts. A Java artifact dependency indicates that when this project is compiled, the Java artifact should be added to the compilation classpath. Spoofax currently does nothing with these dependencies, but they are used by Maven when compiling the project.

- Format: List of Maven artifact identifiers (see `id` option below)
- Default: None
- Example:

```
dependencies:
  java:
    - com.google.guava:guava:19.0
    - com.google.inject:guice:4.0
```

Warning: There is currently a bug in the version parser that parses versions with 1 or 2 components to a version with 3 components. For example, the version *1* is parsed to *1.0.0*, and *4.0* to *4.0.0*. This will cause build failures since dependencies with those versions cannot be found.

Language specification project configuration

A language specification project is a project that contains a language specification, which contains programs of meta-languages, intended to be developed by a language developer. It is a specialization of an end-user project, so all configuration options from end-user projects listed above, can also be used in language specification projects.

The following configuration options are mandatory:

id

Identifier of the language the language specification is describing.

- Format: `groupId:id:version` where `version` uses the Maven version syntax; `major.minor.patch(-qualifier)?`
- Example: `id: org.metaborg:org.metaborg.meta.lang.sdf:2.0.0-SNAPSHOT`

name

Name of the language the language specification is describing.

- Example: `name: SDF`

The following configuration options are optional and revert to default values when not specified:

dependencies

Compile and source dependencies to other language components, and dependencies to Java artifacts.

metaborgVersion

Version of the MetaBorg tools to use.

- Format: Maven version syntax (see `id` option)
- Default: Current version of the running Spoofax
- Example: `metaborgVersion: 2.0.0-SNAPSHOT`

contributions

List of language implementation identifiers the language component generated from this language specification contributes to.

- Format: List of language implementation names and identifiers (see `id` option)
- Default: Contribution to single language implementation with the same name and identifier of this language specification.
- Example:

```
contributions:
- name: Green-Marl
  id: com.oracle:greenmarl:1.5.0-SNAPSHOT
```

generates

List of language names this language specification generates files for, and into which directory.

- Format: List of language name and directory.
- Default: None
- Example:

```
generates:
- language: EditorService
  directory: src-gen
- language: Stratego-Sugar
  directory: src-gen
```

exports

List of files and directories this language specification exports for use in other language components, and optionally to which language those files and directories belong. Exported resources are packaged into the language component artifact when built.

- Format: List of exports. There are 3 kinds of exports which are described below
- Default: None

language-specific directory export

A language-specific directory export, exports a directory with files of a specific language. The directory **must be relative** to the project root. Includes and excludes are relative to the specified directory. These files can be used by other language components by specifying a source dependency on the language component built from this language specification.

- Format: Language name, path to directory, optional list of includes, and optional list of excludes
- Example:

```
exports:
- language: TemplateLang
  directory: syntax
- language: ds
```

(continues on next page)

(continued from previous page)

```

directory: src-gen/ds-signatures
- language: Stratego-Sugar
  directory: trans
  includes: "**/*.str"
- language: Stratego-Sugar
  directory: src-gen
  includes: "**/*.str"

```

Warning: Includes and excludes are only used to package the correct resources into the language component artifact, Spoofax Core does not use the includes and excludes at this moment. This may cause differences in behaviour between development and deployment environments.

language-specific file export

A language-specific file export, exports a single file of a specific language. The file **must be relative** to the project root. The file can be used by other language components by specifying a source dependency on the language component built from this language specification.

- Format: Language name, path to file
- Example:

```

exports:
- language: SDF
  file: include/libanalysis2.def

```

generic resource export

A generic resource export, exports any resources in a directory. The directory **must be relative** to the project root. Includes and excludes are relative to the specified directory. These files can be used for tasks specific to the language specification, for example to bundle library files with the language specification.

- Format: Relative path to directory, optional list of includes, and optional list of excludes
- Example:

```

exports:
- directory: ./
  includes:
    - lib-java/**/*
    - lib-webdsl/**/*
    - lib/webdsl/HQL-pretty.pp.af
    - lib/webdsl/WebDSL-pretty.pp.af

```

Warning: All paths are relative to the project root. Do **NOT** use `${path:root}` to make paths absolute!

Note: For directory exports, a list of includes and excludes can be optionally specified, using the [Ant pattern syntax](#). If no includes or excludes are specified, all files in the directory are assumed to be included recursively.

Note: Use `./` to use the root directory as export directory, `.` triggers an error in the YAML parser.

pardonedLanguages

List of language names that are pardoned; any errors they produce will not fail builds.

- Format: List of language names
- Default: None
- Example:

```
pardonedLanguages:
- EditorService
- Stratego-Sugar
- SDF
```

language

Language specific configuration options.

sdf

Configuration options for SDF2 and SDF3.

enabled

Whether to enable sdf (parse table, parenthesizer) for the current project or not.

- Format: Either `true` or `false`.
- Default: `true`
- Example:

```
language:
  sdf:
    enabled: false
```

parse-table

The relative path to the parse table (if not specified in the ESV).

- Default: `target/metaborg/sdf.tbl`
- Example:

```
language:
  sdf:
    parse-table: "tables/sdf.tbl"
```

completion-parse-table

The relative path to the completions parse table.

- Default: `target/metaborg/sdf-completions.tbl`
- Example:

```
language:
  sdf:
    completion-parse-table: "tables/sdf-completions.tbl"
```

version

Version of SDF to use.

- Format: Either `sdf2` or `sdf3`.
- Default: `sdf3`
- Example:

```
language:
  sdf:
    version: sdf2
```

sdf2table

Version of sdf2table to use.

- Format: Either `c`, `java`, or `dynamic`.
- Default: `c`
- Example:

```
language:
  sdf:
    sdf2table: java
```

jsglr-version

Version of the JGSLR parser to use. The options listed after `v2` are extensions of `v2`, [which are described here](#). Note that some of these extensions are experimental.

- Format: Either `v1`, `v2`, `data-dependent`, `incremental`, `layout-sensitive`, `recovery`, or `recovery-incremental`.
- Default: `v1`
- Example:

```
language:
  sdf:
    jsglr-version: v2
```

placeholder

Configuration for completion placeholders.

- Format: Quoted strings for prefix, and optionally, quoted strings for suffix.
- Default: prefix: `[]`, and suffix: `]]`
- Example:

```
language:
  sdf:
    placeholder:
      prefix: "$"
```

generate-namespaced

Configuration for generating a namespaced grammar. A namespaced grammar can be generated automatically from an SDF3 grammar. This namespacing is done by adding the language name to all module names and sort names. The generated grammar is put in `src-gen/syntax`.

- Format: Either `true` or `false`.
- Default: `false`
- Example:

```
language:
  sdf:
    generate-namespaced: true
```

externalDef

External SDF definition file to use. If this is specified, the `language.sdf.version` and `language.sdf.args` options are ignored, and all SDF2 or SDF3 syntax files are ignored.

- Example:

```
language:
  sdf:
    externalDef: ${path:root}/syntax/Stratego-Sugar.def
```

args

List of additional arguments that are passed to `pack-sdf` when this language specification is built.

- Format: List of command-line arguments.
- Default: None
- Example:

```
language:
  sdf:
    args:
      - -Idef
      - ${path:root}/lib/SDF.def
```

sdf-meta

List of SDF2 files that define a syntax mix of two or more languages, which are turned into extra parse tables when this language specification is built. These are typically used for mixing the grammar of the language with that of Stratego to be able to use concrete syntax of the language to describe abstract syntax in Stratego and transform it.

- Format: List of command-line arguments.
- Default: [Stratego-<language>.sdf]
- Example:

```
language:
  sdf:
    sdf-meta:
      - Stratego-Tiger.sdf
      - Stratego-Tiger-Java15.sdf
```

stratego

Configuration options for Stratego.

format

The output format of the `strj` compiler when this language specification is built.

- Format: Either `ctree` or `jar`.
- Default: `ctree`
- Example:

```
language:
  stratego:
    format: jar
```

args

List of additional arguments that are passed to `strj` when this language specification is built.

- Format: List of command-line arguments.
- Default: None
- Example:

```
language:
  stratego:
    args:
      - -la
      - stratego-lib
      - -la
      - stratego-sglr
      - -la
```

build

useBuildSystemSpec

Whether to use the specification from the build system as a source of configuration for that build system, or this configuration file.

For example, when set to `false` (the default), and Spoofox's Maven plugin's pomless support is enabled through the `.mvn/settings.xml` file, Maven will entirely ignore the contents of the `pom.xml` file, and use this configuration file as a source of information. When this is not desired, for example if your POM file has information that is not covered in this configuration file, set it to `true` to use the POM file. The `id`, `name`, and `dependencies` must be duplicated from this configuration file into the POM file, since this configuration file is ignored by Maven.

Note that Spoofox itself will use this configuration file regardless of this setting. This setting is only used by build systems such as Maven and Gradle.

- Format: Either `true` or `false`.
- Default: `false`
- Example:

```
build:
  useBuildSystemSpec: true
```

Additional build steps

The *build* configuration option also hosts a list of additional build steps.

- Format: List of build steps. There are 2 kinds of additional build steps which are described below. Each build step has a phase in which it is executed, which can be one of the following:
 - `initialize`: runs at the start of a build
 - `generateSources`: runs after compilers for all compile dependencies have generated source files
 - `compile`: runs after the build (i.e. `pack-sdf`, `strj`, etc. have been executed), but before compiling Java files
 - `pkg`: runs after Java files have been compiled, and after packaging the language component
 - `clean`: runs when the language specification is cleaned
- Default: None

stratego-cli (Stratego build step)

Build step that runs a command-line Stratego application.

- Format: phase, strategy to run, and command-line arguments
- Example:

```
build:
  stratego-cli:
    - phase: compile
      strategy: org.strategoxt.tools.main-parse-pp-table
      args:
        - -i
        - ${path:root}/lib/EditorService-pretty.pp
        - -o
        - ${path:root}/target/metaborg/EditorService-pretty.pp.af
```

ant (Ant build step)

Build step that runs a target from an Ant build script.

- Format: phase, path to Ant build script, and target in the build script to execute
- Example:

```
build:
  ant:
    - phase: initialize
      file: ${path:root}/build.xml
      target: main
```

18.1.3 Examples

Our meta-languages and meta-libraries have configuration files which can be used as examples:

- [ESV](#)
- [SDF2](#)
- [SDF3](#)
- [Stratego](#)
- [NaBL](#)
- [TS](#)
- [Analysis library](#)
- [NaBL2](#)
- [Analysis library 2](#)
- [DynSem](#)

18.2 Language Extension

This page describes how to do language extension through the Spoofox mechanism of exporting source files and importing them in the extension project.

18.2.1 Exporting source files in base language

Export all relevant source files through the metaborg.yaml file.

```
exports:
- language: ATerm
  directory: src-gen/syntax
- language: SDF
  directory: src-gen/syntax
  includes: "**/*.sdf"
- language: TemplateLang
  directory: syntax
  includes: "**/*.sdf3"
- language: Stratego-Sugar
  directory: trans
  includes: "**/*.str"
- language: Stratego-Sugar
  directory: src-gen
  includes: "**/*.str"
  excludes: "nabl2/**/*.str"
- language: Stratego-Sugar
```

(continues on next page)

(continued from previous page)

```

    directory: src-gen/nabl2
    includes: "**/*.str"
- language: EditorService
    directory: src-gen
    includes: "**/*.esv"
- language: EditorService
    directory: editor
    includes: "**/*.esv"

```

Note that you need to export all generated files from the meta languages as well (generated Stratego and SDF from SDF3 and NaBL2 in the above example).

Note that `src-gen/nabl2` needs to be its own export, as it is a stratego root directory.

Note that we export the SDF3 source files in order to be able to extend these in a language extension.

Suggestion: move all exported files into a sub-folder with the base-language name to avoid name clashes in the language extension.

18.2.2 Importing source files in language extension

Import the language in the yaml file:

```

dependencies:
  source:
    - org.metaborg.lang:icedust2:0.6.3-SNAPSHOT

```

Import all syntax by importing the top-level AST construct in SDF3:

```

module myextension

imports

  Modules //refers to icedust2/syntax/Modules.sdf3

context-free start-symbols

  Start

```

Make the pretty printer work properly by generating it with the base-language name (in `metaborg.yaml`):

```

language:
  sdf:
    pretty-print: icedust2

```

Import all stratego by importing the main stratego file of the base language:

```

module myextension

imports

  icedust2

```

18.2.3 Limitations

Currently, importing exported source files only works for Spoofax meta language files.

The following types of files need to be copy pasted currently:

- files that need to be exported in the final compiler (no re-exporting)
- native java strategies
- .tbl files (for parsing Stratego mix syntax)
- icon files

More info: <http://yellowgrass.org/issue/Spoofax/211>

Sunshine: the Spoofax Command-Line Interface

Sunshine is the command-line interface of Spoofax that allows you to load Spoofax languages and run tasks such as parsing, analysis and transformations from that language.

19.1 Installation

On any release page you can find the Sunshine JAR under *Command-line utilities*. This is an executable JAR with a command-line interface.

19.2 Usage

Sunshine gives the following options:

- Parse, to parse a single file and print the AST
- Analyze, to parse and analyze a single file and print the analyzed AST
- Transform, to parse, analyze and transform a single file and print the analyzed AST
- Build, to run a builder on a single file, which will parse, possibly analyze, transform and print the result

Use the `--help` flag for details.

19.3 Nailgun

If you use the command-line interface often or multiple times you may wish to speed up the process. Every time to run Sunshine, there is overhead of the JVM startup time and the loading time for the Spoofax language.

If you wish to speed this up you can use the `server` command of Sunshine. This starts up a Nailgun server. Install the *Nailgun client*, then use `ng sunshine --help` to see the commands available.

The commands on the server are slightly different from the normal commands as loading a language is a separate command now. Languages are still loaded lazily, so the first time a language is used it will take longer, but after that things should run noticeably faster.

This is the reference manual for the Spoofox Core API. The Spoofox Core API is a Java API for building and running languages made with Spoofox. To this end, the API exposes services for building languages, language processing services such as parsers, analyzers, and transformations, and editor services such as (info, warning, error) messages, syntax highlighting, reference resolution, code completion.

The introduction gives an overview of the concepts, architecture, and services of the API. The rest of the manual goes into more detail on these subjects.

20.1 Introduction

Spoofox Core, part of Spoofox 2.x, is a Java API for building languages and running languages made with the Spoofox Language Workbench, and a concrete implementation of that API. Spoofox Core is intended to be used by:

- Language designers wishing to embed their language into their application, so that their end-users can use their language.
- Researchers wishing to experiment with (meta-)languages, for example by running their languages in experiments or benchmarks.
- The Spoofox developers themselves, to create plugins for JVM platforms such as Maven, Eclipse, and IntelliJ, which can be used to specify, build, and run Spoofox languages.

The two main use cases of Spoofox Core are running languages and building languages. To give an overview of what is possible, here is a code fragment that uses the API to run a Spoofox language, by loading the language, parsing a file, and displaying an outline:

```
public class Main {
    public static void main(String[] args) throws Throwable {
        try (Spoofox spoofox = new Spoofox()) {
            FileObject languageDir = spoofox.resourceService.resolve(args[0]);
            ILanguageImpl language = spoofox.languageDiscoveryService.
↪ languageFromDirectory(languageDir)
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

FileObject fileToParse = spoofax.resourceService.resolve(args[1]);
String text = spoofax.sourceTextService.text(fileToParse);
ISpoofaxInputUnit input = spoofax.unitService.inputUnit(fileToParse, language,
↪null);
ISpoofaxParseUnit parseUnit = spoofax.syntaxService.parse(input);

IOutline outline = spoofax.outlineService.outline(parseUnit);
System.out.println(outline);
    }
}
}

```

And a code fragment to build a language that has been specified in the Spoofax Language Workbench:

```

public class Main {
    public static void main(String[] args) throws Throwable {
        try(SpoofaxMeta spoofax = new SpoofaxMeta()) {
            FileObject languageSpecificationDir = spoofax.resourceService.resolve(args[0]);
            IProject project = spoofax.parent.projectService.get(languageSpecificationDir);
            ISpoofaxLanguageSpec languageSpecification = spoofax.languageSpecService.
↪get(project);

            LanguageSpecBuildInput input = new
↪LanguageSpecBuildInput(languageSpecification);
            spoofax.metaBuilder.initialize(input);
            spoofax.metaBuilder.generateSources(input, null);
            spoofax.metaBuilder.compile(input);
            spoofax.metaBuilder.pkg(input);
            spoofax.metaBuilder.archive(input);
        }
    }
}

```

Spoofax Core has been designed and implemented with several goals in mind:

- Portability - the ability to build and run languages on multiple platforms
 - For external users, the Spoofax Core support running Spoofax languages on any platform on the JVM, allowing them to embed languages into their JVM application. For example, you could define a DSL in the Spoofax Language Workbench, and then have your users use the DSL by embedding it in your application.
 - For the developers of the Spoofax Language Workbench, it supports building and running languages on any JVM platform. This has allowed us to make plugins for the Maven, Eclipse, and IntelliJ platforms that can build and run any Spoofax language, and will allow us to expand this support to other platforms in the future.
- Maintainability
 - The API to Spoofax Core is separated from the implementation of Spoofax Core. This allows us to update the implementation without changing the API.
- Extensibility
 - Spoofax Core supports extensibility in several places, without having to change the source code.

The source code of Spoofax Core can be found in the [metaborg/spoofax repository on GitHub](https://github.com/metaborg/spoofax).

Next, we describe the architecture of Spoofax Core.

20.2 Architecture

20.2.1 Components

Spoofox Core consists of four components implemented as Java libraries:

- MetaBorg Core (org.metaborg.core)
- Spoofox Core (org.metaborg.spoofox.core)
- MetaBorg-meta Core (org.metaborg.meta.core)
- Spoofox-meta Core (org.metaborg.spoofox.meta.core)

The architecture of these components is as follows:

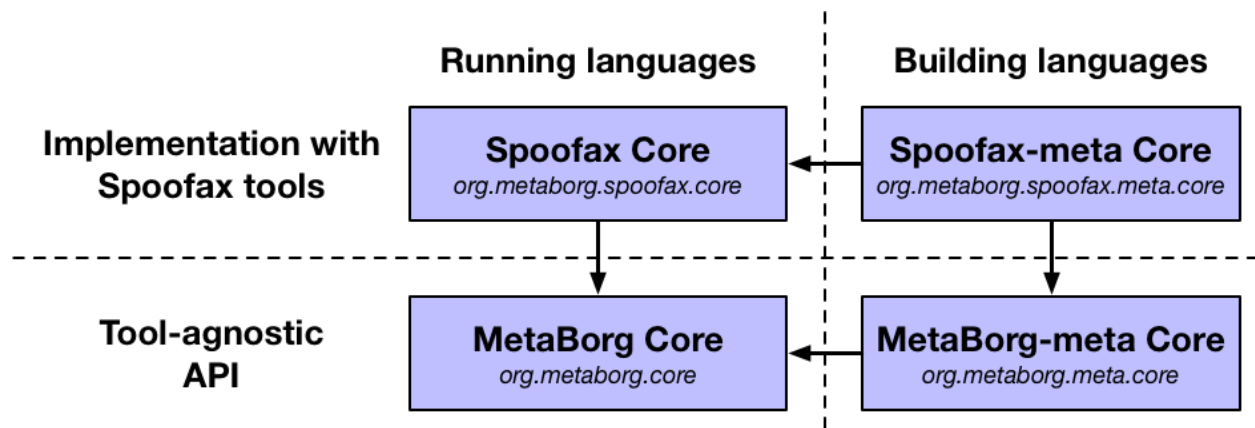


Fig. 1: Architecture with components identified by their name and artifact identifier, with dependencies on each other.

Spoofox Core can be separated into components that *run* languages (left), and *-meta* components that *build* languages (right), with dependencies flowing from right to left. This separation was made so that the functionality of running languages could be deployed without requiring the functionality of building languages. We think this is also a good separation of concerns, minimizing the responsibilities of each component.

The components can also be separated into *Spoofox* components that provide *concrete implementations* (top), and *MetaBorg* components that provide an *API* without tool-specific implementations (bottom), with dependencies flowing from top to bottom. This split separates the concerns of complex tool implementations, with the API that is required to run those tools. This enables generic implementations on top of the API, that do not depend on concrete tool implementations. For example, the concern of dynamically loading languages is implemented generically in `org.metaborg.core`.

20.2.2 Adapters and Plugins

Adapters and plugins adapt Spoofox(-meta) Core to other platforms. For example, the following figure shows a command-line (Cmd-line) adapter, Eclipse plugin, and a Maven plugin:

The command-line adapter provides a batch command-line interface to running languages in Spoofox Core. It only needs to depend on Spoofox Core, not Spoofox-meta Core, because it is only *running* languages, not building them. The Eclipse plugin runs Spoofox languages in the Eclipse IDE, enabling programs of Spoofox languages to be displayed in editors, with editor services such as error markers and syntax highlighting. The Eclipse meta plugin is a separate plugin for *building* Spoofox languages. With this architecture, the Eclipse meta plugin can also be deployed separately from the Eclipse plugin that just *runs* languages.

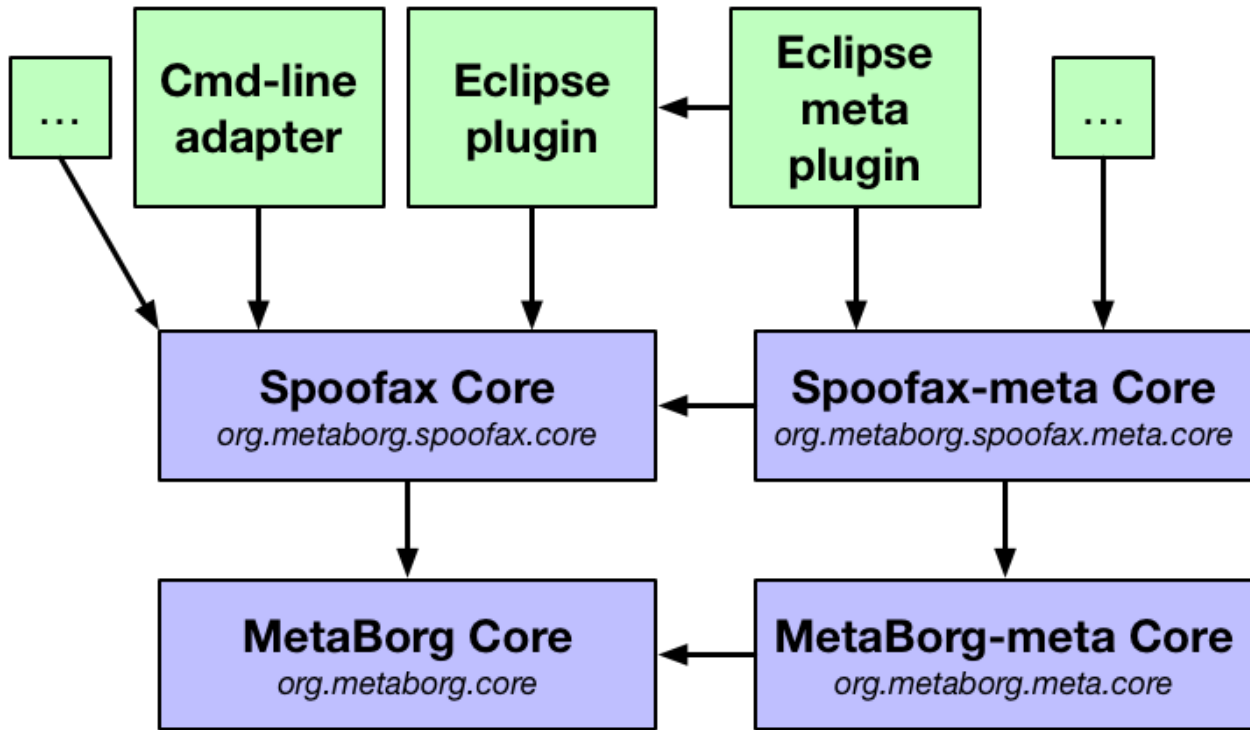


Fig. 2: Architecture with adapters and plugins based on Spoofax Core components.

We have developed the following adapters and plugins:

- [Sunshine](#) (command-line adapter)
- [Maven](#) plugin
- [Eclipse](#) plugin
- [IntelliJ](#) plugin

20.2.3 Dependencies

Components have several internal and external dependencies, where internal dependencies are developed as part of the Spoofax ecosystem, and external dependencies are third-party Java libraries. The following figure shows some of these dependencies:

For example, Spoofax Core depends on the Stratego runtime and JSGLR parser which are developed internally, whereas MetaBorg core only depends on basic third-party libraries like logging and a file system abstraction (VFS). The dependencies for each component are listed in their Maven `pom.xml` files:

- `org.metaborg.core`
- `org.metaborg.spoofax.core`
- `org.metaborg.meta.core`
- `org.metaborg.spoofax.meta.core`

Versions for most dependencies are managed in the `dependencyManagement` section of the [parent POM file](#). Dependencies with version `${metaborg-version}` denote internal dependencies. When embedding Spoofax Core into an application, both external and internal dependencies must be embedded.

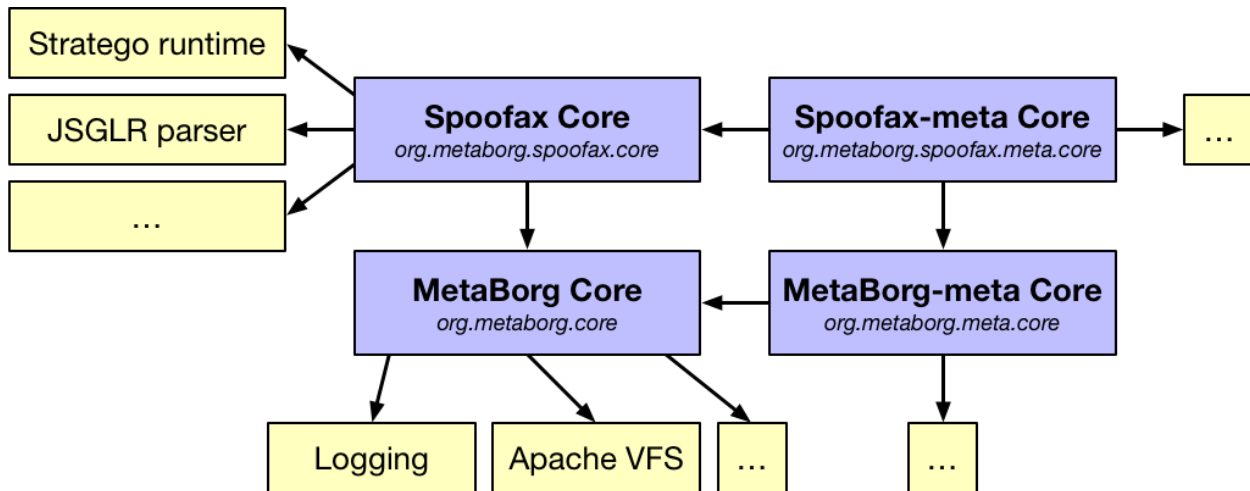


Fig. 3: Architecture including some external dependencies.

20.3 Concepts

Spoofox has several cross-cutting concepts that are used within Spoofox Core. This manual describes those concepts.

20.3.1 Languages

Spoofox runs and builds languages, so it has several concepts related to languages.

Example

We will explain these concepts by using the structure of a Java compiler as example in the following image:

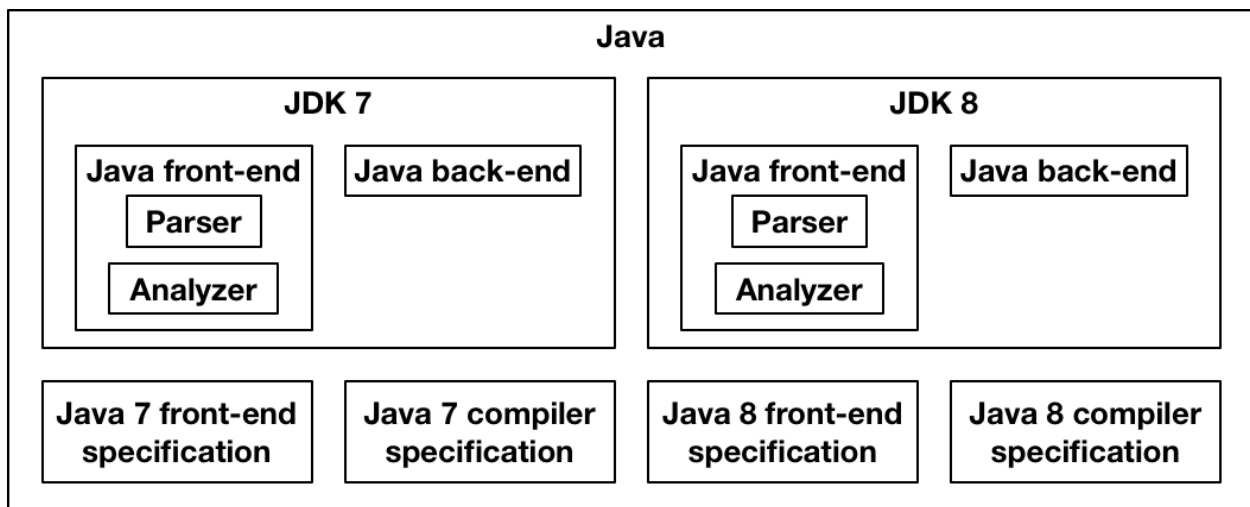


Fig. 4: Part of the internal structure of the JDK7 and 8 compilers.

A Java compiler has a *front-end* that *parses* and *analyzes* Java programs. The front-end is specified as a PDF or (hopefully) as an executable *specification*. It also has a *back-end* that compiles Java programs into Java bytecode,

with a specification. The front-end and back-end together compose a Java compiler, such as the *Java Development Kit version 7 (JDK7)*. Similarly, a front-end and back-end for Java 8 form *JDK8*. Together, JDK7 and JDK8 are part of the *Java* series of compilers, with JDK8 being the latest one.

Language Component and Facets

In Spoofox, the front-end and back-end are **language components** (`ILanguageComponent` in the source code). A language component implements a part of the language. They have an identifier and are versioned, which together form a unique identifier.

The functionality inside a language component, such as the parser and analyzer in the front-end, are **language facets** (`IFacet`). A language facet is an implementation of a facet of a language, that Spoofox knows how to execute.

Language Specifications

The specification of the front-end and back-end components are **language specifications** (`ISpoofaxLanguageSpec`). A language specification implements or specifies (in meta-languages) a part of the language. A single language specification is compiled by Spoofox into a single language component.

Dependencies

Language specifications have two types of dependencies. A **compiler dependency** (`compileDeps`) indicates that when a language specification is compiled, the compiler of a different language component should be executed first. For example, if JDK7 and JDK8 were specified in SDF, the specifications need a compiler dependency on the SDF compiler, to translate the SDF grammar into a parse table.

A **source dependency** (`sourceDeps`) indicates a dependency on source files of another language specification. For example, a JDK back-end needs to depend on the AST signatures of the front-end, to be able to pattern match parsed terms. We can also use source dependencies to depend on libraries. For example, a JDK front-end can depend on the NaBL analysis library, which contains reusable functionality for executing name analysis with NaBL.

Composing Language Implementations

The JDK7 and JDK8 compositions of their front and back-ends are **language implementations** (`ILanguageImpl`) in Spoofox. A language implementation is simply a collection of language components. Like language components, language implementations have an identifier and version, which together form a unique identifier.

Most of the user-facing Spoofox Core API works with language implementations, such that using a language does not require manual composition of components. The main exceptions to this are the compiler dependencies described earlier, and loading a language into Spoofox, which loads language components. The internals of Spoofox Core deal with the composition of language facets in language components of language implementations. For example, a language implementation with multiple components could provide multiple menu facets, which the menu service composes into one menu that is shown to the user.

Language Series

Finally, the Java series of compilers is called a **language** (`ILanguage`) in Spoofox. A language is a collection of language implementations. It does not have an identifier or version, just a name. Within a language, there is the concept of the **active language implementation** (`activeImpl`), which is the language implementation with the highest version. This is JDK8 in the example.

20.3.2 Projects

Projects in Spoofax are used to host language specifications and programs of a language.

An **end-user project** (`IProject`) is a project that contains programs of languages, intended to be developed by an end-user of those languages.

A **language specification project** (`ISpoofaxLanguageSpec`), sometimes called a language project, is a project that contains a languages specification. A language specification contains programs of meta-languages, intended to be developed by a language developer. A language specification project is a specialization of an end-user project.

20.4 Design

Todo: This section of the documentation has not been written yet

20.5 Dependency Injection

Todo: This section of the documentation has not been written yet

20.6 Services

This manual describes all services in Spoofax Core. A service is a class (and typically a singleton) that provides a single piece of functionality to the Spoofax Core system. Services can be accessed through facades, or injected through dependency injection.

Both MetaBorg, Spoofax, non-meta, and meta services are described. Each service has a corresponding interface which is the Java API to that service. Some services have specializations that specialize the interface in order to provide more specialized return types or additional method arguments.

20.6.1 MetaBorg services

Basic

Basic services provide low-level functionality.

Resource service

Interface `org.metaborg.core.resource.IResourceService`

Source text service

Interface `org.metaborg.core.source.ISourceTextService`

Language

Language services provide all language, language implementation, and language component related functionality.

Language service

Interface `org.metaborg.core.language.ILanguageService`

Language identifier service

Interface `org.metaborg.core.language.ILanguageIdentifierService`

Language discovery service

Interface `org.metaborg.core.language.ILanguageDiscoveryService`

Dialect service

Interface `org.metaborg.core.language.dialect.IDialectService`

Language paths service

Interface `org.metaborg.core.build.paths.ILanguagePathService`

Language dependency service

Interface `org.metaborg.core.build.dependency.IDependencyService`

Language processing

Language processing services provide services for the processing of files of a language.

Syntax service

Interface `org.metaborg.core.syntax.ISyntaxService`

Specialization `org.metaborg.spooifax.core.syntax.ISpooifaxSyntaxService`

Analysis service

Interface `org.metaborg.core.analysis.IAnalysisService`

Specialization `org.metaborg.spooifax.core.analysis.
ISpooifaxAnalysisService`

Transformation service

Interface `org.metaborg.core.transform.ITransformService`

Specialization `org.metaborg.spooifax.core.transform.
ISpooifaxTransformService`

Unit service

Interface `org.metaborg.core.unit.IUnitService`

Specialization `org.metaborg.spooifax.core.unit.ISpooifaxUnitService`

Input unit service

Sub-interface of the unit service that only provides methods for creating input units. Use this sub-interface when you only require creating input units for parsing.

Interface `org.metaborg.core.unit.IInputUnitService`

Specialization `org.metaborg.spooifax.core.unit.ISpooifaxInputUnitService`

Builder

Interface `org.metaborg.core.build.IBuilder`

Specialization `org.metaborg.spoofax.core.build.ISpoofaxBuilder`

Parse result processor

Interface `org.metaborg.core.processing.parse.IParseResultProcessor`

Specialization `org.metaborg.spoofax.core.processing.parse.
ISpoofaxParseResultProcessor`

Analysis result processor

Interface `org.metaborg.core.processing.analyze.IAnalysisResultProcessor`

Specialization `org.metaborg.spoofax.core.processing.analyze.
ISpoofaxAnalysisResultProcessor`

Context

Context services in MetaBorg provide a context for language processing tasks.

Project service

Interface `org.metaborg.core.project.IProjectService`

Specialization `org.metaborg.core.project.ISimpleProjectService`

Context service

Interface `org.metaborg.core.context.IContextService`

Editor services

Editor services provide functionality for source code editors.

Categorizer service

Interface `org.metaborg.core.style.ICategorizerService`

Specialization `org.metaborg.spoofax.core.style.ISpoofaxCategorizerService`

Styler service

Interface `org.metaborg.core.style.IStylerService`

Specialization `org.metaborg.spoofax.core.style.ISpoofaxStylerService`

Tracing service

Interface `org.metaborg.core.tracing.ITracingService`

Specialization `org.metaborg.spoofax.core.tracing.ISpoofaxTracingService`

Hover tooltip service

Interface `org.metaborg.core.tracing.IHoverService`

Specialization `org.metaborg.spoofax.core.tracing.ISpoofaxHoverService`

Reference resolution service

Interface `org.metaborg.core.tracing.IResolverService`

Specialization `org.metaborg.spoofax.core.tracing.ISpoofaxResolverService`

Outline service

Interface `org.metaborg.core.outline.IOutlineService`

Specialization `org.metaborg.spoofax.core.outline.ISpoofaxOutlineService`

Completion service

Interface `org.metaborg.core.completion.ICompletionService`

Specialization `org.metaborg.spoofax.core.completion.
ISpoofaxCompletionService`

Menu service

Interface `org.metaborg.core.menu.IMenuService`

Configuration

Configuration services provide read and write access to project and language component configuration at runtime.

See the *language development manual on configuration* for documentation about the Spoofox language specification configuration, which is a superset of the language specification, component, and project configuration.

Project configuration**Service**

Interface `org.metaborg.core.config.IProjectConfigService`

Builder

Interface `org.metaborg.core.config.IProjectConfigBuilder`

Writer

Interface `org.metaborg.core.config.IProjectConfigWriter`

Language component configuration**Service**

Interface `org.metaborg.core.config.ILanguageComponentConfigService`

Builder

Interface `org.metaborg.core.config.ILanguageComponentConfigBuilder`

Writer

Interface `org.metaborg.core.config.ILanguageComponentConfigWriter`

20.6.2 Spoofox services**Term factory service**

Interface `org.metaborg.spoofax.core.terms.ITermFactoryService`

Stratego runtime service

Interface `org.metaborg.spoofax.core.stratego.IStrategoRuntimeService`

Common Stratego functionality

Interface `org.metaborg.spoofax.core.stratego.IStrategoCommon`

20.6.3 MetaBorg-meta services

Project

Project services in MetaBorg-meta provide a context for language specification builds.

Language specification project service

Interface `org.metaborg.meta.core.project.ILanguageSpecService`

Specialization `org.metaborg.spoofax.meta.core.project.
ISpoofaxLanguageSpecService`

Configuration

Configuration services for language specifications. See the *language development manual on configuration* for documentation about the Spoofax language specification configuration.

Language specification configuration

Service

Interface `org.metaborg.meta.core.config.ILanguageSpecConfigService`

Specialization `org.metaborg.spoofax.meta.core.config.
ISpoofaxLanguageSpecConfigService`

Builder

Interface `org.metaborg.meta.core.config.ILanguageSpecConfigBuilder`

Specialization `org.metaborg.spoofax.meta.core.config.
ISpoofaxLanguageSpecConfigBuilder`

Writer

Interface `org.metaborg.meta.core.config.ILanguageSpecConfigWriter`

Specialization `org.metaborg.spoofax.meta.core.config.
ISpoofaxLanguageSpecConfigWriter`

20.6.4 Spoofax-meta services

Language specification builder

Interface `org.metaborg.spoofax.meta.core.build.LanguageSpecBuilder`

20.7 Language Processing

Spoofax, at its core, is a language processing framework. That is, a framework for reading, parsing, analyzing, transforming, and writing programs (files) of a language.

Spoofax provides a completely automated configurable language processing pipeline that works for most language. Spoofax also exposes API for manual language processing, in the case that the automated language processing pipeline does not do what you would like. We describe both ways of language processing in this manual.

20.7.1 Getting a project handle

All language processing happens inside an `IProject`, which is a handle for a project at a certain location, containing source files.

Projects are retrieved with the `IProjectService` service. However, different applications provide different implementations of this service, because they have different means of representing projects. For example, in Eclipse and IntelliJ, projects are top-level directories in the workspace. In Maven, a project is denoted by a directory which has a `pom.xml` file. Therefore, Eclipse, IntelliJ, and Maven provide special implementations of the project service. Creation and removal of projects in those environments is determined by the environment. Projects can be retrieved with the following code:

```
IProjectService projectService = ... // Get through dependency injection
IProject project = projectService.get(location);
if(project == null) {
    // Handle case where project does not exist.
}
```

In a command-line application, projects are typically passed as a command-line argument that points to a directory. Therefore, the simple project service implementation can be used, which allows the command-line application to manage creation and removal of projects. The simple project service is the default implementation, and can be accessed through the `ISimpleProjectService` interface by dependency injection. With the simple project service, projects can be created and removed in a command-line environment with the following code:

```
public class Main {
    public static void main(String[] args) {
        String projectDir = args[1]; // First command-line argument is the project_
        ↪directory.

        // Resolve project directory into a FileObject for use in Spoofax.
        IResourceService resourceService = ... // Get through dependency injection
        FileObject projectLoc = resourceService.resolve(projectDir);

        ISimpleProjectService projectService = ... // Get through dependency injection
        try {
            // Add a project
            IProject project = projectService.create(projectLoc);
        } catch (MetaborgException e) {
            // Handle project creation exception.
        }
    }
}
```

After a project is added, it can be retrieved again with the project service interface. Any files inside the project location (i.e. ancestor of project path) belong to the added project; retrieving the project for those files will return the added project.

20.7.2 Automatic language processing

Automated language processing is performed with the `ISpoofaxBuilder` service. It provides a `build` method for running the language processing pipeline, and a `clean` method for cleaning up generated files produced by the

pipeline.

The `build` method takes a progress reporter for reporting the progress of a build, a cancellation token to cancel a build, and importantly, a `BuildInput` object that specifies what to build and how to build it. A build input object is constructed with the `BuildInputBuilder` class which is a [fluent interface](#) for creating build inputs.

For example, to create a build input which parses, analyzes, and compiles all sources in a project, use the following code:

```
BuildInput input = new BuildInputBuilder(project)
    .withDefaultIncludePaths(true)
    .withSourcesFromDefaultSourceLocations(true)
    .withSelector(new SpoofoxIgnoresSelector())
    .addTransformGoal(new CompileGoal())
    .build(dependencyService, languagePathService)
    ;
```

There are several methods in the build input builder which allow customisation of the build input object, to customise the processing pipeline.

To run the language processing pipeline, pass the build input along with a progress reporter and cancellation token to the builder service:

```
ISpoofoxBuilder builder = ... // Get through dependency injection
ISpoofoxBuildOutput output = builder.build(input);
```

The result of building is a `ISpoofoxBuildOutput` object which denotes if the build was successful, and contains resource changes, parse, analysis, and transformation results, and any messages produced during building. It also includes the state of the build, which can be passed to the next build input to perform incremental processing.

20.7.3 Manual language processing

While automatic language processing provides an easy way for processing programs of a language, sometimes more control is needed. Therefore, we expose the parsing, analysis, and transformation API, to allow custom language processing pipelines.

Units

The processing pipeline works with the concept of units. A unit is a collection of information, about a certain processing aspect, for a single resource, of a certain language. For example, a parse unit contains the parsed AST for a resource, or a collection of error messages if parsing that resource, and is specific to the language that it is parsed with.

In most cases, it is not required to manually construct processing units, since the parse, analyze, and transform services create these units for you. The only unit that must always be created, is the `ISpoofoxInputUnit`, which contains all information to parse a resource. Such a unit can be constructed with the `ISpoofoxInputUnitService` service, for example:

```
FileObject source = ... // Source file to parse
ILanguageImpl language = ... // Language of the source file
// Get contents of the source file
ISourceTextService sourceTextService = ... // Get through dependency injection
String contents = sourceTextService.text(source);
// Create an input unit for the source file
ISpoofoxInputUnitService unitService = ... // Get through dependency injection
ISpoofoxInputUnit inputUnit = unitService.inputUnit(source, contents, language, null);
```

If construction of other units is required, the `ISpooifaxUnitService` service must be used. For example, the following code creates an `ISpooifaxParseUnit` from a custom AST:

```
IStrategoTerm customAST = ... // Custom AST made by the developer
// Create a parse unit using the custom AST
ISpooifaxUnitService unitService = ... // Get through dependency injection
ISpooifaxParseUnit parseUnit = unitService.parseUnit(inputUnit, new ParseContrib(true,
↳ true,
    customAST, Iterables2.<IMessage>empty(), -1));
```

Parsing

The `ISpooifaxSyntaxService` service parse input units into parse units. Parsing can be configured by customizing the input unit. The resulting parse unit contains the parsed AST, any messages produced during parsing, and the duration of parsing.

Analysis

The `ISpooifaxAnalysisService` service parses parse units into analysis results. An analysis result contains an analyze unit, which contains the actual unit produced by analysis, and updates, which contain updates for existing analyze units. Updates are only produced in subsequent calls to the analysis service, to support incremental updates to units.

To be able to analyze something, a `IContext` object is required. A context stores project and language specific information about analysis. A context is retrieved using the `IContextService` service, by calling the `get` method with the resource that you'd like to analyze, its project, and the language of that resource. When performing analysis, the context must be write-locked through the `IContext.write()` method, to ensure that only one thread is writing to the context at any given time.

For example, to analyze a parsed resource:

```
IProject project = ... // Project of the source file
ISpooifaxParseUnit parseUnit = ... // Parsed source file
// Get a context for the parsed source file
IContextService contextService = ... // Get through dependency injection
IContext context = contextService.get(parseUnit.source(), project,
    parseUnit.input().langImpl());
// Analyze the parsed source file
ISpooifaxAnalysisService analysisService = ... // Get through dependency injection
ISpooifaxAnalyzeResult result;
try(IClosableLock lock = context.write()) {
    result = analysisService.analyze(parseUnit, context);
}
ISpooifaxAnalyzeUnit analyzeUnit = result.result();
```

Transformation

The `ISpooifaxTransformService` service transforms parse or analyze units into transform units.

Since there are multiple transformations to choose from, a `ITransformGoal` object is required to choose which transformation to run. There are three transform goals:

- `CompileGoal` which selects the compiler (on-save handler) transformation.
- `NamedGoal` which selects a named builder. A list of names is required to find an action in nested menus.

- `EndNamedGoal` which selects a named builder based on the name of the builder only, ignoring any menus.

No service is needed to instantiate a transform goal, just instantiate one of the goals manually.

To transform a parse or analyze unit, call one of the `transform` methods. A context object is required for transforming. See the section on analysis on how to retrieve a context object. When performing transformations that read analysis data from the context, the context must be read-locked through the `IContext.read()` method, to ensure that no other thread is writing to the context. Since a language implementation can contain multiple transformations for the same goal, executing a transformation can return multiple transform units.

The following example transforms an analyzed resource with the `Compile to Java` transformation:

```
IAalyzeUnit analyzeUnit = ...
IContext context = analyzeUnit.context();
ISpoofaxTransformService transformService = ... // Get through dependency injection
Collection<ISpoofaxTransformUnit<ISpoofaxAnalyzeUnit>> transformUnits;
try(IClosableLock lock = context.read()) {
    transformUnits = transformService.transform(analyzeUnit, context,
        new EndNamedGoal("Compile to Java"));
}
```

Stratego Transformation

The transform service abstracts over the fact that Stratego is perform transformations, by executing transformations through goals. However, sometimes it may still be necessary to call Stratego strategies directly. Therefore, we expose the `IStrategoCommon` class.

The `invoke` methods execute a strategy on a term, and return the transformed term. A context object is required for transformation. See the section on analysis on how to retrieve a context object. When performing transformations that read analysis data from the context, the context must be read-locked through the `IContext.read()` method, to ensure that no other thread is writing to the context. For example, to invoke a strategy on a parsed AST:

```
ISpoofaxParseUnit parseUnit = ...
IContext context = ...
IStrategoCommon strategoCommon = ... // Get through dependency injection
IStrategoTerm transformed;
try(IClosableLock lock = context.read()) {
    transformed = strategoCommon.invoke(parseUnit.input().langImpl(),
        context, parseUnit.ast(), "compile-to-java");
}
```

The `toString` and `prettyPrint` methods of the `IStrategoCommon` class can be used to turn terms into string representations.

Internally, the `IStrategoRuntimeService` service is used, but this has a lower-level interface.

20.8 Core Extensions

Spoofax Core can be extended by providing additional `Guice Modules` with extensions at startup. Additional modules can be hardcoded when creating a Spoofax facade object, or by specifying the module as a plugin. Both Spoofax and meta-Spoofax can be extended with additional modules.

This manual describes the extension points in Spoofax Core, and how additional modules can be specified.

20.8.1 Extension points

Extension points in Spoofax Core are essentially the [Guice multibindings](#) that are being used in Spoofax Core. There are 2 kinds of extension points; `Multibinder` for a *Set* of implementations for a single interface, and `MapBinder` for a *Map* of implementations for a single interface. Guice merges all multibindings from all modules together.

Note: Extension points in Spoofax Core are not to be confused with Eclipse extension points, which are Eclipse-specific.

To add a singleton implementation to a Multibinding for an interface, use the following code inside the configure method of an `AbstractModule`:

```
Multibinder
    .newSetBinder(binder(), interface-class)
    .addBinding()
    .to(implementation-class)
    .in(Singleton.class);
```

For map bindings, use:

```
MapBinder
    .newMapBinder(binder(), key-class, interface-class)
    .addBinding(key)
    .to(implementation-class)
    .in(Singleton.class);
```

These examples use [linked bindings](#) with a singleton scope, but you can use any Guice binding logic after the `addBinding` part.

Some multibindings are annotated with an annotation, they are created by passing the annotation as a third parameter:

```
Multibinder.newSetBinder(binder(), interface-class, annotation)

MapBinder.newSetBinder(binder(), key-class, interface-class, annotation)
```

We describe the extension points for each component in this section. These extension points can also be found in the Guice Modules of the Spoofax Core source code.

MetaBorg extension points

Resource cleanup

Provides a means to clean up resources when the MetaBorg or Spoofax Core API is closed.

Signature `Multibinder<AutoCloseable>`

Interface `AutoCloseable`

Language cache cleanup

Provides a means to clean up cached language component or language implementation resources when a language component or language implementation is reloaded or removed.

Signature `Multibinder<ILanguageCache>`

Interface `ILanguageCache`

Context factory

Interface for creating `IContext` instances, linked to an identifier. A language specification uses a specific context factory by specifying the context type in ESV.

Signature MapBinder<String, IContextFactory>

Interface IContextFactory

Context strategy

Interface for IContext creation/retrieval strategies, linked to an identifier. A language specification uses a specific context strategy by specifying the context strategy in ESV.

Signature MapBinder<String, IContextStrategy>

Interface IContextStrategy

Language path provider

Provides source and include paths for languages.

Signature Multibinder<ILanguagePathProvider>

Interface ILanguagePathProvider

Spoofox extension points

Parser

Parser implementation, linked to an identifier. A language specification uses a specific parser by specifying the parser in ESV. An implementation **must** implement ISpoofoxParser and be bound to **both** signatures listed below for correct operation.

Signature MapBinder<String, IParser<ISpoofoxInputUnit, ISpoofoxParseUnit>>

Signature MapBinder<String, ISpoofoxParser>

Interface IParser

Interface ISpoofoxParser

Analyzer

Analyzer implementation, linked to an identifier. A language specification uses a specific analyzer by specifying the analyzer in ESV. An implementation **must** implement ISpoofoxAnalyzer and be bound to **both** signatures listed below for correct operation.

Signature MapBinder<String, IAnalyzer<ISpoofoxParseUnit, ISpoofoxAnalyzeUnit, ISpoofoxAnalyzeUnitUpdate>>

Signature MapBinder<String, ISpoofoxAnalyzer>

Interface IAnalyzer

Interface ISpoofoxAnalyzer

MetaBorg-meta extension points

Meta-resource cleanup

Provides a means to clean up resources when the MetaBorg-meta or Spoofox-meta Core API is closed. Requires the Meta annotation class, for example:

```
Multibinder.newSetBinder(binder(), AutoCloseable.class, Meta.class)
```

Signature Multibinder<AutoCloseable>

Annotation Meta.class

Interface `AutoCloseable`

Spoofax-meta extension points

Build steps

Build step implementation which can be executed during language specification builds.

Signature `Multibinder<IBuildStep>`

Interface `IBuildStep`

20.8.2 Hardcoding additional modules

Additional modules can be hardcoded when you control the application that you'd like to extend.

To extend Spoofax with additional hardcoded modules, add them when creating a `Spoofax` facade object:

```
final Spoofax spoofax = new Spoofax(new CustomModule(), new OtherCustomModule());
```

Similarly, to extend meta-Spoofax, add modules to the meta-facade `SpoofaxMeta`:

```
final SpoofaxMeta spoofaxMeta = new SpoofaxMeta(spoofax, new CustomMetaModule(),
    new OtherCustomMetaModule());
```

20.8.3 Plugin modules

When you do not control the application you'd like to extend, or if you'd like to extend **all** applications that use Spoofax Core, modules will need to be specified as plugins. Modules can be loaded as plugins through Java service providers for regular Java applications, and through Eclipse extensions for Eclipse plugins.

Java service provider

Java service providers are the standard solution for creating extensible applications on the JVM. Spoofax Core supports specifying additional modules as plugins through a service provider. To register your module as a plugin, [register it as a service provider](#) for the `IServiceModulePlugin` class. For example, if you would like to register the `org.example.CustomModule` and `org.example.OtherCustomModule` module:

1. Create a class implementing `IServiceModulePlugin`:

```
public class org.example.ExtensionModulePlugin implements
↳ IServiceModulePlugin {
    @Override public Iterable<Module> modules() {
        return Iterables2.<Module>from(new org.example.CustomModule(),
            new org.example.OtherCustomModule());
    }
}
```

2. Create the `src/main/resources/META-INF/services/org.metaborg.core.plugin.IServiceModulePlugin` file.
3. Add `org.example.ExtensionModulePlugin` to that file.

Whenever your JAR file is on the classpath together with Spoofax Core, Spoofax Core will pick up the module plugins and load them whenever the Spoofax facade is instantiated.

Similarly, for additional meta-modules, register a service provider for the `IServiceMetaModulePlugin` class:

1. Create a class implementing `IServiceMetaModulePlugin`:

```
public class org.example.ExtensionMetaModulePlugin implements
↳ IServiceMetaModulePlugin {
    @Override public Iterable<Module> modules() {
        return Iterables2.<Module>from(new org.example.CustomMetaModule(),
            new org.example.OtherCustomMetaModule());
    }
}
```

2. Create the `src/main/resources/META-INF/services/org.metaborg.core.plugin.IServiceMetaModulePlugin` file.
3. Add `org.example.ExtensionMetaModulePlugin` to that file.

Eclipse extension

Eclipse does not support Java service providers. To get your module plugins working in Eclipse, they need to be specified as an extension in the `plugin.xml` file.

Add the module classes with the `org.metaborg.spoofax.eclipse.module` extension point. For example:

```
<extension point="org.metaborg.spoofax.eclipse.module">
    <module class="org.example.CustomModule" />
    <module class="org.example.OtherCustomModule" />
</extension>
```

For meta-modules, use the `org.metaborg.spoofax.eclipse.meta.module` extension point. For example:

```
<extension point="org.metaborg.spoofax.eclipse.meta.module">
    <module class="org.example.CustomMetaModule" />
    <module class="org.example.OtherCustomMetaModule" />
</extension>
```

The `CustomModule` and `CustomMetaModule` classes in these examples must implement the `Module` class.

IntelliJ IDEA extension

Java service providers are only supported when building a project using the JPS plugin. For the IDE, you need to depend on the IntelliJ plugin and provide an implementation for the `org.metaborg.intellij.spoofaxPlugin` extension point. For example:

```
<extensions defaultExtensionNs="org.metaborg.intellij">
    <spoofaxPlugin implementation="org.example.CustomPlugin" />
    <spoofaxPlugin implementation="org.example.OtherCustomPlugin" />
</extensions>
```

For meta-modules, use the `org.metaborg.intellij.spoofaxMetaPlugin` extension point. For example:

```
<extensions defaultExtensionNs="org.metaborg.intellij">
    <spoofaxMetaPlugin implementation="org.example.CustomMetaPlugin" />
    <spoofaxMetaPlugin implementation="org.example.OtherCustomMetaPlugin" />
</extensions>
```

The `CustomPlugin` and `CustomMetaPlugin` classes in these examples must implement the `IServiceModulePlugin` and `IServiceMetaModulePlugin` interfaces respectively.

This is the reference manual for building and developing Spoofax, as well as information about its internals.

21.1 Introduction

Spoofax is the integration of many different tools, compilers, (meta-)languages, (meta-)libraries, and runtime components. This integration is made concrete in the [spoofax-releng](#) Git repository on GitHub. This repository contains all components via [Git submodules](#), which are updated by our [build farm](#) that builds Spoofax whenever one of its components in a submodule changes.

Spoofax currently contains the following subcomponents as submodules:

- [releng](#) - release engineering scripts for managing and building the `spoofax-releng` repostory.
- Java libraries and runtimes
 - [mb-rep](#) - libraries for program representation such as abstract terms
 - [mb-exec](#) - Stratego interpreter and utilities
 - [jsglr](#) - JSGLR parser
 - [spoofax](#) - Spoofax Core, a cross platform API to Spoofax languages
 - [spoofax-maven](#) - Maven integration for Spoofax Core
 - [spoofax-sunshine](#) - Command-line integration for Spoofax Core
 - [spoofax-eclipse](#) - Eclipse plugin for Spoofax Core
 - [spoofax-intellij](#) - IntelliJ plugin for Spoofax Core
- Meta-languages and libraries
 - [esv](#) - Editor service language
 - [sdf](#) - Syntax Definition Formalisms, containing the SDF2 and SDF 3 languages
 - [stratego](#) and [strategox](#) - Stratego compiler, runtime, and editor

- `nabl` - Name binding languages, containing the NaBL and NaBL2 languages, and support libraries for NaBL2
- `ts` - Type system language
- `dynsem` - Dynamic semantics language
- `metaborg-coq` - Coq signatures and syntax definition
- `spt` - Spoofox testing language
- `runtime-libraries` - NaBL support libraries, incremental task engine for incremental name and type analysis

Furthermore, this repository contains a Bash script `./b` that redirects into the Python release engineering scripts in the `releng` submodule. These scripts support managing this Git repository, version management, generation of Eclipse instances, building Spoofox, and releasing new versions of Spoofox.

The rest of this manual describes how to set up Maven and other required tools for building and developing Spoofox, how to build and develop Spoofox, how to write this documentation, and explains some of the internals of Spoofox components.

21.2 Requirements

Spoofax can be run on macOS, Linux, and Windows. Building is directly supported on macOS and Linux. Building on Windows is supported through the [Windows Subsystem for Linux \(Bash on Windows\)](#).

The following tools are required to build and develop Spoofox:

Git 1.8.2 or higher Required to check out the source code from our GitHub repositories. Instructions on how to install Git for your platform can be found here: <http://git-scm.com/downloads>.

If you run macOS and have [Homebrew](#) installed, you can install Git by executing `brew install git`. Confirm your Git installation by executing `git version`.

Java JDK 8 or higher Required to build and run Java components. The latest JDK can be downloaded and installed from: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.

On macOS, it can be a bit tricky to use the installed JDK, because Apple by default installs JRE 6. To check which version of Java you are running, execute the `java -version` command. If this tells you that the Java version is 1.8, everything is fine. If not, the Java version can be set with a command. After you have installed JDK, execute:

```
export JAVA_HOME="/usr/libexec/java_home -v 1.8"
```

Note: Such an export is not permanent. To make it permanent, add that line to `~/.bashrc` or equivalent for your OS/shell (create the file if it does not exist), which will execute it whenever a new shell is opened.

Confirm your JDK installation and version by executing `java -version` and `javac -version`.

Python 3.4 or higher Python scripts are used to orchestrate the build. Instructions on how to install Python for your platform can be found here: <https://www.python.org/downloads/>.

If you run macOS and have [Homebrew](#) installed, you can install Python by executing `brew install python3`. Confirm your Python installation by executing `python3 --version` or `python --version`, depending on how your package manager sets up Python.

During the build, Pip will install some Python dependencies into a virtual environment. No extra Python dependencies are required for this (with one small exception, see the note below). The latest version of Pip will automatically be installed inside the virtual environment.

Note: Debian and derivatives (like Ubuntu) do not include the full standard library when installing Python ([bug 1290847](#)), so you will need to install `python3-venv` to make sure the virtual environment can be created.

Maven 3.5.4 or higher (except Maven 3.6.1 and 3.6.2) Required to build most components of Spoofax. Our Maven artifact server must also be registered with Maven since the build depends on artifacts from previous builds for bootstrapping purposes. We explain how to install and set up Maven in the [next section](#).

Note: Spoofax cannot be built using Maven 3.6.1 or 3.6.2 due to these bugs: <https://issues.apache.org/jira/browse/MNG-6642> and <https://issues.apache.org/jira/browse/MNG-6765>

Maven Daemon 0.6.0 or higher Required run a full build of Spoofax with some speedup. This is a separate project and is not installed with Maven. Install as instructed here: <https://github.com/mvndaemon/mvnd#how-to-install-mvnd>.

Coreutils Required on macOS to be able to run the `sdf2table` and `implodePT` legacy binaries. On macOS with [Homebrew](#) installed, you can install them by running `brew install coreutils`.

Docker Required on macOS Catalina, Big Sur, and newer to be able to run the `sdf2table` and `implodePT` legacy binaries. Install it though the [Docker for Mac](#) website.

21.3 Maven

Maven is a project management and build tool for software projects. Most components in Spoofax are built with Maven.

21.3.1 Installing

Maven can be downloaded and installed from <https://maven.apache.org/download.cgi>. We require Maven 3.5.4 or higher. On macOS, Maven can be easily installed with Homebrew by executing `brew install maven`.

Confirm the installation and version by running `mvn --version`.

21.3.2 Memory allocation

By default, Maven does not assign a lot of memory to the JVM that it runs in, which may lead to out of memory exceptions during builds. To increase the allocated memory, execute before building:

```
export MAVEN_OPTS="-Xms512m -Xmx1024m -Xss16m -XX:MaxPermSize=512m"
```

Note: Such an export is not permanent, see previous note about making this permanent.

Note: `-XX:MaxPermSize=512m` is not required for Java 8, and even gives a warning when added.

21.3.3 Proxy settings

If you are behind a proxy, please put the proxy settings in your `~/.m2/settings.xml`. When you use the `b` script to build Spoofox, the `MAVEN_OPTS` environment variable is overridden to ensure the memory options above are supplied, so using commandline options in the environment variable for the proxy settings does not work.

21.3.4 Spoofox Maven artifacts

Spoofox's Maven artifacts are hosted on our artifact server: <https://artifacts.metaborg.org>. To use these artifacts, repositories have to be added to your Maven configuration. This configuration is *required* when building and developing Spoofox. Repositories can be added to your local Maven settings file (which is recommended), or to a project's POM file.

Local settings file

The recommended approach is to add repositories to your local Maven settings file, located at `~/.m2/settings.xml`. If you have not created this file yet, or want to completely replace it, simply create it with the following content:

```
<?xml version="1.0" ?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/
↪2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/
↪xsd/settings-1.0.0.xsd">
  <profiles>
    <profile>
      <id>add-metaborg-release-repos</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <repositories>
        <repository>
          <id>metaborg-release-repo</id>
          <url>https://artifacts.metaborg.org/content/repositories/releases/</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>metaborg-release-repo</id>
          <url>https://artifacts.metaborg.org/content/repositories/releases/</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
```

(continues on next page)

(continued from previous page)

```
<profile>
  <id>add-metaborg-snapshot-repos</id>
  <activation>
    <activeByDefault>true</activeByDefault>
  </activation>
  <repositories>
    <repository>
      <id>metaborg-snapshot-repo</id>
      <url>https://artifacts.metaborg.org/content/repositories/snapshots/</url>
      <releases>
        <enabled>false</enabled>
      </releases>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>metaborg-snapshot-repo</id>
      <url>https://artifacts.metaborg.org/content/repositories/snapshots/</url>
      <releases>
        <enabled>false</enabled>
      </releases>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profile>
</profiles>
</settings>
```

If you’ve already created a settings file before and want to add the repositories, just add the `profile` element (and the `profiles` element if it does not exist yet) to the settings file.

Advanced: project POM file

Repositories can also be added directly to a project’s POM file, which only set the repositories for that particular project. This is not recommended, because it makes repositories harder to change by users, and duplicates the configuration. But it can be convenient, because it does not require an external settings file.

To do this, just add the the following content to the POM file:

```
<repositories>
  <repository>
    <id>metaborg-release-repo</id>
    <url>https://artifacts.metaborg.org/content/repositories/releases/</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
  <repository>
```

(continues on next page)

(continued from previous page)

```

    <id>metaborg-snapshot-repo</id>
    <url>https://artifacts.metaborg.org/content/repositories/snapshots/</url>
    <releases>
      <enabled>>false</enabled>
    </releases>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>

<pluginRepositories>
  <pluginRepository>
    <id>metaborg-release-repo</id>
    <url>https://artifacts.metaborg.org/content/repositories/releases/</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>metaborg-snapshot-repo</id>
    <url>https://artifacts.metaborg.org/content/repositories/snapshots/</url>
    <releases>
      <enabled>>false</enabled>
    </releases>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>

```

21.3.5 Maven central repository mirror

Artifacts of most open source projects are hosted on the [Central Repository](#) server. If you are building any project using Maven, many artifacts will be downloaded from that server. While it is a fast server, it can still take a while to download all required artifacts for big projects.

If you are on the TUDelft network, you can use our local mirror of the Central Repository to speed things up. Using the mirroring requires a change in your local settings.xml file located at ~/.m2/settings.xml. If this file does not exist, create it with the following content:

```

<?xml version="1.0" ?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/
↪2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/
↪xsd/settings-1.0.0.xsd">
  <mirrors>
    <mirror>
      <id>metaborg-central-mirror</id>
      <url>https://artifacts.metaborg.org/content/repositories/central/</url>
      <mirrorOf>central</mirrorOf>
    </mirror>
  </mirrors>

```

(continues on next page)

(continued from previous page)

```
</mirrors>  
</settings>
```

If you've already created a settings file before and want to add the mirror configuration, just add the `mirror` element (and the `mirrors` element if it does not exist yet) to the settings file.

21.4 Building

This section describes how to build Spoofax from scratch, on the command-line.

21.4.1 Cloning the source code

Clone the source code from the [spoofax-releng](https://github.com/metaborg/spoofax-releng) repository with the following commands:

macOS, Linux, Windows

```
git clone --recursive https://github.com/metaborg/spoofax-releng.git  
cd spoofax-releng
```

Cloning and updating submodules can take a while, since we have many submodules and some have a large history.

Note: Windows

Additionally, only on Windows, you have to do the following:

```
cd releng\releng  
py -m pip install -r .\requirements.txt
```

Note: macOS Catalina, Big Sur, or newer

You have to install `coreutils` and `docker` to be able to build Spoofax. This is temporarily, until the 32-bit binaries for `sdf2table` and `implodePT` have been phased out.

- Download [Docker Desktop for Mac](#).
- Install `coreutils` through [Homebrew](#): `brew install coreutils`

21.4.2 Start a build

To build Spoofax, simply execute:

macOS, Linux

```
./b build all
```

Windows

```
.\bd.bat build all
```

This downloads the latest Stratego/XT, and builds Spoofox. If you also want to build Stratego/XT from scratch, execute:

macOS, Linux

```
./b build -st all
```

Windows

```
.\bd.bat build -st all
```

The `-s` flag build Stratego/XT instead of downloading it, and `-t` skips the Stratego/XT tests since they are very lengthy. The `all` part of the command indicates that we want to build all components. If you would only like to build the Java components of Spoofox, and skip the Eclipse plugins, execute:

In Windows, type `.\bd.bat` instead of `./b` in the following commands.

macOS, Linux

```
./b build java
```

Use `./b build` to get a list of components available for building, and `./b build --help` for help on all the command-line flags and switches.

Note: If you have opened a project in the repository in Eclipse, you **must turn off** *Project* → *Build Automatically* in Eclipse, otherwise the Maven and Eclipse compilers will interfere and possibly fail the build. After the Maven build is finished, enable *Build Automatically* again.

21.4.3 Updating the source code

If you want to update the repository and submodules, execute:

macOS, Linux

```
git pull --rebase
./b checkout
./b update
```

The `git pull` command will update any changes in the main repository. The `./b checkout` command will check out the correct branches in all submodules, because Git does not do this automatically. The `./b update` command will update all submodules.

21.4.4 Switching to a different branch

Switching to a different branch, for example the `spoofox-release` branch, is done with the following commands:

macOS, Linux

```
git checkout spoofax-release
git pull --rebase
git submodule update --init --remote --recursive
./b checkout
./b update
```


21.4.5 Troubleshooting

Resetting and cleaning

If updating or checking out a branch of submodule fails (because of unstaged or conflicting changes), you can try to resolve it yourself, or you can reset and clean everything. Reset and clean all submodules using:

macOS, Linux

```
./b reset  
./b clean
```

Warning: Resetting and cleaning DELETES UNCOMMITTED AND UNPUSHED CHANGES, which can cause PERMANENT DATA LOSS. Make sure all your changes are committed and pushed!

Weird compilation errors

If you get any weird compilation errors during the build, make sure that Project → Build Automatically is turned off in Eclipse.

21.5 Developing

If you are developing a project that is included in Spoofax it is recommended to set up a development environment. This section describes how to set up such a development environment.

21.5.1 Requirements

A working **Spoofax build** is required before being able to develop. You must be able to successfully build Spoofax by running *./b build all*. Do not continue if this does not work. Follow the [previous section](#) for instructions on how to build Spoofax.

21.5.2 Eclipse

Currently, an Eclipse development environment is the most supported environment. An Eclipse development environment can be generated with our scripts.

Generating an Eclipse instance

The *./b* script in the spoofax-releng repository can generate an Eclipse installation for you. Change directory into the spoofax-releng repository and run:

```
./b gen-spoofax -l -d ~/eclipse/spoofax-dev
```

This will download and install Eclipse into *~/eclipse/spoofax-dev* with the right plugins and *eclipse.ini* for Spoofax development. The locally built version of the Spoofax plugin will be installed into that Eclipse. Generating an Eclipse installation can take several minutes. After it's done generating, open the Eclipse installation and confirm that it works by creating a Spoofax project.

Note: If you get an error `Installation failed. Cannot complete the install because of a conflicting dependency.`, then make sure there is not an existing Eclipse instance at the destination.

Note: macOS: If upon starting Eclipse you get the error *To open “Eclipse” you need to install the legacy Java SE 6 runtime*, then you should install a Java JDK 6 or newer for Eclipse to use. If you installed one through [SDKMAN!](#) then you have to point Eclipse to it. To do this, edit the `Contents/Eclipse/eclipse.ini` file in the Eclipse application package content. Add the following lines just before the `-vmargs` argument, where `<USERNAME>` is your username:

```
-vm  
/Users/<USERNAME>/sdkman/candidates/java/current/jre/lib/jli/libjli.dylib
```

Fixing Eclipse settings

Some Eclipse settings unfortunately have sub-optimal defaults. Go to the Eclipse preferences and set these options:

- *General*
 - Enable: Keep next/previous editor, view and perspectives dialog open
- *General* → *Startup and Shutdown*
 - Enable: Refresh workspace on startup
- *General* → *Workspace*
 - Enable: Refresh using native hooks or polling
- *Maven*
 - Enable: Do not automatically update dependencies from remote repositories
 - Enable: Download Artifact Sources
 - Enable: Download Artifact JavaDoc
- *Maven* → *Annotation Processing*
 - Enable: Automatically configure JDT APT
- *Maven* → *User Interface*
 - Enable: Open XML page in the POM editor by default
- *Run/Debug* → *Launching*
 - Disable: Build (if required) before launching

Developing

Import the projects you’d like to develop. To import Java and language projects, use *Import* → *Maven* → *Existing Maven Projects*. Eclipse plugins are still imported with *Import* → *General* → *Existing Projects into Workspace*.

Running

To test your changes in the Spoofax Eclipse plugin, import the `org.metaborg.spoofax.eclipse` project from the `spoofax-eclipse` repository, which provides launch configurations for starting new Eclipse instances (a “guest” Eclipse). Press the little down arrow next to the bug icon (next to the play icon) and choose `Spoofax Eclipse Plugin` to start a new Eclipse instance that contains your changes. If it is not in the list of recently used configurations, click `Debug configurations...`, it should be under `Eclipse Application configurations`.

Some tricks:

- If you change a (meta-)language and want to test it in a new Eclipse instance, import that language’s corresponding Eclipse plugin project. For example, `org.metaborg.meta.lang.nabl` has Eclipse plugin project `org.metaborg.meta.lang.nabl.eclipse`. Then compile both those projects from the command-line (don’t forget to turn off build automatically in Eclipse), and start a new Eclipse instance.
- A different way to test the (meta-)language change is to import that language project into the workspace of the guest Eclipse. Because we use Maven snapshot versions, the built-in version will be overridden when you build the language in the guest eclipse.

Troubleshooting

If there are many errors in a project, try updating the Maven project. Right click the project and choose *Maven → Update Project...*, uncheck *Clean projects* in the new dialog and press *OK*. This will update the project from the POM file, update any dependencies, and trigger a build. If this does not solve the problems, try it again but this time with *Clean projects* checked. Note that if you clean a language project, it has to be rebuilt from the command-line. Restarting Eclipse and repeating these steps may also help.

Multiple projects can be updated by selecting multiple projects in the package/project explorer, or by checking projects in the update dialog.

If you have particular trouble with `org.eclipse.*` plugins in the `MANIFEST.MF` file that do not resolve, try the following. Go to *Preferences → Plug-in Development → Target Platform*, most likely there will not be an active Running Platform there. You can use *Add...* to add a new one if there isn’t one already. Select the *Default* option, click *Next*, then click *Finish*. Check the box next to the platform to activate it.

Advanced: developing from scratch

In some cases it can be beneficial to have full control over all projects, instead of relying on Maven artifacts and the installed Spoofax plugin. To develop completely from scratch, uninstall Spoofax from Eclipse, and import all projects by importing `releng/eclipse/import/pom.xml`, which will import all relevant projects automatically.

If you change a language project, build them on the command-line, because languages cannot be built inside Eclipse without the Spoofax plugin.

21.5.3 IntelliJ

Easiest is to *install the latest release of the Spoofax plugin* in an installation of IntelliJ IDEA.

Otherwise, you may want to build it from source, and to run the built plugin inside a special sandbox-instance of IntelliJ IDEA, execute the following command:

```
./gradlew runIdea
```

Alternatively, in IntelliJ IDEA you can invoke the *IntelliJ Plugin* run/debug configuration. You can use this to run or debug the IntelliJ IDEA plugin code. However, this cannot be used to debug the JPS Spoofax build process.

To debug the JPS Spoofax build process, you need to execute the following command:

```
./gradlew debugJps
```

or invoke the *IntelliJ Plugin (Debug JPS)* run configuration (*not debug*) from IntelliJ. Then in the sandbox IntelliJ IDEA instance you enable the “Debug Build Process” action (Ctrl+Shift+A). Then you start a build. IntelliJ will wait for a debugger to be attached to port 5005. Attach a debugger, and the build will continue. From the Spoofax plugin’s IntelliJ IDEA project, you can invoke the *JPS Plugin* remote debug configuration to attach the debugger.

Logging

To get debug logging in IntelliJ, locate the `bin/log.xml` file in the IntelliJ folder and add the following snippet in the `<log4j:configuration>` element, just above the `<root>` element:

```
<category name="#org.metaborg" additivity="true">
  <priority value="DEBUG"/>
  <appender-ref ref="CONSOLE-DEBUG"/>
  <appender-ref ref="FILE"/>
</category>
```

21.6 Naming Conventions

The current code base of Spoofax has a variety of naming conventions. We are in the processes of consolidating this. Any new additions to Spoofax should ideally use the following naming conventions (note that this includes both Java code and Spoofax meta languages):

Artifact ID (Identifier in the Spoofax “new project” wizard) <to-the-point, unique, name, separated by ‘.’ if needed>

Group ID (Group Identifier in the wizard) `org.metaborg`

Package `mb.<Artifact ID>.*`

Directory name (Project name in the wizard unless you specify the location separately) <Artifact ID, separated by ‘.’ or directory structure if needed>

Language name (Name in the wizard) <to-the-point, unique, name>

Everything lowercase and singular.

21.6.1 Examples

NaBL2 language

Artifact ID `nabl2.lang`

Group ID `org.metaborg`

Directory name `nabl2.lang`

Language name `nabl2`

NaBL2 solver Java code

Artifact ID nabl2.solver

Group ID org.metaborg

Package mb.nabl2.solver.*

Directory name nabl2.solver

PIE language

Artifact ID pie.lang

Group ID org.metaborg

Directory name pie/lang

Language name pie

PIE runtime (core)

Artifact ID pie.runtime.core

Group ID org.metaborg

Package mb.pie.runtime.core.*

Directory name pie/runtime/core

PIE runtime (builtin functions)

Artifact ID pie.runtime.builtin

Group ID org.metaborg

Package mb.pie.runtime.builtin.*

Directory name pie/runtime/builtin

21.7 Documentation

This section describes the documentation tools and provides guidelines for writing documentation.

21.7.1 Tools

This documentation is written with the [Sphinx documentation generator](#). Sphinx is the tool that transforms the documentation into a website and other output formats. Documentation can be found in their website:

- [Sphinx-specific Markup Constructs](#)
- [Domains](#)
- [All documentation](#)

Formats

ReStructuredText is the main documentation format used by Sphinx. Documentation can be found at:

- [Quick reference](#)
- [Primer](#)
- [Full reference documentation](#)

Markdown is also supported, but is less powerful for technical documentation purposes. It is supported through the [recommonmark](#) extension. To use markdown, just create `.md` files and link them in a `toctree`.

Converting formats with Pandoc

[Pandoc](#) is a documentation format converting tool. It can be used to convert between various documentation formats. On macOS with Homebrew, it can be installed with `brew install pandoc`.

Conversion is performed by passing the `from` and `to` flags. For example, to convert Markdown to ReStructuredText, run the following command:

```
pandoc --from=markdown --to=rst file.md --wrap=preserve > file.rst
```

See their [manual](#) for more info.

Bibliographies

BibTeX bibliographies and citations are supported through the [sphinxcontrib-bibtex](#) extension.

To create a separate bibliography for a chapter use a label and key prefix as follows:

```
Syntactic completions :cite:`s-AmorimEWV16`

.. bibliography:: ../../../../bib/spoofox.bib
   :style: plain
   :labelprefix: S
   :keyprefix: s-
```

Customizing HTML with CSS and JavaScript

The `_static` directory contains a `style.css` and `script.js` file to customize the HTML output.

21.7.2 Writing guide

General guidelines

Each chapter in the manual, i.e. a top-level entry in the table of contents should start with a paragraph that explains what the chapter is about, including a definition of the thing it documents. For example: “Stratego is a language for defining program transformations ...”

Meta-language documentation

Standard documentation ingredients for meta-language documentation:

- Introduction
 - main concepts and a small example
 - how does it fit in the bigger picture?
- Reference manual
 - a systematic description of all language features
 - include schematic descriptions
 - use examples for illustration
- Configuration
 - how to build it
 - configuration options (yaml, esv, stratego hooks)
 - how to call it / use it
- Examples
 - typical examples
 - examples for some specific features
 - pointers to real projects
- Bibliography
 - list with all or at least key publications
 - discussion of what each publication contributes

It probably makes sense to put each of these in a separate section.

21.8 Releasing Spoofox

This section describes how to release Spoofox.

21.8.1 Requirements

To release Spoofox, you must first be able to build Spoofox. Follow the [Maven](#) and [Building](#) guides first.

To publish releases, you will need write access to the [spoofox-releng](#) repository, to all submodule repositories in that repository, and to this documentation repository. An account with deploy access to our [artifact server](#) is required. Ask an administrator of the Programming Languages group to get access to the repositories and artifact server.

21.8.2 Instructions

1. Prepare Maven deploy settings.
 - (a) Open your `~/.m2/settings.xml` file.
 - (b) Add a `<servers></servers>` section if it does not exist.

- (c) Add a server to the `servers` section with id `metaborg-nexus` that contains your username and password to our artifact server:

```
<server>
  <id>metaborg-nexus</id>
  <username>myusername</username>
  <password>mypassword</password>
</server>
```

- (d) Optionally encrypt your password by following the [Password Encryption guide](#).
2. Prepare the repository containing the build scripts.
 - (a) Clone or re-use an existing clone of `spoofax-releng` on the master branch. See [Cloning the source code](#).
 - (b) **Update it to the latest commit with** `git pull --rebase && ./b checkout -y && ./b update`.
 - (c) Run `./b set-remote -s` to set submodule remotes to SSH URLs, enabling git pushing without having to supply a username and password via git over HTTPS.
 3. Prepare the source code repository.
 - (a) Make a separate clone (or re-use an existing one if you have released Spoofax before) of the `spoofax-release` branch of `spoofax-releng`. This must be a separate clone in a different directory from the first one. See [Cloning the source code](#).

Note: The reason for two separate clones of `spoofax-releng` is that the release script will modify the files in the repository, which could include files of the release script itself. Therefore, we make a separate clone which the release script acts upon, so that it does not interfere with itself.

- (b) **If reusing an existing clone, ensure that it is checked out to** `spoofax-release` **with** `git checkout spoofax-release`, **and update it to the latest commit with** `git pull --rebase && ./b checkout -y && ./b update`.
 - (c) **If there are new submodules repositories, follow the steps for preparing new submodules below.**
 - (d) Run `./b set-remote -s` to set submodule remotes to SSH URLs, enabling git pushing without having to supply a username and password via git over HTTPS.
4. Perform the release.
 - (a) Change directory into the repository cloned in step 2. For example:


```
cd /Users/gohla/spoofax/master/spoofax-releng
```
 - (b) Get an absolute path to the repository cloned in step 3. For example: `/Users/gohla/spoofax/release/spoofax-releng`
 - (c) Determine whether the release will be *patch* or *minor/major*. For a patch release, we do not bump the development version. For a minor or major release, we do.
 - (d) Figure out what the *current development version* of Spoofax is, what the *next release version* should be, and if doing a non-patch release, what the *next development version* should be. The release script will change the current development version into the next release version, deploy that, and then change the current development version to the next development version, and commit that. Setting the next development version is optional.
 - (e) Execute the release script with the parameters you gathered:


```
./b --repo <release-repository> release \
  spoofax-release <release-version> \
  master <current-development-version> \
  --non-interactive \
  --maven-deploy \
  --maven-deploy-identifier metaborg-nexus \
  --maven-deploy-url http://artifacts.metaborg.org/content/repositories/
↳ releases/ \
  --nexus-deploy \
  --nexus-username <artifact-server-username> \
  --nexus-password <artifact-server-password> \
  --nexus-repo releases
```

or for a major version, with `--next-develop-version`:

```
./b --repo <release-repository> release \
  spoofax-release <release-version> \
  master <current-development-version> \
  --next-develop-version <next-development-version> \
  --non-interactive \
  --maven-deploy \
  --maven-deploy-identifier metaborg-nexus \
  --maven-deploy-url http://artifacts.metaborg.org/content/repositories/
↳ releases/ \
  --nexus-deploy \
  --nexus-username <artifact-server-username> \
  --nexus-password <artifact-server-password> \
  --nexus-repo releases
```

For example, if we currently are at development version `2.3.0-SNAPSHOT`, and would like to release minor version `2.3.0`, and update the development version to `2.4.0-SNAPSHOT`, we would execute the following command:

```
cd /Users/gohla/spoofax/master/spoofax-releng
./b --repo /Users/gohla/spoofax/release/spoofax-releng release \
  spoofax-release 2.3.0 \
  master 2.3.0-SNAPSHOT \
  --next-develop-version 2.4.0-SNAPSHOT \
  --non-interactive \
  --maven-deploy \
  --maven-deploy-identifier metaborg-nexus \
  --maven-deploy-url http://artifacts.metaborg.org/content/repositories/
↳ releases/ \
  --nexus-deploy \
  --nexus-username myusername \
  --nexus-password mypassword \
  --nexus-repo releases
```

Unfortunately, it is currently not possible to encrypt the artifact server password passed to the build script.

21.8.3 New spoofax-releng submodules

When adding a new submodule to the `spoofax-releng` repository, the following steps must be performed before starting the automated release process:

- Add a `spoofax-release` branch to the submodule (pointing to the current `master` branch), and push that branch.
- Add the submodule to the `.gitmodule` file in the `spoofax-release` branch of the `spoofax-releng` repository. Make sure that the branch of the submodule is set to `spoofax-release`, and that the remote is using a `https` URL. Commit and push this change.

21.8.4 Updating the release archive

To update the release archive of this documentation site, perform the following steps after a release:

- Update include files:
 - Copy `include/hyperlink/download-<current-release-version>.rst` to new file `include/hyperlink/download-<release-version>.rst`, replace all instances of `<current-release-version>` in that new file with `<release-version>`, and update the date to the current date.
 - In `include/hyperlink/download-rel.rst`, replace all instances of `<current-release-version>` with `<release-version>`.
 - In `include/hyperlink/download-dev.rst`, update the development version to `<next-development-version>`.
 - In `include/_all.rst`, add a new line to include the newly copied file: `.. include:: /include/hyperlink/download-<release-version>.rst`.
- Update `source/release/migrate/<release-version>.rst` (only if migrations are necessary):
 - Remove stub notice.
- Update `source/release/note/<release-version>.rst`:
 - Remove stub notice.
 - Add small summary of the release as an introduction.
 - Include download links, which can be copied and have their versions replaced from a previous release.
- Create new stub files for the next release:
 - Create a new migration guide stub file.
 - Create a new release notes stub file.
- Update `source/release/note/index.rst`:
 - Move stub for this release to the top of the notes.
 - Add new stub file at the bottom of the notes.
- Update `source/release/migrate/index.rst`:
 - Move stub for this release to the top of the migration guides.
 - Add new stub file at the bottom of the migration guides.
- Update `conf.py`:
 - Update version variable.
 - Update `copyright` variable with new year, if needed.

21.9 IntelliJ IDEA Plugin Internals

The Spoofax plugin for IntelliJ IDEA consists of two main parts: the IntelliJ IDEA plugin and the JPS build plugin. They are separate plugins that are run in separate processes and have separate Guice bindings, but since they share a lot of code they live in the same project.

21.9.1 IntelliJ IDEA Plugin

The IntelliJ IDEA plugin is responsible for integrating the Spoofax languages in IntelliJ IDEA. It provides the editor functionality (e.g. syntax highlighting), dialogs and wizards (e.g. New Project dialog) using the IntelliJ IDEA OpenAPI.

Initialization

When the plugin is loaded, a singleton instance of the `IdeaPlugin` application component is created by IntelliJ. This loads the Guice dependency injector.

Bindings

The Guice bindings for the plugin can be found in the `IdeaSpoofaxModule` and `IdeaSpoofaxMetaModule` classes. See [Bindings](#) for more information on how to use Guice bindings.

21.9.2 JPS Plugin

The JPS build plugin is responsible for building a Spoofax language specification or Spoofax-enabled project. It's loaded in a separate process by IntelliJ IDEA when the user executes the *Make* action.

Initialization

When the plugin is loaded, the static members of the `JpsPlugin` class are initialized. This loads the Guice dependency injector.

Bindings

The Guice bindings for the plugin can be found in the `JpsSpoofaxModule` and `JpsSpoofaxMetaModule` classes. See [Bindings](#) for more information on how to use Guice bindings.

Deserialization

The JPS plugin has an internal representation of the project, its modules, files, settings, facets, SDK, and so on, in the `JpsModel` class.

When the JPS plugin is run, IntelliJ creates a JPS model that reflects the IntelliJ project structure. However, IntelliJ has no knowledge of any special extensions and settings that we have applied to the IntelliJ project. We need to update the JPS model with our own information where needed. This is done through the deserializers returned from the `JpsSpoofaxModelSerializerExtension` class.

Gathering build targets

A *build target* is a single compilation unit. JPS now gathers all build targets that can apply to the model, and orders them according to their dependencies.

The `BuilderService` advertises which build target *types* we have implemented. Each `BuildTargetType<T>` implementation then decides which build targets we have. Usually every module that we can compile gets its own `BuildTarget<T>`.

Building

Our `BuilderService` also advertised which *target builders* we have. Every `TargetBuilder<T, U>` has a `build()` method that's executed for every appropriate build target.

21.9.3 Bindings

Both the IntelliJ IDEA plugin and the JPS plugin use Guice bindings for the dependency injection.

Injecting Dependencies

To inject dependencies in a class that is created by Guice, simply annotate the constructor with `@Inject` and add the dependencies as parameters.

```
public final class MyClass implements IMyInterface {

    private final MyDependency dependency;

    @Inject
    public MyClass(final MyDependency dependency) {
        this.dependency = dependency;
    }
}
```

Register the binding in the applicable Guice module.

```
bind(IMyInterface.class).to(MyClass.class).in(Singleton.class);
```

In some cases the class is not created by Guice but by IntelliJ IDEA's `extension system` or by the `Java service loader`. In that case you have to inject your dependencies like this:

```
public final class MyClass implements IMyInterface {

    private MyDependency dependency;

    /**
     * This instance is created by IntelliJ's plugin system.
     * Do not call this constructor manually.
     */
    public MetaborgReferenceContributor() {
        SpoofaxIdeaPlugin.injector().injectMembers(this);
    }

    @SuppressWarnings("unused")
```

(continues on next page)

(continued from previous page)

```
@Inject
private void inject(final MyDependency dependency) {
    this.dependency = dependency;
}
}
```

Note that `SpoofoxIdeaPlugin` in this example can also be `SpoofoxJpsPlugin` if you're writing a JPS plugin class.

Since the class is not created by Guice, you don't need to register a binding if you're not depending on it. However, if you *do* depend on such a class, register the appropriate kind of binding that depends on who created the class and where it was registered.

For an IntelliJ component registered in `META-INF/plugin.xml` under the `<application-components />` tag:

```
install(new IntelliJComponentProviderFactory().provide(IdeaApplicationComponent.
    ↪class));
```

Note: Binding module-level or project-level components is not currently implemented.

For an IntelliJ service registered in `META-INF/plugin.xml` under the `<applicationService />` extension:

```
install(new IntelliJServiceProviderFactory().provide(IMetaborgModuleConfig.class));
```

Note: Binding module-level or project-level services is not currently implemented.

For any other IntelliJ extension registered under the `<extensions>` tag in `META-INF/plugin.xml` (e.g. the `MetaborgFacetType` class registered for the `<facetType />` extension):

```
install(new IntelliJExtensionProviderFactory().provide(
    MetaborgFacetType.class, "com.intellij.facetType"));
```

For a class registered using the Java service provider (mostly used in the JPS plugin):

```
install(new JavaServiceProviderFactory().provide(BuilderService.class));
```

Injecting Arguments

Sometimes you have some arguments that you want to use in the constructor, but the constructor is also used for dependency injection. In that case, annotate those arguments with `@Assisted` and create a factory interface for the class.

```
public final class MyClass {

    private final String name;
    private final MyDependency dependency;

    @Inject
    public MyClass(@Assisted final String name,
                  final MyDependency dependency) {
```

(continues on next page)

(continued from previous page)

```

    this.name = name;
    this.dependency = dependency;
}
}

public interface MyClassFactory {
    MyClass create(String name);
}

```

And register the factory instead of the binding:

```

install(new FactoryModuleBuilder()
    .implement(MyClass.class, MyClass.class)
    .build(MyClassFactory.class));

```

You can use it by depending on the factory instead of the class itself.

Injecting the Logger

In most classes you need the logger. Guice will automatically inject it if you add the following field:

```

@InjectLogger
private ILogger logger;

```

This also works with classes that are not created by Guice and have to use `injectMembers()`, as explained above. Injection is handled by the `MetaborgLoggerTypeListener` and `Slf4JLoggerTypeListener` classes.

In unit tests where Guice is not available, you can inject a logger in a class using the `LoggerUtils#injectLogger()` function.

Singleton Classes

Mark any classes that must only be used as singletons with the `@Singleton` attribute. This prevents issues where the registered binding is incomplete, and makes the intent of the class clear to future maintainers.

```

@Singleton
public final class MyClass implements IMyInterface {
}

```

21.9.4 Dependencies

Dependencies of the Spoofax plugin for IntelliJ IDEA are not actually managed in the plugin project itself, but in a separate project: the `deps/` project.

Add dependencies to or remove them from the `deps/build.gradle` file.

If a dependency causes a JAR version conflict with a JAR that's distributed with IntelliJ IDEA or JPS, see [the troubleshooting page](#) for a solution.

Rebuild the dependencies project to use them in the plugin:

```
./gradlew clean publishToMavenLocal
```

21.9.5 Logging

Metaborg Core uses its own logging implementation of `ILogger`. You can ask `Guice` to inject it into your classes. Underneath it calls `Slf4j` to do the logging. IntelliJ IDEA uses its own built-in `Logger` implementation that is called by the `IntelliJLoggerAdapter` that's bound to the Metaborg Core `Slf4j` logger.

In unit tests where `Guice` is not available, you can inject the `PrintStreamLogger`. See the [Bindings documentation](#) for more information.

Throwing and logging an exception

Often when you want to throw an exception, you first have to format the exception message and then log the message and finally throw the exception. The `LoggerUtils#exception()` functions make this a whole lot easier.

```
throw LoggerUtils.exception(this.logger, RuntimeException.class,
    "Error occurred in {}. ", this.name);
```

This will format the message, construct the given exception class with the current stack trace, and return it. You can then throw the exception. The stack trace is cleaned up such that the `LoggerUtils` class doesn't appear in the trace.

IntelliJ versus Spoofax

Both Spoofax Core (and related Metaborg libraries) and IntelliJ IDEA use `slf4j` for logging. The Spoofax for IntelliJ plugin also uses `slf4j` for logging, so it depends on `org.slf4j:slf4j-api:1.7.10`.

Normally `slf4j` looks for a `StaticLoggerBinder` implementation and bind to that for logging. IntelliJ IDEA provides a `StaticLoggerBinder` by default (from `org.slf4j:slf4j-log4j12:1.7.10`), so we'd like to bind to that. This is `slf4j`'s default behavior, so we don't have to add any dependencies for this. However, if we try that, we get an exception:

```
java.lang.LinkageError: loader constraint violation: when resolving method
"org.slf4j.impl.StaticLoggerBinder.getLoggerFactory()Lorg/slf4j/ILoggerFactory;" the class loader
(instance of com/intellij/ide/plugins/cl/PluginClassLoader) of the current class, org/slf4j/LoggerFactory,
and the class loader (instance of com/intellij/util/lang/UrlClassLoader) for the method's defining class,
org/slf4j/impl/StaticLoggerBinder, have different Class objects for the type org/slf4j/ILoggerFactory
used in the signature
```

Apparently there are two class loaders used to resolve the `StaticLoggerBinder`, and they interfere.

We can use our own logger binder (depend on `org.slf4j:slf4j-simple:1.7.10`), but this has some downsides: we have to configure the logger ourselves, and can't choose the `StaticLoggerBinder` that `slf4j` should bind to. This causes it to warn us that there are now two `StaticLoggerBinder` classes and it doesn't know which one to bind to. It will pick one at random:

```
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/home/daniel/.IdeaIC14/system/plugins-sandbox/
→plugins/spoofax-intellij/lib/slf4j-simple-1.7.10.jar!/org/slf4j/impl/
→StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/home/daniel/apps/1507-IntelliJ/lib/slf4j-log4j12-1.
→7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.SimpleLoggerFactory]
```

We don't have a solution for this yet.

Full stack trace of the error

21.9.6 Virtual File System

Virtual File Systems are used to abstract different file systems in a uniform way. IntelliJ IDEA uses it for its representation of project files, and Metaborg Core uses it for uniform access to resources, regardless of where they physically live.

IntelliJ VFS

IntelliJ IDEA uses its own Virtual File System, whose main class is the `VirtualFile` class. It captures a snapshot of the physical file which is periodically synchronized. You can find more information [here](#) and [here](#).

The textual content of a file is represented by a `Document`. [More information](#).

A file may be part of a PSI tree. In that case the file is a `PsiFile` ([more information](#)) and it contains PSI elements ([more information](#)). Think of the PSI tree as a mix between an Abstract Syntax Tree and a Parse Tree, but it also extends outside the file to represent the project that contains the file.

Note: While `VirtualFile` has an `exists()` method that can be used to check whether a file exists, it's also possible for functions such as the following to return `null` when the file is not found:

```
this.file = LocalFileSystem.getInstance().refreshAndFindFileByPath("/home/user/test.  
↪txt");
```

Apache VFS

Metaborg Core uses [Apache Commons VFS](#), which despite the similar name and function is completely different from IntelliJ's VFS. An implementation of the Apache VFS for IntelliJ resides in the `org.metaborg.intellij.vfs` package.

The default scheme for the implementation is `intellij://`, and any IntelliJ VFS virtual files are resolved to Apache VFS file objects in the `intellij://` file system using the [default implementation](#) of the `IIntelliJResourceService`.

21.9.7 Configurations

This document is about the configuration data used within the Spoofax plugin. Go to the [Spoofax Core documentation](#) for more information on the project and language specification configurations.

There are currently four configurations:

- Application configuration.
- Project configurations.
- Module configurations.
- Facet configurations.

These configurations are persisted between IntelliJ IDEA sessions (in XML-files), and are marshalled to the JPS build plugin when it's invoked.

Below I'm describing the application configuration, but the other configurations are treated and implemented very similarly.

Configuration State

The configuration state (the raw data) is represented by the `MetaborgApplicationConfigState` class. The state class may only contain simple types: numbers, booleans, strings, collections, maps and enums.

The default constructor of the state is used to initialize the state to its default value.

Note: IntelliJ only persists the configuration state when it has changed from the default state.

The state must implement the `equals()` method to allow it to be compared to other states.

Configuration Interface

The configuration state may be very crude, as it contains only simple types. The `IMetaborgApplicationConfig` interface describes how the data can be accessed in a nicer way. This interface is used by both the IntelliJ IDEA and JPS plugins.

IDEA configuration

Asking for the `IMetaborgApplicationConfig` interface in the IntelliJ IDEA plugin will give you the `IdeaMetaborgApplicationConfig` class that implements the `PersistentStateComponent<T>` interface. The class has a `@State` attribute that indicates where the configuration is persisted. By persisting it to default locations, the configuration can be read from within the JPS plugin.

JPS configuration

In the JPS plugin asking for the `IMetaborgApplicationConfig` interface will give you the `JpsMetaborgApplicationConfig` class that reads the configuration back. Reading is automatically done through the `MetaborgApplicationConfigDeserializer` class.

21.9.8 Languages

Language management

All language operations (loading, unloading, activating, deactivating, and discovery) should go through the `ILanguageManager` or `IIdealLanguageManager` interface. The implementations take care in registering, unregistering, activating and deactivating the languages such that the IDE can use them.

Discovering a language doesn't load it. Loading a language makes Metaborg Core aware of it. Activating a language makes it usable in the IDE. For JPS plugins language activation is not available, as it doesn't have an IDE.

Language sources

A Metaborg language might be found in a Maven repository, or installed locally. The `ILanguageSource` interface was introduced to abstract this. Given a language identifier, it will return the location of a language component for that language identifier.

Currently it only looks in the plugin's resources, but future implementations may search Maven Central or artifacts.metaborg.org.

Languages in IntelliJ

IntelliJ expects a unique implementation of the abstract `Language` class for each language. It considers two `Language` to be the same if they are instances of the same class. Therefore, to represent multiple Spoofox languages, we need to create our own `Language` class implementations dynamically.

The built-in `java.lang.reflect.Proxy` class is not sufficient. First of all I'm not sure whether different proxies are different classes, but most importantly, the `Proxy` class only supports implementing interfaces. We need to implement the abstract class `Language`, so this won't work.

There are several third-party libraries that allow you to create classes at runtime. A very promising one is *Javassist*, which I've used before when trying to build a profiler for Stratego. [Here is some information](#).

21.9.9 Lexing and Parsing

IntelliJ relies on a lexer and parser.

Lexer

The lexer shall extend the `com.intellij.lexer.Lexer` class (usually through the `com.intellij.lexer.LexerBase` class). IntelliJ will call its `start` method, providing it with a character buffer and range that need to be lexed, and the lexer state at the start of the region. Then the lexer will lex forward on each call to `advance`, providing information about the current token's type and offset, and the lexer state at that point.

Note that the lexer may not know which file is being lexed, and it needs to be able to start at any arbitrary point in the input. Also, it needs to return *every* token, including layout, even invalid ones.

Parser

The parser is fed the lexer tokens, and turns them into AST nodes. A single AST node can consist of multiple tokens. Again, the parser must be able to parse from within a file.

21.9.10 Projects

A Metaborg project can contain source files written in Metaborg languages. Such projects are represented by classes derived from the `IProject` interface. In the IDEA plugin this is the `IdeaProject` class, but in the JPS plugin this is the `MetaborgJpsProject` class.

A Metaborg language specification defines a new language. Such projects are represented by `IdeaLanguageSpec` (IDEA) and `JpsLanguageSpec` (JPS), both of which derive from the `ISpoofoxLanguageSpec` interface, which indirectly derives from `IProject`.

Additionally, there is the `ArtifactProject` for language artifacts.

Modules

IntelliJ IDEA uses the concept of *modules*, which are similar to projects in Eclipse. To get the Metaborg project that corresponds to a module, use the `IdeaProjectService` methods (in IDEA), or the `IJpsProjectService` (or JPS) methods.

An IntelliJ module lives in a *project*, which corresponds to a workspace in Eclipse. And projects live in the application, which is application-wide.

A module has so-called content roots: folders that hold the content of the module. A simple module just has the module's root folder as a content root, but a more complex module might have several content roots from all over the place. A single file may belong to multiple modules.

Module Type

A language specification project is represented in IntelliJ IDEA as a module with the `MetaborgModuleType`. Such a module is created and managed by the `MetaborgModuleBuilder`. Its `setupRootModel()` method is responsible for setting up the module and creating its files.

21.9.11 Troubleshooting

NoSuchMethodError in IntelliJ IDEA or when running Spoofax build

There may be a JAR version conflict, as IntelliJ and JPS use their own versions of certain common JARs (e.g. Apache's Commons IO) and those may get loading priority.

To fix this, go to the dependencies project (in `deps/`) and edit the `build.gradle` file. In the `shadowJar` section, add a new relation from the original package to the new package (usually the same package name prefixed with `intellij.`):

```
relocate 'org.apache.commons.io', 'intellij.org.apache.commons.io'
```

Now, whenever you want to use a class from that package you have to use the new package name. The build script will take care of renaming usages of the package in the dependencies.

Note: Rebuild the dependencies project.

```
./gradlew clean publishToMavenLocal
```

21.9.12 See also

- Bjorn Tipling — [How to make an IntelliJ IDEA plugin in less than 30 minutes](#)
- JetBrains — [Custom Language Support Tutorial](#)
- JetBrains — [Writing Tests For Plugins](#)
- Terence Parr — [IntelliJ Plugin Development Notes](#)

Latest Stable Release

This page contains download links to all artifacts of the latest stable release of Spoofox.

The latest stable release of Spoofox is 2.5.16, released on 04-06-2021.

We recommend the use of the stable release version, because we test the release, and make sure that it is available indefinitely. Follow the [Installation Guide](#) for instructions on how to install and run Spoofox.

22.1 Eclipse plugin

22.1.1 Premade Eclipse installations

With embedded JRE:

- [Windows 32-bit with embedded JRE](#)
- [Windows 64-bit with embedded JRE](#)
- [Linux 64-bit with embedded JRE](#)
- [macOS Intel with embedded JRE](#)

Without embedded JRE:

- [Windows 32-bit](#)
- [Windows 64-bit](#)
- [Linux 64-bit](#)
- [macOS Intel](#)

22.1.2 Update site

- Eclipse update site: `http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.16/org.metaborg.spoofax.eclipse.update.site-2.5.16-assembly.zip-unzip/`
- [Eclipse update site archive](#)

22.2 IntelliJ plugin

- IntelliJ update site: `http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.16/org.metaborg.intellij.dist-2.5.16.zip`
- [IntelliJ update site archive](#)

22.3 Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

22.4 Core API

- [Spoofax Core uber JAR](#)
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.16`

22.5 StrategoXT

- [StrategoXT distribution](#)
- [StrategoXT JAR](#)

22.6 Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.16. See the instructions on *using MetaBorg Maven artifacts* for more information.

Current Development Release

This page contains download links to all artifacts of the latest development version of Spoofox.

The development version of Spoofox is `2.6.0-SNAPSHOT`.

Development versions of Spoofox use snapshot versioning, meaning that they always point to the latest build that is made whenever a change is pushed to Spoofox's repositories. This means that the development version is untested, and that older builds are not available since they are continuously replaced by newer builds. We do not recommend usage of the development version for these reasons. Only use this version for development, when you absolutely need its features, or if you know what you are doing.

23.1 Eclipse plugin

23.1.1 Premade Eclipse installations

With embedded JRE:

- [Windows 32-bit with embedded JRE](#)
- [Windows 64-bit with embedded JRE](#)
- [Linux 64-bit with embedded JRE](#)
- [macOS Intel with embedded JRE](#)

Without embedded JRE:

- [Windows 32-bit](#)
- [Windows 64-bit](#)
- [Linux 64-bit](#)
- [macOS Intel](#)

23.1.2 Update site

- Eclipse update site: <http://buildfarm.metaborg.org/job/metaborg/job/spoofax-releng/job/master/lastSuccessfulBuild/artifact/dist/spoofax/eclipse/site/>

23.2 IntelliJ plugin

- IntelliJ update site: <http://buildfarm.metaborg.org/job/metaborg/job/spoofax-releng/job/master/lastSuccessfulBuild/artifact/dist/spoofax/intellij/plugin.zip>
- [IntelliJ update site archive](#)

23.3 Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

23.4 Core API

- [Spoofax Core uber JAR](#)
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.6.0-SNAPSHOT`

23.5 StrategoXT

- [StrategoXT distribution](#)
- [StrategoXT JAR](#)

23.6 Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is `2.6.0-SNAPSHOT`. See the instructions on *using MetaBorg Maven artifacts* for more information.

23.7 Build farm

Our build server builds Spoofox whenever a commit to master is made, in the [metaborg/spoofax/master](#) build job. The latest successfully built artifacts from that job are stored [here](#).

This section contains the release notes for all versions of Spoofax, starting at version 1.0.0. Release notes describe the changes made in the release, and provide download links for components of Spoofax of that version.

24.1 Spoofax 2.5.16

Spoofax 2.5.16 contains a couple of small improvements and bug fixes.

24.1.1 Changes

SDF3

- This release fixes the bug with the automatic generation of namespaced grammars, which was introduced in the previous release.

Statix

- Added `stc-get-ast-ref` rule to the Stratego API, which can be used to query `ref` properties.
- The Stratego primitives now issue console warnings when invalid labels or properties are used.
- Fixed a bug where `stx-get-scopegraph-data` would return unification variables instead of their values.
- Changed the default data order to `true`, to make queries where only a label order is provided apply shadowing as expected.
- Added a menu option to execute tests with the concurrent solver
- Fixed a completeness bug in the traditional solver when executing queries in `dataWf` or `dataLeq` predicates.

24.1.2 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- [Windows 32-bits, embedded JRE](#)
- [Windows 64-bits, embedded JRE](#)
- [Linux 64-bits, embedded JRE](#)
- [macOS, embedded JRE](#)

Without embedded JRE:

- [Windows 32-bits](#)
- [Windows 64-bits](#)
- [Linux 64-bits](#)
- [macOS](#)

Update site

- Eclipse update site: <http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.16/org.metaborg.spoofax.eclipse.update.site-2.5.16-assembly.zip-unzip/>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.16/org.metaborg.intellij.dist-2.5.16.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

Core API

- [Spoofax Core uber JAR](#)
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.16`

StrategoXT

- [StrategoXT distribution](#)
- [StrategoXT JAR](#)

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.16. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.2 Spoofox 2.5.15

Spoofax 2.5.15 contains a couple of small improvements and bug fixes, and supports the old SDF2-based parse table generator on macOS Catalina (10.15) and above.

See the corresponding [migration guide](#) for migrating from Spoofox 2.5.14 to Spoofox 2.5.15.

24.2.1 Changes

macOS

- On macOS, Spoofox temporarily requires [Docker](#) and `coreutils` when building Spoofox on macOS Catalina, Big Sur, or newer. (This is only when you build Spoofox yourself instead of downloading it for this website, it does not influence building Spoofox projects.)

SDF3

- Fixed tree indexes in layout constraints/declarations to make them 0-based.
- The generate namespaced grammar option will now generate the namespaced grammar in `src-gen`. This feature can also be set to generate the grammar automatically similar to other extractions of the grammar like Stratego signatures. See the [documentation](#) for more information. Sadly, due to a bug in the changes for automatic generation, a build in Eclipse of a language project with namespaced grammar will work, but the build of that project with Maven will not work.

Statix

- Fixed origin tracking in Statix injection explication for new projects that caused the top-level term of an AST to be missing when a Stratego strategy is applied to an analyzed AST in an SPT test.
- Add a menu action to view the scope graph resulting from Statix analysis.
- Deprecate namespaces, occurrences and query sugar.
- Fix bug in evaluation of `try` construct.
- Improvements to memory usage and runtime of the solver.
- Improve rule overlap handling: consider variables already bound to the left more specific than concrete patterns, to keep with left-to-right specificity.
- Add configuration settings to control trace length and term depth in error messages.

Stratego

- The previously advertised incremental compiler was considered too slow and attempts to make it faster made it less stable. It is currently not recommended for general use, while we develop a new version. The documentation on how to use contains a similar warning now.

24.2.2 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- [Windows 32-bits, embedded JRE](#)
- [Windows 64-bits, embedded JRE](#)
- [Linux 64-bits, embedded JRE](#)
- [macOS, embedded JRE](#)

Without embedded JRE:

- [Windows 32-bits](#)
- [Windows 64-bits](#)
- [Linux 64-bits](#)
- [macOS](#)

Update site

- Eclipse update site: <http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.15/org.metaborg.spoofax.eclipse.update.site-2.5.15-assembly.zip-unzip/>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.15/org.metaborg.intellij.dist-2.5.15.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

Core API

- Spoofax Core uber JAR
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.15`

StrategoXT

- StrategoXT distribution
- StrategoXT JAR

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.15. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.3 Spoofax 2.5.14

Spoofax 2.5.14 was released.

24.3.1 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- Windows 32-bits, embedded JRE
- Windows 64-bits, embedded JRE
- Linux 64-bits, embedded JRE
- macOS, embedded JRE

Without embedded JRE:

- Windows 32-bits
- Windows 64-bits
- Linux 64-bits
- macOS

Update site

- Eclipse update site: `http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.14/org.metaborg.spoofax.eclipse.update.site-2.5.14-assembly.zip-unzip/`
- Eclipse update site archive

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.14/org.metaborg.intellij.dist-2.5.14.zip>
- IntelliJ update site archive

Command-line utilities

- Sunshine JAR
- SPT testrunner JAR

Core API

- Spoofax Core uber JAR
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.14`

StrategoXT

- StrategoXT distribution
- StrategoXT JAR

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.14. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.4 Spoofax 2.5.13

Spoofax 2.5.13 contains a couple of small improvements.

24.4.1 Changes

SDF3

`prefer` and `avoid` are now deprecated. Usages of the operators will be marked with a deprecation warning.

Parser

The JSGLR2 parser variants now report warnings on ambiguously parsed substrings. This includes ambiguities in lexical and layout syntax that do not result into `amb` nodes in the AST.

SPT

The `run` expectation now allows to call strategies with term arguments. It's now also possible to test if a strategy failed. See the SPT documentation for more details.

24.4.2 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- [Windows 32-bits, embedded JRE](#)
- [Windows 64-bits, embedded JRE](#)
- [Linux 64-bits, embedded JRE](#)
- [macOS, embedded JRE](#)

Without embedded JRE:

- [Windows 32-bits](#)
- [Windows 64-bits](#)
- [Linux 64-bits](#)
- [macOS](#)

Update site

- Eclipse update site: <http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.13/org.metaborg.spoofax.eclipse.update.site-2.5.13-assembly.zip-unzip/>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.13/org.metaborg.intellij.dist-2.5.13.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

Core API

- Spoofax Core uber JAR
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.13`

StrategoXT

- StrategoXT distribution
- StrategoXT JAR

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.13. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.5 Spoofax 2.5.12

Spoofax 2.5.12 contains an experimental gradual type system for Stratego, performance improvements to NaBL2 and Statix, and updates to Eclipse installations and their embedded JREs.

24.5.1 Changes

Stratego

Stratego has two new reserved words: `cast` and `is`. Local variables can be reserved words if they start with `'`, so you can use `'cast` and `'is`.

Under the Stratego language options in your `metaborg.yaml` file you can turn on the gradual type system, if you use the incremental compiler. This option is `gradual: static`, and only tests the types statically. The default is `gradual: none` right now, meaning the gradual type system is not on by default. The `is` dynamic type check is not yet supported at runtime, you may get Java compilation errors when attempting to compile Stratego code with that. Dynamic type casts inserted by the gradual type system are also forthcoming, runtime support for this is not yet ready.

NaBL2

- NaBL2 supports a new resolution algorithm based on fixid-point environment computation instead of graph search, which can be enabled by adding `strategy environments` to the `name-resolution` signature section. It has much better performance characteristics, especially when dealing with mutually importing scopes and transitive imports. Compared the the search-based, the environment-based algorithm can get stuck on scope graphs with cycles involving scopes importing references that can be resolved via that same scope. Note that the environment-based algorithm may increase memory usage. The default remains the search-based algorithm.
- If a file was already analyzed in the editor, it is not reanalyzed on save anymore.

Statix

- Analysis times of large, multi-file Statix specifications has improved significantly.
- If a file was already analyzed in the editor, it is not reanalyzed on save anymore.

Eclipse

- Premade Eclipse installations have been updated from Eclipse Photon to Eclipse 2020-6.
- Premade Eclipse installations for 32-bit Linux are no longer created.
- Embedded JRE in premade Eclipse installations has been updated from 8u162 (Oracle JRE) to 8u265-b01 (AdoptOpenJDK).

24.5.2 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- [Windows 32-bits, embedded JRE](#)
- [Windows 64-bits, embedded JRE](#)
- [Linux 64-bits, embedded JRE](#)
- [macOS, embedded JRE](#)

Without embedded JRE:

- [Windows 32-bits](#)
- [Windows 64-bits](#)
- [Linux 64-bits](#)
- [macOS](#)

Update site

- Eclipse update site: <http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.12/org.metaborg.spoofax.eclipse.update.site-2.5.12-assembly.zip-unzip/>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.12/org.metaborg.intellij.dist-2.5.12.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- Sunshine JAR
- SPT testrunner JAR

Core API

- Spoofox Core uber JAR
- Spoofox Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.12`

StrategoXT

- StrategoXT distribution
- StrategoXT JAR

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.12. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.6 Spoofox 2.5.11

Spoofox 2.5.11 contains a dependency bugfix.

24.6.1 Changes

Overall

- Exclude the `hadoop-hdfs-client` transitive dependency from Apache VFS2

24.6.2 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- Windows 32-bits, embedded JRE
- Windows 64-bits, embedded JRE
- Linux 32-bits, embedded JRE
- Linux 64-bits, embedded JRE
- macOS, embedded JRE

Without embedded JRE:

- Windows 32-bits
- Windows 64-bits
- Linux 32-bits
- Linux 64-bits
- macOS

Update site

- Eclipse update site: `http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.11/org.metaborg.spoofax.eclipse.update.site-2.5.11-assembly.zip-unzip/`
- Eclipse update site archive

IntelliJ plugin

- IntelliJ update site: `http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.11/org.metaborg.intellij.dist-2.5.11.zip`
- IntelliJ update site archive

Command-line utilities

- Sunshine JAR
- SPT testrunner JAR

Core API

- Spoofox Core uber JAR
- Spoofox Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.11`

StrategoXT

- StrategoXT distribution
- StrategoXT JAR

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.11. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.7 Spoofox 2.5.10

Spoofox 2.5.10 contains several smaller improvements.

24.7.1 Changes

Overall

- Update Apache VFS2 to 2.6.0
- Support for TypeSmart was removed. We anticipate a more useable type analysis for Stratego in the form of a gradual type system.

The `metaborg.yaml` file of a generated project used to contain a `debug: typesmart: false`. This was to turn off the TypeSmart dynamic analysis by default. This analysis would stop any Stratego code when it tried to construct a tree that did not conform to the grammar of the project.

To our knowledge TypeSmart was not used in any active Spoofox project. It did, however, slow down the build time of all Spoofox projects, because extraction of the grammar into a TypeSmart readable format had to be done even if the analysis was off for that project. These two points, and the anticipation of a gradual type system for Stratego, were the reasons to drop TypeSmart support.

SDF3

Lexical and context-free sort declarations: In SDF3 you can now explicitly declare your sorts. Declare lexical sorts in a `lexical sorts` block, and context-free sorts in a `context-free sorts` block. Sorts declared in a kernel `sorts` block default to declaring context-free sorts until a suffix such as `-LEX` is added. Note that you have to use `sdf2table: java` to support lexical sorts.

Statix

- New project that use Statix automatically have the Statix signature generator enabled. For this to work properly, declare your lexical and context-free sorts in SDF3 explicitly. See the [Statix signature generator](#) documentation for more information.
- Statix specifications are now compiled as much as possible, even if there are errors in some files. Errors in Statix files that are not actually imported, do not cause analysis to fail on an empty specification anymore.
- The AST property `type` is now a built-in, which is automatically used in the default editor hover strategy.

Stratego

Combined compiled Stratego and helper code Compilation of Stratego and helper code written in Java (in `src/main/strategies`) is now combined in a single jar file per Spoofox language instead of two. See the migration guide for more information on what to change in your Spoofox project.

SPT

SPT gains support for the `parse ambiguous` expectation, which succeeds when a fragment parses successfully but with ambiguities. Tests with the `parse succeeds` expectation will now fail when the input parses ambiguously. To write tests for ambiguous parses, use the `parse ambiguous` expectation instead.

24.7.2 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- [Windows 32-bits, embedded JRE](#)
- [Windows 64-bits, embedded JRE](#)
- [Linux 32-bits, embedded JRE](#)
- [Linux 64-bits, embedded JRE](#)
- [macOS, embedded JRE](#)

Without embedded JRE:

- [Windows 32-bits](#)
- [Windows 64-bits](#)
- [Linux 32-bits](#)
- [Linux 64-bits](#)
- [macOS](#)

Update site

- Eclipse update site: <http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofox.eclipse.update.site/2.5.10/org.metaborg.spoofox.eclipse.update.site-2.5.10-assembly.zip-unzip/>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.10/org.metaborg.intellij.dist-2.5.10.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

Core API

- [Spoofox Core uber JAR](#)
- Spoofox Core uber Maven artifact: `org.metaborg:org.metaborg.spoofox.core.uber:2.5.10`

StrategoXT

- [StrategoXT distribution](#)
- [StrategoXT JAR](#)

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.10. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.8 Spoofox 2.5.9

Spoofox 2.5.9 includes dependency upgrades.

24.8.1 Changes

Overall

The following dependencies of Spoofox Core have been updated to the latest version:

- `com.netflix.rxjava:rxjava:0.20.7 -> io.reactivex.rxjava3:rxjava:3.0.2`
 - New transitive dependency: `org.reactivestreams:reactive-streams:1.0.3`
- `org.apache.commons:commons-configuration2:2.2 -> org.apache.commons:commons-configuration2:2.7`
 - New transitive dependency: `org.apache.commons:commons-text:1.8`
- `com.virtlink.commons:commons-configuration2-jackson:0.7.0 -> com.virtlink.commons:commons-configuration2-jackson:0.10.0`
- `com.fasterxml.jackson.core:jackson-core:2.9.5 -> com.fasterxml.jackson.core:jackson-core:2.11.0`
- `com.fasterxml.jackson.core:jackson-databind:2.9.5 -> com.fasterxml.jackson.core:jackson-databind:2.11.0`
- `com.fasterxml.jackson.core:jackson-annotations:2.9.5 -> com.fasterxml.jackson.core:jackson-annotations:2.11.0`
- `com.fasterxml.jackson.core:jackson-dataformat-yaml:2.9.5 -> com.fasterxml.jackson.core:jackson-dataformat-yaml:2.11.0`
- `org.yaml:snakeyaml:1.18 -> org.yaml:snakeyaml:1.26`

The following dependencies of Spoofox-Meta Core have been updated:

- `org.apache.commons:commons-compress:1.16.1 -> org.apache.commons:commons-compress:1.20`

24.8.2 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- [Windows 32-bits, embedded JRE](#)
- [Windows 64-bits, embedded JRE](#)
- [Linux 32-bits, embedded JRE](#)
- [Linux 64-bits, embedded JRE](#)
- [macOS, embedded JRE](#)

Without embedded JRE:

- [Windows 32-bits](#)
- [Windows 64-bits](#)
- [Linux 32-bits](#)
- [Linux 64-bits](#)
- [macOS](#)

Update site

- Eclipse update site: <http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.9/org.metaborg.spoofax.eclipse.update.site-2.5.9-assembly.zip-unzip/>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.9/org.metaborg.intellij.dist-2.5.9.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

Core API

- [Spoofax Core uber JAR](#)
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.9`

StrategoXT

- StrategoXT distribution
- StrategoXT JAR

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.9. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.9 Spoofox 2.5.8

Spoofox 2.5.8 includes several bugfixes and improvements.

24.9.1 Changes

SDF

- The Java version of sdf2table is now slightly faster and takes up less peak memory due to improvements in writing away the parsetable to file.
- The old (Aster based) version of make-permissive (to add error recovery to your grammar) used to be called in a way that create a small memory leak, which would compound over time with subsequent builds. This is has now been fixed. The old version of make-permissive is only in effect if you use `sdf2table: c` in your `metaborg.yaml` file.

Parser

- Add two experimental variants to the JSGLR2 parser: `recovery` and `recovery-incremental`.
- Add Unicode support to the JSGLR1 and JSGLR2 parsers. The meta-languages themselves do not support Unicode yet, because they are bootstrapped with an old version of SDF3. However, other languages built with Spoofox can use Unicode.
- Add logging to the JSGLR2 parser. Configure by setting `language.sdf.jsglr2-logging` to `all`, `none`, `minimal`, `parsing` or `recovery` in `metaborg.yaml`.

Programmatic API

- `TermFactory` for building Stratego terms now supports a builder for lists that creates an arraylist-like structure instead of the standard linkedlist-like structure. This is typically more efficient for building stratego list terms in Java.
- Add `org.spoofax.terms.util.TermUtils` class with functions for working with terms. This replaces the equivalent (now deprecated) functions in `org.spoofax.interpreter.core.Tools`.

NaBL2

- Improve error message location when scopes are used as term indices.
- Dropped support for polymorphism, which was unsound.
- Small improvements to solver performance.
- Add support for external calls for language with Stratego JAR compilation.

Statix

- *Ability to automatically generate* Statix signatures from SDF3 specifications.
- Add support for importing other modules in Statix specifications.
- Add support for custom messages, and a `try` construct for warnings and notes.
- Add support for adding multiple values to AST properties.
- Improve disunification support in the solver.
- Extend reserved keywords to fix parsing problems.
- Several smaller bugfixes.

Overall

- Fixed several issues with files not being released properly, causing file I/O errors on Windows.

24.9.2 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- Windows 32-bits, embedded JRE
- Windows 64-bits, embedded JRE
- Linux 32-bits, embedded JRE
- Linux 64-bits, embedded JRE
- macOS, embedded JRE

Without embedded JRE:

- Windows 32-bits
- Windows 64-bits
- Linux 32-bits
- Linux 64-bits
- macOS

Update site

- Eclipse update site: `http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.8/org.metaborg.spoofax.eclipse.update.site-2.5.8-assembly.zip-unzip/`
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: `http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.8/org.metaborg.intellij.dist-2.5.8.zip`
- [IntelliJ update site archive](#)

Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

Core API

- [Spoofax Core uber JAR](#)
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.8`

StrategoXT

- [StrategoXT distribution](#)
- [StrategoXT JAR](#)

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.8. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.10 Spoofox 2.5.7

Spoofax 2.5.7 includes minor bugfixes and improvements to the experimental Stratego separate compiler.

24.10.1 Changes

FlowSpec

Bugfix: Names with namespaces were broken in an earlier version during performance optimization. The error would like: `java.lang.AssertionError: Unrecognised Namespace: Namespace("Var")`.

Stratego

Stratego separate compilation is now switched to the new system. It no longer has any limitations that were previously mentioned. Do note that separate compilation will give the same stricter error messages that the editor does: You need to import anything you use, you cannot use something that another module imports that imports your module.

24.10.2 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- [Windows 32-bits, embedded JRE](#)
- [Windows 64-bits, embedded JRE](#)
- [Linux 32-bits, embedded JRE](#)
- [Linux 64-bits, embedded JRE](#)
- [macOS, embedded JRE](#)

Without embedded JRE:

- [Windows 32-bits](#)
- [Windows 64-bits](#)
- [Linux 32-bits](#)
- [Linux 64-bits](#)
- [macOS](#)

Update site

- Eclipse update site: <http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.7/org.metaborg.spoofax.eclipse.update.site-2.5.7-assembly.zip-unzip/>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.7/org.metaborg.intellij.dist-2.5.7.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

Core API

- Spoofax Core uber JAR
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.7`

StrategoXT

- StrategoXT distribution
- StrategoXT JAR

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.7. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.11 Spoofax 2.5.6

Spoofax 2.5.6 is a minor bugfix release.

24.11.1 Changes

Overall

Statix

- Fix a crash in single-file analysis.
- Fix several bugs in scope extension checking.
- Fix bug in rule application that dropped cause.

24.11.2 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- Windows 32-bits, embedded JRE
- Windows 64-bits, embedded JRE
- Linux 32-bits, embedded JRE
- Linux 64-bits, embedded JRE
- macOS, embedded JRE

Without embedded JRE:

- Windows 32-bits

- Windows 64-bits
- Linux 32-bits
- Linux 64-bits
- macOS

Update site

- Eclipse update site: `http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.6/org.metaborg.spoofax.eclipse.update.site-2.5.6-assembly.zip-unzip/`
- Eclipse update site archive

IntelliJ plugin

- IntelliJ update site: `http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.6/org.metaborg.intellij.dist-2.5.6.zip`
- IntelliJ update site archive

Command-line utilities

- Sunshine JAR
- SPT testrunner JAR

Core API

- Spoofax Core uber JAR
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.6`

StrategoXT

- StrategoXT distribution
- StrategoXT JAR

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.6. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.12 Spoofax 2.5.5

Spoofax 2.5.5 is a minor bugfix release. There are a few incompatible changes in Statix, which are described in the *migration guide*.

24.12.1 Changes

Overall

- Do not throw away error messages in unchanged files if other files changed, when using constraint analyzer.

JSGLR

- Add missing location information on sublists.

Statix

- Improve speed of normalization.
- Add AST properties and editor reference resolution.
- Regular expression and label order are direct parameters to queries. It is not possible anymore to pass an arbitrary predicate there.
- Special path constraints are removed in favour of concrete path terms that can be matched as terms.
- Functional constraints can only have a single output.
- Namespace based resolution short-hands must contain a occurrence literal, and explicit resolution policies.

24.12.2 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- [Windows 32-bits, embedded JRE](#)
- [Windows 64-bits, embedded JRE](#)
- [Linux 32-bits, embedded JRE](#)
- [Linux 64-bits, embedded JRE](#)
- [macOS, embedded JRE](#)

Without embedded JRE:

- [Windows 32-bits](#)
- [Windows 64-bits](#)
- [Linux 32-bits](#)
- [Linux 64-bits](#)
- [macOS](#)

Update site

- Eclipse update site: `http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.5/org.metaborg.spoofax.eclipse.update.site-2.5.5-assembly.zip-unzip/`
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: `http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.5/org.metaborg.intellij.dist-2.5.5.zip`
- [IntelliJ update site archive](#)

Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

Core API

- [Spoofax Core uber JAR](#)
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.5`

StrategoXT

- [StrategoXT distribution](#)
- [StrategoXT JAR](#)

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.5. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.13 Spoofox 2.5.4

Spoofax 2.5.4 is a minor bugfix release.

24.13.1 Changes

Maven

- Fixed failing SPT tests failing the build immediately. All SPT files are processed and a summary of how many tests failed is shown at the end.

- Fixed class loading errors at the end of Maven builds.
- Fixed application icon from showing up when building languages on some platforms.

Statix

- Fix Statix analysis crash when detailed logging is enabled.

24.13.2 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- [Windows 32-bits, embedded JRE](#)
- [Windows 64-bits, embedded JRE](#)
- [Linux 32-bits, embedded JRE](#)
- [Linux 64-bits, embedded JRE](#)
- [macOS, embedded JRE](#)

Without embedded JRE:

- [Windows 32-bits](#)
- [Windows 64-bits](#)
- [Linux 32-bits](#)
- [Linux 64-bits](#)
- [macOS](#)

Update site

- Eclipse update site: <http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.4/org.metaborg.spoofax.eclipse.update.site-2.5.4-assembly.zip-unzip/>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.4/org.metaborg.intellij.dist-2.5.4.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- Sunshine JAR
- SPT testrunner JAR

Core API

- Spoofax Core uber JAR
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.4`

StrategoXT

- StrategoXT distribution
- StrategoXT JAR

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.4. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.14 Spoofax 2.5.3

Spoofax 2.5.3 is a minor release with bugfixes, performance improvements, and new small and/or experimental features.

24.14.1 Changes

Overall

- Added support for getting the selected term in Stratego builders/transformations. In the builder tuple `(node, _, ast, path, projectPath)`, the first term (`node`) is now the selected term when a builder is executed in the context of an editor with a selection. The term is selected by finding the outermost term that has an origin that fits in the selection.
- Fixed a bug that prevented source transformations from being run if context or analysis were missing.
- Changed constraint analyzer to support more multi-file scenarios.

JSGLR2

- Added an incremental variant of the JSGLR2 parser (experimental).

NaBL2

- Improved performance of AST resolution lookups.

Statix (experimental)

- Fixed bugs and improved performance.

Eclipse

- Added a lifecycle mapping that adds a Spoofax nature to an imported spoofax-project.

24.14.2 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- Windows 32-bits, embedded JRE
- Windows 64-bits, embedded JRE
- Linux 32-bits, embedded JRE
- Linux 64-bits, embedded JRE
- macOS, embedded JRE

Without embedded JRE:

- Windows 32-bits
- Windows 64-bits
- Linux 32-bits
- Linux 64-bits
- macOS

Update site

- Eclipse update site: <http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.3/org.metaborg.spoofax.eclipse.update.site-2.5.3-assembly.zip-unzip/>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.3/org.metaborg.intellij.dist-2.5.3.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

Core API

- [Spoofax Core uber JAR](#)
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.3`

StrategoXT

- [StrategoXT distribution](#)
- [StrategoXT JAR](#)

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.3. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.15 Spoofox 2.5.2

Spoofax 2.5.2 is a minor bugfix and performance improvement release.

24.15.1 Changes

NaBL2

A bug introduced in 2.5.2 would remove the parse error in the editor as soon as analysis failed. This bug has been fixed.

FlowSpec

A whole host of bugs has been fixed in FlowSpec, mostly ones that lead to no clear error message. Much of the system has also been optimized for speed.

Stratego

Separate compilation for Stratego was added in this release. It is currently still experimental. It is documented as a separate item under the Stratego documentation, including instructions on how to opt-in to it and what its limitations are.

The Stratego primitives `all`, `some` and `one` sometimes lost annotations and origins of list tails when an element in the list was transformed. This bug has been fixed.

The Stratego editor used to give spurious errors on missing variable definitions if those were list variables that were bound in a concrete syntax pattern. This long-standing bug has been fixed in this release.

24.15.2 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- [Windows 32-bits, embedded JRE](#)
- [Windows 64-bits, embedded JRE](#)
- [Linux 32-bits, embedded JRE](#)
- [Linux 64-bits, embedded JRE](#)
- [macOS, embedded JRE](#)

Without embedded JRE:

- [Windows 32-bits](#)
- [Windows 64-bits](#)
- [Linux 32-bits](#)
- [Linux 64-bits](#)
- [macOS](#)

Update site

- Eclipse update site: <http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.2/org.metaborg.spoofax.eclipse.update.site-2.5.2-assembly.zip-unzip/>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.2/org.metaborg.intellij.dist-2.5.2.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

Core API

- [Spoofax Core uber JAR](#)
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.2`

StrategoXT

- StrategoXT distribution
- StrategoXT JAR

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.2. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.16 Spoofox 2.5.1

Spoofox 2.5.1 is a minor bugfix release.

24.16.1 Changes

Core

- Fix: ([Spoofox/242](#)) `StrategoMix.def` not found error, after incrementally building a language specification project with a Stratego mix grammar.

24.16.2 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- Windows 32-bits, embedded JRE
- Windows 64-bits, embedded JRE
- Linux 32-bits, embedded JRE
- Linux 64-bits, embedded JRE
- macOS, embedded JRE

Without embedded JRE:

- Windows 32-bits
- Windows 64-bits
- Linux 32-bits
- Linux 64-bits
- macOS

Update site

- Eclipse update site: `http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.1/org.metaborg.spoofax.eclipse.update.site-2.5.1-assembly.zip-unzip/`
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: `http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.1/org.metaborg.intellij.dist-2.5.1.zip`
- [IntelliJ update site archive](#)

Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

Core API

- [Spoofax Core uber JAR](#)
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.1`

StrategoXT

- [StrategoXT distribution](#)
- [StrategoXT JAR](#)

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.1. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.17 Spoofox 2.5.0

Spoofax 2.5 introduces FlowSpec, a new meta-language for intra-procedural data-flow analysis; layout-sensitive parsing in SDF3; and has several small improvements and bug fixes.

24.17.1 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- [Windows 32-bits, embedded JRE](#)
- [Windows 64-bits, embedded JRE](#)
- [Linux 32-bits, embedded JRE](#)
- [Linux 64-bits, embedded JRE](#)
- [macOS, embedded JRE](#)

Without embedded JRE:

- [Windows 32-bits](#)
- [Windows 64-bits](#)
- [Linux 32-bits](#)
- [Linux 64-bits](#)
- [macOS](#)

Update site

- Eclipse update site: <http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.5.0/org.metaborg.spoofax.eclipse.update.site-2.5.0-assembly.zip-unzip/>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.5.0/org.metaborg.intellij.dist-2.5.0.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

Core API

- [Spoofax Core uber JAR](#)
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.5.0`

StrategoXT

- [StrategoXT distribution](#)
- [StrategoXT JAR](#)

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.5.0. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.17.2 Changes

Maven

We updated the Guice version that Spoofax uses to 4.2.0. This has the cascading effect that we need Maven 3.5.4, since Spoofax is used in a maven plugin. Be sure to have this version of Maven installed, or you will run into `MethodNotFoundExceptions` for `com.google.inject.internal.*`.

Core

The `constraint` analyzer was generalized:

- The `constraint` analyzer is now independent of NaBL2, and can be used as a generic analysis mechanism from Stratego. The analysis cycle and the Stratego interface to it are defined and documented in module `libspoofax/analysis/constraint`.
- Fixed a bug where ambiguity errors were not always correctly reported.

FlowSpec

This release introduces FlowSpec, a new meta-language for intra-procedural data-flow analysis. See the documentation of the language for more details.

Stratego

Stratego received some small patches to improve user experience.

- Stratego editor now gives warnings when using `left`, `right` and a couple of other variables names as these are also constructors in `libstratego-sgllr` and interpreted as constructor match patterns.
- When the Stratego compiler generates names for code generation, these now start with the source code name if available or with a constant name related to the language feature (e.g. a `where(s)` is turned into `?where243;s;!where243`). Since some generated names turn up in a stack trace from a Stratego, this should improve readability of the stack trace. Complex closures are still named `lifted26`, as the compiler cannot replace humans in properly naming things.

SPT

Problems related to escaping in string terms of expectations are fixed:

- Double quotes (") in expectation require escaping using backslashes, but were not unescaped when comparing with actual parse results. This made correct tests containing double quotes in strings fail. This is fixed by unescaping the expectation terms.
- Formatting expectation terms as strings was different than default ATerms (" and \ were not escaped), which was confusing when a test fails and the actual and expected terms were reported. This is fixed by aligning the SPT expectation terms formatting with default ATerm formatting.

NaBL2

Small usability improvements:

- Empty parameter tuples in rules can be omitted.
- Accidentally writing a dot instead of a comma before a recursive rule invocation could make that constraint look like a rule without constraints. Layout is now used to give a warning when such a case is written.
- Fix import problems caused by `nabl2.runtime` exports. The exports are restricted such that layout syntax and DynSem signatures are not exported anymore. The sorts defined by the runtime are all prefixed with `NaBL2` to prevent accidental merges with sorts from the importing language.
- Allow all Stratego identifiers to be used as constructor names.

Solver changes:

- Adopt new naming convention, with packages named `mb.nabl2.*`, and artifacts named `nabl2.*`.
- Add classes for matching and substitution of terms, independent of unification.
- Use the generalized `constraint` analyzer for the NaBL2 analysis strategy.

SDF3

The experimental support for generating Scala case classes from an SDF3 specification was removed. It was incomplete, unmaintained and unused.

Added support for *Layout Declarations* for layout-sensitive parsing and pretty-printing.

Eclipse

Small fixes and improvements:

- Execute builders for languages which have no analysis defined. Previously builders would always wait until an analysis result was produced.
- Cancel running SPT test suites. It is now possible to cancel a running SPT test suite in the progress window.

IntelliJ

Small fixes and improvements:

- Can now be installed into any latest IntelliJ, not just the last version we tested
- By default runs in IntelliJ 2018.1.1

- Simplified project structure
- Updated dependencies
- Changes to support Java 9 in the future

24.18 Spoofax 2.4.1

Spoofax 2.4.1 is a minor bugfix release.

24.18.1 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- [Windows 32-bits, embedded JRE](#)
- [Windows 64-bits, embedded JRE](#)
- [Linux 32-bits, embedded JRE](#)
- [Linux 64-bits, embedded JRE](#)
- [macOS, embedded JRE](#)

Without embedded JRE:

- [Windows 32-bits](#)
- [Windows 64-bits](#)
- [Linux 32-bits](#)
- [Linux 64-bits](#)
- [macOS](#)

Update site

- Eclipse update site: <http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.4.1/org.metaborg.spoofax.eclipse.update.site-2.4.1-assembly.zip-unzip/>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.4.1/org.metaborg.intellij.dist-2.4.1.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

Core API

- [Spoofax Core uber JAR](#)
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.4.1`

StrategoXT

- [StrategoXT distribution](#)
- [StrategoXT JAR](#)

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.4.1. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.18.2 Changes

- Fix: remove dependency on `nativebundle` from `jsglr2`, preventing native binaries (with a cygwin vulnerability) from showing up in Spoofax Core.
- Update `jackson-core`, `jackson-databind`, `jackson-annotations`, `jackson-dataformat-yaml` dependencies to 2.9.3 to avoid a vulnerability in those libraries.
- Update `commons-configuration2` to 2.2, `commons-configuration2-jackson` to 0.7.0, and `snakeyaml` to 1.18, for compatibility with Jackson version 2.9.3.

24.19 Spoofax 2.4.0

Spoofax 2.4 fixes several bugs and includes a program generator.

24.19.1 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- [Windows 32-bits, embedded JRE](#)
- [Windows 64-bits, embedded JRE](#)
- [Linux 32-bits, embedded JRE](#)

- Linux 64-bits, embedded JRE
- macOS, embedded JRE

Without embedded JRE:

- Windows 32-bits
- Windows 64-bits
- Linux 32-bits
- Linux 64-bits
- macOS

Update site

- Eclipse update site: <http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.4.0/org.metaborg.spoofax.eclipse.update.site-2.4.0-assembly.zip-unzip/>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.4.0/org.metaborg.intellij.dist-2.4.0.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- Sunshine JAR
- SPT testrunner JAR

Core API

- Spoofax Core uber JAR
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.4.0`

StrategoXT

- StrategoXT distribution
- StrategoXT JAR

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.4.0. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.19.2 Changes

Eclipse Plugin

- Fix: re-parse and re-analyze open editors if the language is reloaded.

NaBL2

- Fix: use deep equality instead of object equality to compare elements in set constraints.
- Fix: prevent clashes of variable names with known lower-case Stratego constructors.
- Improvement: add strategies to the Stratego API to query references and declaration associated with AST nodes.
- Fix: prevent exception traces when hovering over the editor.
- Fix: bug in Stratego generation when complex terms are used in occurrences.
- Fix: bug where editor resolution would only consider leaf nodes, but not parents if the leafs do not resolve.
- Fix: bug where sometimes error messages of files were lost.

Parser

- Improvement: latest JSGLR2 performance optimizations.
- Fix: bug in JSGLR2 where non-default start symbols were not taken into account.

24.20 Spoofox 2.3.0

Spoofax 2.3 fixes several minor bugs, upgrades to the latest Eclipse and Java versions, includes improvements to SDF3 and NaBL2, and introduces experimental parse table generation and parsing features.

24.20.1 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- Windows 32-bits, embedded JRE
- Windows 64-bits, embedded JRE
- Linux 32-bits, embedded JRE
- Linux 64-bits, embedded JRE
- macOS, embedded JRE

Without embedded JRE:

- Windows 32-bits
- Windows 64-bits

- [Linux 32-bits](#)
- [Linux 64-bits](#)
- [macOS](#)

Update site

- Eclipse update site: <http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.3.0/org.metaborg.spoofax.eclipse.update.site-2.3.0-assembly.zip-unzip/>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.3.0/org.metaborg.intellij.dist-2.3.0.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

Core API

- [Spoofax Core uber JAR](#)
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.3.0`

StrategoXT

- [StrategoXT distribution](#)
- [StrategoXT JAR](#)

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.3.0. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.20.2 Changes

Overall

- Improvement: made NaBL2 the default static semantics language.

- Improvement: put deprecated markers on NaBL+TS and Stratego as static semantics languages, and SDF2 as syntax language.
- Improvement: allow configuration of source folders in metaborg.yaml.
- Improvement: allow multiple languages in source and export entries.
- Improvement: add dynsem as a compile dependency to newly generated languages.

Language specification build

- Fix: occasional NPEs when the build failed.
- Fix: hidden dependency error when building Stratego concrete syntax extensions.

Eclipse plugin

- Improvement: updated Eclipse instance generator to generate Eclipse Oxygen instances.
- Improvement: updated Eclipse instance generator to include JDK 8u144b01.
- Improvement: do not reanalyze already analyzed files when opening an editor.
- Improvement: use a default configuration if metaborg.yaml is not present.

NaBL2

- Improvement: extended Stratego API to query reference resolution.
- Improvement: add ? and + operators to regexp syntax for path well-formedness.
- Fix: regexp normalization was only one level deep.
- Fix: non-termination in name resolution in the cases of a direct cycle between a scope.
- Update: conform to latest DynSem version.
- Fix: support all Stratego constructor and sort names, by allowing dashes and single quotes in sort and constructor names.
- Fix: do not crash if dynsem properties file is missing.

SDF3

- Improvement: more stable SDF3 parser generator.
- Improvement: new parenthesizer that considers deep priority conflicts.
- Improvement: (experimental) support for lazy parse table generation, where the parse table is generated on-the-fly by the parser.
- Fix: bug in the SDF3 normalizer for groups of priorities outside of a chain.
- Fix: added support for generating the parse table from a “permissive” grammar
- Fix: not necessary to specify the parse table as `sdf-new.tbl` in the ESV file when using the new parse table generator.

Parser

- Added the new (experimental) SGLR parser implementation JSGLR2.

24.21 Spoofox 2.2.1

Spoofax 2.2.1 is a minor bugfix release.

24.21.1 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- Windows 32-bits, embedded JRE
- Windows 64-bits, embedded JRE
- Linux 32-bits, embedded JRE
- Linux 64-bits, embedded JRE
- macOS, embedded JRE

Without embedded JRE:

- Windows 32-bits
- Windows 64-bits
- Linux 32-bits
- Linux 64-bits
- macOS

Update site

- Eclipse update site: <http://artifacts.metaborg.org/content/unzip/releases-unzipped/org/metaborg/org.metaborg.spoofax.eclipse.update.site/2.2.1/org.metaborg.spoofax.eclipse.update.site-2.2.1-assembly.zip-unzip/>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://artifacts.metaborg.org/content/repositories/releases/org/metaborg/org.metaborg.intellij.dist/2.2.1/org.metaborg.intellij.dist-2.2.1.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- Sunshine JAR
- SPT testrunner JAR

Core API

- Spoofox Core uber JAR
- Spoofox Core uber Maven artifact: `org.metaborg:org.metaborg.spoofox.core.uber:2.2.1`

StrategoXT

- StrategoXT distribution
- StrategoXT JAR

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.2.1. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.21.2 Changes

Overall

- Fix: error when generating projects with analysis enabled (which is enabled by default).
- Fix: possibly erroneous completions file in spoofax meta library.

24.22 Spoofox 2.2.0

Spoofax 2.2 improves on Spoofox 2.1 with a new NaBL2 constraint solver which is optimised for performance, improved progress reporting and cancellation in Eclipse, an experimental replacement for `sdf2table` which fixes several long-standing bugs, improvements to the core API, and several bug fixes.

See the corresponding *migration guide* for migrating from Spoofox 2.1 to Spoofox 2.2.

24.22.1 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- Windows 32-bits, embedded JRE
- Windows 64-bits, embedded JRE
- Linux 32-bits, embedded JRE

- Linux 64-bits, embedded JRE
- macOS, embedded JRE

Without embedded JRE:

- Windows 32-bits
- Windows 64-bits
- Linux 32-bits
- Linux 64-bits
- macOS

Update site

- Eclipse update site: <http://tinyurl.com/spoofax-eclipse-2-2-0>
- Eclipse update site archive

IntelliJ plugin

- IntelliJ update site: <http://tinyurl.com/spoofax-intellij-2-2-0>
- IntelliJ update site archive

Command-line utilities

- Sunshine JAR
- SPT testrunner JAR

Core API

- Spoofax Core uber JAR
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.2.0`

StrategoXT

- StrategoXT distribution
- StrategoXT JAR

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.2.0. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.22.2 Changes

Overall

- The deprecated libraries and files from Spoofox 2.1.0 have been removed. If you have not done so yet, follow the *Spoofox 2.1.0 migration guide* to migrate your project to the new Spoofox library.

Core API

- Improve: [Spoofox/190 - Extend API for language discovery](#). This deprecates several methods in the language discovery API, see the *migration guide* on how to migrate your code.
- Improve: [Spoofox/193 - Stratego warnings in Spoofox language projects with NaBL2 analysis](#). The excessive number of warnings from Stratego compilation are now filtered out.
- Improve: Parsing and analysis can report progress and be cancelled.
- Improve: Builds now report progress.
- Fix: Path and project path that are passed to the editor hover strategy are now consistent with paths passed to other strategies.
- Fix: [Spoofox/187 - Provide simplified builder API](#).
- Fix: [Spoofox/188 - Java type error in documented language processing code](#).

Eclipse

- Upgrade: Eclipse Neon (4.6) is now required.
- Improve: Added several switches to the *Spoofox (meta)* menu for disabling analyses and builds, to improve usability in cases where these operations are very slow.
- Improve: Bind new progress reporting and cancellation in core to Eclipse progress monitors, enabling reporting of builds and cancellation of analysis.
- Fix: Fix cancellation not being propagated in SubMonitors, preventing cancellation from working in many places.

SDF3

- Feature: Re-implemented the parse table generator in Java, removing the dependency on a platform-specific `sdf2table` binary, and fixing several long-standing bugs. This implementation is still being tested, it is therefore only enabled after opt-in. To enable the new implementation, set the following option in your `metaborg.yaml` file:

```
language:
  sdf:
    sdf2table: java
```

- Improve: Moved the `placeholder` and `pretty-print` options in the `metaborg.yaml` file to be under `language.sdf`, as in:

```
language:
  sdf:
    placeholder:
```

(continues on next page)

(continued from previous page)

```
prefix: "[["  
suffix: "]]"  
pretty-print: LangName
```

NaBL2

- Improve: Introduces a new solver implementation with improved performance.
- Improve: Introduces separate signature sections for *constructors*, *relations*, and *functions*.
- Deprecate: The *types* signature, which will be removed in the next release.

SPT

- Fix: Several origin tracking issues related to section markers.

DynSem

- Fix: Analysis crashes on empty **rules** sections (#161)
- Improve: Support for abrupt termination: automatic expansion and propagation of read-write semantic components with default values
- Improve: Analysis performance improvements

24.23 Spoofax 2.1.0

Spoofax 2.1 improves on Spoofax 2.0 with several bug fixes, an implementation of syntactic completions based on SDF3, and addition of the DynSem dynamic semantics specification meta-language.

See the corresponding [migration guide](#) for migrating from Spoofax 2.0 to Spoofax 2.1.

24.23.1 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- Windows 32-bits, embedded JRE
- Windows 64-bits, embedded JRE
- Linux 32-bits, embedded JRE
- Linux 64-bits, embedded JRE
- macOS, embedded JRE

Without embedded JRE:

- Windows 32-bits

- Windows 64-bits
- Linux 32-bits
- Linux 64-bits
- macOS

Update site

- Eclipse update site: <http://tinyurl.com/spoofax-eclipse-2-1-0-fixed>
- Eclipse update site archive

IntelliJ plugin

- IntelliJ update site: <http://tinyurl.com/spoofax-intellij-2-1-0>
- IntelliJ update site archive

Command-line utilities

- Sunshine JAR
- SPT testrunner JAR

Core API

- Spoofax Core uber JAR
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.1.0`

StrategoXT

- StrategoXT distribution
- StrategoXT JAR

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.1.0. See the instructions on *using MetaBorg Maven artifacts* for more information.

24.23.2 Changes

Syntactic Completions

Spoofax now has support for syntactic completions. Syntactic completions are generated automatically from an SDF3 specification. New projects using SDF3 automatically support syntactic completions. Existing projects need to make a few changes, documented in the *migration guide*.

DynSem

DynSem is a DSL for concise and modular specification of dynamic semantics of programming languages. Fully functional interpreters are automatically derived from dynamic semantics specifications. For more information about DynSem, see the following sources:

- [Paper](#)
- [Documentation](#)
- [Getting started tutorial](#)
- [Example language](#)

While DynSem was included in Spoofax 2.0.0, we did not advertise this as it was still under heavy development. Since 2.0.0, the following major improvements were made:

1. [Redesigned semantic component and explication subsystem](#)
2. [Support for tuples](#)
3. [Updated tutorial for SIMPL](#)
4. [Added support for unit testing and continuous integration of generated interpreters](#)

24.24 Spoofax 2.0.0

Spoofax 2.0 is a complete rewrite of Spoofax which improves the architecture by separating Spoofax into the Spoofax Core API and implementations on top of that API, massively improves the language development workflow, and properly supports language extension.

See the corresponding [migration guide](#) for migrating from Spoofax 1.5 to Spoofax 2.0.

24.24.1 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- [Windows 32-bits, embedded JRE](#)
- [Windows 64-bits, embedded JRE](#)
- [Linux 32-bits, embedded JRE](#)
- [Linux 64-bits, embedded JRE](#)
- [Mac OS X \(Intel only\), embedded JRE](#)

Without embedded JRE:

- [Windows 32-bits](#)
- [Windows 64-bits](#)
- [Linux 32-bits](#)
- [Linux 64-bits](#)

- [Mac OS X \(Intel only\)](#)

Update site

- Eclipse update site: <http://download.spoofax.org/update/release/2.0.0/eclipse/site>
- [Eclipse update site archive](#)

IntelliJ plugin

- IntelliJ update site: <http://download.spoofax.org/update/release/2.0.0/intellij/spoofax-intellij-updatesite-2.0.0.zip>
- [IntelliJ update site archive](#)

Command-line utilities

- [Sunshine JAR](#)
- [SPT testrunner JAR](#)

Core API

- [Spoofax Core uber JAR](#)
- Spoofax Core uber Maven artifact: `org.metaborg:org.metaborg.spoofax.core.uber:2.0.0`

StrategoXT

- [StrategoXT distribution](#)
- [StrategoXT JAR](#)

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 2.0.0. See the instructions on [using MetaBorg Maven artifacts](#) for more information.

24.24.2 Known Issues

- Stratego imports do not work. To work around this issue, add an explicit compile dependency to Stratego:

```
dependencies:
  compile:
    - org.metaborg:org.metaborg.meta.lang.stratego:${metaborgVersion}
```

24.24.3 Changes

Architecture

The biggest change in Spoofax 2.0 is the architecture. Previously, Spoofax was built on top of the Eclipse and IMP platform, meaning Spoofax was not usable outside of the Eclipse platform. In Spoofax 2.0, all platform-agnostic functionality such as language management, parsing, analysis, transformation, and editor services, are implemented in Spoofax Core, which is a portable Java library with an API. This means that the Spoofax language workbench, and any language implementations made with Spoofax, can now be used by any application, platform, or framework in the Java ecosystem.

Integrations

We have integrated Spoofax Core with Eclipse, IntelliJ, Maven, and the command-line.

We support the Eclipse platform through a new plugin that integrates Spoofax Core as an Eclipse plugin. The new Eclipse plugin supports language development in Eclipse, and supports exporting languages made with Spoofax as an Eclipse plugin with full-fledged editor support. We have also performed a more faithful Eclipse integration than Spoofax 1.5 did. For example, we now use natures to enable Spoofax for a project, use the incremental builder framework to allow suspending automatic builds, and use Eclipse's menu system for builders instead of non-standard buttons. See the [migration guide](#) for a full list of changes to the Eclipse plugin.

IntelliJ is an experimentally supported platform through the Eclipse IntelliJ plugin. Languages can be developed in IntelliJ, and exported as IntelliJ plugins with full-fledged editor support.

The Spoofax Maven plugin supports command-line builds and continuous integration of language implementations in Maven. Language implementations can be exported as Maven artifacts which can be depended on and used to build programs of that language.

Command-line use of language implementations is supported through Sunshine's integration with Spoofax Core. Sunshine's command-line interface has been simplified to improve ease of use, and now also supports a server mode to reduce the high cost of starting a new JVM and loading a language.

Furthermore, anyone can make new integrations using the Core API.

Language Development Workflow

There are several improvements to the language development workflow in Spoofax 2.0.

Almost all generated files are now generated to the src-gen directory of a language project. All required generated files are now (re)generated when building, so it is no longer necessary to commit generated files to source control. This results in much cleaner projects. Furthermore, the language build is now incremental, which speeds up the build in many cases.

The bootstrapping process of meta-languages has been significantly improved by versioning languages. It is now possible to load multiple versions of the same language implementation into Spoofax. Meta-languages are bootstrapped by building them against baseline versions of the meta-languages. When a meta-language under development breaks, it is possible to revert back to a previous version to get things working again.

Extension

Spoofax 2.0 supports language extension on the source level, without the need to copy-paste files around. A dependency can be made from a language specification to another language specification, which then allows importing modules of the specification into the other. For example, language extensions can depend on a base language and

extend its concepts. Those extensions can be composed together with the base language specification into a new language specification that contains the base and extensions.

There is also limited support for dynamic extension, i.e. extension at the runtime level instead of the source level. A language implementation can be extended with new builders at runtime. This allows adding builders to existing language implementations, and supports separating the front-end and back-end of a language into multiple projects.

License

The license has been changed from LGPLv2.1 to the Apache 2.0 license, to improve adoption of Spoofox. Any contributions made to Spoofox must be licensed under the Apache 2.0 license as well.

Missing Features

A few features didn't make it to Spoofox 2.0, with the biggest one being semantic completions.

Semantic completions were already very dodgy in Spoofox 1.5, only working in some specific cases. This is why we did not port the completion algorithm from Spoofox 1.5 to 2.0, and are instead working on a new completion algorithm that will be included in a future version.

Refactorings were already broken in Spoofox 1.5, so we did not port refactorings to Spoofox 2.0. In the future we will revisit refactorings for Spoofox 2.0 with our new meta-languages.

The Spoofox modelware component was not ported to Spoofox 2.0 since we do not have the knowledge to port this component.

Folding, realtime builders, and the eclipse.ini check are minor features that are not implemented in 2.0, but may be implemented in the future.

A missing integration in Spoofox 2.0 is a Spoofox Gradle plugin, we are working on that integration for inclusion in a future version.

24.25 Spoofox 2.0.0-beta1 (07-04-2016)

This is the first beta release of Spoofox 2.0. These notes provide the download links for the various artifacts.

See the [2.0.0 release notes](#) for more information about Spoofox 2.0. See the [2.0.0 migration guide](#) for migrating a Spoofox 1.5 project to a Spoofox 2.0 project.

Follow the [getting started guide](#) to get started with Spoofox, including installation instructions.

24.25.1 Downloads

Eclipse plugin

Premade Eclipse installations

With embedded JRE:

- [Windows 32-bits, embedded JRE](#)
- [Windows 64-bits, embedded JRE](#)
- [Linux 32-bits, embedded JRE](#)

- Linux 64-bits, embedded JRE
- Mac OS X (Intel only), embedded JRE

Without embedded JRE:

- Windows 32-bits
- Windows 64-bits
- Linux 32-bits
- Linux 64-bits
- Mac OS X (Intel only)

Update site

- Eclipse update site: <http://download.spoofax.org/update/release/2.0.0-beta1/eclipse/site>
- Eclipse update site archive

Sunshine

Sunshine JAR

SPT command-line test runner

SPT testrunner JAR

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is `2.0.0-beta1`. See the instructions on [using MetaBorg Maven artifacts](#) for more information.

24.26 Spoofox 1.5.0 (18-12-2015)

We're happy to announce the release of Spoofox 1.5.0 with new SDF3 features and fixes, and support for Eclipse Mars.

24.26.1 Changes

SDF3

- Feature: [support for case insensitive keywords](#). All keywords in a template production are case insensitive if the production has the attribute `case-insensitive`.
- Feature: pretty-print ambiguous programs (by taking the first alternative).
- Feature: give an error if the filename does not match the module name.
- Fix: [ESV generation with empty imports](#).
- Fix: disallow empty placeholders `<>` in template productions.

Contributor: Eduardo Amorim

Eclipse

- Feature: support for Eclipse Mars.
- Feature: generation of premade Eclipse installations with Spoofax installed.

Contributor: Gabriel Konat

Command-line tools

- Fix: Sunshine now pretty-prints ATerms before presenting them, mimicking the behavior in Eclipse.

Contributor: Gabriel Konat

24.26.2 Downloads

Eclipse plugin

Premade Eclipse installations

New to this release are Eclipse Mars installations with Spoofax preinstalled and correct eclipse.ini settings. They come in a version which uses the Java 7 Runtime Environment (JRE) on your system, and a version with an embedded JRE7. See the [getting started guide](#) for instructions on how to get started with Spoofax.

Download a premade Eclipse installation for your platform, with embedded JRE7:

- [Windows 32-bits, embedded JRE7](#)
- [Windows 64-bits, embedded JRE7](#)
- [Linux 32-bits, embedded JRE7](#)
- [Linux 64-bits, embedded JRE7](#)
- [Mac OS X \(Intel only\), embedded JRE7](#)

Without embedded JRE:

- [Windows 32-bits](#)
- [Windows 64-bits](#)
- [Linux 32-bits](#)
- [Linux 64-bits](#)
- [Mac OS X \(Intel only\)](#)

Update site

The latest stable version of the Spoofax Eclipse plugin is always published to the `stable` update site: <http://download.spoofax.org/update/stable>. See the [getting started guide](#) for instructions on how to get started with Spoofax.

This specific release is also published to the `release/1.5.0` update site: <http://download.spoofax.org/update/release/1.5.0>.

If you'd like to update from a nightly version to this version, you must uninstall the nightly version and restart Eclipse before installing this version, to avoid version conflicts.

Sunshine JAR

Sunshine is the command-line tool for Spoofax that validates and transforms programs of Spoofax language implementations. The Sunshine JAR file corresponding with this release can be [downloaded here](#). See the [Sunshine documentation](#) for more information on using Sunshine.

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is 1.5.0. See the instructions on [using MetaBorg Maven artifacts](#) for more information.

24.27 Spoofax 1.4.0 (06-03-2015)

We're happy to announce the release of Spoofax 1.4.0, a minor release with SDF3 fixes and improvements to language plugins.

24.27.1 Changes

SDF3

- Fix: [Supporting Windows line endings in SDF3](#).
- Fix: [Providing warnings for literals that could be placeholders](#).

Contributors: Eduardo Amorim

Language plugins

Reduced download size of deployed plugins

Previously, when creating an Eclipse update site for your language (see [tutorial](#)), the result was a ~90MB download that included meta-tools such as SDF3 and NaBL. We brought the download size down to ~60MB by removing dependencies to some of the meta-tools, since end-users of deployed languages don't need them. In the future, we plan to bring the size further down by removing more dependencies.

Contributors: Oskar van Rest

Setting Java VM options for your language

For Spoofax and Spoofax-based languages to run smoothly, it is recommended to set Java's `-server` flag and to increase the stack size and memory allocation pool:

- `-Xss<size>` specifies the thread stack size
- `-Xmx<size>` specifies the maximum size, in bytes, of the memory allocation pool.
- `-server` selects server application runtime optimizations.

The server VM will take longer to start and “warm up” but will be more aggressively optimized. The `-server` option only affects 32-bit VMs and has influence on 64-bit VMs because these always use server optimizations. These options can be configured in `eclipse.ini` as described on the [download](#) page. The recommended settings for Spoofax are `-server -Xss8m -Xmx1024m` and a warning will pop-up if Eclipse is started with settings that are too low.

Previously, the same settings were assumed for deployed plugins and were enforced by a similar pup-up warning. With Spoofax 1.4.0, language developers can choose their own Java VM settings, which are then recommended to end-users of their language. This can be configured in `editor/yourlang.main.esv`. The syntax is as follows:

```
jvm opts: [-server | -X[ss|mx]<size>[g|G|m|M|k|K]]+
```

For example: `jvm opts: -server -Xss8m -Xmx1024m`. If multiple Spoofax-based languages are installed, the configuration warning will tell how `eclipse.ini` needs to be updated such that the requirements of all languages are satisfied.

Contributors: Oskar van Rest

24.27.2 Downloads

Eclipse plugin update site

The latest stable version of the Spoofax Eclipse plugin is always published to the stable update site: <http://download.spoofax.org/update/stable>. See the [getting started guide](#) for instructions on how to get started with Spoofax.

This specific release is also published to the `release/1.4.0` update site: <http://download.spoofax.org/update/release/1.4.0>.

Sunshine JAR

The Sunshine JAR file corresponding with this release can be [downloaded here](#). See the [Sunshine documentation](#) for more information on using Sunshine.

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is `1.4.0`. See the instructions on [using MetaBorg Maven artifacts](#) for more information.

24.28 Spoofax 1.3.1 (09-12-2014)

We’re happy to announce the release of Spoofax 1.3.1, a maintenance release which fixes several issues in SDF3, and compatibility with Eclipse 4.3 (Kepler).

24.28.1 Changes

- Fix: [Allowing sequences in SDF3 lexical syntax](#).
- Fix: [Allowing specific arguments of productions in SDF3 priorities](#).
- Fix: SDF3 outliner not working.
- Fix: [Unable to install in Eclipse 4.3](#).

Contributors: Eduardo Amorim

24.28.2 Downloads

Eclipse plugin update site

The latest stable version of the Spoofax Eclipse plugin is always published to the `stable` update site: `http://download.spoofax.org/update/stable`. See the [getting started guide](#) for instructions on how to get started with Spoofax.

This specific release is also published to the `release/1.3.1` update site: `http://download.spoofax.org/update/release/1.3.1`.

Sunshine JAR

The Sunshine JAR file corresponding with this release can be [downloaded here](#). See the [Sunshine documentation](#) for more information on using Sunshine.

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is `1.3.1`. See the instructions on [using MetaBorg Maven artifacts](#) for more information.

24.29 Spoofax 1.3.0 (12-11-2014)

We're happy to announce the release of Spoofax 1.3, which improves SDF3, the build system for languages, and the build system for Spoofax itself.

24.29.1 Porting

Language projects

The build for language projects has changed since the last Spoofax, so your language projects need to be migrated. To automatically migrate your project, follow these steps:

1. Right click the project in Eclipse
2. Choose Spoofax -> Upgrade Spoofax project
3. Press Finish

This will automatically upgrade your project to the latest format, after which you can build your project normally. The build might fail with:

```
sdf2parenthesize:

BUILD FAILED
/Users/gohla/spoofax/workspace/runtime-Spoofax/Entity/build.generated.xml:266: The_
↳ following error occurred while executing this line:
Target "sdf2parenthesize.helper" does not exist in the project "Entity".
```

If this is the case, build again, because some files are only upgraded after building once.

Compiled Java files are now stored in the `target/classes` directory instead of the `bin` directory, so the `bin` directory can be deleted.

SDF3 grammars

SDF2 and old SDF3 grammars can be migrated following these instructions: [How to migrate old SDF grammars into latest SDF3?](#).

Some manual changes might still be necessary since a sort cannot have more than one constructor with the same name and arity, and priority rules may contain the entire production instead of priority shorthands.

24.29.2 Changes

SDF3

Several improvements have been made to increase the consistency and robustness of SDF3.

- Regular productive productions can be mixed with template productions in the context-free syntax section.
- Each production defines a non-unique sort and a unique constructor of that sort. References point to these definitions and errors are given for undefined elements in the grammar.
- Signatures are generated on-save following the sorts used in the right-hand side of a production and the sort and constructor that are being defined.
- All code generated from SDF3 grammars is organized in the `src-gen` directory of the project, which keeps Spoofax projects more clean and structured.

Contributors: Eduardo Amorim

Building language projects

Projects are built using Ant Macros tailored to make the build system more incremental.

Language projects can now also be built on the command-line using Maven. See the instructions on [building Spoofax languages](#) for more information.

Contributors: Eduardo Amorim, Gabriël Konat

Building and developing Spoofax

The build system for Spoofax itself has been refactored from a Nix build into a full Maven build. This enables local builds on any system that Maven supports, which is basically any system that supports Java. See the instructions on [building Spoofax](#) for more information. the instructions on [using MetaBorg Maven artifacts](#) for more information. There is now also documentation on [Setting up Eclipse for Spoofax development](#), which explains how to set up an environment for developing projects which are included in Spoofax.

The nightly version of Spoofax is now built on our Jenkins server: <http://buildfarm.metaborg.org/>. It publishes artifacts to our artifact server: <http://artifacts.metaborg.org/>. To use these artifacts, read the instructions on the instructions on [using MetaBorg Maven artifacts](#) for more information.

Contributors: Gabriël Konat, Danny Groenewegen, Elmer van Chastelet

Other

More changes, additions, and bug fixes can be found in the roadmap on our issue tracker: <http://yellowgrass.org/tag/Spoofax/1.3>

24.29.3 Downloads

Eclipse plugin update site

The latest stable version of the Spoofax Eclipse plugin is always published to the `stable` update site: <http://download.spoofax.org/update/stable>. See the [getting started guide](#) for instructions on how to get started with Spoofax.

This specific release is also published to the `release/1.3.0` update site: <http://download.spoofax.org/update/release/1.3.0>.

Sunshine JAR

The Sunshine JAR file corresponding with this release can be [downloaded here](#). See the [Sunshine documentation](#) for more information on using Sunshine.

Maven artifacts

Maven artifacts can be found on our [artifact server](#). The Maven version used for this release is `1.3.0`. See the instructions on [using MetaBorg Maven artifacts](#) for more information.

24.30 Spoofax 1.2.0 (13-08-2014)

We're happy to announce the release of Spoofax 1.2! This document describes the changes in Spoofax 1.2 since Spoofax 1.1.

24.30.1 Changes

Editor interface

Several aspects of the editor interface for Spoofax languages have been improved.

Menus

Each language now has its own set of menus. These menus replace the `Transform` menu that was shared among all Spoofax-based languages.

- The new menus dynamically pop up in the Eclipse menus toolbar, based on the current active editor.
- There is now support for submenus, icons and menu separators.

Contributors: Oskar van Rest

Outline

Editor outlines are now specified in Stratego instead of ESV, to allow for full customization.

- A library with reusable outline strategies is provided to allow you to quickly realize an outline.
- It is now possible to have icons in your outline.
- It is now possible to also base an outline on the current text selection instead of the complete text.

Contributors: Oskar van Rest

Properties view

A property view has been added that shows properties for the selected AST node.

- By default, the new properties view integrates with NaBL and presents (NaBL) properties associated with the selected text in the editor.
- The properties view can be customized to show different kinds of properties, either to aid language development or to provide language users with additional information about their programs.

Contributors: Daco Harkes, Oskar van Rest

Meta-languages

We have created a new version of SDF, improved NaBL and developed a DSL for describing type systems: TS.

SDF3

SDF3 is the next iteration of SDF, replacing SDF2. The most important new features are:

- Productive productions can be used in context-free syntax sections, improving readability and consistency with main-stream grammar notation.
- Constructors are now formally integrated to the language. A productive production introduces the constructor directly after the left hand side non-terminal.
- By using constructors, productions can now be uniquely identified. Therefore, it is no longer necessary to repeat the entire production in the priorities section, but use its `Sort.Constructor` shorthand.
- Template productions are the major change from SDF2. They can be used to define what the concrete syntax of the language should look like. Syntactic completion and pretty-printer rules are automatically generated from templates.

Documentation

Contributors: Eduardo Amorim, Guido Wachsmuth

NaBL

NaBL has received many bug fixes and several new features. New features include:

- Filter clauses can be used to constrain valid resolution targets based on properties such as types.
- Disambiguation clauses can be used to disambiguate resolutions based on relations between properties, for example type hierarchies.

- It is now possible to specify non-transitive scopes, in which resolution ignores lexical parent scopes.
- Where clauses can include constraints and type calculations in TS syntax.
- We added new scope calculation constructs, which can be used to navigate the scope hierarchy. For example, it is possible to calculate the surrounding class of the current variable scope.

Documentation

For examples of name binding rules, see the [Java front project](#)

Contributors: Guido Wachsmuth, Gabriël Konat

TS

TS is a new meta-language for the specification of type analysis that is complementary to NaBL. Type rules in TS define constraints on abstract syntax tree nodes and may compute a type or other property. In addition, type rules can define subtyping predicates (relations on types) and type functions.

Documentation

For examples of type system rules, see the [Java front project](#)

Contributors: Eelco Visser, Guido Wachsmuth, Gabriël Konat

Command-line integration

Programs of your language can now be parsed, analyzed, and transformed from the command-line using Sunshine (in contrast with an Eclipse). Sunshine can also be used as a Java API to develop new language tooling.

Documentation

Contributors: Vlad Vergu

Finer-grained incrementality

Incrementality in the previous version of Spoofax was based on files. Any file that changed, and any dependent files would be reparsed and reanalysed completely. In the new version of Spoofax, there is more fine-grained dependency tracking which allows more fine-grained incrementality. If a file changes, that file is reparsed, but only affected computations are recomputed, and other files are never reparsed. Name and type computations which are described in NaBL and TS are incrementally executed. Incrementality is powered by a task engine, described in [our paper](#).

Contributors: Gabriël Konat, Guido Wachsmuth

Modelware

Spoofax Modelware is a new Spoofax component that provides integration with the [Eclipse Modeling Framework \(EMF\)](#) and the [Graphical Modeling Framework \(GMF\)](#) to allow for real-time synchronized textual and graphical editors and/or views. It also allows you to use other EMF-based tooling in combination with Spoofax.

Documentation

Contributors: Oskar van Rest

Documentation

We have moved most of our documentation to the [doc repository on GitHub](#). We're still in the process of moving over other documentation and writing more documentation.

There are also two new tutorials available:

- [Questionnaire language tutorial](#): learn to create a questionnaire language. This tutorial was given in a hands-on session at the Code Generation conference in 2014.
- [Compiler Construction lab assignments](#): a more in-depth tutorial. These are the assignments from our Compiler Construction lab where we teach students to create MiniJava inside Spoofox.

Contributors: Guido Wachsmuth and others

Other

To reduce maintenance effort, we have dropped support for Eclipse 3.7 (Indigo) and lower. We now support Eclipse 4.2 (Juno), 4.3 (Kepler), and 4.4 (Luna). We recommend you to [download and install Eclipse 4.4 \(Luna\) for Java Developers](#).

We have also dropped support for Java 5 and 6. Java 7 and 8 are supported, we recommend you to [download and install Java 8](#). Note that on OSX, Java 6 is installed by default which is not enough to run Spoofox, install Java 8 from the previous link.

All source code has been moved to our [organization on GitHub](#). Feel free to fork our code and send pull requests for patches or improvements!

More changes, additions, and bug fixes can be found in the roadmap on our issue tracker: <http://yellowgrass.org/tag/Spoofax/1.2>

Contributors: Vlad Vergu, Gabriël Konat

24.30.2 Downloads

Eclipse plugin update site

The latest stable version of the Spoofox Eclipse plugin is always published to the `stable` update site: `http://download.spoofax.org/update/stable`. See the [getting started guide](#) for instructions on how to get started with Spoofox.

This specific release is also published to the `release/1.2.0` update site: `http://download.spoofax.org/update/release/1.2.0`.

24.31 Spoofox 1.1.0 (25-03-2013)

We are happy to announce the release of Spoofox 1.1! This is the first major release since version 1.0.2 and includes major features and improvements. Spoofox 1.1 supports all current Eclipse versions, up to version 4.2.2.

24.31.1 Changes

NaBL

One of the most important improvements in Spoofax 1.1 is the inclusion of NaBL, the Spoofax Name Binding Language. NaBL is used in all new projects created and significantly simplifies name binding analysis, as well as any editor services that depend on it (e.g., code completion, reference resolving)

NaBL is documented at the following pages:

- [Tutorial](#)
- [Research paper](#)

Other

Other highlights of the 1.1 release include:

- Improved build process: generated files can be deleted, building & loading are separated, projects can be cleaned (<http://yellowgrass.org/issue/Spoofax/577>, <http://yellowgrass.org/issue/Spoofax/591>, <http://yellowgrass.org/issue/Spoofax/578>)
- Improved Stratego editor with multi-file reference resolving based on NaBL (<http://yellowgrass.org/issue/Spoofax/12>)
- Extended support for customizing refactoring UI (<http://yellowgrass.org/issue/Spoofax/440>)
- Automatic configuration of git/svn ignore settings (<http://yellowgrass.org/issue/Spoofax/573>)
- Added support loading for Java-based plugin dependencies, in case your plugin depends on some other plugin such as EMF (<http://yellowgrass.org/issue/Spoofax/322>)

And there were a number of notable changes under the hood:

- Much improved completion engine (<http://yellowgrass.org/issue/Spoofax/360>)
- We now show a nice warning if Eclipse is not configured with a proper stack and heap size (<http://yellowgrass.org/issue/Spoofax/86>)
- Files are now queued for re-analysis even if their editor is not open (<http://yellowgrass.org/issue/Spoofax/224>)

A comprehensive list of changes can be viewed at <http://yellowgrass.org/tag/Spoofax/1.1>.

24.31.2 Downloads

Eclipse plugin update site

The latest stable version of the Spoofax Eclipse plugin is always published to the stable update site: <http://download.spoofax.org/update/stable>. See the [getting started guide](#) for instructions on how to get started with Spoofax.

This specific release is also published to the release/1.1.0 update site: <http://download.spoofax.org/update/release/1.1.0>.

24.32 Spoofax 1.0.2 (15-02-2012)

Today we're releasing a minor maintenance release of Spoofax, version 1.0.2. This release fixes a memory leak that was introduced in the 1.0 release. There are no new features in this release, those will be included in the upcoming 1.1 release instead.

24.33 Spoofox 1.0.0 (28-12-2011)

We're pleased to announce the release of Spoofox 1.0.0. A number of significant new features have been added since the last stable release, a long list of bugs has been fixed, and various minor improvements were introduced.

Highlights of the release include:

- Support for [writing tests for language definitions](#)
- Support for defining refactorings
- Major improvements to content completion: [Spoofox/289](#), [Spoofox/357](#)
- Support for using rewrite rules to disambiguate syntax: [Spoofox/328](#)

In addition to these features, we're actively working on improving Spoofox with new features. In particular, we are now working on providing full support for debugging, on an interactive shell for Stratego and custom languages, and a new meta-language called SpoofoxLang to define languages in a more modular fashion.

A full list of feature requests and issues addressed in the new version is provided at <http://yellowgrass.org/tag/Spoofax/1.0>.

The following release notes are stubs:

24.34 Spoofox vNext

These are the release notes for Spoofox vNext.

!!! error "Update the new documentation *vnex.rst* file instead" We have migrated to a [new website](<https://www.spoofax.dev/>), update the *vnex.md* in the [new documentation repository](<https://github.com/metaborg/metaborg.github.io/blob/main/docs/release/note/vnext.md>).

See the corresponding [migration guide](#) for migrating from Spoofox vPrev to Spoofox vNext.

24.34.1 Changes

This section contains the migration guides for releases and nightly builds of Spoofax.

25.1 Spoofax 2.5.15 Migration Guide

25.1.1 Statix Injection Explication

There was an issue with Statix injection explication where the origin of the top-level term was lost and this caused SPT tests of Stratego strategies on analyzed ASTs to fail. Fix this by wrapping the bodies of the `pre-analyze` and `post-analyze` strategies in `analyze.str` with `origin-track-forced`, like this:

```
imports libspoofox/term/origin

rules
  pre-analyze  = origin-track-forced(explicate-injections-MyLang-Start)
  post-analyze = origin-track-forced(implicate-injections-MyLang-Start)
```

This is already fixed in new projects.

25.2 Spoofax 2.5.10 Migration Guide

A change in Statix need migration for users of the Stratego API.

25.2.1 SDF3

In an upcoming version of Spoofax 2 it will be required to properly declare sorts in SDF3 syntax definitions. Sorts for which context-free rules are defined should be declared in a `context-free sorts` block:

```
context-free sorts
  Stmt Expr
```

Note: For backward compatibility, sorts declared in a plain `sorts` block are treated as context-free sorts. So this is equivalent and also fine:

```
sorts
  Stmt Expr
```

Sorts for which lexical rules are defined should be declared in a `lexical sorts` block:

```
lexical sorts
  ID INT STRING
```

Note: Lexical sorts are not supported in combination with `sdf2table: c`.

25.2.2 Typesmart

If your `metaborg.yaml` file still contains mention of Typesmart (e.g. `debug: typesmart: false`), you can remove it. See the release notes for why Typesmart support was removed.

25.2.3 Stratego

Spoofox languages used to always generate `target/metaborg/stratego-javastrat.jar` which contains the compiled Java code from `src/main/stratego`. Conditional on your settings in the `metaborg.yaml` file, your Stratego code would be turned into `target/metaborg/stratego.ctree` or `target/metaborg/stratego.jar` depending on whether you chose compilation or interpretation. As of this release, there is no longer a separate `stratego-javastrat.jar`. Instead `stratego.jar` is always generated and always contains at least the compiled Java code from `src/main/stratego`. If you choose compilation for your Stratego code, the compiled Stratego code is added to the `stratego.jar` file as was already the case originally.

What you need to do: Go to your `editor/main.esv` file and find the `provider: ...` lines (or search your other ESV files if it's not there). The line `provider: target/metaborg/stratego-javastrat.jar` should be replaced by `provider: target/metaborg/stratego.jar`. If you already have a `provider: target/metaborg/stratego.jar`, one is enough and you can remove the `stratego-javastrat.jar` provider directive entirely.

25.2.4 Statix

The AST property type is now a built-in property. Users of the Stratego API to get this property should change their API calls. Instead of

```
stx-get-ast-property(|a, "type")
```

one should now use:

```
stx-get-ast-type(|a)
```

25.2.5 SPT

In SPT, `parse succeeds` tests will now fail when the input parses ambiguously. If this is intended, use `parse ambiguous` instead.

25.3 Spoofax 2.5.5 Migration Guide

A few changes in Statix need migration for specs written for older versions.

25.3.1 Statix

A few features in Statix were not well-defined and removed until a better way to support them is found. Therefore, the following changes may require you to make small modifications to a specification:

1. Functional constraints can only have a single output.
2. Regular expression and label order must be direct parameters to queries.
3. Namespace based query short-hands require
 - a literal occurrence as argument, and
 - an explicit resolution policy entry.

Functional constraint outputs

Functional constraints previously were allowed to have multiple output arguments, such as:

```
f : T * T -> T * T`
```

This was sometimes indistinguishable from types with a single tuple output, such as:

```
f : T * T -> (T * T) `
```

From now on, functional constraints can only have a single output. Constraints with multiple outputs need to be rewritten to return a tuple instead.

Query parameters

Queries would accept arbitrary predicates for the label regular expression and label order. For example, a query could be written as:

```
query filter myWf and true
      min myOrd and false
      in s |-> _
```

with

```
myWf(ls) :- pathMatch[P*](ls).
myOrd(l1,l2) :- pathLt[$ < P](l1,l2).
```

This is not possible anymore, instead the regular expression and label order must be direct parameters of the query. The query above should be directly written as:

```
query filter P* and true
      min $ < P and false
      in s |-> _
```

The following syntax is still accepted, but deprecated:

```
query filter pathMatch[P*] and true
      min pathLt[$ < P] and false
      in s |-> _
```

Namespace-based resolution shorthands

The requirements for namespace-based resolution shorthands have become stricter.

First, if they are used, there must be an explicit resolution policy for the namespace. For example, using a constraint such as:

```
Var{x@-} in s |-> _
type of Var{x@-} in s |-> _
```

requires in the signature at least:

```
signature
  name-resolution
  resolve Var
```

A full resolution policy with regular expression and label order looks like this:

```
signature
  name-resolution
  resolve Var filter P* min $ < P
```

Second, the namespace-based constraints require an occurrence literal. The following does not work anymore:

```
d == Var{x@-},
d in s |-> _
```

The occurrence has to be repeated in the constraints, as:

```
Var{x@-} in s |-> _
```

25.4 Spoofax 2.2.0 Migration Guide

This migration guide describes how to migrate from Spoofax 2.1.0 to 2.2.0.

25.4.1 Overall

The deprecated libraries and files from Spoofax 2.1.0 have been removed. If you have not done so yet, follow the *Spoofax 2.1.0 migration guide* to migrate your project to the new Spoofax library.

25.4.2 Spoofax Core

The language discovery API was refactored into 2 separate types: a low-level interface for requesting creation of language components `ILanguageComponentFactory`, and a high-level interface `ILanguageDiscoveryService` that acts as a facade for loading language components and implementations.

The existing methods are still available, but are deprecated and will be removed after the next release. You should change your code to use the new methods. The deprecation documentation on the Java methods show which new methods to use.

25.4.3 Eclipse

Eclipse Neon (4.6) is now required. The Eclipse instances with integrated Spoofax we generate are already using Neon. However, if you manage your own Eclipse instance, you will need to upgrade to Neon.

25.4.4 SDF3

Move the `placeholder` and `pretty-print` options in the `metaborg.yaml` file to be under `language.sdf`, as in:

```
language:
  sdf:
    placeholder:
      prefix: "[["
      suffix: "]]"
    pretty-print: LangName
```

25.5 Spoofax 2.1.0 Migration Guide

This migration guide describes the differences between Spoofax 2.0 and 2.1 and how to convert to 2.1.

25.5.1 New Stratego library for Spoofax

Historically, the `org.metaborg.meta.lib.analysis` library (also called the *runtime libraries*) from [this repo](#), was first used as a library to support NaBL, TS, and task engine based static analysis. However, a lot of other functionality such as completions, refactoring, origin tracking, and annotation handling, was also added to the library for convenience. Likewise, the `src-gen/stratego/metaborg.str` generated file also contains arbitrary functionality such as parsing and import path primitives, and the `src-gen/editor/Colorer.generated` generated file contains a default coloring scheme.

Since this kind of functionality does not belong the analysis library and generated files, we have moved this into a new library, `libspoofax`, which can be found at [this repo](#).

Migration

The `org.metaborg.meta.lib.analysis` library still contains the old arbitrary functionality, but is now *deprecated*, meaning we will not update that functionality any more, and that it will be removed in a future version. Any functionality pertaining NaBL, TS, and task engine based static analysis will of course be retained. Likewise, the `src-gen/stratego/metaborg.str` and `src-gen/editor/Colorer.generated` generated files are also *deprecated*, meaning that they will stop being generated in a future version.

The new `libspoofax` library is now required for every Spoofax project. Add a source dependency to `org.metaborg:meta.lib.spoofax:${metaborgVersion}` in your `metaborg.yaml` file. For example, change the following dependencies:

dependencies:

compile:

- org.metaborg:org.metaborg.meta.lang.esv:\${metaborgVersion}
- org.metaborg:org.metaborg.meta.lang.template:\${metaborgVersion}
- org.metaborg:org.metaborg.meta.lang.nabl:\${metaborgVersion}
- org.metaborg:org.metaborg.meta.lang.ts:\${metaborgVersion}

source:

- org.metaborg:org.metaborg.meta.lib.analysis:\${metaborgVersion}

into:

dependencies:

compile:

- org.metaborg:org.metaborg.meta.lang.esv:\${metaborgVersion}
- org.metaborg:org.metaborg.meta.lang.template:\${metaborgVersion}
- org.metaborg:org.metaborg.meta.lang.nabl:\${metaborgVersion}
- org.metaborg:org.metaborg.meta.lang.ts:\${metaborgVersion}

source:

- org.metaborg:meta.lib.spoofax:\${metaborgVersion}
- org.metaborg:org.metaborg.meta.lib.analysis:\${metaborgVersion}

If you do not use NaBL, TS, and task engine based analysis any more, you can also delete the `org.metaborg:org.metaborg.meta.lib.analysis:${metaborgVersion}` source dependency.

Some imports have to be changed to point to the new libspoofax library:

- In `editor/Syntax.esv` or your equivalent ESV file that handles coloring:
 - Change import `editor/Colorer.generated` to `libspoofax/color/default`
- In `trans/analysis.str` for NaBL/TS projects:
 - Add import `libspoofax/core/language`
- In `trans/outline.str`:
 - Remove imports to the runtime libraries
 - Add import `libspoofax/editor/outline`
- In `trans/pp.str`:
 - Remove imports to the runtime libraries
 - Add imports `libspoofax/sdf/pp` and `libspoofax/editor/refactoring/-`
- In other Stratego files:
 - Remove all imports to the runtime libraries that do not pertain NaBL, TS, and task engine based static analysis, and replace them with `libspoofax` equivalents.
 - Remove all imports to the `stratego/metaborg` generated file, and replace them with `libspoofax` equivalents.

Here is a list of other imports and strategies that were moved:

- Imports:
 - `runtime/editor/origins` -> `libspoofax/term/origin`
 - `runtime/editor/annotations` -> `libspoofax/term/annotation`
 - `runtime/completion/-` -> `libspoofax/editor/completion/-`
- Strategies:

- `*-at-position: libspoofax/term/position`
- `parse-file: libspoofax/core/parse`
- `language: libspoofax/core/language`
- `project-path: libspoofax/resource/path`
- `open-import: libspoofax/resource/cache`
- `refresh-workspace-file: removed, not needed any more since all file system operations go through VFS, which routes them through Eclipse.`

25.6 Spoofax 2.0.0 Migration Guide

This migration guide describes the differences between Spoofax 1.5 and 2.0 and describes the steps to convert a Spoofax 1.5 project to Spoofax 2.0 project.

To gather the required knowledge for migrating a language project, go through the documentation in the following order:

1. [Language Development Getting Started](#), to install and get familiar with Spoofax 2.0.
2. [Spoofax 2.0 Release Notes](#), for a general overview of the changes in Spoofax 2.0.
3. This document, for the concrete differences and steps to convert your Spoofax project.

25.6.1 Differences

Concepts

Spoofax 2.0 introduces several new concepts and terminology.

A *language specification* is the specification of a language using meta-languages.

A *language specification project* specifies a *language component*. When the specification is compiled, the result is a component which can be loaded into Spoofax. A language component has specifications and implementations for parts of a language, such as its parser, pretty-printer, analysis, transformations, editor services, etc.

A component contributes these parts to a *language implementation*. Multiple components can contribute to the same language implementation, and components can contribute to multiple language implementations. In the most simple case, a single component contributes all parts of the language to a single implementation.

Language components can *depend* on other language components to depend on parts of a language. Currently, there are two kinds of dependencies: compile and source dependencies.

A *compile dependency* on a language component is used to compile source files of that language component. For example, a compile dependency on NaBL will ensure that all `.nab` files are compiled into `.str` files.

A *source dependency* is used to depend on source files of a language component. Source dependencies are used to depend on libraries, for example to depend on a Stratego library for name and type analysis. They are also used to compose multiple language components into a single language component, for example to do language extension.

A *language* is the more abstract notion of a language, which has multiple language implementations. For example, the Java language has the JDK7 and JDK8 *implementations*, which each have front-end and back-end *components*.

An *end-user project* is a project with programs of an end-user language, in contrast to a language specification project which has programs of meta-languages. For example, a Java project is a Java end-user project, whereas the JDK project is a language specification project.

Project structure

The project structure of language specification projects has significantly changed. The biggest change is that these projects are no longer Eclipse (plugin) projects, so that they can be used outside of the Eclipse platform as well. Ant build files have also been removed since we do not use Ant to build projects any more. Many ESV files have been deprecated, and all generated ESV files in the editor directory have been removed.

The following files and directories are no longer part of the project structure:

- Ant build: `.externalToolBuilders`, `build.generated.xml`, `build.main.xml`
- Eclipse plugin: `plugin.xml`, `META-INF`
- Eclipse project: `.settings`, `.classpath`, `.project`, `build.properties`
- Refactoring: `lib/refactor-common.generated.str`
- Deprecated ESV files: `editor/langname-Completions.esv`, `editor/langname-Folding.esv`, `editor/langname-Outliner.str`, `editor/langname-Refactorings.esv`
- Generated ESV files: `editor/langname-*.generated.esv`, `editor/langname-Outliner.generated.str`
- The RTG and signatures files are no longer generated for SDF3 projects, since SDF3 generates its own signatures.
- The generated box pp files are no longer generated, and box pp files are no longer converted into pp.af files.

The following files and directories have been moved:

- ESV
 - Main ESV file must be at `editor/Main.esv`. If it does not exist, no packed ESV file will be generated.
 - Packed ESV file: `target/metaborg/editor.esv.af`
- SDF
 - Definition: `src-gen/syntax/[LanguageName].def`
 - Permissive definition: `src-gen/syntax/[LanguageName]-permissive.def`
 - Parenthesizer: `src-gen/pp/[LanguageName]-parenthesize.str`
 - Parse table: `target/metaborg/sdf.tbl`
- Stratego
 - ‘editor-common.generated’ file: `src-gen/stratego/metaborg.str`
 - Ctree: `target/metaborg/stratego.ctree`
 - Generated Java files: `src-gen/stratego-java`
 - JAR: `target/metaborg/stratego.jar`
 - Java strategies: `src/main/strategies`
 - Java strategies JAR: `target/metaborg/stratego-javastrat.jar`
 - Build cache: `target/stratego-cache`
- DynSem
 - Manual Java: `src/main/ds`
 - Generated Java: `src-gen/ds-java`

The following generated files and directories still exist, but should not be published to source control any more:

- lib/editor-common.generated.str or stratego/metaborg.str
- src-gen

When importing a language specification project into Eclipse or IntelliJ, several platform-specific files will be generated. These files should not be published to source control to keep projects as platform-agnostic as possible.

Eclipse

Importing

To import a language specification project into Eclipse, use Import... → Maven → Existing Maven Projects. We use Maven to set up the correct Java dependencies, which is why there is no special ‘Existing Spoofax Projects’ importer.

Builds

Eclipse has the concept of incremental project builders, which incrementally parse, analyze, and compile files inside a project. An example of such a project builder is the Eclipse JDT builder which incrementally builds Java files. Spoofax 1.5 did not use this functionality, but the new Eclipse plugin in Spoofax 2.0 does.

The project builder for Spoofax parses, analyses, executes transformations, and shows all error markers, for all language files (Stratego files, SDF3 files, files of your language, etc.) in the project. If the project is opened for the first time, a full build will occur, building all language files. When changes occur in the project, an incremental build occurs, building only changed files.

The commands under the Project menu in Eclipse now also affect Spoofax projects. Executing Project → Build... (or pressing Ctrl/Cmd+Alt+B) will build the project.

Executing Project → Clean... will delete the .cache directory, reset the index and task engine, remove all error markers, and reanalyze and rebuild all files in the project. This makes the Reset and Reanalyze builder unnecessary, since this is now properly integrated with Eclipse.

Automatic building can also be turned off by disabling Project → Build Automatically. Builds will then only occur if Project → Build Project is executed or if Ctrl/Cmd+Alt+B is pressed.

Furthermore, the language specification build is no longer written in Ant, but in Java using the [Pluto](#) incremental build system.

Natures

The Spoofax language specification project builder is not enabled by default, to enable it a ‘Spoofax meta nature’ must be added to the project. A nature in Eclipse is a project tag which enables functionality for that project. To add the Spoofax nature to a project, right click the project, and choose Spoofax (meta) → Add Spoofax meta nature. When importing a language specification, this nature is automatically added.

For end-user projects, right click the project, and choose Spoofax → Add Spoofax nature to add a nature for end-user projects.

Editors will parse and analyze files regardless of there being a Spoofax nature, but the on-save handler will not be called.

Builders

Builders for the open editor are now located in the Spoofax main menu instead of buttons on the tool bar. Builders wait for the analyzed AST if needed, so the issue where builders are sometimes not executed on the analyzed AST should be solved now.

Builders can also be executed on a set of files by selecting the files in the project or package explorer, right clicking the files, selecting the language name from the menu, and then selecting a builder.

Cancellation

Editor updates can now be cancelled by clicking the red stop button in the progress view. If the progress view is not visible, you can open it by choosing Window → Show View → Progress. If the editor update is not responsive (it is looping for example), the thread running the editor update will be killed after a while.

Killing a thread during analysis may leave the index and task engine in an inconsistent state. If this happens, clean the project using Project → Clean... to force a full build, which makes the state consistent again. Killing a thread is not very well supported in Java and may break Eclipse or even the JVM, which then requires a restart.

Project builds and transformations can also be cancelled in the progress view.

Console logging

Console logging in the new plugin is more prominent so that we can diagnose problems more easily. If the console is not visible, you can open it by choosing Window → Show View → Console. The console does not automatically pop-up when there is a message any more, so it can also be hidden by just closing it.

All warning and error messages are also sent to Eclipse's error log. The error log can sometimes contain more information about exceptions and stack traces in errors. If the error log is not visible, you can open it by choosing Window → Show View → Error Log.

Manually loading/unloading a language

A language can be manually loaded or reloaded by right clicking a project and choosing Spoofax (meta) → Load language, and unloaded with Spoofax (meta) → Unload language.

External dependencies

The new plugin does not depend on a modified version of IMP, making it possible to install the Rascal plugin alongside the Spoofax plugin. All other external dependencies are limited to the Spoofax plugin, which should prevent conflicts with other Eclipse plugins.

25.6.2 Converting a project

If your project is simple (e.g. it only has syntax and a few transformations), the easiest way to convert your project is to create a new Spoofax language specification project, and to copy your files into that project.

Otherwise, Spoofax 2.0 supports converting an old Spoofax project into a new Spoofax project, but some conversions need to be done manually.

Warning: Converting a Spoofax project is a destructive operation, some files will be deleted, replaced, or changed. Make a backup of your projects before doing any conversions!

Automatic conversion

First, import your existing Spoofax project into Eclipse using File → Import... → Existing Projects into Workspace. Right click the project, and choose Spoofax (meta) → Upgrade language project. A wizard screen will pop up where you have to fill in some details about your language.

If a packed.esv file was found, Spoofax will try to fill in some fields for you. If not, all fields need to be filled in manually. The *id* and *name* of your language can be found in the main ESV file. For *group id*, use a Maven identifier for your organization (e.g. org.metaborg), and as *version*: 1.0.0-SNAPSHOT.

Warning: Make sure that the *id* and *name* fields match exactly with the fields in your ESV file, otherwise the conversion will go wrong.

Press finish to convert the language project.

Manual conversion

Unfortunately, not all required conversions can be done automatically. Do the following conversions manually.

Project configuration

Most of the project configuration is now in the metaborg.yaml file. The [manual page on configuration](#) lists all configuration options.

- Add/remove compile and source dependencies as needed.
- Add build configuration, such as Stratego compiler arguments, SDF compiler arguments, external def files, and external JAR files.

Imports

In Stratego, SDF2, SDF3, NaBL, and TS files:

- Remove `src-gen`, `lib`, and `trans`, from module names and imports. These paths are now on the source path of the SDF and Stratego compilers.

In Stratego, NaBL, and TS files:

- Instead of importing `lib/editor-common.generated`, import `stratego/metaborg`.
- Instead of importing `include/<langname>-parenthesize`, import `pp/<langname>-parenthesize`.
- If you're using SDF3:
 - Instead of importing the signatures from `include/<langname>`, import them from `signatures/<langname>-sig`. These signatures are spread over multiple files, import all the required files to fix

errors, since the Stratego editor does not handle transitive imports. You can also use the wildcard import `signatures/-` to import all signature files, if your syntax definition is not spread over multiple directories.

- If you're using SDF2 or an external definition file:
 - Instead of importing the signatures from `include/<langname>`, import them from `signatures/<langname>`.

SDF

If you are still using SDF2 instead of SDF3, add the following setting to the `metaborg.yaml` file:

```
language:
  sdf:
    version: sdf2
```

NaBL and TS

If you're using a NaBL/TS based analysis, perform the following changes:

- NaBL files are now generated into `src-gen/names`, fix imports to NaBL files, delete old generated NaBL files.
- TS files are now generated into `src-gen/types`, fix imports to TS files, delete old generated TS files.
- The `editor-analyze` calls have been changed. Remove `analysis-single-default-interface`, `analysis-multiple-default-interface`, and `editor-analyze`. Replace it with:

```
editor-analyze = analyze-all(pre-analysis, post-analysis, pp-message|<language>)
```

with the `pre-analysis`, `post-analysis`, and `pp-message` arguments that you were using before. Also make sure the observer (in your `esv`) has the `(multifile)` property.

- The `editor-save` call to `analysis-save-default(|<language>)` has been removed, remove that call. You can remove `editor-save` entirely if you don't do any generation, also remove the `on save` strategy from `ESV` if you do. If you do generation but do not return a `(file, text)` tuple from `editor-save`, be sure to return a `!None()` to tell Spoofax that you're returning nothing.
- The `index-setup` and `task-setup` strategies have been removed, Spoofax Core does this for you now. Remove all calls to these strategies.
- Remove the `path` argument to `analysis-resolve` in `editor-resolve`.
- Remove the `path` argument to `analysis-propose-completions` in `editor-complete`.
- Remove the `debug-reanalyze` strategy, and remove it from your menu in `ESV`. You can reanalyze by cleaning the project.

ESV

- The following `ESV` files are now deprecated, delete and remove any imports to these files:
 - `editor/langname-Completions.esv`
 - `editor/langname-Folding.esv`
 - `editor/langname-Refactorings.esv`

- Previously generated ESV files in the editor directory are not generated any more. Delete the generated files and remove the imports to generated files.
- The colorer ESV file is now generated to `src-gen/editor/Colorer.generated.esv`, import it with `imports editor/Colorer.generated` in an ESV file.
- The generated syntax ESV file is no longer generated. If you were using the defaults from the generated file, add them to an ESV file:

```
language

line comment : "//"
block comment : "/*" * "*/"
fences       : [ ] ( ) { }
```

- The outliner (`editor/langname-Outliner.str`) must be moved to the `trans` directory. Rename it to `trans/outline.str`, change its module to `outline`, and fix imports of the outliner.
- Change the file name of the main ESV file to `Main.esv`, and change its module to `Main`.
- In the main ESV file:

- Change the parse table:

```
table : target/metaborg/sdf.tbl
```

- Change the Stratego providers

- * For ctree projects:

```
provider : target/metaborg/stratego.ctree
```

- * For jar projects:

```
provider : target/metaborg/stratego.jar
```

- * For projects with Java strategies:

```
provider : target/metaborg/stratego.jar
provider : target/metaborg/stratego-javastrat.jar
```

Java strategies

If your project has Java strategies:

- Create the `src/main/strategies` directory.
- Move Java strategies from `editor/java` into the `src/main/strategies` directory. Be sure to preserve the existing Java package structure.
- Perform a Maven update by right clicking the project and choosing `Maven → Update Project...`, to update the Java source directories of the project.

DynSem

If your project has manual DynSem Java files:

- Create the `src/main/ds` directory.

- Move manual DynSem Java files from editor/java into the src/main/ds directory. Be sure to preserve the existing Java package structure.
- Perform a Maven update by right clicking the project and choosing Maven → Update Project. . . , to update the Java source directories of the project.

Ant build customization

Language specification builds do not use Ant any more, so any customizations to the build.main.xml are lost. To perform an Ant task before and after the build, add the following configuration option to your metaborg.yaml file:

```
build:
  ant:
    - phase: preCompile
      file: ${path:root}/ant.xml
      target: generate-xml
    - phase: postCompile
      file: ${path:root}/ant.xml
      target: package-xml
```

See the [manual page on configuration](#) for more information about configuring Ant build steps.

Eclipse plugin

Language specification projects are not Eclipse plugins any more. To create an Eclipse plugin for your language, follow the [guide for exporting a language](#) as an Eclipse plugin.

Git

If you're using Git, the .gitignore file is replaced with a new one, add entries that you need again. All generated files that were previously not ignored, are ignored now. To delete all ignored files from the Git index (the files will remain on the filesystem but Git will forget about them), make sure all your useful changes are committed and pushed, then run the following commands:

```
git rm -r --cached .
git add .
git commit -am "Remove ignored files"
```

Building

When you are done with converting the project, build it with Cmd+Shift+B or Project → Build. If the build does not work, try cleaning the project first with Project → Clean, and then building again. Also make sure that Project → Build Automatically is turned on.

25.7 New Completions Framework

In the latest Spoofax, we provide (beta) support for syntactic code completion.

Code completion can happen in three ways: expanding an explicit placeholder, expanding recursive structures, lists or nullable structures, and finishing a structure that has part of it already in the program and thus contains a syntax error due to missing symbols.

25.7.1 Expanding an explicit placeholder

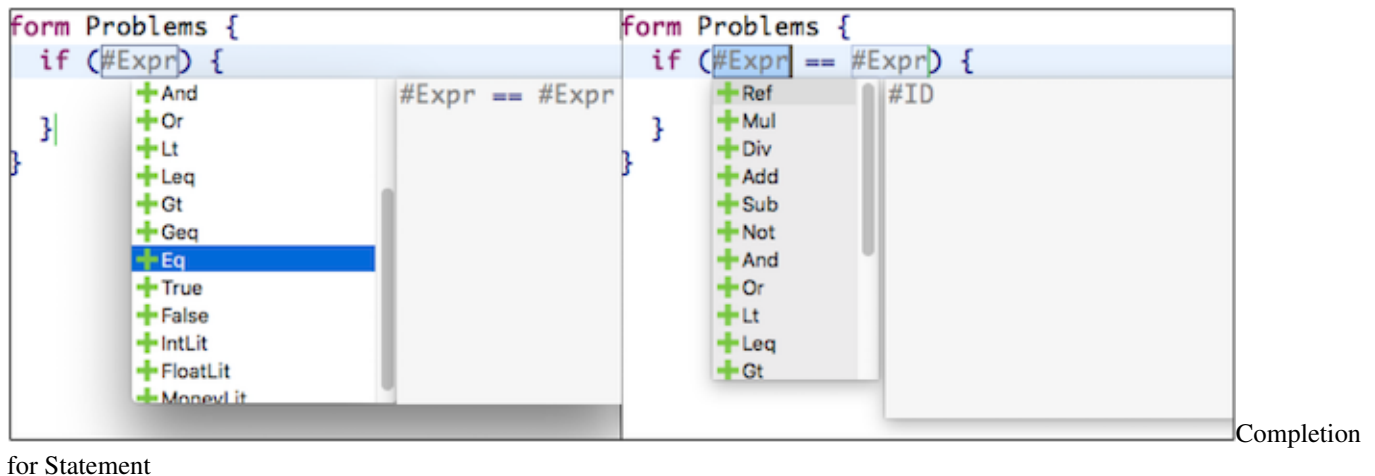
Programs may contain explicit placeholders to represent incomplete structures. When positioning the cursor inside a placeholder and triggering code completion, the editor shows proposals in a pop-up and each proposal represents a placeholder expansion. Each non-terminal in the syntax definition generates a placeholder, and a placeholder can appear wherever the non-terminal can appear in the program. Users are able to edit the format of a placeholder (to avoid clashes with actual elements of the language) by editing the YAML file in the project. For example, if the yaml file contains the following lines:

```
language:
  sdf:
    placeholder:
      prefix: "#"
      suffix: "#"
```

all placeholders in the program are going to be `#NAME#`, where `NAME` is the name of the non-terminal that the placeholder represents. Suffix are not mandatory and if no configuration is specified, placeholders will be formatted as `[[NAME]]`.

Note: It is necessary to clean the project after changing the format of placeholders in the YAML file.

An example of completing a program by expanding a placeholder is shown below:

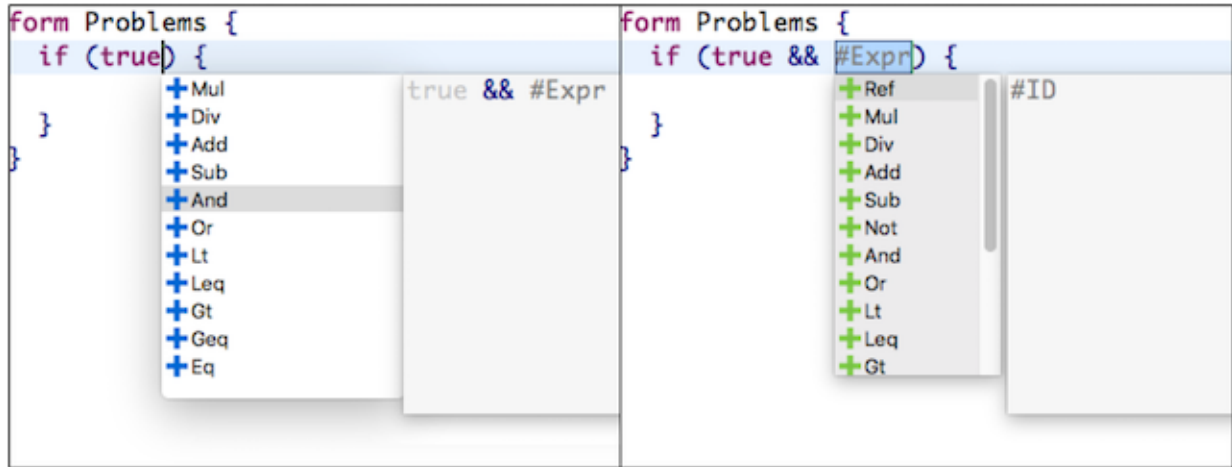


for Statement

In this case, the explicit placeholder `#Expr` can be extended by any of the shown proposals. When selecting the proposal `Eq` the placeholder is replaced by `#Expr == #Expr` and the process of completing the program continues with the next placeholder in the program.

25.7.2 Expanding recursive, lists and nullable structures

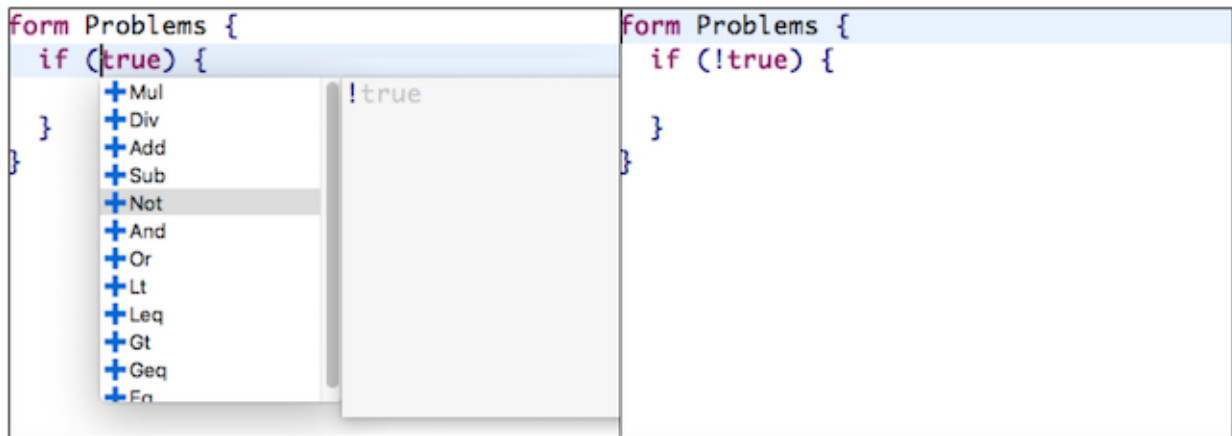
Code completion may also allow extending the program when the cursor is at specific terms - recursive terms, lists or nullables - by term transformation. In the case of recursive terms, if a term is left-recursive and the cursor is positioned at the right of the term (considering its surrounding layout), we use the current term as the as the leftmost child in any of the left recursive productions of the same sort. We show an example to illustrate this scenario below. In the example we expand the boolean expression `true` by triggering completion after the expression, transforming it into an `And` term, and therefore turning `true` into `true && #Expr`.



Left

Recursive code completion

Similarly, we can expand terms considering its right recursive productions. The example below shows how to expand the boolean expression `true` into its negation `!true` using syntactic code completion.

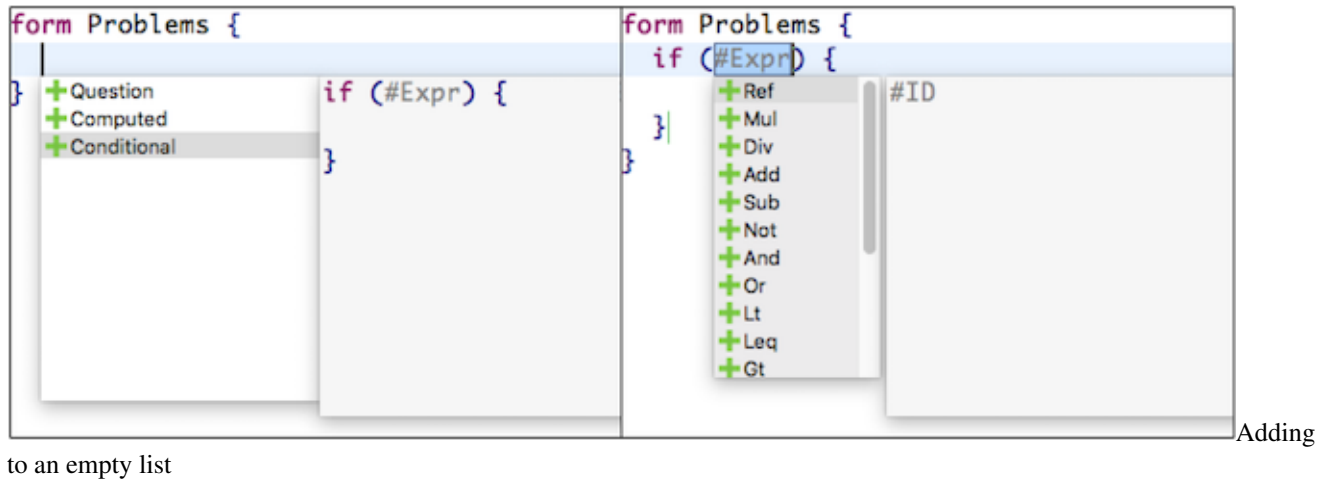


Right

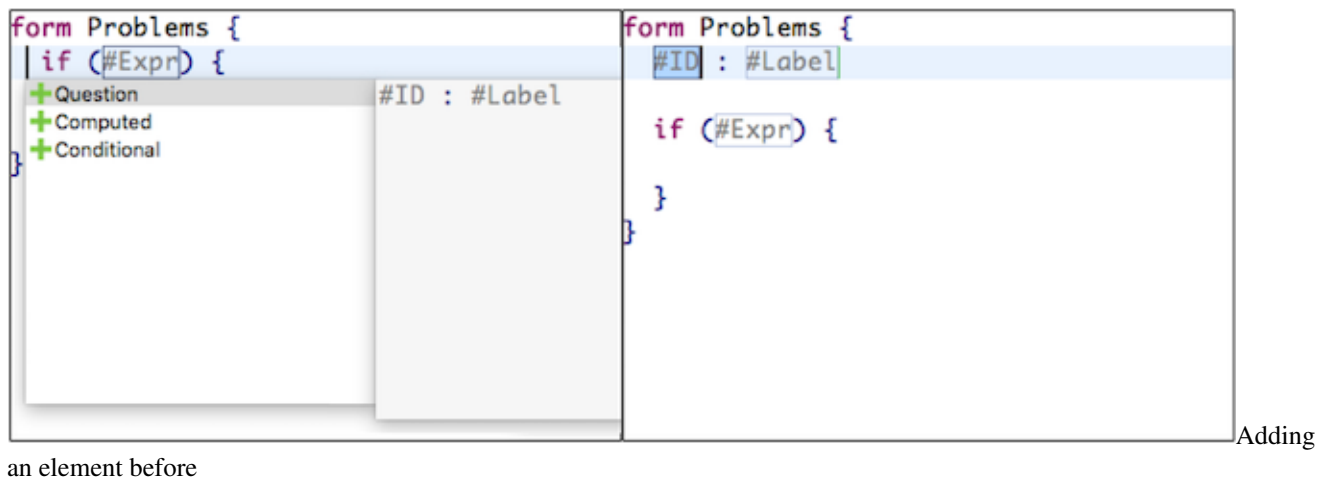
Recursive code completion

It is also possible to use syntactic code completion to add elements to lists in the program by positioning the cursor inside the list (considering the list's surrounding layout). In the example below, the completion framework adds an element to an empty list, before or after another element.

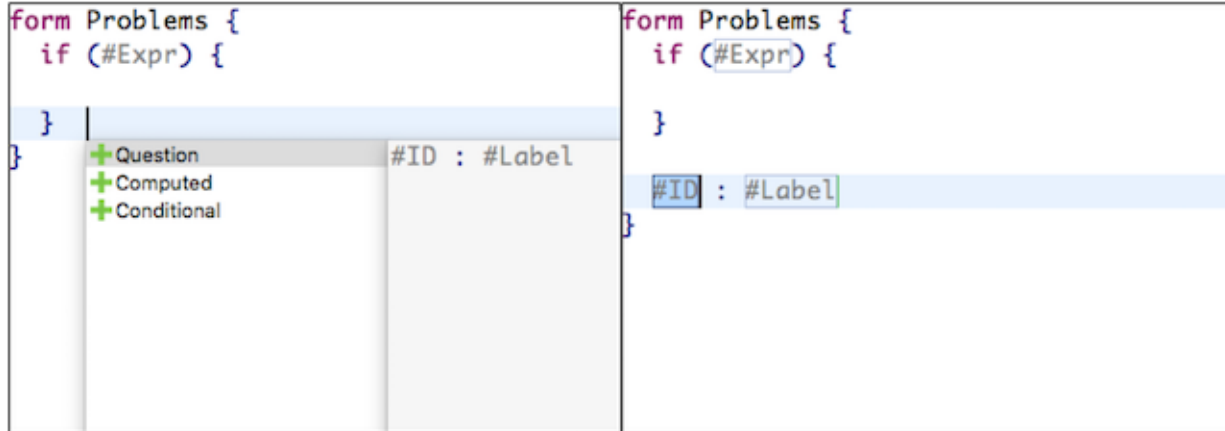
Empty list



Adding before



Adding after

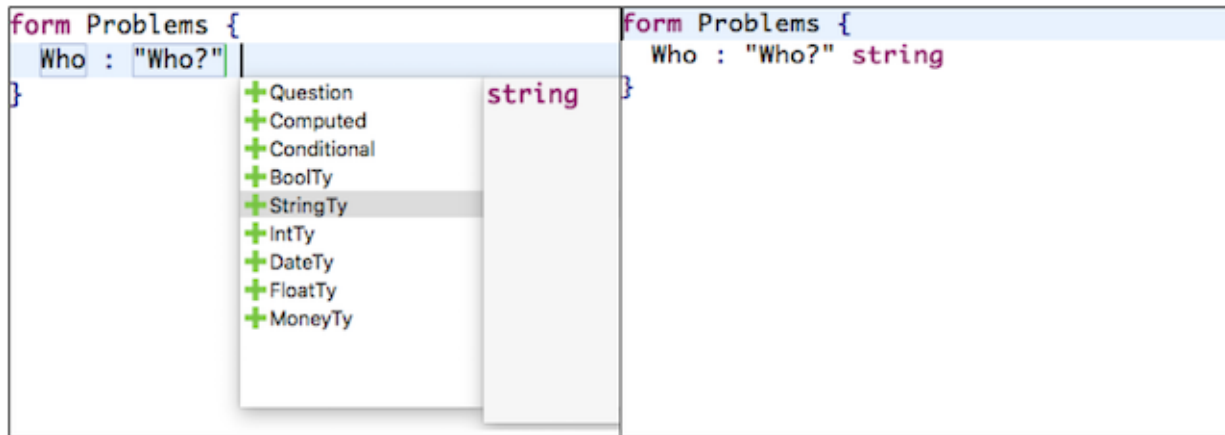


Adding

an element after

Note: When adding an element to a list the separator is inferred from the grammar and automatically added.

Finally, syntactic code completion can also insert nullable/optional nodes that are not present in the program. For example, consider an optional **Type** at the end of a **Question**. When triggering completion at the layout after a question, like in the example below, it is possible to add the optional type (and also another element to the list of questions, as the layout of the list and the optional terms overlap).

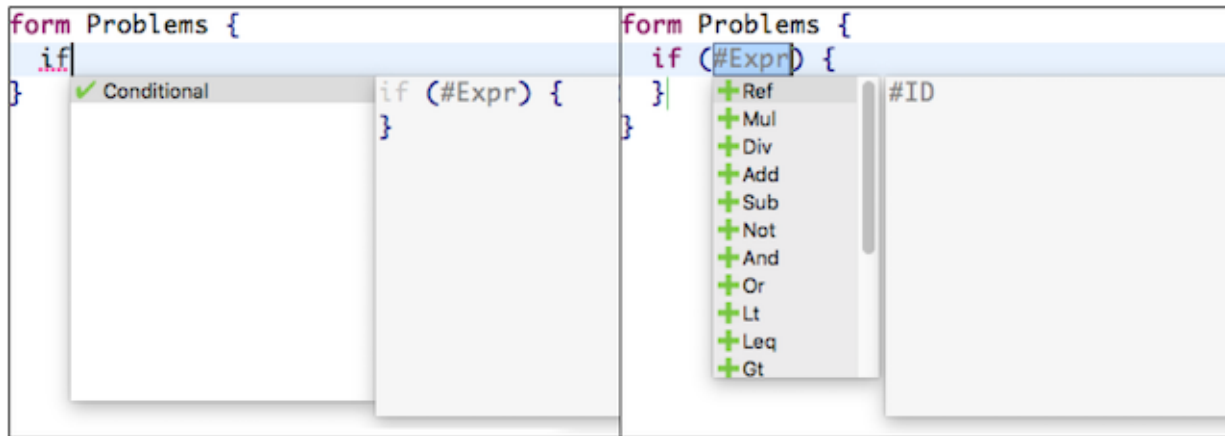


Optional

Completion

25.7.3 Syntactic completion as error recovery

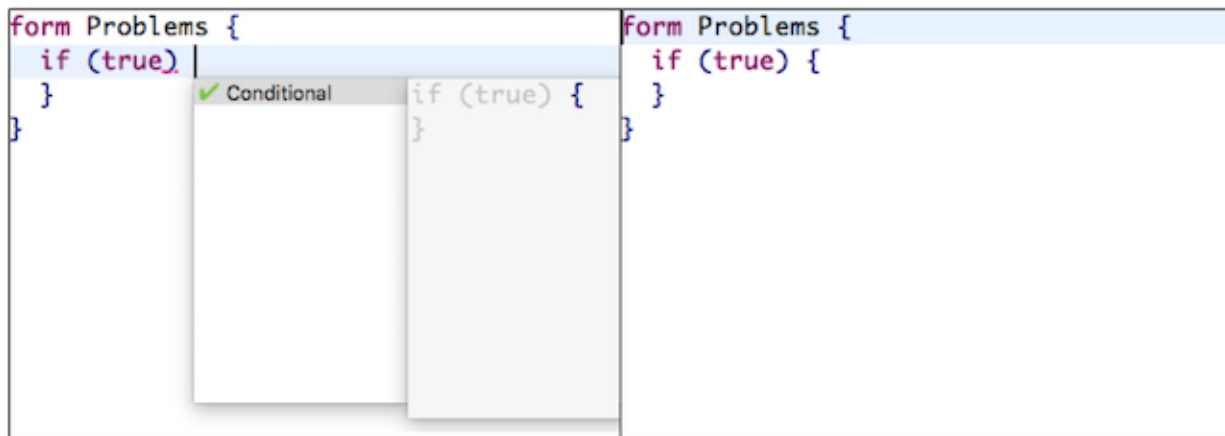
Programs might already contain part of a structure in the input and thus a syntax error. For erroneous programs, syntactic code completion offers all possible ways to finish a structure by adding missing symbols at the cursor position. For example, by typing `if` and triggering code completion, the framework proposes completing the program by inserting an `if` statement.



Completion

as recovery

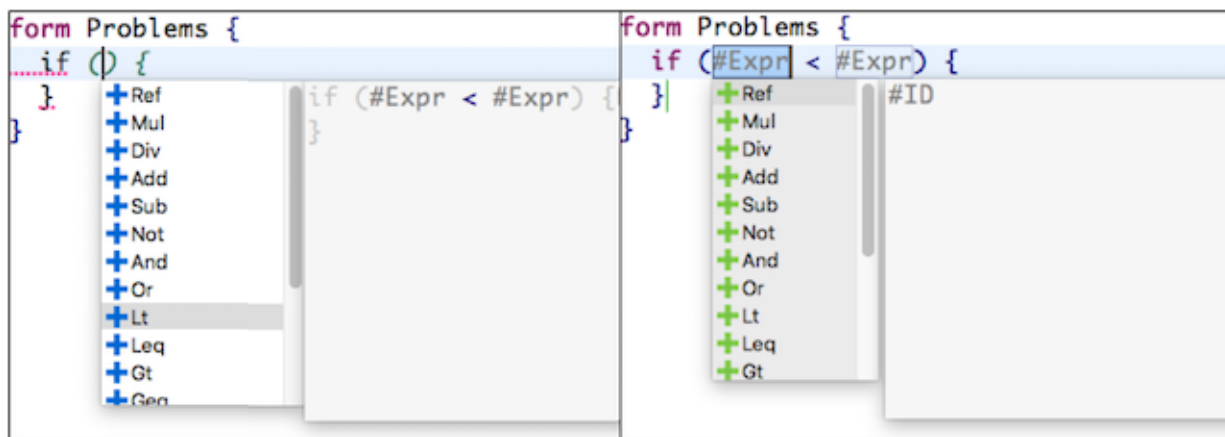
Completion as recovery can also add only lexical symbols to fix the program. In the example below, syntactic code completion fixes the if statement by adding the opening `{`.



Recover

lexical

When a single placeholder is necessary to fix the program, the completion framework shows all placeholder expansions as proposals.



Recover

placeholder

Note: Proposals are formatted according to SDF3 productions.

25.7.4 How to migrate old projects?

New projects come automatically with support for the new completions framework. To migrate old projects it is necessary to:

- add the following imports to the main Stratego file:
 - `completion/completion`. This imports the Stratego library for syntactic code completion.
- add the following import to `<LanguageName>-Colorer.esv`:
 - `completion/colorer/<LanguageName>-cc-esv`. This import the editor files responsible for coloring the explicit placeholders.

In case of any issue or suggestion for improving the framework, please create an entry detailing your suggestion/issue with a tag `completion` in <http://yellowgrass.org/project/SpoofoxWithCore>.

25.8 Directory structure migration

To clean up the structure of a language specification project, we've made the following changes:

- **ESV**
 - Main ESV file must be at `editor/Main.esv`. If it does not exist, no packed ESV file will be generated.
 - Packed ESV file: `target/metaborg/editor.esv.af`
- **SDF**
 - The RTG and signatures files are no longer generated for SDF3 projects, since SDF3 generates its own signatures.
 - The generated box pp files are no longer generated, and box pp files are no longer converted into pp.af files.
 - Definition: `src-gen/syntax/[LanguageName].def`
 - Permissive definition: `src-gen/syntax/[LanguageName]-permissive.def`
 - Parenthesizer: `src-gen/pp/[LanguageName]-parenthesize.str`
 - Parse table: `target/metaborg/sdf.tbl`
- **Stratego**
 - 'editor-common.generated' file: `src-gen/stratego/metaborg.str`
 - Ctree: `target/metaborg/stratego.ctree`
 - Generated Java files: `src-gen/stratego-java`
 - JAR: `target/metaborg/stratego.jar`
 - Java strategies: `src/main/strategies`
 - Java strategies JAR: `target/metaborg/stratego-javastrat.jar`
 - Build cache: `target/stratego-cache`
- **DynSem**
 - Manual Java: `src/main/ds`
 - Generated Java: `src-gen/ds-java`
- **Other**

- Pluto build cache: target/pluto

To migrate your project, make the following changes:

- Change the file name of the main ESV file to Main.esv, and change its module to Main.
- In the main ESV file:

- Change the parse table:

```
table : target/metaborg/sdf.tbl
```

- Change the Stratego providers

- * For ctree projects:

```
provider : target/metaborg/stratego.ctree
```

- * For jar projects:

```
provider : target/metaborg/stratego.jar
```

- * For projects with Java strategies:

```
provider : target/metaborg/stratego.jar
provider : target/metaborg/stratego-javastrat.jar
```

- In all Stratego, NaBL, TS files

- Instead of importing lib/editor-common.generated, import stratego/metaborg.

- Instead of importing include/<langname>-parenthesize, import pp/<langname>-parenthesize.

- If you're using SDF3:

- * Instead of importing the signatures from include/<langname>, import them from signatures/<langname>-sig. These signatures are spread over multiple files, import all the required files to fix errors, since the Stratego editor does not handle transitive imports. You can also use the wildcard import signatures/- to import all signature files, if your syntax definition is not spread over multiple directories.

- If you're using SDF2 or an external definition file:

- * Instead of importing the signatures from include/<langname>, import them from signatures/<langname>.

- If your project has Java strategies:

- Create the src/main/strategies directory.
- Move Java strategies from editor/java into the src/main/strategies directory. Be sure to preserve the existing Java package structure.

- If your project has manual DynSem Java files:

- Create the src/main/ds directory.
- Move manual DynSem Java files from editor/java into the src/main/ds directory. Be sure to preserve the existing Java package structure.

- Perform a Maven update by right clicking the project and choosing Maven → Update Project. ... Enable Force Update of Snapshots/Releases in the new window and press Ok. This updates the Java source directories of the project.

- If you are still using SDF2 instead of SDF3, add the following setting to the metaborg.yaml file:

```
language:  
  sdf:  
    version: sdf2
```

The following migration guides are stubs:

25.9 Spoofax vNext Migration Guide

This is a stub for the migration guide of Spoofax vNext.

Spoofax and its components have been developed by many researchers, student, and developers supported by universities and funding agencies.

26.1 Universities

The main research effort on Spoofax and its predecessors (SDF2, Stratego, XT) was conducted at the following universities

- University of Amsterdam
- Oregon Graduate Institute
- Utrecht University
- University of Bergen
- Delft University of Technology

26.2 Funding

Funding was provided by the following funding agencies and companies

- [The Netherlands Organisation for Scientific Research \(NWO\)](#)
- Philips Research
- Oracle Labs
- Capes, Coordenação de Bolsas, Brasil

26.3 Tooling

The following tools were used to develop and maintain Spoofax



We use the [YourKit Java profiler](#) to diagnose and fix performance problems in Spoofax, provided free-of-charge by [YourKit](#).

26.4 Contributors

Many people have contributed to Spoofax as bachelor student, master student, PhD student, postdoc, professor, or as an external contributor. This is an attempt at listing at least the main contributors divided in main development periods.

- SDF2
 - Eelco Visser
 - Jeroen Scheerder
 - Mark van den Brand
 - Jurgén Vinju
- Stratego/XT
 - Eelco Visser
 - Martin Bravenboer
 - Karina Olmos
 - Karl Trygve Kalleberg
 - Anya Bagge
 - Joost Visser
 - Merijn de Jonge
 - Rob Vermaas
 - Eelco Dolstra
- Spoofax 1
 - Eelco Visser
 - Lennart Kats
 - Karl Trygve Kalleberg
 - Rob Vermaas
 - Guido Wachsmuth
 - Maartje de Jonge
 - Ricky Lindeman
 - Sebastian Erdweg
 - Tobi Vollebregt

- Spoofax 2
 - Eelco Visser
 - Gabriël Konat
 - Eduardo Amorim
 - Vlad Vergu
 - Volker Lanting
 - Daniel Pelsmaecker
 - Andrew Tolmach
 - Pierre Néron
 - Hendrik van Antwerpen

Spoofax is a platform for developing textual (domain-specific) programming languages. The platform provides the following ingredients:

- Meta-languages for high-level declarative language definition
- An interactive environment for developing languages using these meta-languages
- Code generators that produces parsers, type checkers, compilers, interpreters, and other tools from language definitions
- Generation of full-featured Eclipse editor plugins from language definitions
- Generation of full-featured IntelliJ editor plugins from language definitions (experimental)
- An API for programmatically combining the components of a language implementation

With Spoofax you can focus on the essence of language definition and ignore irrelevant implementation details.

Developing Software Languages

Spoofox supports the development of *textual* languages, but does not otherwise restrict what kind of language you develop. Spoofox has been used to develop the following kinds of languages:

Programming languages Languages for programming computers. Implement an existing programming language to create an IDE and other tools for it, or design a new programming language.

Domain-specific languages Languages that capture the understanding of a domain with linguistic abstractions. Design a DSL for your domain with a compiler that generates code that would be tedious and error prone to produce manually.

Scripting languages Languages with a special run-time environment and interpreter

Work-flow languages Languages for scheduling actions such as building the components of a software system

Configuration languages Languages for configuring software and other systems

Data description languages Languages for formatting data

Data modeling languages Languages for describing data schemas

Web programming languages Languages for programming web clients or servers

Creating Full-Featured Editors

From a language definition Spoofax generates full-featured Eclipse and IntelliJ editor plugins, as well as a command-line interface. Generated editors support the following features:

- Syntactic editor services
 - Syntax highlighting
 - Syntax checking
 - Parse error recovery
 - Outline view
 - Syntactic code completion
 - Formatting
- Semantic editor services
 - Name checking
 - Type checking
 - Inline error markers
 - Reference resolution: navigate to declaration
- Builders: custom operations for invoking
 - Code generation
 - Interpreter
 - Transformations
 - Refactorings

Declare Your Language

We design Spoofax according to the following guiding principles:

Separation of concerns Separate specification from implementation. Separate language-specific aspects from language-independent aspects. Separate definition of separate aspects (e.g. separate syntax definition and static semantics definition)

Single source Instead of repeating a language aspect in many different implementation components, we aim to generate many different artifacts from a single source.

Declarative language definition Language designers should focus on what distinguishes their language and should not be distracted by writing boilerplate for recurring implementation details. Rather than confronting each language designer with these implementation details, we factor them out into language-independent abstractions and corresponding implementations.

Following these guidelines, Spoofax provides the following high-level, declarative meta-languages:

SDF3 The SDF3 Syntax Definition Formalism allows language designers to focus on the structure of programs rather than on debugging parser implementations by means of the following features: support for the full class of context-free grammars by means of generalized LR parsing, integration of lexical and context-free syntax through scannerless parsing, safe and complete disambiguation using priority and associativity declarations, an automatic mapping from parse trees to abstract syntax trees through integrated constructor declarations, automatic generation of formatters based on template productions, syntactic completion proposals in editors.

NaBL2 The NaBL2 ‘Name Binding Language’ supports the definition of the static semantics of languages including name binding and type analysis. NaBL2 rules define a mapping from abstract syntax trees to name and type constraints. The generated constraints are solved by a language-independent solver and produce error messages to display in an editor and a symbol table for the analyzed abstract syntax tree for use in further processing. Name analysis in NaBL2 is based on scope graphs, a language-independent model for name resolution and scoping.

FlowSpec The FlowSpec Data-Flow Analysis Specification Language supports the specification of control-flow and intra-procedural data-flow analysis. FlowSpec control-flow rules map abstract syntax trees to control-flow edges. Data-flow properties in FlowSpec represent the results of different data-flow analyses, and the analyses are specified through property rules per type of control-flow node. FlowSpec depends on the use of NaBL2 for name binding.

Stratego The Stratego transformation language supports the definition of transformations of abstract syntax terms using rewrite rules and programmable rewriting strategies. Strategies enable concise definition of traversals over trees. Stratego is used to define desugarings, transformations, optimizations, and code generation (translation to another language).

DynSem The DynSem Dynamic Semantics specification language supports the definition of the execution behavior of programs by means of reduction rules that are typically used to define *natural semantics* or *big-step operational semantics*. DynSem specifications are compiled to interpreters targeting the Truffle/Graal stack.

SPT The SPT testing language supports the definition of tests for all aspects of a language definition.

ESV The ESV editor services language is used to configure language definitions.

A Platform for Language Engineering

Spoofax is a platform for language engineers. That is, it provides full support for software engineering of language implementations.

Agile Language Development An important feature of Spoofax is its support for agile language development. The development of a language definition and testing that language definition in the generated IDE for the language under development is done in the *same* Eclipse instance. This enables a quick turn-around time between language development and language testing.

IDE generation Spoofax generates a full fledged editor plugin from a language definition.

API Spoofax does not only provide an IDE for interactively developing and using languages, it also provides a programmatic interface that enables embedding languages and their implementations directly in application code or to invoke language components from build systems or the command line.

Bootstrapped Language Workbench Spoofax has been bootstrapped. That is, Spoofax is used for the definition of its own meta-languages and the workbench is the composition of plugins generated for these meta-languages.

Continuous integration The Spoofax sources are continuously built on a buildfarm at TU Delft, which reports build errors to Spoofax developers and provides a complete build for various platforms of the latest version.

Open source Spoofax is open source and available under the Apache 2.0 license. The sources are maintained in the [MetaBorg](#) github organization; pull requests are welcome.

A Platform for Research

Spoofax is a platform for language engineering research. Due to its modular architecture it is easy to extend the workbench with new experimental meta-languages and tools. For example, the current version comes with an improved parser generator in addition to the old SDF2 parser generator, and it provides the NaBL2 static semantics specification language next to the old NaBL/TS solution.

Bibliography

- [R1] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008. doi:[10.1016/j.scico.2007.11.003](https://doi.org/10.1016/j.scico.2007.11.003).
- [R2] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, 444–463. Reno/Tahoe, Nevada, 2010. ACM. doi:[10.1145/1869459.1869497](https://doi.org/10.1145/1869459.1869497).
- [R3] Lennart C. L. Kats, Richard Vogelij, Karl Trygve Kalleberg, and Eelco Visser. Software development environments on the web: a research agenda. In Gary T. Leavens and Jonathan Edwards, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2012, part of SPLASH '12, Tucson, AZ, USA, October 21-26, 2012*, 99–116. ACM, 2012. doi:[10.1145/2384592.2384603](https://doi.org/10.1145/2384592.2384603).
- [R4] Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF meta-environment: a component-based language development environment. In Reinhard Wilhelm, editor, *Compiler Construction, 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2027 of Lecture Notes in Computer Science, 365–370. Springer, 2001. doi:[10.1016/S1571-0661\(04\)80917-4](https://doi.org/10.1016/S1571-0661(04)80917-4).
- [R5] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
- [R6] Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. Building program optimizers with rewriting strategies. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, 13–26. Baltimore, Maryland, United States, 1998. ACM. doi:[10.1145/289423.289425](https://doi.org/10.1145/289423.289425).
- [R7] Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. A language designer’s workbench: a one-stop-shop for implementation and verification of language designs. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, 95–111. ACM, 2014. doi:[10.1145/2661136.2661149](https://doi.org/10.1145/2661136.2661149).
- [S1] Luis Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. Principled syntactic code completion using placeholders. In Tijs van der Storm, Emilie Balland, and Dániel Varró, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language*

- Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, 163–175. ACM, 2016. doi:<http://dl.acm.org/citation.cfm?id=2997374>.
- [S2] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Layout-sensitive generalized parsing. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of Lecture Notes in Computer Science, 244–263. Springer, 2012. doi:[10.1007/978-3-642-36089-3_14](https://doi.org/10.1007/978-3-642-36089-3_14).
- [S3] Jan Heering, P. R. H. Hendriks, Paul Klint, and Jan Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989. doi:[10.1145/71605.71607](https://doi.org/10.1145/71605.71607).
- [S4] Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In *Proceedings of the ASMICS Workshop on Parsing Theory*. Milano, Italy, October 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell’Informazione, Università di Milano.
- [S1] Jeff Smits and Eelco Visser. Flowspec: declarative dataflow analysis specification. In Benoît Combemale, Marjan Mernik, and Bernhard Rumpe, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, 221–231. ACM, 2017. doi:[10.1145/3136014.3136029](https://doi.org/10.1145/3136014.3136029).
- [S6] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.
- [S7] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
- [S8] Tobi Vollebregt. Declarative specification of template-based textual editors. Master’s thesis, Delft University of Technology, Delft, The Netherlands, April 2012. doi:<http://resolver.tudelft.nl/uuid:8907468c-b102-4a35-aa84-d49bb2110541>.
- [S9] Tobi Vollebregt, Lennart C. L. Kats, and Eelco Visser. Declarative specification of template-based textual editors. In Anthony Sloane and Suzana Andova, editors, *International Workshop on Language Descriptions, Tools, and Applications, LDTA ‘12, Tallinn, Estonia, March 31 - April 1, 2012*, 1–7. ACM, 2012. doi:[10.1145/2427048.2427056](https://doi.org/10.1145/2427048.2427056).
- [S1] Jeff Smits and Eelco Visser. Flowspec: declarative dataflow analysis specification. In Benoît Combemale, Marjan Mernik, and Bernhard Rumpe, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, 221–231. ACM, 2017. doi:[10.1145/3136014.3136029](https://doi.org/10.1145/3136014.3136029).