Declare Your Language

Eelco Visser

TU Delft

June 12, 2015 – June 14, 2015

This book is on github at https://github.com/MetaBorgCube/declare-your-language

Contents

Preface

This is a book about declarative language definition with the Spoofax Language Workbench. The aim of Spoofax is to separate the various concerns of language definition and implementation and provide high-level declarative meta-languages for each concern. These meta-languages are declarative in the sense that they abstract from the *how* of language implementation and focus on the *what* of language design. For example, "what is the syntax of my language?", instead of "how do I implement a parser for my language?". Thus, a language designer should not be distracted by language implementation details.

Resources This book is a response to the demand for better documentation for Spoofax. This demand suggests a sparsity of material. However, there is rather a lot of material available online. The metaborg.org website provides information about installation and documentation about the individual meta-languages. Furthermore, there is a large number of research papers about Spoofax and its meta-languages; see for example the online bibliography for this book. All that material can be rather overwhelming for new users. It may not be quite clear where to start. Furthermore, since Spoofax and its predecessor Stratego/XT have been under development for more than 15 years, the (syntactic) details of papers may be out-of-date. Hence, the goal of this book is to provide a single coherent guide to the basic concepts for new users of the workbench. This book draws on the available material and provides references to papers and web pages for further reading.

In Progress This is a book in progress, just as Spoofax itself is very much in progress. (Although Spoofax is much further along than this book :-) Spoofax is the product of an ongoing research project that investigates the nature of software language definition. We are continuously experimenting with better abstractions for various concerns. As a result, new features are being added to Spoofax and we are also always working on the guts of the workbench. Therefore, working with Spoofax can be a somewhat rocky experience.

Indeed, while I start writing this book in June 2015 using the nightly-build descendant of Spoofax 1.4, Spoofax is about to enter a new era. Within the coming months, we are planning to switch to a new version of Spoofax based on a complete overhaul of the internal architecture of the workbench. We are replacing IMP, the framework that provided the binding of Spoofax meta-languages to Eclipse, with Spoofax Core, a framework that defines IDE services independently from Eclipse. This will allow us to target other IDE containers such as NetBeans and IntelliJ (modulo engineering work) as well as support robust command-line implementations of languages defined in Spoofax. So there is a risk that some of the text of this book and example project that comes with it, will have to be rewritten some.

GitHub This book and the accompanying Spoofax projects are on github. I will consider pull requests with minor or major contributions to the book and accompanying projects.

– Eelco Visser

June 14, 2015

Chapter 1

Getting Spoofax

In this chapter we discuss how to install Spoofax and setting up language projects.

1.1 Eclipse + Spoofax

Spoofax is an Eclipse plugin. The regular way to install an Eclipse plugin is to use its update site to add the plugin to an existing Eclipse installation. However, Spoofax requires a few tweaks to be applied to the Eclipse configuration and it requires the separate installation of Java 7 or later. To avoid all this hassle, Spoofax is now also distributed as a complete Eclipse installation with Spoofax pre-installed and all configurations set corectly. The download page

`http://metaborg.org/download`

provides a link to the integrated distributions. Note that this distribution is currently only available for the bleeding edge continuous build version of Spoofax.

Download the `spoofax-<os>-<arch>-jre.zip` for your computer's operating system and architecture, unzip, and launch the Eclipse application inside.

The first thing that Eclipse will ask is which Workspace to use. The Workspace is the default directory where projects are created. Just create a new directory with an appropriate name (e.g. `Workspace-Spoofax`) in an appropriate location in your file system.

The first thing that I do when installing a new Eclipse is changing its appearance. In the Eclipse menu choose `Preferences`. In the dialog window go to `General > Appearance` and choose theme `Classic`. This is completely optional though.

Another setting that is useful to adjust is that for refresh. In the search box in the Preference dialog type 'refresh'. Under `Startup and Shutdown` select `Refresh workspace on startup`. Under `Workspace` unselect `Build automatically` and select `Refresh on access`.

The default font size is configured to be 11pt, which is too small for my eyes. Adjust the font size in the `Preferences > General > Appearance > Colors and Fonts`, and there select `Basic > Text Font` and choose something appropriate. I find 14pt Monaco to work out pretty well.

JRE Error:
At the first time that I try out the Eclipse with pre-packaged JRE, I get the following error:

> 'Update Installed JREs' has encountered a problem. Resource '/.org.eclipse.jdt.core.external.folders' already exists.

After ignoring the error, Spoofax appears to work fine. It is not clear at this point whether that is due to the fact that I had already installed JRE7.

1.2 Creating a Project

To start a new language with Spoofax you need to create an Eclipse project.

In the `File` menu select `New > Project ...`. In the dialog window select `Spoofax editor project` and hit the `Next >` button. This presents the `Spoofax Editor Project` wizard dialog in which you should indi-

cate the name and file extension of your project and language. This information is used to instantiate all the files that are needed in a Spoofax project.

The wizard will create a project directory in your Workspace with the following properties:

Project name This will be the name of your project and the directory that contains it.

Language name This is the name of your language, which means that it will be used as the basis for several file names.

Plugin ID and package name This is the name of the Java package and plugin that is generated from your project.

File extensions This is the file extension that the program files in your language will have

Generate .gitignore file Of course you will maintain your project's version history in git. Check the box so that generated code that needs not to be versioned is ignored by git.

Generate minimal project only Check this box to start with a fresh language. In this book I will walk you through building the various elements that you need for your language.

Choosing the name for your language is important. Unfortunately, the name that you choose will be hardwired at many places in your project. Therefore, **renaming** your project and language afterwards **is virtually impossible**. The usual way to achieve a renaming is to create a new project with the right name and manually copy over the files from your old project.

1.3 Importing an Existing Project

Another way to use Spoofax is to import an existing project. The github repository for this book provides a series of example projects which

are the basis for the text in the book. To use those projects check out the git repository and import the projects into Eclipse as follows. In the `File` menu select `Import ...`. In the import dialog select `General > Existing Project into Workspace`. Browse to the `declare-your-language/languages` directory and select a project to import (or select the entire directory, which will allow you to import all projects at once). In the `Projects:` are select all projects that you want to import. Hit the `Finish` button.

Importing a project will add it to your workspace without copying it to the Workspace directory.

You can remove a project from your workspace using `Edit > Delete`. This will not remove the files from the file system, unless you select that option using the check box.

1.4 Building the Project

After creating a project and later after changing a project, you will need to build it in order to use it. From the `Project` menu select the `Build Project` entry or invoke it through the short-cut, which is Alt-Command-B on the Mac. When building a project, the `Console` window will pop up and shows the build log. If that log ends with `BUILD SUCCESSFUL` all is good and your language has been built *and* deployed in your Eclipse instance, ready to be tested.

1.5 Structure of a Spoofax Project

The Spoofax Editor Project wizard generates a complete Eclipse plugin project for your language. All that is left to do is fill in the language-specific bits. That is great, but the sheer number of directories and files in a project may seem rather overwhelming. However, it is not all that bad. You can ignore most of this stuff, certainly at first. Let's have a look what the wizard has generated.

- `.cache/`: A cache of intermediate results produced by processing

the language definition. You should never have to look at this.

- `.externalToolBuilders/`: Automatically generated Ant files for building stuff.
- `.settings/`: Some Eclipse settings
- `editor/`: This directory contains `*.esv` files, which configure various aspects of the IDE for your language. Here you can change the color that syntax highlighting gives to certain tokens of your syntax, or define the outline view for programs in your language. A basic definition of these configurations is generated automatically, so you can ignore this for now. But we will get back here.
- `editor/java`: This sub-directory of the `editor/` directory is unrelated to the configuration files. It contains some project-specific Java code inserted by the wizard that binds it to the language-independent Spoofax framework. The directory is also used as target for Java code generated from the DynSem meta-language for dynamic semantics. And later on in the book we will add some glue code for initialization of DynSem-based interpreters.
- `icons/`: This is where icons to be used in the outline view are stored. It is empty by default.
- `include/`: This directory contains files that are generated from the syntax definition for the language. (Eventually these files will end up in the `src-gen` directory.)
- `lib/`: This directory contains the common run-time library for Spoofax projects. (Eventually this should be a binary dependency.)
- `META-INF/`: This directory contains the manifest with configuration information for building the Eclipse plugin.
- `src-gen/`: This directory contains code generated from the language definition.

- `syntax/`: This is where one typically puts the modules making up the syntax definition. In today's Spoofax, syntax definitions are defined using the SDF3 syntax definition formalism.
- `target/`: This directory contains the class files resulting from the compilation of Java code.
- `trans/`: This is where one usually puts the transformations defining the non-syntactic aspects of a language definition, including source-to-source transformations, interpretation, and code generation. All these aspects used to be defined using the Stratego transformation language. However, we will see that name binding and type checking are now done using the NaBL and TS meta-languages, and that operational semantics is defined using the DynSem meta-language.
- `utils/`: This directory contains Spoofax Java libraries.

Then there are some files at the top level of the project:

- `.classpath`: The Java class path for the project
- `.gitignore`: A specification of the (generated) files that can be ignored by git version management.
- `.project`: The file that makes the project directory into an Eclipse project.
- `build.properties`: Some parameter bindings for the Ant build that determines some of the directories above, and where they could be changed. But we will just stick to the standard layout.
- `build.generated.xml`: The generated Ant build file that defines the tasks for compiling language definitions.
- `build.main.xml`: The project-specific Ant build file that binds the generated build file to the project-specific properties. The file is instantiated by the wizard and is one of the places where your language name gets used.

- `plugin.xml`: Configuration of the Eclipse plugin for your language.
- `pom.xml`: A Maven file. In the next version of Spoofax (see Preface), building and dependency management will make heavy use of Maven.

In summary, only the `editor/`, `syntax/`, and `trans/` directories contain language-specific code. The other directories contain either standard Spoofax code that is copied into the project or code that is generated from language definitions.

1.6 Language Concerns

The structure of a Spoofax project does not reveal the conceptual structure of a Spoofax language definition. In the rest of this book we will be primarily be studying the definition of aspects of a language using declarative meta-languages. We distinguish the following concerns:

Tests As with any form of software development, developing a test suite is useful as a partial specification and for catching regressions. In Chapter 2 we study the SPT testing language.

Syntax A syntax definition describes the syntactically well-formed sentences (programs) of a language *and* the structure of these programs. Since all other operations on programs are driven by this structure, syntax definition are the corner stone of language definitions in Spoofax. We will study syntax definition in SDF3 in Chapter 3.

Transformation The parser derived from a syntax definition turns well-formed programs into abstract syntax trees. A wide range of semantic manipulations of programs can be expressed as transformations on abstract syntax trees. In Chapter **??** we will study the definition of basic transformations such as desugarings using the Stratego transformation language. In early versions of Spoofax, all semantic concerns were adressed using Stratego. In recent years

we have been working to add higher-level languages that capture our understanding of particular aspects of semantics specification.

Names Abstract syntax trees do not take into account the graph structure induced by names in programs. Names are the key technique to facilitate abstraction in programming languages. Name resolution is concerned with resolving uses of names with declarations of names. In most tools, name resolution requires a programmatic encoding of the name binding rules of a language. In Chapter **??** we will study the definition of name binding rules using the NaBL name binding language, which abstracts from the implementation of name resolution algorithms.

Type Constraints Many languages apply restrictions to the set of programs that is considered valid beyond the syntactic well-formedness contraints imposed by a grammar. Such restrictions are typically formalized in terms of a type system. In Chapter **??** we study the formalization of such constraints using the TS type system specification language.

Dynamic semantics Language workbenches traditionally use code generation to define the semantics of a language. For many scenarios that is the appropriate thing to do. However, the definition of a code generator often obscures the intended dynamic semantics of a language. Thus, morally, it is a good idea to specify the dynamic semantics of a language directly. Such a specification can then be used to reason about the correctness of the code generator. Going further, the specification of the dynamic semantics may be interpreted directly to produce an *interpreter* for the language. In Chapter **??** we study the DynSem DSL for the specification of the dynamic (operational) semantics of programming languages.

1.7 Getting the Example Projects

The next chapters discuss the support that Spoofax provides for these language concerns. Each chapter uses one or more example projects to

illustrate the presented concepts. Download the projects from languages to follow along.

Chapter 2

Testing

Testing is indispensible during language development to validate that your language definition corresponds to your mental design of the language. Automatic testing is also indispensible for maintenance of your language; does your language extension still support all the original use cases?

In this chapter we discuss how to test a Spoofax language definition using the SPT testing language [4, 5] and using plain example files. You can find the code for this chapter in example project LanguageA. The project defines a language with the name `LangA`.

2.1 Testing with SPT

We start by adding a `test/` directory to our project and in that directory we create `test/expr-syntax.spt` to contain tests for the expressions in our language.

An SPT test module first declares its name and the language that its tests are about:

```
module expr-syntax
language LangA
```

Next it declares the start symbol with respect to which the tests will be parsed:

```
start symbol Expr
```

For each kind of program fragment that should be tested separately, such as expressions here, the corresponding non-terminal symbol should be declared as start symbol in the SPT file *and* in the syntax definition, as we will see in the next chapter.

After these initial declarations follows a series of tests of the form

```
test <name> [[ <fragment> ]] <property>
```

where `<name>` is the descriptive name of the test (an arbitrary string), `<fragment>` is the language fragment to test, and `<property>` defines the property that should be tested. For example, the test

```
test addition [[
  1 + 2
]] parse succeeds
```

declares a test with name `addition` that states that the string `1 + 2` should be parsed succesfully.

Tests can also be negative, i.e. require that some operation fails. For example, the test

```
test double plus [[
  1 ++ 2
]] parse fails
```

states that the string `1 ++ 2` should *not* be parsed successfully. That is, the test would *fail* if the string would be parsed successfully.

In addition to testing for parse success or failure, SPT tests can require a range of other properties. We will see some of these in further chapters.

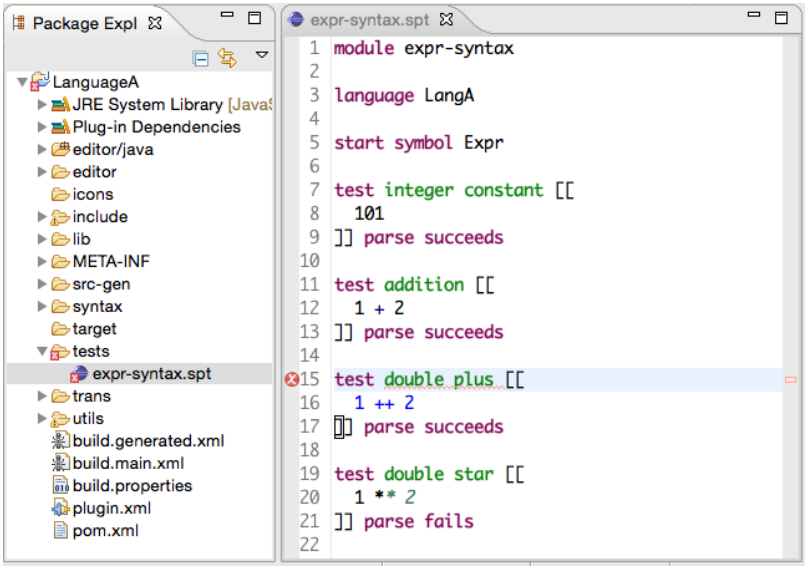

Figure 2.1: Live evaluation of SPT test module with failing test case.

2.2 Running Tests

SPT tests are evaluated automatically when the test module is open in Eclipse. Thus, after modifying and building a language definition, all tests in open test modules are updated immediately. The editor identifies using error markers the tests that fail as illustrated in Fig. 2.1.

Fig. 2.2 illustrates another feature of SPT. The code fragments in a test is presented with all the features of a regular program editor. For example, the syntax highlighting follows the syntax of the program under test and syntax errors according to the syntax under test are hightlighted.

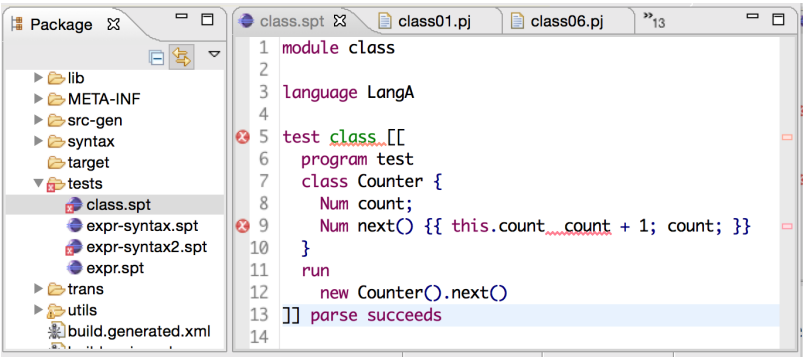

Figure 2.2: Language-specific syntax highlighting and syntax checks in test templates.

2.3 Examples

In addition to SPT tests, I typically maintain an `examples/` directory with example programs in the language under development. Such examples are useful to experience the IDE under development and exercise the various operations being defined on programs.

2.4 Further Reading

The OOPSLA 2011 papers [4, 5] provide a complete description of the design and implementation of SPT. The SPT page on the metaborg site provides a complete overview of the testing features currrently supported by the language.

Chapter 3

Syntax

Spoofax takes a *syntax first* approach to language definition.

We will illustrate the material in this book with a series of Spoofax projects names `LangA`, `LangB`, `LangC`, etc.

3.1 The Start Symbol

changing the start symbol in main.esv

3.2

The syntax

In this project

```
module
context-free syntax
  Exp.Var = ID
  Exp.Mul = [[Exp] * [Exp]] {left}
  Exp.Add = [[Exp] + [Exp]] {left}
context-free priorities
  Exp.Mul > Exp.Add
```

templates [10]

3.3 Error Recovery

[3]

3.4 Further Reading

[9]: definition of SDF2

[6]: Pure and declarative syntax definition: paradise lost and regained

[**?**]: An Algorithm for Layout Preservation in Refactoring Transformations

[1]: Parse table composition

[2]: syntax of AspectJ

[7, 8]: disambiguation filters

Bibliography

[1] M. Bravenboer and E. Visser. Parse table composition. In D. Gasevic, R. Lämmel, and E. V. Wyk, editors, *Software Language Engineering, First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers*, volume 5452 of *Lecture Notes in Computer Science*, pages 74–94. Springer, 2009. 21

[2] M. Bravenboer, Éric Tanter, and E. Visser. Declarative, formal, and extensible syntax definition for AspectJ. In P. L. Tarr and W. R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*, pages 209–228. ACM, 2006. 21

[3] M. de Jonge, L. C. L. Kats, E. Visser, and E. Söderberg. Natural and flexible error recovery for generated modular language environments. *ACM Transactions on Programming Languages and Systems*, 34(4):15, 2012. 21

[4] L. C. L. Kats, R. Vermaas, and E. Visser. Integrated language definition testing: enabling test-driven language development. In C. V. Lopes and K. Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 139–154. ACM, 2011. 16, 19

[5] L. C. L. Kats, R. Vermaas, and E. Visser. Testing domain-specific languages. In C. V. Lopes and K. Fisher, editors, *Companion to the 26th*

Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011, pages 25–26. ACM, 2011. 16, 19

[6] L. C. L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 918–932, Reno/Tahoe, Nevada, 2010. ACM. 21

[7] P. Klint and E. 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. 21

[8] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 2002. 21

[9] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997. 21

[10] T. Vollebregt, L. C. L. Kats, and E. Visser. Declarative specification of template-based textual editors. In A. Sloane and S. Andova, editors, *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*, page 8. ACM, 2012. 20