Declare Your Language

Eelco Visser

TU Delft

Version 0.4

June 12, 2015 – June 22, 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 [19]. 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predecessor Stratego/XT have been under development for more than 15 years, the (syntactic) details of papers may be out-of-date.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.

If you are an academic reader, note that this book is *not a primary source*. All

material in this book is based on previously published research. By all means do cite this book, but that does not absolve you from doing your homework and citing the proper primary sources. The *Further Reading* sections in this book should help you in identifying those sources. And the online bibliography referred to above provides you with ready made bibtex entries.

In Progress This is a book in progress, just as Spoofax itself is very much in progress. (Although surely 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, especially if you are using the latest features, which we are prone to advertise.

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 15, 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.

JRE Error:
When first running the Eclipse with pre-packaged JRE, you may get the following error:

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

This should be a one time error during unpacking of the JRE, which you can safely ignore. Spoofax should work fine and the problem should not be reported again on restarting Eclipse.

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` 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.

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 indicate 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 4 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 with SPT

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 [17, 18] 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.

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

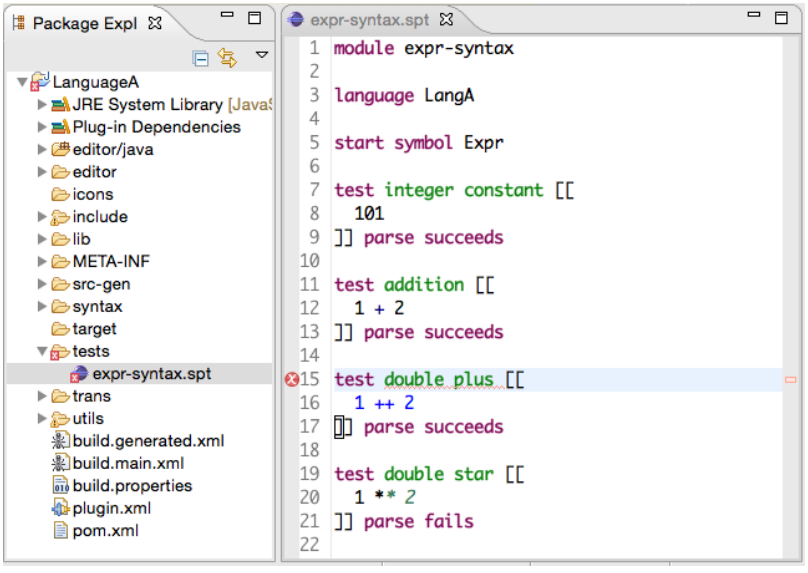

Figure 2.1: Live evaluation of SPT test module with failing test case.

in open test modules are updated immediately. The editor identifies using error markers the tests that fail as illustrated in Figure 2.1.

An alternative way of running tests is by invoking the Spoofax Test Runner (via the Transform menu), which invokes all test in the current test suite or all test suites in the project. As illustrated in Figure 2.2, the test runner then shows a list of all tests colored green or red to indicate that they passed or failed.

Figure 2.3 illustrates another feature of SPT. The code fragment in a test is handled as text in the language under test, separate from the testing language itself. For example, the syntax highlighting follows the syntax of the program under test, keywords of the testing language are not keywords of the language under test, and syntax errors according to the syntax under

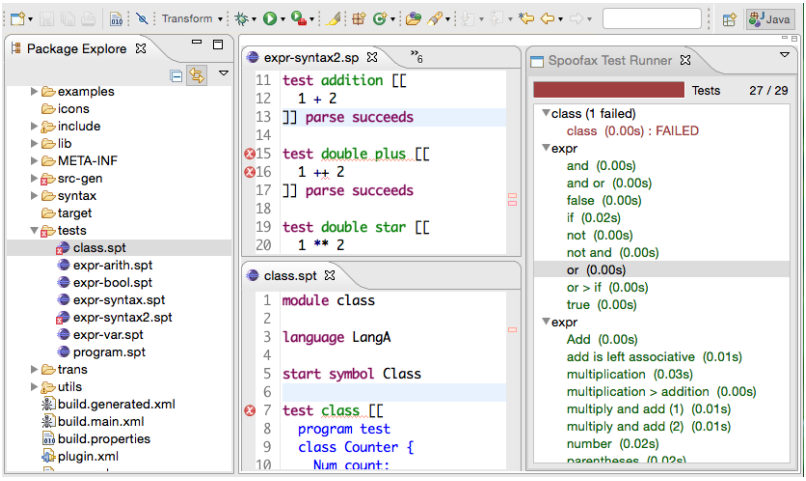

Figure 2.2: Running SPT tests using the Spoofax Test Runner invoked from the Transform menu.

test are hightlighted.

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/SPLASH 2011 papers [17, 18] 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.

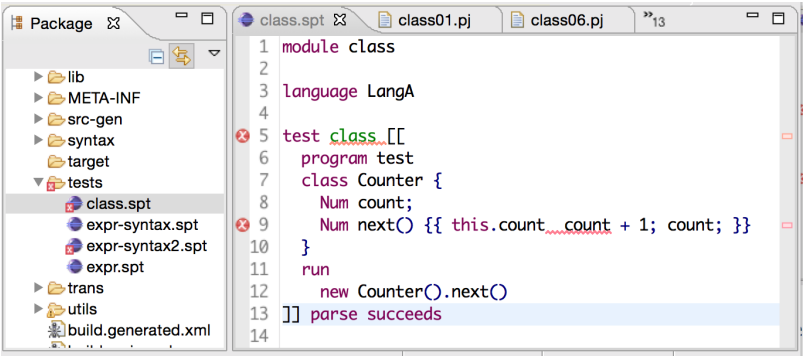

Figure 2.3: Language-specific syntax highlighting and syntax checks in test templates.

Chapter 3

Syntax Definition with SDF3

In this chapter we study declarative syntax definition with SDF3. Spoofax takes a *syntax first* approach to language definition. All semantic operations that we will consider in the rest of the book work on abstract syntax trees. Instead of separately describing a grammar, an algebraic signature, and a mapping from parse trees to abstract syntax trees, an SDF3 syntax definition describes all three at once. Indeed, as we will see, from just an SDF3 syntax definition, we can generate a range of artifacts including the abstract syntax signature, error recovery rules, a parser, a mapping from parse trees to abstract syntax trees, a pretty-printer (mapping from abstract syntax trees to text), syntax highlighting rules, and folding rules.

Developing and testing a syntax definition in Spoofax are seamless. As illustrated in Figure 3.1, a language and programs in the language can be edited in the same Eclipse environment, providing a quick turn around time for changes to the language. The workflow for developing a syntax definition is simple: make a change to the definition, build the project, and watch the open editors for SPT tests and example programs being re-analyzed, or invoke the Spoofax Test Runner to run all tests in the project.

In this chapter we use example project LanguageA as illustration. The project defines the syntax of a small Java-like language styled after an assignment that comes with Krishnamurthi's Programming Languages book [23].

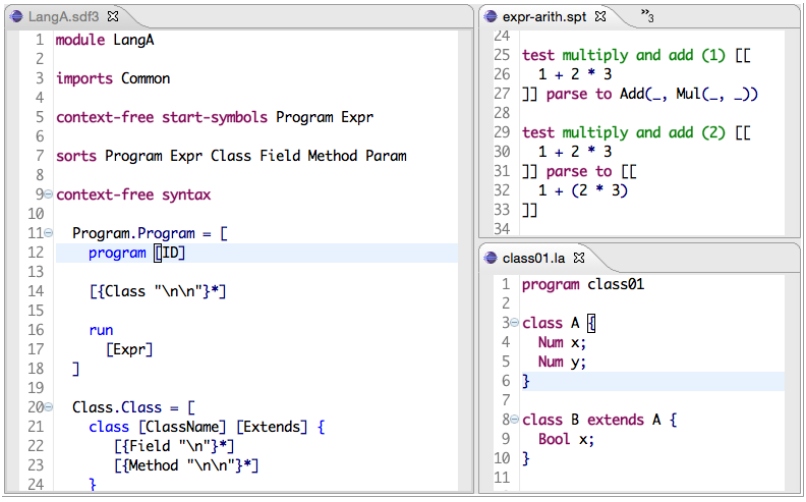

Figure 3.1: Developing and testing syntax definition in the same IDE context.

3.1 Configuration

As we indicated in Chapter 1, the name that you choose for your language project is important since it ties a bunch of things in the configuration of your project together. The main module of the syntax definition in directory syntax/ has the name of the language. Thus, we find syntax/LangA.sdf3 in the example project. Furthermore, the language name is used as prefix of the `esv` configuration files in the editor/ directory. The main configuration module is editor/LangA.main.esv. It declares the file extension to be used by programs in the language (`la`) and the start symbol to be used for such files. By default the start symbol is set to `Start`. For `LanguageA` I have changed it to `Program`, since that is the sort in the syntax definition that represents the content of a file.

3.2 Structure of a Syntax Definition

An SDF3 syntax definition consists of a collection of modules with the following outline:

```
module [ModuleName]
imports [ModuleName*]
context-free start-symbols [Symbol*]
[SyntaxSection*]
```

A module may import other modules using their declared name. The start symbols tell the parser with which syntactic sorts to start parsing. While start symbols may be declared in any module, they are typically declared in the main module of language. As noted above, when changing the start symbol to something else than the default `Start`, this should be changed in the `main.esv` module as well.

Let's have a look at the definition of LanguageA. Its main syntax module defines the sorts `Program` and `Expr` as start symbols and imports the modules `Programs`, `Classes`, and `Expressions`:

```
module LangA
imports Programs Classes Expressions
context-free start-symbols Program Expr
```

3.3 Concrete and Abstract Syntax

The programs in LanguageA have a name, then define a list of zero or more classes, and finally an expression to be executed in the context of these class definitions. The following is a minimal program in the concrete syntax of the language that executes the expression `1` in the context of no classes:

```
program program01 run 1
```

Here is the same program in abstract syntax representation:

```
Program("program01", [], Num("1"))
```

The abstract syntax tree is represented as a term with constructors such as `Program` and `Num` creating tree nodes, and strings to represent lexemes such as identifiers and numbers.

An SDF3 definition defines both the concrete syntax and abstract syntax views of a language. Here is a (plain version of) the `Programs` module:

```
module ProgramsPlain
imports Classes Expressions
sorts Program
context-free syntax
  Program.Program = [program [ID] [Class*] run [Expr]]
```

The module imports the syntax of `Classes` and `Expressions` and defines a *context-free syntax production* to define the syntax of programs. In general, an SDF3 production has the form

```
  Sort0.C = [lit3 [Sort1] lit2 [Sort2] lit3 ...]
```

where `Sort0` is the non-terminal sort being defined, `C` is the abstract syntax tree constructor, and the brackets contain the body of the production. The body is a sequence of literal symbols, whitespace, and non-terminal sorts. The notation for productions uses inverted quotation. Instead of quoting the literal symbols, the sorts are quoted. Thus, a so-called template production is equivalent to a normal context-free grammar production of the form:

```
  Sort0 = "lit3" Sort1 "lit2" Sort2 "lit3"
```

Thus, our `Program` production is equivalent to the context-free grammar production

```
  Program = "program" Class* "run" Expr
```

defining the concrete syntax notation of programs. By the way, the $*$ in `Class*` is the Kleene-star and denotes a sequence of zero or more strings matching the `Class` non-terminal sort.

So, what is the use of the constructor? As indicated above, an SDF3 definition not only defines the concrete syntax (the notation), but also the

```
module src-gen/signatures/Programs-sig

imports
  src-gen/signatures/Classes-sig
  src-gen/signatures/Expressions-sig


signature
  sorts
    Program

  constructors
    Program : ID * List(Class) * Expr → Program
```

Figure 3.2: Algebraic signature derived from syntax definition module `Program`.

abstract syntax, the tree structure representing the underlying structure of the notation. Thus, for each SDF3 production there is a corresponding algebraic signature declaration derived by stripping the literals and whitespace from the production. For the general production form above, we would get the following constructor declaration:

```
C : Sort1 * Sort2 → Sort0
```

That is, given abstract syntax trees `t1` and `t2` of sorts `Sort1` and `Sort2`, respectively, `C(t1, t2)` is a well-formed term of sort `Sort0`.

Figure 3.2 shows the Stratego module `Programs-sig` with the signature declaration corresponding to our `Programs` modules. The constructor declaration for `Program` describes the structure of the abstract syntax term that we saw above. Spoofax automatically generates a signature module in the `src-gen/signatures` directory of the project for each SDF3 module.

The parser generated from the underlying context-free grammar internally produces a *derivation* (also known as *parse tree*) that uses context-free grammar productions as node constructors. Following the correspondence

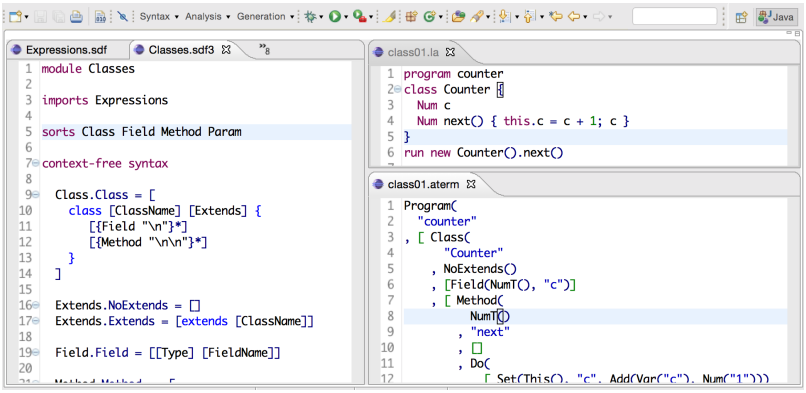

Figure 3.3: Abstract syntax term `class01.aterm` for `class01.la` obtained using the `Show Abstract Syntax` entry in the `Syntax` menu.

between productions and constructors sketched above, such a parse tree is converted to an abstract syntax tree automatically.

In the generated editor for your language, you can get the abstract syntax term of a program using the `Show Abstract Syntax` entry in the `Syntax` menu, as illustrated in Figure 3.3. This is a great tool for 'debugging' your syntax definition. Rather than tracing the behaviour of the parser, you can inspect how your syntax definition maps program texts to abstract syntax tree. And if that mapping is not to your liking, you can adapt the syntax definition, build, and see how the AST is updated. We will see in Section 3.6 how this tool can be used to inspect ambiguities in your syntax definition.

We have seen that by extending standard context-free grammar rules with constructor declarations, we get a parser, an algebraic signature (abstract syntax tree schema), and a mapping from parse trees to abstract syntax trees for free. But in addition to constructors, there is another difference between context-free grammar rules and template productions.

3.4 Template Layout

The inverted quotation templates used as the body of an SDF3 production can be seen as esthetically pleasing by getting rid of the double quotes for quoting literals. However, we replace the double quotes by square brackets (or angle brackets `<Expr>`), and it requires more of those. And the esthetic argument will undoubtely run into people with different tastes. So, the argument for inverted quotation cannot just be esthetics. And indeed it is not.

Consider the concrete and abstract syntax of the `counter` program in Figure 3.4. Figure 3.5 defines the complete syntax of classes, fields, and methods, except for the syntax of `Block` that makes the bodies of methods. The syntax of a class is defined by the following production:

```
Class.Class = [
  class [ID] [Extends] {
     [{Field "\n"}*]
     [{Method "\n\n"}*]
  }
]
```

That is, a class has a name, an extends clause, a list of zero or more fields, and a list of zero or more methods. `Class` is both the non-terminal sort and the constructor for class terms. Thus, an abstract syntax term for class has the form `Class(name, extends, fields, methods)`.

The production above is equivalent, in terms of parsing and abstract syntax to the much simpler looking production

```
Class.Class =
  [class [ID] [Extends] { [Field*] [Method*] }]
```

What is the use of the multi-line layout of the production above? First, the production is layed out as one might format a concrete class in a program. This (arguably) helps the readability of the syntax definition. For various reasons it is necessary to map an abstract syntax tree back to a readable textual representation. It is not sufficient to create a flat concatenation of the literals and lexicals from the production, since that would result in a single

```
program counter
class Counter {
  Num c
  Num next() { this.c = c + 1; c }
}
run new Counter().next()
```

```
Program(
  "counter"
, [ Class(
      "Counter"
    , NoExtends()
    , [Field(NumT(), "c")]
    , [ Method(
          NumT()
        , "next"
        , []
        , Do(
            [ Set(This(), "c", Add(Var("c"), Num("1")))
            , Var("c")
            ]
          )
        )
      ]
    )
  ]
, Call(New("Counter"), "next", [])
)
```

Figure 3.4: Counter class in concrete and abstract syntax.

line of text. It would useful to deduce from the productions of a grammar a mapping to text that is readable and pretty, i.e. with newlines in the right places and proper indentation. No such thing exists, as far as I know. (But it would be great if someone would come up with an algorithm!) Therefore, in order to produce a useful pretty-printer, we need to specify where to put the newlines and indentation. In Stratego/XT [5] this was done by generating a default pretty-printer from the grammar and then manually adapting that

```
module Classes

imports Expressions Common

sorts Class Field Method Param

context-free syntax

  Class.Class = [
    class [ID] [Extends] {
      [{Field "\n"}*]
      [{Method "\n\n"}*]
    }
  ]

  Extends.NoExtends = []
  Extends.Extends = [extends [ID]]

  Field.Field = [[Type] [ID]]

  Method.Method =  [
    [Type] [ID]([{Param ", "}*]) [Block]
  ]

  Param.Param = [[Type] [ID]]
```

Figure 3.5: Syntax of classes with fields and methods.

to get a good result. This was tedious work and had to be re-done on each change of the grammar. As a result, pretty-printers were often out of sync with the grammar. This is the problem that template productions address.

The layout (whitespace) in the template is used as an example-based specification for formatting programs. That is, newlines and spaces (for separation and indentation) in a template are interpreted for *formatting* of ASTs. For parsing this whitespace is ignored. Or rather, it is not taken as literally as it is in formatting. Whitespace between two symbols in a production is taken to signify the presence of optional layout (whitespace

and comments) between those symbols.

Spoofax generates from an SDF3 definition a pretty-printer in the `src-gen/pp/` directory of your project. The pretty-printer consists of Stratego rewrite rules that map an abstract syntax term to a term in the abstract syntax of the Box formatting language. For example, the generated rule for `Class` has the form

```
prettyprint-Class :
  Class(t1, t2, t3, t4) → [..t1'..t2'..t3'..t4'..]
  with t1' := <prettyprint-ID> t1
  with t2' := <prettyprint-Extends> t2
  with t3' := <prettyprint-Field> t3
  with t4' := <prettyprint-Method> t4
```

That is, it calls the pretty-print rules for the sub-terms, and combines the results using combinators of the Box language. I'm omitting the details of the Box language here, since for most purposes the automatically generated pretty-printer should be sufficient. If you are curious have a look at the generated files, such as `src-gen/pp/Classes-pp.str`.

The generated pretty-printer for your language can be invoked directly from the `Format` entry in the `Syntax` menu of the editors for your language. For example, using the pretty-print rules generated from the productions in Figure 3.5, the program in Figure 3.4 is formatted as follows:

```
program counter

class Counter  {
   Num c
   Num next() {
     this.c = c + 1;
     c
   }
}

run
  new Counter().next()
```

Tweaking Formatting The formatting of productions can be tweaked by adapting their templates. The use of newlines and indentation should be obvious from the `Class` example. We can learn some further techniques from the `Method` production in Figure 3.5:

```
Method.Method =  [
  [Type] [ID]([{Param ", "}*]) [Block]
]
```

The separation of symbols by a space in the template means that they will be separated by a space in the formatted text as well. But the *lack of a space separation* is significant as well. For example, this template specifies that the name of the method and the subsequent parenthesis should *not* be separated by a space, nor should there be spaces after the open parenthesis and before the closing parenthesis of the parameter list. However, following good typographic practice, the commas in the list of parameters *should* be followed by a space.

By the way, note that the notation `{Param ","}*` denotes a sequence of zero or more `Param`s *separated by* commas. Compare this to the usual way to encode lists with separators in grammars.

Code Completion The layout information in templates is not just useful for pretty-printing, but is also applied in the production of code completion templates. For each (template) production, Spoofax generates a template such as the following for the `Class` rule (in the `src-gen/completions/Classes-esv.esv` module):

```
completion template Class : "class ID Extends { }" =
  "class " <ID:ID> " " <Extends:Extends> " {\n\t"
     (cursor) "\n}" (blank)
```

The layout directives in the template (space, newlines, tabs) are derived from the layout in the production template.

Given such templates, the editor for your language supports syntactic code completion. When hitting the Control-Space key combination in an editor, the editor proposes a list of possible completions at that point in

the program.

Discussion: User-Defined Preferences Template-based formatting for a language hard-wires the definition of formatting. Possibly, you might want to provide the users of your language with the possibility of adapting the formatting to their taste. This scenario is currently not supported by Spoofax. It is conceivable that such a feature could use the template-based preferences as language defaults and as an indication of the choice points for such a preference dialog.

However, one could also adopt the point of view of the Go language and require all programs committed to version control to be formatted with the same standardized formatting rules. This ensures that all programs look the same and no time is wasted on petty formatting wars. Of course, this requires the language is suitable for such standardized formattting; for (functional) languages with large expressions, good formatting often requires insight in the meaning of the program.

Summary To summarize, Figure 3.5 illustrates the basic features that we have learned about SDF3 so far:

- Modules divide a syntax definition in (reusable, importable) parts.
- The syntax of a non-terminal is defined by multiple productions, one for each alternative.
- A production defines the name of the constructor used in the construction of the corresponding abstract syntax tree node.
- The template body of a production uses inverted quotation to capture a program pattern.
- Sequences and sequences with separators are concisely captured by the `A*` and `{A sep}*` symbols.
- The whitespace in a template body informs the generation of pretty-print and code completion rules.

```
module BasicExpressions
imports Common
sorts Expr
context-free syntax
  Expr     = [([Expr])] {bracket}
  Expr.Num = INT
  Expr.Min = [-[Expr]]
  Expr.Add = [[Expr] + [Expr]]
  Expr.Sub = [[Expr] - [Expr]]
  Expr.Mul = [[Expr] * [Expr]]
```

Figure 3.6: Definition of arithmetic expressions.

3.5 Expression Syntax

Let's try to apply these features to the definition of expression syntax. The module in Figure 3.6 defines some common operators for arithmetic expressions. Note that these productions exactly define the abstract syntax structure that we would like, which is confirmed by the succeeding tests in Figure 3.7. A test such as

```
test Mul Add (1) [[
  1 * (2 + 4)
]] parse to Mul(Num("1"), Add(Num("2"), Num("4")))
```

not only tests that parsing succeeds, as we saw in Chapter 2, but also matches the tree produced for the code fragment against the expected term in the test.

However, the `BasicExpressions` definition fails the tests in Figure 3.8, which contains tests of the form

```
test multiplication > addition (1) [[
  1 + 2 * 3
]] parse to [[
  1 + (2 * 3)
]]
```

```
module expr-arith
language LangA
start symbol Expr

test number [[
  1
]] parse to Num("1")

test number [[
  30128
]] parse to Num("30128")

test Add [[
  16 + 21
]] parse to Add(Num("16"), Num("21"))

test Mul [[
  1 * 2
]] parse to Mul(Num("1"), Num("2"))

test parentheses [[
  (1 + 2)
]] parse to Add(Num("1"), Num("2"))

test Mul Add (1) [[
  1 * (2 + 4)
]] parse to Mul(Num("1"), Add(Num("2"), Num("4")))

test Mul Add (2) [[
  (1 * 2) + 4
]] parse to Add(Mul(Num("1"), Num("2")), Num("4"))
```

Figure 3.7: Testing structure of arithmetic expressions

Instead of specifying an abstract syntax term as expected result, these tests use a concrete syntax criterion, declaring that both expressions should result in the same abstract syntax term.

```
module expr-arith-prec
language LangA
start symbol Expr

test multiplication > addition (1) [[
  1 + 2 * 3
]] parse to [[
  1 + (2 * 3)
]]

test multiplication > addition (2) [[
  1 * 2 + 3
]] parse to [[
  (1 * 2) + 3
]]

test addition is left associative [[
  1 + 2 + 3
]] parse to [[
  (1 + 2) + 3
]]

test addition, subtraction is left associative [[
  1 - 2 + 3
]] parse to [[
  (1 - 2) + 3
]]
```

Figure 3.8: Testing precedence of expressions

The problem is that the `BasicExpressions` grammar is *ambiguous*; there are two (or more) possible parses for the same expression.

Ambiguity is not so much a problem for SDF3. It happily accepts an ambiguous grammar and generates a parser for it. When applying the parser to unambiguous sentences such as those in Figure 3.7, it will return the one possible abstract syntax term. In case the parser encounters an ambiguous sentence, such as in the tests in Figure 3.8, it returns a *parse*

forest representing all possible parse trees. For example, for the sentence `1 + 2 * 3` it produces the term

```
amb([ Mul(Add(Num("1"), Num("2")), Num("3"))
    , Add(Num("1"), Mul(Num("2"), Num("3")))])
```

where the `amb([..])` (ambiguity) node represents a list of all possible parses for the sentence. Such ambiguities can be shared. For example, in case the ambiguity concerns a fragment of the whole program, the ambiguity is restricted to the smallest part of the tree that contains it. For example, using the ambigous expression as body of a method

```
Num next() { 1 + 2 * 3 }
```

leads to an abstract syntax with an ambiguity as sub-term:

```
Method(NumT(), "next", [], Do([
  amb([ Mul(Add(Num("1"), Num("2")), Num("3"))
      , Add(Num("1"), Mul(Num("2"), Num("3")))])]))
```

Inspecting Ambiguities A Spoofax generated editor gives a warning when a sub-term is ambiguous. To inspect and understand the ambiguity, it is most useful to inspect the abstract syntax term that it generates. This can be done through the `Show abstract syntax` entry in the `Syntax` menu. Usually it helps to reduce the text to a minimal example that exhibits the ambiguity. The constructors involved in the ambiguity indicate which productions are involved.

After making a diagnosis of the ambiguity, it is typically necessary to solve the ambiguity by disambiguating the grammar. There are several approaches to doing this.

3.6 Disambiguation

There are several techniques for resolving ambiguities. The basic and universal technique is to transform the grammar such that it no longer exhibits the ambiguity. How to do that depends on the nature of the ambiguity. For expressions as seen above, there is a standard grammar

```
module PrecedenceExpressions
imports Common
sorts Expr
context-free syntax
  Expr.Add   = [[Term] + [Expr]]
  Expr.Sub   = [[Term] - [Expr]]
  Expr       = Term
  Term.Mul   = [[Factor] * [Term]]
  Term       = Factor
  Factor.Min = [-[Factor]]
  Factor.Num = INT
  Factor.Par = [([Expr])]
```

```
constructors
  Add : Term * Expr → Expr
  Sub : Term * Expr → Expr
      : Term → Expr
  Mul : Factor * Term → Term
      : Factor → Term
  Min : Expr → Factor
  Num : INT → Factor
  Par : Expr → Factor
```

Figure 3.9: Encoding precedence in grammar and corresponding algebraic signature.

transformation technique, which involves introducing a new non-terminal sort for each precedence level. For the arithmetic expressions, this results in the grammar of Figure 3.9.

This technique works and produces a deterministic, non-left recursive grammar. However, it is rather tedious to develop, requires inventing names for precedence level (and using them consistently), and adding a new operator requires figuring out how to fit in the precedence chain. In addition, the algebraic signature corresponding to this grammar also introduces the intermediate sorts (Figure 3.9).

```
module ArithmeticExpressions
imports Common
sorts Expr
context-free syntax
  Expr     = [([Expr])] {bracket}
  Expr.Num = INT
  Expr.Min = [-[Expr]]
  Expr.Add = [[Expr] + [Expr]] {left}
  Expr.Sub = [[Expr] - [Expr]] {left}
  Expr.Mul = [[Expr] * [Expr]] {left}
context-free priorities
  Expr.Min > Expr.Mul > {left: Expr.Add Expr.Sub}
```

Figure 3.10: Disambiguation using associativity and priority.

Associativity and Priority To avoid these encoding problems SDF3 provides disambiguation by means of associativity and priority declarations. In Figure 3.10 these are used to disambiguate the arithmetic expressions we saw before.

An associativity annotation on a production indicates whether it is left or right associative. That is, whether an operator application `x o y o z` should be parsed as left associative (`(x o y) o z`) or as right associative (`x o (y o z)`).

The priority relation > declares which production has higher priority (precedence). In the specification of the relation productions are identified using the pair non-terminal and constructor name. Thus, the usual precedence for arithmetic expressions is defined by the following priority declarations:

```
Expr.Min > Expr.Mul > {left: Expr.Add Expr.Sub}
```

Limitations The interpretation of a priority declaration such as `Expr.Mul > Expr.Add` is that terms constructed with `Expr.Add` may not be (direct) children of terms constructed with `Expr.Mul`. This nicely captures the idea of precedence between binary operators. However, the approach has a limitation in the definition of unary prefix operators of lower priority.

Take for example the following productions defining a let binding construct:

```
context-free syntax
  Expr.Let  = [let [{Bind "\n"}*] in [Expr]]
  Bind.Bind = [[Type] [ID] = [Expr]]
context-free priorities
  Expr.Add > Expr.Let
```

The let binding is given lower priority than the addition. That is the right thing to do if we consider an expression like `let Num x = 3 in x + 4`, which will be parsed as

```
Let([Bind(NumT(),"x",Num("3"))], Add(Var("x"),Num("4")))
```

That is, the expression is parsed as `let Num x = 3 in (x + 4)` rather than as `(let Num x = 3 in x)+ 4`.

However, the priority declaration is too strong if we consider the expression `4 + let x = 3 in x`. The only way to parse this expression is as

```
Add(Num("4"), Let([Bind(NumT(),"x",Num("3"))], Var("x")))
```

However, this term is ruled out by the priority declaration.

This poses a limitation on the coverage of SDF2 and SDF3 regarding the specification of existing languages. For example, languages in the ML family have let constructs that are treated in this manner. There are solutions to the problem, but not as elegant as the declaration of associativity and priority rules.

As a workaround, users of our example language have to 'disambigutate' the expression explicitly using parentheses as `4 + (let x = 3 in x)`. Arguably this is a prefered way of writing the expression anyway.

The complete syntax definitions of the expressions of our example languages are listed in Section 3.10.

3.7 Lexical Syntax

So far, we have completely ignored the lexical syntax of languages. We have used the `ID` and `INT` sorts to introduce identifiers and integer literals in our language, but have not looked at their definition, nor have we looked at the specification of whitespace and comments. We got away with that by importing the `Common` module that the Spoofax wizard includes in the `syntax/` directory of a new project. It defines the lexical syntax of common lexical categories such as identifiers, strings, whitespace, and comments. For many languages, it will suffice to use these. But if you need to tweak the lexical syntax, or need to add new categories then it is useful to know how it is done. Fortunately, it is not difficult at all.

Figure 3.11 contains the content of module `Common`. The lexical syntax section contains productions for lexical syntax sorts. The productions are general context-free grammar productions of the form `[NT] = [Symbol *]` defining an alternative for the non-terminal `NT`. The difference between lexical syntax productions and context-free syntax productions is that the former typically do not use template notation, but rather a plain sequence of symbols (but they may use templates). Furthermore, lexical syntax productions do not define a constructor since their parse tree is flattened to a single string when producing an abstract syntax tree.

Character Classes In addition to string literals such as `"class"` that we encountered in context-free productions, lexical production also use *character classes* that denote the choice between a range of characters. For example, the production for identifiers

```
ID = [a–zA–Z] [a–zA–Z0–9]*
```

uses the character classes `[a–zA–Z]` and `[a–zA–Z0–9]`.

Layout Separation The key difference between lexical syntax and context-free syntax is that the symbols of context-free productions are separated by optional layout. That is, a context-free syntax production such as

```
Exp.Add = Exp "+" Exp
```

```
module Common

lexical syntax
  ID              = [a-zA-Z] [a-zA-Z0-9]*
  INT             = "-"? [0-9]+ {prefer}
  STRING          = "\"" StringChar* "\""
  StringChar      = ~[\"\n]
  StringChar      = "\\\""
  StringChar      = BackSlashChar
  BackSlashChar   = "\\"
  LAYOUT          = [\ \t\n\r]
  CommentChar     = [\*]
  LAYOUT          = "/*" InsideComment* "*/"
  InsideComment   = ~[\*]
  InsideComment   = CommentChar
  LAYOUT          = "//" ~[\n\r]* NewLineEOF
  NewLineEOF      = [\n\r]
  NewLineEOF      = EOF
  EOF             =

lexical restrictions
  // Ensure greedy matching for lexicals
  CommentChar   -/- [\/]
  INT           -/- [0-9]
  ID            -/- [a-zA-Z0-9\_]

  // EOF may not be followed by any char
  EOF           -/- ~[]

  // Backslash chars in strings may not be followed by "
  BackSlashChar -/- [\"]

context-free restrictions
  // Ensure greedy matching for comments
  LAYOUT? -/- [\ \t\n\r]
  LAYOUT? -/- [\/].[\/]
  LAYOUT? -/- [\/].[\*]
```

Figure 3.11: The `Common` SDF3 module included in new Spoofax projects, defining the syntax of common lexical categories.

is sugar for

```
Exp.Add = Exp LAYOUT? "+" LAYOUT? Exp
```

which states that the symbols of an addition can be separated by whitespace or comments. This desugaring is *not* applied to lexical syntax productions, which entails that the symbols making up a lexeme cannot be separated by layout.

Lexical Disambiguation When parsing is implemented using a separate scanner based on a finite automaton, it typically implements greedy matching for tokens. That is, when a potential identifier token is followed by a letter, the automaton will continue and add that letter to the sequence of characters of the identifier. In a scannerless parser based on generalized parsing, the parser can make the decision that there are two possible parses, one where the letter is included in the identifier, and one where it is not. For example, take the following grammar for function application in a functional language:

```
Exp.Var = ID
Exp.App = [[Exp] [Exp]] {left}
```

Clearly, a string `f x y` denotes the term `App(App(Var("f"),Var("x")), Var("y"))` in this language. However, without counter measures, the string `fxy` *also* denotes that term. In fact, it may ambiguously denote the terms `Var("fxy")`, `App(Var("fx"),Var("y"))`, `App(Var("f"),Var("xy"))`, and `App(App(Var("f"),Var("x")),Var("y"))`.

Greedy matching is modeled in SDF3 by means of follow restrictions that rule out the scenario that a character is not included in the sequence of characters of a token. For example, the follow restriction for identifiers

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

states that an identifier may not be followed by a letter, digit, or underscore.

Reserved Words Another form of lexical ambiguity is caused by the overlap between identifiers and keywords. For example, the Boolean constant `true` also matches the syntax of identifiers. This form of ambiguity is addressed by means of `reject` productions

```
lexical syntax // reserved words
  ID      = Keyword {reject}
  Keyword = "Num"
  Keyword = "Bool"
  Keyword = "true"
  Keyword = "false"
  Keyword = "this"
  Keyword = "new"
```

Layout-Sensitive Syntax Languages such as Haskell and Python have layout-sensitive syntax. That is, newlines and indentation are taken into account for disambiguation. SDF3 supports Erdweg's layout constraints to define layout-sensitive languages [14].

3.8 Error Recovery

When programming in an IDE, the edited code is most of the time in a syntactically incorrect state. Nonetheless, even when our program is in such a state, we would like to get editor services such as syntax highlighting, type checking, and reference resolution. However, such services require an abstract syntax tree to work with, which cannot be delivered by a failing parser. Therefore, it is crucial that a parser for an IDE supports error recovery. Spoofax realizes parse error recovery by generating an extension of the user provided syntax definition that makes the grammar more *permissive*. This permissive grammar is then interpreted by a back-tracking extension of the JSGLR parser that uses permissive productions when parsing with regular productions fails. For a full treatment of this topic see the TOPLAS 2012 article [10]. An understanding of the approach is not needed for the casual SDF3 user. Spoofax automatically generates a permissive grammar and deploys it in the editor for your language. To get an impression of the generated rules, have a look at the generated `include/LangA—`

`Permissive.def` syntax definition in the LanguageA project.

3.9 Further Reading

The goal of declarative syntax definition is to enable language designers to think about syntax in terms of the (tree) structures underlying programs instead of in terms of parsing algorithmics [20]. SDF3 is the third generation Syntax Definition Formalism that takes declarative syntax definition as its guiding design principle.

The original *Syntax Definition Formalism SDF* [15] introduced the specification of lexical syntax and context-free syntax in a single formalism, the adaptation of Tomita's generalized LR-parsing algorithm to programming language grammars [25, 26], the automatic mapping of parse trees to compact abstract syntax trees, and disambiguation by means of associativity and priorities. The second generation SDF2 truly integrated lexical and context-free syntax by using a single formalism (context-free grammar productions) for the specification of both [32, 30] and supporting this by the introduction of scannerless, generalized (LR) parsing [31]. This required the introduction of lexical disambiguation techniques such as reject productions and follow restrictions. The third generation SDF3 introduced the notion of template productions for the derivation of pretty-printers and code completion schemas [39]. Furthermore, SDF3 first formalized the declaration of abstract syntax tree constructors as part of production rules and their use in shortcuts for the specification of priorities. (In the application of SDF in ASF+SDF, trees were never presented in abstract syntax form. In SDF2 production annotations were used to attach constructor names to productions, but these had no formal status in the language.)

The notion of *disambiguation filters* provides a general framework to formalize the separation of grammar productions, defining the structure of sentences, and disambiguation rules, selecting a tree from a parse forest [22, 27].

Pretty-printing using the box language is implemented in the libstratego-gpp Stratego library, based on the work of Merijn de Jonge [9]. The

general approach of generating pretty-printers was first explored in the ASF+SDF MetaEnvironment [38, 29]. The pretty-printers generated by Spoofax are not (yet) true formatters, since they erase the whitespace and comment in the original program. However, Spoofax provides all the ingredients for layout preserving formatters as developed to support refactoring transformations [11].

SDF3 features modules to divide a syntax definition into reusable parts. The implementation of SDF3 is modular to some extent. Generation of signatures, completion rules, and pretty-printers are translated module by module. However, the core of the implementation, the generation of parse tables, is not modular due to the non-compositional nature of LR parse table generation. The parse table composition approach of Bravenboer and Visser [7] was shown to yield considerable performance gain for practical grammars, but does not yet support cross-module priority declarations, and is therefore not yet integrated in Spoofax.

Determining whether a context-free grammar is ambiguous is an undecidable problem. However, using a generalized parsing algorithm such as GLR or GLL makes it possible to work with ambiguous grammars and explore ambiguity as it is encountered in practical programs. In recent years a number of algorithms and tools such as AmbiDexter [1] have been developed that automate this exploration attempting to either verify unambiguity or find counter examples. Spoofax has not yet integrated such tools.

Scannerless generalized parsing is an excellent basis for supporting the combination of languages. This has been demonstrated in a wide range of applications, including meta-programming with concrete object syntax [34], embedding of domain-specific languages in a general purpose (Java) language [6], and the syntax of AspectJ [8]. The SugarJ extensible language [13] uses SDF2 (and Stratego) to define modules that extend the host language with new syntax. Many of these extension projects rely on parameterized modules in SDF2, a feature that has not been ported to SDF3.

3.10 Complete Syntax Definition

This section gives the complete syntax definition for our example language (except for the `Common` module shown in Figure 3.11).

```
module LangA
imports Programs Classes Expressions
context-free start-symbols Program Expr
```

```
module Programs

imports Classes Expressions

sorts Program

context-free syntax

  Program.Program = [
    program [ID]

    [{Class "\n\n"}*]

    run
      [Expr]
  ]
```

```
module Classes

imports Expressions Common

sorts Class Field Method Param

context-free syntax

  Class.Class = [
    class [ID] [Extends] {
      [{Field "\n"}*]
      [{Method "\n\n"}*]
    }
  ]
```

```
  Extends.NoExtends = []
  Extends.Extends = [extends [ID]]

  Field.Field = [[Type] [ID]]

  Method.Method =  [
    [Type] [ID]([{Param ", "}*]) [Block]
  ]

  Param.Param = [[Type] [ID]]
```

```
module Expressions

imports Common

sorts Expr
context-free syntax // arithmetic
  Expr     = [([Expr])] {bracket}
  Expr.Num = INT
  Expr.Min = [-[Expr]]
  Expr.Add = [[Expr] + [Expr]] {left}
  Expr.Sub = [[Expr] - [Expr]] {left}
  Expr.Mul = [[Expr] * [Expr]] {left}

context-free syntax // booleans
  Expr.True  = [true]
  Expr.False = [false]
  Expr.Not   = [![Expr]]
  Expr.And   = [[Expr] && [Expr]] {assoc}
  Expr.Or    = [[Expr] || [Expr]] {assoc}
  Expr.Eq    = [[Expr] == [Expr]] {non-assoc}
  Expr.Neq   = [[Expr] != [Expr]] {non-assoc}

  Expr.If = [
    if( [Expr] )
      [Expr]
    else
      [Expr]
  ] {right}

sorts Block
```

```
context-free syntax

  Expr = Block
  Block.Do = [
    {
      [{Seq ";\n"}*]
    }
  ]

  Seq = Expr
  Seq.Skip = []

context-free syntax // variables

  Expr.Var = ID

  Expr.Let = [
    let [{Bind "\n"}*]
     in [Expr]
  ]

  Bind.Bind = [[Type] [ID] = [Expr]]

context-free syntax // objects

  Expr.Get   = [[Expr].[ID]]
  Expr.Set   = [[Expr].[ID] = [Expr]]
  Expr.Call  = [[Expr].[ID]([{Expr ", "}*])]

  Expr.New   = [new [ID]()]
  Expr.This  = [this]
  Expr.Null  = [null [ID]]
  Expr.Cast  = [([ID])[Expr]]

  Type.NumT   = [Num]
  Type.BoolT  = [Bool]
  Type.ClassT = ID

context-free priorities

  {Expr.Get Expr.Call Expr.Cast}
```

```
  > {Expr.Not Expr.Min}
  > Expr.Mul
  > {left: Expr.Add Expr.Sub}
  > Expr.Eq
  > Expr.And
  > Expr.Or
  > Expr.Set
  > {Expr.If Expr.Let}

lexical restrictions
  "new" —/— [a—zA—Z]

lexical syntax // reserved words
  ID      = Keyword {reject}
  Keyword = "Num"
  Keyword = "Bool"
  Keyword = "true"
  Keyword = "false"
  Keyword = "this"
  Keyword = "new"
```

3.11 Complete Signature

This section gives the complete algebraic signature derived from the syntax definition of our example language.

```
module src—gen/signatures/LangA—sig

imports
  src—gen/signatures/Programs—sig
  src—gen/signatures/Classes—sig
  src—gen/signatures/Expressions—sig


signature
```

```
module src—gen/signatures/Programs—sig

imports
  src—gen/signatures/Classes—sig
```

```
  src-gen/signatures/Expressions-sig


signature
  sorts
    Program

  constructors
    Program : ID * List(Class) * Expr → Program
```

```
module src-gen/signatures/Classes-sig

imports
  src-gen/signatures/Expressions-sig
  src-gen/signatures/Common-sig


signature
  sorts
    Class Field Method Param

  constructors
    Class
      : ID * Extends * List(Field) * List(Method) → Class
    NoExtends : Extends
    Extends   : ID → Extends
    Field     : Type * ID → Field
    Method    : Type * ID * List(Param) * Block → Method
    Param     : Type * ID → Param
```

```
module src-gen/signatures/Expressions-sig

imports
  src-gen/signatures/Common-sig


signature
  sorts
    Expr

  constructors
```

```
  Num : INT → Expr
  Min : Expr → Expr
  Add : Expr * Expr → Expr
  Sub : Expr * Expr → Expr
  Mul : Expr * Expr → Expr

constructors
  True  : Expr
  False : Expr
  Not   : Expr → Expr
  And   : Expr * Expr → Expr
  Or    : Expr * Expr → Expr
  Eq    : Expr * Expr → Expr
  Neq   : Expr * Expr → Expr
  If    : Expr * Expr * Expr → Expr

sorts
  Block

constructors
        : Block → Expr
  Do    : List(Seq) → Block
        : Expr → Seq
  Skip : Seq

constructors
  Var  : ID → Expr
  Let  : List(Bind) * Expr → Expr
  Bind : Type * ID * Expr → Bind

constructors
  Get    : Expr * ID → Expr
  Set    : Expr * ID * Expr → Expr
  Call   : Expr * ID * List(Expr) → Expr
  New    : ID → Expr
  This   : Expr
  Null   : ID → Expr
  Cast   : ID * Expr → Expr
  NumT   : Type
  BoolT  : Type
  ClassT : ID → Type
```

```
constructors
   : String → Keyword
```

Chapter 4

Transformation with Stratego

The parsers generated from SDF3 syntax definitions produce abstract syntax terms. Semantic manipulation of programs in Spoofax is done by manipulating these terms. In this chapter we study the definition of transformations on (abstract syntax) terms with the Stratego strategic rewriting language. The code in this chapter can be found in the LanguageB project.

4.1 Term Transformations

A desugaring transformation simplifies the programs of a language to use only a subset of the constructs of the language or only simplified applications of constructs. For example, Figure 4.1 shows a program before and after desugaring let bindings and sequential compositions. After desugaring, let bindings contain only a single binding sequential composition 'blocks' contain at most two expresssions (and no 'skips'). Figure 4.2 shows the abstract syntax term of the same program before and after desugaring.

Figure 4.3 defines the syntax of the format of terms that we use here. A term is a string literal, a number literal, a constructor application, or a list of terms (between square brackets.) The algebraic signature corresponding to a syntax definition describes the subset of terms that are well-formed abstract syntax terms.

Thus, defining a transformation such as desugaring on programs, requires

```
program letdo
run
  let Num x = 1
      Num y = x + 1
   in { o.x = x;
        o.y = y;
        o.x + o.y; }
```

```
program letdo
run
  let Num x = 1
   in let Num y = x + 1
    in { o.x = x;
         { o.y = y;
           o.x + o.y } }
```

Figure 4.1: Program before and after desugaring (concrete syntax).

```
Program("letdo", []
, Let([ Bind(NumT(), "x", Num("1"))
      , Bind(NumT(), "y", Add(Var("x"), Num("1")))]
  , Do([ Set(Var("o"), "x", Var("x"))
       , Set(Var("o"), "y", Var("y"))
       , Add(Get(Var("o"), "x"), Get(Var("o"), "y"))
       , Skip()]
    )
  )
)
```

```
Program("letdo", []
, Let([Bind(NumT(), "x", Num("1"))]
  , Let([Bind(NumT(), "y", Add(Var("x"), Num("1")))]
    , Do([ Set(Var("o"), "x", Var("x"))
         , Do([ Set(Var("o"), "y", Var("y"))
              , Add(Get
                  (Var("o"), "x"), Get(Var("o"), "y"))])]
      )
    )
  )
)
```

Figure 4.2: Program before and after desugaring (abstract syntax).

```
module Terms
imports Common
sorts Term Constr
context-free syntax
  Constr   = STRING
  Constr   = ID
  Term.Str = STRING
  Term.Num = NUMBER
  Term.App = [[Constr]([{Term ","}*])]
  Term.Lst = <[<{Term ","}*>]>
```

Figure 4.3: Syntax of terms.

a transformation on the underlying abstract syntax terms. The Stratego language supports the definition of such transformations by means of term rewrite rules.

4.2 Term Rewrite Rules

A term rewrite rule is a schematic description of a transformation on terms. For example, the rule

```
desugar-let :
  Let([b1, b2 | bs], e) → Let([b1], Let([b2 | bs], e))
```

transforms a `Let` term with a list containing at least two bindings (`b1` and `b2`) and some other bindings (`bs`), into a `Let` term with the first binding `b1` and a nested `Let` with the other bindings. In general, a term rewrite rule has the form

```
l : lhs → rhs
```

where `l` is the name of the rule, and `lhs` and `rhs` are *term patterns*. A rewrite rule applies to a term if the left-hand side pattern *matches* the term, and results in the instantiation of the right-hand side pattern, replacing the pattern variables with the terms bound to them by the match.

```
module desugar-rules
imports src-gen/signatures/Expressions-sig

rules

  desugar-do :
    Do([e]) → e  where <not(?Skip())> e

  desugar-do :
    Do([e | es@[_,_|_]]) → Do([e, Do(es)])

  desugar-do :
    Do([Skip(), e]) → e

  desugar-do :
    Do([e, Skip()]) → e

  desugar-let :
    Let([], e) → e

  desugar-let :
    Let([b1, b2 | bs], e) → Let([b1], Let([b2 | bs], e))

  desugar-or :
    Or(e1, e2) → If(e1, True(), e2)

  desugar-and :
    And(e1, e2) → If(e1, e2, False())
```

Figure 4.4: Term rewrite rules for desugaring expressions.

Figure 4.4 gives more examples of desugaring rules on expressions.

Term Patterns and Pattern Matching A term `pattern` is a term extended with variables and list match patterns as formalized by the syntax definition in Figure 4.5. Roughly, a term matches a pattern if it has the same shape as the pattern. That is, a term matches a pattern if its outermost constructor is the same as the outermost constructor of the pattern, and

the sub-terms match the corresponding sub-patterns, or the pattern is a variable. Figure 4.6 gives a precise definition of pattern matching.

4.3 Term Rewriting Strategies

Term rewriting is the process of applying term rewrite rules to (the sub-terms of) a term until a *normal form* is reached. That is, until no more sub-terms match the left-hand side of a rule. In traditional rewrite engines, rules are applied using a standard *strategy* to find *reducible expressions* (or *redices*). However, in practice rewrite rules (for program transformation) may be *non-confluent*, i.e. produce multiple different results depending on the application order, and/or may be *non-terminating*. Circumventing such problems in a traditional rewrite engine requires extending terms with additional information in order to control execution.

In order to provide more control over the rewriting process, Stratego supports *programmable* rewriting strategies, which make it possible to *select* the rules to apply and to determine the search algorithm used to find redices. Indeed, such strategies do not necessarily lead to a (unique) normal form in the traditional sense.

The Stratego library provides a large collection of generic (traversal) strategies. These strategies are generic in the sense that they do not depend on the abstract syntax of a particular language. For example, Figure 4.7 uses the `innermost` strategy from the library to define several desugaring transformations using the desugaring rules from Figure 4.4.

4.4 Defining Strategies

A Stratego strategy is a partial function from terms to terms. A strategy is a partial function since the application to a term may fail. For example, applying a rewrite rule to a term fails if the term does not match the left-hand side of the rule. Strategies are defined in terms of a small set of basic transformation operations:

`id` The identity strategy always succeeds and returns the term to which

```
module Patterns
imports Common Terms
sorts Pat
context-free syntax
  Pat.Pstr = STRING
  Pat.Pnum = NUMBER
  Pat.Papp = [[Constr]([{Pat ","}*])]
  Pat.Plst = <[<{Pat ","}*>]>
  Pat.Pltl = <[<{Pat ","}*> | <Pat>]>
  Pat.Pvar = ID
```

Figure 4.5: Syntax of patterns.

A term matches a term pattern if one of the following cases aplies

- The pattern is a string literal and the term is the same string literal.
- The pattern is a number literal and the term is the same number literal.
- The pattern is a constructor application $\texttt{c(p}_1\texttt{,..,p}_n\texttt{)}$, the term is a constructor application $\texttt{c(t}_1\texttt{,..,t}_n\texttt{)}$ of the same constructor `c`, and the sub-terms $\texttt{t}_1\texttt{,..,t}_n$ match the sub-patterns $\texttt{p}_1\texttt{,..,p}_n$.
- The pattern is a list pattern $\texttt{[p}_1\texttt{,..,p}_n\texttt{]}$, the term is a list $\texttt{[t}_1\texttt{,..,t}_n\texttt{]}$, and the terms $\texttt{t1,..,t}_n$ match the patterns $\texttt{p}_1\texttt{,..,p}_n$.
- The pattern is a list pattern $\texttt{[p}_1\texttt{,..,p}_n\texttt{|p]}$, the term is a list $\texttt{[t}_1\texttt{,..,t}_m\texttt{]}$, $n \leq m$, the terms $\texttt{t}_1\texttt{,..,t}_n$ match the patterns $\texttt{p}_1\texttt{,..,p}_n$, and $\texttt{[t}_{m+1}\texttt{,..,t}_m\texttt{]}$ matches the pattern `p`.
- The pattern is a pattern variable, in which case the term is bound to the variable.

Figure 4.6: Definition of term pattern matching.

```
module desugar
imports src-gen/signatures/Expressions-sig
imports desugar-rules

strategies

  pre-desugar =
    innermost(desugar-let)

  post-desugar =
    innermost(desugar-or <+ desugar-and <+ desugar-do)

  desugar-all =
    innermost(desugar-or <+ desugar-and
              <+ desugar-do <+ desugar-let)
```

Figure 4.7: Strategies for desugaring expressions.

it is applied.

`fail` The failure strategy always fails.

`f` The application of a rewrite rule or strategy `f` to the current term. If successful, returns the instantiation of the right-hand side of the rule.

`<f>p` The application of a rewrite rule or strategy `f` to the term pattern `p`.

`?p` Match the term against the pattern `p`.

`!p` Replace the term by instantiating the pattern `p` (also known as *build*).

These basic transformation operations are combined using strategy *combinators*:

s_1 `;` s_2 The sequential composition of two strategies. s_2 is applied to the term returned by the application of s_1. Fails if either strategy fails.

```
module traversals
strategies

  try(s) = s <+ id
  // try to apply s to a term

  repeat(s) = try(s; repeat(s))
  // repeatedly apply s to a term

  topdown(s) = s; all(topdown(s))
  // apply s to each sub-term in a pre-order traversal

  bottomup(s) = all(bottomup(s)); s
  // apply s to each sub-term in a post-order traversal

  downup(s) = s; downup(s); s
  // apply s on the way down and on the way up

  innermost(s) = bottomup(try(s; innermost(s)))
  // exhaustively apply s in a post-order traversal

  oncetd(s) = s <+ one(oncetd(s))
  // depth-first search

  alltd(s) = s <+ all(alltd(s))
  // apply s to a frontier of sub-terms
```

Figure 4.8: Definitions of traversal strategies (included in the standard Stratego library).

`s1 <+ s2` The ordered choice between two strategies. If the application of `s1` is successful, its result is returned, otherwise `s2` is applied to the original term.

`all(s)` A one-level traversal operator that visit all direct sub-terms of a term, applying `s` to each, returning the application of the original constructor to the transformed sub-terms.

`one(s)` This one-level traversal operator visits the direct sub-terms of

a term from left to right, applying `s` until it succeeds, returning the original term with the first sub-term for which `s` succeeds replaced.

`crush(n,c,s)` Apply strategy `s` to the direct sub-terms of a term and fold the resulting list using the nil and cons operations `n` and `c`.

Figure 4.8 shows the application of these combinators in the definition of several strategies from the standard Stratego library.

4.5 Parameterized and Conditional Rules

In Figure 4.8 we saw that strategy definitions can be parameterized with strategy arguments. The strategy `bottomup(s)` takes a strategy parameter `s` that is instantiated with some transformation to apply to each node of a term. Rules can be parameterized in the same way. Figure 4.9 gives examples of such parameterized rules. For example, the `map(s)` rule applies a transformation `s` to all the elements of a list, and `filter(s)` filters the elements of a list using (the success/failure behaviour of) the transformation parameter `s`. Note that `<s>x` denotes the application of the strategy (parameter) `s` to term (variable) `x`.

Rules and strategies have two parameter lists, one for by-name parameters, and one for by-value parameters, which are separated by a bar `|`. Thus, in general a strategy call has the form $\texttt{f(}s_1\texttt{,..,}s_n\texttt{|}t_1\texttt{,..,}t_m\texttt{)}$, where the s_i are strategy (by-name) parameters, and the t_i are term (by-value) parameters. The bar can be left out if there are no by-value parameters. For example, in the `inc(|n)` strategy takes a term parameter `n` and calls the `map` strategy with a strategy parameter `\ x -> <add>(x, n)\`, which will be applied to each element of the list to which `inc(|n)` is applied.

Rules can have side conditions prefixed with `where` or `with` followed by a strategy expression. Both kinds can compute an intermediate result needed by the next condition or the right-hand side of the rule. (A pattern matching assignment `p := t` matches the result of the `t` against the pattern `p`.) In addition, a `where s` condition tests whether its strategy argument succeeds or fails; in the latter case, the rule fails. For

```
module list-strategies
rules

  map(s) : [] → []
  map(s) : [x | xs] → [<s>x | <map(s)>xs]
  // apply s to elements of a list; fails if one fails

  filter(s) : [] → []
  filter(s) : [x | xs] → [<s>x | <filter(s)>xs]
  filter(s) : [x | xs] → <filter(s)>xs
  // apply s to elements of a list;
  // keep only successful ones

  inc(|n) = map(\ x → <add>(x, n) \)
  // increment all numbers in a list with n

  subst(|x, e) = alltd((Var(x) → e))
  // substitute all occurrences of Var(x) by e

  collect-all(s) :
    t → [t' | ts]
    where t' := <s>t
    with ts := <crush(![], union, collect-all(s))>t
  collect-all(s) :
    t → <crush(![], union, collect-all(s))>t
  // collect set of all sub-terms (transitive) for
  // which s succeeds
```

Figure 4.9: Parameterized rules.

example, the `collect-all(s)` strategy in Figure 4.9 uses the condition `where t' := <s>t` to test whether the application `<s>t` succeeds; if it does not, the next rule is tried. On the other hand, a `with s` condition expects its strategy argument `s` to succeed; if it does not, the program throws an exception. For example, the `collect-all(s)` strategy uses the condition `with ts := <crush(![], union, collect-all(s))>t` to compute the collection from the sub-terms of `t` and expects this to always succeed.

4.6 String Interpolation

The proper way to define a translation from one language to another is through 'code generation by model transformation'. That is, transform the abstract syntax terms from the source language to abstract syntax terms of the target language and then pretty-print the resulting term. This separates program composition from pretty-printing and allows transformations to be applied to the generated target code. Since I'm in a hurry to get to the next chapters, I will only discuss a quick-and-dirty approach to code generation using *string interpolation* (for now), and refer to the further reading section for references to code generation by model transformation.

Code generation by string composition has a bad reputation. Among its many short-comings, the most mundane one is quotation hell. The characters of a string are enclosed in double quotes. Any double quotes to be included in the string need to be escaped. When the language does not support multi-line strings, newlines have to be included as escaped characters. Splicing in strings resulting from a sub-computation, such as a recursive invocation of the translation function, requires breaking the string into sub-strings and composing those with a string concatenation operator. The result is typically quite unreadable.

String interpolation (or string templates) addresses these problems (and is supported by more and more programming languages). In Stratego a string interpolation expression has the form `$[..[..]..]`, where the (multi-line) text between `$[` and `]` is taken as literal characters, while the embedded `[..]` expressions are escapes to splice in the values of variables or the results of sub-computations. (To avoid quotation conflicts one can choose different quotation symbols such as `$<..<>..>`.) For example, Figure 4.10 defines the translation of programs in our example language to XML documents. The body of the string template for the `Program` rule consists of the literal XML pattern for a `<program>` document. Using escapes, the translations of the sub-terms (`name`, `cs`, and `e`) are inserted into the template. Note the difference between the literal `name` attribute and the splicing of the `name` variable in `name="[name]"`, and the application

```
module to-xml
imports src-gen/signatures/Programs-sig
imports to-xml/classes-to-xml
imports to-xml/expressions-to-xml
rules

  to-xml : Program(name, cs, e) →
    $[<program name="[name]">
        <classes>
          [<map(to-xml <+ dbg(|"class: "))> cs]
        </classes>
        <run>
          [<to-xml <+ dbg(|"exp: ")>e]
        </run>
      </program>]
```

Figure 4.10: Using string interpolation to translate programs to XML documents.

of strategy expressions in escapes such as as `[<map(to-xml)> cs]`.

In addition to solving quotation hell, string templates take care of getting indentation right. The common prefix (caused by the indentation of the meta-program) is discarded. On the other hand, relevant indentation, such as the indentation of classes in Figure 4.10, is applied uniformly to the lines in the text spliced into the template.

4.7 Debugging

Unfortunately, Stratego lacks an interactive debugger that let's you step through the execution of your code. So, the usual approach to debugging Stratego programs is pretty classical. First, reduce the input to a transformation to the smallest term that exhibits the unexpected behavior. Second, use good old 'println' debugging to inspect to where a program proceeds and inspect run-time values. The stratego library defines the `dbg(|msg)` strategy, which prints the current term prefixed with `msg`.

```
module LangB-Menu-Transformation
menus
  menu: "Transformation" (openeditor) (realtime)
    action: "Desugar"     = editor-desugar
    action: "Desugar AST" = editor-desugar-ast
    action: "To XML"      = editor-to-xml
```

Figure 4.11: Configuration of Transformation menu.

4.8 Spoofax Builders

We have defined two transformations on the abstract syntax trees of our example language, desugaring and generation of XML documents. Now we would like to apply these transformations in the editor for our language. This requires the definition of *builders*. Builders are actions that can be applied to a program from the editor. The definition of these actions consists of three ingredients. We already took care of the first ingredient, the definition of the transformation to apply.

Second, we need to define menu entries that let us invoke the transformations. To that end, we create a new editor services file `LangB-Menu-Transformation.esv` in the `editor/` directory and import it in the main `LangB-Menus.esv` module. In the module, we declare a menu with the name `Transformation` and containing three 'actions' as show in Figure 4.11. These actions associate a menu entry with a name (e.g. `Desugar`) and a strategy to apply (e.g. `editor-desugar`).

Third, we need to define the glue code that implements the interface between a menu and the transformation strategy. Figure 4.12 defines the strategies that implement the builders for the three menu items in Figure 4.11. Each of these define a transformation from a tuple `(selected, position, ast, path, project-path)` — consisting of the selected AST, the position of the cursor, the entire AST of the editor, the path name of the file, and the path name of the project directory — to a pair `(filename, result)`, consisting of a file name to write the

```
module transformation
imports desugar
imports pp
imports to-xml/to-xml
rules

  editor-desugar:
    (selected, position, ast, path, project-path)
    → (filename, result)
    with
      ext      := <get-extension> path;
      filename :=
        <guarantee-extension(|$[desugared.[ext]])> path;
      result   := <desugar-all; pp-debug> selected

  editor-desugar-ast:
    (selected, position, ast, path, project-path)
    → (filename, result)
    with
      filename :=
        <guarantee-extension(|$[desugared.aterm])> path;
      result   := <desugar-all> selected

  editor-to-xml:
    (selected, position, ast, path, project-path)
    → (filename, result)
    with
      filename :=
        <guarantee-extension(|$[xml])> path;
      result   := <to-xml> selected
```

Figure 4.12: Implementation of builders for the Transformation menu.

result to and the string representation of the result.

Useful strategies from the Stratego library to use here include `get-extension` to get the file extension from a path, and `guarantee-extension(|ext)` to replace the extension of a file name with `ext`.

4.9 Further Reading

Stratego emerged from my experience with term rewriting in the ASF+SDF MetaEnvironment [21, 28]. The ASF formalism uses equations to define algebraic equivalences of the terms induced by an algebraic signature. It was implemented using a rewriting engine based on innermost exhaustive reduction. The need to control the application of rules, prompted the introduction of auxiliary function symbols.

Inspired by the notion of strategies in ELAN [3, 2] and the modal operators of modal logic, Luttik and I introduced one-level traversal combinators (all, some, one) for generic term traversal [24]. These provided the basis for the design of the Stratego transformation language [37] and System S as a core language for rewriting [36]. The Stratego only appeared later in publication [33], although it was used from 1998.

Stratego was first implemented as an interpreter in ML. Using that interpreter a compiler from Stratego to C was defined in Stratego and then bootstrapped. The compiler is organized in a front-end that reduces Stratego Sugar to Stratego Core (which may be stored in .ctree files), and a back-end that translates to a target language. After multiple iterations of the C translation scheme, the back-end of the compiler was ported to produce Java code around 2006, which is used by Spoofax. In addition to a compiler, Stratego is supported by an interpreter implemented in Java.

Programming in the Stratego language is supported by a set of libraries that provide standard transformation utilities, such as parsing and pretty-printing. Originally these libraries were developed under the name of the XT transformation tools. Several iterations of these tools along with the development of the Stratego are described in publications: XT [12], Stratego/XT 0.9 [35], Stratego/XT 0.17 [5]. After the 0.17 release of Stratego/XT, the language was ported to Java and Stratego/XT is now distributed as a JAR, including the compiler and several pre-compiled Stratego libraries. The Stratego/XT distribution is now included in the distribution of the Spoofax Language Workbench [19].

Stratego supports meta-programming with concrete object syntax, which entails that terms in rewrite rules can be written using the concrete syntax of the object language [34]. This approach is a key ingredient of the 'code generation by model transformation' approach [16].

Online Documentation The online StrategoXT Manual provides a fairly complete overview of the Stratego language and the XT toolset [4]. While the tool aspects are not up-to-date with the current Java-based libraries, the Stratego/XT Tutorial provides an extended introduction to the language.

Bibliography

[1] B. Basten and T. van der Storm. Ambidexter: Practical ambiguity detection. In *Tenth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2010, Timisoara, Romania, 12-13 September 2010*, pages 101–102. IEEE Computer Society, 2010. 44

[2] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of elan. *Electronic Notes in Theoretical Computer Science*, 15:55–70, 1998. 66

[3] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. Elan: A logical framework based on computational systems. *Electronic Notes in Theoretical Computer Science*, 4:35–50, 1996. 66

[4] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. *Stratego/XT Reference Manual*, 2003-2008. 67

[5] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008. 27, 66

[6] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In J. M. Vlissides and D. C. Schmidt, editors, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004*, pages 365–383, Vancouver, BC, Canada, 2004. ACM. 44

[7] 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. 44

[8] 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. 44

[9] M. de Jonge. Pretty-printing for software reengineering. In *18th International Conference on Software Maintenance (ICSM 2002), Maintaining Distributed Heterogeneous Systems, 3-6 October 2002, Montreal, Quebec, Canada*, pages 550–559. IEEE Computer Society, 2002. 43

[10] 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. 42

[11] M. de Jonge and E. Visser. An algorithm for layout preservation in refactoring transformations. In A. M. Sloane and U. Aßmann, editors, *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers*, volume 6940 of *Lecture Notes in Computer Science*, pages 40–59. Springer, 2011. 44

[12] M. de Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. *Electronic Notes in Theoretical Computer Science*, 44(2):79–86, 2001. 66

[13] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: library-based syntactic language extensibility. 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 391–406. ACM, 2011. 44

[14] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Layout-sensitive generalized parsing. In K. Czarnecki and G. 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*, pages 244–263. Springer, 2012. 42

[15] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989. 43

[16] Z. Hemel, L. C. L. Kats, D. M. Groenewegen, and E. Visser. Code generation by model transformation: a case study in transformation modularity. *Software and Systems Modeling*, 9(3):375–402, 2010. 67

[17] 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. 15, 18

[18] 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. 15, 18

[19] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. 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 444–463, Reno/Tahoe, Nevada, 2010. ACM. 4, 66

[20] 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. 43

[21] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993. 66

[22] 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. 43

[23] S. Krishnamurthi. *Programming and Programming Languages*. 2014. 20

[24] B. Luttik and E. Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF 1997)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag. 66

[25] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, January 1992. 43

[26] M. Tomita. An efficient context-free parsing algorithm for natural languages. In *IJCAI*, pages 756–764, 1985. 43

[27] 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. 43

[28] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In R. 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*, pages 365–370. Springer, 2001. 66

[29] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering Methodology*, 5(1):1–41, 1996. 44

[30] E. Visser. A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam, August 1997. 43

[31] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997. 43

[32] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997. 43

[33] E. Visser. Stratego: A language for program transformation based on rewriting strategies. In A. Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–362. Springer, 2001. 66

[34] E. Visser. Meta-programming with concrete object syntax. In D. S. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315. Springer, 2002. 44, 67

[35] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer, 2003. 66

[36] E. Visser and Z.-E.-A. Benaissa. A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15:422–441, 1998. 66

[37] E. Visser, Z.-E.-A. Benaissa, and A. P. Tolmach. Building program optimizers with rewriting strategies. In M. Felleisen, P. Hudak, and C. Queinnec, editors, *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 13–26, Baltimore, Maryland, United States, 1998. ACM. 66

[38] E. Visser and M. G. J. van den Brand. From box to TEX: An algebraic approach to the generation of documentation tools. Technical Report P9420, Programming Research Group, University of Amsterdam, July 1994. 44

[39] 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. 43