Creating Domain-Specific Languages with *Simpl*

24th January 2011

Contents

1 Introduction

This document describes *Simpl* – a toolkit for creating textual doman-specific languages (DSLs). The goal of Simpl is to make it easy to use language-oriented programming paradigm when developing enterprise information systems. The main design criteria of the Simpl toolset are the following.

- Visual (editor, IDE) and non-visual (parser, code generator) parts of a DSL implementation must be separate. The non-visual part does not depend on the visual part.
- Non-visual part of the DSL implementation is easily integrable into larger systems and does not make assumptions on how the larger system is implemented.
- Using Simpl, the language developer must be able to create a working DSL implementation with little effort. For a relatively simple language, the time to create a working code generator and an IDE should be less than a day.

Based on these design criteria, the Simpl builds on existing tools by combining them and creating a user-friendly frontend. In particular, Simpl is based on the following tools:

- ANTLR parser generator,
- Eclipse IDE,
- IDE Meta-Toling Platform (IMP),
- Scala programming language,
- ant build system.

The following sections describe the use of Simpl in more detail. The text assumes some familiarity with concepts related to grammars and other tools for implementing formal languages.

1.1 Basic Concepts

A DSL implementation created with Simpl consists of two parts.

- *tool* – the non-visual part of the DSL implementation. The tool part contains the DSL parser and code generator. It is meant to be integrated into a bigger system or a build process.
- *plugin* – the IDE part of the DSL implementation, packaged as an Eclipse plugin. The .jar file also includes the classes that make up the code generator for the DSL.

Most of the DSL implementation revolves around abstract representation of the DSL program (in the following text, it is referred to as Abstract Syntax Tree or AST). In Simpl, the AST is represented as Scala case classes that are derived from DSL grammar. For example, the AST node corresponding to if statement in a typical programming language could be represented using the following Scala class.

```
case class IfStatement(
    var condition: Expression,
    var thenStatement: Statement,
    var elseStatement: Statement)
  extends Statement {}
```

Figure 1 illustrates workflow how Simpl processes DSL programs. The parser converts the textual DSL program to AST. The AST is then processed by other tools. On the one (non-visual) side, the AST is fed to the code generator that produces code in the target language. On the other (visual) side, the AST is used by DSL IDE to provide editing features, such as outline view, hyperlinking, automatic completion etc.

2 Installing Simpl

2.1 Linux

1. Install Java 6 (on Debian and Ubuntu systems, *openjdk-6-jdk* package is sufficient).

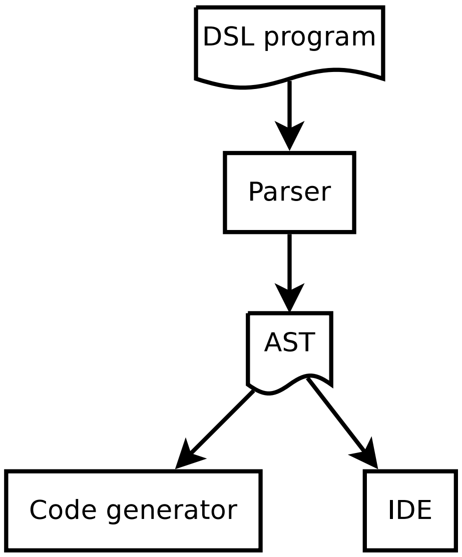

Figure 1: Workflow for processing DSL programs

2. Install ant 1.8 (on Ubuntu, *ant1.8* package is required, the default *ant* package installs 1.7).

3. Download and install Eclipse 3.5 (Galileo) SR2 for RCP/Plugin developers, available from `http://www.eclipse.org/downloads/packages/eclipse-rcpplug-developers/galileosr2`.

4. Install Simpl Eclipse plugin from update site `http://research.cyber.ee/simpl/update-site/`.

5. Install Scala Eclipse plugin from update site `http://www.scala-ide.org/`.

2.2 Windows

1. Download and install JDK 6 (available from `http://www.oracle.com/technetwork/java/index.html`).

2. Download and install Apache Ant 1.8 (available from `http://ant.apache.org/`).

3. Download and install Eclipse 3.5 (Galileo) SR2 for RCP/Plugin developers, available from `http://www.eclipse.org/downloads/packages/eclipse-rcpplug-developers/galileosr2`.

4. Install Simpl Eclipse plugin from update site `http://research.cyber.ee/simpl/update-site/`.

5. Install Scala Eclipse plugin from update site `http://www.scala-ide.org/`.

3 Using Simpl

Using Simpl, creation of a new DSL generally consists of the following steps.

1. Create a new project.
2. Specify the DSL grammar.
3. Create the language runtime (code checker, code generator).
4. Create IDE for the language.

The next sections present these steps based on often-used state machine language from Martin Fowler (see `http://martinfowler.com/dslwip/Intro.html`). The example program in this language is shown in Figure 2.

4 Creating a New Project

4.1 Using the New Project Wizard

Simpl comes with new project wizard that can be called from the Eclipse IDE or from the command line.

To invoke the wizard from Eclipse, just select *File* ▶ *New* ▶ *Project* ▶ *Simplicitas Wizards* ▶ *Simplicitas Project*. In the dialog, fill in the fields and press *Finish*. To invoke the wizard from command line, issue the following command:

```
events
  doorClosed  D1CL
  drawOpened  D2OP
  lightOn     L1ON
  doorOpened  D1OP
  panelClosed PNCL
end

resetEvents
  doorOpened
end

commands
  unlockPanel PNUL
  lockPanel   PNLK
  lockDoor    D1LK
  unlockDoor  D1UL
end

state idle
  actions {unlockDoor lockPanel}
  doorClosed => active
end

state active
  drawOpened => waitingForLight
  lightOn    => waitingForDraw
end
...
state unlockedPanel
  actions {unlockPanel lockDoor}
  panelClosed => idle
end
```

Figure 2: State machine example program

```
java -jar simplicitas-wizard.jar -d <directory> [parameters]
```

You may need to prefix the .jar file with directory where it is stored. The *-d* parameter is used to specify a directory where the new project will be created.

Both wizards take the same parameters that are described in table 1 on page 27 (command-line option only applies to the command-line version of the wizard). After running wizard you should have compiling and working DSL implementation, based on the example grammar selected in the wizard.

4.2 Building Language Implementation

Due to several limitations of the Eclipse build system, the language implementation must be built from the command line (however, it is also possible to invoke the build script from the Eclipse).

The build process of the language implementation can be configured using the file *ant.properties*. The wizard automatically generates two files: *ant.properties* and *ant.properties.sample*. The idea is that the file with .sample extension is included in the source control and serves as example that the users can use to develop *ant.properties* file with their own personal settings. The *ant.properties* file is specific to particular working environment and thus should not be committed to the VCS (it is recommended to add *ant.properties* to the ignore list of your VCS software).

The default build system for wizard-generated DSL implementation takes the required libraries from two separate places. First, non-eclipse-related libraries are pulled from Maven2 repository (see *maven.repo.local* parameter in *ant.properties* file). Second, all the Eclipse-related dependencies are pulled from the user's Eclipse installation. Therefore, currently building the DSL implementation requires installation of Eclipse RCP. Hopefully, in future versions this restriction will be lifted. You can set up the location of Eclipse installation using the *eclipse* parameter in *ant.properties* file.

4.3 Using Language Implementation

To use the DSL plugin in Eclipse, just copy the *dslname-plugin.jar* to *plugins* directory of your Eclipse installation. This will associate DSL files (determined by the extension given to new project wizard) with the DSL implementation[1].

[1]Note: if you modify files *MANIFEST.MF* or *plugin.xml* (which is normally not necessary), you must start Eclipse with command-line option -clean to force it to reload the plugin information.

The new project wizard automatically creates a code generator for the DSL. This code generator is located in the *dslname-tool.jar* file. The .jar file only contains the generator classes and not the necessary libraries. It is assumed that the tool part of the DSL implementation is not invoked directly from command line but instead from a build script. See the *tool-run* task in the build.xml file for an example on how to run the code generator.

The code generator can also be called from the Eclipse by right-clicking on a DSL file and selecting *Generate*. To enable this, you need to override the *runGenerator* method in *YourLangConfig* class (see section 7 for more information about customizing the Eclipse-based IDE).

5 Specifying DSL Grammar

Figure 3 shows the full content of the state machine grammar. The next sections explain the details of this.

The Simpl grammar is essentially an ANTLR grammar that is annotated with information about the shape of the AST.

All the rules have name that will be used for an AST node corresponding to the result of applying this rule. The rule name must always begin with an uppercase letter.

The first rule of the grammar becomes the start symbol (i.e., the topmost rule in the grammar and the root of the AST).

5.1 Terminals

5.1.1 Simple rules

Simple terminal rules have the form:

```
terminal RuleName: Pattern;
```

The pattern can consist of the constructs shown in table 2 (here the *P* and *Pn* are patterns).

```
grammar ee.cyber.simplicitas.fowlerdsl.Fowler;
Program: (imports=Import | machines=Machine | externals=ExternalMachine)+;
Import: ’import’ importURI=Str;
ExternalMachine: ’external’ Id;
Machine: ’machine’
    name=Id
    events=EventList
    resetEvents=ResetEvents?
    commands=CommandList
    states=State+
    ’init’ initState=StateRef;
ResetEvents: ’resetEvents’ events=EventRef+ ’end’;
EventList: ’events’ events=Event+ ’end’;
Event: name=Id code=Id;
CommandList: ’commands’ commands=Command+ ’end’;
Command: name=Id code=Id;
State: ’state’
    name=Id
     (’actions’ ’{’ actions=CommandRef+ ’}’)?
     transitions=Transition*
     ’end’;
Transition: event=EventRef ’=>’ state=StateRef;
EventRef: Id;
CommandRef: Id;
StateRef: Id;
option Reference {var ref: NamedItem = null; def id: Id}:
    EventRef | CommandRef | StateRef;
option NamedItem {def name: Id;}: Event | Command | State;
terminal Id: (’a’..’z’|’A’..’Z’|’_’) (’a’..’z’|’A’..’Z’|’_’|’0’..’9’)*;
fragment StrBody: ~(’\\’|’"’) | ’\\’ .;
terminal Str(value: String = {$_.substring(1, $_.length() - 1)}):
’"’ StrBody* ’"’ | ’\’’ StrBody* ’\’’;
fragment MlComment: ’/*’ (~’*’ | ’*’ ~’/’)* ’*/’;
fragment SlComment: ’//’ ~(’\n’|’\r’)*;
hidden terminal Ws: (’ ’|’\t’|’\r’|’\n’|SlComment|MlComment)+;
```

Figure 3: State machine grammar

Pattern	Explanation
’x’ .. ’y’	match any single character between range x and y, inclusively.
’Str’	match the literal string.
.	match any character
P1 P2	match pattern *P1* followed by *P2*
P1 | P2	Match either pattern *P1* or *P2*
~P	negation – match characters not belonging to pattern *P*
P?	match the optional pattern *P*
P*	Match any number of occurrences of pattern *P*
P+	Match one or more occurrences of *P*
(P)	Match *P*. Parentheses can be used to group patterns.
Ref	Reference to another terminal rule or fragment (see section 5.1.2).

5.1.2 Fragments

Fragments can be used to structure the terminal rules (by splitting a big terminal rule into several parts). The syntax of the fragment is almost the same as terminal rule:

```
fragment RuleName: Pattern;
```

The main difference is that fragment rules do not create tokens when parsing (and therefore cannot be called by the context-free rules). Also, AST classes are not generated for fragment rules. Fragment rules can only be called by terminal rule.

5.2 Context-free Syntax

In Simpl, the rules for describing context-free syntax are also used for describing the AST classes.

5.2.1 Context-Free Rules

Simple rules have the form:

```
Name: Pattern;
```

The pattern can contain constructs listed in table 3.

Pattern	Description
'Str'	Matches literal string *'str'*
[name=]Ref	Calls another rule (non-terminal rule, terminal rule or option rule). The part of the AST matched by the rule will be assigned to attribute *name*.
P1 P2	Match pattern *P1* followed by *P2*
P1 | P2	Match either pattern *P1* or *P2*. In this case, the AST class for the rule will include fields from all the options in the pattern (only the fields from the matching pattern are filled in).
(P)	Match *P*. Parentheses can be used to group patterns.
P?	Match the optional pattern *P*
P*	Match any number of occurrences of pattern *P*
P+	Match one or more occurrences of *P*

Table 3: Context-free patterns

5.2.2 Generating AST Classes

The grammar generator will generate an AST class for every rule in the grammar (except for fragment rules). The name of the class will be the name of the rule. Each AST class has attributes corresponding to rule references from this rule. If the references are not named, the name of the attribute will be derived from the type. For example,

```
Foo: Id Str;
```

... will generate class *Foo* with fields *id* and *str*. Attribute names must be unique, thus the following rule is illegal:

```
Foo: Id Id;  // two attributes named 'id'
```

You can also explicitly name the attributes:

```
Foo: myId=Id myStr=Str;
```

The attributes of AST classes are mutable (using keyword *var*). This is done to allow the DSL implementation to post-process the AST after parsing.

5.2.3 Option rules

Option rules have the form:

```
option Name: Rule1 | Rule2;
```

The main difference between simple rule and option rule is that all the simple rules will generate classes that inherit from *CommonNode*. For example:

```
case class Rule1 extends CommonNode {...}
case class Rule2 extends CommonNode {...}
```

However, using the option rule example from above will make *Name* to be a base class for all the options in the rule, resulting in

```
case class Rule1 extends Name {...}
case class Rule2 extends Name {...}
```

Option rules can also be used to shape the class hierarchy of the AST classes. For this purpose, one can create artificial option rules that are not called by any other rules and are therefore strictly not part of the grammar. For example, this can be used to create base classes for all the different types of references in the grammar. You can also use decorations (see section 5.3) to add methods and attributes to the base class.

5.3 Decorating Generated Classes

Both terminal and context-free rules can include code blocks that will be placed inside generated classes.

```
Foo {code}: ...
```

This will place the code inside the class definition:

```
case class Foo(var field: Type, ...) {code}
```

This can be used to add additional fields or methods to the AST classes. For example:

```
terminal IntTerm {
    val value: Int = text.toInt
}: ('0'..'9')+;
```

This generates the following classes:

```
case class IntTerm(text: String) extends TerminalNode {
    val value: Int = text.toInt
    ...
}
```

5.4 Abstract Representation of Program

The parser converts the DSL program into abstract representation that uses Scala case classes to express the DSL program. In addition to the abstract representation, the programmer also has access to concrete representation of the DSL program in the form of token list. This token list contains al the information about source file, including white space and comments.

All the fields in the AST classes are created mutable (using *var* keyword). This allows implementing post-processing step after the parsing.

See ScalaDoc comments for *CommonNode* and *CommonToken* for more information. Also, see the generated AST classes.

5.5 Shaping the Generated Abstract Representation

Simpl uses the *returns* keyword to influence the AST that is created by the parser. The *returns* keyword can be followed by return type, return code block, or both.

5.5.1 Using Return Type

One can specify the return type of the rule using returns keyword, followed by the type. For example:

```
Plus returns Expr: left=Num ‘‘+’’ right=Num;
```

This has the following effects.

- The class *Plus* will be generated exactly as before, containing two constructor parameters *left* and *right*.
- The generated AST class *Plus* will extend class *Expr*.
- If the grammar does not contain rule named *Expr*, then a new abstract class is created:

  ```
  trait Expr extends CommonNode
  ```

- If some rule calls rule *Plus*, then type of the parameter will be *Expr*, instead of *Plus*. For example, take the following rule.

  ```
  UsesPlus: ‘‘eval’’ expr=Plus;
  ```

  This generates the following class. Note that constructor parameter *expr* has type *Expr*, not *Plus*.

  ```
  case class UsesPlus(var expr: Expr) extends CommonNode
  ```

The return type is intended to be used for making the rule return a more general type (i.e., supertype). It cannot be used to change the return type – make the rule return some type that is not related to the rule class.

5.5.2 Using Return Code Blocks

In addition to changing the return type of the rule, it is also possible to modify the AST node returned by the rule using return code blocks (or return expressions). Return code block is a Scala expression that returns an object corresponding to AST node. The expression must be surrounded with curly brackets ("*{}*"). Note that the type of the returned object must match return type of the grammar rule (if not explicitly specified with the *returns* keyword, then class named after the rule).

Return code block is executed in the scope that contains all the parameters of the rule. The AST node corresponding to the rule itself is also accessible using identifier *_self*. For example, the following rule matches parenthesized expressions.

```
ParenExpression
    returns Expression {expr}
    : ‘‘(’’ expr=Expression ‘‘)’’;
```

The code block in this example cleans up the resulting AST. Without the code block, `‘‘(1)’’` would parse as `ParenExpression(Num(‘‘1’’))`. With code block this expression parses as `Num(‘‘1’’)`.

Another example is ML-style function application. Function *f* applied to parameters *a*, *b* and *c* is written as "*f a b c*". However, simply *"f"* is just a variable reference. This construct can be parsed with the following rule.

```
ApplyExpression
    returns Expression
        {if (params.isEmpty) fun else _self}
    : fun=PrimaryExpr (params=PrimaryExpr)*;
```

The following table shows AST for different input strings with and without using the return expression.

Input string	Without return expression	With return expression

`f`	`ApplyExpression(`
	` Id("f"),`
	` List())`
		`Id("f")`

`f a b`	`ApplyExpression(`
	` Id("f"),`
	` List(Id("a"), Id("a")))`
		`ApplyExpression(`
		` Id("f"),`
		` List(Id("a"), Id("a")))`

In general, it is a good idea to keep the return expressions short in order to keep the grammar file readable. It is recommended to put the longer code in a separate Scala file and leave only function calls to return expressions. In order to make code in grammar shorter, the *scalaheader* directive (followed by block of code in curly brackets) can be used to add import statements to the generated Scala code. For example, the following example makes all the members of *GrammarUtils* object accessible from the return code blocks.

```
scalaheader {
import GrammarUtils._
}
```

Objects returned by the return expressions are not limited to AST nodes generated from the grammar. It is possible to define new AST classes and use them from the return expressions. However, this has some restrictions.

- AST classes must be subclasses of *CommonNode*.
- AST node returned from return expressions must belong to subclass of the return type of the rule. If AST node is not subclass of the AST class created for this rule, the return type of the rule must be used.
- When constructing nested AST nodes, the inner nodes must be manually assigned node locations. All the AST nodes contain location of the node in the source file (starting and ending character position, see the documentation of *CommonNode* class for further details). The top-level object returned by return expression is automatically assigned a location. However, for the nodes inside the top-level object the programmer must manually set the location of the node. The following example code parses C-style array access operator *X[Y]* and constructs expression containing pointer arithmetic: **(X + Y)*.

```
ArrayDeref
    returns Expression
        { Deref(
            Plus(array, index).setLocation(_self)
          ) }
    : array=Expression “[“ index=Expression ”]”;
```

Here the *setLocation(_self)* is used to set the source location of the *Plus* AST node. The location does not have to be copied entirely from a single node. The methods *setStart* and *setEnd* can be used to copy starting point from one node and ending point from another node.

5.5.3 Usage Scenarios

The main reasons to use return types and return code blocks are the following purposes.

- Return types can be used to introduce abstract base classes without using artificial option rule.
- Return code blocks can be used to remove unnecessary AST nodes (e.g., rules corresponding to parentheses).
- Return code blocks can be used to generate nice-looking AST for arithmetic operators. See **??** for longer description.

5.6 ANTLR Grammar Options

It is possible to pass grammar-level options to ANTLR. This can be done by including *options* block right after the grammar line. Option syntax is just like in ANTLR, except that instead of curly braces, normal parentheses are used. For example:

```
grammar foo.Bar;
options (backtrack=true; memoize=true;)
...
(rest of the grammar)
```

6 Creating Language Runtime

6.1 The Main Program

The wizard automatically generates a main program for the tool part of the DSL implementation. It takes as command line arguments a list of source files and name of the target directory. By default it runs StringTemplate to generate the code. Using StringTemplate is no obligation though – you can use any other templating engine for code generation.

6.2 Resolving Links

One typical task in processing DSL programs is resolving links between program elements. This information is used both for checking correctness of the DSL program (right after parsing) and in the IDE for implementing hyperlinking service. This section details one possible implementation of this task, using the state machine DSL as an example.

6.2.1 Add References to Grammar

In the state machine language, the referenceable objects are events, commands and states. The first step is to add grammar rules for designating references:

```
EventRef: Id;
CommandRef: Id;
StateRef: Id;
```

Next, take the new non-terminals into use:

```
ResetEvents: 'resetEvents' events=EventRef+ 'end';
Transition: event=EventRef '=>' state=StateRef;
...
```

In principle, it is possible to do without the **Ref* rules and just use the *Id* rule directly:

```
Transition: event=Id '=>' state=Id;
```

However, the intermediate rules are useful because this makes later AST processing simpler: it is possible to determine whether a node is a reference just by looking at its type.

Typically, the reference node contains link to the referenced object. This link can be filled in postprocessing stage and used later in code generation or to implement hyperlinking service in the DSL IDE. In the example, the link is stored in a base class that is generated using synthetic *option* rule:

```
option Reference {
    var ref: NamedItem = null
    def id: Id
}: EventRef | CommandRef | StateRef;
```

In this code, the *ref* is a mutable variable that will be filled after parsing. The *id* method pulls the read-only *id* property to the base class.

6.2.2 Resolve Links During Postprocessing

The links are normally resolved during postprocessing stage. In this stage, the parsed AST is processed to find semantical errors (such as broken links or other consistency issues). Also, the AST can be modified to make the work of code generation and IDE services easier. After postprocessing the AST is used for code generation and for providing the language IDE.

Figure 4 on the next page shows code for *FowlerPostProcess* class that is responsible for resolving links and reporting broken links. The main entry point is method *validate* that walks through the AST and invokes the method *resolve* on every node. *resolve* looks at type of the node and if the node is a reference, tries to resolve it. The *resolveLink* method tries to find the referenced item from the symbol table (*ctx.events*, *ctx.commands* or *ctx.states*). If the referenced item is found, it fills the node's *ref* field. Otherwise, it logs source error.

```
/**
 * Validates the program and does additional post-processing,
 * e.g. resolving links. The processing step updates errors
 * in the ctx variable.
 */
class FowlerPostProcess(val ctx: FowlerCtx) {
  import collection.mutable.Map
  def validate() {
    // Resolve links.
    ctx.tree.walkTree(resolve)
  }

  def resolve(node: CommonNode) {
    node match {
      case ref: EventRef =>
        resolveLink(ctx.events, "event", ref)
      case ref: CommandRef =>
        resolveLink(ctx.commands, "command", ref)
      case ref: StateRef =>
        resolveLink(ctx.states, "state", ref)
      case _ => ()
    }
  }
  def resolveLink[T <: NamedItem](map: Map[String, T],
      kind: String, node: Reference) {
    if (map.contains(node.id.text)) {
      node.ref = map(node.id.text)
    } else {
      ctx.errors += new SourceMessage(
          "Undefined " + kind + ": \"" + node.id.text + "\"",
          SourceMessage.Error, node.id)
    }
  }
}
```

Figure 4: Resolving links in Fowler DSL

7 Creating Language IDE

7.1 Overview

When generating a new language project, the plugin part of the project will contain two files: *YourLangConfig.scala* and *YourLangServices.scala*. The latter contains technical glue code that is responsible for instatiating various language services, referenced from the *plugin.xml* configuration file. In general, the DSL developer should not edit this file and instead use the *YourLangConfig.scala* file for specifying behaviour of the DSL IDE.

Language-specific functionality of the IDE resides in the *YourLangConfig* class that overrides methods in *APluginConfig* class with language-specific implementations. See the *APluginConfig* class for list of all the methods that can be overriden. The following sections explain how to implement various IDE services.

Note: only one instance of the YourLangConfig *class will be constructed during the work of the Eclipse plugin. It is therefore inadvisable to store mutable working data in the* YourLangConfig *class.*

7.2 Editor Assistance

This category provides hints to the code editor about the general syntax of the DSL.

7.2.1 Fences

```
def fences: Array[Tuple2[String, String]]
```

This method is used by the bracket matching feature of the code editor. This method should return list of brackets whose conterpart will be highlighted in the editor. For example, if the DSL uses "()" and "[]" as brackets, then the method could be implemented as

```
override def fences = Array(("(", ")"), ("[", "]"))
```

7.2.2 Prefix for Single-Line Comments

```
def singleLineCommentPrefix: String
```

This method should return prefix that is added or removed by the "toggle comments" feature of the DSL editor. Implement it to return string that is used for comments that start with a comment marker and last until the end of current line (such as `//` in Java).

7.3 Syntax Highlighting

Simpl uses two-level scheme for defining syntax colouring for the DSL. First, you have to create list of all the different syntax elements that you wish to highlight. Typically this would include keywords, comments, strings and various other language elements. In Simpl, syntax highlighting is configurable, the DSL developer has to provide default values for colors and fonts.

Note: Simpl automatically highlights keywords and comments, based on grammar description. Therefore you do not need to do anything about them unless you want to change the default behaviour (such as colouring some keywords differently than others).

First step in implementing syntax colouring is to enumerate all the syntax elements with different colours. This is done by changing variable *colors* in the automatically-generated *YourLangConfig* object. The variable contains map, indexed by a symbol, whose values are descriptions of syntax element. The description consists of three parts: human-readable name, default color (encoded as "*R,G,B*") and default font style (using constants from the *SWT* class). This data is mainly used to build preferences page.

```
val colors = Map(
    'strings -> ("Strings", "0, 128, 0",
        SWT.BOLD | SWT.ITALIC),
    'code -> ("Embedded code", "128, 0, 0",
        SWT.NORMAL))
```

The second step involves overriding the *getTokenColor* method:

```
def getTokenColor(token: GenericToken): Symbol
```

This method takes as input a token from the DSL program source and returns a symbol corresponding to a particular syntax element. *getTokenColor* can only return symbols that are used as keys in the *colors* map. Following is an example implementation, corresponding to syntax element list from the previous example.

```
override def getTokenColor(token: GenericToken): Symbol = {
    val myToken =
        token.asInstanceOf[CommonToken[SimplKind.Kind]]
    myToken.kind match {
        case SimplKind.Str => 'strings
        case SimplKind.Code => 'code
        case _ => null
    }
}
```

7.4 Outline View

Simpl makes it easy to create outline view for a DSL. This is done with the help of three methods:

```
def treeLabel(node: CommonNode): String
def treeImage(node: CommonNode): Image
def addToTree(node: CommonNode): Boolean
```

The *treeLabel* method takes as input an AST node and returns human-readable string that is used to represent this node in the outline view. The *treeImage* method returns 16x16 icon that is displayed next to the outline view item. The recommended way to load the image is in the *initializeImages* method of the *YourLangConfig* class.

```
def initializeImages(addFun: (String, String) => Image)
```

This method is called by the automatically generated framework when the plugin is loaded. In the body of this method, call the *addFun* function to load images that are packaged with the plugin (do not forget to edit the build file to include icon files in the plugin .jar). This function takes as the first parameter key that is used to reference the image in plugin's image registry. The second parameter is path to the image (relative to plugin's directory). The key must be unique for all the loaded images. The loaded images should be saved somewhere where they can be accessed by the *treeImage* method.

The *addToTree* method returns true, if the node should be displayed in the outline view. The default behaviour is that node is displayed in the tree if the *treeLabel* method returns non-null value. If, for some node, *treeLabel* method returns null and *addToTree* returns true, the display string is obtained with *toString* method.

7.5 Code Folding

Simpl supports folding in the source code editor. This can be controlled using *isFoldable* method.

```
def isFoldable(node: CommonNode): Boolean
```

The method takes an input an AST node and returns true if this node should be foldable in the editor. By default behaviour, the node is foldable if it is displayed in the outline view (see the *addToTree* method). The following example folds all nodes that span three or more lines.

```
override def isFoldable(node: CommonNode) =
    node.endLine - node.startLine > 1
```

7.6 Documentation Hovers

Simpl supports documentation hovers which are displayed when the user hovers mouse over some part of the code.

```
def getDocumentation(node: CommonNode): String
```

The *getDocumentation* method takes as input a node and returns string that should be displayed in a pop-up window when the user hovers text over source code representing this node. HTML tags are supported in the text.

7.7 Hyperlinking

Hyperlinking in Simpl works as usual in Eclipse – the user holds down Ctrl key when clicking on an identifier. This positions cursor at the definition of the identifier. This can be accomplished by implementing *referenceTarget* method.

```
def referenceTarget(node: CommonNode): CommonNode
```

This method takes as input a node (the link) and returns another node that functions as target to that link. If the input node is not link, just return *null*.

Parameter	Command-line option	Description
Package name	-package	Package that will contain classes that implement the new DSL.
Class name prefix	-class	Prefix that will be used in generating DSL implementation classes. For example, using prefix *Foo* will result in classes *FooGrammar*, *FooGenerator*, *FooConfig*.
DSL file extension	-ext	This extension will be associated with newly generated language in Eclipse.
Identifier of the DSL	-id	This is an unique identifier that will be used to identify the new language in Eclipse.
Textual DSL description	-description	This description will be used as name of the generated Eclipse plugin.
Base language to use	-base	The generated project contains a working sample language. Use this option to select, which sample you want to use. Select *bean* if you want fully functional language that implements code generator and more common IDE services. Select *empty* if you want empty project that does not include any services that depend on the sample language and must be changed (the empty project does not contain references to any AST classes and should successfully compile and run if you replace the sample grammar with your own).

Table 1: New project wizard parameters