Implementing the Syntax Definition Formalism as an Embedded Language

Pablo Hoch

TECHNISCHE
UNIVERSITÄT
DARMSTADT

1 Introduction

SDF DSL is an implementation of the Syntax Definition Formalism (SDF) as an Embedded Domain Specific Language (EDSL). It transforms a grammar specification written in SDF into a simple BNF grammar that is then used by *9am*, a Java implementation of an Earley parser. SDF grammars can be specified either by calling the methods of the *SdfDSL* class or by using the TigersEye Eclipse plugin, which allows embedding of standard SDF syntax. The implementation is based on the language described in *The Syntax Definition Formalism Reference Manual*[1]. This document gives a quick overview over the implementation of SDF as well as an example of how to specify a simple grammar using SDF and parse an input string with the generated *9am* grammar.

2 Implementation

The implementation of the SDF DSL consists of two main parts. The first part is the *SdfDSL* class, which provides the interface to users of the DSL to specify an SDF grammar and retrieve the generated *9am* grammar, which can then be used directly with the Earley parser. The SdfDSL class provides methods to specify every SDF construct such as modules, symbols and productions. By calling these methods, a tree representing the SDF grammar is created internally using classes from the *sdf.model* package. This model represents the complete input grammar, without any transformations applied and is basically an AST of the SDF grammar specification. The names of these methods and model classes correspond to the names used in the SDF documentation. Also, all classes in the *sdf.model* package include JavaDoc documentation with links to the respective parts of the SDF documentation.

After all SDF modules have been specified, a *9am* grammar is generated from the model representing the SDF grammar. This is done in two steps. First, since SDF grammars can consist of multiple modules, all modules have to be merged into one module by resolving all imports. This is done by the *ModuleMerger* class, which is implemented as a visitor for the model classes and takes the name of the top-level module as a parameter. Note that the *ModuleMerger* does not modify the original model, but creates a new model with all imports resolved. *ModuleMerger* performs two important tasks:

- When encountering an *import statement*, the specified module is processed recursively by using a *ModuleMerger*, resulting in a module without unresolved imports. The imported statements are then added to the current module in a new *exports* or *hiddens section*, depending on where the import statement occurs.
- Symbol renamings are also performed in the *ModuleMerger*. This includes parameters and renamings specified in *import statements* (which will only be applied to the imported module) as well as *aliases*. The renamings are performed when visiting a symbol in the *ModuleMerger*.

Once the model has been processed by the *ModuleMerger*, it consists of exactly one module which does not contain any *import statements* or unresolved *aliases*. This model is then passed to the *SdfToParlexGrammarConverter* class, which builds an equivalent BNF grammar for the *9am* Earley Parser. This is done by applying the transformations described in the SDF documentation. The following list provides an overview of the most important transformations.

- Basic *sort symbols*, e.g. `Expr` or `Number` (represented by *sdf.model.SortSymbol*), are turned into non-terminal categories.
- *Case-sensitive literal symbols*, such as `"true"` (represented by *sdf.model.LiteralSymbol*), are turned into terminal categories. *Case-insensitive literal symbols*, e.g. `'true'`, are processed using regular expressions in the *9am* parser.
- Compound symbols, such as *optional symbols* and *repetition symbols*, create additional rules as described in the SDF documentation. For example, the *alternative symbol* `"true"|"false"` (represented by *sdf.model.AlternativeSymbol*) results in two rules, α `::= "true"` and α `::= "false"`, where α is a special non-terminal representing the alternative symbol.
- *Character class symbols*, e.g. `[a-z]` (represented by *sdf.model.CharacterClassSymbol*), are implemented using regular expressions. Complex character classes that are combined using the set operators, e.g. `[a-z]`

[1] `http://www.syntax-definition.org/Sdf/SdfDocumentation` (version 2007-10-22 17:18:09 +0200)

`\/ [0-9]`, also result in a single regular expression, thanks to the extended regular expressions supported by Java[2].

- At the end, symbols are renamed to include the namespace they occur in (e.g. symbols occuring in a context-free grammar specification are renamed to include the "CF"-namespace, so a sort symbol `Expr` would become `<Expr-CF>`). Also, optional layout symbols are inserted on the left-hand side of productions in a context-free grammar.
- For productions with *attributes* (e.g. `left, right, prefer, avoid, reject`), the generated rules are annotated with *9am* rule annotations (e.g. associativity annotations). The *9am* Earley parser uses these annotations to select the desired AST.
- *Priorities* result in priority rule annotations, which are also processed by the Earley parser.

At the end, a *GrammarCleaner* is invoked by default. *GrammarCleaner* checks the generated *9am* grammar and removes all rules and categories that are not required, i.e. cannot be reached from the start rule. This step is optional and can be disabled.

The resulting *9am* grammar can then be directly passed to the *9am* Earley parser in order to parse input strings.

Other parts of the SDF DSL implementation include:

- Test cases in the seperate project *de.tud.stg.tigerseye.sdfdsl.tests*. These also demonstrate how to use the *SdfDSL* class directly.
- Example SDF DSL programs using the concrete syntax in the *de.tud.stg.tigerseye.sdfdsl.languagetestbench* project. These examples require the Tigerseye plugin.
- A *GrammarDebugPrinter* class (in the *sdf.util* package) that can be used to create an HTML file that lists all categories and productions of a *9am* grammar. Categories are linked to productions having that category on the left-hand side, which can be useful for debugging a grammar. An example output can be found in the *de.tud.stg.tigerseye.sdfdsl.languagetestbench* project (*debug/ArithExpr.html*, created by the *ArithExpr.sdf.dsl* file).

3 Using the SDF DSL

There are two ways to use the SDF DSL, either by calling the methods of the *SdfDSL* class directly to specify the grammar (abstract syntax), or by using standard SDF syntax (concrete syntax) in a Tigerseye DSL file. The grammar in listing 1 can be used to parse simple arithmetic expressions and uses some of the SDF specific features such as character classes, repetitions and layout (i.e. whitespace is allowed in the input string). To demonstrate how developers can implement the example grammar with both methods, listing 2 shows the version that directly calls the *SdfDSL* class methods to create the grammar. Also included is a simple example that shows how to parse an input string. The code shown in the listing is a shortened version of the file *sdf/test/ArithExprSdfTest.java* from the *de.tud.stg.tigerseye.sdfdsl.tests* project.

Listing 3 shows the version using the concrete SDF syntax. This file must be run from inside the Tigerseye plugin. This code is included in the *de.tud.stg.tigerseye.sdfdsl.languagetestbench* project in the *sdf/test/dsl/ArithExpr.sdf.dsl* file. The specification of the SDF grammar has been directly copied from the SDF file and some additional code to parse an example string has been added.

```
1 module ArithExpr
2 exports
3   context-free start-symbols Expr
4   sorts Expr Number Term Factor
5
6   lexical syntax
7     [0-9]+          -> Number
8     [ ]+            -> LAYOUT
```

[2] http://download.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html

```
9
10   context-free syntax
11     Expr "+" Term   -> Expr
12     Expr "-" Term   -> Expr
13     Term            -> Expr
14
15     Term "*" Factor -> Term
16     Term "/" Factor -> Term
17     Factor          -> Term
18
19     Number          -> Factor
20     "(" Expr ")"    -> Factor
```

Listing 1: ArithExpr.sdf

```
1  import sdf.SdfDSL;
2  import sdf.model.*;
3  import de.tud.stg.parlex.core.Grammar;
4  import de.tud.stg.parlex.parser.earley.Chart;
5  import de.tud.stg.parlex.parser.earley.EarleyParser;
6
7  public class ArithExprSdfTest {
8    SdfDSL sdf;
9    Grammar grammar;
10
11   public void setUp() {
12     sdf = new SdfDSL();
13
14     Sorts sorts = sdf.sortsDeclaration(new SortSymbol[] {
15         sdf.sortSymbol("Expr"), sdf.sortSymbol("Number"),
16         sdf.sortSymbol("Term"), sdf.sortSymbol("Factor") });
17
18     Syntax lexSyntax = sdf.lexicalSyntax(new Production[] {
19         sdf.production(new Symbol[] {
20             sdf.repetitionSymbolAtLeastOnce(
21                 sdf.characterClassSymbol("0-9")) },
22             sdf.sortSymbol("Number")),
23         sdf.production(new Symbol[] {
24             sdf.repetitionSymbolAtLeastOnce(
25                 sdf.characterClassSymbol(" ")) },
26             sdf.sortSymbol("LAYOUT")), });
27
28     Syntax cfSyntax = sdf.contextFreeSyntax(new Production[] {
29         sdf.production(
30             new Symbol[] { sdf.sortSymbol("Expr"),
31                 sdf.caseSensitiveLiteralSymbol("+"),
32                 sdf.sortSymbol("Term") },
33             sdf.sortSymbol("Expr")),
34         sdf.production(
35             new Symbol[] { sdf.sortSymbol("Expr"),
36                 sdf.caseSensitiveLiteralSymbol("-"),
37                 sdf.sortSymbol("Term") },
38             sdf.sortSymbol("Expr")),
39         sdf.production(new Symbol[] { sdf.sortSymbol("Term") },
40             sdf.sortSymbol("Expr")),
41
```

```
42          sdf.production(
43              new Symbol[] { sdf.sortSymbol("Term"),
44                  sdf.caseSensitiveLiteralSymbol("*"),
45                  sdf.sortSymbol("Factor") },
46              sdf.sortSymbol("Term")),
47          sdf.production(
48              new Symbol[] { sdf.sortSymbol("Term"),
49                  sdf.caseSensitiveLiteralSymbol("/"),
50                  sdf.sortSymbol("Factor") },
51              sdf.sortSymbol("Term")),
52          sdf.production(new Symbol[] { sdf.sortSymbol("Factor") },
53              sdf.sortSymbol("Term")),
54
55          sdf.production(new Symbol[] { sdf.sortSymbol("Number") },
56              sdf.sortSymbol("Factor")),
57          sdf.production(
58              new Symbol[] { sdf.caseSensitiveLiteralSymbol("("),
59                  sdf.sortSymbol("Expr"),
60                  sdf.caseSensitiveLiteralSymbol(")"), },
61              sdf.sortSymbol("Factor")), });
62
63      StartSymbols startSymbols = sdf
64          .contextFreeStartSymbols(new Symbol[] { sdf.sortSymbol("Expr") });
65
66      Exports exports = sdf.exports(new GrammarElement[] { startSymbols,
67          sorts, lexSyntax, cfSyntax });
68
69      Module module = sdf.moduleWithoutParameters(new ModuleId("ArithExpr"),
70          new Imports[] {}, new ExportOrHiddenSection[] { exports });
71
72      grammar = sdf.getGrammar("ArithExpr");
73    }
74
75    public void testEarleyParserWithExpr1() {
76      EarleyParser parser = new EarleyParser(grammar);
77      Chart chart = (Chart) parser.parse("2+3*5");
78      chart.rparse((de.tud.stg.parlex.core.Rule) grammar.getStartRule());
79
80      System.out.println("AST:");
81      System.out.println(chart.getAST().toString());
82    }
83  }
```

Listing 2: ArithExpr.java

```
1  import sdf.model.*
2
3  sdf(name:'ArithExpr'){
4
5    module ArithExpr
6    exports
7      context-free start-symbols Expr
8      sorts Expr Number Term Factor
9
10     lexical syntax
11       [0-9]+          -> Number
```

```
12        [ ]+              -> LAYOUT
13
14      context-free syntax
15        Expr "+" Term     -> Expr
16        Expr "-" Term     -> Expr
17        Term              -> Expr
18
19        Term "*" Factor -> Term
20        Term "/" Factor -> Term
21        Factor            -> Term
22
23        Number            -> Factor
24        "(" Expr ")"      -> Factor
25
26
27    printGeneratedGrammarHTML "ArithExpr" "debug/ArithExpr.html"
28    parse "ArithExpr" "4 + 8 * (15 / (16+23)) - 42"
29
30  }
```

Listing 3: ArithExpr.sdf.dsl

4 Limitations of the SDF DSL

The following features of SDF are currently not implemented in the SDF DSL:

- For priorities, *priorities in specific arguments*[3] are not supported. Also, relative associativity labels inside groups are not supported.
- *Variables*[4] are not supported since their purpose is not clear from the SDF documentation.
- *Follow restrictions*[5] are not supported, because they are not needed for most languages. Support for follow restrictions could be added using a custom disambiguation oracle for the *9am* Earley parser.

[3] http://homepages.cwi.nl/~daybuild/daily-books/syntax/2-sdf/sdf.html#section.priorities
[4] http://homepages.cwi.nl/~daybuild/daily-books/syntax/2-sdf/sdf.html#section.variables
[5] http://homepages.cwi.nl/~daybuild/daily-books/syntax/2-sdf/sdf.html#section.restrictions