A Quick Introduction to SDF

Joost Visser (CWI)
Joost.Visser@cwi.nl

Jeroen Scheerder (CWI)
js@cwi.nl

April 25, 2000

1 Questions and Answers

What is SDF? SDF stands for Syntax Definition Formalism, and it's just that: a formalism to define syntax.

How is SDF different from BNF? SDF is richer than BNF. It is modular. It can be used to define lexical and context-free grammars alike. It is supported by a parser generator.

How is SDF different from Yacc? Yacc is not purely declarative and it restricts the class of grammars that can be defined. It is not purely declarative in the sense that all grammar rules are decorated with semantic actions consisting of procedural code. It is restrictive in the sense that it can generate a parser only if the input grammar falls within the LALR subclass.

Why use SDF? SDF's expressiveness allows defining syntax concisely and naturally. SDF's modularity facilitates reuse. SDF's declarativeness makes it easy and retargetable. But the most important reason to use SDF is that it is supported by *Scannerless Generalized LR Parsing*.

What is 'scannerless' parsing? In scannerless parsing, no separate lexical scanning tool is needed to tokenize the input stream before parsing. Lexical analysis (tokenization) and context-free analysis (parsing) are uniformly handled by a single tool.

What is 'generalized' parsing? In generalized parsing, ambiguities are allowed. This implies that the parsing process may be indeterminate, i.e. can have several equally correct results. Generalized parsing removes the restriction to a non-ambiguous subclass of the context-free grammars, such as the LR(k) class.

Why is generalized parsing better? Firstly, generalized parsing imposes less restrictions on the grammar writer, hence allows a maximally natural expression of the intended syntax. There is no more need for 'bending over backwards' to encode the intended grammar in a restricted subclass.

Secondly, generalized parsing leads to better modularity. When two LR grammars are merged, there is no guarantee that the resulting grammar will be LR again, and will not need to be manually modified to fit this mould. In fact, none of the non-ambiguous grammar subclasses has this modularity property. Only the full class of grammars can be composed modularly. And generalized LR parsing supports this class.

2 Step by Step

We will guide you step by step through the process of defining a grammar in SDF.

Decide on a name We will assume the name of the language you want to define is `L`. The name of the SDF file in which you are going to define its syntax will be `L.def`. Create this file:

```
> touch L.def
```

Set up your tests Before you do anything, you should decide what exactly you want to do. The best way to fix your goal is to identify some representative terms of the language you want to describe. Put these in a subdirectory named `data`. We assume an initial test set with a single term: `0+-1+2`.

Now, create a `Makefile` with the following content:

```
Terms = $(wildcard $(DATA)/*)

test: $(GRAMMAR).tbl
    for Term in $(Terms) ;\
    do \
        sglr -v -p $(GRAMMAR).tbl \
          -i $$Term -o /dev/null ; \
    done

%.tbl: %
    sdf2table -i $< -o $@
```

And you are ready to run your first test:

```
> gmake GRAMMAR=L.def DATA=data test
```

Needless to say, it fails.

Define your syntax Put the following in `L.def`:

```
definition

module Main
exports
  syntax
    [0-9]+      -> N
    N "+" N     -> N
    "-" N       -> N
    N [\n]      -> <START>
```

And rerun your test:

```
> gmake GRAMMAR=L.def DATA=data test
```

Your test succeeds! But, as the message indicates, the parsed term is ambiguous, which means that the result is a parse forest instead of a single parse tree.

Disambiguate SDF offers various mechanisms for disambiguation, and our little example was of course constructed to give us the opportunity to explain these mechanisms to you.

Associativity To indicate whether you want operators to associate to the left or to the right, the `left` and `right` attributes are available. Add them to the production for `+` as follows:

```
N "+" N -> N {left}
```

Rerun your test and discover that the number of ambiguities has decreased.

Priorities Using associativity attributes, ambiguities between various applications of the same production are resolved. To resolve ambiguities between different productions you can define relative priorities between them. To give `-` priority over `+`, add:

```
priorities
  "-" N -> N   >   N "+" N -> N
```

The number of ambiguities now drops to zero.

Add layout Add the following term to your test suite: `1 + 1`. Run the test to discover that the whitespace in the term leads to a parse error. To get this term to parse, you can add the following:

```
syntax
  [\ \t\n]         -> LAYOUT
  LAYOUT LAYOUT    -> LAYOUT {left}
restrictions
  LAYOUT? -/- [\ \t\n]
```

And change the productions for `+` and `-` into:

```
N LAYOUT? "+" LAYOUT? N -> N {left}
"-" LAYOUT? N -> N
```

`LAYOUT?` means optional `LAYOUT`. Now the new term parses correctly.

Using the keywords `sorts`, `context-free` and `lexical`, SDF allows these things to be written down more concisely as follows:

```
definition

module Main
exports
  sorts N
  lexical syntax
    [0-9]+      -> N
  context-free syntax
    N "+" N     -> N    {left}
    "-" N       -> N
  context-free priorities
    "-" N -> N   >   N "+" N -> N
  lexical syntax
    [\ \t\n]    -> LAYOUT
  context-free restrictions
    LAYOUT? -/- [\ \t\n]
```

Thus, `LAYOUT` is treated as a special sort with some implicit syntax rules. In `context-free syntax` sections, `LAYOUT?` is implicitly present between all explicit symbols in the left hand side of productions. The declaration of sort `N` makes the production for `<START>` redundant, because all declared sorts are implicitly reachable from the start symbol.

Add precision SDF offers rejects and restrictions to increase the precision of parsing.

Rejects Add the terms `01` and `-0` to the test suite, and suppose you want to disallow leading zeros and negated zeros. You can use productions with a `reject` attribute for this purpose:

```
[0][0-9]+    -> N {reject}
"-" [0]      -> N {reject}
```

Sure enough, when you now run your test the undesired terms fail to parse.

Restrictions Follow restrictions can be used to express the rule "Prefer longest match". The restriction on `LAYOUT?` above is an example of this. Another example is:

```
context-free restrictions
  N -/- [0-9]
```

This restriction expresses that a term of the sort `N`, appearing in a context-free position, is not allowed to be followed by a digit. Thus, only parses of `N` that consume all digits are allowed.

Refactor Your grammar has gradually grown warts. Time to refactor, using the modularity features of SDF:

```
definition

module Main
imports
   Disambiguate AddPrecision Layout

module N
exports
  sorts N
  lexical syntax
    [0-9]+      -> N
  context-free syntax
    N "+" N     -> N
    "-" N       -> N

module Disambiguate
imports N
exports
  context-free priorities
    "-" N -> N   >   N "+" N -> N {left}
```

```
module AddPrecision
imports N
exports
  context-free syntax
    [0][0-9]+   -> N    {reject}
    "-" [0]     -> N    {reject}

module Layout
exports
  lexical syntax
    [\ \t\n]    -> LAYOUT
  context-free restrictions
    LAYOUT? -/- [\ \t\n]
```

Note that productions for the same sort are allowed to be spread over different modules. Also, modules can be mutually recursive. We moved the `left` attribute of the production for `+` from the context-free syntax to the priorities. This is equivalent.

Take your pick The `priority` construct described above expresses an ordering between explicitly stated productions. Using `prefer` and `avoid` attributes you can express a *general* preference or dislike for a production. We will show you how in an example that harvests mail addresses from arbitrary character streams.

The following fragment defines messages as arbitrary character streams:

```
module Message
exports
  syntax
    Msg         -> <START>
    X*          -> Msg
    ~[]         -> X {avoid}
```

The `avoid` attribute indicates that the final production is a last resort, to be used only when all other rules fail. We now add syntax for mail addresses:

```
module MailAddress
exports
  syntax
    Segment "@" HostName      -> X
    [A-Za-z][A-Za-z0-9\-\_]* -> Segment
    {Segment "." }+           -> HostName
```

A parse of a part of the character stream as a mail address will take precedence over the parse as an arbitrary sequence, because of the latter's `avoid`.

The same behaviour is obtained by marking the second rule for `X` with `prefer` instead of marking the first with `avoid`.

Get funky By now, the general idea of SDF is clear to you. To give you an impression of what else is possible with SDF, we show two modules that define lists of naturals:

```
module List[X]
exports
  context-free syntax
    "[" Xs "]"    -> L
  aliases
    {X ","}*      -> Xs

module NList
imports
  List[N][L => NList]
exports
  sorts NList
```

The first module is parameterized in the sort `X`. It uses the `aliases` construct to define an auxiliary sort `Xs`. This module is imported by module `NList`, where its parameter is instantiated to `N`. Also, the renaming `[L => NList]` is applied to it, to change the imported sort name `L` into `NList`.

3 More

To learn more about SDF we suggest the following:

- Consult the `man` pages of `sglr` and `sdf2table`. You will find an exhaustive account of the usage of these tools.
- Have a look at the grammars in the Grammar Base [1] for examples. A reliable source of SDF lore.
- Have a look at the SDF definition of SDF itself which can be found in the Grammar Base as well. It gives a complete overview of all constucts available in SDF.
- Read about SDF in [2].
- Mail questions and suggestions to `meta-users@cwi.nl`.

Acknowledgments Thansk to Merijn de Jonge for useful remarks on drafts of this paper.

References

[1] M. de Jonge, E. Visser, and J. Visser. GB: Grammar Base. Available from `http://www.cs.uu.nl/~visser/gb/`.

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