ASF+SDF Meta-Environment User Manual

Revision : 1.149

M.G.J. van den Brand and P. Klint

Centrum voor Wiskunde en Informatica (CWI),

Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

17th January 2005

Abstract

This is a preliminary user manual for the ASF+SDF Meta-Environment Release 1.5. This is work under construction.

Contents

Update with respect to previous version

- Adapted to version 1.5 of the ASF+SDF Meta-Environment.
- Added a description of the unit tests in ASF.
- Added a detailed description of the error messages (see Section 4).
- Described the `context-free start-symbols` (see Section 2.8).
- Rewritten the section on stand-alone tools.

1 Overview

1.1 When to use the ASF+SDF Meta-Environment?

The ASF+SDF Meta-Environment is an interactive development environment for the automatic generation of interactive systems for manipulating programs, specifications, or other texts written in a formal language. The generation process is controlled by a definition of the target language, which typically includes such features as syntax, pretty printing, type checking and execution of programs in the target language. The ASF+SDF Meta-Environment can help you if:

- You have to write a formal specification for some problem and you need interactive support to do this.
- You have developed your own (application) language and want to create an interactive environment for it.
- You have programs in some existing programming language and you want to analyze or transform them.

The ASF+SDF formalism allows the definition of syntactic as well as semantic aspects of a (programming) language. It can be used for the definition of languages (for programming, for writing specifications, for querying databases, for text processing, or for dedicated applications). In addition it can be used for the formal specification of a wide variety of problems. ASF+SDF provides you with:

- A general-purpose algebraic specification formalism based on equational logic.
- Modular structuring of specifications.
- Integrated definition of lexical, context-free, and abstract syntax.
- User-defined syntax, allowing you to write specifications using your own notation.
- Complete integration of the definition of syntax and semantics.

The ASF+SDF Meta-Environment offers:

- Syntax-directed editing of ASF+SDF specifications.
- Testing of specifications by means of interpretation.
- Compilation of ASF+SDF specifications into dedicated interactive environments containing various tools such as a parser, a pretty printer, a syntax-directed editor, a debugger, and an interpreter or compiler.

The advantages of creating interactive environments in this way are twofold:

- *Increased uniformity*. Similar tools for different languages often suffer from a lack of uniformity. Generating tools from language definitions will result in a large increase in uniformity, with corresponding benefits for the user.
- *Reduced implementation effort.* Preparing a language definition requires significantly less effort than developing an environment from scratch.

1.2 Global Structure of the Meta-Environment

You can create new specifications or modify and test existing ones using the Meta-Environment. Specifications consist of a series of modules, and individual modules can be edited by invoking editors for the syntax part and the equations part of a module. All editing in the Meta-Environment is done by creating instances of a *generic syntax-directed editor*.

After each editing operation on a module, its *implementation* is updated immediately. It consists of a parser, a pretty printer, and a term rewriting system which are all derived from the module automatically.

A module can be tested by invoking a *term editor* to create and evaluate terms defined by the module. Term editors use the syntax of the module for parsing the textual representation of terms and for converting them to internal format (syntax trees). The equations of the module are then used to reduce the terms into normal form. This result is, in its turn, converted back to textual form by pretty printing it.

1.3 About this Manual

This manual is intended for those users that want to write ASF+SDF specifications. This manual is still under development and we welcome all feed back and comments.

The focus of this manual will be on how to write ASF+SDF specifications, the ASF+SDF language features are explained as well as some methodology in using ASF+SDF when defining a (programming) language. In the last part of the manual we will address the following technical problems.

- How to compile a specification.
- How to parse a term outside the ASF+SDF Meta-Environment.
- How to rewrite a term using a compiled specification outside the ASF+SDF Meta-Environment.
- How to unparse parsed and/or normalized terms.

Finally, we assume the reader has read the ASF+SDF guided tour and knows how to use the ASF+SDF Meta-Environment.
We do *not* explain in detail:

- The architecture and implementation of the system. We only give a brief sketch of the underlying technology and architecture of the ASF+SDF Meta-Environment.
- The stand-alone usage of various parts of the system. We only describe the usage of the most important components.

1.4 Downloading the ASF+SDF Meta-Environment

You can download the ASF+SDF Meta-Environment from the following location:

```
http://www.cwi.nl/projects/MetaEnv/
```

It provides links to the software as well as to related documents. Furthermore, via this link bugs can be submitted.

1.5 Further Reading

There are many publications about the ASF+SDF Meta-Environment itself, about the implementation techniques used, and about applications. Also see the overview of architecture and implementation techniques (Section 5.1). We give here a brief overview of selected publications:

Overviews: [24], [30], [3], [4].

General ideas: [25], [26], [22].

ASF: [1].

SDF: [23], [35].

ASF+SDF: [1], [21].

Parser generation and parsing: [32], [27], [34], [33], [35], [17], [14].

Pretty printing: [19], [28].

Rewriting and Compilation: [10], [6], [18], [11], [7].

ToolBus: [2].

ATerms: [8].

Applications: [5],[16], [15].

Generic debugging: [31].

Traversal functions: [12], [13].

User manuals: [20], [29], [9].

Acknowledgements

Peter D. Mosses, Albert Hofkamp, Akim Demaille, Jurgen Vinju, Bas Terwijn.

2 An Introduction to ASF+SDF

ASF+SDF is the result of the marriage of two formalisms ASF (Algebraic Specification Formalism) and SDF (Syntax Definition Formalism). ASF is based on the notion of a module consisting of a signature defining the abstract syntax of functions and a set of conditional equations defining their semantics. Modules can be imported in other modules. SDF allows the simultaneous definition of concrete (i.e., lexical and context-free) and abstract syntax and implicitly defines a translation from text strings to abstract syntax trees.

The main idea of ASF+SDF is to identify the abstract syntax defined by the signature in ASF specifications with the abstract syntax defined implicitly by an SDF specification, thus yielding a standard mapping from text to abstract syntax tree. This allows the association of semantics with (the tree representation of) text and introduces user-defined notation in specifications.

ASF+SDF is therefore a modular specification formalism for the integrated definition of syntax and semantics of a (programming) language. Other views on ASF+SDF are:

- a first-order functional programming language.
- an algebraic specification formalism.

Whatever viewpoint is taken, ASF+SDF is a powerful formalism for the declarative description of programming languages.

2.1 (E)BNF and Lex+Yacc versus SDF

(E)BNF-like and Lex+Yacc-like grammar formalisms are well-known. Although Lex+Yacc is more a domain specific language than a grammar formalism, the grammar of a lot of programming languages are presented as Lex+Yacc definition. There a number of differences between both (E)BNF-like and Lex+Yacc-like formalisms and SDF.

We assume that the reader of this manual has some experience with formalisms like (E)BNF and Lex+Yacc.

SDF allows a modular definition of your syntax formalism. This allows re-use of parts of other grammar definitions. This is only possible given the fact that the underlying parsing technology is based on Generalized LR parsing, see [33] and [35] for more details.

SDF imposes no restrictions on the grammar. In contrast to Lex+Yacc, restricted to the class of LALR(1)-grammars, we do not impose these restrictions. The fact that we do not impose these restrictions enables us to have this modular grammar definition formalism. Restricted classes, such as LALR(1), are not closed under union. Two grammars, both LALR(1) for instance, need not result in a LALR(1) grammar if they are combined.

The most striking difference between SDF and (E)BNF-like and Lex+Yacc-like formalisms is the way the production rules are written in SDF. In (E)BNF and Lex+Yacc one writes production rules as

```
P ::= 'b' D S 'e'
```

whereas in SDF this is written as

```
"b" D S "e" -> P
```

So, the left- and right-hand side of the production rules are swapped.

SDF provides an integrated definition of lexical and context-free syntax. (E)BNF does not provide, or only very restrictive, support for defining lexical syntax rules. In Lex+Yacc the lexical syntax is more or less defined in a separate formalism.

SDF also allows an integrated way of defining associativity and priorities between production rules, see Section 2.13.

Finally, SDF provides an automatic way of constructing syntax trees. In Lex+Yacc the specification writer has to program how syntax trees are constructed.

2.2 Modules and Modular Structure

An ASF+SDF specification consists of a sequence of module declarations. Each module may define syntax rules as well as semantic rules and the notation used in the semantic rules depends on the definition of syntax rules. The entities declared in a module may be visible or invisible to other modules. A module can use another module from the specification by importing it. As a result, all visible names of the imported module become available in the importing module.

The overall structure of a module is:

```
module <ModuleName>

  <ImportSection>*

  <ExportOrHiddenSection>*

equations
  <ConditionalEquation>*
```

A module consists of a module header, followed by a list of zero or more import sections, followed by zero or more hidden or export sections and an optional equations section that defines conditional equations. In Section 2.15 we will see that modules can alse be *parameterized* and that they can be *renamed on import*.

Conceptually, a module is a single unit but for technical reasons the syntax sections and the equations section are stored in physically separate files. For each module M in a specification two files exist: 'M`.sdf`' contains the syntax sections of M and 'M`.asf`' contains the equations section of M.

A `<ModuleName>` is either a simple `<ModuleId>` or a `<ModuleId>` followed by zero or more parameter symbols, e.g., `<Module>[<Symbol>*]`, the symbols will be explained in Section 2.5. The `<ModuleId>` may be compount module name, the `ModuleId` reflects the directory structure. For example `basic/Booleans` means that the module `Booleans` is found in the subdirectory `basic`.

An `<ExportOrHiddenSection>` is either an *export section* or a *hidden section*. The former starts with the keyword `exports` and makes all entities in the section visible to other modules. The latter starts with the keyword `hiddens` and makes all entities in the section local to the module.

An `<ExportOrHiddenSection>` has thus one of the two forms:

```
exports
  <Grammar>+
```

or

```
hiddens
  <Grammar>+
```

A `<Grammar>` can be a definition of one of the following:

- Imports (Section 2.4).
- Aliases (Section 2.6).
- Sorts (Section 2.7).
- Start-symbols (Section 2.8).
- Lexical syntax (Section 2.9).
- Context-free syntax (Section 2.10).
- Priorities (Section 2.13).
- Variables (Section 2.16).

```
module basic/Comments

imports basic/Whitespace

%% In this module we define the
%% comment convention for Sdf.

exports
  lexical syntax
    "%%" ~[\n]* "\n" -> LAYOUT
    "%" ~[\n\%]+ "%" -> LAYOUT

  context-free restrictions
    LAYOUT? -/- [\%]
```

Figure 1: SDF comment

Note that it is possible to have hidden imports as well, this means that the contents of a hidden imported module in some module M is visible in M but is not exported to modules which import M.

There are a number of related properties which have an effect across the various grammar items, these items are intermixed with the discussion of the grammars:

- SDF comment convention (Section 2.3).
- Symbols (Section 2.5).
- Attributes of Lexical and Context-free Functions (Section 2.12).
- Disambiguation (Section 2.14).
- Parameterization and Renamings (Section 2.15).
- Libraries (Section 2.17).
- Equations (Section 2.18).

Each of these entities and properties will now be described and illustrated by examples.

2.3 SDF comment convention

The comment convention within an SDF specification is that character between `"%%"` and the end of line is comment as well as every character between two `"%"` including the newline character. An example of the use of comments is given in Figure 1.

This definition also defines the comment convention in SDF itself. More details on defining layout can be found in Section 2.13.5.

2.4 Imports

Each `<ImportSection>` starts with the keyword `imports` followed by zero or more module names:

```
imports
  <ModuleName>*
```

Modules can be combined by importing one module in another. Imports can occur as `<ImportSection>` at the topmost level of a module or they can occur within an exports or hiddens section.

When importing modules at the topmost level of a module or when the import section occurs within the scope of an exports keyword, all exported entities of the imported module (and of all modules that are imported indirectly by it) become available in the importing module. In addition, they are also exported by the importing module. However, if the import section occurs within the scope of a hiddens keyword, the exported entities are only visible in the importing module but they are not exported by the importing module.

An imported module can be parameterized or decorated with renamings, see Section 2.15 for more details.

The name of the imported module can also be a compound module name. In Figure 1 the imported module `basic/Whitespace` is an example of such a compound module name.

2.5 Symbols

The elementary building block of SDF syntax rules is the "symbol". It is comparable to terminals and non-terminals in other grammar definition formalisms. The elementary symbols are:

- *sort*: corresponds to a non-terminal, e.g., `Bool`. Sort names always start with a capital letter and may be followed by letters and/or digits. Hyphens ("`-`") may be embedded in a sort name.
- *literal*: corresponds to a terminal, e.g., `"true"` or `"&"`. Terminals must always be quoted, also the terminals consisting of only letters.
- *character class*: corresponds to a set of characters, e.g., `[a-z]`. Character classes will be explained in Section 2.9.2, they are mainly used when describing the lexical syntax of a language.

Starting with the elementary symbols, more complex symbols can be constructed by way of the following operators.

Examples of the use of the various operators will be given in Sections 2.9 and 2.10.

2.5.1 Option

The postfix option operator `?` describes an optional part in a syntax rule. For instance, `ElsePart?` defines zero or exactly one occurrence of `ElsePart`.

2.5.2 Sequence

The sequence operator `(...)` describes the grouping of two or more symbols, e.g., `(Bool "&")`. Sequences are mostly used to group symbols together to form a more complex symbol using one of the available operators, e.g., `(Bool "&")*`. It has no effect to construct a sequence consisting of a single symbol. The empty sequence is represented as `()`.

2.5.3 Repetition

Repetition operators express that a symbol should occur several times. In this way it is possible to construct flat lists and therefore we usually refer to repetitions as *lists*.

Repetition operators come in two flavors, with and without separators. Furthermore, it is possible to express the minimal number of repetitions of the symbol: at least zero times (`*`) or at least one time (`+`). Examples are:

- `Bool*` (a list of zero or more `Bool`s).
- {`Bool ","`}`+` (a list of one or more `Bool`s separated by comma's).

In case of a separated list the element can be an arbitrary symbol, but the separator can only be a plain literal. It is possible to write, for instance {`Int Bool`}* or {`Int (",","|";")`}*, but the `asfsdf-checker`, see Section 4, will produce a warning indicating that this type of symbol is not supported. Both the interpreter, see Section 5.4, and the compiler, see Section 5.6, do not support this type of separated lists.

2.5.4 Alternative

The alternative operator `|` expresses the choice between two symbols, e.g., `true | false` represents that either a `true` symbol or a `false` symbol may occur here. The alternative operator is right associative and binds stronger than any other operator on symbols. This is important because `Bool "," | Bool ";"` expresses `Bool ("," | Bool) ";"` instead of `(Bool ",") | (Bool ";")`. So, in case of doubt use the sequence operator in combination with the alternative operator.

2.5.5 Tuple

The tuple operator describes the grouping of a sequence of symbols of a fixed length into a tuple. The notation for tuples is `< , , >`, i.e., a comma-separated list of elements enclosed in angle brackets. For example, `<Bool, Int, Id>` describes a tuple with three elements consisting of a `Bool`, an `Int` and an `Id` (in that order). For instance, `<true, 3, x>` is a valid example of such a tuple.

2.5.6 Function

The function operator `(...=>...)` allows the definition of function types. Left of `=>` zero or more symbols may occur, right of `=>` exactly one symbol may occur. For example, `(Bool Int) => Int` represents a function with two argument (of types `Bool` and `Int`, respectively) and a result type `Int`.

2.5.7 Parameterized Sorts

Sort names can have parameters. This provides a way of distinguishing a generic sort `List` for integers, e.g. `List[[Int]]`, from booleans, e.g. `List[[Bool]]`. These sort parameters can be instantiated via the parameters of the module name. A parameterized sort may have the form `List[[X,Y]]` where `X` and `Y` are generic sorts which will be provided via the parameters of the module name. See Section 2.15 for more details. The context-free syntax rule describing parameterized sorts is:

```
Sort "[[" {Symbol ","}+ "]]" -> Symbol
```

2.6 Aliases

In ordinary programming it is good practice to use named constants to represent literals or constant values. In SDF it is good practice to give a name (“alias”) to complicated symbols that occur repeatedly in the specification. An alias is thus a named abbreviation for a complicated symbol. For example,

```
aliases
  <Bool, Int, Id> -> Tuple3
```

introduces the alias `Tuple3` for the symbol `<Bool, Int, Id>` and instead of using `<Bool, Int, Id>` one can use the alias `Tuple3`. During parse table generation the alias is replaced by the actual symbol. It is not allowed to give an alias for an alias or to redefine aliases. For example, the following definitions are illegal:

```
aliases
  Tuple3          -> SuperTuple
  <Bool, Int, Id> -> Tuple3
```

(An alias is defined for the alias `Tuple3`.)

```
aliases
  <Bool, Int, Id> -> Tuple3
  <Bool, Int>     -> Tuple3
```

(The alias `Tuple3` is redefined.)

Note, the aliases are a convenient short hand for more complex symbols, but a drawback is that during parse table generation the aliases completely disappear. They are replaced by the actual symbols. This can have some unexpected behaviour when parsing or reducing terms.

2.7 Sorts

Sorts are declared by listing their name in a sorts section of the form:

```
sorts
  <Symbol>*
```

Only plain `Sort`s and parameterized `Sort`s can be declared in the `sorts` section, but more complex `Symbol`s will be syntactically recognized, but the `asfsdf-checker`, see Section 4, will generate a warning. It is required that all sorts that occur in some symbol in the specification are declared.
Recall that a sort name should start with a capital letter and may be followed by letters and/or digits. Hyphens (`-`) may be embedded in sort names. There is one predefined sort name (`LAYOUT`). It is described in Lexical Syntax Section 2.9.

It is not allowed (or necessary) to define the sorts `LAYOUT` and `CHAR`. These two sorts are always available.

2.8 Context-free start-symbols

Via the context-free start symbols section the symbols are explicitly defined which will serve as start symbols when parsing terms. If no start symbols are defined it is not possible to recognize terms. This has the effect that input sentences corresponding to these symbols can be parsed. So, if we want to recognize booleans terms we have to define explicitly the sort `Boolean` as a start symbol in the module `Booleans`. Any symbol, also lists, tuples, etc., can serve as a start-symbol.

```
context-free start-symbols
  <Symbol>*
```

Context-free start-symbol sections can either be hidden or exported. The effect of defined symbols as start symbols for the grammar may lead to an explosion of start states for the parser and thus lead to performance loss. To prevent this it can be advisable to hide start-symbol sections. The symbols defined are only visible in the module containing this hidden start-symbols section, but not in modules importing this module.

2.9 Lexical Syntax

The lexical syntax describes the low level structure of text by means of *lexical tokens*. A lexical token consists of a sort name (used to distinguish classes of tokens like identifiers and numbers), and the actual text of the token. The lexical syntax also defines which substrings of the text are layout symbols or comments and are to be skipped.

A lexical syntax contains a set of declarations for *lexical functions*, each consisting of a regular expression and a result sort. All functions with the same result sort together define the lexical syntax of tokens of that sort. Regular expressions may contain any basic symbol and any symbol operator as described in Section 2.5. Spaces are only significant inside strings and character classes.

The sort name `LAYOUT` is predefined and may not be redeclared. `LAYOUT` defines which parts of the text are *layout symbols* (also known as *white space*) between lexical tokens and are to be skipped during lexical analysis. It may only be used as result sort of lexical functions (Section 2.9.1). When a string

```
module LeesPlank

imports basic/Whitespace

exports
  context-free start-symbols LeesPlank
  sorts Aap Noot Mies LeesPlank
  lexical syntax
    "aap"         -> Aap
    "noot"        -> Noot
    "mies"        -> Mies
    Aap Noot Mies -> LeesPlank
```

Figure 2: Simple lexical functions

is matched by both a LAYOUT function and by other non-LAYOUT functions, then the interpretation as layout symbol is ignored. `LAYOUT` is typically used for defining layout and comment conventions.

Traditionally, lexical syntax and context-free syntax are treated differently. They are defined by different notations and implemented by means of different techniques. SDF provides a much more uniform treatment. In SDF, the only significant difference between the two is that no layout will be accepted while recognizing the members of the left-hand side of a lexical function, whereas layout *will* be accepted between the members of the left-hand side of a context-free function. At the implementation level, both are implemented using a single parsing technique.

Technically, there exist only *syntax* sections. Both lexical syntax sections and context-free syntax sections are transformed into such syntax sections after appropriate insertion of optional layout between the elements of context-free functions. In rare cases, the specification writer may want to control this process explicitly and write syntax sections directly. This will not be discussed in this manual, but further details can be found in [35].

2.9.1 Lexical Functions

In their simplest form, declarations of lexical functions consist of a sequence of zero or more symbols followed by '$\rightarrow$' and a result symbol, say L. A lexical function may be followed by a list of attributes. The regular expression associated with L consists of the logical *or* of all left-hand sides of lexical functions with result sort L. All sort names appearing in left-hand sides of declarations are replaced by the regular expression associated with them. Figure 2 shows an example of a simple lexical function definition for defining the first three words that Dutch children learn to read. The three sorts `Aap`, `Noot` and `Mies`, each recognize, respectively, the strings `aap`, `noot` and `mies`. The sort `LeesPlank` (a reading-desk used in primary education) recognizes the single string `aapnootmies`.

Lexical constructor functions For each sort `L` that appears as result sort in the lexical syntax a lexical constructor function of the form `l "(" CHAR+ ")" -> L` is automatically added to the context-free syntax of the specification. Here, '`l`' is the name of sort '`L`' written in lower-case letters. In this way, you can get access to the characters of lexical tokens.

The lexical constructor functions will be discussed in more detail in the section on see Section 2.18.

2.9.2 Character Classes

Enumerations of characters occur frequently in lexical definitions. They can be abbreviated by using character classes enclosed by '`[`' and '`]`'. A character class contains a list of zero or more characters (which stand for themselves) or character ranges such as, for instance, `[0-9]` as an abbreviation for the characters `0`, `1`, ..., `9`. In a character range of the form c_1`-`c_2 one of the following restrictions should apply:

```
module LettersDigits1

imports basic/Whitespace

exports
  context-free start-symbols Letter Digit
  sorts LCLetter UCLetter Letter Digit
  lexical syntax
    [a-z]    -> LCLetter
    [A-Z]    -> UCLetter
    [a-zA-Z] -> Letter
    [0-9]    -> Digit
```

Figure 3: Defining letter (lower-case and upper-case) and digit

```
module LettersDigits2
imports basic/Whitespace

exports
  context-free start-symbols LetterOrDigit
  sorts LetterOrDigit
  lexical syntax
    [a-z]    -> LetterOrDigit
    [A-Z]    -> LetterOrDigit
    [0-9]    -> LetterOrDigit
```

Figure 4: Defining a single letter or digit

- c_1 and c_2 are both lower-case letters and c_2 follows c_1 in the alphabet, or
- c_1 and c_2 are both upper-case letters and c_2 follows c_1 in the alphabet, or
- c_1 and c_2 are both digits and the numeric value of c_2 is greater than that of c_1, or
- c_1 and c_2 are both escaped non-printable characters and the character code of c_2 is greater than that of c_1.

Definitions for lower-case letter (`LCLetter`), upper-case letters (`UCLetter`), lower-case and upper-case letters (`Letter`) and digits (`Digit`) are shown in Figure 3.

Figure 4 gives a definition of the sort `LetterOrDigit` that recognizes a single letter (upper-case or lower-case) or digit.

Escape Conventions Characters with a special meaning in ASF+SDF may cause problems when they are needed as ordinary characters in the lexical syntax. The backslash character ('\') is used as escape character for the quoting of special characters. You should use '$\backslash c$' whenever you need special character c as ordinary character in a definition. All individual characters in character classes, except digits and letters, are *always* escaped with a backslash.

In literal strings, the following characters are special and should be escaped:

- `"`: double quote
- \: escape character.

```
module LettersDigits3
exports
  context-free start-symbols LetterOrDigit
  sorts LetterOrDigit
  lexical syntax
    [a-z] \/ [A-Z] \/ [0-9]   -> LetterOrDigit
```

Figure 5: Defining a single letter or digit using the alternative operator

You may use the following abbreviations in literals and in character classes:

- `\n`: newline character
- `\r`: carriage return
- `\t`: horizontal tabulation
- `\`nr: a non-printable character with the decimal code nr.

Character Class Operators The following operators are available for character classes

- `~`: complement of character class. Accepts all characters not in the original class.
- `/`: difference of two character classes. Accepts all characters in the first class unless they are in the second class.
- `/\`: intersection of two character classes. Accepts all characters that are accepted by both character classes.
- `\/`: union of two character classes. Accepts all characters that are accepted by either character class.

The first operator is a unary operator, whereas the other three are left-associative binary operators.

The example in Figure 5 gives the definion of a single letter or digit using the alternative operator `\/`. This definition is equivalent to the one given earlier in Figure 4.

Another example is shown in Figure 6. This definition of characters contains all possible characters, either by means of the ordinary representation or via their decimal representation.

2.9.3 Repetition

Lexical tokens are often described by patterns that exhibit a certain repetition. The operator described in Section 2.5.3 can be used to express repetitions.

The example in Figure 7 demonstrates the use of the repetition operator `*` for defining identifiers consisting of a letter followed by zero or more letters or digits.

2.9.4 Option

If zero or exactly one occurrence of a lexical token is desired the option operator described in Section 2.5.1 can be used.

The use of the option operator is illustrated in Figure 8. Identifiers are defined consisting of one letter followed by one, optional, digit. This definition accepts `a` and `z8`, but rejects `ab` or `z789`.

```
module Characters

imports basic/Whitespace

exports
  context-free start-symbols L-Char
  sorts AlphaNumericalEscChar DecimalEscChar EscChar L-Char
  lexical syntax
    "\\" ~[]                -> AlphaNumericalEscChar

    "\\" [01] [0-9] [0-9]    -> DecimalEscChar
    "\\" "2" [0-4] [0-9]     -> DecimalEscChar
    "\\" "2" "5" [0-5]       -> DecimalEscChar

    AlphaNumericalEscChar    -> EscChar
    DecimalEscChar           -> EscChar

    ~[\0-\31\"\\] \/ [\t\n]  -> L-Char
    EscChar                  -> L-Char
```

Figure 6: Example of character classes

```
module Identifiers-repetition

imports basic/Whitespace

exports
  context-free start-symbols Id
  sorts Letter DigitLetter Id
  lexical syntax
    [a-z]       -> Letter
    [a-z0-9]    -> DigitLetter

    Letter DigitLetter* -> Id
```

Figure 7: Defining identifiers using the repetition operator *

```
module Identifiers-optional

imports basic/Whitespace

exports
  context-free start-symbols Id
  sorts Letter Digit Id
  lexical syntax
    [a-z]  -> Letter
    [0-9]  -> Digit

    Letter Digit? -> Id
```

Figure 8: Defining a letter followed by an optional number using the option operator `?`

```
module Identifiers-alternative1

imports basic/Whitespace

exports
  context-free start-symbols Id
  sorts LCLetter UCLetter Digit Id
  lexical syntax
    [A-Z]   -> UCLetter
    [a-z]   -> LCLetter
    [0-9]   -> Digit

  UCLetter LCLetter* | UCLetter* | Digit* -> Id
```

Figure 9: Example of alternative operator `|`

2.9.5 Alternative

Functions with the same result sort together define the lexical syntax of tokens for that sort. The left-hand sides of these function definitions form the alternatives for this function. Sometimes, it is more convenient to list these alternatives explicitly in a single left-hand side or to list alternative parts inside a left-hand side. This is precisely the role of the alternative operator described in Section 2.5.4.

The example in Figure 9 shows how this operator can be used. It describes identifiers starting with an upper-case letter followed by one of the following:

- zero or more lower-case letters,
- zero or more upper-case letters, or
- zero or more digits.

According to this definition, `Aap`, `NOOT`, and `B49` are acceptable, but `MiES`, `B49a` and `007` are not.

Note that the relation between juxtaposition and alternative operator is best understood by looking at the line defining `Id`. A parenthesized version of this same line would read as follows:

```
UCLetter (LCLetter* | UCLetter* | Digit*) -> Id
```

As an aside, note that moving the `*` outside the parentheses as in

```
module Identifiers-alternative2

imports basic/Whitespace

exports
  context-free start-symbols Id
  sorts UCLetter LCLetter Digit Id
  lexical syntax
    [A-Z]   -> UCLetter
    [a-z]   -> LCLetter
    [0-9]   -> Digit

    (UCLetter LCLetter*) | (UCLetter UCLetter*) | (UCLetter Digit*) -> Id
```

Figure 10: Example of alternative operator `|`

```
UCLetter (LCLetter | UCLetter | Digit)* -> Id
```

yields a completely different definition: it describes identifiers starting with an uppercase letter followed by zero or more lower-case letters, uppercase letters or digits. According to this definition `MiES`, `B49a` and `Bond007` would, for instance, be acceptable.
A slightly more readable definition that is equivalent to the previous one in Figure 9 is shown in Figure 10. In any case, we recommend to use parentheses to make the scope of alternatives explicit.

2.9.6 Miscellaneous Operators

The other operators described in Section 2.5 are less frequently used within lexical syntax definitions and will not be illustrated by means of an example.

2.9.7 Examples of Lexical Syntax Definitions

We will present a number of non-trivial lexical syntax definitions in order to get some ideas what can be specified using SDF.

Defining Numbers Definitions of integers and real numbers are shown in Figure 11. Note the use of the alternative operator in the definitions of `UnsignedInt` and `Number`. Also note the use of the option operator in the definitions of `SignedInt` and `UnsignedReal`.

Defining Strings Figure 12 gives the lexical definition of strings which may contain escaped double quote characters. It defines a `StringChar` as either

- zero or more arbitrary characters except double quote or newline, or
- an escaped double quote, i.e., `\"`.

A string consists of zero or more `StringChar`s surrounded by double quotes.

2.10 Context-free Syntax

The context-free syntax describes the concrete and abstract syntactic structure of sentences in a language. A context-free syntax contains a set of declarations for *context-free functions*, each consisting of zero or more symbols followed by '→' and a result symbol. They may be followed by attributes that control how parentheses and brackets affect the abstract syntax, by attributes that define the associativity of a rule, or

```
module Numbers

imports basic/Whitespace

exports
  context-free start-symbols Number
  sorts UnsignedInt SignedInt UnsignedReal Number

  lexical syntax
    [0] | ([1-9][0-9]*)                          -> UnsignedInt

    [\+\-]? UnsignedInt                          -> SignedInt

    UnsignedInt "." UnsignedInt ([eE] SignedInt)? -> UnsignedReal
    UnsignedInt [eE] SignedInt                   -> UnsignedReal

    UnsignedInt | UnsignedReal                   -> Number
```

Figure 11: Lexical definition of Numbers

```
module Strings

imports basic/Whitespace

exports
  context-free start-symbols String
  sorts String StringChar

  lexical syntax
    ~[\"\n]               -> StringChar
    [\\][\"]              -> StringChar
    "\"" StringChar* "\"" -> String
```

Figure 12: Lexical definition of String

by attributes (Section 2.12) which influence the rewriting process. All functions with the same result sort together define the alternatives for that symbol.

Elements of the left-hand side of a context-free function are separated by an invisible non-terminal `LAYOUT?` (optional `LAYOUT`) in order to permit layout between these members. This optional layout non-terminal is automatically inserted.

2.10.1 Context-free Functions

In their simplest form, declarations of context-free functions consist of a sequence of zero or more symbols followed by '→' and a result symbol. All literal strings appearing in a context-free function declaration are implicitly added to the lexical syntax. Consider the language of coordinates and drawing commands presented in Figure 13.

An equivalent conventional BNF grammar (and not considering lexical syntax) of the above grammar of Figure 13 is presented in Figure 14.

When a literal in a context-free function consists only of lower-case letters and digits and is not a keyword of ASF+SDF, it need not be surrounded by quotes. You may therefore write '`move to COORD`

```
module DrawingCommands

imports basic/Whitespace

exports
  context-free start-symbols CMND
  sorts NAT COORD CMND

  lexical syntax
    [0-9]+ -> NAT

  context-free syntax
    "(" NAT "," NAT ")" -> COORD
    "line" "to" COORD   -> CMND
    "move" "to" COORD   -> CMND
```

Figure 13: Simple context-free syntax definition

```
<COORD> ::= "(" <NAT> "," <NAT> ")"
<CMND>  ::= "line" "to" <COORD> | "move" "to" <COORD>
```

Figure 14: BNF definition of simple grammar

`-> CMND`' instead of the previous definition given in Figure 13. But is better to always write the quotes.

2.10.2 Prefix Functions

Prefix functions are a special kind of context-free functions. They have a "fix" syntax. They can be considered as an abbreviation mechanism for functions written as expected. For instance the function `f(X,Y) -> Z` is a prefix function. This function can also be defined as an ordinary context-free function `"f" "(" X "," Y ")" -> Z`. The prefix functions are often used in combination with ASF equations.

2.10.3 Lists

Context-free syntax often requires the description of the repetition of a syntactic notion or of list structures (with or without separators) containing a syntactic notion. The repetition operator described in Section 2.5.3 can be used for this purpose.

Lists may be used in both the left-hand side and right-hand side of a context-free function as well as in the right-hand side of a variable declaration (see Section 2.16).

Figure 15 shows how lists can be used to define the syntax of a list of identifiers (occurring in a declaration in a Pascal-like language).

2.10.4 Chain Functions

A context-free syntax may contain functions that do not add syntax, but serve the sole purpose of including a smaller syntactic notion into a larger one. This notion is also known as *injections*. Injections are functions "without a name" and with one argument sort like `Id -> Data`. A typical example is the inclusion of identifiers in expressions or of natural numbers in reals. Such a *chain function* has one of the following forms:

- `SMALL -> BIG`

```
module Decls

imports basic/Whitespace

exports
  context-free start-symbols Decl
  sorts Id Decl Type

  lexical syntax
    [a-z]+ -> Id

  context-free syntax
    "decl" {Id ","}+ ":" Type -> Decl
    "integer"                 -> Type
    "real"                    -> Type
```

Figure 15: Definition of a list of identifiers

- {SMALL SEP}* -> BIG
- SMALL* -> BIG
- {SMALL SEP}+ -> BIG
- SMALL+ -> BIG
- {SMALL SEP}n+ -> BIG
- SMALLn+ -> BIG

Chain functions do not appear in the abstract syntax but correspond to a *subsort relation* between SMALL and BIG. If SORT-A is a subsort of SORT-B then in the abstract syntax tree a tree of sort SORT-A can be put wherever a tree of sort SORT-B is required. In Figure 16 the symbols Nat and Var are injected in Exp.

2.10.5 Miscellaneous Operators

In Section 2.5 a number of sophisticated operators, like alternative, option, set, function, sequence, tuple, and permutation are discussed. These operators allow a concise manner of defining grammars. There are, however, a number of issues to be taken into consideration when using this operators.

Definition of lists In the example in Figure 17, two different lists are defined, List1 represents a list of naturals separated by commas whereas List2 represents a list of naturals separated by commas and terminated by a comma.

Alternative alternatives The choice between two symbols can be defined in two different ways: by two separate syntax rules or by a single syntax rule using an alternative operator. Both styles are shown in Figure 18.

The definition of the binary operators "|" and "&" can be made more concise as shown by Bool2, however, it is now impossible to express that "&" has a higher priority than "|", see Section 2.13 for more details on priority definitions.

```
module Exp

imports basic/Whitespace

exports
  context-free start-symbols Exp
  sorts Nat Var Exp

  lexical syntax
    [0-9]+   -> Nat
    [XYZ]    -> Var

  context-free syntax
    Nat                 -> Exp
    Var                 -> Exp
    Exp "+" Exp         -> Exp
```

Figure 16: Definition of expressions that uses injections

2.10.6 Lists in combination with optionals or empty producing sorts

The combination of lists and optionals or empty producing sorts leads to cycles in the parse tree. Cycles are considered parse errors. The parser will produce an error message whenever during parsing a cycle is detected. No parse tree is constructed in such a case. Cycles will not lead to non-termination during parsing. See Figure 19 for an example of such a specification.

Sometimes commenting out parts of a production rule may lead to cycles, because a non-terminal becomes an empty producing non-terminal. This in combination with lists may then produce unexpected cycles.

2.11 Labels in the left-hand side of Functions

It is possible to decorate the members in the left-hand side of a production rule with labels. These labels have no effect when parsing input terms. However, when an SDF module is used as input for generating APIs these labels will be used. See Figure 20 for an example of an SDF specification containing labels.

2.12 Attributes of Lexical and Context-free Functions

The definition of a lexical or context-free functions may be followed by *attributes* that define additional (syntactic or semantic) properties of that function. The attributes are written between curly brackets after the non-terminal in the right hand side. If a production rule has more than one attribute they are separated by commas.

```
context-free syntax
   "{" {Attribute ","}* "}" -> Attributes {cons("attrs")}
                            -> Attributes {cons("no-attrs")}
```

The following syntax-related attributes exist:

- `bracket` allows the definitions of parenthesis and other kinds of brackets that are mostly used for overruling the priorities of operators in expressions (see Section 2.13.1).
- `left`, `right`, `non-assoc`, and `assoc` are used to define the associativity of functions (see Section 2.13).

```
module Lists

imports basic/Whitespace

exports
  context-free start-symbols List1 List2
  sorts Nat List1 List2

  lexical syntax
    [0-9]+   -> Nat

  context-free syntax
    {Nat ","}+ -> List1
    (Nat ",")+ -> List2
```

Figure 17: Definition of two list variants

- `prefer` is used to indicate that the attributed function should always be preferred over other functions (without this attribute) in certain cases of syntactic ambiguity (see Section 2.13.6).
- `avoid` is used to indicate that a function should only be used as a last resort in certain cases of syntactic ambiguity (see Section 2.13.6).
- `reject` can be used to explicitly forbid certain syntactic constructs (see Section 2.13.6).

The remaining attributes define semantic properties of a function:

- `constructor` declares a function to be a *constructor function*, this means that for this function *no* equations may be defined with this function as outermost function symbol in the left hand side.
- `memo` declares a function to be a *memo function* for which all calls and results will be cached during evaluation (see Section 2.18.7).
- `traversal` is used to declare so-called traversal functions that greatly simply the specification of functions that have to visit (parts of) a term (see Section 2.18.8).
- arbitrary ATerms may also be used as attributes. In the context-free syntax definition of the `Attributes`, the ATerms `cons("attrs")` and `cons("no-attrs")` are used. The `cons` attribute is used by other tools, such as ApiGen.

Not all combinations of attributes make sense. If one uses the attribute `left` in combination with `bracket`, `right`, `assoc` or `non-assoc`, this will result in an error message. The combination of `avoid` and `prefer` does not make sense either. Furthermore, the combination of the traversal attributes is also very strict.

2.13 Priorities

The context-free syntax defined in an ASF+SDF specification may be ambiguous: there are sentences which have more than one associated tree. The common example is the arithmetic expression in which definitions of the priority or associativity of operators are missing. There are three mechanisms for defining associativity and priority:

- Relative priorities of functions (see Section 2.13.2) defined in the `context-free priorities` section.

```
module Bool

imports basic/Whitespace

exports
  context-free start-symbols Bool1 Bool2
  sorts Bool1 Bool2

  context-free syntax
    "true"                  -> Bool1
    "false"                 -> Bool1
    Bool1 "|" Bool1         -> Bool1 {left}
    Bool1 "&" Bool1         -> Bool1 {left}

    "true" | "false"        -> Bool2
    Bool2 ("|" | "&") Bool2 -> Bool2 {left}
```

Figure 18: Two ways of defining | and &

```
module Cycle

imports basic/Whitespace

exports
  context-free start-symbols T
  sorts A P T

  context-free syntax
    "a"        -> A
    A?         -> P
    "[" P+ "]" -> T
```

Figure 19: Dangerous combination of lists and optionals

```
module Booleans

imports basic/Whitespace

exports
  context-free start-symbols Boolean
  sorts Boolean

  context-free syntax
    lhs:Boolean "|" rhs:Boolean -> Boolean
    lhs:Boolean "&" rhs:Boolean -> Boolean
```

Figure 20: The module `basic/Booleans` decorated with labels

```
module BracketExpr

imports basic/Whitespace
imports basic/NatCon

exports
  context-free start-symbols E
  sorts E

  context-free syntax
    NatCon    -> E
    "(" E ")" -> E {bracket}
```

Figure 21: Syntax definition with a bracket function

- Associativity of functions (see Section 2.13.3) defined as attributes following the function declaration.
- Associativity of groups of functions (see Section 2.13.4) defined in the `context-free priorities` section.

Closely related with priorities are brackets that can be used to overrule priorities. We will first describe bracket functions, and then the various methods for defining priorities.

2.13.1 Bracket Functions

A bracket function has the form '$open\ \ S\ \ close$ `->` S' where $open$ and $close$ are literals acting as opening and closing parenthesis for sort S. Examples are '`(`' and '`)`' in arithmetic expressions. In most cases, such brackets are only introduced for grouping and disambiguation, but have no further meaning. By adding the attribute `bracket` to the function declaration, it will not be included in the abstract syntax. The definition of a bracket function for the sort `Expr` is given in Figure 21.

Since brackets are necessary for overruling the priority and associativity of functions, it is required that bracket functions are declared for the argument and result sorts of

- all functions appearing in priority declarations, and
- all functions having one of the attributes `left`, `right`, `assoc`, or `non-assoc`.

2.13.2 Relative Priorities

The relative priority of two functions is defined in the '`context-free priorities`' section by including F `>` G, where F and G are as written in the context-free grammar. Functions with a higher priority bind more strongly than functions with lower priorities and the nodes corresponding to them should thus appear at lower levels in the tree than nodes corresponding to functions with lower priorities. Lists of functions may be used in a priority declaration: F `>` $\{G,\ H\}$ is an abbreviation for F `>` G, F `>` H. Note that this tells us nothing about the priority relation between G and H.

2.13.3 Associative Functions

Associativity attributes can be attached to binary functions of the form '$S\ \ op\ \ S$ `->` S', where op is a symbol or empty. Without associativity attributes, nested occurrences of such functions immediately lead to ambiguities, as is shown by the sentence '`S-string op S-string op S-string`' where '`S-string`' is a string produced by symbol S. The particular associativity associated with op determines the intended interpretation of such sentences.

```
module SimpleExpr

imports basic/Whitespace
imports basic/NatCon

exports
  context-free start-symbols E
  sorts E

  context-free syntax
    NatCon    -> E
    E "+" E   -> E {left}
    E "*" E   -> E {left}
    "(" E ")" -> E {bracket}

  context-free priorities
    E "*" E -> E >
    E "+" E -> E
```

Figure 22: Simple context-free priority definition

We call two occurrences of functions F and G *related*, when the node corresponding to F has a node corresponding to G as first or last child. The associativity attributes define how to accept or reject trees containing related occurrences of the same function, F:

- `left`: related occurrences of F associate from left to right.
- `right`: related occurrences of F associate from right to left.
- `assoc`: related occurrences of F associate from left to right.
- `non-assoc`: related occurrences of F are not allowed.

Currently, there is no syntactic or semantic difference between '`left`' and '`assoc`', but we may change the semantics of the '`assoc`' attribute in the future.

Figure 22 gives an example of a definition of simple arithmetic expressions with the usual priorities and associativities.

2.13.4 Groups of Associative Functions

Groups of associative functions define how to accept or reject trees containing related occurrences of different functions with the same priority. They are defined by prefixing a list of context-free functions in a priority declaration with one of the following attributes:

- `left`: related occurrences of F and G associate from left to right.
- `right`: related occurrences of F and G associate from right to left.
- `non-assoc`: related occurrences of F and G are not allowed.

where F and G are functions appearing in the list. Figure 23, an example of the use of grouped associativity.

```
module ComplexExpr

imports basic/Whitespace
imports basic/NatCon

exports
  context-free start-symbols E

  sorts E

  context-free syntax
    NatCon    -> E
    E "+" E   -> E {left}
    E "-" E   -> E {non-assoc}
    E "*" E   -> E {left}
    E "/" E   -> E {non-assoc}
    E "^" E   -> E {right}
    "(" E ")" -> E {bracket}

  context-free priorities
    E "^" E -> E >
    {non-assoc: E "*" E -> E
                E "/" E -> E} >
    {left: E "+" E -> E
           E "-" E -> E}
```

Figure 23: More complex associativity and priority definitions

```
module Functional

imports basic/Whitespace

exports
  context-free start-symbols Term
  sorts Var Term
  lexical syntax
    [a-z]+ -> Var
  context-free syntax
    Var                          -> Term
    Term Term                    -> Term {left}
    "let" Var "=" Term "in" Term -> Term

  lexical restrictions
    "let" "in" -/- [a-z]

  context-free restrictions
    Var -/- [a-z]
```

Figure 24: Using restrictions in the definition of a simple functional language

2.13.5 Restrictions

Lexical syntax can be highly ambiguous. Consider a simple lexical definition for identifiers like the one given earlier in Figure 7. When recognizing the text `abc`, what should we return: `a`, `ab` or, `abc`? We discuss the strategy *Prefer Longest Match* for resolving this kind of ambiguity in Section 2.14.1.

Here, we describe the notion of *restrictions* that enable the formulation of this and other lexical disambiguation strategies.

A restriction limits the *lookahead* for a given symbol; it indicates that a symbol may not be followed by a character from a given character class. A lookahead may consist of more than one character class. Restrictions come in two flavors:

- lexical restrictions;
- context-free restrictions.

The general form of a restriction is

```
<Symbol>+ -/- <Lookaheads>
```

In case of lexical restrictions `<Symbol>` may be either a literal or sort. In case of context-free restrictions only a sort or symbol is allowed. The restriction operator `-/-` should be read as "may not be followed by". Before the restriction operator `-/-` a list of symbols is given for which the restriction holds.

In the example[1] in Figure 24 both `let` and `in` may not be followed by a letter. This example shows how lexical restrictions can be used to prevent the recognition of erroneous expressions in a small functional language. The lexical restriction deals with the possible confusion between the reserved words `let` and `in` and variables (of sort `Var`). It forbids the recognition of, for instance, `let` as part of `letter`. Without this restriction `letter` would be recognized as the keyword `let` followed by the variable `ter`. The context-free restriction forbids that a variable is directly followed by a letter. It does not forbid layout characters between the letters, e.g. `a b` is a legal recognizable string.

`<Lookaheads>` are slightly more complex. The most compact way is to give the SDF definition of the `<Lookaheads>` and illustrate their use by means of some examples.

[1] Taken from [35]

```
module basic/Whitespace

exports
  lexical syntax
    [\ \t\n] -> LAYOUT

  context-free restrictions
    LAYOUT? -/- [\ \t\n]
```

Figure 25: Safe way of defining LAYOUT

```
module Comment

imports basic/Whitespace

exports
  sorts ComWord Comment
  lexical syntax
    ~[\ \n\t\/]+ -> ComWord

  context-free syntax
    "/*" ComWord* "*/" -> Comment
    Comment            -> LAYOUT

  context-free restrictions
    LAYOUT? -/- [\ \t\n]
    LAYOUT? -/- [\/].[\*]
```

Figure 26: Definition of C comments

```
context-free syntax
  CharClass                    -> Lookahead
  CharClass "." Lookaheads     -> Lookahead
  Lookahead                    -> Lookaheads
  Lookaheads "|" Lookaheads    -> Lookaheads {right}
  "(" Lookaheads ")"           -> Lookaheads {bracket}
  "[[" {Lookahead ","}* "]]"   -> Lookaheads
```

The next example illustrates the use of restrictions to define a 'safe' way of layout. Recall from Section 2.9 that optional layout, represented by the symbol LAYOUT?, may be recognized between the members of the left-hand side of a context-free syntax rule.

However, if a such a member recognizes the empty string, this gives rise to a lexical ambiguity (Section 2.14.1). This problem is avoided by the definition given in Figure 25: it simply forbids that optional layout is followed by layout characters.

The example shown in Figure 26 illustrates the use of restrictions to extend the previous layout definition with C-style comments. For readability we give here *two* restrictions whereas the first one is already imported from module basic/Whitespace (Figure 25). The repetition of this first restriction is redundant and could be eliminated.

A frequently asked question is when to use *lexical* restrictions and when to use *context-free* restrictions. In one of the previous examples (Figure 24) the lexical restrictions on let and in cannot be defined using context-free restrictions because these keywords do not "live" at the context-free level. Is it possible to put

```
module RestrictedExpressions

imports basic/Whitespace

exports
  context-free start-symbols Expr
  sorts Expr

  lexical syntax
    [a-z]+ -> Expr

  context-free syntax
    Expr Expr    -> Expr {left}
    "(" Expr ")" -> Expr {bracket}

  context-free restrictions
    Expr -/- [a-z]
```

Figure 27: Erroneous use of restrictions in the definition of simple expressions

a lexical restriction on `Var`? Yes, but it will have no effect, because internally the lexical `Var` is injected in the context-free `Var`. The general rule is to define the restrictions always on the context-free level and not on the lexical level unless a situation as will be discussed in the next paragraph occurs.

The specification in Figure 27 is an example of an erroneous use of context-free expressions, because it prevents the recognition of `(abc)def`. If we want to enforce the correct restriction, it is necessary to transform this context-free restriction into a lexical restriction.

2.13.6 Preferring, Avoiding or Rejecting Parses

Priorities can be used to define a priority between two functions or between two groups of functions. In both cases the functions involved have to be listed explicitly in the priority declaration. In certain cases, however, it is desirable to define that a single rule has higher or lower priority than all other functions or to explicitly reject certain syntactic constructs. The former is achieved by the attributes `prefer` and `avoid`. The latter by the attribute `reject`.

The use of the `reject` attribute leads also to improvements in the performance of the parser, see [17] for more implementation details.

If a function F is attributed with `prefer` and there is a syntactic ambiguity in which it is involved, only the parse using F will remain.

If a function F is attributed with `avoid` and there is no ambiguity, then F will be used. If there is an ambiguity, then F will be immediately removed from the set of ambiguities.

If a function F is attributed with `reject`, then independently of the number of ambiguities, the parse using F will be removed. While restrictions (Section 2.13.5) only impose limitations on the immediate lookahead that follows a symbol, the reject mechanism can be used to eliminate complicated syntactic structures.

Examples of the use of `prefer`, `avoid` and `reject` in order to solve lexical ambiguities are discussed in Section 2.14.1. In Section 2.14.2 we will give examples of how to use these attributes to solve context-free ambiguities, such as the famous dangling else problem.

2.14 Disambiguation

2.14.1 Lexical Ambiguities

SDF provides a number of elementary lexical disambiguation features but does not offer *fully automated*

```
module Identifiers-restrict

imports basic/Whitespace

exports
  context-free start-symbols Id
  sorts Id
  lexical syntax
    [a-zA-Z][a-zA-Z0-9]* -> Id

  context-free restrictions
    Id -/- [a-zA-Z0-9]
```

Figure 28: Using context-free restrictions to define a longest match for identifiers

lexical disambiguation. As a result, the specification writer has to be aware of lexical ambiguities and has to specify disambiguation rules explicitly. We will discuss various approaches to lexical disambiguation and illustrate them by means of examples.

Prefer Longest Match per Sort Reject all interpretations of the input text that are included in a longer interpretation of the same sort. Given a standard definition of identifiers, the input '`xyz`' will thus lead to recognition of the identifier '`xyz`' and not to either '`x`' or '`xy`'.

This is achieved by defining a restriction on this lexical sort. This can be done using either lexical or context-free restrictions (see Section 2.13.5). The specification in Figure 28 shows how to enforce the longest match for the sort `Id`.

Prefer Literals In the left-hand side of a context-free syntax rule literals (keywords and/or operators) may be used. If these literals overlap with more general lexical tokens (such as identifier) this causes ambiguities.

The strategy *Prefer Literals* gives preference to interpretation as a literal, over interpretation as a more general lexical token. For instance, the keyword `begin` may be recognized as an identifier given the lexical definition in Figure 28.

There are two approaches to implement Prefer Literals.

In the first approach, we can explicitly forbid the recognition of literals as tokens of a specific sort using the reject mechanism (see Section 2.13.6). The idea is to define context-free grammar rules for all literals with the undesired lexical sort (e.g., `Id`) in the right-hand side followed by the attribute `reject`. This is illustrated in Figure 29. The `reject` attribute indicates here that the recognition of a keyword as a literal of the sort `Id` should be rejected. This approach has the major disadvantage that the addition of a literal in any context-free rule also requires the addition of a new reject rule for that literal.

The second approach is more attractive. The lexical definition of the general notion that interferes with our literals is written in such a way that it is only used as a last resort. In other words, it is avoided as much as possible and is only used when no alternative exists. The attribute `avoid` defines precisely this behaviour (see Section 2.13.6). Figure 30 shows how the lexical definition of `Id` is attributed with `avoid`.

Although the first approach is more tedious, it allows more flexibility than the second one.

Prefer Non-Layout If there are interpretations of the text as layout symbol and as non-layout symbol, eliminate all interpretations as layout symbol. This is built-in behaviour of ASF+SDF.

Prefer Variables Give preference to interpretation as a variable (as defined in a variables section) over interpretation as a lexical token. Thus built-in behaviour of ASF+SDF. It is achieved by automatically extending each variable declaration with the attribute `prefer` (see Section 2.13.6).

```
module Identifiers-reject

imports basic/Whitespace

exports
  context-free start-symbols Id
  sorts Id

  lexical syntax
    [a-zA-Z][a-zA-Z0-9]* -> Id

  context-free restrictions
    Id -/- [a-zA-Z0-9]

  context-free syntax
    "begin" -> Id {reject}
```

Figure 29: Using `reject` to implement Prefer Literals

```
module Identifiers-avoid

imports basic/Whitespace

exports
  context-free start-symbols Id
  sorts Id

  lexical syntax
    [a-zA-Z][a-zA-Z0-9]* -> Id {avoid}

  context-free restrictions
    Id -/- [a-zA-Z0-9]
```

Figure 30: Using `avoid` to implement Prefer Literals

2.14.2 Context-free Ambiguities

Context-free grammars may be ambiguous and, as a result, the parser may yield different parses of a text. More precisely, the result of a parse is a single tree in which the ambiguities are explicitly marked. Each marked ambiguity consists of a set of different parse trees for that ambiguity. Many—but not all!— of these different parses can be eliminated by the following strategies that are built-in the ASF+SDF Meta-Environment. These strategies use the priorities and associativities as defined in the specification. In addition, some standard heuristics are used.

Associativity filtering The associativity filtering is performed during the generation of the parse table. Based on the associativity relations certain entries in the parse table are removed.

Removing Trees containing Conflicts The simplest application of priority and associativity declarations is the elimination of trees that contain conflicts:

- A parent node has a child with a lower priority than the parent itself.
- A parent has a first or last child that is in conflict with an associativity relation between this parent and child.

Reconsidering the example of complex priorities shown in Figure 23 we will give a number of example sentences and the interpretation given to them by that language definition.

Sentence	Interpretation
`1^2^3`	`1^(2^3)`
`1^2*3`	`(1^2)*3`
`1*2*3`	`(1*2)*3`
`1/2/3`	error
`1*2/3`	error
`1-2-3`	error
`1+2+3`	`(1+2)+3`
`1-2+3`	`(1-2)+3`
`1+2-3`	`(1+2)-3`

Removing Trees using `prefer`/`avoid` Attributes at the Root The priority declarations are used to eliminate trees in three phases:

1. If there are trees of which the syntax rule at the top node has a `prefer` attribute, all other trees are removed.
2. If there are trees of which the syntax rule at the top node has an `avoid` attribute and there are other trees without an `avoid` attribute at the root node, then all trees with `avoid` attribute are removed.

Removing Trees containing `prefer`/`avoid` Attributes After removing all trees containing conflicts, more than one tree may still remain. To further reduce this set of remaining trees, the number of context-free functions with `prefer`/`avoid` attributes is calculated and compared. A tree in the set is then rejected if there is another tree in the set with more `prefer`s and less or equal `avoid`s, or with equal `prefer`s and more `avoid`s.

Injection count Finally, the number of injections in each of the resulting trees is calculated, the tree with the smallest number of injections is prefered.

```
module Eqn

imports basic/Whitespace

exports
  context-free start-symbols E
  sorts E

  context-free syntax
    E "sub" E         -> E {left}
    E "sup" E         -> E {left}
    E "sub" E "sup" E -> E {prefer}
    "{" E "}"         -> E {bracket}
    "a"               -> E
```

Figure 31: Syntax definition of EQN expressions

```
module DanglingElse

imports basic/Whitespace

exports
  context-free start-symbols S
  sorts E S

  context-free syntax
    "a"                      -> E
    "if" E "then" S          -> S {prefer}
    "if" E "then" S "else" S -> S
    "s"                      -> S
```

Figure 32: Syntax definition of conditionals

Examples The following examples show how the interaction (and resulting ambiguities) between general context-free functions and special case functions can be described using `prefer` attribute.

The first example(Figure 31 concerns expressions for describing subscripts and superscripts in the typesetting language EQN. The crucial point is that, for typesetting reasons, we want to treat a subscript followed by a superscript in a special way. Therefore, the special case '`E sub E sup E -> E`' is introduced, which is prefered over a combination of the two functions '`E sub E -> E`' and '`E sup E -> E`'.

In the second example (Figure 32) the `prefer` attribute is used to solve the dangling else problem in a nice way. The input sentence `"if 0 then if 1 then hi else ho"` can be parsed in two ways: `if 0 then (if 1 then hi) else ho` and `if 0 then (if 1 then hi else ho)`. We can select the latter derivation by adding the `prefer` attribute to the production without the `else` part. The parser will still construct an ambiguity node containing both deriviations, namely, `if 0 then (if 1 then hi {prefer}) else ho` and `if 0 then (if 1 then hi else ho) {prefer}`. But given the fact that the *top* node of the latter derivation tree has the prefer attribute this derivation is selected and the other tree is removed from the ambiguity node.

```
module Pair[X Y]

imports basic/Booleans

hiddens
  sorts X Y

exports
  context-free start-symbols Pair[[X,Y]]
  sorts Pair[[X,Y]]

  context-free syntax
    "[" X "," Y "]"      -> Pair[[X,Y]]

    make-pair(X, Y)      -> Pair[[X,Y]]
    first(Pair[[X,Y]])   -> X
    second(Pair[[X,Y]])  -> Y
    is-pair(Pair[[X,Y]]) -> Boolean
```

Figure 33: Definition of generic pairs

2.15 Parameterization and Renaming

Parameterization and renaming were in fact features of the original ASF as described in [1], but they were never supported by the ASF+SDF used in the first version of the ASF+SDF Meta-Environment [30]. Based on the work described in [35], ASF+SDF is extended with parameterization and symbol renaming[2]. We will first explain the notion of parameterization, later we will give details on symbol renaming.

2.15.1 Parameterization

Module parameterization allows the definition of generic modules for lists, pairs, sets, etc. The operations defined in these modules are independent of a specific type. When importing a parameterized module and instantiating the formal by actual parameters the operations become ”sort” specific.

Modules can have formal parameters when defining them. The module name is then followed by a list of symbols, representing the formal parameters of this module. The specification in Figure 33 shows an example of a parameterized module. In this example the formal parameters are used in the parameterized sorts as well, in order to increase readability and to avoid name clashes between different instances of the same module.

When importing a parameterized module the formal parameters have to be replaced by actual parameters. The specification in Figure 34 shows an example of a rather complicated import of a parameterized module. The symbols `Pair[[Boolean,Boolean]]` and `Pair[[Integer,Integer]]` are the actual parameters of the module `Pair[X Y]` in the last import.

2.15.2 Symbol Renaming

Symbol renaming is in fact very similar to parameterization except that it is not necessary to add formal parameters to a module. The mechanism of symbol renaming allows the overriding of one symbol or a set of symbols by another symbol or symbols, respectively. It allows a flexible and concise way of adapting specifications. The specification in Figure 35 shows an example of the `Pair` module without parameters. The idea is to achieve the same effect as parameterization by explicitly renaming `X` and `Y` to the desired names when `Pair` is imported.

[2] In [35] the notion of production renaming is also introduced, but this will not be supported.

```
module TestPair

imports basic/Booleans Pair[Boolean Boolean]
        basic/Integers Pair[Integer Integer]
        Pair[Pair[[Boolean,Boolean]] Pair[[Integer,Integer]]]
```

Figure 34: Use of generic pair module

```
module Pair

imports basic/Booleans

hiddens
  sorts X Y

exports
  context-free start-symbols Pair[[X,Y]]
  sorts Pair[[X,Y]]

  context-free syntax
    "[" X "," Y "]"      -> Pair[[X,Y]]

    make-pair(X, Y)      -> Pair[[X,Y]]
    first(Pair[[X,Y]])   -> X
    second(Pair[[X,Y]])  -> Y
    is-pair(Pair[[X,Y]]) -> Boolean
```

Figure 35: Definition of generic pairs

```
module TestPair

imports basic/Booleans Pair[X => Boolean Y => Boolean]
        basic/Integers Pair[X => Integer  Y => Integer]
        Pair[X => Pair[[Boolean,Boolean]] Y => Pair[[Integer,Integer]]]
```

Figure 36: Use of generic pair module

```
module VarDecls

imports basic/Whitespace

exports
  context-free start-symbols Decl
  sorts Id Decl Type

  lexical syntax
    [a-z]+ -> Id

  context-free syntax
    "decl" {Id ","}+ ":" Type -> Decl
    "integer"                 -> Type
    "real"                    -> Type

hiddens
  variables
    "Id"           -> Id
    "Type"[0-9]*   -> Type
    "Id-list"[\']* -> {Id ","}*
    "Id-ne-list"   -> {Id ","}+
```

Figure 37: Variable declarations using naming schemes

During import such module symbols can be renamed via symbol renaming. The specification in Figure 36 shows an example of a rather complicated import of the module `Pair` using renamings. Renaming `X` to `Boolean` is, for instance, written as `X => Boolean`.

2.16 Variables

Variables are declared in the '`variables`' section of a module. Like all other entities in a module—except equations—variables may be exported (see Section 2.2). A variables section consists of a list of variable names followed by a symbol. In fact, a variable declaration can define an infinite collection of variables by using a *naming scheme* instead of a simple variable name. A naming scheme is a regular expression like the ones allowed in the lexical syntax (Section 2.9) except that sorts are not allowed. A variable may represent any symbol.

In the specification in Figure 37, '`Id`', '`Type3`', and '`Id-list`' are examples of variables declared by the naming schemes in the `variables` section. Strings that occur in the left-hand side of variable declarations should *always* be quoted.

Declared variables can only be used when defining equations. It is not allowed to use them in terms.

Ambiguities due to variables are resolved by the *Prefer Variables* strategy that was discussed in Sec-

tion 2.14.1.

2.17 Libraries

Via the graphical user interface of the ASF+SDF Meta-Environment one has access to a number of predefined modules. These library modules are divided into 4 different categories:

1. `basic`
2. `containers`
3. `languages`
4. `utilities`

Each of these categories offers a number of useful library modules, e.g., the category `basic` offers the library modules for `Booleans`, `Integers`, `Strings`, etc. The category `containers` offers modules `List`, `Set`, and `Table`. The category `languages` offers a number of predefined syntax definitions.

One can import these library modules in a specification in the following way: `imports basic/Booleans` for a library module without parameters and `imports containers/List[Boolean]` if it is a parameterized library module. In the various examples presented upto now the library modules are used whenever appropriate.

When developing a new specification make sure that you check the libraries in order to reduce the amount of work.

2.17.1 Position Information

In a number of cases, for instance when producing error messages, it can be very useful to retrieve the position information of subterms.

2.17.2 Error messages

2.18 Equations

With equations a meaning or semantics may be added to functions declared in the lexical and context-free syntax sections. In particular, equations consist of two *open terms*, i.e. terms possibly containing variables.

In the context of ASF+SDF, an open term is any string that can be parsed according to one of the sorts in the specification (possibly including variables). Examples of (open) terms are '`true`', '`not(false)`', and '`true | Bool`'.

2.18.1 Unconditional Equations

An equality then consists of two (possibly open) terms L (lefthand side) and R (righthand side) such that:

- L and R are of the same sort.
- L is not a single variable.
- The variables that occur in R also occur in L.

It is assumed that the variables occurring in the equation are universally quantified. In other words, the equality holds for all possible values of the variables.

The equality of two terms L and R is defined in ASF+SDF by the following *unconditional* equation:

$$[TagId]\ L\ =\ R$$

where $TagId$ is a sequence of letters, digits, and/or minus signs (`-`) starting with a letter or a digit.

2.18.2 Conditional Equations

An unconditional equation is a special case of a *conditional equation*, i.e., an equality with one or more associated conditions (premises). The equality is sometimes called the *conclusion* of the conditional equation.

In ASF+SDF a conditional equation can be written in three (syntactically different, but semantically equivalent) ways:

(a) [$TagId$] L `=` R `when` C_1`,` C_2`, ...`

(b) [$TagId$] C_1`,` C_2`, ...` `===>` L `=` R

(c) [$TagId$] C_1`,` C_2`, ...`
`==========`
L `=` R

where C_1, C_2, ... are conditions which may be either matching (and have the form 'S `:=` T'), negative matching (and have the form 'S `:!=` T'), positive (and have the form 'S `==` T'), or negative (and have the form 'S `!=` T').

The conditions of an equation are evaluated from left to right. Let, initially, V be the set of variables occurring in the left-hand side L of the conclusion of the equation. For the evaluation of matching conditions we have the following case:

- Left-hand side of a matching condition must contain at least one new or fresh variable not in V. Reduce the right-hand side of the matching condition to a normal form and the matching condition succeeds if this normal form and the left-hand side of the condition match. The new variables resulting from this match are added to V. This kind of condition is called a *match* condition. The variables occurring in both V and the left-hand side must represent the syntactically the same subterm.

For the evaluation of each positive condition we distinguish the following cases:

- The condition contains only variables in V. Reduce both sides of the condition to normal form and the condition succeeds if both normal forms are identical. Technically, this is called a *join* condition.

The evaluation of negative conditions is described by replacing in the above description 'identical' and 'match' by 'not identical' and 'do not match', respectively.

Warning: It is not allowed to introduce new variables in a negative condition.

After the successful evaluation of the conditions, all variables occurring in the right-hand side of the conclusion of the equation should be in V.

New variables (see above) should therefore **not** occur on *both* sides of a positive condition, in a negative condition, or in the right-hand side of the conclusion.

2.18.3 Executing Equations

In the ASF+SDF Meta-Environment, equations can be executed as *rewrite rules*. The above equation is thus executed as the rewrite rule $L \rightarrow R$. This can be used to reduce some initial closed term (i.e., not containing variables) to a *normal form* (i.e., a term that is not reducible any further) by repeatedly applying rules from the specification.

A term is always reduced in the context of a certain module, say M. The rewrite rules that may be used for the reduction of the term are the rules declared in M itself and in the modules that are (directly or indirectly) imported by M.

The search for an applicable rule is determined by the reduction strategy, that is, the procedure used to select a subterm for possible reduction. In our case the *leftmost-innermost* reduction strategy is used. This means that a left-to-right, depth-first traversal of the term is performed and that for each subterm encountered an attempt is made to reduce it.

Next, the rules are traversed one after the other. The textual order of the rules is irrelevant. Instead they are ordered according to their *specificity*: more specific rules come before more general rules and default

equations (see Section 2.18.6) come last. *Independent of the specificity, a specification should always be confluent and terminating.*

If the selected subterm and the left-hand side of a rule (more precisely: of the left-hand side of its conclusion) match, we say that a *redex* has been found and the following happens. The conditions of the rule are evaluated and if the evaluation of a condition fails, other rules (if any) with matching left-hand sides are tried. If the evaluation of all conditions succeeds, the selected subterm is replaced by the right-hand side of the rule (more precisely: the right-hand side of the conclusion of the rule) after performing proper *substitutions*. Substitutions come into existence by the initial matching of the rule and by the evaluation of its conditions. For the resulting term the above process is repeated until no further reductions are possible and a normal form is reached (if any).

2.18.4 List Matching

List matching, also known as *associative matching*, is a powerful mechanism to describe complex functionality in a compact way.

The example in Figure 38 shows a compact specification to remove double elements from a set.

Unlike the matching of ordinary (non-list) variables, the matching of a list variable may have more than one solution since the variable can match lists of arbitrary length.

As a result, backtracking is needed. For instance, to match `X Y` (a list expression containing the two list variables `X` and `Y` indicating the division of a list into two sublists) with the list `ab` (a list containing two elements) the following three alternatives have to be considered:

```
X = (empty), Y = ab,
X = a, Y = b,
X = ab, Y = (empty).
```

In the unconditional case, backtracking occurs only during matching. When conditions are present, the failure of a condition following the match of a list variable leads to the trial of the next possible match of the list variable and the repeated evaluation of following conditions.

Another example of list matching in combination with the evaluation of conditions is shown in Figure 39. A list of elements is split into two parts of equal length, if the list has an even number of elements. In case of a list of uneven length the middle element is ignored. The first part of the list is returned as result.

2.18.5 Lexical Constructor Functions

The only way to access the actual characters of a lexical token is by means of the so-called *lexical constructor functions*. For each lexical sort LEX a lexical constructor function is automatically derived, the corresponding syntax definition is: lex`( CHAR* ) ->` LEX. The sort `CHAR` is a predefined sort to access the characters.

Characters can be directly addressed by the representation or via variables which may be of the sorts `CHAR`, `CHAR*`, or `CHAR+`. The latter two represent lists of characters. In the example in Figure 40 the lexical constructor function `nat-con` is used to remove the leading zeros from a number.

Warning: The argument of a lexical constructor may be an arbitrary list of characters and there is *no check that they match the lexical definition of the corresponding sort.* This means that when writing a specification one should be aware that it is possible to construct illegal lexical entities, for instance, by inserting letters in an integer. In the example in Figure 41 via the lexical constructor function `nat-con` a natural number containing the letter `a` is constructed.

2.18.6 Default Equations

The evaluation strategy for normalizing terms given the equations is based on innermost rewriting. All equations have the same priority. Given the outermost function symbol of a redex the set of equations with this outermost function symbol in the left-hand side is selected and all these rules will be tried. However, sometimes a specification writer would like to write down a rule with a special status "*try this rule if all other rules fail*". A kind of default behaviour is needed. ASF offers functionality in order to obtain this

```
module Sets

imports basic/Whitespace

exports
  context-free start-symbols Set
  sorts Elem Set

  lexical syntax
    [a-z]+ -> Elem

  context-free syntax
    Set[Elem] -> Set

hiddens
  variables
    "Elem"[0-9]*  -> Elem
    "Elem*"[0-9]* -> {Elem ","}*

equations

  [set] {Elem*1, Elem, Elem*2, Elem, Elem*3} = {Elem*1, Elem, Elem*2, Elem*3}
```

Figure 38: Set specification

behaviour. If the $TagId$ of an equation starts with `default-` this equation is considered to be a special equation which will only be applied if no other rule matches. The specification in Figure 42 shows an example of the use of a default equation.

2.18.7 Memo Functions

Computations may contain unnecessary repetitions. This is the case when a function with the same argument values is computed more than once. Memo functions exploit this behaviour and can improve the efficiency of ASF+SDF specifications.

Given a set of argument values for some function the normal form can be obtained via rewriting. It is possible that some function is called with the same set of arguments over and over again. Each time the function is rewritten to obtain the same normal again. By means of adding the `memo` attribute, this behaviour is improved by storing the set of argument values and the derived normal form in a memo-table. For each set of argument values it is checked whether there exists a normal form in the memo-table. If so, this normal form is returned. If not, the function given this set of argument values is normalized and stored in the memo-table. There is some overhead involved in accessing the memo-table. Therefore, it is not wise to add the memo attribute to each function. With respect to the operational behaviour adding a memo attribute does not have any effect.

The Fibonacci function shown in Figure 43 is decorated with the memo attribute to improve its efficiency.

The resulting improvement in performance is shown in Table 1.

2.18.8 Traversal Functions

Program analysis and program transformation usually take the syntax tree of a program as starting point. One common problem that one encounters is how to express the *traversal* of the tree: visit all the nodes of the tree and extract information from some nodes or make changes to certain other nodes.

```
%% Split.sdf
module Split

imports basic/Integers
imports basic/Whitespace

exports
  context-free start-symbols List
  sorts El List

  lexical syntax
    [a-z]+ -> El
  context-free syntax
    {El ","}*          -> List
    length(List)       -> Integer
    split-in-two(List) -> List

hiddens
  variables
    "El"[0-9]*  -> El
    "El*"[0-9]* -> {El ","}*

%% Split.asf
equations

  [l-1] length() = 0

  [l-2] length(El, El*) = 1 + length(El*)

  [s-1] length(El*1) == length(El*2)
        ====>
        split-in-two(El*1, El*2) = El*1

  [s-1] length(El*1) == length(El*2)
        ====>
        split-in-two(El*1, El, El*2) = El*1
```

Figure 39: Split-in-two specification

The kinds of nodes that may appear in a program's syntax tree are determined by the grammar of the language the program is written in. Typically, each rule in the grammar corresponds to a node category in the syntax tree. Real-life languages are described by grammars which can easily contain several hundred, if not thousands of grammar rules. This immediately reveals a hurdle for writing tree traversals: a naive recursive traversal function should consider many node categories and the size of its definition will grow accordingly. This becomes even more dramatic if we realize that the traversal function will only do some real work (apart from traversing) for very few node categories.

Traversal functions in ASF+SDF [13] solve this problem. We distinguish three kinds of traversal functions, defined as follows.

Transformer: a sort-preserving transformation that will traverse its first argument. Possible extra arguments may contain additional data that can be used (but not modified) during the traversal. A transformer is declared as follows:

$$f(S_1, ..., S_n) \rightarrow S_1\{\texttt{traversal(trafo, ...)}\}$$

```
module Nats

imports basic/Whitespace

exports
  context-free start-symbols Nat-con
  sorts Nat-con

  lexical syntax
    [0-9]+ -> Nat-con

hiddens
  variables
    "Char+"[0-9]* -> CHAR+

equations

  [1] nat-con("0" Char+) = nat-con(Char+)
```

Figure 40: Use of lexical constructor functions

fib(n)	Time without memo (sec)	Time with memo (sec)
fib(16)	2.0	0.7
fib(17)	3.5	1.1
fib(18)	5.9	1.8
fib(19)	10.4	3.3

Table 1: Execution times for the evaluation of *fib*(n)

Because a transformer always returns the same sort, it is type-safe. A transformer is used to transform a tree.

Accumulator: a mapping of all node types to a single type. It will traverse its first argument, while the second argument keeps the accumulated value. An accumulator is declared as follows:

$$f(S_1, S_2, ..., S_n) \rightarrow S_2\{\texttt{traversal(accu, ...)}\}$$

After each application of an accumulator, the accumulated argument is updated. The next application of the accumulator, possibly somewhere else in the term, will use the *new* value of the accumulated argument. In other words, the accumulator acts as a global, modifiable, state during the traversal.

An accumulator function never changes the tree, only its accumulated argument. Furthermore, the type of the second argument has to be equal to the result type. The end-result of an accumulator is the value of the accumulated argument. By these restrictions, an accumulator is also type-safe for every instantiation.

An accumulator is meant to be used to extract information from a tree.

Accumulating transformer: a sort preserving transformation that accumulates information while traversing its first argument. The second argument maintains the accumulated value. The return value of an accumulating transformer is a tuple consisting of the transformed first argument and accumulated value. An accumulating transformer is declared as follows:

```
module Nats

imports basic/Whitespace

exports
  context-free start-symbols Nat-con
  sorts Nat-con

  lexical syntax
    [0-9]+ -> Nat-con

hiddens
  variables
    "Char+"[0-9]* -> CHAR+

equations

  [1] nat-con(Char+) = nat-con(Char+ "a")
```

Figure 41: Illegal use of lexical constructor functions

$$f(S_1, S_2, ..., S_n) \rightarrow < S_1, S_2 > \{\texttt{traversal(accu, trafo, ...)}\}$$

An accumulating transformer is used to simultaneously extract information from a tree and transform it.

Having these three types of traversals, they must be provided with visiting strategies. Visiting strategies determine the order of traversal and the "depth" of the traversal. We provide the following two strategies for each type of traversal:

Bottom-up: the traversal visits *all* the subtrees of a node where the visiting function applies in an *bottom-up* fashion. The annotation `bottom-up` selects this behavior. A traversal function without an explicit indication of a visiting strategy also uses the bottom-up strategy.

Top-down: the traversal visits the subtrees of a node in an top-down fashion and stops recurring at the first node where the visiting function applies and does not visit the subtrees of that node. The annotation `top-down` selects this behavior.

Beside the three types of traversals and the order of visiting, we can also influence whether we want to stop or continue at the matching occurrences:

Break: the traversal stops at matching occurrences.

Continue: the traversal continues at matching occurrences.

The visiting strategies in combination with the continuation strategies is visualized in the "traversal cube", see Figure 44. The current implementation of the traversal mechanism only supports the left-to-right visiting strategy.

We give two simple examples of traversal functions that are both based on the tree language defined in Figure 45 that describes binary prefix expressions with natural numbers as leaves. Examples are `f(0,1)` and `f(g(1,2), h(3,4))`.

Our first example (Figure 46) transforms a given tree into a new tree in which all numbers have been incremented.

```
module Types

imports basic/Booleans

exports
  context-free start-symbols Type
  sorts Type

  context-free syntax
    "natural"     -> Type
    "string"      -> Type
    "nil-type"    -> Type
    compatible(Type, Type) -> Boolean

hiddens
  variables
    "Type"[0-9]*  -> Type

equations

  [Type-1]  compatible(natural, natural) = true

  [Type-2]  compatible(string, string) = true

  [default-Type] compatible(Type1,Type2) = false
```

Figure 42: Using a default equation

```
module Fib

imports basic/Whitespace

exports
  context-free start-symbols Int
  sorts Int

  context-free syntax
    "0"             -> Int
    "s" "(" Int ")" -> Int

  context-free syntax
    add(Int, Int) -> Int

    fib(Int)      -> Int {memo}

hiddens
  variables
    [xy][0-9]* -> Int

equations

  [add-s] add(s(x), y) = s(add(x, y))
  [add-z] add(0, y) = y

  [fib-z] fib(0) = s(0)
  [fib-o] fib(s(0)) = s(0)
  [fib-x] fib(s(s(x))) = add(fib(s(x)), fib(x))
```

Figure 43: Using the memo attribute when defining Fibonacci

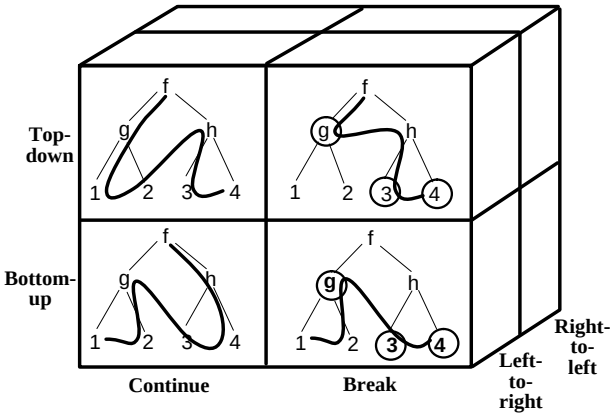

Figure 44: The “traversal cube”: principal ways of traversing a tree

```
module Tree-syntax

imports basic/Integer
imports basic/Whitespace

exports
  context-free start-symbols Tree
  sorts Tree

  context-free syntax
    Integer       -> Tree
    f(Tree, Tree) -> Tree
    g(Tree, Tree) -> Tree
    h(Tree, Tree) -> Tree
```

Figure 45: A simple tree language

```
%% Tree-inc.sdf
module Tree-inc
imports Tree-syntax

exports
  context-free syntax
    inc(Tree) -> Tree {traversal(trafo, top-down, continue)}

hiddens
  variables
    "N"[0-9]* -> Integer

%% Tree-inc.asf
equations
  [1] inc(N) = N + 1
```

Figure 46: The transformer `inc` increments all numbers in a tree

```
%% Tree-sum.sdf
module Tree-sum
imports Tree-syntax
exports
  context-free syntax
    sum(Tree, Integer) -> Integer {traversal(accu, top-down, continue)}

hiddens
  variables
    "N"[0-9]* -> Integer

%% Tree-inc.asf
equations
  [1] sum(N1, N2) = N1 + N2
```

Figure 47: The accumulator `sum` that sums all numbers in a tree.

Our second example (Figure 47) computes the sum of all numbers in a tree. For more examples and a detailed description of traversal functions see [11].

The SDF definition of a traversal function has to fulfill a number of requirements:

- Traversal functions can only be defined in the context-free syntax section.
- Traversal functions must be prefix functions, see Section 2.10.2.
- The first argument of the prefix function is always a sort of a node of the tree that is traversed, for both accumulating as well as transformation functions.
- In case of a transformation function the result sort should always be same as the sort of the first argument.

  ```
  tf(Tree, A_1, ..., A_n) -> Tree {traversal(trafo, ...)}
  ```

- In case of an accumulating function, the second argument represents the accumulator and the result sort should be of the same sort.

  ```
  tf(Tree, Accu, A_1, ..., A_n) -> Accu {traversal(accu, ...)}
  ```

- In case of an accumulating transformation function, the first argument represents the tree node, the second the accumulator, and the result sort should be a tuple consisting of the tree node sort (first element of the tuple) and the accumulator (second element of the tuple).

  ```
  tf(Tree, Accu, A_1, ..., A_n) -> <Tree, Accu> {traversal(accu, trafo, ...)}
  ```

- The traversal functions may have more arguments, the only restriction is that they should be consistent over the various occurrences of the same traversal function.

  ```
  tf(Tree1, Accu, A_1, A_2, ..., A_n) -> Tree1 {traversal(trafo, continue, top-down)}
  tf(Tree2, Accu, A_2, A_1, ..., A_n) -> Tree2 {traversal(trafo, continue, top-down)}
  ```

- The order of the traversal attributes is free, but should be used consistently, for instance the following definition is not allowed.

  ```
  tf(Tree1, Accu, A_1, ..., A_n) -> Tree1 {traversal(trafo, top-down, continue)}
  tf(Tree2, Accu, A_1, ..., A_n) -> Tree2 {traversal(trafo, continue, top-down)}
  ```

```
module ItemSet

imports basic/Whitespace

exports
  context-free start-symbols Set
  sorts Item Set

  lexical syntax
    [a-z]+ -> Item

  context-free syntax
    Set[Item] -> Set

hiddens
  variables
    "i"[0-9]* -> Item
    "l"[0-9]* -> {Item ","}*

equations

  [1] {l1, i, l2, i, l3}   = {l1, i, l2, l3}
  [2] {l1, i1, l2, i2, l3} = {l1, i2, l2, i1, l3}
```

Figure 48: Non-executable specification for sets

- If the number of arguments of the traversal function changes, you should introduce a new function name. The following definitions are not correct:

  ```
  tf(Tree1, Accu, A_1, A_2) -> Tree1 {traversal(trafo, top-down, continue)}
  tf(Tree2, Accu, A_1, A_2, A_3) -> Tree2 {traversal(trafo, continue, top-down)}
  ```

  but should be:

  ```
  tf1(Tree1, Accu, A_1, A_2) -> Tree1 {traversal(trafo, top-down, continue)}
  tf2(Tree2, Accu, A_1, A_2, A_3) -> Tree2 {traversal(trafo, continue, top-down)}
  ```

In the SDF part of a module it is needed to define traversal functions for all sorts which are needed in the equations.

2.18.9 Which Specifications are Executable?

Which ASF+SDF specifications can be executed? The specification of sets in Figure 48 illustrates a non-executable specification, since equation `[2]`, which expresses that two elements in a set may be exchanged, will lead to an infinite rewriting loop.

2.18.10 Common Errors when Executing Specifications

- When using the inequality operator `!=` in a condition, no new variables may be introduced in either side of the inequality.
- If the normal form of a term still contains function symbols that should have been removed during rewriting, you probably have forgotten one or more equations that define the function. A typical situation is that you have given an *incomplete* set of equations defining the function.

```
module Eval

imports SimpleExpr
imports basic/Integers

exports
  context-free syntax
    eval(E) -> Integer

hiddens
  variables
    "E"[0-9]* -> E
    "N"[0-9]* -> NatCon

equations
 [e1] eval(E1 + E2) = eval(E1) + eval(E2)
 [e2] eval(E1 * E2) = eval(E1) * eval(E2)
 [e3] eval(N) = N
```

Figure 49: ASF+SDF specification of a very simple `eval` function

- The rewriting process does not stop. Your equations probably contain an infinite loop.
- Be careful when a condition contains both instantiated and uninstantiated variables.

2.19 Tests

3 Examples of ASF+SDF Specifications

Here are some examples of ASF+SDF specifications, which are selected to illustrate specific features of the formalism. Larger examples can be found in the online specifications.

3.1 A simple evaluation function

Suppose we want to define a very simple evaluation function for the expressions as defined in Figure 22. Figure **??** shows the specification.

3.2 Symbolic Differentiation

Computing the derivative of an expression with respect to some variable is a classical problem. Computing the derivative of `X * (X + Y + Z)` with respect to `X gives:`

```
d(X * (X + Y + Z)) / dX = X + Y + Z + X
```

Differentiation is defined in several stages in the specification in Figure 50. First, the sorts `Nat` (natural numbers), `Var` (variables), and `Exp` (expressions) are introduced. Next, a differentiation operator of the form `d` E `/ d` V is defined. Then, the differentiation rules are defined (equations [1]-[5]). Finally, some rules for simplifying expressions are given. As the above example shows, further simplification rules could have been added to collect multiple occurrences of a variable (giving `2*X + Y + Z`) or to compute constant expressions.

```
module Diff

imports basic/Whitespace
imports basic/NatCon

exports
  context-free start-symbols Exp
  sorts Var Exp

  lexical syntax
    [XYZ]    -> Var

  context-free syntax
    NatCon              -> Exp
    Var                 -> Exp
    Exp "+" Exp         -> Exp {left}
    Exp "*" Exp         -> Exp {left}
    "(" Exp ")"         -> Exp {bracket}
    "d" Exp "/" "d" Var -> Exp

 context-free priorities
   Exp "*" Exp -> Exp > Exp "+" Exp -> Exp

hiddens
  variables
    "N"       -> NatCon
    "V"[0-9]* -> Var
    "E"[0-9]* -> Exp

equations
  [ 1] dN/dV = 0            [ 2] dV/dV = 1
  [ 3] V1 != V2 ==> dV1/dV2 = 0
  [ 4] d(E1+E2)/dV = dE1/dV + dE2/dV
  [ 5] d(E1*E2)/dV = dE1/dV * E2 + E1 * dE2/dV
  [ 6] E + 0 = E            [ 7] 0 + E = E
  [ 8] E * 1 = E            [ 9] 1 * E = E
  [10] 0 * E = 0            [11] E * 0 = 0
```

Figure 50: ASF+SDF specification for differentiation

```
module Flag

imports basic/Whitespace

exports
  context-free start-symbols Flag
  sorts Color Flag

  context-free syntax
    "red"          -> Color
    "white"        -> Color
    "blue"         -> Color
    "{" Color+ "}" -> Flag

hiddens
  variables
    "Cs"[0-9]* -> Color*
    "C"[0-9]*  -> Color

equations

  [1] {Cs1 white red Cs2}  = {Cs1 red white Cs2}

  [2] {Cs1 blue white Cs2} = {Cs1 white blue Cs2}

  [3] {Cs1 blue red Cs2}   = {Cs1 red blue Cs2}
```

Figure 51: ASF+SDF specification for sorting

3.3 Sorting

The use of list structures is illustrated by the specification of the *Dutch National Flag* problem presented in Figure 51: given an arbitrary list of the colours red, white and blue, sort them in the order as they appear in the Dutch National Flag. We want:

{`white blue red blue red white red`} $\Rightarrow$
{`red red red white white blue blue`}

In this specification, the list variables `Cs1` and `Cs2` permit a succinct formulation of the search for adjacent colours that are in the wrong order.

3.4 Code Generation

Consider a simple statement language (with assignment, if-statement and while-statement) and suppose we want to compile this language to the following stack machine code:

```
module BasicNotions
exports
  context-free start-symbols Nat Id
  sorts Nat Id

  lexical syntax
    [0-9]+         -> Nat
    [a-z][a-z0-9]* -> Id
```

Figure 52: ASF+SDF specification for BasicNotions

`push` N	Push the number N
`rvalue` I	Push the contents of data location I
`lvalue` I	Push the address of data location I
`pop`	Remove the top of the stack
`copy`	Push a copy of the top value on the stack
`assign`	The r-value on top of the stack is stored in the l-value below it and both are popped
`add, sub, mul`	Replace the two values on top of the stack by their sum (difference, product)
`label` L	Place a label (target of jumps)
`goto` L	Next instruction is taken from statement following label L
`gotrue` L	Pop the top value; jump if it is nonzero
`gofalse` L	Pop the top value; jump if it is zero

The statement:

```
while a do a := a - 1; b := a * c od
```

will now be translated to the following instruction sequence:

```
label xx ;
rvalue a ;
gofalse xxx ;
lvalue a ;
rvalue a ;
push 1 ;
sub ;
assign ;
lvalue b ;
rvalue a ;
rvalue c ;
mul ;
assign ;
goto xx ;
label xxx
```

Basic notions The specification in Figure 52 defines the sorts `Nat` (numbers) and `Id` (identifiers).

Expressions and Statements Given these basic notions the expressions, see Figure 53, and statements, see Figure 54, of our little source language are defined.

Assembly language The instructions of the assembly language for the stack machine are defined in Figure 55.

```
module Expressions

imports BasicNotions

exports
  context-free start-symbols Exp
  sorts Exp

  context-free syntax
    Nat         -> Exp
    Id          -> Exp
    Exp "+" Exp -> Exp {left}
    Exp "-" Exp -> Exp {left}
    Exp "*" Exp -> Exp {left}

  context-free priorities
    Exp "*" Exp -> Exp > {left: Exp "+" Exp -> Exp
                                Exp "-" Exp -> Exp}
```

Figure 53: ASF+SDF specification for Expressions

Label generation Next, we define a function to construct a next label given the previous one in Figure 56. It is defined on the lexical notion of labels (`Label`). The scheme of appending the character '`x`' to the previous label is, of course, naive and will in real life be replaced by a more sophisticated one.

Codegenerator It remains to define a function '`tr`' that translates statements into instructions. During code generation we should generate new label names for the translation of if- and while-statements. This is an instance of a frequently occurring problem: how do we maintain global information (in this case: the last label name generated)? A standard solution is to introduce an auxiliary sort (`Instrs-Lab`) that contains both the generated instruction sequence and the last label name generated so far. The SDF part (Figure 57) and the ASF part (Figure 58) of module `CodeGenerator` define the actual translation function.

This completes the specification of our code generator.

3.5 Large ASF+SDF Specifications

There are a quite a few very large ASF+SDF specifications around:

- The ASF+SDF2C compiler.
- A part of the parse table generator for SDF.
- The SDF checker.
- The syntax and type checking of a domain specific language for describing financial products.
- A compiler from UML diagrams to various target languages (Progress, Java, DB2).
- Transformation system for improving Cobol programs.
- A system for Java refactoring.
- Tooling for Action Semantics.
- Tooling for CASL.
- Tooling for ELAN.

```
module Statements

imports Expressions

exports
  context-free start-symbols Stats
  sorts Stat Stats

  context-free syntax
    Id ":=" Exp                 -> Stat
    "if" Exp "then" Stats "fi"  -> Stat
    "while" Exp "do" Stats "od" -> Stat
    {Stat ";"}+                 -> Stats
```

Figure 54: ASF+SDF specification for Statements

4 Well-formedness checks on SDF

In order to improve the quality of the written specifications, a number of checks are performed before an SDF specification is transformed into a parse table. The checks are performed on two levels: the first level are SDF specific check, the second level are the ASF+SDF specific check.

There are various categories of messages in the ASF+SDF Meta-Environment.

1. Parse errors.
2. SDF type check warnings.
3. SDF type check errors.
4. ASF type check errors.

We will briefly discuss each of the error messages and indicate what is exactly wrong in the specification. Furthermore we will hint at how the error can be fixed.

4.1 Parse Errors

There are three different types of parse errors:

1. A syntax error, which is reported by pinpointing the exact location in the file and the message `Parse error near cursor` in case of an editor or in the message pane an error message similar to `Parse error: character '<c>' unexpected`. This means that the parser detected a syntax error in the text to be parsed and can not proceed its parsing process. Clicking on the error in the `Errors`-pane moves the cursor to the exact error location and launches if needed the editor. A variant of the syntax error message is: `Parse error: eof unexpected`.
2. A cycle, in case of an editor the cursor is positioned at the position where the first cycle is detected in the input and the message is `Cycle: <list_of_production_rules>` is printed. A cycle is reported whenever the parser detects a non terminating chain of reductions. All production rules on the cycle are shown as `<list_of_production_rules>`.
3. An ambiguity, again in case of an editor the cursor is positioned at the position where the first ambiguity is detected in the input and the message `Ambiguity: <list_of_production_rules>` is printed. An ambiguity is reported whenever the parser was able to recognized a (part of) the input sentence in different ways. The `<list_of_production_rules>` shows all production rules that are involved in the ambiguity.

```
module AssemblyLanguage

imports BasicNotions
exports
  context-free start-symbols Instrs
  sorts Label Instr Instrs

  lexical syntax
    [a-z0-9]+ -> Label
  context-free syntax
    "push" Nat      -> Instr
    "rvalue" Id     -> Instr
    "lvalue" Id     -> Instr
    "assign"        -> Instr
    "add"           -> Instr
    "sub"           -> Instr
    "mul"           -> Instr
    "label" Label   -> Instr
    "goto" Label    -> Instr
    "gotrue" Label  -> Instr
    "gofalse" Label -> Instr
    {Instr ";"}+    -> Instrs
```

Figure 55: ASF+SDF specification for AssemblyLanguage

```
module NextLabel

imports AssemblyLanguage

exports
  context-free syntax
    "nextlabel" "(" Label ")" -> Label

hiddens
  variables
    "Char*"[0-9]* -> CHAR*

equations

 [1] nextlabel(label(Char*)) = label(Char* "x")
```

Figure 56: ASF+SDF specification for NextLabel

```
module CodeGenerator

imports Statements AssemblyLanguage NextLabel

exports
  context-free start-symbols Instrs

  context-free syntax
    tr(Stats) -> Instrs

hiddens
  sorts Instrs-lab
  context-free syntax
    Instrs # Label -> Instrs-lab

  context-free syntax
    tr(Stats, Label) -> Instrs-lab
    tr(Exp)          -> Instrs

hiddens
  variables
    "Exp"[0-9\']*        -> Exp
    "Id"[0-9\']*         -> Id
    "Instr"[0-9\']*      -> Instr
    "Instr-list"[0-9\']* -> {Instr ";"}+
    "Label"[0-9\']*      -> Label
    "Nat"[0-9\']*        -> Nat
    "Stat"[0-9\']*       -> Stat
    "Stat+"[0-9\']*      -> {Stat ";"}+
```

Figure 57: ASF+SDF specification for CodeGenerator

```
equations

[1] <Instr-list, Label> := tr(Stat-list, x)
    ====>
    tr(Stat-list) = Instr-list

[2] <Instr-list1, Label'> := tr(Stat, Label),
    <Instr-list2, Label''> := tr(Stat-list, Label')
    ====>
    tr(Stat ; Stat-list, Label) = <Instr-list1 ; Instr-list2, Label''>

[3] Instr-list := tr(Exp)
    ====>
    tr(Id := Exp, Label) = <lvalue Id; Instr-list; assign, Label>

[4] Instr-list1 := tr(Exp),
    <Instr-list2, Label'> := tr(Stat-list, Label),
    Label'' := nextlabel(Label')
    ====>
    tr(if Exp then Stat-list fi, Label) =
    <Instr-list1; gofalse Label''; Instr-list2; label Label'', Label''>

[5] Instr-list1 := tr(Exp),
    <Instr-list2, Label'> := tr(Stat-list, Label),
    Label'' := nextlabel(Label'),
    Label''' := nextlabel(Label'')
    ====>
    tr(while Exp do Stat-list od, Label) =
    <label Label''; Instr-list1; gofalse Label'''; Instr-list2;
     goto Label''; label Label''', Label'''>

[6] Instr-list1 := tr(Exp1),
    Instr-list2 := tr(Exp2)
    ====>
    tr(Exp1 + Exp2) = Instr-list1; Instr-list2; add

[7] Instr-list1 := tr(Exp1),
    Instr-list2 := tr(Exp2)
    ====>
    tr(Exp1 - Exp2) = Instr-list1; Instr-list2; sub

[8] Instr-list1 := tr(Exp1),
    Instr-list2 := tr(Exp2)
    ====>
    tr(Exp1 * Exp2) = Instr-list1; Instr-list2; mul

[9] tr(Nat) = push Nat

[10] tr(Id) = rvalue Id
```

Figure 58: ASF+SDF specification for CodeGenerator

4.2 Type check warnings for plain SDF

The warnings and error for SDF are separated into 4 sections. First we will discuss the type check warnings and errors (see Section 4.3) for plain SDF. This variant of SDF is independent of ASF. Later we will discuss the warnings (see Section 4.4) and errors (see Section 4.5) for SDF used in combination with ASF. In this case we need to be more strict and every SDF construct is supported by ASF.

Warnings do not break the specification, but it is advisable to fix them anyway. Often they point out some not well-formed part in the specification.

- `undeclared sorts` This warning indicates that a sort is used which is not explicitly declared, or it is declared but in a hidden section.
- `double declared sort` This warning points out that the sort is already declared somewhere in this module, or in one of the imported modules.
- `double declared start-symbol` This warning indicates that the start-symbol is previously defined as start-symbol as well. This can be in the current module or in one of the imported modules.
- `illegal attribute:` {`bracket, left, right, assoc, non-assoc`} This warning is generated because the syntactic form the production rule and the attribute do not match. Given this mismatch the intended behaviour will not be effective.
- `used in priorities but undefined` This warning is generated whenever a production rule is used in a priority section which is not defined in this module or in one of the imported modules. It is possible that this production rule will be defined in one of the modules which imports this module. Normally, this indicates some typo.
- `inconsistent rhs in priorities` This warning is caused by a production rule which has not the same right-hand side as the other production rules in the priority relation. Whenever this occurs the effect of the expressed priority relation will be ignored. This check is performed modulo injections.
- `unknown constructor used in priorities` This warning indicates the use of a constructor which is not used in the corresponding set of production rules with the same right-hand side. This is a very weak check on consistent use of constructor information.
- `sort CHAR used in production rule`
- `deprecated tuple notation`
- `deprecated unquoted symbol notation`
- `deprecated non-plain sort definition`
- `aliased symbol already declared`

4.3 Type check errors for plain SDF

- `module not available`
- `start-symbols in <ModuleName> not defined in any right-hand`
- `literal in right-hand-side not allowed`
- `only sort allowed in right-hand-side of lexical-function`
- `double used label`
- `constructor has already been used` The combination of right-hand symbol and the constructor information should be unique. This warning points this out. It is advisable not to ignore this warning. In fact, for the parser these double constructors are no problem, but there are tools based on SDF for which this is problematic.

4.4 Type check warnings for ASF+SDF

- exported variables section
- kernel syntax construction
- production renamings not supported
- not supported symbol

4.5 Type check errors for ASF+SDF

- traversal attributes in non-prefix function
- illegal traversal attribute
- missing bottom-up or top-down attribute
- missing break or continue attribute
- missing trafo and/or accu attribute
- accu should return accumulated type
- trafo should return traversed type
- accutrafo should return tuple of correct types
- inconsistent arguments of traversal productions
- inconsistent traversal attributes
- asf equation sort must not be used
- charclasses not allowed in context-free syntax

4.6 Type check warnings for ASF

- Lexical probably intended to be a variable
- Deprecated condition syntax "="
- constructor not expected as outermost function symbol of left hand side

4.7 Type check errors for ASF

- equations contain ambiguities
- uninstantiated variable occurrence
- negative condition introduces variable(s)
- uninstantiated variables in both sides of condition
- uninstantiated variables in equality condition
- right-hand side of matching condition introduces variables
- matching condition does not introduce new variables
- strange condition encountered
- Left hand side is contained in a list
- no variables may be introduced in left hand side of test

5 Building stand-alone environments using ASF+SDF Meta-Environment technology

In this chapter we describe how to build simple and more complex stand-alone environments using the various components of the ASF+SDF Meta-Environment. It does not include building an environment based on the ToolBus.

First we will briefly introduce the underlying technology and architecture of the Meta-Environment. Secondly, we will describe a selection of components that are useful for building stand-alone tools. *Be aware that the information in the following pages is still very much in a state of flux.* Note that all components mentioned have a `-h` options that gives an overview of their command line options. Finally, we will describe how these components can be combined into stand-alone environments. We will describes various levels of integration. The first level is with UNIX pipes and files and the second level is invoking the various components from within C programs. The last level, which we will not discuss, is by writing ToolBus scripts.

5.1 Technology and Architecture of the ASF+SDF Meta-Environment

So far, we have explained the functionality of the ASF+SDF Meta-Environment as an interactive development environment for ASF+SDF specifications. There are, however, good reasons to have a look under the hood and understand the architecture and technologies that have been used:

- Both architecture and technologies are very innovative and it is worthwhile to learn about them.
- The ASF+SDF Meta-Environment has been constructed as a collection of cooperating components. All of these components have merits of their own and can be used independently of the ASF+SDF Meta-Environment.

If you want to reuse components of the ASF+SDF Meta-Environment or want to build variants of it, then the following information is for you.

5.1.1 Technological Background

ToolBus In many large systems control flow and actual computation are complete entangled. This leads to poor extensibility and maintenance problems. To separate coordination from computation we use the ToolBus coordination architecture [2], a programmable software bus based on process algebra. Coordination is expressed by a formal description of the cooperation protocol between components while computation is expressed in components that may be written in any language. We thus obtain interoperability of heterogeneous components in a (possibly) distributed system.

ATerms Coordination protocol and components have to share data. We use ATerms [8] for this purpose. These are trees with optional annotations on each node. The annotations are used to store tool-specific information like text coordinates or color attributes. The implementation of ATerms has two essential properties: terms are stored using maximal subterm sharing (reducing memory requirements and making deep equality tests very efficient) and they can be exchanged using a very dense binary encoding that preserves sharing. As a result very large terms (with over $1,000,000$ nodes) can be processed.

SGLR In our language-centric approach the parser is an essential tool. We use scannerless, generalized-LR parsing [35]. In this way we can parse arbitrary context-free grammars, an essential property when combining and parsing large grammars for (dialects of) real-world languages.

Term rewriting ASF+SDF specifications are executed as (conditional) rewrite rules. Both interpretation and compilation (using the ASF2C compiler [10]) of these rewrite rules are supported. The compiler generates very efficient C code that implements pattern matching and term traversal. The generated code uses ATerms as its main data representation, and ensures a minimal use of memory during normalization of terms.

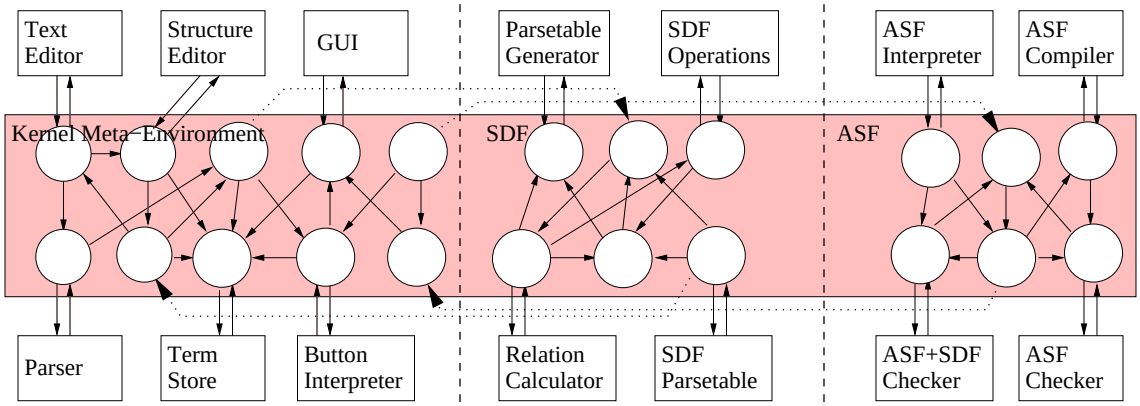

Figure 59: Architecture of the ASF+SDF Meta-Environment

5.1.2 Architecture

The architecture of the ASF+SDF Meta-Environment is shown in Figure 59. It consists of a ToolBus that interconnects the following components:

- **User interface (GUI)**: the top level user-interface of the system. It consists primarily of a graph browser for the import graph of the current specification. Furthermore, it offers facilities to visualize parse trees of selected pieces of parsed texts.
- **Text Editor**: a customized version of `XEmacs` for text editing.
- **Structure Editor**: a syntax-directed editor that closely cooperates with the Text Editor.
- **Parser**: scannerless, generalized-LR parser (SGLR) that is parametrized with a parse table.
- **Parsetable generator**: takes an SDF syntax definition as input and generates a parse table for SGLR.
- **Term Store**: stores all terms corresponding to specification modules, parse tables, user-defined terms, etc.
- **ASF Compiler**: the ASF2C compiler.
- **ASF Interpreter**: executes specifiations by direct interpretation.
- **Unparser generator**: generates prettyprinters.

5.2 Components

In the previous section we have described the overall architecture and very briefly mentioned the main components of the Meta-Environment. In order to understand how to build stand-alone environments, it is necessary to understand some of the ”flow-of-control” and ”data-flow” within the Meta-Environment. There are in fact two main scenarios:

- Parsing a term
- Reducing a term

An integrated environment as the ASF+SDF Meta-Environment takes care of a lot of administrative issues when developing your specifications. For syntax editors a fixed parse table will be used, namely a parse table derived from the SDF grammar. But when you invoke an equation or term editor and hit the parse button, the environment will generate a parse table, if necessary, before parsing the equations or term.

In order to do that the underlying syntax definition will be checked on completeness and well-formedness, and if this is the case the parse table will be generated and stored in a repository. This storing of the parse table is just for caching.

If you select the reduce button in a term editor all relevant equations have to be parsed, checked and combined into one list in order to be used by the interpreter for rewriting the term.

These parse tables and list of equations play an important role in the stand-alone setting.

5.2.1 Parse Table Generation

Before you can parse terms over a module `M` outside the ASF+SDF Meta-Environment you have to obtain a parse table for this module. There are currently three ways to achieve this:

- Interactively using the ASF+SDF Meta-Environment by selecting module `M` and pushing the `Term ParseTable...` button in the menu `Export` for the module pop-up menu. A window pops up in which you can select the directory where the parse table will be saved and in which you have to give the desired name for the table.
- From the command line using the command `pt-dump`. In the background the same actions are performed as in the interactive setting, only no user interface is activated. If you want to obtain the parse table for `basic/Booleans` you can do this via `pt-dump -m basic/Booleans -o Booleans.trm.tbl`. For further information use `pt-dump -h`.
- From the command line using the command `sdf2table`. You should be aware that `sdf2table` needs as input the *complete* grammar of `M`. This means in particular that all imports of `M` should be expanded before `sdf2table` can be used. Command line tools are available to do this, but we will not describe them here.

  For further information use `sdf2table -h`.

Before a parse table is generated the SDF definition is checked with respect to well-formedness. When using the `sdf2table` only the checks on SDF are performed, in the other two cases the checks on SDF and on ASF+SDF specific features are performed as well.

5.2.2 Obtaining Equations

In order to rewrite a term, see Section 5.4,, or compile a specification, see Section 5.5, it is necessary to obtain the equations. The generation of equations given an ASF+SDF specification can be performed in two different ways:

- Interactively using the ASF+SDF Meta-Environment by selecting module `M` and pushing the `Dump Equations...` button in the pop-up menu.
- From the command line using the command `eqs-dump`. In the background the same actions are performed as in the interactive setting, only no user interface is activated. For further information use `eqs-dump -h`.

The result is a file named `M.eqs`.

5.3 Parsing

Given a parse table `M.trm.tbl` for module `M`, terms can be parsed by using the command `sglr`:

```
sglr -m -p M.trm.tbl -i term.txt -o term.tree
```

or, alternatively, via

```
sglr -m -p M.trm.tbl < term.txt > term.tree
```

The output of `sglr` is either an error message (if the input contains a syntax error) or a parse tree in a format called AsFixMe. Note that `sglr` can generate parse trees in several formats:

- Use the flag `-2` to generate AsFix2: this is the format introduced for the latest version of SDF. This format is only in use by some tools outside the ASF+SDF Meta-Environment. AsFix2 is very verbose: even lexical tokens and layout are represented as trees. For efficiency reasons, the ASF+SDF Meta-Environment uses the more concise format AsFixMe.
- Use the flag `-m` to generate AsFix2ME: this is the preferred parse tree format that is becoming the standard inside the ASF+SDF Meta-Environment.

5.4 Rewriting a Term using the Evaluator

In order to rewrite a term `term.txt` using the ASF+SDF evaluator `asfe` two inputs are required:

- The parse tree of `term.txt`. See parsing of termsSection 5.3 how to do this. The result is `term.tree`.
- The equations to be used for rewriting. See obtaing of equations SectionR̃ef for obtaining the equations given an ASF+SDF specification.

Given this two inputs it is possible to call the interpreter and reduce terms via the command line. This can be performed in two different manners: The first is:

```
asfe -e M.eqs < term.tree > reduct.tree
```

The second is:

```
asfe -e M.eqs -i term.tree -o reduct.tree
```

Section 5.7 explains how `reduct.tree` can be converted to a textual representation.

5.5 Compiling a Specification

By means of the `Compile ...` button in the pop-up menu of the import pane the ASF+SDF2C compiler can be activated. C code is generated for the selected module (exactly *one* module can be selected for compilation) including the imported modules. The C code is immediately compiled into an executable.
Warning: The specification should be complete, i.e., no missing modules are allowed, and the equations sections should be error free.

If this requirement is satisfied, C code can be generated. This process of C code generation is described in detail in [10]. We shall give a brief description of the main steps performed by the ASF+SDF2C compiler.

In ASF+SDF there is *no* restriction in which module an equation is defined, except that all applied syntax rules should be defined. This freedom asks for a 'reshuffling' of the equations given the set of defined syntax rules. This reshuffling is needed because for each syntax rule a separate C function is generated. This C function must contain the C code for *all* equations with the C function as outermost function symbol in the left-hand side. By having all equations with the same outermost function symbol in the left-hand side together an optimal matching automaton can be generated. Equations with the same outermost function symbol are moved to the corresponding syntax rule.

5.6 Rewriting a Term using a Compiled Specification

In Section 5.5 we explained how a given module `M` can be compiled into C code. Here we describe how to compile and use this generated C code. The steps are as follows:

- Go to the directory where the ASF+SDF2C compiler has generated the C code.

- Check whether there is a `Makefile`. If this is not the case or you expect it to be invalid, generate a new one as follows:

  ```
  genmakefile -m M > Makefile
  ```

  where `M` is the name of the top module for which the C code was generated. There should also exist a file `ModuleName.module-list` in the directory with generated C code.

- Use `make` to compile the generated C code. The result is both a library `libM.a` and an executable `M`. This library can be used when several compiled specifications have to be combined into a single executable or when compiled specifications have to be combined with hand-written C code.

- After parsing the term (as explained in Section 5.3) that has to be reduced, this term can be reduced via the compiled code as follows:

  ```
  COMPILER_OUTPUT/ModuleName < term.tree > reduct.tree
  ```

  Section 5.7 explains how `reduct.tree` can be converted to a textual representation.

5.7 Unparsing a (Parsed/Normalized) Term

The unparsing of parsed/normalized terms is currently quite primitive. It is achieved as follows:

```
unparsePT < reduct.tree > reduct.txt
```

5.8 Applying a Function to a Term

In many applications it is desirable to apply a function to a given term before reducing it. A typical example is the type checking of a program: given a parse tree `T` for a program we first want to apply the typecheck function `tc` to it before reduction. In the context of term rewriting this means first constructing the term `tc(T)` and then reducing it.

The construction of this new term is achieved by the following command:

```
apply-function -f <name> -s <sort> -m <modulename> -i <in> -o <out>
```

A term is constructed consisting of an application of function `<name>` with result sort `<sort>` defined in module `<modulename>` to a term `<in>`. The resulting term is written to `<out>`. In order to actually *apply* `<name>`, the term `out` has to be reduced by `asfe` (Section 5.4).

6 Examples of Stand-alone Tools

6.1 A Stand-alone Boolean Tool

A stand-alone tool for parsing and reducing Boolean terms can be created in the following steps:

- Goto the directory `demo/pico`.
- Start the ASF+SDF Meta-Environment.
- Create a parse table for `Pico-Booleans`:
  - Right click on `Pico-Booleans` and select `New Term`.
  - Enter the text `true | false`.
  - Save this text as `term.txt`.
  - Push the `Parse` button in the `Meta-Environment` menu of the editor.

– Push the `Save` button of the `File` menu of the ASF+SDF Meta-Environment.

The parse table `Pico-Booleans.trm.tbl` has now been created.

- Dump the equations for `Pico-Booleans`:

  – Right click on `Pico-Booleans` and select `Dump Equations`.

  The equations file `Pico-Booleans.eqs` has now been created.

- Parse `term.txt`:

```
sglr -p Pico-Booleans.trm.tbl -i term.txt -o term.tree
```

  The result is the parse tree `term.tree`

- Reduce `term.tree`:

```
asfe -e Pico-Booleans.eqs  <term.tree >reduct.tree
```

- Unparse `reduct.tree`:

```
unparsePT <reduct.tree >reduct.txt
```

  The result (in textual form) of reducing `term.txt` is now `reduct.txt`

Of course, the last steps can be written more concisely in a pipeline:

```
sglr -p Pico-Booleans.trm.tbl -i term.txt | \
asfe -e Pico-Booleans.eqs | asource > reduct.txt
```

6.2 A Stand-alone Pico Typechecker

Now we show how to create a stand-alone typechecker for the Pico language. We follow the same steps as in the previous example, but there is one additional step required: given a parsed Pico program, we have to wrap the function symbol `tcp( )` around the Pico program, before we reduce the term. The steps are as follows:

- As before, generate a parse table for `Pico-syntax` (result: `Pico-syntax.trm.tbl`) and equations for `Pico-typecheck` (result: `Pico-typecheck.eqs`).

- Parse the input term `term.txt` (result: `term.tree`).

- Wrap `tcp( )` around `term.tree`:

```
apply-function -f tcp                       \
               -s PICO-BOOL                 \
               -m Pico-typecheck            \
               -i term.tree -o tcterm.tree
```

- Reduce `tcterm.tree` and unparse the result as before.

References

[1] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.

[2] J.A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.

[3] M. G. J. van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Implementation of a prototype for the new ASF+SDF meta-environment. In *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Electronic Workshops in Computing. Springer, 1997.

[4] M.G.J. van den Brand, A. van Deursen, Jan Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction 2001 (CC 2001)*, LNCS, pages 365–370. Springer, 2001.

[5] M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E.A, van den Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *LNCS*. Springer-Verlag, 1996.

[6] M.G.J. van den Brand, J. Heering, P. Klint, and P. Olivier. Compiling language definitions: The asf+sdf compiler. July 2000. See: CoRR E-print Server cs.PL/0007008.

[7] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: The asf+sdf compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[8] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.

[9] M.G.J. van den Brand and P. Klint. *ASF+SDF Meta-Environment User Manual*.

[10] M.G.J. van den Brand, P. Klint, and P. A. Olivier. Compilation and memory management for ASF+SDF. In S. Jähnichen, editor, *Compiler Construction (CC '99)*, volume 1575 of *Lecture Notes in Computer Science*, pages 198–213. Springer-Verlag, 1999.

[11] M.G.J. van den Brand, P. Klint, and J. Vinju. Term rewriting with traversal functions. To appear.

[12] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with type-safe traversal functions. In B. Gramlich and S. Lucas, editors, *Second International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, volume 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

[13] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology*, 2003. to appear.

[14] M.G.J. van den Brand, A.S. Klusener, L. Moonen, and J.J. Vinju. Generalized parsing and term rewriting - semantics directed disambiguation. In B. Bryant and J. Saraiva, editors, *Third Workshop on Language Descriptions Tools and Applications (LDTA 2003)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, 2003.

[15] M.G.J. van den Brand and C. Ringeissen. ASF+SDF parsing tools applied to ELAN. In *Third International Workshop on Rewriting Logic and Applications*, ENTCS, 2000.

[16] M.G.J. van den Brand and J. Scheerder. Development of Parsing Tools for CASL using Generic Language Technology. In D. Bert, C. Choppy, and P. Mosses, editors, *Workshop on Algebraic Development Techniques (WADT'99)*, volume 1827 of *LNCS*. Springer-Verlag, 2000.

[17] M.G.J. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In N. Horspool, editor, *Compiler Construction (CC 2002)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer-Verlag, 2002.

[18] M.G.J. van den Brand and J. Vinju. Rewriting with layout. In *Workshop on Rule-based Programming (PLI2000)*, 2000.

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

[20] H. de Jong and P. Olivier. *ATerm Library User Manual*.

[21] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[22] J. Heering. Application software, domain-specific languages, and language design assistants. May 2000. see: CoRR E-print Server cs.PL/0005002.

[23] 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.

[24] J. Heering, G. Kahn, P. Klint, and B. Lang. Generation of interactive programming environments. In *ESPRIT '85: Status Report of Continuing Work*, pages 467–477. North-Holland, 1986. Part I.

[25] J. Heering and P. Klint. Towards monolingual programming environments. *ACM Transactions on Programming Languages and Systems*, 7(2):183–213, 1985.

[26] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, March 2000. also: ACM CoRR E-print Server xxx.lanl.gov/abs/cs.PL/9911001.

[27] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990. Also in: *SIGPLAN Notices*, 24(7):179-191, 1989.

[28] M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.

[29] P. Klint. *A Guide to ToolBus Programming*. Included in ToolBus distribution.

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

[31] P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. PhD thesis, University of Amsterdam, 2000.

[32] J. Rekers. A parser generator for finitely ambiguous context-free grammars. In *Conference Proceedings of Computing Science in the Netherlands, CSN'87*, pages 69–86, Amsterdam, 1987. SION.

[33] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[34] J. Rekers and W. Koorn. Substring parsing for arbitrary context-free grammars. *SIGPLAN Notices*, 26(5):59–66, 1991.

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

Index