The Syntax Definition Formalism SDF —Reference Manual—

J. Heering,[1] P.R.H. Hendriks,[2] P. Klint[1,3] and J. Rekers[4]

December 6, 1992

Abstract

SDF is a formalism for the definition of syntax which is comparable to BNF in some respects, but has a wider scope in that it also covers the definition of lexical and abstract syntax. Its design and implementation are tailored towards the language designer who wants to develop new languages as well as implement existing ones in a highly interactive manner. It emphasizes compactness of syntax definitions by offering (a) a standard interface between lexical and context-free syntax; (b) a standard correspondence between context-free and abstract syntax; (c) powerful disambiguation and list constructs; and (d) an efficient incremental implementation which accepts arbitrary context-free syntax definitions. SDF can be combined with a variety of programming and specification languages. In this way these obtain fully general user-definable syntax.

1989 CR Categories:
D.2.1 [**Software Engineering**]: Requirements/ Specifications—*Languages*; D.2.6. [**Software Engineering**]: Programming Environments; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax, Semantics*; D.3.4 [**Programming Languages**]: Processors.
1985 Mathematics Subject Classification:
68N15 [*Software*]: Programming Languages; 68N20 [*Software*]: Compilers and generators.
Key Words & Phrases: lexical syntax, context-free syntax, concrete syntax, abstract syntax, ambiguous grammar, grammar with priorities, user-definable syntax, syntax definition formalism, scanner generation, parser generation, editor generation.
Note: Partial support received from the European Communities under ESPRIT projects 348 (Generation of Interactive Programming Environments—GIPE, and project 2177—GIPEII).

[1]Department of Software Technology, Centre for Mathematics and Computer Science P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

[2]*Current address*: Software Engineering Research Centre, Lange Viestraat 365, 3511 BK Utrecht, The Netherlands

[3]Programming Research Group, University of Amsterdam, P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

[4]*Current address*: University of Leiden, Department for Mathematics and Computer Science, P.O. Box 9512, 2300 RA Leiden, The Netherlands

Preface to the revised edition

This is a revision of the SDF manual that has originally been published as [HHKR89]. It differs from the original in the following respects:

- Two new predefined sorts have been added:
  - `IGNORE` for describing strings *inside* lexical tokens that have to be ignored (see Section 4.5); and
  - `REJECT` for forbidden combinations of adjacent lexical tokens (see Section 4.8).
- Lexical disambiguation has been changed (see Section 4.9):
  - The old Prefer Longest Match rule has been replaced by a new Prefer Longest Match per Sort rule.
  - A Prefer Variables rule has been added.
- Some new features in the context-free syntax were added:
  - Injections (see Sections 5.5)
  - Explicit naming of functions in the abstract syntax (see Section 5.6)
- In the definition of the priority ordering >>, injection functions are now taken into account (see Section 6).

1 Introduction

1.1 General design considerations

Since good notations may improve the readability of programs and other formal specifications considerably, many programming and specification languages give the user some control over the syntax that can be used. We refer the interested reader to Section 11.4 of this manual for a brief survey of the development of user-definable syntax. Although an issue of general importance, user-definable syntax becomes a sine qua non if a language is to be used for defining other languages. A language definition expressed in such a meta-language might consist of the following parts:

- a definition of the concrete (i.e., *lexical* and *context-free*) as well as abstract syntax of the language to be defined;
- a definition of its static semantics; and
- a definition of its dynamic semantics.

In the GIPE project (Generation of Interactive Programming Environments [HKKL86]) we are constructing an environment for the interactive development and implementation of languages and it is in this context that we designed SDF to handle the syntax part of language definitions.

Although the static and dynamic semantics parts of language definitions do not concern us in this manual, coupling semantics to syntax does, and this is one of the points where abstract syntax comes in. From the viewpoint of concrete syntax it is natural to describe this coupling in terms of string or text matching. A string pattern containing variables is matched to the program text. If the matching succeeds at some point in the text, the corresponding string values of the variables are transmitted to the semantic rule attached to the pattern. In practice, however, the use of string matching gives rise to ambiguities that are difficult to control. We therefore decided that semantics should be coupled to syntax by means of tree matching, which is much less susceptible to unintended ambiguities.

Now the question arises whether the parse tree or the abstract syntax tree of the program text should be used. The parse tree is the derivation showing how the text in question can be derived from the start symbol of the grammar. Its interior nodes are non-terminals of the grammar and its leaves are the lexical tokens, such as keywords, operator symbols, identifiers, etc., making up the original text. The abstract syntax tree only contains the essential information describing the text; its interior nodes are the constructors (also called operators) of the language and the direct descendants of each node are its operands. The leaves of the abstract syntax tree are identifiers, integer constants, etc. The main argument in favor of performing matching on the basis of abstract syntax trees rather than parse trees is that the latter may contain many "unnecessary" nodes that can be omitted in the abstract syntax tree.

SDF supports patterns with variables by allowing the declaration of a (possibly infinite) number of variables with user-defined names in the variables-section of a syntax definition. A text containing variables (a text pattern) corresponds to an abstract syntax tree with variables (a tree pattern or an open term). The actual matching uses the abstract syntax tree pattern. Texts with variables may also be viewed as being incomplete, the variables playing the role of gaps that have yet to be filled in.

It should be emphasized that a coupling of semantics to syntax based on abstract syntax tree matching does not preclude the use of concrete syntax in the semantics parts of a language definition. In fact, we are very much in favor of doing this. The price to be paid is a discrepancy between the text matching viewpoint, which is strongly suggested by the use of concrete syntax, and the abstract syntax tree matching viewpoint, which is the one actually intended. To minimize the gap separating both viewpoints, SDF implicitly defines a standard translation from context-free to abstract syntax, i.e., the translation from context-free to abstract syntax is the same for all SDF-definitions. This implies that SDF has less expressive power than, for instance, the syntax definition formalism Metal [KLMM83]. The abstract and context-free syntax defined by a Metal specification may differ from each other in important respects not reproducible in SDF. For example, the order of the arguments of the abstract and concrete forms of a function need not be the same in Metal. For the reason given above, we believe a uniform and predictable relationship between abstract and context-free syntax to be a distinct advantage. If required, transformation of abstract syntax trees to some "deeper" form has to be expressed in the programming or specification language in which SDF is embedded. In our case this is the algebraic specification formalism ASF [BHK89, Chapter 1], in which we express the static and dynamic semantics of languages.

In the GIPE environment a syntax-directed editor is generated from the syntax part of a language definition [Log88, DK90, Koo92] and this is a second point where abstract syntax comes in. Syntax-directed movements through the text can be performed much more efficiently on the basis of the corresponding abstract syntax tree than on the basis of the parse tree. As explained above, the latter generally contains many redundant nodes which give rise to void text movements that may be quite irritating. On the other hand, text movements on the basis of abstract syntax cannot become too unpredictable due to the fixed correspondence between abstract and context-free syntax enforced by SDF.

Although we do not discuss modular SDF-definitions in this manual, modularity has been an important design issue from the outset. In particular, putting together SDF-definitions that were developed independently from each other should be facilitated. Using SDF, the language designer should be able to borrow parts of already existing language definitions without undue difficulty. We therefore adopted the viewpoint that SDF would not only have to allow definition of all context-free languages (BNF also permits this), but that its implementation would have to support this feature without reserve and impose no additional LR- or LL-like constraints. This is an advantage in another respect as well which has nothing to do with modularization. If full context-free power is available, syntax definitions no longer have to be brought into an acceptable, but often unnatural form that obeys the restrictions imposed by the implementation. This in turn means that context-free and abstract syntax may bear a closer resemblance to each other, which is exactly what SDF requires.

An important question remains, however: what about ambiguous syntax definitions? This is a difficult problem, which is not solved by SDF but which has had implications for its design. Usually, ambiguous definitions are eliminated by applying the LR- or LL-constraints we just dismissed as too limiting for the language designer. In the general case supported by our implementation, ambiguous definitions are handled correctly in the sense that all parses of an ambiguous sentence are returned, but freedom from ambiguities can no longer be guaranteed in advance due to the fact that it is no longer decidable. Obviously, unexpected ambiguities may cause fatal run-time errors. On the other hand, allowing syntactic ambiguities leaves room for semantic disambiguation schemes, similar perhaps to some of the ones people use to disambiguate sentences in natural language.

Whereas semantic disambiguation falls outside its scope, SDF attempts to alleviate the problem of syntactic disambiguation by offering a powerful priority construct. This does not solve the problem of unexpected syntactic ambiguities, of course. It only helps the language designer to eliminate sources of ambiguity he is already aware of. Syntactic disambiguation proceeds in two phases: (1) all parse trees containing internal priority conflicts are rejected, and (2) the remaining parse trees (if any) are compared using a multiset ordering derived from the priority rules given in the SDF-definition. Unlike the first phase (which handles ordinary arithmetical expressions, for instance), the second phase allows parses that are totally different to be compared with each other. The power of the priority mechanism is such that unambiguous SDF-definitions can be given for some inherently ambiguous context-free languages, i.e., languages that do not have an unambiguous context-free grammar. This does not mean, however, that every desired disambiguation can be obtained by adding appropriate priorities to a given ambiguous SDF-definition without priorities.

The availability of a separate disambiguation construct means that disambiguation need not be expressed in terms of the syntax rules themselves. As a result, parse trees will be much smaller and closer to abstract syntax trees. In addition to the availability of full context-free syntax mentioned before, this is an important reason why the fixed correspondence between context-free and abstract syntax offered by SDF turns out to be satisfactory.

In addition to context-free and abstract syntax, lexical syntax can also be defined in SDF. Lexical disambiguation cannot be controlled by means of the priority mechanism available at the context-free level, but is primarily based on giving precedence to the longest lexemes and to literals occurring in the context-free syntax. Any remaining ambiguities are passed on to the context-free level.

In summary, SDF allows the definition of concrete and abstract syntax in a single framework. Its design and implementation are tailored towards the language designer who wants to develop new languages as well as implement existing ones in a highly interactive manner. It emphasizes compactness of syntax definitions by offering (a) a standard interface between lexical and context-free syntax; (b) a standard correspondence between context-free and abstract syntax; (c) powerful disambiguation and list constructs; and (d) an efficient incremental implementation which accepts arbitrary context-free syntax definitions. It can be combined with a variety of programming and specification languages. In this way these obtain fully general user-definable syntax.

1.2 A first example

An SDF definition consists of five sections and has the following overall structure:

`sorts:` *names of domains or non-terminals to be used in the other sections of the specification*

`lexical syntax:` *the rules of the lexical syntax*

`context-free syntax:` *the rules of the concrete and abstract syntax* `priorities:` *definition of priority relations between rules of the context-free syntax*

`variables:` *naming schemes for variables*

We introduce the most significant features of SDF by means of an example in which we define the lexical, concrete and abstract syntax of a simple programming language (see Figure 1).[1] In the `sorts` section, six names are declared. These names can be interpreted in two ways:

- as non-terminals of a lexical or a context-free grammar, and
- as names of the domains used to construct abstract syntax trees.

We will use this dual interpretation of sorts to achieve an automatic mapping between sentences and abstract syntax trees. In the lexical syntax section, we define a space, a tabulation, and a newline character as layout characters (line 3). In addition, the form of identifiers (line 4) and numeric constants (line 5) is defined.

In the `context-free syntax` section, the concrete *and* abstract syntax are defined:

- The concrete syntax is obtained by using the "non-terminal" interpretation of sorts and reading the rules from "right to left" as ordinary grammar rules.
- The abstract syntax is obtained by using the "domain" interpretation of sorts and reading the rules from left to right as definitions of (typed) constructor functions for abstract syntax trees. The sort names appearing in function definitions define the types of the arguments as well as of the result of these functions.

Some other features illustrated by the context-free syntax in Figure 1 are:

- Rules can define lists with or without separators (line 9).
- Rules may have a `bracket` attribute. Such rules are used only for grouping language constructs, but do not contribute to the abstract syntax (lines 8 and 17).
- Rules may have various attributes defining their associativity properties (lines 13–16). We allow the definition of associative and left-, right-, or non-associative operators.

In the `priorities` section, priority relations between rules in the context-free syntax are defined as well as the associativity of groups of different operators. As shown here, the operators `*` and `/` have a higher priority than the operators `+` and `-`.

Finally, in the `variables` section, naming schemes for variables are given. These variables can be used in two ways:

[1]In all examples, we will add *line numbers* to SDF definitions for ease of reference in the text. They are not a part of the SDF definition proper.

```
sorts ID NAT PROGRAM STATEMENT SERIES EXP
lexical syntax
  [ \t\n\r]                          -> LAYOUT
  [a-z] [a-z0-9]*                    -> ID
  [0-9]+                             -> NAT
context-free syntax
  program SERIES                     -> PROGRAM
  begin SERIES end                   -> SERIES {bracket}
  { STATEMENT ";" }*                 -> SERIES
  ID ":=" EXP                        -> STATEMENT
  if EXP then SERIES else SERIES     -> STATEMENT
  until EXP do SERIES                -> STATEMENT
  EXP "+" EXP                        -> EXP  {left}
  EXP "-" EXP                        -> EXP  {non-assoc}
  EXP "*" EXP                        -> EXP  {left}
  EXP "/" EXP                        -> EXP  {non-assoc}
  "(" EXP ")"                        -> EXP  {bracket}
  ID                                 -> EXP
  NAT                                -> EXP
priorities
  {left: EXP "*" EXP -> EXP, EXP "/" EXP -> EXP} >
  {left: EXP "+" EXP -> EXP, EXP "-" EXP -> EXP}
variables
  Exp             -> EXP
  Series          -> SERIES
```

Figure 1: A simple programming language.

- as variables in semantics definitions added to the SDF definition,
- as "holes" in programs during syntax-directed editing.

1.3 Organization of this manual

The chapters of this manual can be divided in three groups. Chapters 1 and 2 have an introductory character: the main issues in the design of SDF are discussed and basic concepts and notations are introduced.

The second group of chapters contains the actual description of SDF. Each of the Chapters 3, 4, 5, 6 and 7 is devoted to a detailed description of one of the five sections of an SDF definition. Chapter 8 contains a summary of these definitions.

In the last group, miscellaneous subjects are addressed. In Chapter 9, we show how semantic rules can be attached to SDF definitions. This leads to specification formalisms with considerable syntactic freedom. In Chapter 10, the techniques required to implement SDF are sketched. A discussion of the results obtained and a comparison with related work is given in Chapter 11.

Two appendices complete this manual: they present SDF definitions of SDF itself (Appendix A) and of Berkeley Pascal (Appendix B).

2 Preliminary definitions

In this Chapter we introduce the basic notions to be used in the definition of SDF: regular grammars (Section 2.1), context-free grammars (Section 2.2), and signatures (Section 2.3). In Section 2.4 follows a motivation for these definitions, and in Section 2.5 we give an overview of the SDF definition and show how the basic notions are used in it.

2.1 Regular grammars

Regular expressions over a given alphabet Σ have the usual meaning [ASU86]. With each regular expression r, we associate a language $L(r)$ of *lexemes* accepted by r:

1. The empty string ϵ is a regular expression with associated language $L(\epsilon) = \emptyset$.
2. If $a \in \Sigma$, then "a" is a regular expression with associated language $L($"a"$) = \{a\}$.
3. Suppose r and s are regular expressions with associated languages $L(r)$ and $L(s)$, then
   (a) (r)|(s) is a regular expression with the associated language $L(r) \cup L(s)$ of lexemes that are either in $L(r)$ or in $L(s)$.
   (b) $(r)(s)$ is a regular expression with the associated language $L(r)L(s)$ of lexemes consisting of a concatenation of two parts: the first part in $L(r)$ and the second part in $L(s)$.
   (c) (r)* is a regular expression with the associated language $L(r)$* of lexemes consisting of zero or more concatenations of lexemes in $L(r)$.
   (d) (r) is a regular expression with associated language $L(r)$, i.e., parentheses may be placed around a regular expression without changing its meaning.

We will extend this definition in the following ways:

- (r)+ is a regular expression with the associated language $L(r)$+ of lexemes consisting of one or more concatenations of lexemes in $L(r)$. Note that (r)+$=$ $(r)(r)$*.

- Parentheses may be omitted by using the following priorities for the operators (going from high to low priority): `*` and `+`, concatenation, and `|`. In addition to this, the concatenation and `|`-operator are left-associative.

- We will use string constants to abbreviate sequences of concatenations of single characters of the alphabet, i.e., "a_1" "a_2" ... "a_n" will be abbreviated to "$a_1a_2...a_n$".

- A lexical non-terminal of the form <N> is also a regular expression (see below).

A regular grammar is now defined as an ordered list of named regular expressions. To emphasize the fact that they are similar to non-terminals in context-free grammars, we call names of regular expressions *lexical non-terminals*. The occurrence of a lexical non-terminal in a regular expression is to be interpreted as an abbreviation for the regular expression associated with it earlier in the list. The general form of a regular grammar is thus:

<N_1> = r_1
<N_2> = r_2
...
<N_k> = r_k

where <N_i> may only occur in r_j provided that $i < j$. The language associated with this regular grammar is the union of the languages associated with each r_j. We will use lexical non-terminals to identify the lexical category of lexemes and call a (lexical non-terminal, lexeme)-pair a *lexical token*.

2.2 Context-free grammars

We will use a variant of the well-known BNF notation to define the grammars corresponding to SDF definitions. The terminals of a grammar are either literal strings, which appear in grammar rules between double quotes ("), or lexical non-terminals. Lexemes are strings over a given character alphabet and have to be defined by a separate regular grammar. Non-terminals of the grammar are written between angle brackets < and >. They may consist of arbitrary symbols except < and >. We will use the convention that lexical non-terminals have the suffix "`-LEX`".[2] A grammar rule consists of a non-terminal, the symbol `::=`, and a list of zero or more rule-elements. An empty list of rule-elements is denoted by the symbol ϵ. A rule-element is either

- a terminal or non-terminal, or

- {<N> *sep*}$\otimes$, where *sep* is an optional terminal and $\otimes$ is either `*` or `+`. This denotes a list of zero or more (`*`) or one or more (`+`) repetitions of the non-terminal separated by *sep*. When *sep* is not present, the brackets should be omitted and the rule-element is abbreviated to <N>$\otimes$.

[2]In Chapter 7 we will also use lexical non-terminals with the suffix "`-VAR`"; they denote the contributions of variable declarations to the lexical grammar.

We will use the symbol `|` to abbreviate rules with the same right-hand side, i.e. `<`N`> ::=` Rhs_1 `|` Rhs_2, is equivalent to

> `<`N`> ::=` Rhs_1
> `<`N`> ::=` Rhs_2.

Each non-terminal in the grammar acts as a start symbol.

A *parse tree* for a combination of a given regular grammar and a given BNF grammar is an ordered labeled tree. Leaves of the tree are either literals of the grammar or characters of lexemes. Nodes are either labeled with a lexical non-terminal, a non-terminal, or a list. This is described in more detail in the following definition.

An ordered labeled tree P is a *parse tree* for a given regular grammar and BNF grammar if

- the root of P is labeled with L, where L is a lexical non-terminal, all its children are labeled with characters from the lexical alphabet, and the characters together form a lexeme accepted by the regular expression associated with L in the regular grammar, or
- the root of P is labeled with `<`N`>`, has children $C_1, ..., C_n (n \geq 0)$ whose roots are labeled with $X_1, ..., X_n$, the rule `<`N`>::=` $X_1 \ldots X_n$ is a rule of the grammar, and all C_i 's are either parse trees or literals, or
- the root of P is labeled with $\{$`<`N`>` $sep\}\otimes$, has children $C_1, ..., C_n$ ($n \geq 0$ if $\otimes =$`*`, $n \geq 1$ if $\otimes =$`+`) and either
  - *sep* is empty and all n children are parse trees labeled with `<`N`>` or
  - *sep* is not empty, n is either zero or odd, all odd-numbered children are parse trees whose root is labeled with `<`N`>`, while all even-numbered children are equal to the literal *sep*.

Consider, for instance, the following BNF grammar

```
<DECL> ::= "decl" { <ID> "," }+ ":" <TYPE>
<TYPE> ::= "integer"
<TYPE> ::= "real"
<ID>   ::= <ID-LEX>
```

and the regular grammar

```
<LETTER-LEX> = "a" | ... | "z"
<ID-LEX> = <LETTER-LEX>+
```

The literals of the BNF grammar are : `"decl"`, `","`, `":"`, `"integer"`, and `"real"`. `<ID-LEX>` is the only lexical token. The non-terminals are `<DECL>`, `<ID>` and `<TYPE>`. The parse tree for the sentence `decl ab, c, xyz : real` is shown in Figure 2. Note that the lexeme for each `<ID-LEX>` is represented by a sequence of characters. Any structure appearing in the rules defining `<ID-LEX>` is lost in the parse tree.

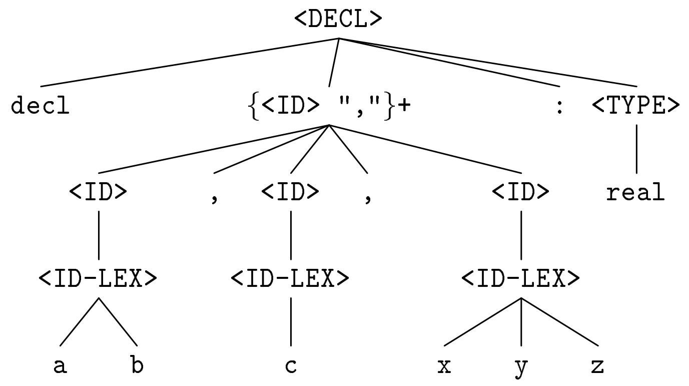


Figure 2: Parse tree for the sentence: `decl ab, c, xyz : real`.

In the sequel, we will add all literals occurring in the BNF grammar to the regular grammar and will identify an occurrence of the literal with an occurrence of the corresponding lexical non-terminal (see Section 5.2).

2.3 Signatures

The abstract syntax corresponding to an SDF definition will be described by a *signature*, which defines a set of *abstract syntax trees* (also known as *terms*). It consists of three parts:

- Declarations of sorts defining the basic domains of trees. For each declared sort S (*basic sort*), the sorts $\{S\ \mathit{sep}\}$`*` and $\{S\ \mathit{sep}\}$`+`, with *sep* an arbitrary identifier or empty, are implicitly defined (*list sorts*)[3]. They denote lists of, respectively, zero or more and one or more elements of sort S. The symbol *sep* is part of the list constructor and lists with elements of the same sort but different separators are distinct. Like before, we will use the abbreviation $\otimes$ to denote either `*` or `+`.

- A strict (i.e., non-reflexive) *partial order* on sorts defining inclusion relations between sorts (*subsorts*). This partial order is obtained by taking the transitive closure of the subsort declarations in the signature. In addition to this, the following subsort relations hold automatically:
  - $\{S\ \mathit{sep}\}\texttt{+} < \{S\ \mathit{sep}\}\texttt{*}$ for all basic sorts S;
  - $\{S\ \mathit{sep}\}\otimes < \{S'\ \mathit{sep}\}\otimes$ for all basic sorts S and S' with $S < S'$. Note that the relations $S < S\texttt{+}$ or $S < S\texttt{*}$ do *not* hold automatically!

- Declarations of *functions* defining nodes in the tree by giving for each function a name, the number and sort of its children (arguments), and

[3]An occurrence of $\{S\ \mathit{sep}\}$`*` or $\{S\ \mathit{sep}\}$`+` thus implicitly declares *sep* as a (varyadic) constructor function for lists with elements of sort S.

its result sort. The result sort of a function must always be a basic sort. Only the argument sorts may be list sorts. Functions without arguments are also called *constants*. The attribute `assoc` denotes an associative function and may be attached to binary functions with identical argument and result sorts.

We will adopt the convention that names of sorts, functions and separators in signatures either consist solely of alphanumeric characters and hyphens (e.g., `a-sort-name`) or, if not, are surrounded by double quotes (e.g., `"/"`).

Signatures do not contain a separate section for the declaration of variables. Instead, variables are implicitly defined and they are denoted by a sort, followed by a hyphen, followed by a (possibly quoted) identifier (e.g., `NAT-x` denotes a variable of sort `NAT`, and `{ID ","}+-Ids` denotes a variable of sort `{ID ","}+`). Note that variables may range both over basic sorts as well as over list sorts.

Trees can be constructed by combining functions and variables in ways compatible with their definition, i.e., each child of each node in the tree should be of a sort that is identical to or smaller than the one appearing in the declaration of the function corresponding to the node. There is one exception to this general rule: lists may contain list variables of the same or a smaller list sort with the same separators.

An *abstract syntax tree* (or *term*) t for a given signature is defined inductively as follows:

- If $t = c$, where c is a constant of basic sort s, i.e., a function without arguments with result sort s, then t is a term of basic sort s.

- If $t = v$, where v is a variable of (basic or list) sort s, then t is a term of sort s.

- If $t = f(t_1, ..., t_n)$, where f is a function with arguments of sorts $s_1, ..., s_n$, $(n \geq 1)$ and the basic sort s as result sort, and $t_1, ..., t_n$, are terms of sort $s'_1, ..., s'_n$ with $s'_i \leq s_i$, then t is a term of basic sort s.

- If $t = s[sep : t_1, ..., t_n]$, where s is a basic sort, sep is a separator or empty, $n \geq 0$, and t_i is either

  - a term of basic sort s'_i, with $s'_i \leq s$, or
  - a variable of list sort $\{s'_i \;\; sep\}\otimes$, with $s'_i \leq s$,

  then

  - t is a term of list sort $\{s \;\; sep\}$`*` if $n = 0$ or all t_i are variables of sort $\{s_i \;\; sep\}$`*` $(s_i \leq s)$,
  - otherwise, t is a term of list sort $\{s \;\; sep\}$`+`.

Note that this definition does *not* always associate the smallest possible sort with each term.

Consider, for example, the following signature:

```
sorts ID TYPE DECL NAT EXP ARG-LIST
subsorts ID < EXP  NAT < EXP  {EXP ","}* < ARG-LIST
functions
  a       :                   -> ID
  b       :                   -> ID
  integer :                   -> TYPE
  real    :                   -> TYPE
  decl    : TYPE # {ID ";"}+ -> DECL
  0       :                   -> NAT
  succ    : NAT               -> NAT
  "*"     : EXP # EXP         -> EXP
  "/"     : EXP # EXP         -> EXP
  call    : ID # ARG-LIST     -> EXP
```

The basic sorts in this signature are `ID`, `TYPE`, `DECL`, `NAT`, `EXP` and `ARG-LIST`. The partial order on sorts is defined such that both `ID` and `NAT` are included in `EXP` and that `{EXP ","}*` is included in `ARG-LIST`. Next, the functions `a`, `b`, `integer`, `real`, `decl`, `0`, `succ`, `"*"`, `"/"`, and `call` are defined with the type of their arguments and that of their result. Some examples of abstract syntax trees over this signature are shown graphically in Figures 3 and 4. In prefix notation, the trees in these figures can be written as

(*a*) `0`
(*b*) `succ(0)`
(*c*) `"*"(succ(0), "/"(0,b))`
(*d*) `decl(real,ID[";":a,b])`
(*e*) `decl(real,ID[";":a,b,b,a,b,a])`
(*f*) `call(a,EXP[",":])`

respectively as

(*a′*) `NAT-n`
(*b′*) `succ(NAT-n)`
(*c′*) `"*"(succ(NAT-n), "/"(0,ID-id))`
(*d′*) `decl(TYPE-t, {ID ";"}+-x)`
(*e′*) `decl(real, ID[";":a,{ID ";"}+-x,a,ID-id,a])`
(*f′*) `call(a, EXP[",":  {EXP ","}*-e, {ID ","}+-y])`

The following variables appear in these examples:

Variable	**Sort**
`NAT-n`	`NAT`
`ID-id`	`ID`
`TYPE-t`	`TYPE`
`{ID ";"}+-x`	`{ID ";"}+`
`{ID ","}+-y`	`{ID ","}+`
`{EXP ","}+-e`	`{EXP ","}+`

Observe that the list sort `{ID ","}+` does not appear explicitly in the signature given in the example, but that the use of variables of this sort is permitted.

We also need the notion of *selective substitution* of variables in abstract syntax trees. A subset of the occurrences of a variable may be replaced by a tree of a sort that is identical to or smaller than the sort of the variable. Since a variable may occur more than once in an abstract tree, we assume that the occurrences that are to be affected by the substitution are explicitly *marked*.

List variables occurring as elements of lists are treated in a special way during substitution: when replacing such variables by a list of elements, those elements are inserted directly as elements of the original list, instead of producing a list with a nested structure.

For a given term t containing marked occurrences of a variable x of sort s, and a term t' of sort $s' \leq s$, we define the *selective substitution* of t' for x in t (denoted by $t[x := t']$) as follows:

- If $t = c$, then $t[x := t'] = c$.
- If $t = x$ with x marked, then $t[x := t'] = t'$.
- If $t = f(t_1, ..., t_n)$ $(n \geq 1)$, then $t[x := t'] = f(t_1[x := t'], ..., t_n[x := t'])$.
- If $t = s[sep : t_1, ..., t_n]$ $(n \geq 0)$, define substitution by induction on the length of the list:
  - If $t = s[sep :]$ (the empty list), define $t[x := t'] = s[sep :]$.
  - The result of the substitution for $s[sep : t_2, ..., t_n]$ is always a list of the form $s[sep : u_1, ..., u_m]$ $(m \geq 0)$. Now distinguish two cases:
    - The result of the substitution $t_1[x := t']$ is a list. In that case t_1 is a marked variable of sort $\{s_1 \;\; sep\}\otimes$ with $s_1 \leq s$, and $t_1[x := t'] = s_1[sep : e_1, ..., e_k]$ $(k \geq 0)$ and define $t[x := t'] = s[sep : e_1, ..., e_k, u_1, ..., u_m]$.
    - Otherwise, the result of the substitution $t_1[x := t']$ is a term of sort s' $(s' < s)$, define $t[x := t'] = s[sep : t_1[x := t'], u_1, ..., u_m]$.
- Otherwise, $t[x := t'] = t$.

The trees in (*a*–*f*) in Figures 3 and 4 can be obtained from the corresponding trees in (a'–f') by the following variable substitutions (assuming that *all* occurrences of variables are marked):

(a) `NAT-n := 0`
(b) `NAT-n := 0`
(c) `NAT-n := 0`, `ID-id := b`
(d) `TYPE-t := real`, `{ID ";"}+-x := ID[";":a,b]`
(e) `{ID ";"}+-x := ID[";":b,b]`, `ID-id := b`
(f) `{EXP ","}*-e := EXP[",":]`, `{ID ","}*-y := ID[",":]`.

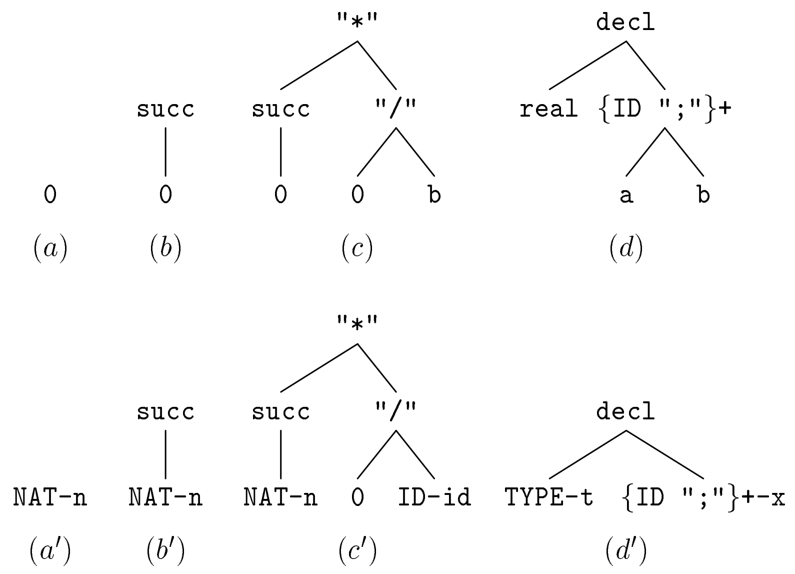


Figure 3: Examples of abstract syntax trees without (a–d) and with (a'–d') variables.

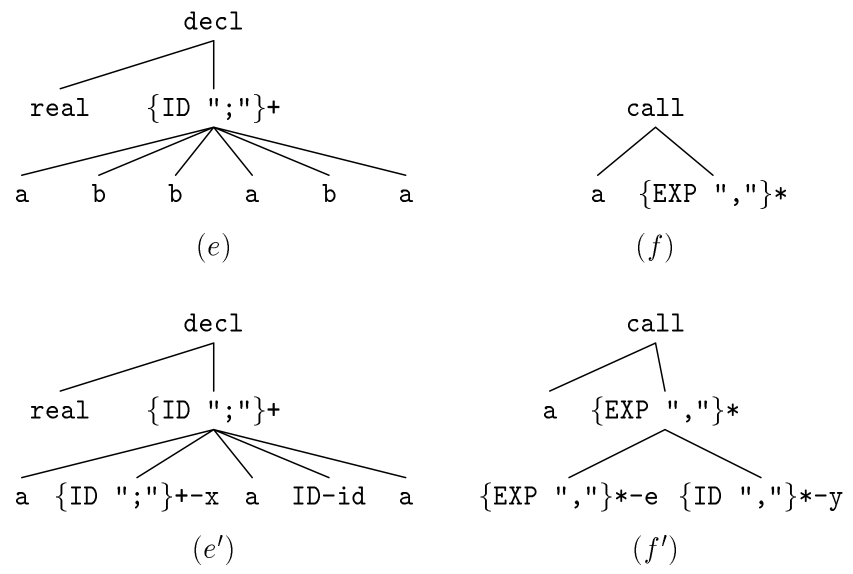


Figure 4: Examples of abstract syntax trees without (e,f) and with (e',f') variables.

2.4 Rationale for the definitions of context-free grammar and signature

The definitions of the notions introduced in the previous sections differ, in some respects, from the usual ones. Here, we give the arguments for the particular choices we made.

2.4.1 BNF grammars

- All non-terminals act as start symbol. This is desirable from the perspective of syntax-directed editing, where the need arises to parse only parts of programs.
- The terminal alphabet consists of literal strings as well as of lexical non-terminals.
- Because our notion of BNF grammars directly support list constructs with separators, we can define parse trees in such a way that all list elements are treated symmetrically. An alternative approach would have been to use left- or right-recursive grammar rules, leading to a correspondingly biased representation of lists. This asymmetrical representation would then have to be translated to a symmetrical representation in the abstract syntax tree, in order to avoid the asymmetry becoming visible during the syntax-directed editing of list elements.

2.4.2 Signatures

- For each basic sort, we implicitly define different list sorts for all possible separators (constructor functions). In this way, lists with elements of equal type but different separators can be distinguished.
- Subsorts serve the purpose of eliminating chain rules of the form `<`N`>` `::=` `<`M`>` in the BNF grammar. These would otherwise give rise to many void text movements in the syntax-directed editor.
- There is an interaction between subsorts and lists. The relation $\{S\ sep\}$`+` $<$ $\{S\ sep\}$`*` holds automatically for all sorts and separators. The inclusion $S <\{S\ sep\}$`+` does not hold automatically, however, since—from the perspective of syntax-directed editing—the distinction between a single element and a list consisting of one element would disappear (see, for instance, Figure 24 in Chapter 5).
- We allow the declaration of associative functions in the signature (by means of the attribute `assoc`). At the syntactic level, associative functions may cause ambiguities which we resolve by always choosing a left-associative parse from all possible (associatively equivalent) parses (see Section 6.2). Declaring a function in the signature to be associative allows the reconstruction of all these variants using the associative law.

- Variables are not explicitly declared, but we use a general naming scheme for generating infinitely many variables. We need this generality to describe the naming schemes for variables in SDF variable declarations.
- The naming convention for sorts and functions is more general than normally found: we allow names either consisting of alphanumeric characters and hyphens, or containing arbitrary characters but surrounded by double quotes. This allows us great flexibility when generating names for sorts and functions.

Finally, some comments on substitution are in order:

- We have chosen to define substitution for selected occurrences of a variable and not for all occurrences. Clearly, the latter can easily be defined by first marking all occurrences of a variable and then applying our notion of selective substitution. On the other hand, selective substitution is *the* fundamental operation used during syntax-directed editing.
- The sort of a term may become *smaller* after substitution. In particular, a list of sort $\{S\ \ sep\}$`*` may become a list of sort $\{S\ \ sep\}$`+` after substitution.
- The sorts of the elements in a list may all become smaller than the original element sort of that list as a result of substitution.

2.5 Overview of the SDF definition

An SDF definition consists of five sections, defining

- sorts,
- lexical syntax,
- context-free syntax,
- priorities, and
- variables.

As a whole, the SDF definition has the purpose to define

- A set of sentences (strings).
- A set of abstract syntax trees.
- A relation between (parse trees of) sentences and abstract syntax trees.

To this end, we can derive the following information from an SDF definition:

- A derived regular grammar and a derived BNF grammar, defining a set of sentences.

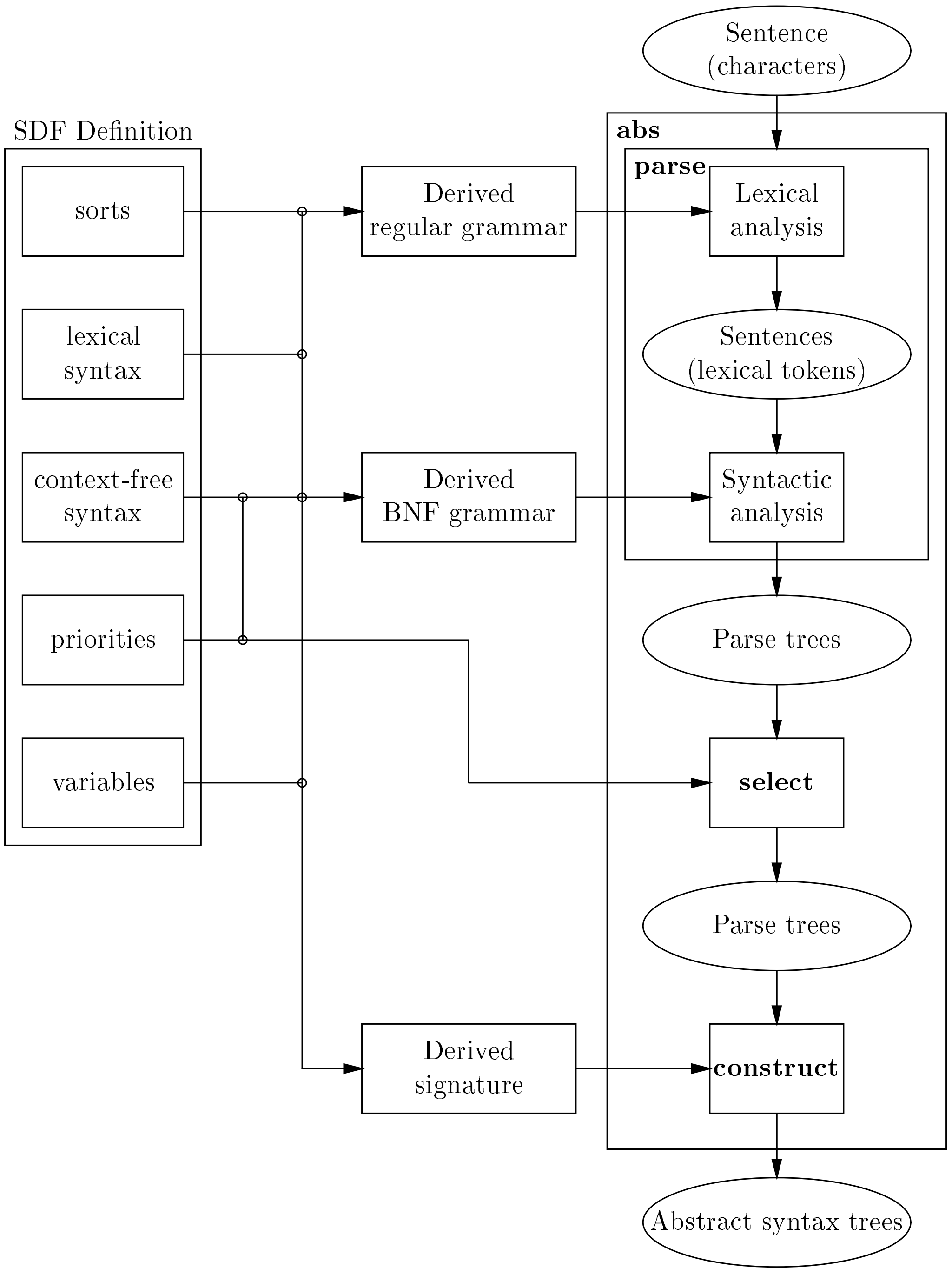


Figure 5: From sentence to abstract syntax tree(s).

- A derived signature, defining a set of abstract syntax trees.
- A mapping from parse trees over the derived BNF grammar to abstract syntax trees over the derived signature. This is a *fixed* mapping that is part of the definition of SDF. Other mappings can only be obtained by adding semantic rules to SDF definitions (see Chapter 9).

The relation between the different sections of an SDF definition and the information derived from it is shown in Figure 5. This figure also shows the function **abs** that converts an input sentence into (one or more) abstract syntax trees, together with the three auxiliary functions **parse**, **select** and **construct**. We will return to this figure in the following chapters.

3 Sorts

Sorts are declared by listing their name in the sorts section of the SDF definition. Some constraints are imposed on the use of sorts:

- The sorts `LAYOUT`, `IGNORE`, and `REJECT` are predefined and may not be redeclared. They may only be used as result sort of functions in the lexical syntax (see Sections 4.1, 4.4, and 4.9).
- The sort `CHAR` is predefined and may not be redeclared either. It may only be used in declarations of variables (see Chapter 7).

Each declared sort S becomes a non-terminal with the name <S> in the derived BNF grammar. If a sort appears as result sort of a lexical function, it will also be added to the derived regular grammar. A declaration for the predefined sort `CHAR` as well as declarations for all sorts declared in the SDF definition will be added to the derived signature. This is described by rules R1, G1, S1 and S2 in Chapter 8.

4 Lexical syntax

4.1 Introduction

Lexical syntax describes the low level structure of sentences (sequences of characters) in terms of *lexical tokens*. A lexical token is a pair consisting of a *sort name* and a *lexeme*. The former is used to distinguish classes of lexical tokens such as, e.g., identifiers and numbers. The latter is the actual text of the token. The lexical syntax defines:

- The substrings in the sentence that are lexical tokens.
- The substrings in the sentence that are layout symbols *between* lexical tokens and are to be skipped, i.e. layout symbols only serve the purpose of separating consecutive lexical tokens.

- The substrings in the sentence appearing *inside* lexical tokens that are to be ignored, i.e. ignored symbols do not contribute to the lexeme of the enclosing lexical token.

- Sequences of consecutive lexical tokens that are to be rejected.

The definition of lexical syntax consists of a set of one or more *function declarations*, each consisting of a regular expression and a result sort. All function declarations with a given output sort together define the lexical syntax of lexemes of that sort.

The layout symbols are defined by functions that have the predefined sort `LAYOUT` as result sort. They are typically used to define layout and comment conventions.

The ignored symbols are defined by functions that have the predefined sort `IGNORE` as result sort. They are typically used to define languages where certain layout symbols or comments may appear inside lexical tokens.

The rejected symbols are defined by functions that have the predefined sort `REJECT` as result sort. They are typically used to forbid certain sequences of lexical tokens.

From a given SDF definition, one can derive a regular grammar. This derived regular grammar is obtained as follows:

- Regular expressions are extracted from the lexical syntax (described in this chapter).

- All literals appearing in the context-free syntax are enumerated (Section 5.2).

- Regular expressions are extracted from variable declarations (Chapter 7).

- Regular expressions defining the complete alphabet are added when variables of sort `CHAR` are declared (Chapter 7).

Lexical analysis based on this derived regular grammar will skip tokens of the sort `LAYOUT`, and will present tokens of other sorts as input to the syntactic analysis phase. As a result, an input sentence will be subdivided in a not necessarily unique sequence of lexemes. Even more than one sort may be associated with each lexeme. The situation is shown in Figure 5.

A comparison between the classical model for lexical analysis and the model used in SDF is shown in Figure 6. In the classical model for lexical analysis (Figure 6(a)) the input string is subdivided in a single stream of lexical tokens which are separated by layout. All lexical ambiguities are resolved by the lexical analysis phase.

In the model used in SDF (Figure 6(b)) the input string may be subdivided in *several* streams of lexical tokens. Not all lexical ambiguities are resolved by the lexical analysis phase but they are passed on to the syntax analysis phase where more context information may be available to resolve them.

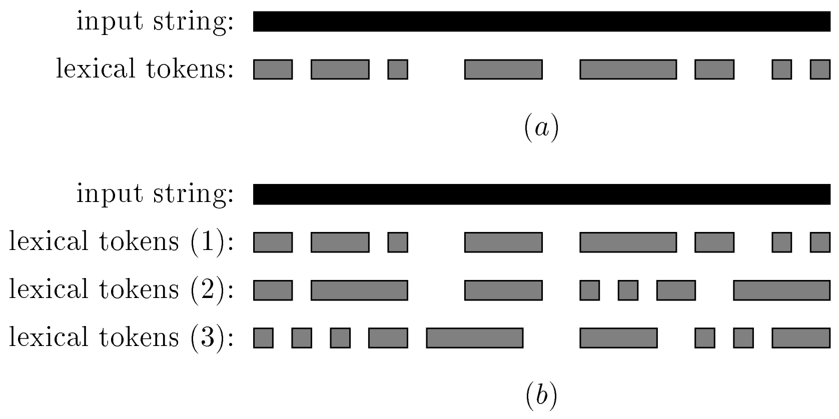


Figure 6: (*a*) Classical model for lexical analysis, (*b*) the SDF model.

The organization of this chapter is as follows. In Section 4.2 elementary lexical functions are described. Lexical function definitions may also contain character classes (Section 4.3) and repetition operators (Section 4.6). The definition of layout functions is described in Section 4.4 and the definition of ignore functions is dealt with in Section 4.5. Literals are discussed in Section 4.7. Reject functions are treated in Section 4.8. The disambiguation rules we use are given in Section 4.9. A rationale for the way in which we define lexical syntax concludes this chapter (Section 4.10).

4.2 Lexical functions

The lexical syntax contains declarations of *lexical functions* defining the syntactic form of lexemes. In their simplest form, lexical function declarations consist of one or more literal strings or names of sorts, and their result sort.

The effects on the derived regular grammar, BNF grammar and signature are as follows. For each sort L (with $L \notin \{\texttt{LAYOUT}, \texttt{IGNORE}, \texttt{REJECT}\}$) that appears as result sort of a lexical function:

- Define the symbol `<`L`-LEX>` both as lexical non-terminal in the derived regular grammar and as terminal symbol in the derived BNF grammar. (For $L \in \{\texttt{LAYOUT}, \texttt{IGNORE}, \texttt{REJECT}\}$) we only define the lexical non-terminal `<`L`-LEX>`).

- Add to the derived BNF grammar a rule of the form:

  `<`L`> ::= <`L`-LEX>`,

  where `<`L`-LEX>` is the name associated with L in the derived regular grammar.[4]

[4]We will allow context-free functions with a lexical sort as output sort as well as variables over lexical sorts. Therefore, three different entities may be associated with a

```
sorts VOWEL
lexical syntax
  "a"  -> VOWEL
  "e"  -> VOWEL
  "i"  -> VOWEL
  "o"  -> VOWEL
  "u"  -> VOWEL
  "y"  -> VOWEL
```

Figure 7: Vowels

- Add to the derived signature declarations of constants of sort `CHAR` for all characters in the alphabet as well as declarations for *lexical constructor functions* of the form

  l : `CHAR+ ->` *L*,

  where *l* is the name *L* written in lower case letters. This definition reflects the fact that lexemes appear as strings of characters in the abstract syntax tree.

The regular expression associated with each symbol `<`*L*`-LEX>` consists of the *or* of all left-hand sides of lexical function declarations for sort *L*. Sort names appearing in these left-hand sides are replaced by the symbol defined in the previous step. Note that, since we want to extract a regular grammar from the SDF definition, no cyclic dependency may exist between lexical function declarations. The above step leads to rules R1, G2, G3, S1, S4, and S5 in Chapter 8.

In Figure 7 one sort (`VOWEL`) and six lexical functions, are defined. All functions have result sort `VOWEL`. The derived regular grammar of this example is:

```
<VOWEL-LEX> = "a" | "e" | "i" | "o" | "u" | "y"
```

Given this definition, the following three tokens will be associated with the input sentence "`aio`":

```
(<VOWEL-LEX>, "a"),
(<VOWEL-LEX>, "i"),
(<VOWEL-LEX>, "o").
```

4.3 Character classes

Enumerations of characters occur frequently in lexical definitions. They can be abbreviated by using character classes enclosed by the symbols `[`

sort: (a) `<`*L*`-LEX>` denotes all lexical functions with result sort *L*; (b) `<`*L*`-VAR>` denotes all variables of sort *L* (see Chapter 7); (c) `<`*L*`>` denotes all entities with result sort *L* (i.e., lexical and context-free functions, and variables).

```
sorts VOWEL
lexical syntax
  [aeiouy] -> VOWEL
```

Figure 8: Shorter version of vowels using a character class.

```
sorts VOWEL DIGIT OTHER
lexical syntax
  [aeiouy]     -> VOWEL
  [0-9]        -> DIGIT
  ~[aeiouy0-9] -> OTHER
```

Figure 9: Vowels, digits and other characters.

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`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, and `9`. A character range of the form c_1-c_2 should satisfy the following restrictions:

- c_1 and c_2 are both lower case letters and c_1 precedes c_2 in the alphabet, or
- c_1 and c_2 are both upper case letters and c_1 precedes c_2 in the alphabet, or
- c_1 and c_2 are both digits and the numeric value of c_1 is smaller than that of c_2, or
- c_1 and c_2 are both escaped non-printable characters (see below) and the character code of c_1 is smaller than that of c_2.

Character classes may also be *complemented*. This is denoted by the prefix operator `~`. A complemented character class accepts all characters *not* listed in it.

In the derived regular grammar, character classes are replaced by an *or* of all characters in the class. See rule R3 in Chapter 8.

The definition of vowels in Figure 8 is equivalent to the one in Figure 7, but has been abbreviated by using a character class.

Given the lexical syntax in Figure 9, the input sentence "`3 aios`", will be subdivided in the following sequence of tokens:

```
(<DIGIT-LEX>, "3")
(<OTHER-LEX>, " ")
(<VOWEL-LEX>, "a")
(<VOWEL-LEX>, "i")
```

```
(<VOWEL-LEX>, "o")
(<OTHER-LEX>, "s")
```

Note that the space character occurring between the characters `3` and `a` as well as the character `s` at the end of the input string are classified as `<OTHER-LEX>`. The proper treatment of layout characters is discussed in Section 4.4.

Several characters have a special meaning in SDF and may cause problems when they are needed as ordinary characters in the definition. The backslash character (\) is used as escape character for the quoting of special characters. Thus the sequence *c* should be used whenever special character *c* is needed as ordinary character in the definition. In character classes, the following characters have a special meaning:

`[`	begin of character class
`]`	end of character class
`-`	character range
\	escape character.

In literal strings, the following characters have a special meaning:

`"`	double quote
\	escape character.

The literal strings containing one double quote or one backslash are thus written as `"\""` and `"\\"`, respectively. In addition to this escape convention, abbreviations exist for some frequently used layout characters as well as for arbitrary, non-printable, characters:

`\n`	newline character
`\r`	carriage return
`\t`	horizontal tabulation
ddd	a non-printable character with three digit (octal) code *ddd*.

These abbreviations can be used both in character classes and in literal strings.

4.4 Layout

Strings matched by lexical functions with the predefined sort `LAYOUT` as result sort will be skipped. When a string is matched by both a `LAYOUT` function and (one or more) functions with another result sort, the interpretation as layout symbol is suppressed; all other interpretations are passed to the syntax analysis phase (see Section 4.9).

The lexical syntax in Figure 10 defines the sort `LETTER` and defines three functions: one with output sort `LETTER` and two with the (predefined) sort `LAYOUT`. The derived regular grammar is:

```
<LETTER-LEX>    = "a" | ... | "z"
<LAYOUT-LEX>    = " " | "\n"
```

```
1.  sorts  LETTER
2.  lexical syntax
3.    [a-z] -> LETTER
4.    " "   -> LAYOUT
5.    "\n"  -> LAYOUT
```

Figure 10: Letters and spaces.

```
1.  sorts  ID
2.  lexical syntax
3.    [a-z] [0-9] -> ID
4.    "@"         -> IGNORE
```

Figure 11: Identifiers with embedded ignored symbols.

The input string “`x y z`” will, initially, be subdivided in the following five tokens:

```
(<LETTER-LEX>, "x"),
(<LAYOUT-LEX>, " "),
(<LETTER-LEX>, "y"),
(<LAYOUT-LEX>, " "),
(<LETTER-LEX>, "z").
```

Since all layout tokens will be skipped, only the tokens of sort `LETTER` will remain. In other words, the result of lexical analysis will consist of the following three tokens:

```
(<LETTER-LEX>, "x"),
(<LETTER-LEX>, "y"),
(<LETTER-LEX>, "z").
```

Note that lines (4) and (5) in this example can be replaced by the line:
`[ \n] -> LAYOUT`.

4.5 Ignore

Strings matched by lexical functions with the predefined sort `IGNORE` as result sort may appear inside a lexical token but do not contribute to the lexeme of that token. In Figure 11 we see a lexical syntax defining simple, two character, identifiers (sort `ID`) in combination with a rule with `IGNORE` as result sort. The derived regular grammar is:

```
<ID-LEX>     = ("a" | ... | "z") ("0" | ... | "9")
<IGNORE-LEX> = "@"
```

```
sorts LETTER LETTER-OR-DIGIT ID
lexical syntax
  [a-z]                   -> LETTER
  [a-z0-9]                -> LETTER-OR-DIGIT
  LETTER LETTER-OR-DIGIT* -> ID
  " "                     -> LAYOUT
```

Figure 12: Identifiers.

The input strings "`a4`", "`a@4`", "`@@a@@4@@`" (to mention a few examples) will all yield the single lexical token (`<ID-LEX>`, `"a4"`).

Ignored symbols can be used to describe, for instance, continuation lines and the treatment of space symbols in Fortran or the embedding of layout characters and comments in identifiers and keywords in Algol68.

4.6 Repetition: * and +

In many cases, lexical tokens can only be described by patterns that exhibit a certain repetition. Two postfix operators are available for this purpose. The operators `*` and `+` denote, respectively, zero or more and one or more repetitions of the sort, literal, or character class to which they are attached.

In the derived regular grammar, repetitions are mapped on the corresponding repetition operators in the regular grammar. See rule R4 in Chapter 8.

Figure 12 illustrates the definition of identifiers by means of the `*`-operator. The derived regular grammar for this example is:

```
<LETTER-LEX>          = "a" | ... | "z"
<LETTER-OR-DIGIT-LEX> = "a" | ... | "z" | "0" | ... | "9"
<ID-LEX>              = <LETTER-LEX> <LETTER-OR-DIGIT-LEX>*
<LAYOUT-LEX>          = " "
```

The input string "`r2d2 2de`", will be subdivided in the following three lexical tokens (after elimination of the single layout token occurring in it):

```
(<ID-LEX>, "r2d2"),
(<LETTER-OR-DIGIT-LEX>, "2"),
(<ID-LEX>, "de").
```

This subdivision may seem arbitrary, but is justified in the discussion of lexical ambiguities in the next section.

A more concise definition of identifiers is given in Figure 13. Here, the `*`-operator is applied directly to a character class. Note that, due to the elimination of the sort `LETTER-OR-DIGIT`, the above input string "`r2d2 2de`" is now erroneous.

```
sorts ID
lexical syntax
  [a-z] [a-z0-9]* -> ID
  " "             -> LAYOUT
```

Figure 13: More concise definition of identifiers.

```
1.  sorts ID INT
2.  lexical syntax
3.    [a-z] [a-z0-9]* -> ID
4.    [0-9]+          -> INT
5.    [ \n]           -> LAYOUT
6.    INT ID          -> REJECT
```

Figure 14: Rejecting adjacent integers and identifiers.

4.7 Literals

In Chapter 5 we will see that all literal symbols such as keywords and operator symbols that appear in the context-free syntax are added to the derived regular grammar. We reserve the symbol `<LITERAL-LEX>` for this purpose in the derived regular grammar. It will contain all literal symbols in the context-free grammar as alternatives. See rule R5 in Chapter 8.

A context-free syntax containing the literals `"if"`, `"then"`, `"("`, `")"` and `","` will, for instance, lead to the following definition of `<LITERAL-LEX>`:

```
<LITERAL-LEX>   = "if" | "then" | "(" | ")" | ","
```

4.8 Reject

All constructs described so far are "positive", i.e., they allow the explicit description of new forms of lexical tokens. The predefined sort `REJECT` is the only "negative" mechanism for defining lexical syntax, i.e., it allows to forbid the occurrence of certain classes of strings in the lexical syntax being defined. Each string matched by a lexical function with `REJECT` as result sort as well as all of its substrings are rejected as lexical token. If the string in question is embedded in a longer string that is acceptable as lexical token, that token will be returned. Otherwise, there is a lexical error.

The lexical syntax in Figure 14 defines identifiers (sort `ID`) and integer constants (sort `INT`). The rule in line (6) defines that an integer constant followed by an identifier should be rejected or, stated otherwise, that integer constants and identifiers may not be adjacent. From the input string "`a 2b`", first the lexical token (`ID`, `"a"`) will be recognized, a space will be skipped, but then a lexical error is found, due to the succession of the tokens (`INT`, `"2"`) and (`ID`, `"b"`). The input string "`a2b`", however, is correct and will be interpreted as the single lexical token (`ID`, `"a2b"`).

4.9 Disambiguation of lexical syntax

We first illustrate ambiguities at the lexical level by means of Figure 12. Consider all ten possible interpretations of the input string "`xy`":

1. (`<LETTER-LEX>`, `"x"`), (`<LETTER-LEX>`, `"y"`)
2. (`<LETTER-LEX>`, `"x"`), (`<LETTER-OR-DIGIT-LEX>`, `"y"`)
3. (`<LETTER-LEX>`, `"x"`), (`<ID-LEX>`, `"y"`)
4. (`<LETTER-OR-DIGIT-LEX>`, `"x"`), (`<LETTER-LEX>`, `"y"`)
5. (`<LETTER-OR-DIGIT-LEX>`,`"x"`), (`<LETTER-OR-DIGIT-LEX>`,`"y"`)
6. (`<LETTER-OR-DIGIT-LEX>`, `"x"`), (`<ID-LEX>`, `"y"`)
7. (`<ID-LEX>`, `"x"`), (`<LETTER-LEX>`, `"y"`)
8. (`<ID-LEX>`, `"x"`), (`<LETTER-OR-DIGIT-LEX>`, `"y"`)
9. (`<ID-LEX>`, `"x"`), (`<ID-LEX>`, `"y"`)
10. (`<ID-LEX>`, `"xy"`)

Ambiguities are taken as seriously as possible in SDF, but it is clear that some automatic disambiguation method is needed to reduce the vast amount of lexical ambiguities. Four disambiguation rules are used (in this order):

- Prefer Longest Match per Sort,
- Prefer non-Layout,
- Prefer Literals,
- Prefer Variables.

The Prefer Longest Match per Sort rule rejects all interpretations of the input string that are included in a longer interpretation for the same sort. In other words, for each sort the longest possible match is preferred. In Figure 12, only the interpretation (`<ID-LEX>`, `"xy"`) remains after application of this rule. Note that this is the only possible interpretation in Figure 13. Given the lexical syntax in Figure 15, the input string "`3.NE.4`" will lead to the following two lexical interpretations:

1. (`INT`, `"3"`), (`OP`, `".NE."`), (`INT`, `"4"`)
2. (`REAL`, `"3."`), (`ID`, `"NE"`), (`REAL`, `".4"`)

```
sorts INT REAL ID OP
lexical syntax
  [0-9]+              -> INT
  [0-9]* "." [0-9]*   -> REAL
  [A-Z]+              -> ID
  ".NE."              -> OP
  [ \n]               -> LAYOUT
```

Figure 15: Integers, reals and the `.NE.` operator.

Both interpretations will be passed on to the syntax analysis phase and it is likely that only the first one can be used to continue parsing.

After application of the Prefer Longest Match per Sort Rule, there may still be more than one interpretation of the lexical token left. The remaining three rules serve the purpose of reducing the number of possibilities still further.

The Prefer non-Layout rule eliminates all interpretations of the lexical token as layout symbol. When, after applying this rule, no interpretation remains, the next lexical token is taken from the input sentence.

The Prefer Literals rule applies to literals such as keywords and operator symbols that are introduced in the context-free syntax. Ambiguities may arise because some literal strings may be equal to lexemes defined in the lexical syntax. The Prefer Literals rule gives precedence to interpretations of the input string as literals. This applies, for instance, to identifiers defined in the lexical syntax that are also keywords defined in the context-free syntax. The Prefer Literals rule thus favours a so-called reserved word strategy for keywords. (In Section 11.3 it will be shown how this restriction can be circumvented.)

Ambiguities may also arise because an input string can both be interpreted as a lexical function or literal and as a variable defined in the variables section (see Section 7.2). In such cases the Prefer Variables rule applies and all interpretations other than that as variable are suppressed.

After applying the above disambiguation rules, all remaining interpretations of the input string are passed to the syntactic analysis phase.

4.10 Rationale for the definition of lexical syntax

There are three major reasons for making a distinction between lexical syntax and context-free syntax. First, the handling of ambiguities is different. At the lexical level myriads of (uninteresting or unintended) ambiguities exist. One needs fixed disambiguation rules to eliminate them (see Section 4.9). All possible interpretations that remain after application of these rules are returned by the lexical scanner. We even allow lexical ambiguities within the same sort. However, since lexemes are represented as linear strings of characters, the distinction between these ambiguities will be lost.

At the context-free level, the resolution of ambiguities has to be defined in the specification. For instance, by giving the priority and associativity of operators in the context-free syntax (see Chapter 6). In general, all disambiguation rules have disadvantages if one considers the problem of combining several modules containing SDF definitions (see Chapter 9).

Second, the value constructed after recognition is different. At the lexical level, tokens are constructed consisting of a name and a string. At the context-free syntax level, abstract syntax trees are constructed. This difference has consequences for the association of semantics with SDF definitions (see Chapter 9).

Finally, implementation techniques (and associated implementation tradeoffs) are different for lexical syntax and for context-free syntax (see Chapter 10). The limitations of our current disambiguation rules are discussed in Section 11.3.

5 Context-free syntax

5.1 Introduction

Context-free syntax describes the concrete and abstract syntactic structure of sentences in a language. The definition of context-free syntax in an SDF definition consists of declarations of *context-free functions*.

In Section 5.2, the simplest form of context-free functions is described. In Section 5.3 syntactic repetition is discussed. Chain and bracket functions are described in Section 5.4. Both lead to the elimination of certain nodes from abstract syntax trees.

Note that context-free functions may be followed by a list of attributes. In this chapter we will only describe the bracket attribute (Section 5.4). All other attributes have to do with priorities and are described in Chapter 6.

5.2 Context-free functions

In their simplest form, context-free functions are declared by giving their syntax (a list consisting of zero or more literal symbols and/or names of sorts) and their result sort.

We define the meaning of context-free functions by specifying their contribution to the derived regular grammar, derived BNF grammar, derived signature, and the mapping from parse trees to abstract syntax trees.

The contributions to the derived regular grammar, derived BNF grammar and derived signature are now described in more detail:

- For each literal symbol `"lit"` appearing in a function definition in the context-free syntax do the following:
  - add `"lit"` as alternative of the predefined lexical non-terminal `<LITERAL-LEX>` in the derived regular grammar;
  - add `"lit"` as terminal to the derived BNF grammar.

The lexical token `<LITERAL-LEX, "lit">` is identified with the BNF terminal `"lit"`.

- For each function declaration in the context-free syntax do the following:
  - add the grammar rules obtained by exchanging the left-hand and right-hand side of the function declarations to the derived BNF grammar;
  - add function declarations to the derived signature[5]: for each function some *new name* (see below) is generated and the sorts of its arguments and of its result are obtained from the sort names appearing in the original function definition.

See rules R5, G2, G5 and S6 in Chapter 8.

The *new names* mentioned above may be chosen arbitrarily as long as the same context-free function always gets the same abstract name and no two different context-free functions obtain the same abstract name. SDF does not prescribe any particular naming scheme in this case (but also see Section 5.6). In this manual we generate abstract function names by concatenating (some of) the literals in the original SDF function definition according to the following rules:

1. consecutive literals in the new name are separated by hyphens;
2. if there occur no literals in the function declaration, the name `empty` is used;
3. if the name generated by rules (1)–(2) contains characters that are not alpha-numerics or hyphens, it is surrounded by double quotes;
4. all names should be unique: a name generated by rules (1)–(3) can be postfixed with a hyphen and an arbitrary numeric string to make it unique.

Finally, we define the contribution of the context-free syntax to the mapping from parse trees to abstract syntax trees. In principle, there is a one-to-one correspondence between function definitions in the context-free syntax and both rules in the derived BNF grammar and function definitions in the derived signature[6]. This makes it straightforward to define the mapping from parse trees over the derived BNF grammar to abstract syntax trees over the derived signature. The mapping is recursively defined by giving the transformation for a parse tree P to an abstract syntax tree:

- If the root of P is labeled with a non-terminal and has a single child C whose root is labeled with lexical non-terminal `<`L`-LEX>`, with $L \neq$

[5]Except chain and bracket functions as we will see in Section 5.4.

[6]Exceptions to this general rule will be introduced in the description of lists (Section 5.3), and chain and bracket functions (Section 5.4).

```
1.  sorts NAT COORD CMND PROGRAM
2.  lexical syntax
3.    [0-9]+              -> NAT
4.    [ \n]               -> LAYOUT
5.  context-free syntax
6.    "(" NAT "," NAT ")" -> COORD
7.    "line" "to" COORD   -> CMND
8.    "move" "to" COORD   -> CMND
9.    CMND PROGRAM        -> PROGRAM
10.                       -> PROGRAM
```

Figure 16: Simple drawing language.

LITERAL, the result of transforming P is an abstract syntax tree whose root is labeled with the lexical constructor function for sort L (see Section 4.2) and whose single child is the result of transforming C.

- Otherwise, the root of P is labeled with a non-terminal and P is transformed as follows:
  - Recursively transform all children of the root of P that are labeled with a non-terminal. This gives a list A of abstract syntax trees[7].
  - The root of P corresponds to some rule R in the derived BNF grammar and some function definition F in the derived signature. P is now transformed into an abstract syntax tree consisting of the function defined in F with the elements of A as children.

See rules M1, M2 and M8 in Chapter 8.

When the mapping from parse tree to abstract syntax tree is used to associate an abstract syntax tree with a string, one should be aware of the fact that more than one parse tree may exist for a given string, and that more than one abstract syntax tree may thus be associated with it. This problem is discussed in Chapter 6.

Figure 16 describes a simple drawing language. A drawing program consists of a sequence of zero or more commands. There are two commands: one for moving the pen to a position in the plane, and one for drawing a line from the current position to a new position. A position is denoted by a pair of numbers. The derived regular grammar of this SDF definition is:

```
[3]  <NAT-LEX>      = ("0" | ... | "9")+
[4]  <LAYOUT-LEX>   = " " | "\n"
[6,7,8]
     <LITERAL-LEX> = "(" | "," | ")" | "line" | "to" | "move"
```

[7]Note that all children that are labeled with a terminal are eliminated and do not appear in the abstract syntax tree.

The derived BNF grammar is:

```
[3] <NAT> ::= <NAT-LEX>
[6] <COORD> ::= "(" <NAT> "," <NAT> ")"
[7] <CMND> ::= "line" "to" <COORD>
[8] <CMND> ::= "move" "to" <COORD>
[9] <PROGRAM> ::= <CMND> <PROGRAM>
[10] <PROGRAM> ::= ε
```

The derived signature is:

```
    sorts
[1]   CHAR NAT COORD CMND PROGRAM
    functions
      a       :                -> CHAR
      b       :                -> CHAR
      ...Declarations of constants for all characters...
[3]   nat     : CHAR+          -> NAT
[6]   "(-,-)" : NAT # NAT      -> COORD
[7]  line     : COORD          -> CMND
[8]  move     : COORD          -> CMND
[9]  empty-1  : CMND # PROGRAM -> PROGRAM
[10] empty-2  :                -> PROGRAM
```

In this signature we have introduced new names for the functions in the signature using the name generation rules given previously. Note that a certain freedom exists in the choice of names to be generated. For instance, the names `move`, `move-to` and `to` can be generated for the function corresponding to the move command. Lines 9 and 10 illustrate the use of the name `empty` for functions without syntax. Also note the use of numeric postfixes to make the name of each function unique.

Using the derived BNF grammar and the mapping from parse trees to abstract syntax trees defined above, the parse trees and corresponding abstract syntax trees for the drawing language programs `line to (15, 2)` and `line to (15,2) move to (3,4) line to (5,6)` are shown in Figures 17 and 18.

In Figure 16 we have quoted *all* literals occurring in the definition of context-free functions. As an abbreviation, these quotes may be omitted when the literal begins with a lower case letter and consists only of letters and digits. To avoid confusion between unquoted literals and sorts, we require that sort names begin with an upper case letter. Line 7, for instance, can be abbreviated to:

```
7.  line to COORD  -> COORD.
```

We will always use this convention in all examples (see also Appendix A).

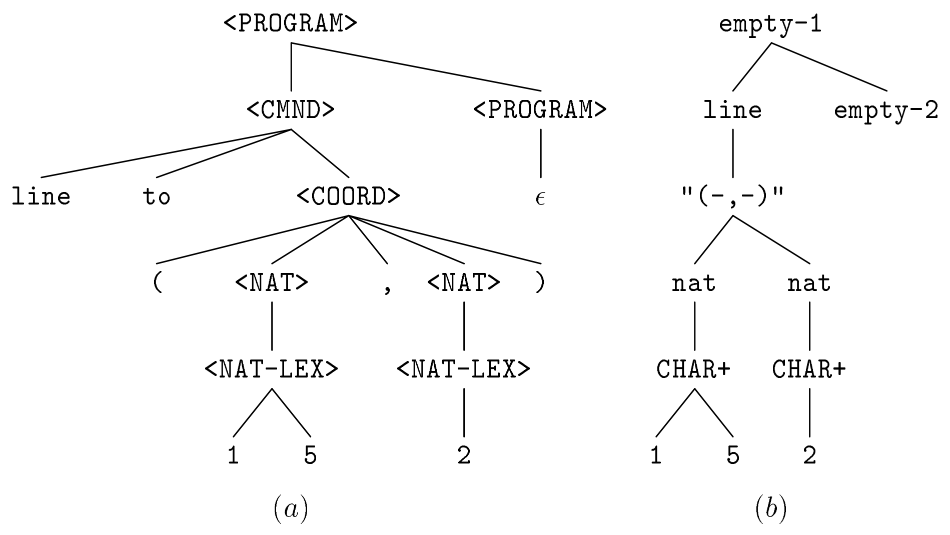


Figure 17: (*a*) Parse tree, and (*b*) corresponding abstract syntax tree.

5.3 Repetition: * and +

The context-free syntax in Figure 16 explicitly defines a drawing program as a recursively defined sequence of commands. The resulting abstract syntax trees encode the list structure by means of a fixed-arity list constructor (`empty-1` in the example). There are several problems with this approach:

- The SDF definition does not express the repetitive nature of the syntax in a clear way.
- The resulting abstract syntax trees are deeply nested and have a bias towards the left or the right, depending on the form of the SDF definition.

When attempting to define lists with separators using this style, the poor readability of the resulting definition becomes even more apparent. Therefore, SDF supports list constructs separately. They can have the following two forms:

- *Lists without separators* of the form S`*` or S`+` indicating respectively zero or more and one or more repetitions of sort S.
- *Lists with separators* of the form $\{S\ \ sep\}\otimes$, where $\otimes$ is either `*` or `+`, indicating respectively zero or more or one or more repetitions of sort S separated by the literal *sep*.

Lists may only be used in the left-hand side of a function definition (and may thus not appear as the result of a function).

In this manual, we will frequently write $\{S\ \ sep\}\otimes$ to denote all of the above list constructs, where $\otimes$ is either `*` or `+` and *sep* may be the empty

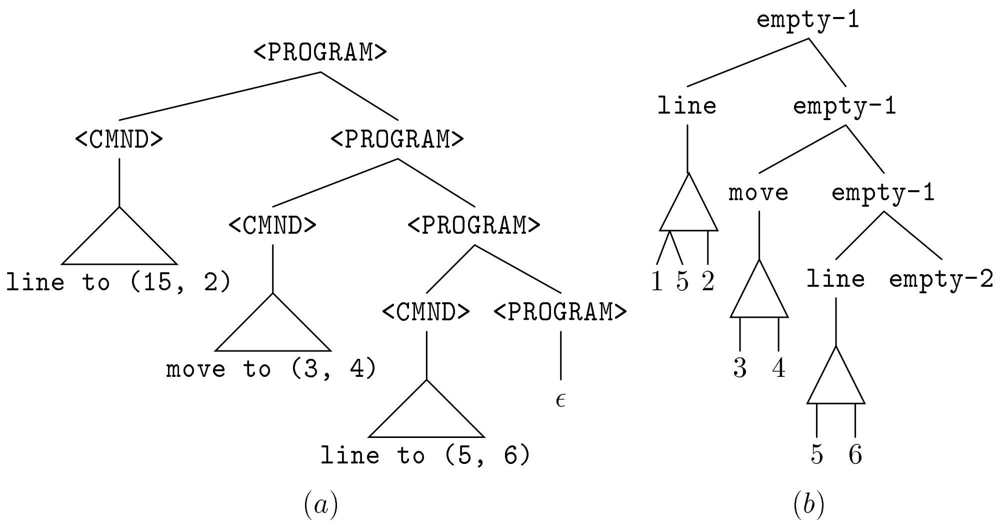


Figure 18: (a) Parse tree, and (b) corresponding abstract syntax tree.

string. The case of lists without separators will be treated as the special case that *sep* is equal to the empty string.

The effect of list constructs on the derived grammar and signature is as follows. For each list construct of the form $\{S \;\; sep\}\otimes$ that appears in a function declaration, define a non-terminal with the name `<`S`-in-`*sep*`-list>` and add the rule

`<`S`-in-`*sep*`-list> ::= <`S`>`.

Each occurrence of a list construct in the SDF definition is translated into the corresponding list construct in the grammar, i.e., a list of the form $\{S \;\; sep\}\otimes$ is translated to $\{$`<`S`-in-`*sep*`-list>` $sep\}\otimes$.[8]

The definition of the derived signature is also affected. In all function definitions, occurrences of list constructs of the form $\{S \;\; sep\}\otimes$ are translated to corresponding lists of the element sort in the signature, i.e. to $\{S \;\; sep\}\otimes$.

The mapping from parse trees to abstract syntax trees is extended as follows. Let P be a parse tree whose root is labeled with $\{S \;\; sep\}\otimes$. There are two cases:

- *sep* is empty: let $C_1, ..., C_k$ be the k children of the root of P; or
- *sep* is not empty, the root of P has $2k - 1$ children and let $C_1, ..., C_k$ be the odd-numbered children (i.e., the children corresponding to list elements and not to separators).

[8]One may wonder why we use this, seemingly too complex, translation scheme. A much simpler method would be to map list constructs directly on the corresponding construct in the grammar, i.e., $\{S \;\; sep\}\otimes$ in a function definition is mapped on $\{S \;\; sep\}\otimes$ in the derived grammar. However, in Chapter 7 we will see that this simple method cannot be extended to the case that the SDF definition contains definitions of variables.

```
1.  sorts NAT COORD CMND PROGRAM
2.  lexical syntax
3.    [0-9]+              -> NAT
4.    [ \n]               -> LAYOUT
5.  context-free syntax
6.    "(" NAT "," NAT ")" -> COORD
7.    line to COORD       -> CMND
8.    move to COORD       -> CMND
9.    CMND*               -> PROGRAM
```

Figure 19: Use of a list operation in definition of the drawing language.

The result of transforming P is an abstract syntax tree whose root is labeled with $\{S \;\; sep\}\otimes$ and with the transformed versions of $C_1, ..., C_k$ as children. Nodes in the parse tree that are labeled with `<`S`-in-`*sep*`-list>` are removed and do not appear in the abstract syntax tree. The effects of list constructs are summarized in rules G4, G5, S6, M3, M7 in Chapter 8.

The use of list constructs in the definition of the drawing language is shown in Figure 19. Lines 9 and 10 of Figure 16 have been replaced by the single line 9. The derived regular grammar is not affected by this modification. The derived BNF grammar becomes:

```
[3]  <NAT>           ::= <NAT-LEX>
[6]  <COORD>         ::= "(" <NAT> "," <NAT> ")"
[7]  <CMND>          ::= "line" "to" <COORD>
[8]  <CMND>          ::= "move" "to" <COORD>
[9]  <PROGRAM>       ::= <CMND-in--list>*
[9]  <CMND-in--list> ::= <CMND>
```

The derived signature becomes:

```
     sorts
[1]    CHAR NAT COORD CMND PROGRAM
     functions
       a       :             -> CHAR
       b       :             -> CHAR
       ...Declarations of constants for all characters...
[3]    nat     : CHAR+       -> NAT
[6]    "(-,-)" : NAT # NAT -> COORD
[7]    line    : COORD       -> CMND
[8]    move    : COORD       -> CMND
[9]    empty   : CMND*       -> PROGRAM
```

Figure 20 shows the parse tree and corresponding abstract syntax for the same sentence as used in Figure 18. Note, how the list construct is represented by a node with a variable number of children.

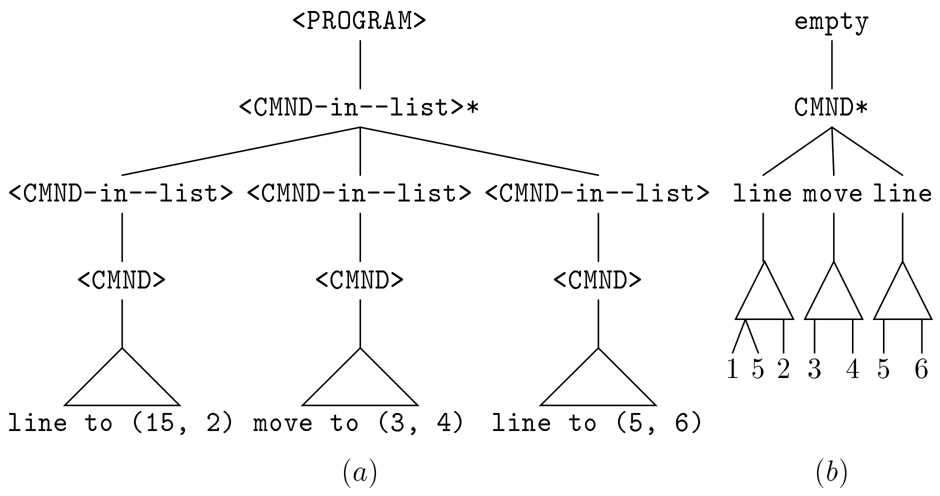


Figure 20: (*a*) Parse tree and (*b*) abstract syntax tree containing a `*` operator.

```
1.  sorts ID DECL TYPE
2.  lexical syntax
3.    [a-z]+                    -> ID
4.    [ \n]                     -> LAYOUT
5.  context-free syntax
6.    decl { ID ","}+ ":" TYPE -> DECL
7.    integer                   -> TYPE
8.    real                      -> TYPE
```

Figure 21: Declarations.

In Figure 21 the use of a list constructor with a separator is illustrated. The example shows a simple form of variable declarations in a Pascal-like language. A declaration consists of the keyword `decl`, one or more identifiers separated by commas, followed by a colon and a type. The derived regular grammar is:

```
[3]  <ID-LEX>      = ("a" | ... | "z")+
[4]  <LAYOUT-LEX>  = " " | "\n"
[6,7,8]
     <LITERAL-LEX> = "decl" | "," | ":" | "integer" | "real"
```

The derived BNF grammar is:

```
[3]  <ID>           ::= <ID-LEX>
[6]  <DECL>         ::= "decl" {<ID-in-,-list> ","}+ ":" <TYPE>
[6]  <ID-in-,-list> ::= <ID>
[7]  <TYPE>         ::= "integer"
[8]  <TYPE>         ::= "real"
```

The derived signature is:

```
     sorts
[1]    CHAR ID DECL TYPE
     functions
       ...Declarations of constants for all characters...
[3]    id     : CHAR+             -> ID
[6]    decl   : {ID ","}+ # TYPE -> DECL
[7]    integer:                   -> TYPE
[8]    real   :                   -> TYPE
```

Figure 22 shows the parse tree and corresponding abstract syntax tree for the sentence `decl ab, c, xyz :  real`.

5.4 Chain and bracket functions

Abstract syntax trees form a concise way of representing the structure of sentences. Further abbreviations are possible, however, by eliminating certain classes of nodes from the tree. SDF defines abbreviations for chain and bracket functions.

A *chain function* has a definition of one of the following forms:

1a. S -> B
1b. $\{S\ sep\}\otimes$ -> B

where S ("small") and B ("big") are previously defined sorts and *sep* is a literal. A chain function only establishes an inclusion relation between two sorts. The above function declarations will lead to the rules

[1a] <B> ::= <S>
[1b] <B> ::= $\{$<S-in-*sep*-list> *sep*$\}\otimes$
<S-in-*sep*-list> ::= <S>

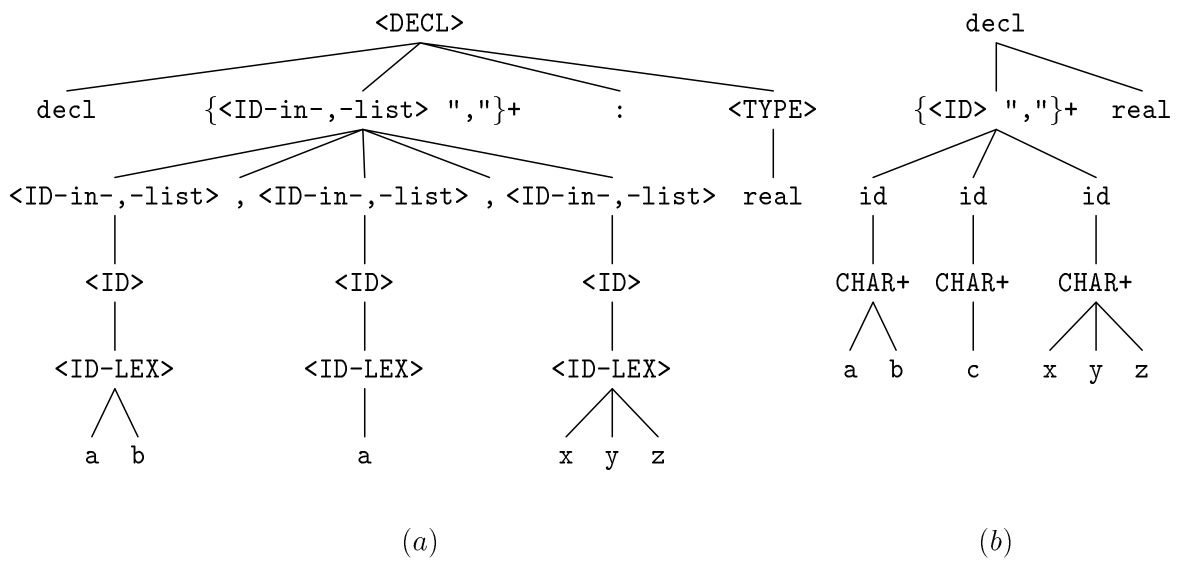


Figure 22: (*a*) Parse tree and (*b*) corresponding abstract syntax tree.

in the derived BNF grammar and to the function declarations

[1a] `empty :` S `->` B
[1b] `empty :` $\{S\ \mathit{sep}\}\otimes$ `->` B

in the derived signature. According to the rules of abstract syntax tree formation given earlier, a tree of sort $\{S\ \mathit{sep}\}\otimes$ cannot occur directly at a position where a tree of sort B is required, but will have to be embedded in an application of the function `empty`. Such nodes corresponding to chain functions are undesirable for two reasons:

- When the abstract syntax tree is used by a syntax-directed editor, these nodes may be visited during a walk through the tree. However, since no syntax is associated with chain functions, no textual distinction can be made between a visit to the node itself or a visit to either its parent or its child.

- When semantic rules are associated with abstract syntax trees, extra rules are needed for the processing of chain functions, but in general the concrete form of such rules will be ambiguous.

In SDF, the nodes corresponding to chain functions are therefore eliminated from abstract syntax trees in the following way:

- In the derived signature subsort declarations rather than ordinary function declarations are generated for each chain function.

- In the mapping from parse tree to abstract syntax tree the result of transforming a parse tree P whose root node corresponds to a chain function is defined as the result of transforming the single child of the root of P. The resulting tree is correct due to the subsort declaration generated in the previous step.

This leads to rules S3, S6 and M4 in Chapter 8.

In the sequel, it will be convenient to have a parent-child relationship in parse trees that disregards nodes corresponding to chain rules. Therefore, we will use “child” (i.e., child between double quotes) to mean either an immediate child or a child that can be reached via one or more chain rules.

A *bracket function* has a definition of the following form:

1. `"`*open*`"` S `"`*close*`" ->` S

where `"`*open*`"` and `"`*close*`"` are arbitrary, non-empty, literals acting as opening and closing brackets for the sort S. Typical examples are the parentheses `"("` and `")"` in arithmetic expressions and the delimiters `"begin"` and `"end"` in programs. Here again, application of the standard definitions leads to the appearance of additional nodes in the abstract syntax tree corresponding to bracket functions. In most cases, brackets are only introduced for grouping and disambiguation (see Chapter 6), but have no further meaning. Therefore, SDF provides the facility for declaring that a certain function is used only for bracketing by attaching the attribute `bracket` to it. More precisely, by declaring the above function as

1. `"`*open*`"` S `"`*close*`" ->` S {`bracket`}

the following effect is achieved:

- In the derived signature no function declaration is generated for functions with the attribute `bracket`.
- In the mapping from parse tree to abstract syntax tree the result of transforming a parse tree P whose root corresponds to a bracket function is defined as the result of transforming the second child of the root of P, i.e., the only child labeled with a non-terminal. The resulting tree is correct since the argument and result sorts of a bracket function are equal.

This leads to rules S6 and M5 in Chapter 8.

Since brackets are necessary for overruling the priority and associativity of functions (see Chapter 6), we require that suitable bracket functions are declared for the argument and result sorts of

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

In this way we guarantee that every abstract tree t over the derived signature has a corresponding string representation s such that $\mathbf{abs}(s) = \{t\}$ (cf. Figure 5).

Figure 23 illustrates the concepts of chain functions and bracket functions. A simple programming language is defined with identifiers and natural numbers as lexical sorts, and several language constructs which can be combined into programs. The function in line 8 defines `begin`–`end` brackets for lists of statements. The function in line 9 is a chain function from {`STAT` `";"`}* to `SERIES`. The functions in lines 13 and 14 are chain functions that convert identifiers and numbers to expressions. The derived regular grammar for Figure 23 is:

```
[3]  <ID-LEX>      = ("a" | ... | "z")+
[4]  <NAT-LEX>     = ("0" | ... | "9")+
[5]  <LAYOUT-LEX>  = " " | "\t" | "\n"
[7,8,9,10,11,12]
     <LITERAL-LEX> = "program" | "begin" | "end" | ";" |
                     ":=" | "if" | "then" | "else" | "fi" |
                     "until" | "do" | "od"
```

The derived BNF grammar is:

```
[3]  <ID>             ::= <ID-LEX>
[4]  <NAT>            ::= <NAT-LEX>
[7]  <PROGRAM>        ::= "program" <SERIES>
[8]  <SERIES>         ::= "begin" <SERIES> "end"
[9]  <SERIES>         ::= { <STAT-in-;-list> ";" }*
[9]  <STAT-in-;-list> ::= <STAT>
[10] <STAT>           ::= <ID> ":=" <EXP>
[11] <STAT>           ::= "if" <EXP> "then" <SERIES>
                          "else" <SERIES> "fi"
[12] <STAT>           ::= "until" <EXP> "do" <SERIES> "od"
[13] <EXP>            ::= <ID>
[14] <EXP>            ::= <NAT>
```

The derived signature is:

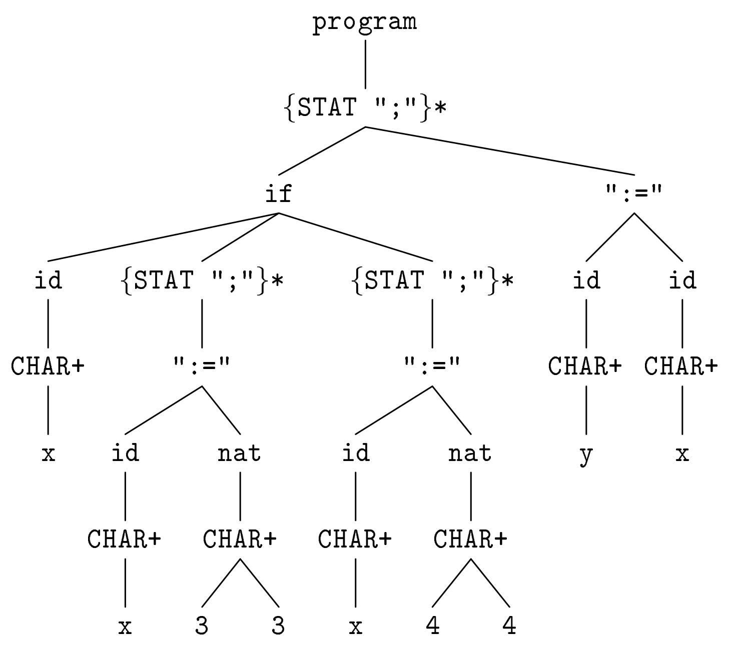
```
     sorts
[1]    CHAR ID NAT PROGRAM SERIES STAT EXP
     subsorts
[9]    {STAT ";"}* < SERIES
[13]   ID < EXP
[14]   NAT < EXP
     functions
       ...Declarations of constants for all characters...
[3]    id      : CHAR+                -> ID
[4]    nat     : CHAR+                -> NAT
[7]    program : SERIES               -> PROGRAM
[10]   ":="    : ID # EXP             -> STAT
[11]   if      : EXP # SERIES # SERIES -> STAT
[12]   until   : EXP # SERIES         -> STAT
```

```
1.  sorts ID NAT PROGRAM STAT SERIES EXP
2.  lexical syntax
3.    [a-z]+                            -> ID
4.    [0-9]+                            -> NAT
5.    [ \t\n]                           -> LAYOUT
6.  context-free syntax
7.    program SERIES                    -> PROGRAM
8.    begin SERIES end                  -> SERIES {bracket}
9.    { STAT ";" }*                     -> SERIES
10.   ID ":=" EXP                       -> STAT
11.   if EXP then SERIES else SERIES fi -> STAT
12.   until EXP do SERIES od            -> STAT
13.   ID                                -> EXP
14.   NAT                               -> EXP
```

Figure 23: A simple programming language.

```
program
 begin
  if x then
     x := 33
  else
     begin
       x := 44
     end
  fi;
  y := x
 end
```

(*a*)



(*b*)

Figure 24: (*a*) Program and (*b*) corresponding abstract syntax tree.

```
sorts E
lexical syntax
  [0-9]+    -> E
  [ \n]     -> LAYOUT
context-free syntax
  E "+" E   -> E
  E "*" E   -> E
  "(" E ")" -> E {bracket}
```

Figure 25: Ambiguous definition of simple arithmetic expressions

5.5 Injections

(** to be written **)

5.6 Explicit naming of functions in the abstract syntax

Recall from Section 5.2 how the names of the functions in the derived signature were derived automatically from the SDF definition. This is fine for writing self-contained SDF definitions, but makes it difficult to let external tools access the abstract syntax trees generated by the SDF implementation. SDF therefore provides a mechanism to control the names in the derived signature.

To name a rule explicitly, write the function name followed by a colon before the context-free function declaration, like in

```
plus :  EXP "+" EXP -> EXP
```

For list constructs, names are generated of the form *Ssep*`_list`, where *S* is the element sort and *sep* is a keyword replacing the iterated separator. For instance, the rule for a list of expressions separated by a semi-colon:

```
{ EXP ";"}+ -> EXP_S
```

generates the name "`EXPSEMIC_list`" which may be used elsewhere. This automatically generated name may be overruled by inserting a name, preceded by a colon, after the iterator:

```
{ EXP ";"}+:exp_s -> EXP_S
```

Each user-defined abstract function name may only be declared once.

6 Priorities

6.1 Motivation

The derived BNF grammar of an SDF definition may be ambiguous. As a consequence, there may be sentences in the language defined by the derived

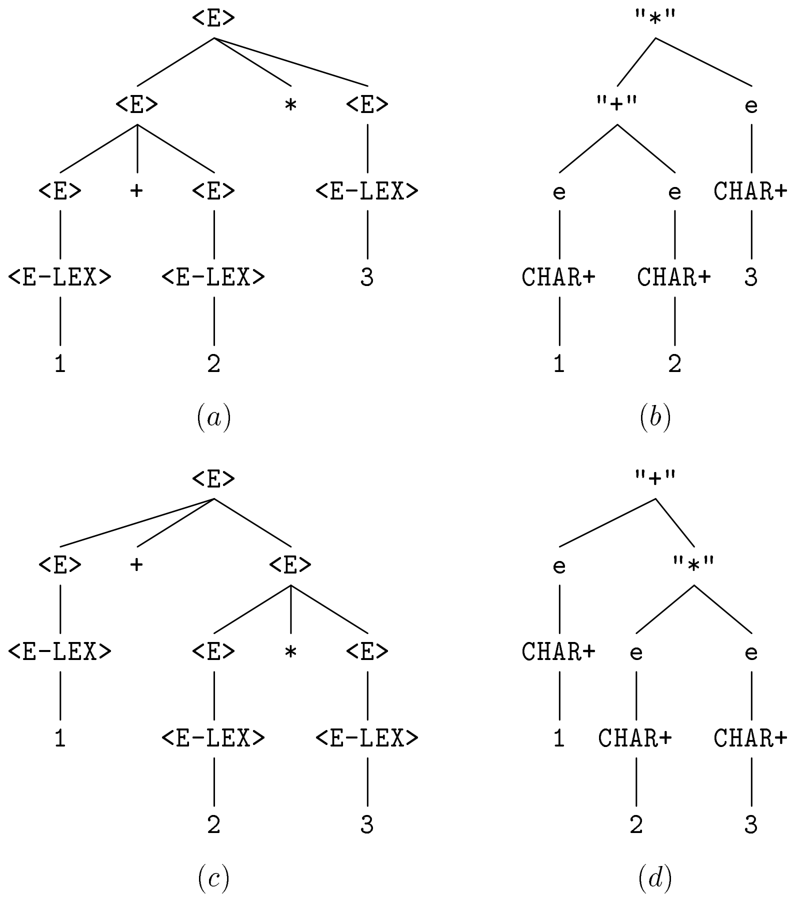


Figure 26: (*a*,*c*) Parse trees and (*b*,*d*) corresponding abstract syntax trees for `1+2*3`.

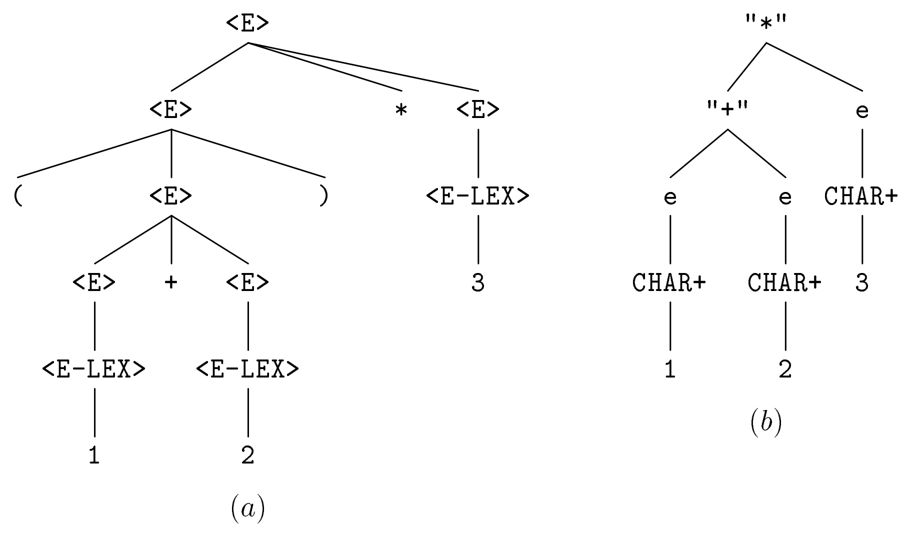


Figure 27: (*a*) Parse tree and (*b*) abstract syntax tree for `(1+2)*3`.

grammar which have more than one parse tree (and thus more than one abstract syntax tree). Consider the SDF definition of arithmetic expressions with addition and multiplication in Figure 25. An ambiguous sentence in this language is, for instance, `1+2*3`, with parses `(1+2)*3` and `1+(2*3)`. The associated trees are shown in Figure 26. The ambiguity in this example is clearly caused by the lack of a definition of the priority (and, in other sentences, associativity) of the operators `+` and `*`. The parse tree in Figure 26(a) should be rejected when, as usual, `*` has a higher priority than `+`. In Figure 27 the trees for the sentence `(1+2)*3` are shown. It is instructive to compare the (identical) abstract syntax trees in Figures 26(b) and in 27(b). The former has to be rejected because the corresponding parse tree contains a priority conflict between `+` and `*`, whereas the latter is acceptable. This shows that parse trees rather than abstract syntax trees should be used for disambiguation.

There are at least two general methods to remove ambiguities from a context-free grammar. The first method is to introduce new non-terminals encoding the priority and associativity of the operator symbols in the grammar. The new non-terminals prevent certain derivations that were possible in the original grammar and made it ambiguous. This method has the disadvantage that the definition of priorities and associativities is implicit and that additional measures are necessary to prevent the new non-terminals from affecting the abstract syntax. The second method is to add explicit definitions of priority and associativity to the grammar. This latter method is adopted in SDF.

6.2 Priority and associativity

SDF provides three mechanisms for the definition of priority and associativity of functions in the context-free syntax (and thus of rules in the derived BNF grammar):

- relative priorities of functions,
- associativity of functions, and
- associativity of groups of functions.

The *relative priority of functions* is established by declarations of the form

f `>` g

where f and g are *complete* function declarations. Notice that we do not assign absolute priority levels to functions, but establish only their priority relative to other ones. By taking the transitive closure of the priority declarations given in the SDF definition, a strict, i.e., non-reflexive, partial order on the set of context-free functions should be obtainable.

Functions with a higher priority bind more strongly and the corresponding nodes in the parse tree should thus appear at lower levels in the tree than nodes corresponding to functions with lower priorities. Chain and bracket functions may not appear in priority declarations. Priorities are declared in a separate section in the context-free syntax. Lists of function declarations serve as abbreviation, i.e., f `>` `{`g`,` h`}` is an abbreviation for f `>` g`,` f `>` h.

Associativity attributes can be attached to binary functions of the form S *op* S `->` S, where *o*p is a literal 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 sentence of sort S. The particular associativity associated with *op* determines the intended parse of such sentences. When a node corresponding to a function f has a first or last "child" (i.e., disregarding any intermediate chain function nodes) corresponding to a function g, we call these occurrences of f and g *related*. Associativity attributes define how to accept or reject parse trees containing related occurrences of a single function. The following associativity attributes can be attached to a 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.

```
sorts E
lexical syntax
  [0-9]+    -> E
  [ \n]     -> LAYOUT
context-free syntax
  E "+" E   -> E  {left}
  E "*" E   -> E  {left}
  "(" E ")" -> E  {bracket}
priorities
  E "*" E -> E > E "+" E -> E
```

Figure 28: Unambiguous version of example in Figure 25

From a syntactic point of view there is no difference between `left` and `assoc`. However, the attribute `assoc` in the SDF definition is translated into an `assoc` attribute in the derived signature (see rule S6 in Chapter 8). Therefore, `assoc` also has semantic consequences.

Group associativity attributes define how to accept or reject parse trees containing related occurrences of different functions with the same priority. Group associativity is defined by prefixing a list of function declarations 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 different functions appearing in the list.

The simplest application of priority and associativity declarations is elimination of parse trees that contain one of the following *conflicts*:

- a parent has a "child" with a lower priority than the parent itself;
- a parent has a first or last "child" that is in conflict with the associativity of parent and "child".

Three examples will illustrate this use of priority and associativity. One way of making the example in Figure 25 unambiguous is shown in Figure 28. The choices made in this definition are:

- `*` has a higher priority than `+`. The only interpretation of `1+2*3` thus becomes `1+(2*3)`.
- `+` is left-associative. The only interpretation of `1+2+3` is thus `(1+2)+3`.
- `*` is also left-associative.

A more elaborate version of arithmetic expressions is given in Figure 29. The operators in these expressions have the following properties:

```
sorts E
lexical syntax
  [0-9]+     -> E
  [ \n]      -> LAYOUT
context-free syntax
  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}
priorities
  E "^" E -> E >
  {non-assoc: E "*" E -> E, E "/" E -> E} >
  {left: E "+" E -> E, E "-" E -> E}
```

Figure 29: More elaborate arithmetic expressions.

- `^` has the highest priority and associates from right to left.
- `*` and `/` are next in the priority ordering, `*` is left-associative, `/` is non-associative, and `*` and `/` form a non-associative group.
- `+` and `-` have the lowest priority, `+` is left-associative, `-` is non-associative, and `+` and `-` form a left-associative group.

This leads to the following association of parses with sentences:

Sentence	**Parse**
`1^2^3`	`1^(2^3)`
`1^2*3`	`(1^2)*3`
`1*2*3`	`(1*2)*3`
`1/2/3`	*none*
`1*2/3`	*none*
`1-2-3`	*none*
`1+2+3`	`(1+2)+3`
`1-2+3`	`(1-2)+3`
`1+2-3`	`(1+2)-3`

The solution of the classical *dangling else* problem is shown in Figure 30. The priority declaration selects the parses in which `else`-parts are associated with the nearest possible preceding `if`-part. The sentence `if x then if y then stat else stat` will thus be parsed as `if x then begin if y then stat else stat end`.

The examples in Figures 25, 28, 29 and 30 illustrate how the definition of priority and associativity can be used to reject certain parse trees of a sentence. There are, however, SDF definitions for which this form of

```
sorts ID STAT
lexical syntax
  [a-z]+                    -> ID
  [ \n]                     -> LAYOUT
context-free syntax
  if ID then STAT           -> STAT
  if ID then STAT else STAT -> STAT
  stat                      -> STAT
  begin STAT end            -> STAT {bracket}
priorities
  if ID then STAT else STAT -> STAT > if ID then STAT -> STAT
```

Figure 30: Solution to the dangling else problem.

disambiguation is not sufficient since they generate sentences that have more than one *conflict-free* parse tree. How can in such cases the set of remaining parse trees be reduced still further? Our solution is to extend the priority ordering $>$ on functions to *a priority ordering* $>>$ *on parse trees* and to reject a parse tree if there is another parse tree with a higher priority. The relation $P_1 >> P_2$ holds between two parse trees P_1 and P_2 if (** injections **)

- P_1 is not equal to P_2, and
- if, for any f, P_1 contains more nodes corresponding to f than P_2, then P_2 contains more nodes corresponding to some function g with $g < f$ than P_1. (This also covers the cases that there are no nodes corresponding to f in P_2 or no nodes corresponding to g in P_1).

The ordering $>>$ is a variant of the so-called *multiset ordering* [DM79, JL82].

The parse trees for a given sentence are thus selected in two phases. First, the parse trees containing conflicts are removed. Next, all parse trees are removed that are smaller than another remaining parse tree. Note that whereas the first phase may invalidate a sentence by blocking all its parse trees, the second phase always selects at least one parse tree, so a sentence that passes the first selection phase also passes the second one. The total selection process is illustrated in Figure 31.

We illustrate this approach with the example in Figure 32 in which arithmetic expressions with the operators `+` and `*` are defined over natural and real numbers. In this example, the sorts `N` and `R` stand for, respectively, the natural and real numbers. On both sorts, the operators `+` and `*` are defined. Finally, there is a chain function embedding `N` in `R`. Ambiguities may arise in this example since one can sometimes choose between using an operator defined on `N` together with a conversion from `N` to `R`, or using the operator defined on `R` directly. Therefore, priorities are defined such that

- `*` on `N` has a higher priority than `*` on `R`;

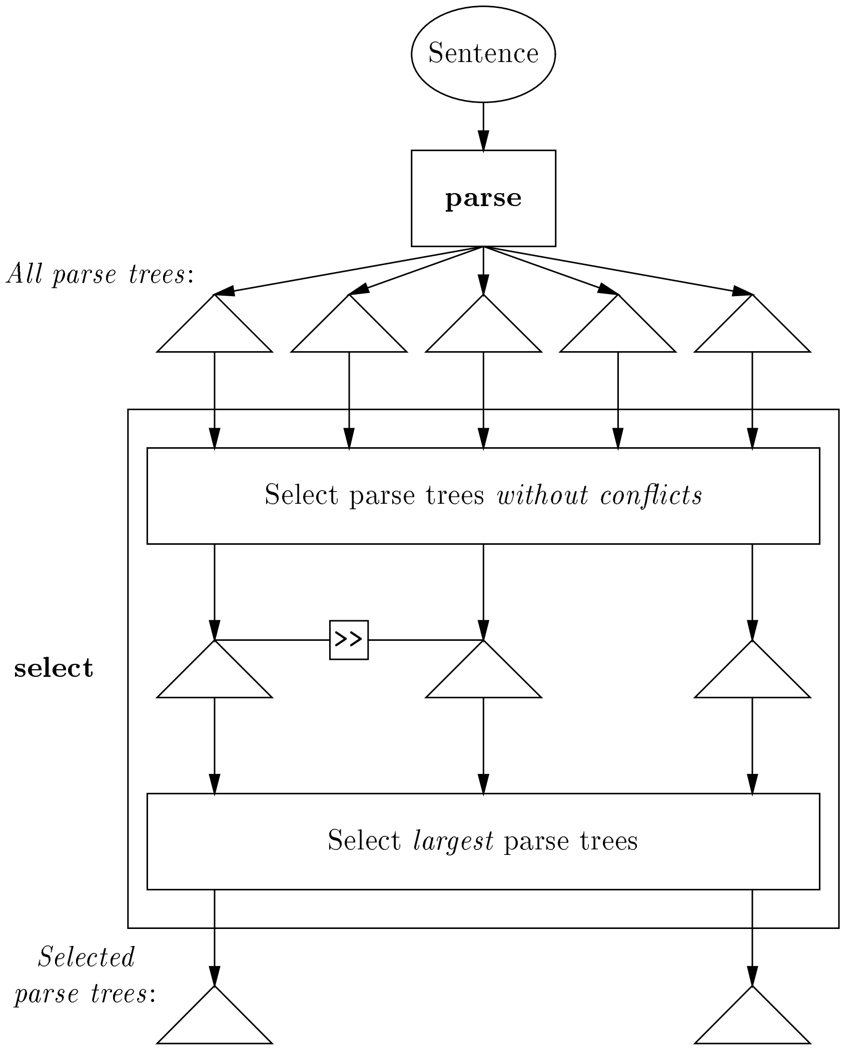

Sentence
parse
All parse trees:
Select parse trees without conflicts
>>
select
Select largest parse trees
Selected parse trees:

Figure 31: Selection of parse trees for a given sentence.

```
sorts D N R
lexical syntax
  [0-9]+  -> D
  D       -> N
  D "." D -> R
  [ \n]   -> LAYOUT
context-free syntax
  N "+" N -> N  {left}
  N "*" N -> N  {left}
  N       -> R
  R "+" R -> R  {left}
  R "*" R -> R  {left}
priorities
  N "*" N -> N >  R "*" R -> R >
  N "+" N -> N  > R "+" R -> R
```

Figure 32: Arithmetic expressions for natural and real numbers.

- `+` on `N` has a higher priority than `+` on `R`;
- `*` always has a higher priority than `+` irrespective of the sort involved (as usual).

These priorities select parses with the ordinary priority of `+` and `*` and a preference for functions on `N`. Figure 33 shows the three possible parse trees for the sentence `1.5+2*3`. The priorities from the example are now applied as follows. Tree (a) contains a conflict since a node corresponding to `*` on `R` has as child a node corresponding to `+` on `R`. Trees (b) and (c) are conflict-free. Comparing these two trees, one sees that $(c) >> (b)$ holds. The only function that occurs more often in (c) than in (b) is `*` on `N`: it occurs once in (c) but not in (b). This is compensated for by the fact that `*` on `R` (which has a lower priority) appears once in (b) but not in (c). Hence, tree (c) is the (only) parse tree associated with the sentence `1.5+2*3`.

As a final example, we show in Figure 34 how the interaction between general context-free functions and special case functions can be described by means of priorities. The example originates from [AJU75] and concerns expressions 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`" is introduced in line 5, which should have priority over a combination of the functions defining `sub` and `sup` in lines 3 and 4.

Figure 35 shows the three parse trees of the sentence `a sub a sup a`. Parse tree (a) has to be rejected since it contains a conflict (caused by left group associativity). From the two remaining trees, (c) is selected since it is larger (in the multiset ordering) than (b).

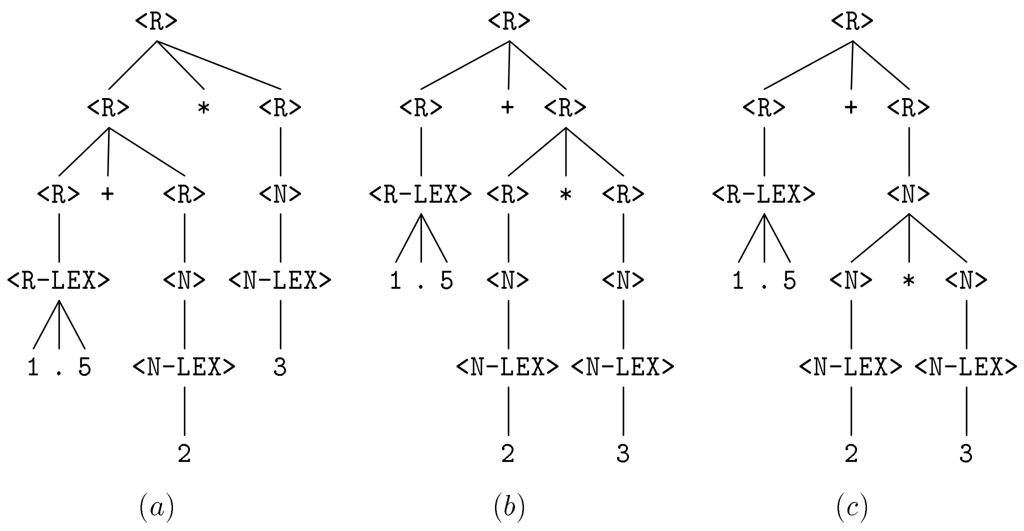


Figure 33: The three parse trees of `1.5 + 2 * 3`.

```
1.  sorts E
2.  context-free syntax
3.    E sub E         -> E {left}
4.    E sup E         -> E {left}
5.    E sub E sup E   -> E
6.    "{" E "}"       -> E {bracket}
7.    a               -> E
8.  priorities
9.    E sub E sup E -> E >
10.   {left: E sub E -> E, E sup E -> E}
```

Figure 34: Expressions for subscripts and superscripts.

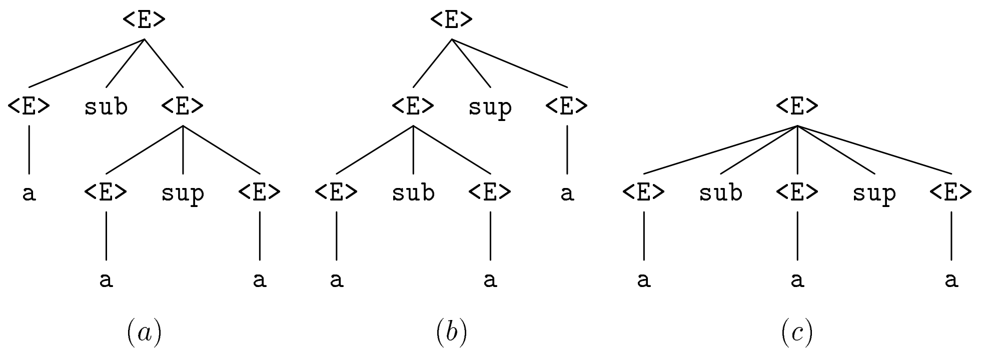


Figure 35: The three parse trees of `a sub a sup a`.

7 Variables in SDF definitions

7.1 Motivation

The lexical and context-free syntax sections in an SDF definition completely define which input sentences are legal and how they should be mapped onto (one or more) abstract syntax trees. These input sentences are constant and completely fixed; there is no systematic way to extend or modify them. There are, however, cases in which the need arises for *incomplete* or *parameterized* input sentences. Two important cases are:

- If one associates semantic rules with SDF functions (see Chapter 9), it is necessary to use variables in these rules. For instance, when defining the rules for multiplication on natural numbers, one needs rules of the form `x*(y+1) = x*y+x`, where `x` and `y` are variables over the sort of natural numbers.

- During syntax-directed editing, the program text under construction is most likely to be incomplete. The text contains "holes" that still have to be filled in with strings of a certain type. It is desirable to be able to determine the syntactic correctness of such incomplete programs.

To handle such cases, SDF allows the definition of *variables*.

7.2 Definition of variables

Variables are declared in the fifth (and last) section of an SDF definition. The variables section consists of a list of declarations of variables together with their sort. Each declaration defines a *naming scheme* for variables and may thus declare an unlimited number of variables. A naming scheme is an arbitrary regular expression like the ones allowed in a lexical function declaration, except that we do *not* allow sorts in it. The sort of a variable is a basic sort that may be contained in a list construct with or without separators.

We first describe the overall effect of variables on the derived BNF grammar, derived signature, and mapping from parse trees to abstract syntax trees. Next, we discuss the treatment of variables of the predefined sort `CHAR`.

Variables lead to additional rules in the derived regular grammar and the derived BNF grammar. For each variable declaration of the form

naming-scheme `->` S,

the rule

`<`S`-VAR> =` *naming-scheme*

is added to the derived regular grammar. Note that S may be either a basic sort or a list sort. Now there are two cases:

```
sorts ID DECL TYPE
lexical syntax
  [a-z]+                     -> ID
  [ \n]                      -> LAYOUT
context-free syntax
  decl { ID ","}+ ":" TYPE -> DECL
  integer                    -> TYPE
  real                       -> TYPE
variables
  Id       -> ID
  Type     -> TYPE
  Ids0     -> {ID ","}*
  Ids1     -> {ID ","}+
```

Figure 36: Declarations extended with variables.

1. If the declaration has the form *naming-scheme* `->` S with S a basic sort, add the rule

   `<`S`> ::= <`S`-VAR>`

   to the derived BNF grammar;

2. If the declaration has the form *naming-scheme* `->` $\{S\ \mathit{sep}\}\otimes$, the rule

   `<`S`-in-`*sep*`-list> ::= <`$\{S\ \mathit{sep}\}\otimes$`-VAR>`

   is added to the derived BNF grammar. In principle, lists declared as $\{S\ \mathit{sep}\}\otimes$ may contain variables of type S, $\{S\ \mathit{sep}\}$`*` or $\{S\ \mathit{sep}\}$`+` as elements. A subtle point arises when a list declared as $\{S\ \mathit{sep}\}$`+` contains only variables of type $\{S\ \mathit{sep}\}$`*`. In that case, the list may become empty after substitution, and this is clearly incorrect in view of the type of the original list. Therefore, we impose the restriction that a variable of type $\{S\ \mathit{sep}\}$`*` may only occur in a list of type $\{S\ \mathit{sep}\}$`+` if that list contains at least one other element of type S or $\{S\ \mathit{sep}\}$`+`. This restriction is not encoded in the derived BNF grammar, but it forms an additional constraint on it.

In the mapping from parse trees to abstract syntax trees, variables are mapped on corresponding variables in the derived signature. A variable declaration of the form

naming-scheme `->` S,

will lead to variables named S`-name1`, S`-name2`, etc., where `name1` and `name2` are strings matched by the regular expression *naming-scheme*. This leads to rules R6, G6, and M6 in Chapter 8.

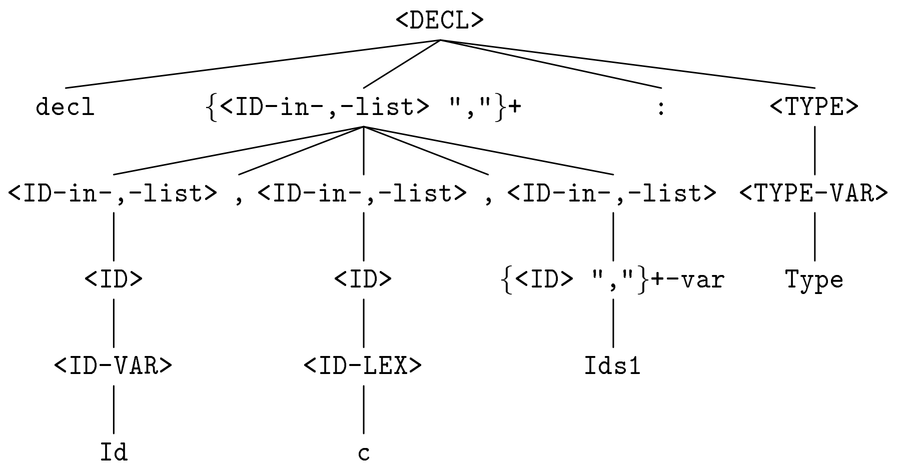


Figure 37: Parse tree for `decl Id, c, Ids1 :  Type`

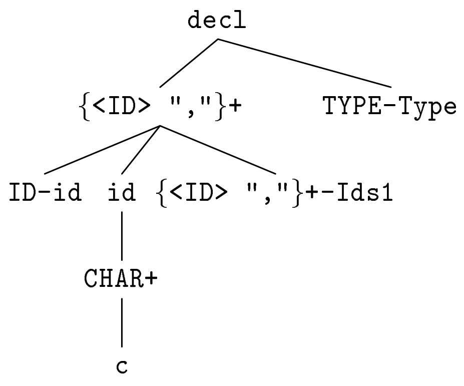


Figure 38: Abstract syntax tree for `decl Id, c, Ids1 :  Type`.

Recall that such variables are implicitly defined by a signature (see Section 2.3). Also recall from Section 4.9 that the Prefer Variables rule gives precedence to variables over the literals of the context-free syntax and over other tokens defined in the lexical syntax.

In Figure 36 we have added variables to the declaration language introduced earlier in Figure 21. The derived regular grammar is:

```
[3]  <ID-LEX>        = ("a" | ... | "z")+
[4]  <LAYOUT-LEX>    = " " | "\n"
[6,7,8]
     <LITERAL-LEX>   = "decl" | "," | ":" | "integer" | "real"
[10] <ID-VAR>        = "Id"
[11] <TYPE-VAR>      = "Type"
[12] <{ID ","}*-VAR> = "Ids0"
[13] <{ID ","}+-VAR> = "Ids1"
```

The derived BNF grammar is:

```
[3,10]
     <ID>           ::= <ID-LEX> | <ID-VAR>
[6]  <DECL>         ::= "decl" { <ID-in-,-list> "," }+ ":" <TYPE>
[6,12,13]
     <ID-in-,-list> ::= <ID> | <{ID ","}*-VAR> | <{ID ","}+-VAR>
[7,8,11]
     <TYPE>         ::= "integer" | "real" | <TYPE-VAR>
```

The derived signature is:

```
     sorts
[1]    CHAR ID DECL TYPE
     functions
       ...Declarations of constants for all characters...
[3]  id     : CHAR+            -> ID
[6]  decl   : {ID ","}+ # TYPE -> DECL
[7]  integer:                  -> TYPE
[8]  real   :                  -> TYPE
```

Figures 37 and 38 show the parse tree and the corresponding abstract syntax tree for the sentence `decl Id, c, Ids1 :  Type`, which contains occurrences of the variables `Id`, `Ids1` and `Type`. Note that variables, unlike lexical items, do not have any further structure in the abstract syntax tree.

Now we turn our attention to the special case of variables of the predefined sort `CHAR`. Apart from the standard treatment of variables given above, some additional steps are necessary:

- Add to the derived regular grammar the symbol `<CHAR-LEX>` and associate with it a regular expression that enumerates all characters in the alphabet.

- Add to the derived BNF grammar the non-terminal `<CHAR>` as well as the rules

```
1.  sorts ID DECL TYPE
2.  lexical syntax
3.    [a-z]+                  -> ID
4.    [ \n]                   -> LAYOUT
5.  context-free syntax
6.    decl { ID ","}+ ":" TYPE -> DECL
7.    integer                 -> TYPE
8.    real                    -> TYPE
9.  variables
10.   Id      -> ID
11.   Type    -> TYPE
12.   Ids0    -> {ID ","}*
13.   Ids1    -> {ID ","}+
14.   Char    -> CHAR
15.   Chars   -> CHAR+
```

Figure 39: Declarations further extended with variables of sort `CHAR`

`<CHAR> ::= <CHAR-LEX>`, and
`<CHAR-in--list> ::= <CHAR>`.

- For each lexical sort S add a rule of the form

  `<`S`> ::= "`s`" "(" <CHAR-in--list>+ ")"`,

  which defines a concrete representation for the lexical constructor function associated with S.

This leads to rules R2 and G7 in Chapter 8.

The effect of adding variables over the predefined sort `CHAR` is shown in Figure 39. The only difference with the previous example is the addition of the variables `Char` and `Chars`. The derived regular grammar now becomes:

```
[3]  <ID-LEX>        = ("a" | ... | "z")+
[4]  <LAYOUT-LEX>    = " " | "\n"
[6,7,8]
     <LITERAL-LEX>   = "decl" | "," | ":" | "integer" | "real"
[10] <ID-VAR>        = "Id"
[11] <TYPE-VAR>      = "Type"
[12] <{ID ","}*-VAR> = "Ids0"
[13] <{ID ","}+-VAR> = "Ids1"

[14,15]
     <CHAR-LEX>      = "a" | ... | "z" | "0" | ... | "9" | ...
[14] <CHAR-VAR>      = "Char"
[15] <CHAR+-VAR>     = "Chars"
```

The derived BNF grammar is:

```
[3,10,14,15]
     <ID>            ::= <ID-LEX> | <ID-VAR> |
                         "id" "(" <CHAR-in--list>+ ")"
[6]  <DECL>          ::= "decl" { <ID-in-,-list> "," }+ ":" <TYPE>
[6,12,13]
     <ID-in-,-list>  ::= <ID> | <{ID ","}+-VAR> | <{ID ","}*-VAR>
[7,8,11]
     <TYPE>          ::= "integer" | "real" | <TYPE-VAR>
[14,15]
     <CHAR>          ::= <CHAR-LEX> | <CHAR-VAR>
[14,15]
     <CHAR-in--list> ::= <CHAR> | <CHAR+-VAR> | <CHAR*-VAR>
```

The derived signature is:

```
     sorts
[1]    CHAR ID DECL TYPE
     functions
       ...Declarations of constants for all characters...
[3]    id     : CHAR+            -> ID
[6]    decl   : {ID ","}+ # TYPE -> DECL
[7]    integer:                  -> TYPE
[8]    real   :                  -> TYPE
```

Figures 40 and 41 show the parse tree and corresponding abstract syntax tree for the sentence `decl ab, id(Char y Chars) :  Type`.

7.3 Substitution of variables

Is it possible to define a notion of string substitution for variables in concrete sentences that is directly related to variable substitution in the corresponding abstract syntax trees as defined in Section 2.3? Given a string s containing a marked occurrence of a variable x of sort A and a string s' of sort $A' \leq A$, we want to define $s[x := s']$ as the syntactically correct string in which the marked occurrence of x has been replaced by s' in such a way that

$$\mathbf{abs}(s[x := s']) = \mathbf{abs}(s)[A\text{-}x := \mathbf{abs}(s')]$$

where $\mathbf{abs}(s)$ is the set of abstract syntax trees of s (cf. Figure 5), A-x is the variable of sort A that corresponds to x in the derived signature, and the :=-operator in the right-hand side is the substitution operator on abstract syntax trees defined in Section 2.3. For the above requirement to be meaningful we assume that s, s' and $s[x := s']$ have only a *single* abstract syntax tree.

Even if all three abstract syntax trees involved are unique, the :=-operator on strings cannot be as simple as that on abstract syntax trees for two reasons:

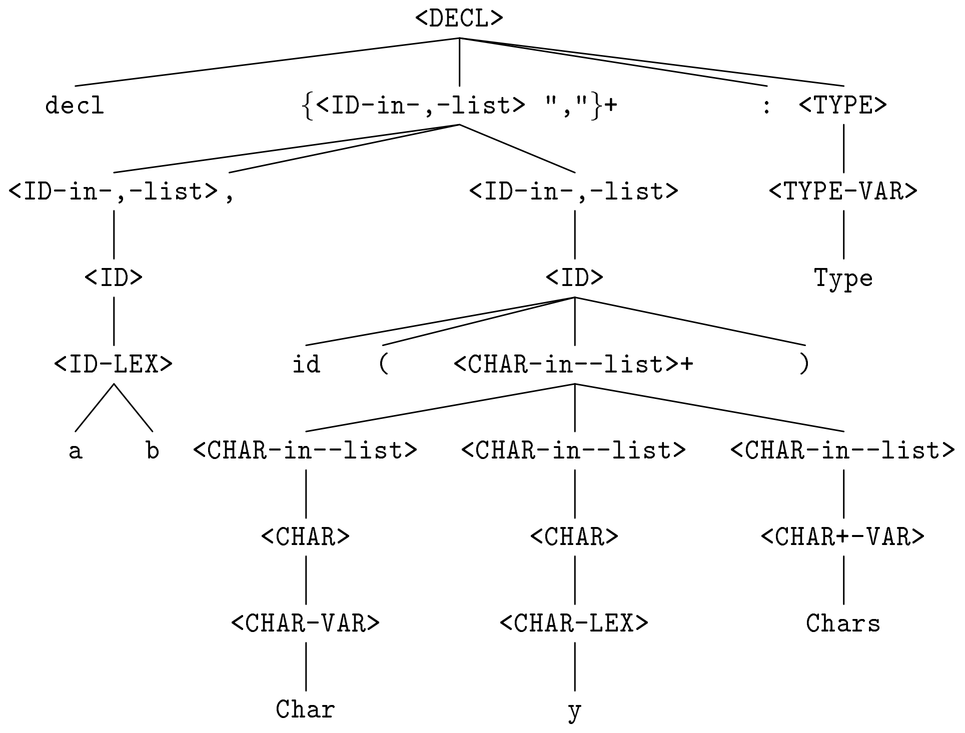


Figure 40: Parse tree for `decl ab, id(Char y Chars) : Type`

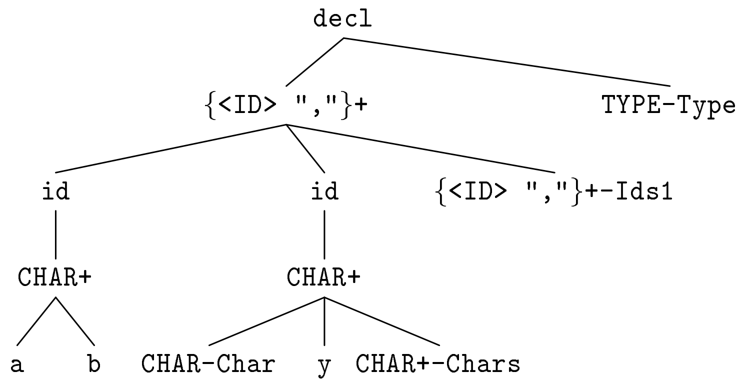


Figure 41: Abstract syntax tree for `decl ab, id(Char y Chars) : Type`

- To ensure that **abs** distributes over :=, as we require, it will sometimes be necessary to enclose s' in brackets before the actual string substitution is performed.
- If x is a list variable occurring in a list with non-empty separators and s' is the empty string, substitution of s' for x may require removal of one of the list separators adjacent to x.
- It will sometimes be necessary to insert a layout symbol before and after the substituted string to avoid differences in the lexical interpretation of the string before and after substitution.

We now discuss the first two points in more detail and assume that the third point can always be resolved. We call a string substitution $s[x := s']$ *improper* if the abstract syntax tree obtained after tree substitution is not equal to the tree obtained after performing the string substitution and reparsing the resulting string, or, in other words, if

$$\mathbf{abs}(s[x := s']) \neq \mathbf{abs}(s)[A\text{-}x := abs(s')].$$

We first define $prot(s')$, the *protected* version of s' in the substitution $s[x := s']$. There are two cases:

- If the substitution $s[x := s']$ is improper, define $prot(s')$ as the string $open\ s'\ close$, where `"`$open$`"` A' `"`$close$`"` `->` A' is an arbitrary bracket function for the sort A'. Such a function always exists (cf. Section 5.4).
- Otherwise, $prot(s') = s'$.

Next, we define $s[x := s']$ as the string obtained by replacing a substring z of s by $prot(s')$, where z is defined as follows:

- If x is a variable of sort $\{S\ \ sep\}$`*`, with *sep* not empty, that appears as element of a list, and t' is the empty list distinguish three cases:
  - if x is the only element of the list, z is the marked occurrence of x;
  - otherwise, if x is the last element of the list, z is the marked occurrence of x together with the separator *sep* preceding it and the layout in between (if any);
  - otherwise, z is the marked occurrence of x together with the separator *sep* following it and the layout in between (if any).
- In all other cases, z is the marked occurrence of x.

We give some examples. Continuing the example giving in Figure 36, let

s = `decl Ids1, c, Id :  Type`, with corresponding abstract syntax tree t,
x = `Ids1`, and
s' = `a, b`, with corresponding tree t'.

The value of the string substitution s[`Ids1` := s'] is `decl a, b, c, Id : Type` with corresponding abstract syntax tree t[{`ID ","`}`+-Ids1`:= t'].

As a second example, let

> s = `decl Ids0, c, Id : Type`, with corresponding tree t,
> x = `Ids0`, and
> $s' = \epsilon$, with corresponding tree t'.

The value of the string substitution s[`Ids0` := s'] is `decl c, Id : Type` with corresponding abstract syntax tree t[{`ID`","} $*$ $-$`Ids0` := t']. Note the elimination of the separator `","` following `Ids0`.

Next, consider the arithmetic expressions defined in Figure 25. We add a declaration

```
11. variables
12.   Exp -> E
```

for the variable `Exp` to it. First look at

> s = `Exp + 4`, with corresponding tree t,
> x = `Exp`, and
> s' = `2 * 3`, with corresponding tree t'.

The value of s[`Exp` := s'] is `2 * 3 + 4` with abstract syntax tree t[`E-Exp`:= t']. Next, look at

> s = `Exp * 4`, with corresponding tree t,
> x = `Exp`, and
> s' = `2 + 3`, with corresponding tree t'.

In this case, brackets have to be inserted since the function at the root of s' (i.e. `"+"`) has a lower priority than the function at the parent node of `Exp` (i.e. `"*"`). Therefore, the value of s[`Exp` := s'] becomes `(2 + 3) * 4`. Clearly, `2 + 3 * 4` would be an incorrect value for s[`Exp` := s'], since t[`E-Exp`:= t'] is not an abstract syntax tree for it.

A final example will illustrate the effect of the second phase (i.e., application of the multiset ordering) of the selection of parse trees based on the priority declarations. Consider the example given in Figure 34, after adding the declaration

```
10. variables
11.   Exp -> E
```

for the variable `Exp` to it. Now look at

> s = `a sub Exp`, with corresponding tree t,
> x = `Exp`, and
> s' = `a sup a`, with corresponding tree t'.

To block the higher priority but improper parse using the `E sub E sup -> E` function, brackets have to be inserted and the value of s[`Exp` := s'] becomes `a sub {a sup a}` with abstract syntax tree t[`E-Exp`:= t'].

7.4 Discussion

Two problems have not yet been solved very satisfactorily in the above definition of variables. First, a naive implementation of substitution would have to reparse the string after substitution to determine whether this substitution was proper or not. We expect that *sufficient* conditions can be found to determine this and that an implementation based on these conditions can correctly, and efficiently, implement substitution. Such an implementation will, of course, not insert a *minimal* number of brackets.

Second, some comments are in order regarding our uniform treatment of variables and "holes" (also known as "meta-variables"). Clearly, SDF variables allow a general description of holes, without fixing a particular syntax for them. However, the implementation of a syntax-directed editor is considerably simplified when certain assumptions about the syntax of holes can be made. Therefore, we take the point of view that the syntax-directed editor may automatically extend a given SDF definition with standard declarations for variables for each sort. In our current implementation (Generic Syntax-directed Editor, see Chapter 10), for each declared sort S, a variable declaration of the form `"<`S`>" ->` S is generated. These variables are the *only* variables that can be used during editing and they may not be used otherwise.

8 Summary of SDF

8.1 Deriving grammars, signature and parse-tree-to-abstract-tree mapping

We can now give the final version of the definitions of derived regular grammar, derived BNF grammar, derived signature, and mapping from parse trees to abstract syntax trees.

8.1.1 Derived regular grammar

The derived regular grammar is obtained as follows:

R1 For each sort L (including the predefined sorts `LAYOUT`, `IGNORE`, and `REJECT`) that appears as the result sort of a lexical function, introduce the lexical non-terminal `<`L`-LEX>` and associate a regular expression with it consisting of the *or* of all left-hand sides (translated using rules R3, and R4) of lexical function declarations for sort L. Replace sort names appearing in these left-hand sides by the corresponding lexical non-terminals defined before.

R2 If the variables section contains declarations for variables of the predefined sort `CHAR`, introduce the (reserved) lexical non-terminal `<CHAR-LEX>` and associate a regular expression with it that enumerates all characters in the alphabet.

R3 (Negated) character classes are replaced by an *or* of all characters in the class.

R4 Repetitions are mapped onto the corresponding repetition operators in the regular grammar.

R5 Each literal symbol occurring in a function definition in the context-free syntax is added as alternative to the (reserved) lexical non-terminal `<LITERAL-LEX>`.

R6 For each variable declaration of the form

> *naming-scheme* `->` *S*

where *naming-scheme* is a regular expression, and *S* may be either a simple sort or a list sort, add a regular expression of the form

> `<`*S*`-VAR> =` *naming-scheme*

to the derived regular grammar.

8.1.2 Derived BNF grammar

In the following definition we will frequently say that a non-terminal or rule is "added to the derived grammar". In those cases we always mean that the non-terminal or rule is added *unless it is already defined in the derived grammar*. The derived BNF grammar is obtained as follows:

G1 Introduce a non-terminal `<`*S*`>` for each sort *S*.

G2 The terminals are both the literal symbols appearing in context-free function declarations as well as the lexical non-terminals of the derived regular grammar, except `<LAYOUT-LEX>`, `<IGNORE-LEX>`, and `<REJECT-LEX>` and `<LITERAL-LEX>`.

G3 For each sort *L* that appears as the result sort of a lexical function add a rule of the form:

> `<`*L*`> ::= <`*L*`-LEX>`,

where `<`*L*`-LEX>` is a lexical non-terminal introduced as terminal in G2.

G4 For each list construct of the form $\{S\ \ sep\}\otimes$ appearing in a context-free function declaration or a variable declaration, introduce a non-terminal `<`*S*`-in-`*sep*`-list>` and add a rule of the form:

> `<`*S*`-in-`*sep*`-list> ::= <`*S*`>`.

G5 For each context-free function declaration add rules obtained by exchanging the left-hand side and the right-hand side of the function declaration. Translate each occurrence of a sort into the corresponding non-terminal as introduced in G1, and translate each occurrence of a list construct in the function declaration into the corresponding list construct in the grammar, i.e., a list of the form $\{S\ sep\}\otimes$ is translated to `{<`S`-in-`sep`-list>` $sep\}\otimes$, where `<`S`-in-`sep`-list>` is the non-terminal introduced in G4.

G6 For each variable declaration distinguish two cases:

1. If the declaration has the form *naming-scheme* `->` S, with S a basic sort, add the rule

   `<`S`> ::= <`S`-VAR>`,

   where `<`S`-VAR>` is a lexical non-terminal of the derived regular grammar (R6).
2. If the declaration has the form *naming-scheme* `->` $\{S\ sep\}\otimes$, add the rule:

   `<`S`-in-`sep`-list> ::= <`$\{S\ sep\}\otimes$`-VAR>`

   where `<`$\{S\ sep\}\otimes$`-VAR>` is a lexical non-terminal of the derived regular grammar (R6).

G7 If the variables section contains declarations using the predefined sort `CHAR` do the following:

1. Add the rules

   `<CHAR> ::= <CHAR-LEX>`, and
   `<CHAR-in--list> ::= <CHAR>`,

   where `<CHAR-LEX>` is a lexical non-terminal introduced as terminal symbol by G2.
2. For each sort $S \notin \{$`LAYOUT`, `IGNORE`, `REJECT`$\}$ that appears as the result sort of a lexical function, add a rule of the form

   `<`S`> ::= "`s`" "(" <CHAR-in--list>+ ")"`.

   This rule defines the concrete representation for the lexical constructor function associated with S (see S5 below).

8.1.3 Derived signature

The derived signature has the form:

S1 A declaration for the predefined sort `CHAR`.

S2 Declarations for all sorts declared in the SDF definition.

S3 Subsort declarations for all chain functions in the context-free syntax.

S4 Declarations of constants of sort `CHAR` for all characters in the alphabet.

S5 Declarations for *lexical constructor functions* of the form

   l `:  CHAR+ ->` L

for each sort L that appears as result sort of a lexical function, where l is the name L written in lower case letters.

S6 Function declarations for each function declared in the context-free syntax except chain functions and functions with the bracket attribute: for each such function some *unique* new name is generated (in a manner *not* prescribed by SDF) and the types of its arguments and of its result are obtained from the sort names appearing in the original function definition. List constructs of the form $\{S\ \ sep\}\otimes$ are translated to corresponding lists in the signature, i.e., to $\{S\ \ sep\}\otimes$. If the attribute `assoc` is attached to the original function declaration, it is inherited by the function declaration in the derived signature.

Recall that variables are implicitly defined by a signature (Section 2.3), therefore there is no explicit translation from the variables section of an SDF definition to its derived signature (also see M6 in Section 8.1.4).

8.1.4 Mapping from parse trees to abstract syntax trees

The mapping from parse trees to abstract syntax trees is defined as follows. Let P be a parse tree:

M1 If the root of P is labeled with a non-terminal and has a single child C whose root is labeled with lexical non-terminal `<`L`-LEX>`, with $L \neq$ `LITERAL`, the result of transforming P is an abstract syntax tree whose root is labeled with the lexical constructor function for sort L and whose single child is the result of transforming C (by using rule M2).

M2 If the root of P is labeled with lexical token `<`L`-LEX>`, with $L \neq$ `LITERAL`, its children form a list C of characters. Such a P is transformed into an abstract syntax tree whose root is labeled with `CHAR+` and with the characters in the list C as children.

M3 If the root of P is labeled with $\{S\ \ sep\}\otimes$, there are two cases:

1. *sep* is empty: let $C_1, ..., C_k$ be the k children of the root of P; or
2. *sep* is not empty, the root of P has $2k-1$ children and let $C_1, C_3, ..., C_{2k-1}$ be the odd numbered children (i.e. the children corresponding to list elements and not to separators). Let A be the list of abstract syntax trees obtained by recursively transforming the trees C_i. The result of transforming P is a tree whose root is labeled with $\{S\ \ sep\}\otimes$ and that has the elements of A as children.

M4 If the root of P is labeled with a non-terminal and corresponds to a chain function, the result of transforming P is the transformation of the single child of its root.

M5 If the root of P is labeled with a non-terminal and corresponds to a function with the bracket attribute, the result of transforming P is the transformation of the second child of its root, i.e., the only child labeled with a non-terminal.

M6 If the root of P is labeled with lexical token `<`L`-VAR>`, it must have a list consisting of characters $C_1, ..., C_k$ as its single child. Such a P is transformed into the corresponding variable L-$C_1...C_k$ in the derived signature.

M7 If the root of P is labeled with `<`S`-in-sep-list>`, the result of transforming P is the transformation of its single child.

M8 Otherwise, the root of P is labeled with a non-terminal and P is transformed as follows:

1. All children of the root of P that are labeled with a non-terminal are transformed recursively. This gives a (possibly empty) list A of abstract syntax trees.
2. The root of P together with its children correspond to the application of some rule R in the derived BNF grammar and some function F in the derived signature. P is now transformed into an abstract syntax tree whose root is labeled with F and with the elements of A as children.

8.2 Static constraints on SDF definitions

All constraints on SDF definitions are now listed for ease of reference (some of them were already mentioned in the preceding Chapters 3, 4, 5 and 6).

Sorts

- Multiple declarations of sorts are forbidden.
- The names `LAYOUT`, `IGNORE`, `REJECT` and `CHAR` are predefined and cannot be used in a sort declaration.
- The sorts `LAYOUT`, `IGNORE`, `REJECT` may only occur as result sort of a lexical function.
- The sort `CHAR` may only be used as result sort of variable declarations.
- All sort names used in lexical or context-free functions, or variable declarations should be declared.
- Each sort should occur as result sort of at least one (lexical or context-free) function declaration.

Lexical syntax

- Multiple declarations of lexical functions are forbidden.
- All sorts used in lexical function declarations should be declared.

- Each sort occurring in the left-hand side of a lexical function declaration should occur as result sort in at least one other lexical function declaration.
- No cyclic dependency may exist between lexical function declarations.
- Each result sort (except `LAYOUT`, `IGNORE`, `REJECT`) should occur in the left-hand side of at least one (lexical or context-free) function declaration.
- Character classes should satisfy the restrictions given in Section 4.3.

Context-free syntax

- Multiple declarations of context-free functions are forbidden (even if they have different attributes).
- All sorts used in context-free function declarations should be declared.
- Each sort occurring in the left-hand side of a context-free function declaration should occur as result sort in at least one other (lexical or context-free) function declaration.
- No cyclic dependency may exist between chain functions.
- The attribute `bracket` is only allowed for functions of the form "*open*" *S* "*close*" `->` *S* (see Section 5.4).
- The associativity attributes `left`, `right`, `assoc`, and `non-assoc` are only allowed for binary functions of the form *S* "*op*" *S* `->` *S* (see Section 6.2).
- A suitable bracket function should be declared for the argument and result sorts of all context-free functions that either have an associativity attribute or occur in priority or group associativity declarations.

Priorities

- All functions occurring in a priority declaration should be declared in the context-free syntax.
- The group associativity attributes `left`, `right`, and `non-assoc` are only allowed for binary functions of the form *S* "*op*" *S* `->` *S* (see Section 6.2).
- Chain and bracket functions are not allowed in priority declarations.
- No cyclic dependency may exist between context-free functions due to priority declarations.

Variables

- All result sorts (except `CHAR`) occurring in variable declarations should be declared.
- All result sorts (except `CHAR`) occurring in variable declarations should occur as result sort in the context-free syntax.

9 Attaching semantics to SDF definitions

Having completed the definition of SDF itself, we have now at our disposal a powerful method for transforming concrete strings into (sets of) abstract terms. This is sufficient to generate a syntax-directed editor from a given SDF definition, and, as explained in the introduction, it can also be used to couple semantics to it by means of abstract syntax tree matching[9].

In Section 9.1 we will show SDF can be combined with algebraic semantics. In Section 9.2 we introduce modularity constructs. In both cases, ASF [BHK89, Chapter 1] will be used as semantic specification formalism.

9.1 Attaching algebraic semantics to SDF

In its simplest form, a module in the algebraic specification formalism ASF consists of a signature, declaring sorts and functions, and a set of conditional equations over this signature. The functions declared in such a module have a fixed syntax. We will now combine ASF with SDF (obtaining ASF+SDF) in the following manner:

- Replace the signature in the ASF module by an SDF definition (this defines a derived signature as well as concrete notation for all functions in the signature);
- Replace equations written in abstract form by equations written in concrete form (see below).

The result is a formalism with completely user-definable syntax.

In its simplest form, an ASF equation is described by the grammar rule

```
<equation> ::= <tag> <term> "=" <term>
```

where `<tag>` is a label identifying the equation and `<term>` defines all well-formed terms (in prefix form).The two terms appearing in an equation should be of the same sort. An example of an ASF equation is:

```
[B1] or(true, false) = true.
```

Now, the key idea is to replace the (abstract) terms in the above definition of equations by their concrete counterpart as defined by an SDF definition. For each specification *SPEC*, this can be achieved by using the following, parameterized, grammar rule `<equation>[`*SPEC*`]` for equations:

```
<equation>[SPEC] ::=
<tag> <S_1> "=" <S_1> |
...  |
<tag> <S_n> "=" <S_n>
```

where $S_1, ..., S_n$ are the sorts declared in the (SDF part of the) specification *SPEC*, and <S_1>, ...,<S_n> are the non-terminals corresponding to these sorts in the derived BNF-grammar. Each concrete equation

[9]Lists, subsorts and associative functions should be taken care of by the matching process.

```
sorts BOOL                            (SDF definition)
lexical syntax
  [ \n]           -> LAYOUT
context-free syntax
  true            -> BOOL
  false           -> BOOL
  BOOL "&" BOOL   -> BOOL {assoc}
  BOOL "v" BOOL   -> BOOL {assoc}
  "!" BOOL        -> BOOL
  "(" BOOL ")"    -> BOOL {bracket}
priorities
  "!" BOOL -> BOOL > BOOL "&" BOOL -> BOOL >
  BOOL "v" BOOL -> BOOL
variables
  B [0-9]*        -> BOOL

equations                             (Semantic rules)
  true & true     = true
  true & false    = false
  false & true    = false
  false & false   = false
  B v true        = true
  B v false       = B
  B1 v B2         = B2 v B1
  ! true          = false
  ! false         = true
```

Figure 42: `Booleans`: a specification of the Boolean data type.

```
1.  sorts ID PAIR TABLE                    (SDF definition)
2.  lexical syntax
3.    [a-z]+            -> ID
4.    [ \n]             -> LAYOUT
5.  context-free syntax
6.    "(" ID ":" ID ")"  -> PAIR
7.    empty-table        -> TABLE
8.    PAIR "@" TABLE     -> TABLE
9.    lookup ID in TABLE -> ID
10. variables
11.   Id "'"*            -> ID
12.   T                  -> TABLE

13. equations                              (Semantic rules)
14.   lookup Id in (Id : Id') @ T = Id'

15.             Id != Id'
      ==============================================
      lookup Id in (Id' : Id'') @ T = lookup Id in T
```

Figure 43: `Tables`: specification of a table data type.

[1] u = v

with u and v strings, can now be transformed into the abstract equation

[1'] **abs**(u) = **abs**(v)

provided that

- u and v are both syntactically correct strings;
- both u and v are unambiguous (alternatively, one could impose the weaker condition that the sets of abstract syntax trees for u and v both contain only a single tree of the same sort).

This same mapping can be carried out for conditional equations in the specification. The grammar to be used for parsing each equation of the specification *SPEC* thus consists of:

- an instance of the parameterized rule `<equation>[`*SPEC*`]` given above;
- the derived grammar for the SDF part of *SPEC*.

The example in Figure 42 will illustrate this method. The derived regular grammar is:

```
[3]  <LAYOUT-LEX>  = " " | "\n"
[5,6,7,8,9,10]
     <LITERAL-LEX> = "true" | "false" | "&" | "v" | "!" |
                     "(" | ")"
[14] <BOOL-VAR>    = "B" ("0" | "1" | "2" | "3" | "4" |
                            "5" | "6" | "7" | "8" | "9")*
```

The derived BNF grammar is:

```
[5,6,7,8,9,10]
     <BOOL>        ::= "true" | "false" | <BOOL> "&" <BOOL> |
                       <BOOL> "v" <BOOL> | "!" <BOOL> |
                       "(" <BOOL> ")" | <BOOL-VAR>
```

The derived signature is:

```
[1]  sorts BOOL
     functions
[5]    true :              -> BOOL
[6]    false:              -> BOOL
[7]    "&"  : BOOL # BOOL -> BOOL
[8]    "v"  : BOOL # BOOL -> BOOL
[9]    "!"  : BOOL         -> BOOL
```

Finally, the instantiation of `<equation>[Booleans]` for the example in Figure 42 is:

```
     <equation>[Booleans] ::= <tag> <BOOL> "=" <BOOL>
```

Given these ingredients it should now be obvious how equations in concrete form are mapped to equations in abstract form. Equations 17 and 20, for instance, will be mapped to

```
[17] "&"(true, false)  = false
[20] "v"(BOOL-B, true) = true
```

We conclude this section with three more examples. Figure 43 shows a definition of a table data type. It illustrates the use of a conditional equation in line 15. Figure 44 shows a definition of identifiers on which a length function is defined. Here we use character variables in combination with the (implicitly generated) lexical function `id`. Figure 45 shows a definition of natural numbers based on their decimal representation. It illustrates, among other things, the use of list variables.

9.2 Binding SDF to a modular specification formalism

Once we have explained how ASF and SDF can be combined at the level of a single module, it is relatively straightforward to cover the modularization constructs of ASF as well. The operations to be considered are:

- export/hiding of sorts and functions;

```
sorts ID NAT                            (SDF definition)
lexical syntax
  [a-z] [a-z0-9]*   -> ID
  [ \n]             -> LAYOUT
context-free syntax
  0                 -> NAT
  succ "(" NAT ")"  -> NAT
  length "(" ID ")" -> NAT
variables
  Char              -> CHAR
  Chars             -> CHAR+

equations                               (Semantic rules)
  length(id(Char))       = succ(0)
  length(id(Chars Char)) = succ(length(id(Chars)))
```

Figure 44: `Identifiers`: specification of identifiers with a length function.

- import of a module in another module;
- renaming of sorts and functions of a module;
- actualization of parameterized modules.

In ASF, the meaning of these operations is defined by means of a normalization procedure, i.e., a textual expansion procedure that eliminates all modular structure from a specification and yields a single, unstructured, specification consisting of a signature and equations. By and large, the same method can be applied to the combination of ASF and SDF. Examples of ASF+SDF specifications can be found in [Meu88, BHK89, Hen91].

10 Notes on the implementation of SDF

SDF has been implemented as part of the "ASF+SDF Meta-environment" described in [Kli91]. It contains the Generic Syntax-directed Editor (GSE) [Log88, DK90, Koo92] mentioned earlier in Section 7.4.

The SDF implementation is written in LeLisp and uses ISG (Incremental Scanner Generator [HKR87]) to generate lexical scanners from the lexical syntax of the SDF definition, and IPG (Incremental Parser Generator [HKR89]) to generate parsers from the context-free syntax of the SDF definition. Both generators are *lazy* as well as *incremental* (i.e., only those parts of the scanner and parser are generated that are actually needed, and modifications to the lexical and context-free syntax are propagated to the already generated scanner and parser, thus avoiding their complete regeneration). The underlying principles have been described in [HKR91].

Another distinctive feature of ISG and IPG is that they allow ambiguities. ISG may produce more than one interpretation for a lexeme; IPG can handle arbitrary context-free grammars and may thus produce more than one abstract syntax tree for a given input string.

11 Discussion

11.1 Basic assumptions

We briefly recall some of the main assumptions and choices underlying the design of SDF:

- The implementation of SDF does not impose restrictions on the class of acceptable grammars. This has several major advantages: it is never necessary to rewrite a given grammar to fit a particular subclass of the context-free grammars and the composition of grammars is always possible. It also means that context-free and abstract syntax bear a close resemblance to each other. As a consequence, there is little need for explicit abstract syntax tree construction rules (see next point). The other side of the picture is that grammars may become ambiguous.
- There is a fixed mapping between parse trees and abstract syntax trees. The advantages of this choice are twofold: the writer of the specification does not have to give a separate definition of the tree construction process and the user of the specification sees trees that correspond to grammar rules in a completely predictable way. Clearly, a degree of freedom has been lost here. Associating "non-standard" trees with a grammar rule can only be achieved, in our case, by computing them by means of semantic rules.
- Lexical entities are represented as strings of characters and not as trees. While the latter representation is attractive from a theoretical point of view, we opted for the former for reasons of efficiency in the SDF implementation.

11.2 Omissions

Some of the features *not* included in SDF are:

- There is no way of expressing prettyprinting formats for functions. The main reason for this omission is that adding this information would make SDF function definitions harder to read. The editors generated from SDF-definitions by the GSE editor generator mentioned in the previous Chapter do not use prettyprinting but maintain both the original text representation as well as the corresponding abstract syntax tree.
- There is no notion of optional constructs in context-free functions.
- There is no alternative operator.

```
1.  sorts DIGIT NAT NAT-LIST              (SDF definition)
2.  lexical syntax
3.    [0-9]                  -> DIGIT
4.    DIGIT+                 -> NAT
5.  context-free syntax
6.    NAT "+" NAT            -> NAT {left}
7.    NAT "*" NAT            -> NAT {left}
8.    {NAT ","}*             -> NAT-LIST
9.    move DIGIT in NAT-LIST -> NAT-LIST
10. priorities
11.   NAT "*" NAT -> NAT > NAT "+" NAT -> NAT
12. variables
13.   n [0-9]*               -> NAT
14.   nats [0-9]*            -> {NAT ","}*
15.   c [0-9]*               -> CHAR
16.   x [0-9]*               -> CHAR*
17.   y [0-9]*               -> CHAR+
18. equations                             (Semantic rules)
19.   move 0 in nats                            = nats
20.   move 1 in n1,nats                         = nats
21.   move 2 in n1,n2,nats                      = nats
22.   move 3 in n1,n2,n3,nats                   = nats
23.   move 4 in n1,n2,n3,n4,nats                = nats
24.   move 5 in n1,n2,n3,n4,n5,nats             = nats
25.   move 6 in n1,n2,n3,n4,n5,n6,nats          = nats
26.   move 7 in n1,n2,n3,n4,n5,n6,n7,nats       = nats
27.   move 8 in n1,n2,n3,n4,n5,n6,n7,n8,nats    = nats
28.   move 9 in n1,n2,n3,n4,n5,n6,n7,n8,n9,nats = nats
29.   nat(0 y) = nat(y)
30.   move digit(c1) in 0,1,2,3,4,5,6,7,8,9,
                     10,11,12,13,14,15,16,17,18 = nats1,
                   move digit(c2) in nats1 = n, nats2
      ====================================================
                        nat(c1) + nat(c2) = n)

31.   nat(c1) + nat(c2) = nat(x c),
      nat(0 x1) + nat(0 x2) + nat(0 x) = nat(y)
      =========================================
      nat(x1 c1) + nat(x2 c2) = nat(y c))
32.   n * 0   = 0          33.   n * 1   = n
34.   n * 2   = n + n      35.   n * 3   = n + n * 2
36.   n * 4   = n + n * 3  37.   n * 5   = n + n * 4
38.   n * 6   = n + n * 5  39.   n * 7   = n + n * 6
40.   n * 8   = n + n * 7  41.   n * 9   = n + n * 8

42.            nat(y1) * nat(y2) = nat(y)
      =================================================
      nat(y1) * nat(y2 c) = nat(y 0) + nat(y1) * nat(c)
```

Figure 45: `Naturals`: specification of decimal natural numbers.

- Repetitions (in lexical syntax as well as in the context-free syntax) cannot be restricted to a fixed number of repetitions. This has to be expressed by means of semantic rules.

11.3 Limitations of lexical syntax

The Prefer Literals rule (Section 5.2) gives precedence to literals in the context-free syntax over other lexical tokens. As a consequence, the sentence `if := while + then` will not be accepted as a legal statement in the example in Figure 46, since the tokens `if`, `while`, and `then` are only recognized as literals, and not also as identifiers. One, not very satisfactory, way to circumvent this limitation is shown in Figure 47: all keywords have been added explicitly as alternatives of sort `ID` to the context-free syntax. This definition has the disadvantage that identifiers (defined by a function in the lexical syntax) and keywords (defined by constants in the context-free syntax) are represented differently in the abstract syntax. This difference will manifest itself when string matching is attempted on an identifier that happens to be equal to a literal. Alternatives for solving this problem are:

- Forbid constants in the context-free syntax.
- Add exceptions to the definition of the derived signature and of the mapping from parse trees to abstract syntax trees in such a way that constants defined in the context-free syntax and lexical functions are represented in a uniform way.
- Add a mechanism to (de)activate the Prefer Literals rule.

Further research is needed to determine the best approach to this problem.

Another, general, limitation of the definition of lexical syntax in SDF, is that no "actions" (i.e., updating of counters, consulting symbol tables, etc.) can be performed when a token is being recognized. As a result, several aspects of lexical syntax are difficult to express in SDF:

- Nesting of comments (as, for instance, in C);
- The use of indentation for indicating block structure (as, for instance, in ABC and OCCAM).

Apart from a default solution in the form of attaching extra semantic rules to the SDF definition, we do not yet see a really elegant solution to this class of problems.

11.4 Related work

There have been many attempts to introduce user-definable syntax in programming languages. These ideas have led to user-definable syntax for operators in Algol68, Prolog, Snobol4 and other programming languages, and to various styles of macro-definitions (PL/I, Lisp). Around 1970, there was

```
1.  sorts ID EXP STAT
2.  lexical syntax
3.    [a-z]+            -> ID
4.    [ \n]             -> LAYOUT
5.  context-free syntax
6.    if EXP then STAT  -> STAT
7.    while EXP do STAT -> STAT
8.    ID ":=" EXP       -> STAT
9.    EXP "+" EXP       -> EXP
10.   ID                -> EXP
```

Figure 46: A simple language.

```
1.  sorts ID EXP STAT
2.  lexical syntax
3.    [a-z]+            -> ID
4.    [ \n]             -> LAYOUT
5.  context-free syntax
6.    if                -> ID
7.    then              -> ID
8.    while             -> ID
9.    do                -> ID
10.   if EXP then STAT  -> STAT
11.   while EXP do STAT -> STAT
12.   ID ":=" EXP       -> STAT
13.   EXP "+" EXP       -> EXP
14.   ID                -> EXP
```

Figure 47: Language from Figure 46 with overlapping keywords and identifiers.

much interest in so-called *extensible languages* [Iro70, Sta75, Weg70]. The aim of this line of research was to define a small base language in combination with a syntax definition formalism. New language constructs could then be added to the base language by defining their syntax and by describing their semantics in terms of the base language. For various reasons, however, the overall goal of full syntactic and semantic language extensibility has never been completely achieved. Although it does not have a syntactic extension mechanism, Smalltalk-80 may be viewed as the most successful extensible language in existence. (Smalltalk-72—a predecessor of Smalltalk-80—did support syntactic extensibility: each class had to parse the messages send to it explicitly. This feature has been replaced by a more limited scheme of keyword parameters in Smalltalk-80. This new scheme result in more readable programs and allows a more efficient implementation.)

A successful method for defining the syntax of language constructs is by means of syntax-directed translations [Iro61, ASU86]. Lex/Yacc and Metal (to be discussed below) fall into this category. The Lithe system [San82] combines syntax-directed translations with classes. Apart from the fact that its lexical syntax is fixed, Lithe has user-definable LR(1)-syntax. O'Donnell's notational specifications [O'D85] constitute an alternative to syntax-directed translations. Most closely related to our work are the user-definable distfix operators in OBJ [FGJM85] and Asspegique/Cigale [Voi86].

The following features can be used to characterize syntax definition formalisms:

- The class of acceptable grammars.
- The integration between the description of lexical and of context-free syntax.
- The class of context-free syntax/abstract syntax pairs that can be described.
- The integration between the description of context-free syntax and of abstract syntax.
- The treatment of variables.

We will restrict ourselves to some representative examples, namely Lex [LS86], Yacc [Joh86], Metal (the specification language of the Mentor system [KLMM83]), SSL (the specification language of the Synthesizer Generator [Rep82, RT89]), and the specification language of the PSG system [BS86]. In [HK89] we have already compared SDF with Lex, Yacc and Metal by studying several examples in each of these formalisms.

The syntax definitions for Lex/Yacc, Metal, and SSL are restricted to LALR(1) grammars, those for PSG to LL(1) grammars, while SDF definitions allow arbitrary context-free grammars.

In Lex/Yacc explicit token names are used for the communication between lexical and context-free syntax. A similar method is used in Metal, SSL and PSG. In SDF, there is a better integration between lexical and context-free syntax. Token names are, for instance, generated automatically.

Lex/Yacc allow the greatest freedom in the creation of abstract syntax trees (i.e., arbitrary C data structures). In Metal, abstract syntax is described by means of a signature-like notation. With each rule in the context-free grammar, a tree construction rule is associated which defines the abstract syntax tree to be constructed for strings accepted by this grammar rule. This allows the construction of arbitrary trees provided that they conform to the given signature. In SSL, the construction of abstract syntax trees is described by a separate attribute grammar. In this respect, SSL has the same expressive power as Metal. In PSG, abstract syntax is defined by a collection of class and constructor rules resembling signatures, while concrete syntax is defined by a string-to-tree transformation grammar that defines the abstract syntax tree to be associated with each string. In particular, the name of a constructor of the abstract syntax is associated with each context-free grammar rule (this limits the set of trees that can be constructed for each grammar rule). In SDF, there is a fixed correspondence between rules in the context-free syntax and rules in the abstract syntax. It is therefore impossible to associate an arbitrary tree with strings accepted by a given syntax rule.

Metal, SSL and PSG have built-in notations for variables (also called meta-variables, placeholders, or holes). SDF has no such standard convention but allows the definition of naming schemes for variables.

11.5 Future developments

The following problems remain to be solved:

- The treatment of ambiguities needs further research: (a) it is desirable to have a sufficient but not overly restrictive condition on SDF definitions that guarantees that they define a non-ambiguous language; (b) the use of type information to resolve ambiguities should be studied.

- The combination of SDF with other formalisms such as TYPOL [Des84, Kah87], Prolog, and first-order logic should be studied. Some of the problems involved are: (a) signature and term translations (necessary to map the derived signature of SDF definitions and terms over it on signatures and terms as they exist in the target specification formalism); (b) modeling of relations (SDF only provides definitions of functions).

12 Acknowledgements

Hans van Dijk suggested to retain only parse trees without priority conflicts rather than, for instance, parse trees with a minimal number of conflicts. Freek Wiedijk commented on an earlier version of this manual. The SDF definition of Pascal in Appendix B was written by Karin Vos.

References

[AJU75] A.V. Aho, S.C. Johnson, and J.D. Ullman. Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 18(8):441–452, 1975.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BS86] R. Bahlke and G. Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547– 576, 1986.

[Des84] T. Despeyroux. Executable specification of static semantics. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 215–233. Springer-Verlag, 1984.

[DK90] M.H.H. van Dijk and J.W.C. Koorn. GSE, a generic syntax-directed editor. Report CS-R9045, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1990.

[DM79] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[FGJM85] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In B. Reid, editor, *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.

[Hen91] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[HHKR89] 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.

[HK89] J. Heering and P. Klint. PICO revisited. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 359–379. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 9.

[HKKL86] 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.

[HKR87] J. Heering, P. Klint, and J. Rekers. Incremental generation of lexical scanners. Report CS-R8761, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1987. To appear in *ACM Transactions on Programming Languages and Systems*.

[HKR89] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *SIGPLAN Notices*, 24(7):179–191, 1989.

[HKR91] J. Heering, P. Klint, and J. Rekers. Lazy and incremental program generation. Report CS-R9124, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1991. To appear in *ACM Transactions on Programming Languages and Systems*.

[Iro61] E.T. Irons. A syntax-directed compiler for algol 60. *Communications of the ACM*, 4(1):51–55, 1961.

[Iro70] E.T. Irons. Experience with an extensible language. *Communications of the ACM*, 13(1):31–40, 1970.

[JGH$^+$86] W.N. Joy, S.L. Graham, Ch.B. Haley, M.K. McKusick, and P.B. Kessler. *Berkeley Pascal User's Manual.* Computer Systems Research Group, University of California, Berkeley, 1986. Version 3.1, April 1986, 4.3 Berkeley Software Distribution.

[JL82] J.-P. Jouannaud and P. Lescanne. On multiset orderings. *Information Processing Letters*, 15(2):57–63, 1982.

[Joh86] S.C. Johnson. *YACC: yet another compiler-compiler.* Bell Laboratories, 1986. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).

[Kah87] G. Kahn. Natural semantics. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[Kli91] P. Klint. A meta-environment for generating programming environments. In J.A. Bergstra and L.M.G. Feijs, editors, *Proceedings of the METEOR workshop on Methods Based on Formal Specification*, volume 490 of *Lecture Notes in Computer Science*, pages 105–124. Springer-Verlag, 1991.

[KLMM83] G. Kahn, B. Lang, B. Mélèse, and E. Morcos. Metal: a formalism to specify formalisms. *Science of Computer Programming*, 3:151–188, 1983.

[Koo92] J.W.C. Koorn. GSE: A generic text and structure editor. Report P9202, University of Amsterdam, 1992.

[Log88] M.H. Logger. An integrated text and syntax-directed editor. Report CS-R8820, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1988.

[LS86] M.E. Lesk and E. Schmidt. *LEX - A lexical analyzer generator.* Bell Laboratories, unix programmer's supplementary documents, volume 1 (ps1) edition, 1986.

[Meu88] E.A. van der Meulen. Algebraic specification of a compiler for a language with pointers. Report CS-R8848, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1988.

[O'D85] M.J. O'Donnell. Equational Logic as a Programming Language. Technical report, MIT Press, 1985.

[Rep82] T. Reps. Generating language-based environments. Technical report TR 82-514, Cornell University, Ithaca, 1982. Ph.D. Thesis.

[RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator: a System for Constructing Language-Based Editors.* Springer-Verlag, 1989.

[San82] D. Sandberg. LITHE: A language combining a flexible syntax and classes. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 142–145. ACM, 1982.

[Sta75] T. A. Standish. Extensibility in programming language design. *SIGPLAN Notices*, 10(7):18–21, 1975.

[Voi86] F. Voisin. CIGALE: a tool for interactive grammar construction and expression parsing. *Science of Computer Programming*, 7:61–86, 1986.

[Weg70] B. Wegbreit. *Studies in Extensible Programming Languages.* PhD thesis, Harvard, 1970. Reprinted by Garland Publishing, 1980.

A SDF in SDF

```
sorts
  Id Iterator
 EscChar C-Char CharRange  L-Char
  CharClass Literal Module Section SyntaxSection
  LexicalFunction SpecialLexId BasicLexElem LexElem
  CfFunction BareFunction CfElem Attributes
  PriorChain FunctionList
  Variable VarSort
  FunOpName ListOpName

lexical syntax
  [A-Z]                                          -> Id
  [A-Z] [A-Za-z0-9\-_]* [A-Za-z0-9]              -> Id
  [+*]                                           -> Iterator
  "\\" ~ []                                           -> EscChar
  "\\" [01] [0-7] [0-7]                               -> EscChar

  ~ [\000-\037\-\[\]\\]                               -> C-Char
  EscChar                                             -> C-Char
  C-Char                                              -> CharRange
  C-Char "-" C-Char                                   -> CharRange
  "[" CharRange* "]"                                  -> CharClass

  ~ [\000-\037"\\]                                    -> L-Char
  EscChar                                             -> L-Char
  "\"" L-Char* "\""                                   -> Literal
  [a-z]                                               -> Literal
  [a-z] [A-Za-z0-9\-_]* [A-Za-z0-9]                   -> Literal

  [ \t\n]                                             -> LAYOUT
  "%%" ~[\n] * "\n"                                   -> LAYOUT
  "%" ~[\n%] + "%"                                    -> LAYOUT

context-free syntax
  "module" Id Section*                                -> Module
  "module" Id SyntaxSection*                          -> Module

  "imports" Id+                                       -> Section
  "exports" SyntaxSection+                            -> Section
  "hiddens" SyntaxSection+                            -> Section
  "sorts" Id+                                         -> SyntaxSection
  "lexical" "syntax" LexicalFunction+                 -> SyntaxSection
  "context-free" "syntax" CfFunction+                 -> SyntaxSection
  "priorities" {PriorChain ","}+                      -> SyntaxSection
  "variables" Variable+                               -> SyntaxSection

  LexElem+ "->" Id                                    -> LexicalFunction
  LexElem+ "->" SpecialLexId                          -> LexicalFunction
  "LAYOUT"                                            -> SpecialLexId
  "IGNORE"                                            -> SpecialLexId
  "REJECT"                                            -> SpecialLexId
  Id                                                  -> BasicLexElem
  Literal                                             -> BasicLexElem
  CharClass                                           -> BasicLexElem
  "~" CharClass                                       -> BasicLexElem
  BasicLexElem Iterator                               -> LexElem
  BasicLexElem                                        -> LexElem

  FunOpName CfElem* "->" Id Attributes                -> CfFunction
  Literal "(" {CfElem ","}* ")" "->" Id Attributes    -> CfFunction
  Id                                                  -> CfElem
  Literal                                             -> CfElem
  Id Iterator ListOpName                              -> CfElem
  "{" Id Literal "}" Iterator ListOpName              -> CfElem
  % EMPTY %                                           -> Attributes
  "{" {Literal ","}+ "}"                              -> Attributes
  Literal ":"                                         -> FunOpName
```

```
    % EMPTY %                                        -> FunOpName
    ":" Literal                                      -> ListOpName
    % EMPTY %                                        -> ListOpName

    FunctionList ">" {FunctionList ">"}+             -> PriorChain
    FunctionList "<" {FunctionList "<"}+             -> PriorChain
    "{" Literal ":" {BareFunction ","}+ "}"          -> PriorChain
    BareFunction                                     -> FunctionList
    "{" {BareFunction ","}+ "}"                      -> FunctionList
    "{" Literal ":" {BareFunction ","}+ "}"          -> FunctionList
    CfElem* "->" Id                                  -> BareFunction
    Literal+                                         -> BareFunction
    Literal "(" {CfElem ","}* ")" "->" Id            -> BareFunction

    LexElem+ "->" VarSort                            -> Variable
    Id                                               -> VarSort
    Id Iterator                                      -> VarSort
    "{" Id Literal "}" Iterator                      -> VarSort
    "CHAR"                                           -> VarSort
    "CHAR" Iterator                                  -> VarSort
```

The above definition gives the syntax of SDF as currently implemented in the ASF+SDF Meta-environment and contains module headers, as well as sections for defining imports, exports and hiddens. Observe that:

- The module header is *only* required in stand-alone SDF implementations and should *not* be used when writing ASF+SDF modules. In other words, the SDF-part of ASF+SDF modules has the form `Section*`.
- Compared to previous versions of SDF attributes like `left`, `right`, etc., no longer appear as explicit literals in the definition but they are all represented by the sort `Literal`. This was done to make it easier to add new attributes in the future. In addition, a non-empty list of attributes *should* now be enclosed by { and }.

B Berkeley Pascal in SDF

```
  sorts
    StarComChar Id StringElem
    CharString Base UnsignedInt SignedInt
    UnsignedReal Number OctalConst

    Program ProgHeading Block Decl LabelDecl
    ConstDecl TypeDecl VarDecl ProcDecl FuncDecl LabelList
    ProcHeading FuncHeading Pars Par
    Const Type SimpleType FileType NonfileType
    StructType Field FieldList VariantPart Variant
    Var QualifiedVar Statement Assignment
    CaseListElem Expr SetElem ActualPar
```

```
lexical syntax
  [ \t\n\r]                                    -> LAYOUT
  "{" ~[}] * "}"                               -> LAYOUT
  ~ [*]                                        -> StarComChar
  "*" ~ [)]                                    -> StarComChar
  "(*" StarComChar "*)"                        -> LAYOUT
  "#include" ~[\n] * "\n"                      -> LAYOUT

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

  "oct"                                        -> Base
  "hex"                                        -> Base

  ~ ['\n]                                      -> StringElem
  "''"                                         -> StringElem
  "'" StringElem+ "'"                          -> CharString
  "#" StringElem+ "#"                          -> CharString

  [0-7]+ [bB]                                  -> OctalConst
  [0-9]+                                       -> UnsignedInt
  UnsignedInt                                  -> SignedInt
  [+\-] UnsignedInt                            -> SignedInt

  UnsignedInt "." UnsignedInt                  -> UnsignedReal
  UnsignedInt "." UnsignedInt [eE] SignedInt   -> UnsignedReal
  UnsignedInt [eE] SignedInt                   -> UnsignedReal

  UnsignedInt                                  -> Number
  UnsignedReal                                 -> Number
  OctalConst                                   -> Number

context-free syntax
  ProgHeading Decl* Block "."                  -> Program
  Decl*                                        -> Program
  program Id "(" {Id ","}+ ")" ";"             -> ProgHeading
  begin {Statement ";"}+ end                   -> Block

  label {UnsignedInt ","}+                     -> Decl

  const {ConstDecl ";"}+ ";"                   -> Decl
  Id "=" Const                                 -> ConstDecl
  CharString                                   -> Const
  Id                                           -> Const
  Number                                       -> Const
  "+" Number                                   -> Const
  "-" Number                                   -> Const

  type {TypeDecl ";"}+ ";"                     -> Decl
  Id "=" Type                                  -> TypeDecl
  SimpleType                                   -> Type
  "^" Id                                       -> Type
  FileType                                     -> Type
  StructType                                   -> Type
```

```
packed StructType                          -> Type

Id                                         -> SimpleType
"(" {Id ","}+ ")"                          -> SimpleType
Const ".." Const                           -> SimpleType
file of NonfileType                        -> FileType
SimpleType                                 -> NonfileType
"^" Id                                     -> NonfileType
StructType                                 -> NonfileType
packed StructType                          -> NonfileType

array "[" {SimpleType ","}+ "]" of Type    -> StructType
set of SimpleType                          -> StructType
record FieldList end                       -> StructType
{Field ";"}+ VariantPart                   -> FieldList
{Id ","}+ ":" Type                         -> Field
% empty %                                  -> Field
case Id of {Variant ";"}+                  -> VariantPart
case Id ":" Id of {Variant ";"}+           -> VariantPart
% empty %                                  -> VariantPart
{Const ","}+ ":" "(" FieldList ")"         -> Variant
% empty %                                  -> Variant

var {VarDecl ";"}+ ";"                     -> Decl
{Id ","}+ ":" Type                         -> VarDecl

ProcDecl                                   -> Decl
procedure Id Pars ";"                      -> ProcHeading
ProcHeading Decl* Block ";"                -> ProcDecl
ProcHeading forward ";"                    -> ProcDecl
ProcHeading external Id ";"                -> ProcDecl
ProcHeading external ";"                   -> ProcDecl

FuncDecl                                   -> Decl
function Id Pars ":" Type ";"              -> FuncHeading
FuncHeading Decl* Block ";"                -> FuncDecl
FuncHeading forward ";"                    -> FuncDecl
FuncHeading external Id ";"                -> FuncDecl
FuncHeading external ";"                   -> FuncDecl

"(" {Par ";"}+ ")"                         -> Pars
% empty %                                  -> Pars
{Id ","}+ ":" Id                           -> Par
var {Id ","}+ ":" Id                       -> Par
procedure Id Pars                          -> Par
function Id Pars ":" Id                    -> Par

Id                                         -> Var
QualifiedVar                               -> Var
Id "[" {Expr ","}+ "]"                     -> QualifiedVar
QualifiedVar "[" {Expr ","}+ "]"           -> QualifiedVar
Id "." Id                                  -> QualifiedVar
QualifiedVar "." Id                        -> QualifiedVar
```

```
Id "^"                                        -> QualifiedVar
QualifiedVar "^"                              -> QualifiedVar

Var ":=" Expr                                 -> Assignment
Assignment                                    -> Statement
begin {Statement ";"}+ end                    -> Statement
if Expr then Statement                        -> Statement
if Expr then Statement else Statement         -> Statement
while Expr do Statement                       -> Statement
repeat {Statement ";"}+ until Expr            -> Statement
for Assignment to Expr do Statement           -> Statement
for Assignment downto Expr do Statement       -> Statement
case Expr of {CaseListElem ";"}+ end          -> Statement
with {Var ","}+ do Statement                  -> Statement
Id                                            -> Statement
Id "(" {ActualPar ","}+ ")"                   -> Statement
UnsignedInt Statement                         -> Statement
goto UnsignedInt                              -> Statement
% empty %                                     -> Statement

{Const ","}+ ":" Statement                    -> CaseListElem
% empty %                                     -> CaseListElem

Number                                        -> Expr
nil                                           -> Expr
CharString                                    -> Expr
Var                                           -> Expr
Id "(" {ActualPar ","}+ ")"                   -> Expr
not Expr                                      -> Expr
"~" Expr                                      -> Expr
"+" Expr                                      -> Expr
"-" Expr                                      -> Expr
"[" {SetElem ","}* "]"                        -> Expr
Expr "+" Expr                                 -> Expr {left}
Expr "-" Expr                                 -> Expr {left}
Expr "|" Expr                                 -> Expr {left}
Expr or Expr                                  -> Expr {left}
Expr "*" Expr                                 -> Expr {left}
Expr "&" Expr                                 -> Expr {left}
Expr "/" Expr                                 -> Expr {left}
Expr div Expr                                 -> Expr {left}
Expr mod Expr                                 -> Expr {left}
Expr and Expr                                 -> Expr {left}

Expr "=" Expr                                 -> Expr {non-assoc}
Expr ">" Expr                                 -> Expr {non-assoc}
Expr "<" Expr                                 -> Expr {non-assoc}
Expr "<>" Expr                                -> Expr {non-assoc}
Expr "<=" Expr                                -> Expr {non-assoc}
Expr ">=" Expr                                -> Expr {non-assoc}
Expr in Expr                                  -> Expr {non-assoc}
"(" Expr ")"                                  -> Expr {bracket}
```

```
  Expr                                       -> SetElem
  Expr ".." Expr                             -> SetElem

  Expr                                       -> ActualPar
  Expr ":" Expr                              -> ActualPar
  Expr ":" Expr ":" Expr                     -> ActualPar
  Expr Base                                  -> ActualPar
  Expr ":" Expr Base                         -> ActualPar


priorities
  { not Expr -> Expr, "~" Expr -> Expr,
    "+" Expr -> Expr, "-" Expr -> Expr }
  >
  { left:
    Expr "*" Expr -> Expr, Expr "&" Expr -> Expr,
    Expr "/" Expr -> Expr, Expr div Expr -> Expr,
    Expr mod Expr -> Expr, Expr and Expr -> Expr }
  >
  { left:
    Expr "+" Expr -> Expr, Expr "-" Expr -> Expr,
    Expr "|" Expr -> Expr, Expr or Expr -> Expr }
  >
  { non-assoc:
    Expr "=" Expr -> Expr, Expr ">" Expr -> Expr,
    Expr "<" Expr -> Expr, Expr "<=" Expr -> Expr,
    Expr ">=" Expr -> Expr, Expr "<>" Expr -> Expr,
    Expr in Expr -> Expr }
priorities
  if Expr then Statement else Statement -> Statement >
    if Expr then Statement -> Statement
```

Note:

- This definition of Pascal conforms to [JGH$^+$86].

```
(****
Notities
2.3. Signatures
        (* Is hier een mooiere conventie te bedenken ?? *)
        (** nog toevoegen: assoc **)
        (* P.M. de gevallen S < {S' sep}+, enz. nog bekijken *)
5.4. Chain and bracket functions
        (* cycles verbieden *)
6.2. Priority and associativity
        (* wanneer is een combinatie van attributen correct? )
6.3. Ambiguities
(**
        - assoc attribuut
        - Mogelijkheden en beperkingen:
                - disambiguering van inherent ambigue grammatica (dubbele palindromen
                - wanneer is voldoende gedisambigueerd? (in het algemeen onbeslisbaar
                - Definitie met operatoren + en * in soort OP is niet te disambiguere
**)
Algemeen
- injecties hebben semantische consequenties (welke?)
- Afkortingsregels voor functie-definities in prioriteiten.
- Definitie/Implementatie:  \ escape in literals in cfg.
- Argumenten waarom
        (a) functies geen  lijsten als output soort mogen hebben
        (b) er geen vergelijkingen over lijsten mogen zijn.

(** status variabelen <-> holes **)

*********)
```