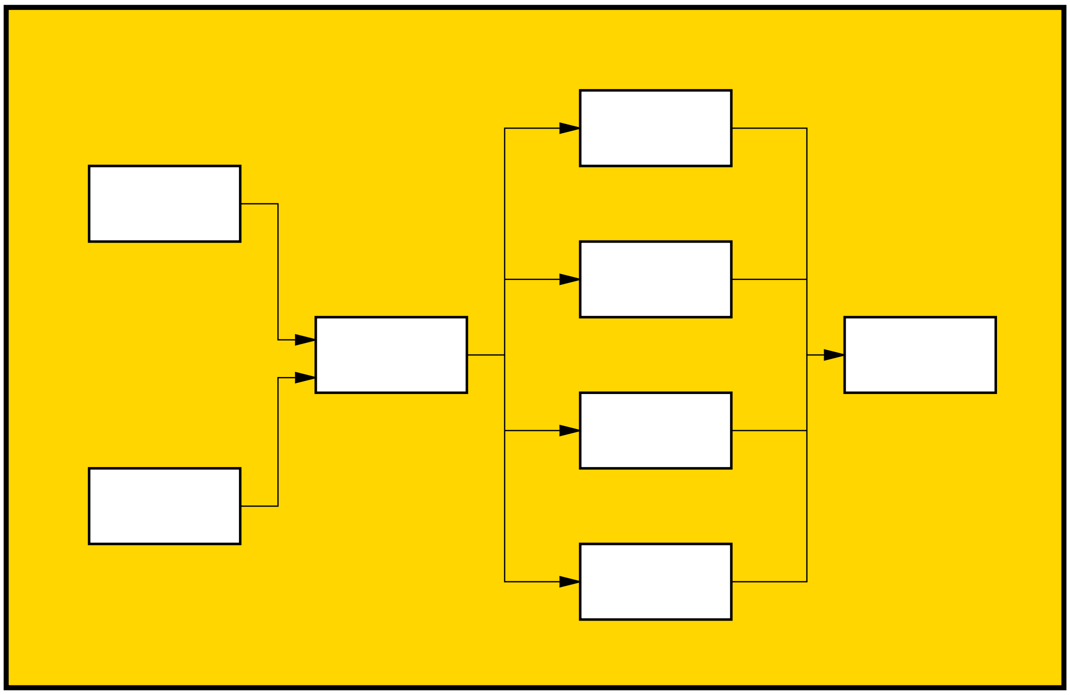
**University of Amsterdam**
*Programming Research Group*



# A Family of Syntax Definition Formalisms

Eelco Visser

Report P9706 August 1997

University of Amsterdam
Department of Computer Science
Programming Research Group

# A family of syntax definition formalisms

Eelco Visser

Report P9706 August 1997

E. Visser

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7590
e-mail: visser@wins.uva.nl

An earlier version of this report appeared in M. G. J. van den Brand et al., *ASF+SDF'95. A Workshop on Generating Tools from Algebraic Specifications*, Technical Report P9504, Programming Research Group, University of Amsterdam. pp 89–126, May 1995.

**Acknowledgements** The author thanks Arie van Deursen, Jan Heering, Tobias Kuipers, Paul Klint, Mark van den Brand, Merijn de Jonge and Alex Sellink for useful suggestions and comments on previous versions of this report.

This research was supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organisation for Scientific Research (NWO). Project 612-317-420: Incremental parser generation and context-sensitive disambiguation: a multi-disciplinary perspective.

Universiteit van Amsterdam, 1997

# Contents

# 1

# A Family of Syntax Definition Formalisms

In the next chapters we present the design and specification of a family of syntax definition formalisms. The kernel of this family of formalisms is formed by context-free grammars. A number of orthogonal extensions to the kernel is defined. Many of these extensions are defined in terms of the primitives of the kernel by means of normalization functions. This provides a framework for constructing new formalisms by adapting and extending previous ones.

Included in the family are the following extensions of context-free grammars: uniform definition of lexical and context-free syntax, variables, disambiguation by priorities, follow restrictions and reject productions, a rich set of regular expressions defined in terms of context-free productions, character classes, aliases, parameterized modules with hidden imports and renamings. The accumulation of these extensions is the syntax definition formalism SDF2.

This chapter provides an introduction to SDF2 and gives an overview of the design and specification of the family of formalisms.

## 1.1 Introduction

New programming, specification and special purpose languages are being developed continuously. Syntax definition formalisms play a crucial role in the design and implementation of new languages. Syntax definition formalisms also play a role embedded in other languages: regular expressions in edit operations, macro definitions for macro preprocessors, user definable infix or distfix operators in programming languages, grammars as signatures in algebraic specification formalisms, and documents that contain a description of their own syntax.

The core of many syntax definition formalisms is formed by context-free grammars, which are widely used in computer science since their introduction by Chomsky (1956). A context-free grammar is a set of string rewrite rules of the form $\alpha \to \mathcal{A}$ with $\alpha$ a string $\mathcal{A}_1 \ldots \mathcal{A}_n$ of zero or more symbols and $\mathcal{A}$ a symbol. A string (a sequence of symbols) $w$ is a member of the language described by a grammar $\mathcal{G}$ if it can be rewritten to the start symbol $S$, i.e., if there is a sequence $w = \alpha_0 \to \alpha_1 \to \ldots \to \alpha_n = S$ such that each step has the form $\alpha_i \to \alpha_{i+1}$ with $\alpha_i = \beta_1\beta_2\beta_3$ and $\alpha_{i+1} = \beta_1 B \beta_3$ and $\mathcal{G}$ contains a production $\beta_2 \to B$.

Despite, or maybe due to, the simplicity of this basic structure there has never emerged a standard formalism for syntax definition. The Backus Naur

Form (BNF), originally developed by Backus (1959) and Naur *et al.* (1960) for the definition of the syntax of Algol, is a commonly used notation for context-free grammars, but it does not have the status of a standard and many variants are in use. Several standard notations for syntax definition have been proposed (e.g., Wirth, 1977, Williams, 1982). None of these has been convincing, instead a number of similar or overlapping formalisms exist.

The reason for this divergence is that a practical syntax definition formalism serves not only to define languages, i.e, sets of strings. Syntax definitions are also interpreted as recognizers that decide whether a string belongs to a language, as parsers that map strings to parse trees, as mappings from parse trees to abstract syntax trees and as syntax directed editors. Plain context-free grammars are not adequate for this purpose. To support the compact definition of languages, formalisms can provide a variety of features as extensions to the basic structure: character classes, regular expressions, disambiguation by associativity and priority declarations, reuse by modularization, parameterization of language definitions, interfacing between the formalism and its environment, e.g., mapping to abstract syntax.

Various extensions of context-free grammars have been developed for attaching semantics to grammars: Attribute grammars (Knuth, 1968) attach attribute evaluation rules to productions. The computation of the semantics of a parse tree consists in computing the values of all attributes. This computation is orthogonal to parsing. Affix grammars (Koster, 1971) and extended affix grammars (Watt, 1977) are similar to attribute grammars, but predicates on affix values can play a role in disambiguation during parsing. Definite clause grammars (Pereira and Warren, 1980) are based on the Horn clauses of logic programming. Parsing is performed by the SLD resolution evaluation mechanism. Semantic values are represented by means of terms and passed around using unification and logic variables. Other approaches including algebraic specification use a separate formalism to define the semantics.

Traditionally, compiler construction is the main application area for syntax definition formalisms. The most well-known is the pair LEX/YACC. The formalism LEX (Lesk and Schmidt, 1986) is used to define the lexical syntax of a language using regular expressions. According to the regular expressions a string is analyzed and divided into tokens. In case more than one regular expression can be matched, a number of disambiguation rules such as prefer longest match and prefer regular expressions appearing earlier in the file. The 'compiler compiler' YACC (Johnson, 1975) is used for the definition of the context-free syntax of a language. An LALR(1) parser generator translates grammars to C programs if the grammar is LALR(1). Some conflicts in the parse table caused by ambiguous expression grammars can be solved by means of binary and unary precedences based on the ideas of Aho *et al.* (1975). Furthermore, the order of productions in the grammar is used to solve conflicts. Trees for a string can be constructed by calling C functions from the grammar productions.

Recent formalisms are generally based on the same deterministic parsing techniques, but extend the expressivity and declarativeness of syntax definition by providing mechanisms for building trees and coupling to other phases of compilation. Some examples are: The Cocktail compiler generator (Grosch, 1990) provides a BNF-like formalism with an LALR parser generator. The Eli system (Gray *et al.*, 1992) is a collection of tools for developing all aspects of compilers. The syntax definition formalism is based on context-free grammars. Attribute rules are added to define semantics computations. The tree transformation language TXL (Cordy and Carmichael, 1993) is a programming language for source to source transformations by means of transformation rules on parse trees. The syntax definition formalism of TXL is based on context-free grammars extended with some regular operators. Lexical syntax is defined by means of predefined lexical notions and by means of regular expressions over character classes. PCCTS (Parr and Quong, 1994) is a formalism based on top-down LL(1) parsing. The problems of unbounded lookahead are dealt with by means of backtracking and syntactic predicates that can be used to try out a variant before deciding which production to predict.

An application domain derived from compiler construction is the area of programming environments. A programming environment is a collection of tools for interactively developing and testing programs in some programming language. These tools are usually centered around an interactive syntax directed editor. A syntax editor has knowledge of the language of the programs being edited and provides support for checking the syntax of programs and for presenting and manipulating the structure of programs. In order to rapidly process changes to a program, incremental parsing and incremental compilation are used. Syntax definition formalisms developed for derivation of programming environments include the grammar formalism of the Synthesizer Generator (Reps and Teitelbaum, 1989), PSG (Bahlke and Snelting, 1986), METAL (Kahn *et al.*, 1983) and SDF (Heering *et al.*, 1989). The ASF+SDF Meta-Environment (Klint, 1993) is a programming environment for developing and generating programming environments from algebraic specifications. To speed up the development cycle for syntax definitions, incremental parser generation is used to only regenerate those parts of the parser that have been affected by a change.

Syntax definition in algebraic specification takes the form of grammars as algebraic signatures. The motivation here is to provide flexible notation for functions and constructors in abstract data type specifications and less the description of real programming languages. Therefore, the requirements on lexical syntax are not so strong. The correspondence of context-free grammars and many-sorted signatures was first described by Rus (see Hatcher and Rus, 1976). Goguen *et al.* (1977) showed that this correspondence could be used to define the semantics of programming languages. The correspondence was exploited in a number of algebraic specification formalisms to provide flexible, user-definable notation for functions and constructors. The first formalism to incorporate this was OBJ (Futatsugi *et al.*, 1985). Others are Cigale (Voisin, 1986), ASF+SDF (Heering *et al.*, 1989, Bergstra *et al.*, 1989a), the Meta Notation, used in action semantics (Mosses, 1992), and Elan (Vittek, 1994).

The combination of features that a formalism provides is, necessarily, rather arbitrary and strongly influenced by the expected application of definitions and the environment in which generated tools have to operate. Although it is not desirable to include all conceivable features in a formalism—some features can not be combined with others and too many features results in an unmanageable formalism—the similarities between different formalisms can be exploited by reusing parts of the design and implementation of old formalisms. However, formalisms are conventionally designed in a monolithic way, containing an intertwined mix of features, resulting in a formalism with a lack of orthogonality and uniformity that is difficult to implement, extend and use for other applications than the originally intended ones. Syntax definition formalisms form no exception to this rule.

Here we set out to design syntax definition formalisms in a modular way, as a family of formalisms each extending a small kernel with some feature for syntax definition. This approach should result in more orthogonal and uniform formalisms and should make it easier to (a) construct formalisms that use some subset of a set of known features, (b) adapt formalisms for use in other application areas, (c) implement tools for such formalisms and (d) design new formalisms that combine new features with existing ones.

As a first step to accomplish this goal we design a concrete formalism with a set of features that is useful in many application areas, but in particular in the application of grammars as signatures for algebraic specifications of programming languages. The result is the syntax definition formalism SDF2 that is a generalization of SDF. It incorporates several concepts and techniques introduced by Heering *et al.* (1989) in a more orthogonal and uniform way and adds several new features.

We use the algebraic specification formalism ASF+SDF to formally specify the family of syntax definition formalisms. For an introduction to ASF+SDF see Van Deursen *et al.* (1996).

In this chapter we outline the main features of SDF2 and examine the structure and design principles of the specification.

## 1.2 An Overview of SDF2

SDF2 is a syntax definition formalism based on context-free grammars, extended with character classes, sorts, literals, priorities, regular expressions, renamings, aliases and modules and combines the definition of lexical and context-free syntax into one formalism. The syntax definition in Figure 1.1 on page 7, taken from Visser (1997c), presents (the syntax of) a small untyped, first-order functional programming language, the data type environments and the evaluation function that interprets such functional programs using an instantiation of the environments data type. A program in this language might contain the following definition of a function `map` that applies a function `F` to all elements of a list `L`:

```
function map(F, L) is
          if(is-nil(L), nil(),
             cons(call(F, head(L)), map(F, tail(L))))
```

We sketch the main features of SDF2 and use the syntax definition in Figure 1.1 as running example.

### 1.2.1 Context-free Productions

The basis of the formalism is formed by *context-free productions*. These are rules of the form $\alpha \to \mathcal{A}$, where $\alpha$ is a list of symbols $\mathcal{A}_1 \ldots \mathcal{A}_n$ ($n \geq 0$) and $\mathcal{A}$ a symbol. A production declares that a string of category $\mathcal{A}$ can be constructed by concatenating strings of the categories $\mathcal{A}_i$. For instance, the production

```
Fun "(" Terms ")" -> Term
```

defines that a term can be constructed by means of a function symbol followed by a list of terms separated by commas between parentheses. Conventionally, context-free productions are written as $\mathcal{A} \to \alpha$ or as $\mathcal{A} ::= \alpha$. In SDF2 productions are written the other way around to make the similarity to function declarations more apparent. This is useful because SDF2 definitions are used as signatures in algebraic specifications such that productions correspond to algebraic operators. For example, in a conventional signature one would declare the evaluation function that computes the value of a term with respect to a program and some environment by means of the function declaration

```
eval : Program # Term # Env -> Term
```

The production

```
eval "[[" Program "]]" "(" Term ")" "_" Env -> Term
```

not only defines a function with the same input types, but also the syntax for its applications, i.e., the program argument should be enclosed in double square brackets and the term argument should be enclosed in parentheses.

### 1.2.2 Character Clasess

Syntax definitions describe languages consisting of strings of *characters*, where the set of all characters can be encoded by a finite set of consecutive natural numbers. *Character Classes* are compact descriptions of sets of characters and are typically used in the definition of lexical categories such as layout, identifiers and numbers. The example contains the following character classes: the characters space, tab and newline `[\ \t\n]`, all characters except newline `~[\n]`, all uppercase letters `[A-Z]`, all lowercase letters `[a-z]`, all letters and digits and the hyphen character `[a-zA-Z0-9\-]`.

### 1.2.3 Literals

*Literals* are strings of characters between double quotes that stand for exactly that string of characters. These are used to represent keywords—such as `"function"` and `"program"`—and operators and other literal symbols—such as `"["`, `"|->"`, `"|>"`, and `"*"`. The definition of the function `eval` does not use quotes for the function name. This is an exception to the general rule: identifiers starting with a lowercase letter can also be used as literals.

### 1.2.4 Sorts

The basic nonterminal symbols used in productions are *sorts*, which are written as identifiers starting with a capital letter. Sorts should be declared in a sorts section. The example defines the sorts `Var`, `Fun`, `Term`, etc.

### 1.2.5 Regular Expressions

More complex nonterminal symbols can be formed by means of *regular expressions* that provide abbreviations for tupling $(\_)$, iteration $\_*$ and $\_+$, optional constructs $\_?$, and alternatives $\_|\_$. For example, `[a-zA-Z0-9\-]*` denotes a list of zero or more characters from the set of letters, digits and hyphens, `{Term ","}*` declares a list of terms separated by commas and the expression `(Key "|->" Value)*` denotes lists of zero or more tuples consisting of a key, the symbol `"|->"` and a value.

### 1.2.6 Aliases

Since such regular expressions can become quite tedious to type, it can be useful to introduce a shorter name for such symbols. This can be done by introducing a *symbol alias*. For example the declaration

```
aliases
  {Term ","}* -> Terms
```

introduces `Terms` as an alias for the regular expression `{Term ","}*`.

### 1.2.7 Priorities

Some productions that have a sensible type are syntactically ambiguous. For instance, the two productions for destructive and consistent environment update

```
Env "|>" Env -> Env
Env "*"  Env -> Env
```

are ambiguous with respect to themselves and to each other, e.g., the environment expression `Env |> Env * Env` can be constructed as `(Env |> Env) * Env` or as `Env |> (Env * Env)`. *Associativity* and *priority* declarations are a way to resolve most ambiguities of this type. In the example, the ambiguity in the expressions above is resolved by means of the priority declaration

```
Env "|>" Env -> Env > Env "*" Env  -> Env
```

that declares `"|>"` to have higher priority than `"*"`, which entails the `(Env |> Env) * Env` interpretation. The `left` attribute of a production declares that the operator is left-associative.

### 1.2.8 Lexical and Context-free Syntax

The phrases making up a string over a language are usually divided into lexical tokens—the words of a sentence—and context-free phrases. The distinction between tokens and phrases is that the tokens making up a phrase can be separated by layout (whitespace and comments) while the characters comprising a token cannot. In definitions this distinction is indicated by means of *lexical* and *context-free* productions. For example, the lexical definition

```
[a-z][a-zA-Z0-9\-]* -> Fun
```

indicates that function symbols consist of a number of adjacent characters starting with a lowercase letter, followed by zero or more letters, digits or hyphens. While the tokens in the term `succ ( zero() )` can be separated by spaces, the characters in the token `succ` cannot. The layout that can occur between tokens should also be specified. The symbol `LAYOUT` is reserved for this purpose. In the example, layout is declared as

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

meaning that spaces, tabs and newlines (also called whitespace) are layout and that any suffix of a line starting with two percent signs is comment.

```
module Functional-Programs
exports
  sorts Var Fun Term FunDef Program
  aliases
    {Term ","}* -> Terms
    {Var  ","}* -> Vars
  lexical syntax
    [\ \t\n]            -> LAYOUT
    "%%" ~[\n]*         -> LAYOUT
    [A-Z][a-zA-Z0-9\-]* -> Var
    [a-z][a-zA-Z0-9\-]* -> Fun
  context-free syntax
    Var                -> Term
    Fun "(" Terms ")" -> Term

    "function" Fun "(" Vars ")" "is" Term -> FunDef
    "program" FunDef*                     -> Program

module Environments
exports
  sorts Key Value Env
  context-free syntax
    "[" (Key "|->" Value)* "]" -> Env
    Env "(" Key ")"            -> Value
    Env "|>" Env               -> Env {left}
    Env "*" Env                -> Env {left}
    "(" Env ")"                -> Env {bracket}
  context-free priorities
    Env "|>" Env -> Env > Env "*" Env  -> Env

module Function-Eval
imports Functional-Programs
        Environments [Key => Var Value => Term]
exports
  context-free syntax
    eval "[[" Program "]]" "(" Terms ")" "_" Env -> Terms
```

Figure 1.1: SDF2 definition of the *syntax* of a small functional programming language and its main evaluation function.

### 1.2.9 Modules

Grammars can be divided in a number of *modules* such that parts of a grammar can be reused in various language definitions. Modules consist of a list of exports and hiddens sections. An import of a module $M$ into a module $N$ denotes the inclusion of the exported grammar of $M$ into $N$. Thus the import of module `Terms` in module `Functions` means that the syntax of terms is included in the syntax of programs. To prevent name clashes or to instantiate generic modules, *renamings* of symbols and productions can be applied to imported modules. For example, the module `Function-Eval` specifying the evaluation function, imports the generic module defining environments by means of

```
imports Environments [Key => Var Value => Term]
```

renaming the sort `Key` to `Var` and the sort `Value` to `Term`, thus instantiating it for use with the terms of the functional programming language.

Modules can also be parameterized with a list of parameter symbols that can be instantiated on import. For instance, module `Environments` might also have been declared as

```
module Environments[Key Value]
```

declaring `Key` and `Value` as parameters. The import

```
imports Environments[Var Term]
```

would then perform the instantiation.

A complete syntax definition consists of a list of modules and a designated top module. The language defined by such a definition is the one defined by the grammar associated to the top module. Of course, in a programming environment for SDF2 this list does not have to reside in a single file. More likely, each module will be defined in one file with the module name as the file name.

## 1.3 Design

The next chapters give a formal algebraic specification of the syntax and semantics of SDF2. The semantics of a syntax definition is characterized by the well-formed trees it generates. A tree is associated with a sentence—its *yield*. The language associated to a definition is the set of sentences that are yields of trees generated by the definition. A parser is a function that given a sentence, produces the tree (or set of trees) that have that sentence as yield. We do not describe parsing as part of the specification of SDF2, but specify the output required of a parser and allow any implementation that does so. Parsing for SDF2 grammars is described in Visser (1997e).

### 1.3.1 Modularization

The formalism SDF2 is not designed monolithically, but modularized, as a family of formalisms. The kernel of this family is formed by context-free grammars. All features are defined as independent extensions of the kernel. The combination of the features forms SDF2. This setup makes it easier to construct a variant of the formalism by adding, removing or modifying features. Figure 1.2 depicts the structure of the family by means of (an abstraction of) the import graph of the specification.

Furthermore, the specification of SDF2 covers several aspects. The *syntax* of the formalism consists of the definition of the form of all its constructs. *Projection* functions on these constructs are defined in order to extract information from them. *Normalization* functions transform a syntax definition in order to simplify it. The specification of *parse trees* consists of several parts. A generic format for the representation of structured data called ATerms (Van den Brand

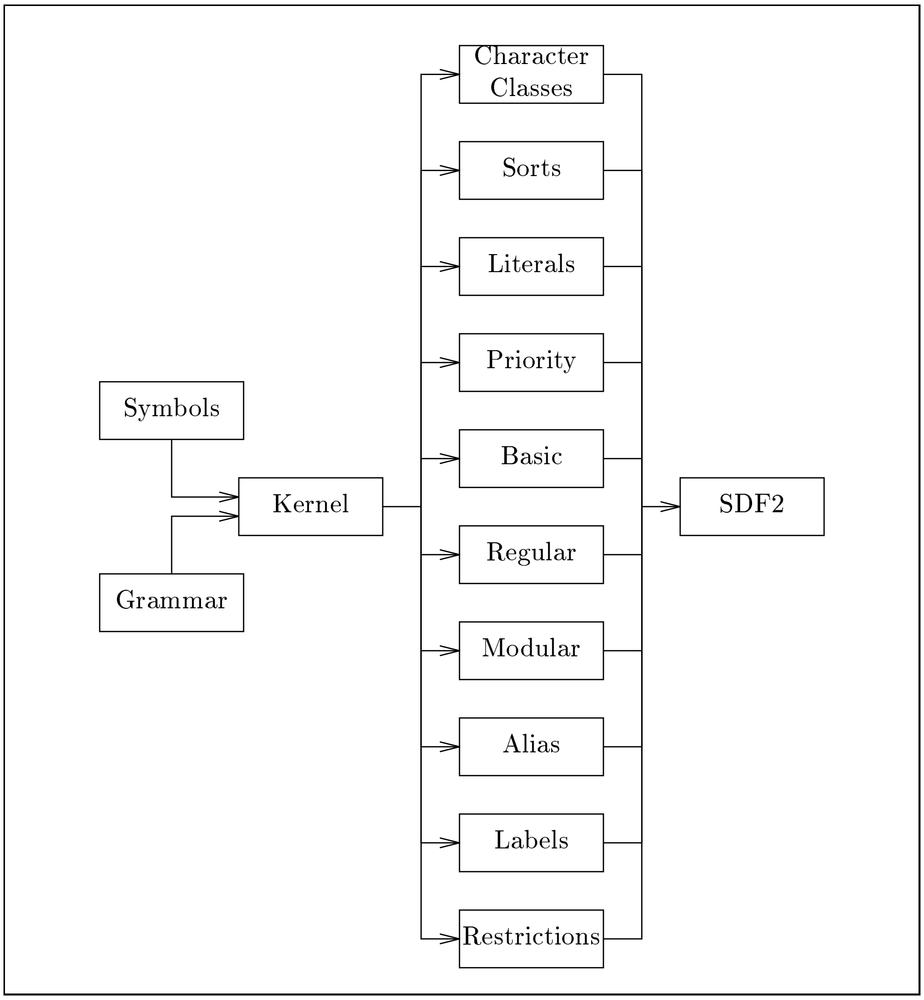



Figure 1.2: Import graph for the definition of SDF2.

*et al.*, 1997) is used to represent parse trees. In order to use this format for a specific purpose, *constructor* names have to be defined. To represent grammar information in parse trees, several constructs of the formalism have to be *encoded as ATerms.* Given this framework, the well-formedness of a tree with respect to a grammar can be defined. Furthermore, the *yield* of trees and the *equality* of trees are defined.

For each feature a number of modules are defined that each define an aspect of the formalism for that feature. The result is the matrix of modules listed in Table 1.1. The rows of the matrix contain the modules for one feature. The columns of the matrix contain all modules for one aspect. Each module in the matrix has a name consisting of the name of the feature and the name of the aspect separated by -Sdf-. For instance module Kernel-Sdf-Syntax specifies the syntax of the constructs in the kernel. So for each feature X we have modules X-Sdf-Syntax, X-Sdf-Projection, X-Sdf-Normalization, X-Sdf-Renamings, X-Sdf-Constructors, X-Sdf-ATerms, X-Sdf-Trees and X-Sdf-Equality. With the

| | *Syntax* | *Projection* | *Normalization* | *Renaming* | *Constructors* | *ATerms* | *Trees* | *Equality* |
|---|---|---|---|---|---|---|---|---|
| Symbols | 2.1 | | | | | 2.5.3 | | |
| Grammar | 2.2 | | 2.3.3 | | A.2.1 | A.2.2 | | |
| Kernel | 2.3.1 | 2.3.2 | 2.3.3 | 4.1.3 | 2.5.2 | 2.5.4 | 2.5.5 | 2.5.8 |
| Sorts | 2.4.1 | 2.4.1 | 2.4.1 | A.3 | A.2.1 | A.2.2 | | |
| CC | 2.4.2 | | 2.4.2 | A.3 | A.2.1 | A.2.2 | 2.5.6 | |
| Literals | 2.4.3 | | 2.4.3 | A.3 | A.2.1 | A.2.2 | | |
| Priority | 3.1.1 | 3.1.2 | 3.1.3 | A.3 | | A.2.2 | 3.1.4 | |
| Regular | 3.2.1 | | 3.2.2 | A.3 | A.2.1 | A.2.2 | | |
| Basic | 3.3.1 | | 3.3.2 | A.3 | A.2.1 | A.2.2 | | 3.3.3 |
| Restrictions | 3.4.1 | 3.4.2 | 3.4.3 | A.3 | | | | |
| Renaming | 4.1.1 | 4.1.2 | | 4.1.3 | | | 4.1.4 | |
| Alias | 4.2.1 | 4.2.2 | 4.2.3 | 4.2.3 | | | | |
| Modular | 4.3.1 | 4.3.2 | 4.3.3 | 4.3.4 | | | | |
| Label | | | | | | | | |
| Sdf2 | 5.1.1 | A.4 | 5.1.2 | A.4 | A.4 | A.4 | A.4 | A.4 |

Table 1.1: Modules of the family of syntax definition formalisms. The last row contains the collecting modules for SDF2. There are no collection modules for the rows. The numbers refer to the sections presenting the modules.

exception that if some feature does not change some aspect, the module is omitted.

### 1.3.2 Normalization

An important role in the design of SDF2 is played by the normalization function. In general, a normalization function defines a transformation on an expression that yields an expression in the same language, which uses less features. The normalized expression has the same meaning as the original one. Thus, a normalization is a mapping from the language onto (a subset of) the same language. Ideally, a normalization function should be idempotent, i.e., yield the same result when applied twice. An implementation for such a language only has to consider the simplified expressions, while users have a more expressive language at their disposal.

The requirement that normalization produces an expression in the language itself entails that all constructs used for encodings should also be present in the original language, i.e., the language should be closed under normalization. For example, one of the normalizations in this definition renames a symbol $\mathcal{A}$ into $\langle\mathcal{A}\texttt{-LEX}\rangle$ if it occurs in the lexical syntax. Therefore, the constructor $\langle\_\texttt{-LEX}\rangle$ introduced for the purpose of normalization also becomes a construct of the language before normalization.

A consideration in the definition of a normalization is whether two different expressions that are equivalent with respect to the semantics have the same normal form. This can be useful when expressions have to be compared. This is for example the case in the normalization of character classes. In Visser (1997b) a normalization of character classes to a unique normal forms is defined such that two character classes that represent the same set of characters are normalized to the same character class expression. In general, however, the normalizations in this paper will not have this property. For instance, all permutations of a list of productions are equivalent. Although such lists could be ordered by imposing an ordering on productions, this is not done here, since comparisons of lists of productions are not needed. In such cases definitions can not use syntactic equality to determine equivalence.

Using this approach SDF2 is an expressive formalism that depends on a small set of features, i.e., we have:

$$
\begin{array}{lcl}
\text{SDF2} & = & \text{context-free grammars} \\
 & + & \text{priorities} \\
 & + & \text{character-classes} \\
 & + & \text{reject productions} \\
 & + & \text{follow restrictions}
\end{array}
$$

Features that are provided in the formalism, but that are eliminated, i.e., expressed using the features above are: literals, regular expressions, lexical and context-free syntax, variables, modules, renamings, and aliases. Furthermore, character classes, priority declarations and grammar composition are simplified considerably.

The normalization of SDF2 is defined as a pipeline of normalizations, as is illustrated in Figure 1.3. This modularization of the definition of normalization makes it easy to define an extension and express it in existing features using a new normalization function. The overall normalization is extended by adding the new function to the normalization pipeline.

## 1.4 Organization

The next chapters discuss the specification of the family of syntax definition formalisms that is the basis of SDF2. Chapter 2 defines context-free grammars, the basic symbols sorts, character classes and literals and defines the well-formed parse trees characterized by a grammar. Chapter 3 defines disambiguation by means of priorities, regular expressions, lexical and context-free syntax and restrictions for lexical disambiguation. Chapter 4 introduces renamings, aliases and modules. Chapter 5 these extensions are combined in the formalism SDF2. The formalism is compared to SDF and a discussion of possible improvements and extensions is given. Appendix A gives some auxiliary modules for the specification of SDF2.

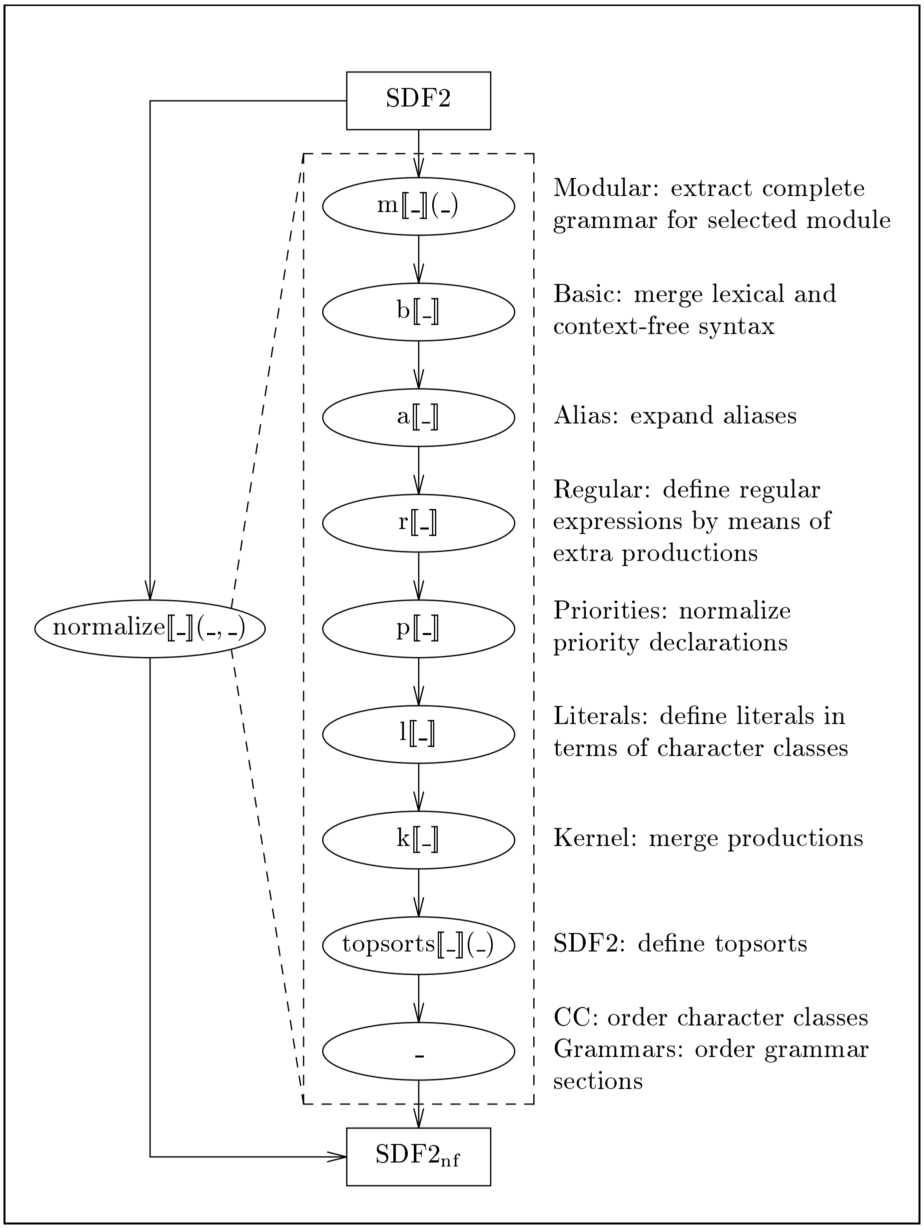



Figure 1.3: The normalization of SDF2 definitions consists of a series of independent transformations. The last step is not performed by a transformation function, but by rewrite rules acting on the constructors themselves.

# 2

# Context-Free Grammars

In this chapter a context-free grammar formalism is defined. First an abstract framework of symbols and grammars is introduced. In this framework a grammar is interpreted by means of several predicates and functions that characterize the trees and strings of symbols generated by a grammar. A well-formedness predicate on parse trees characterizes the trees over a grammar. From the parse trees over a grammar the strings of the language defined by the grammar are derived. Parse trees are represented in the annotated term format ATerms. An instance of this framework is set up with as kernel context-free productions. We introduce three kinds of basic symbols to be used in productions: sorts, character classes and literals.

## 2.1 Symbols

Syntax definitions define languages, i.e., sets of strings of *symbols*. A string of symbols is a list of zero or more symbols. The sort Symbol is declared without actually specifying any constructors for it. This entails that the sort is empty at this point, but can be extended later on with constructors. We do not make a distinction between terminal and nonterminal symbols. Whether a symbol is a terminal or nonterminal symbol is determined by the interpretation and is not fixed syntactically. A symbol that plays the role of a terminal in one view can be a nonterminal in another view. An example is a literal that can be considered as a terminal token or as a nonterminal that is defined in terms of characters.

**module** Symbols
**imports** Layout
**exports**
  **sorts** Symbol Symbols
  **context-free syntax**
    "(" Symbol ")" $\rightarrow$ Symbol {**bracket**}
    Symbol$*$ $\rightarrow$ Symbols
  **variables**
    $[\mathcal{ABC}][\mathit{0\text{-}9'}]*$ $\rightarrow$ Symbol
    $[\alpha\beta\gamma][\mathit{0\text{-}9'}]*$ $\rightarrow$ Symbol$*$
    $[\alpha\beta\gamma]$"+"$[\mathit{0\text{-}9'}]*$ $\rightarrow$ Symbol+

### 2.1.1 Projection

The function $+\!\!+$ concatenates strings and $|\_|$ gives the length of a string. The predicate $\in$ decides list membership.

**module** Symbols-Projection
**imports** Symbols$^{2.1}$ Booleans Integers
**exports**
  **context-free syntax**
    Symbols "$+\!\!+$" Symbols → Symbols {**assoc**}
    "|" Symbols "|" → Int
    Symbol "$\in$" Symbols → Bool
**equations**
Concatenation, length and membership of symbol lists.

| | |
|---|---|
| [1] | $\alpha +\!\!+ \beta = \alpha\ \beta$ |
| [2] | $\mid\ \mid = 0$ |
| [3] | $\mid \mathcal{A}\ \alpha \mid = \mid \alpha \mid + 1$ |
| [4] | $\mathcal{A} \in\ = \bot$ |
| [5] | $\mathcal{A} \in \mathcal{A}\ \alpha = \top$ |
| [6] | $\mathcal{A} \in \mathcal{B}\ \alpha = \mathcal{A} \in \alpha \quad$ **when** $\mathcal{A} \neq \mathcal{B}$ |

The concatenation function on sorts such as Symbols is needed because the concatenation of the built-in associative lists (e.g., Symbol∗) of ASF+SDF is not inherited through the injection Symbol∗ → Symbol. The injection is needed because list sorts cannot be output sorts of functions in ASF+SDF.

### 2.1.2 Sets

From lists of symbols we construct sets of symbols by means of the operation $\{\_\}$. Although this constructor does not remove double elements from the list, it signifies that the number of occurrences in the list does not matter. Operations on sets are union ($\cup$), difference ($/$) and membership ($\in$). The union $\upsilon_1 \cup \upsilon_2$ adds only those elements of $\upsilon_1$ to $\upsilon_2$ that do not already occur in $\upsilon_2$. If a set is constructed by means of union from singleton sets, the resulting set does not contain double elements. Although this is not strictly necessary it is useful when later on something has to be done once for each symbol in some set.

**module** Symbol-Sets
**imports** Symbols-Projection$^{2.1.1}$ Booleans
**exports**
  **sorts** SymbolSet
  **context-free syntax**
    "{" Symbols "}" → SymbolSet
    SymbolSet "$\cup$" SymbolSet → SymbolSet {**right**}
    SymbolSet "/" SymbolSet → SymbolSet {**left**}
    "(" SymbolSet ")" → SymbolSet {**bracket**}
    Symbol "$\in$" SymbolSet → Bool

**priorities**
SymbolSet "/" SymbolSet → SymbolSet >
SymbolSet "∪" SymbolSet → SymbolSet
**variables**
"$\upsilon$" [*0-9'*]∗ → SymbolSet
**equations**
Membership

[1] $$\mathcal{A} \in \{\alpha\} = \mathcal{A} \in \alpha$$

Union

[2] $$\{\} \cup \upsilon = \upsilon$$
[3] $$\upsilon \cup \{\} = \upsilon$$
[4] $$\{\alpha^+ \ \beta^+\} \cup \upsilon = \{\alpha^+\} \cup \{\beta^+\} \cup \upsilon$$
[5] $$\{\mathcal{A}\} \cup \upsilon = \upsilon \quad \textbf{when} \ \ \mathcal{A} \in \upsilon = \top$$
[6] $$\{\mathcal{A}\} \cup \{\alpha\} = \{\mathcal{A} \ \alpha\} \quad \textbf{otherwise}$$

Difference

[7] $$\{\} \ / \ \upsilon = \{\}$$
[8] $$\upsilon \ / \ \{\} = \upsilon$$
[9] $$\{\alpha^+ \ \beta^+\} \ / \ \upsilon = \{\alpha^+\} \ / \ \upsilon \cup \{\beta^+\} \ / \ \upsilon$$
[10] $$\{\mathcal{A}\} \ / \ \upsilon = \{\} \quad \textbf{when} \ \ \mathcal{A} \in \upsilon = \top$$
[11] $$\{\mathcal{A}\} \ / \ \upsilon = \{\mathcal{A}\} \quad \textbf{otherwise}$$

## 2.2 Grammars

A syntax definition consists of a grammar. The only generic operations on grammars that we define at this point are an associative composition operation that is used to combine grammars and the constant ∅ representing the empty grammar.

**module** Grammar-Syntax
**imports** Layout
**exports**
**sorts** Grammar
**context-free syntax**
"∅" → Grammar
Grammar Grammar → Grammar {**assoc**}
"(" Grammar ")" → Grammar {**bracket**}
**variables**
"$\mathcal{G}$" [*0-9'*]∗ → Grammar

### 2.2.1 Interpretation

A grammar defines a set of strings of symbols, a language. We specify the language derived by a grammar indirectly, via the trees it generates. Parse trees will be represented by means of ATerms, a term format for the representation and exchange of structured data (Van den Brand *et al.*, 1997). The format will be introduced in §2.5.1. In fact we will not just define parse trees, but parse forests. A parse forest is a compact encoding of a collection of parse trees in which contexts are shared. A parse forest is used to represent all parse trees for an ambiguous sentence.

The interpretation of a grammar is now given by two predicates and a function. The predicate $\mathcal{G} \vdash T$ characterizes the terms $T$ that are *well-formed parse forests* over grammar $\mathcal{G}$. The function $\mathrm{yield}[\![\mathcal{G}]\!](T)$ maps a parse tree $T$ to a string of symbols. The predicate $T \mathrel{\dot\in} T'$ determines membership of a tree $T$ in a forest $T'$. The set of trees generated by a grammar is denoted by $\mathcal{T}[\![\mathcal{G}]\!]$.

Given these functions we can derive the notion of the language generated by a grammar and the notion of a parser for a grammar. The *language* generated by a grammar corresponds to the set of yields of the parse trees it generates. In other words, a string of symbols $\alpha$ is an element of the language defined by a grammar $\mathcal{G}$, if there exists a well-formed tree $T$ over the grammar with $\alpha$ as its yield. A *parser* $\Pi[\![\mathcal{G}]\!](\alpha)$ is a function that maps a string $\alpha$ in $\mathcal{L}[\![\mathcal{G}]\!]$ to a parse forest $T$ containing all well-formed trees such that their yield is $\alpha$. The module below summarizes these definitions. The equations define the set of parse trees $\mathcal{T}[\![\mathcal{G}]\!]$, the language $\mathcal{L}[\![\mathcal{G}]\!]$ and the parser $\Pi[\![\mathcal{G}]\!](\_)$ in terms of well-formedness, yield and parse forest membership.

**module** Grammar-Interpretation
**imports** Grammar-Syntax[2.2] Symbols[2.1] Booleans ATerms[2.5.1] Symbols-Sets ATerm-Sets
**exports**
**context-free syntax**

| | |
|---|---|
| Grammar "$\vdash$" ATerm | $\rightarrow$ Bool |
| ATerm "$\dot\in$" ATerm | $\rightarrow$ Bool |
| yield "$[\![$" Grammar "$]\!]$" "(" ATerm ")" | $\rightarrow$ Symbols |
| "T" "$[\![$" Grammar "$]\!]$" | $\rightarrow$ ATermList |
| "$\mathcal{L}$" "$[\![$" Grammar "$]\!]$" | $\rightarrow$ SymbolsSet |
| "$\Pi$" "$[\![$" Grammar "$]\!]$" "(" Symbols ")" | $\rightarrow$ ATerm |

**equations**

[1] $$\frac{\mathcal{G} \vdash T = \top}{T \in \mathrm{T}[\![\mathcal{G}]\!] \ = \ \top}$$

[2] $$\frac{\mathcal{G} \vdash T = \top,\ \mathrm{yield}[\![\mathcal{G}]\!](T) = \alpha}{\alpha \in \mathcal{L}[\![\mathcal{G}]\!] \ = \ \top}$$

[3] $$\frac{\mathcal{G} \vdash T = \top,\ \mathrm{yield}[\![\mathcal{G}]\!](T) = \alpha}{T \mathrel{\dot\in} \Pi[\![\mathcal{G}]\!](\alpha) \ = \ \top}$$

Note that these equations are non-constructive, i.e., do not provide decision procedures, but are merely a specification of the required behaviour. (Module Symbols-Sets defines sets of strings of symbols in a similar way as sets of symbols are defined in module Symbol-sets.)

### 2.2.2 Overview

In the rest of this chapter we will provide the specifications of well-formedness, yield and forest membership for a context-free grammar formalism. In §2.3 we define the syntax and normalization of context-free productions. In §2.4 we define basic symbols to be used in grammars: sorts, character classes and literals. In §2.5 we define the well-formed parse trees generated by a context-free grammar. In the next chapters this formalism is extended with a number of features.

An implementation of a parser is not specified, but can be chosen such as to comply with this specification. One possible implementation is discussed in Visser (1997e).

## 2.3 Context-Free Grammars (Kernel)

### 2.3.1 Syntax

The kernel of SDF2 is formed by context-free grammars. A context-free production is a structure $\alpha \rightarrow \mathcal{A}$, where $\alpha$ is a list of symbols and $\mathcal{A}$ a symbol. A context-free grammar is formed by a list of productions preceded by the keyword `syntax`.

Conventionally, since their introduction by Chomsky (1956), context-free productions are written as $\mathcal{A} \rightarrow \alpha$ to emphasize the generative view of grammars. A grammar *generates* a string from a symbol, by repeatedly replacing some symbol in a string by the symbols on the right-hand side of a production. There exist many variants of this 'standard' notation, e.g., $\mathcal{A} ::= \alpha$ in BNF (Backus, 1959) and $\mathcal{A} : \alpha_1 | \ldots | \alpha_n;$ in YACC (Johnson, 1975).

The unconventional $\alpha \rightarrow \mathcal{A}$ notation for productions introduced by Heering *et al.* (1989) emphasizes the *functional* view of productions when used in the context of algebraic specification. A production coincides with the declaration of the name and type of a function. This notation is a unification of the definition of context-free productions with the declaration of mixfix functions in algebraic specification formalisms. For example, the declaration of the infix addition operator on natural numbers that is declared as `Nat ::= Nat "+" Nat` in BNF, is declared as `op _ + _ : nat nat -> nat .` in OBJ (Futatsugi *et al.*, 1985) and as `@ + @ : (nat nat) nat` in Elan (Vittek, 1994). In SDF this becomes `Nat "+" Nat -> Nat`.
All these notations are equivalent in expressive power and could be used instead of the current one. We could effortlessly define a version of SDF that uses the $\mathcal{A} ::= \alpha$ notation of BNF and define its meaning by translation to the notation used here. Note, however, that this does not mean that other aspects of these formalisms have the same expressive power nor that the parsing techniques coupled to these formalisms all have the same power.

Optionally, productions can have a list of attributes. An *attribute* is an annotation of a production that gives some extra syntactic or semantic information about the production. An example of an attribute that will be introduced in §3.1 is `left` that indicates left associativity of the production. Productions can have any number of attributes. The kernel does not provide any attributes, but to be able to introduce attributes later on without having to introduce an extra constructor for productions, the attribution of a production is defined here.

**module** Kernel-Sdf-Syntax
**imports** Symbols[2.1] Grammar-Syntax[2.2]
**exports**
  **sorts** Attribute Attributes Production Productions
  **context-free syntax**

| | |
|---|---|
| "{" {Attribute ","}$*$ "}" | $\to$ Attributes |
| | $\to$ Attributes |
| Symbols "$\to$" Symbol Attributes | $\to$ Production |
| Production$*$ | $\to$ Productions |
| "syntax" Productions | $\to$ Grammar |

  **variables**

| | |
|---|---|
| "*attr*"[*0-9'*]$*$ | $\to$ Attribute |
| "*attr*""$*$"[*0-9'*]$*$ | $\to$ {Attribute ","}$*$ |
| "*attr*""+"[*0-9'*]$*$ | $\to$ {Attribute ","}+ |
| "$"[*0-9'*]$*$ | $\to$ Attributes |
| [*p*][*0-9'*]$*$ | $\to$ Production |
| [*p*]"$*$"[*0-9'*]$*$ | $\to$ Production$*$ |
| [*p*]"+"[*0-9'*]$*$ | $\to$ Production+ |

### 2.3.2 Projection

We define concatenation functions for lists of productions and lists of attributes. The concatenation function for attributes removes duplicates. A production with an empty list of attributes is equal to a production without attributes. The projection function 'P' gives the productions of a grammar, the function '$\overline{\mathrm{P}}$' gives the non-production parts of a grammar (to be defined later), and the function '$\mathrm{P}_{\mathcal{A}}$' gives all productions defining a symbol $\mathcal{A}$. The function 'symbols' gives the set of all symbols in a grammar. The function 'reachable' gives all productions reachable from some set of symbols, i.e., used in the definition of those symbols.

**module** Kernel-Sdf-Projection
**imports** Kernel-Sdf-Syntax[2.3.1] Symbol-Sets[2.1.2]
**exports**
  **context-free syntax**

| | | |
|---|---|---|
| Productions "$+\!\!+$" Productions | $\to$ Productions | {**right**} |
| Attributes "$+\!\!+$" Attributes | $\to$ Attributes | {**right**} |
| Production "$\in$" Productions | $\to$ Bool | |
| Productions "$\subseteq$" Productions | $\to$ Bool | |
| Production "$\cong$" Production | $\to$ Bool | |
| "P"(Grammar) | $\to$ Productions | |
| "$\overline{\mathrm{P}}$" "(" Grammar ")" | $\to$ Grammar | |
| "P" "_" Symbol "(" Productions ")" | $\to$ Productions | |
| symbols(Productions) | $\to$ SymbolSet | |
| symbols(Grammar) | $\to$ SymbolSet | |
| reachable(SymbolSet, SymbolSet, Productions) | $\to$ Productions | |
| reachable(SymbolSet, Grammar) | $\to$ Grammar | |

**equations**

Concatenation of lists of productions, membership and subset of a list of productions.

[1] $p_1^* \mathbin{+\!\!+} p_2^* = p_1^*\ p_2^*$
[2] $p \in p_1^*\ p\ p_2^* = \top$
[3] $p \in p^* = \bot$ **otherwise**
[4] $\subseteq p^* = \top$
[5] $p\ p_1^* \subseteq p_2^* = p \in p_2^* \wedge p_1^* \subseteq p_2^*$

Concatenation of attribute lists. Attributes ocurring in both lists are added only once.

[6] $\{attr_1^+,\ attr_2^+\} \mathbin{+\!\!+} \{attr^*\} = \{attr_1^+\} \mathbin{+\!\!+} \{attr_2^+\} \mathbin{+\!\!+} \{attr^*\}$
[7] $\{attr\} \mathbin{+\!\!+} \{attr_1^*,\ attr,\ attr_2^*\} = \{attr_1^*,\ attr,\ attr_2^*\}$
[8] $\{attr\} \mathbin{+\!\!+} \{attr^*\} = \{attr,\ attr^*\}$ **otherwise**
[9] $\$ \mathbin{+\!\!+} \ = \$$
[10] $\mathbin{+\!\!+} \$ = \$$
[11] $\{\} =$

The last equation states that an empty list of attributes $\{\}$ is equal to no attributes.

Two productions are similar if they are the same up to their attributes

[12] $\alpha \to \mathcal{A}\ \$_1 \cong \alpha \to \mathcal{A}\ \$_2 = \top$
[13] $p_1 \cong p_2 = \bot$ **otherwise**

Function 'P' gives all productions of a grammar and function '$\overline{\mathrm{P}}$' gives all non-syntax parts of a grammar. The function $\mathrm{P}_{\mathcal{A}}$ gives all productions defining the symbol $\mathcal{A}$.

[14] $\mathrm{P}(\text{syntax } p^*) = p^*$
[15] $\mathrm{P}(\mathcal{G}_1\ \mathcal{G}_2) = \mathrm{P}(\mathcal{G}_1) \mathbin{+\!\!+} \mathrm{P}(\mathcal{G}_2)$
[16] $\mathrm{P}(\mathcal{G}) =$ **otherwise**
[17] $\overline{\mathrm{P}}(\text{syntax } p^*) = \emptyset$
[18] $\overline{\mathrm{P}}(\mathcal{G}_1\ \mathcal{G}_2) = \overline{\mathrm{P}}(\mathcal{G}_1)\ \overline{\mathrm{P}}(\mathcal{G}_2)$
[19] $\overline{\mathrm{P}}(\mathcal{G}) = \mathcal{G}$ **otherwise**
[20] $\mathrm{P}_{\mathcal{A}}() =$
[21] $\mathrm{P}_{\mathcal{A}}(\alpha \to \mathcal{A}\ \$\ p^*) = \alpha \to \mathcal{A}\ \$ \mathbin{+\!\!+} \mathrm{P}_{\mathcal{A}}(p^*)$
[22] $\mathrm{P}_{\mathcal{A}}(p\ p^*) = \mathrm{P}_{\mathcal{A}}(p^*)$ **otherwise**

The function 'symbols' gives the set of symbols of a list of productions or a grammar.

[23] $\text{symbols}() = \{\}$
[24] $\text{symbols}(\alpha \to \mathcal{A}\ \$\ p^*) = \{\alpha\} \cup \{\mathcal{A}\} \cup \text{symbols}(p^*)$
[25] $\text{symbols}(\text{syntax } p^*) = \text{symbols}(p^*)$
[26] $\text{symbols}(\mathcal{G}_1\ \mathcal{G}_2) = \text{symbols}(\mathcal{G}_1) \cup \text{symbols}(\mathcal{G}_2)$
[27] $\text{symbols}(\mathcal{G}) = \{\}$ **otherwise**

The function 'reachable' gives the subgrammar with those productions reachable from some set of symbols $v$. It is defined by applying the auxiliary 'reachable' function to the productions of the grammar. Observe how the resulting grammar is a composition of the reachable productions and the non-production parts of a grammar. The auxiliary function selects for each symbol in the original set the productions for that symbol from the original grammar. This is applied recursively to the symbols used in the left-hand sides of those productions. The first set argument of the auxiliary function represents the symbols already handled. The second set contains the symbols for which the productions still have to be looked up.

[28] $$\text{reachable}(\upsilon, \mathcal{G}) \;=\; \text{syntax reachable}(\{\}, \upsilon, \mathrm{P}(\mathcal{G}))\ \overline{\mathrm{P}}(\mathcal{G})$$

[29] $$\text{reachable}(\upsilon, \{\}, p^*) \;=\;$$

[30] $$\frac{\mathcal{A} \in \upsilon_1 = \bot,\ \ \mathrm{P}_{\mathcal{A}}(p_1^*) = p_2^*,\ \ \text{symbols}(p_2^*)\ /\ \upsilon_1 = \upsilon_2}{\text{reachable}(\upsilon_1, \{\mathcal{A}\ \alpha\}, p_1^*) \;=\; p_2^* +\!\!+ \text{reachable}(\upsilon_1 \cup \{\mathcal{A}\}, \upsilon_2 \cup \{\alpha\}, p_1^*)}$$

[31] $$\text{reachable}(\upsilon_1, \{\mathcal{A}\ \alpha\}, p^*) \;=\; \text{reachable}(\upsilon_1, \{\alpha\}, p^*)$$
**otherwise**

### 2.3.3 Normalization

*Grammar Normalization* Composition of grammars is commutative and the empty grammar is a unit for grammar composition. Since commutativity cannot be expressed by means of a terminating rewrite system, the following module normalizes grammar compositions as a right associative list, where the grammars are ordered or merged as specified by the operation $\diamond$. If $\mathcal{G}_1 \diamond \mathcal{G}_2$ yields a pair $\langle \mathcal{G}_3, \mathcal{G}_4 \rangle$ this means that the composition $\mathcal{G}_1 \mathcal{G}_2$ should be replaced by $\mathcal{G}_3 \mathcal{G}_4$. The definition of $\diamond$ can either merge the two grammars into one, yielding the pair $\langle \mathcal{G}_3, \emptyset \rangle$, or exchange the grammars yielding $\langle \mathcal{G}_2, \mathcal{G}_1 \rangle$. The termination of this normalization depends on the property of $\diamond$ that a swap cannot be undone. The definition of $\diamond$ should be extended for each new grammar constructor. For example, the merging of the productions of two adjacent syntax sections is expressed in the next module. The ordering could also be defined directly on the grammar composition operator, but that would entail that two equations would have to be written for each pair of constructors that have to be merged or swapped, corresponding to the last two equations below.

**module** Grammar-Normalization
**imports** Grammar-Syntax$^{2.2}$
**exports**
  **sorts** Grammar-Grammar
  **context-free syntax**
    Grammar "$\diamond$" Grammar → Grammar-Grammar
    "<" Grammar "," Grammar ">" → Grammar-Grammar
**equations**
The empty grammar $\emptyset$ is a unit for composition and composition is associative.

[1] $$\emptyset\ \mathcal{G} = \mathcal{G}$$
[2] $$\mathcal{G}\ \emptyset = \mathcal{G}$$
[3] $$\mathcal{G}_1\ (\mathcal{G}_2\ \mathcal{G}_3) = \mathcal{G}_1\ \mathcal{G}_2\ \mathcal{G}_3$$

Subgrammars can be swapped as specified by the function $\diamond$.

[4] $$\mathcal{G}_1\ \mathcal{G}_2 = \mathcal{G}_1'\ \mathcal{G}_2' \quad \textbf{when} \quad \mathcal{G}_1 \diamond \mathcal{G}_2 = \langle \mathcal{G}_1', \mathcal{G}_2' \rangle$$

[5] $$\mathcal{G}_1\ \mathcal{G}_2\ \mathcal{G}_3 = \mathcal{G}_1\ \mathcal{G}_2'\ \mathcal{G}_3' \quad \textbf{when} \quad \mathcal{G}_2 \diamond \mathcal{G}_3 = \langle \mathcal{G}_2', \mathcal{G}_3' \rangle$$

*Context-free Grammar Normalization* The normalization function $\mathrm{k}[\![\_]\!]$ for the kernel, merges productions with the same arguments and result symbols. If such productions have different attributes, these are joined. This normalization entails that two occurrences of the same production are identified and do therefore not cause an ambiguity. Consequently, other normalization functions can generate a production more than once, without changing the meaning of the grammar. This strategy will be relevant later on when we introduce modularization of grammars. The identification of productions means that a production that is declared in two or more different modules is identified when these modules are imported in the same module.

**module** Kernel-Sdf-Normalization
**imports** Kernel-Sdf-Projection$^{2.3.2}$ Grammar-Normalization$^{2.3.3}$
**exports**
  **context-free syntax**
    "k" "[\![" Grammar "]\!]" → Grammar
    merge(Productions) → Productions
**equations**

An empty list of productions is equivalent to an empty grammar and multiple syntax sections are merged into one.

[1] $$\text{syntax} = \emptyset$$

[2] $$\text{syntax } p_1^* \diamond \text{syntax } p_2^* = \langle \text{syntax } p_1^*\ p_2^*, \emptyset \rangle$$

The normalization function 'k' merges productions with the same arguments and result, using the auxiliary function 'merge'.

[3] $$\mathrm{k}[\![\mathcal{G}]\!] \ = \ \text{syntax merge}(\mathrm{P}(\mathcal{G}))\ \overline{\mathrm{P}}(\mathcal{G})$$

[4] $$\frac{p_1 = \alpha \rightarrow \mathcal{A}\ \$_1,\ \ p_2 = \alpha \rightarrow \mathcal{A}\ \$_2,\ \ \$_1 +\!\!+ \$_2 = \$_3,\ \ p_3 = \alpha \rightarrow \mathcal{A}\ \$_3}{\text{merge}(p_1^*\ p_1\ p_2^*\ p_2\ p_3^*) \ = \ \text{merge}(p_1^*\ p_3\ p_2^*\ p_3^*)}$$

[5] $$\text{merge}(p^*) \ = \ p^* \quad \textbf{otherwise}$$

## 2.4 Basic Symbols

The kernel formalism presented in the previous section is a complete definition of context-free grammars, except for the notation of symbols. In this section we present three extensions of the kernel that provide notation for basic symbols needed in syntax definition. Sorts represent the non-terminals of grammars, the categories or domains that the grammar introduces. Character classes are used to represent the terminals of grammars, the characters from which strings are built. Literals are convenient abbreviations for fixed strings of characters. With these extensions we will have a complete notation for context-free grammars. The extensions in later sections will provide features to make this formalism more expressive.

### 2.4.1 Sorts

*Syntax* Sorts are the symbols that represent the basic domains or categories of a syntax definition. A sort identifier is a word starting with an uppercase letter followed by zero or more letters or digits. Hyphens can be used between the first and last character. Sorts used in the productions of a grammar should be declared in a separate sorts section that consists of the keyword 'sort' and a list of symbols.

**module** Sorts-Sdf-Syntax
**imports** Kernel-Sdf-Syntax$^{2.3.1}$
**exports**
  **sorts** Sort
  **lexical syntax**
    `[A-Z]` → Sort
    `[A-Z][A-Za-z0-9\-]*[A-Za-z0-9]` → Sort
  **context-free syntax**
    Sort → Symbol
    "sorts" Symbols → Grammar
  **variables**
    "$\mathcal{S}$"[*0-9'*]* → Sort

*Normalization* Ordering of sorts and syntax sections (sorts are placed before syntax sections) and merging of sorts sections.

**module** Sorts-Sdf-Normalization
**imports** Sorts-Sdf-Syntax$^{2.4.1}$ Kernel-Sdf-Normalization$^{2.3.3}$
**equations**

[1] $$\text{sorts} = \emptyset$$
[2] $$\text{sorts } \alpha \diamond \text{sorts } \beta = \langle \text{sorts } \alpha\ \beta,\ \emptyset \rangle$$
[3] $$\text{syntax } p^* \diamond \text{sorts } \alpha = \langle \text{sorts } \alpha,\ \text{syntax } p^* \rangle$$

*Projection* The projection function 'S' gives the list of sorts of a grammar.

**module** Sorts-Sdf-Projection
**imports** Kernel-Sdf-Projection$^{2.3.2}$ Sorts-Sdf-Syntax$^{2.4.1}$
**exports**
  **context-free syntax**
    "S"(Grammar) → Symbols
**equations**
The declared sorts of a grammar.

[1] $$\mathrm{S}(\text{sorts } \alpha) = \alpha$$
[2] $$\mathrm{S}(\mathcal{G}_1\ \mathcal{G}_2) = \mathrm{S}(\mathcal{G}_1) +\!\!\!+\ \mathrm{S}(\mathcal{G}_2)$$
[3] $$\mathrm{S}(\mathcal{G}) = \quad \textbf{otherwise}$$

### 2.4.2 Character Classes

A character class is an expression such as, for example, `[a-z\']` that denotes a set of characters, in this case the set of all lower case letters and a prime. For example, the following definition defines identifiers as lists of characters starting with a lowercase letter followed by zero or more lowercase letters or digits.

```
sorts Id
syntax
  [a-z]        -> Id
  Id [a-z0-9] -> Id
```

The meaning of character classes could be defined in terms of productions and characters, effectively eliminating them from the formalism. For instance, the character class `[a-z]` is completely defined by 26 productions of the form:

```
[a] -> [a-z]  [b] -> [a-z]      ...      [z] -> [a-z]
```

However, this would cause an enormous increase in the number of productions. Therefore, the interpretation of character classes is not defined by translating character classes out of the language. This means that interpretation functions should be extended to character classes.

We do not give the complete specification of character classes and character class arithmethic. A full specification of character classes can be found in Visser (1997b). The normalization defined there ensures that two classes that contain the same elements have the same normal form.

*Characters* A character is a constant of the form $\backslash d_1 \ldots d_n$, where the $d_i$ are decimal digits, denoting the $d_1 \ldots d_n$-th member of some finite, linearly ordered universe of characters. Since specifying characters by their index in some encoding scheme is difficult, we provide easier syntax for specification of characters. Alphanumeric characters (letters and digits) can be specified as themselves. Other visible characters in the ASCII set can be specified by escaping them using a backslash, e.g., `\(` for left parenthesis, `\-` for a hyphen and `\ ` (a backslash followed by a space) for space. The characters `\t` and `\n` represent tabs and newlines. Finally, there are two special characters, `\EOF` and `\TOP`. `\EOF` is the character used to indicate represent the end of a file. `\TOP` is used to represent the largest character in the character universe.

**module** Character-Syntax
**imports** Layout
**exports**
  **sorts** Character NumChar ShortChar
  **lexical syntax**
    [\][0-9]+ → NumChar
    [a-zA-Z0-9] → ShortChar
    [\]~[\000-\037A-Za-mo-su-z0-9] → ShortChar
  **context-free syntax**
    NumChar → Character
    ShortChar → Character
    "\TOP" → Character
    "\EOF" → Character
  **variables**

"c"[*0-9′*]∗ → Character

*Character Classes* A set of characters—a character class—is represented by a list of characters and character ranges between square brackets [ and ]. A list is constructed by an injection of characters into lists and by a right associative binary concatenation operator on lists. Operations on character classes are difference (/), intersection (∧), union (∨) and complement with respect to the complete character set, i.e., the characters in the range `\0-\TOP`, (∼).

**module** Character-Class-Syntax
**imports** Character-Syntax[2.4.2]
**exports**
  **sorts** CharRange CharRanges OptCharRanges CharClass
  **context-free syntax**

| | | |
|---|---|---|
| Character | → CharRange | |
| Character "−" Character | → CharRange | |
| CharRange | → CharRanges | |
| CharRanges CharRanges | → CharRanges | {**right**} |
| "(" CharRanges ")" | → CharRanges | {**bracket**} |
| | → OptCharRanges | |
| CharRanges | → OptCharRanges | |
| "[" OptCharRanges "]" | → CharClass | |
| "∼" CharClass | → CharClass | |
| CharClass "/" CharClass | → CharClass | {**left**} |
| CharClass "∧" CharClass | → CharClass | {**left**} |
| CharClass "∨" CharClass | → CharClass | {**left**} |
| "(" CharClass ")" | → CharClass | {**bracket**} |

  **priorities**
  "∼"CharClass → CharClass > CharClass "/"CharClass → CharClass > CharClass "∧"CharClass → CharClass > CharClass "∨"CharClass → CharClass
  **variables**

| | |
|---|---|
| "*cr*"[*0-9′*]∗ | → CharRange |
| "*cr*""∗"[*0-9′*]∗ | → OptCharRanges |
| "*cr*""+"[*0-9′*]∗ | → CharRanges |
| "*cc*"[*0-9′*]∗ | → CharClass |

*Syntax* The kernel formalism is extended by adding character classes as symbols.

**module** CC-Sdf-Syntax
**imports** Character-Class-Syntax[2.4.2] Kernel-Sdf-Syntax[2.3.1]
**exports**
  **context-free syntax**
  CharClass → Symbol

*Normalization* Character classes can be normalized to a unique normal form by ordering the ranges such that all characters are translated to their numeric equivalent and such that smaller characters are before larger characters and by fusing adjacent or overlapping ranges. For example, the class `[A-ZO-9\%]` has normal form `[\37\48-\57\65-\90]`, because `\37` is the numerical representation of `\%`, `\48-\57` of `0-9`, `\65-\90` of `A-Z` and these do not overlap and are ordered. This normalization is specified in module Character-Class-Normalization that can be found in Visser (1997b).

**module** CC-Sdf-Normalization
**imports** CC-Sdf-Syntax[2.4.2] Character-Class-Normalization
Kernel-Sdf-Normalization[2.3.3]

### 2.4.3 Literals

Literals are abbreviations for fixed lists of characters. For example, the following production uses literals to define the keywords of a conditional statement.

```
"if" Exp "then" Stat "else" Stat -> Stat
```

The meaning of literals is expressed by means of a production that specifies the sequence of characters that makes up the literal. For instance, the meaning of the literals above is expressed by the productions

```
[\105] [\102]                 -> "if"
[\116] [\104] [\101] [\110] -> "then"
[\101] [\108] [\115] [\101] -> "else"
```

Literals that are identifiers starting with a lowercase letter can be specified without the double quotes.

Another useful abbreviation in this category is the definition of the syntax of prefix functions in the form

```
add(Nat, Nat) -> Nat
```

as an abbreviation of

```
"add" "(" Nat "," Nat ")" -> Nat
```

*Syntax* Literals consist of a list of characters between double quotes. For the complete syntax of literals see §A.1. Literals that start with a lowercase letter can be written without quotes, hence the name ‘unquoted literals’. Prefix functions can be declared by means of a special form of productions, where the double quotes for the parentheses and commas can be omitted.

**module** Literals-Sdf-Syntax
**imports** Kernel-Sdf-Syntax[2.3.1] Literals[A.1]
**exports**
**sorts** UQLiteral
**lexical syntax**
[a-z] → UQLiteral
[a-z][A-Za-z0-9\−]∗[A-Za-z0-9] → UQLiteral
**context-free syntax**
UQLiteral → Literal
Literal → Symbol
Literal “(” {Symbol “,”}∗ “)” “→” Symbol Attributes → Production

*Normalization* The normalization function ‘l’ generates a defining production for each literal that is used as a symbol in one of the productions of the grammar.

**module** Literals-Sdf-Normalization
**imports** Literals-Sdf-Syntax$^{2.4.3}$ CC-Sdf-Normalization$^{2.4.2}$
**exports**
  **context-free syntax**
    “l⟦” Grammar “⟧” → Grammar
    literals(SymbolSet) → Productions
    chars(Literal) → Symbols
    symbols({Symbol “,”}∗) → Symbols
  **variables**
    “*c*”[*0-9′*]∗ → CHAR
    “*c*” “+”[*0-9′*]∗ → CHAR+
    “*c*” “∗”[*0-9′*]∗ → CHAR∗
    “*L*” → Literal
    “*γ*”[*0-9′*]∗ → {Symbol “,”}∗
    “*γ*” “+”[*0-9′*]∗ → {Symbol “,”}+
**equations**
Unquoted literals are translated to quoted literals.

$$[1] \qquad \text{uqliteral}(c^{+}) = \text{literal}(\texttt{"\""}\ c^{+}\ \texttt{"\""})$$

The function l⟦_⟧ generates a production for each literal symbol in the grammar. The production generated for a literal $L$ has the form $\alpha \to L$, where $\alpha$ is a list of singleton character classes representing the characters of $L$. This list is produced by the function ‘chars’.

$$[2] \qquad \text{l}[\![\mathcal{G}]\!] = \mathcal{G}\ \text{syntax literals}(\text{symbols}(\mathcal{G}))$$
$$[3] \qquad \text{literals}(\{L\}) = \text{chars}(L) \to L$$
$$[4] \qquad \text{literals}(\{\mathcal{A}\}) = \quad \textbf{otherwise}$$
$$[5] \qquad \text{literals}(\{\}) =$$
$$[6] \qquad \text{literals}(\{\alpha^{+}\ \beta^{+}\}) = \text{literals}(\{\alpha^{+}\}) +\!\!+ \text{literals}(\{\beta^{+}\})$$

The function ‘chars’ scans the characters in the literal string, translating them to short characters. These are then normalized to numeric character codes by character normalization. The third equation tries if the first character of the string is a short-character by normalizing it and then testing whether it has reduced to a numeric character. This works for letters and digits. If this fails, the fourth equation translates the character to an escaped short-character, which succeeds for all other characters. Characters that are already escaped are handled by the second equation.

$$[7] \qquad \text{chars}(\texttt{""}) =$$

$$[8] \qquad \frac{[\text{shortchar}(\texttt{"\textbackslash"}\ c)] = cc}{\text{chars}(\text{literal}(\texttt{"\""}\ \texttt{"\textbackslash"}\ c\ c^{*}\ \texttt{"\""})) = cc +\!\!+ \text{chars}(\text{literal}(\texttt{"\""}\ c^{*}\ \texttt{"\""}))}$$

$$[9] \qquad \frac{[\text{shortchar}(c)] = cc,\ \ cc = [\text{numchar}(c^{+})]}{\text{chars}(\text{literal}(\texttt{"\""}\ c\ c^{*}\ \texttt{"\""})) = cc +\!\!+ \text{chars}(\text{literal}(\texttt{"\""}\ c^{*}\ \texttt{"\""}))}$$

$$[10] \quad \frac{[\text{shortchar}(\texttt{"\textbackslash"}\ c)] = cc, \quad \alpha = \text{chars}(\text{literal}(\texttt{"""}\ c^*\ \texttt{"""}))}{\text{chars}(\text{literal}(\texttt{"""}\ c\ c^*\ \texttt{"""})) \ = \ cc +\!\!+ \alpha} \quad \textbf{otherwise}$$

Prefix function productions are translated to normal productions by enclosing the parentheses and commas in double quotes.

$$
\begin{array}{lrl}
[11] & L(\gamma) \to \mathcal{A}\ \$ = & L\ \texttt{"("} +\!\!+ \text{symbols}(\gamma) +\!\!+ \texttt{")"} \to \mathcal{A}\ \$ \\
[12] & \text{symbols}() = & \\
[13] & \text{symbols}(\mathcal{A}) = & \mathcal{A} \\
[14] & \text{symbols}(\gamma_1^+, \gamma_2^+) = & \text{symbols}(\gamma_1^+) +\!\!+ \texttt{","} +\!\!+ \text{symbols}(\gamma_2^+)
\end{array}
$$

## 2.5 Parse Trees

Now we can define the interpretation of grammars, that is, the well-formed trees characterized by a grammar and the yield of those trees. The general idea is that a context-free production $p = \mathcal{A}_1 \ldots \mathcal{A}_n \to \mathcal{A}_0$ constructs trees of type $\mathcal{A}_0$ labeled with the production $p$ and with a list of direct descendants of type $\mathcal{A}_1 \ldots \mathcal{A}_n$. Such trees are represented by means of terms. The constructor 'appl' builds an application of a production $p$ to a list of trees, i.e., if $T_1 \ldots T_n$ are trees of type $\mathcal{A}_1 \ldots \mathcal{A}_n$ then $\text{appl}(p, [T_1, \ldots, T_n])$ is a tree of type $\mathcal{A}_0$. Parse forests are constructed by representing choice nodes or *ambiguity nodes* by means of the constructor 'amb'. If $T_1 \ldots T_n$ are all trees of the same type $\mathcal{A}$, then $\text{amb}([T_1, \ldots, T_n])$ is an ambiguity node of type $\mathcal{A}$.

To formally define this notion of trees we introduce the notion of terms. We first present the generic term format that is used to encode parse trees and the encoding of symbols and grammars in that format. With those tools in place, we define the well-formedness rules in §2.5.5.

### 2.5.1 Term Format

Van den Brand *et al.* (1997) introduce the generic, annotated term format ATerms for the representation and exchange of structured data. The format is designed such that all kinds of data can be represented in a single, fixed format, with the purpose of exchanging such data between tools and providing generic operations on these data. The definition of the format comes with an extensive library of (higher-order) functions. We will use the ATerm format to represent parse trees.

The syntax of *ATerms* is defined in module ATerms below. Terms are constructed by means of four constructors, i.e., an ATerm is one of the following:

— A *constant* (ACon), which is either an integer constant or a real number constant.

— A *list* of terms (ATermList), which is either empty [], or a list of one or more terms separated by commas between square brackets $[T_1, \ldots, T_n]$. The sort ATerms represents lists of one or more terms separated by commas.

— A *function symbol* (AFun).

— An *application* of a function symbol to a list of one or more terms separated by commas.

Furthermore, each of these constructors can be *annotated* by a list of one or more terms between { and } (Ann). Literals are strings of characters between double quotes. Integer constants are lists of digits and real constants are floating point numbers with an optional exponent. For the syntax of literals, integers and reals see Van den Brand *et al.* (1997).

**module** ATerms
**imports** Literals$^{A.1}$ IntCon RealCon
**exports**
  **sorts** ATerms ATermList ACon AFun ATerm Ann
  **context-free syntax**

| | |
|---|---|
| ATerm | → ATerms |
| ATerm "," ATerms | → ATerms |
| "[" "]" | → ATermList |
| "[" ATerms "]" | → ATermList |
| IntCon | → ACon |
| RealCon | → ACon |
| Literal | → AFun |
| ACon | → ATerm |
| ATermList | → ATerm |
| AFun | → ATerm |
| AFun "(" ATerms ")" | → ATerm |
| "{" ATerms "}" | → Ann |
| ACon Ann | → ATerm |
| ATermList Ann | → ATerm |
| AFun Ann | → ATerm |
| AFun "(" ATerms ")" Ann | → ATerm |

  **variables**

| | |
|---|---|
| "*Ts*"[*0-9′*]∗ | → ATerms |
| "*Tl*"[*0-9′*]∗ | → ATermList |
| "*ACon*"[*0-9′*]∗ | → ACon |
| "*AFun*"[*0-9′*]∗ | → AFun |
| "*T*"[*0-9′*]∗ | → ATerm |
| "*Ann*"[*0-9′*]∗ | → Ann |

### 2.5.2 Constructors for Parse Trees

Function symbols can be literals—strings of characters between double quotes—or identifiers. Specification of the identifiers is not included in the ATerm format. For each application of ATerms, an appropriate set of AFuns should be declared, with the requirement that they are restricted to names of the form `[a-z][a-zA-Z0-9\-]*`. For the representation of grammars, symbols, productions and trees we define the following function symbols.

**module** Kernel-Sdf-Tree-Constructors
**imports** Grammar-Tree-Constructors$^{A.2.1}$
**exports**
  **context-free syntax**
    "prod" → AFun

"no-attrs" $\rightarrow$ AFun
"attrs" $\rightarrow$ AFun
"atr" $\rightarrow$ AFun
"syntax" $\rightarrow$ AFun

"appl" $\rightarrow$ AFun
"amb" $\rightarrow$ AFun

The function symbols 'appl' and 'amb' will be used to represent parse trees. The others will be used in the encoding of grammar structures. Each extension of the kernel that adds new constructors for symbols or grammars should also add the corresponding ATerm function symbols. These are included in §A.2.

### 2.5.3 ATerm Encoding

Now we can encode symbols, grammars and productions as terms. For each sort $\mathcal{S}$ a function aterm($\mathcal{S}$) $\rightarrow$ ATerm is defined that encodes $\mathcal{S}$-expressions as ATerms. The encoding is injective. For each sort $\mathcal{S}$ a decoding function s(ATerm) $\rightarrow \mathcal{S}$ is defined such that s(aterm($s$)) $= s$. Figure 2.1 illustrates the encoding of symbols and productions as ATerms.

The following module defines the encoding of lists of symbols. The encoding of constructors for symbols is defined in a module for each extension of the kernel; see §A.2.

**module** Symbols-ATerms
**imports** Symbols-Projection$^{2.1.1}$ ATerm-Lists
**exports**
**context-free syntax**
aterm(Symbol) $\rightarrow$ ATerm
atermlist(Symbols) $\rightarrow$ ATermList
symbol(ATerm) $\rightarrow$ Symbol
symbols(ATermList) $\rightarrow$ Symbols
**equations**
Encoding lists of symbols.

[1] $$\mathrm{atermlist}(\mathcal{A}) = [\mathrm{aterm}(\mathcal{A})]$$
[2] $$\mathrm{atermlist}(\alpha) = [\,] \quad \textbf{when} \quad \alpha =$$
[3] $$\mathrm{atermlist}(\alpha^+\ \beta^+) = \mathrm{atermlist}(\alpha^+) \mathbin{+\!\!+} \mathrm{atermlist}(\beta^+)$$

Decoding lists of symbols.

[4] $$\mathrm{symbols}([\,]) =$$
[5] $$\mathrm{symbols}([T]) = \mathrm{symbol}(T)$$
[6] $$\mathrm{symbols}([T,\ Ts]) = \mathrm{symbol}(T) \mathbin{+\!\!+} \mathrm{symbols}([Ts])$$

where we have the requirement that

[7] $$\mathrm{symbol}(\mathrm{aterm}(\mathcal{A})) = \mathcal{A}$$

The last equation requires of each future definition that it should be such that the decoding of an encoded symbol gives the original symbol.

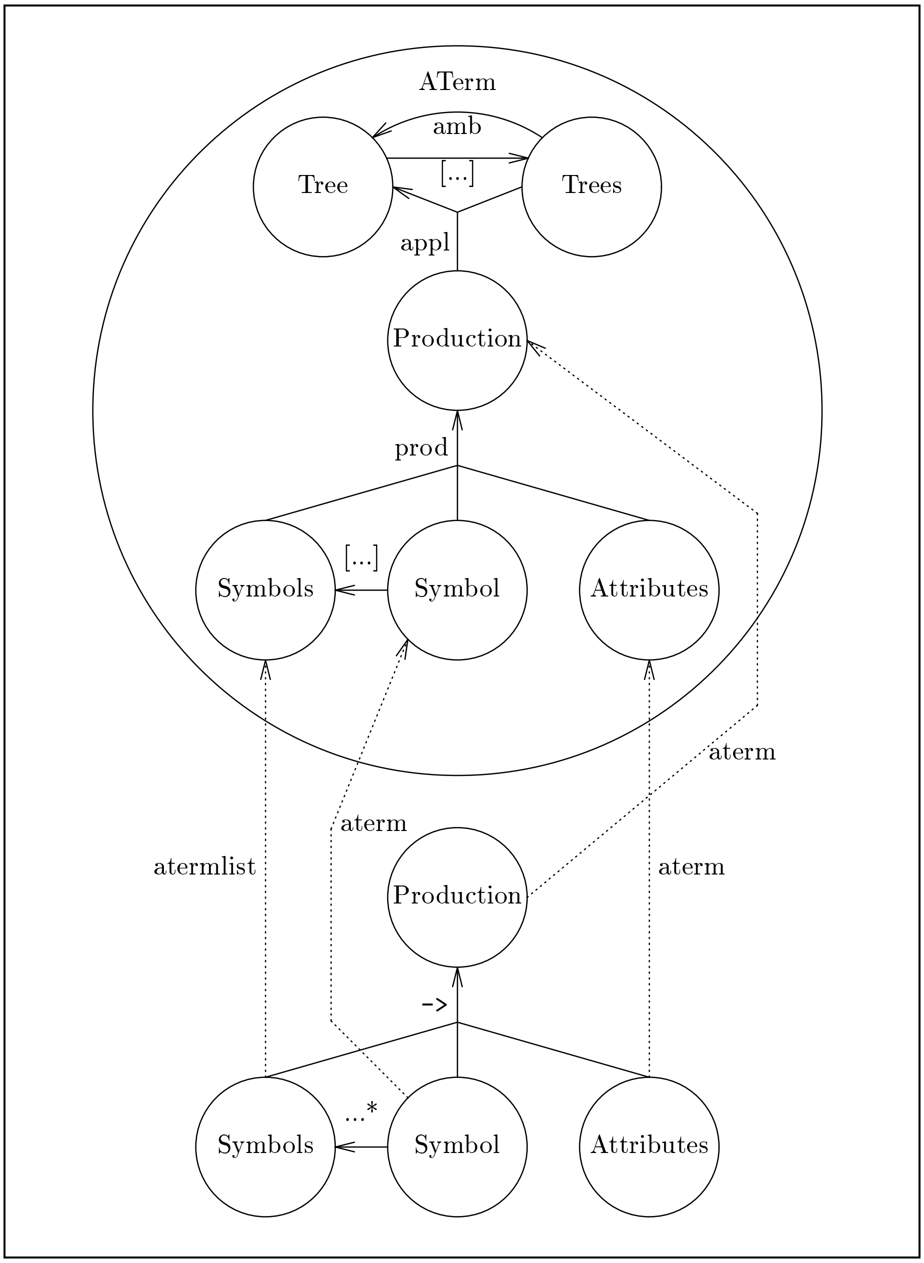



Figure 2.1: Encoding symbols, productions and parse trees in a fixed term format. Grammar domains such as Symbol and Production are mapped (the dotted arrows) onto subsets of the set of ATerms. Parse Trees are another subset of ATerms formed by the constructor 'appl' from a production and a list of trees, or by the constructor 'amb' from a list of trees.

### 2.5.4 Encoding Productions

The following module defines the encoding of productions. As an example, consider the production

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

which is encoded as

```
prod([sort("Exp"),lit("+"),sort("Exp")],
     sort("Exp"),attrs([atr("left")]))
```

A production is represented by the function symbol 'prod' and the attributes of the production are represented by 'attrs'. Note that this makes use of the encoding of sort and literal symbols that is defined in §A.2.

**module** Kernel-Sdf-ATerms
**imports** Kernel-Sdf-Projection$^{2.3.2}$ Kernel-Sdf-Tree-Constructors$^{2.5.2}$
Symbols-ATerms$^{2.5.3}$ Grammar-ATerms$^{A.2.2}$ ATerm-Lists
**exports**
**context-free syntax**

| | |
|---|---|
| aterm(Production) | → ATerm |
| aterm(Attributes) | → ATerm |
| atermlist({Attribute ","}∗) | → ATermList |
| aterm(Attribute) | → ATerm |
| atermlist(Productions) | → ATermList |
| production(ATerm) | → Production |
| attributes(ATerm) | → Attributes |
| attribute(ATerm) | → Attribute |
| productions(ATermList) | → Productions |

**equations**
Encoding productions and attributes.

[1] $\text{aterm}(\alpha \to \mathcal{A}\ \$) = \text{prod}(\text{atermlist}(\alpha), \text{aterm}(\mathcal{A}), \text{aterm}(\$))$
[2] $\text{aterm}() = \text{no-attrs}$
[3] $\text{aterm}(\{attr^*\}) = \text{attrs}(\text{atermlist}(attr^*))$
[4] $\text{atermlist}(attr^*) = [\,]\quad \textbf{when}\ \{attr^*\} = \{\}$
[5] $\text{atermlist}(attr) = [\text{aterm}(attr)]$
[6] $\text{atermlist}(attr_1^+,\ attr_2^+) = \text{atermlist}(attr_1^+) +\!\!+\ \text{atermlist}(attr_2^+)$

Decoding productions and attributes.

[7] $\text{production}(\text{prod}(Tl,\ T,\ T')) = \text{symbols}(Tl) \to \text{symbol}(T)\ \text{attributes}(T')$
[8] $\text{attributes}(\text{no-attrs}) =$
[9] $\text{attributes}(\text{attrs}(Tl)) = \text{attributes}(Tl)$
[10] $\text{attributes}([\,]) = \{\}$
[11] $\text{attributes}([T]) = \{\text{attribute}(T)\}$
[12] $\text{attributes}([T,\ Ts]) = \{\text{attribute}(T)\} +\!\!+\ \text{attributes}([Ts])$

where we have the requirement that

[13] $\text{attribute}(\text{aterm}(attr)) = attr$

Encoding grammars and lists of productions.

[14] $\text{aterm}(\text{syntax } p^*) = \text{syntax}(\text{atermlist}(p^*))$
[15] $\text{atermlist}(p) = [\text{aterm}(p)]$
[16] $\text{atermlist}(p^*) = [\,]$ **when** $p^* =$
[17] $\text{atermlist}(p_1^+\ p_2^+) = \text{atermlist}(p_1^+) +\!\!+ \text{atermlist}(p_2^+)$

Decoding of grammars and lists of productions.

[18] $\text{grammar}(\text{syntax}(Tl)) = \text{syntax productions}(Tl)$
[19] $\text{productions}([\,]) =$
[20] $\text{productions}([T]) = \text{production}(T)$
[21] $\text{productions}([T,\ Ts]) = \text{production}(T) +\!\!+ \text{productions}([Ts])$

### 2.5.5 Well-formed Parse Trees

Now we have prepared all equipment for the characterization of the terms that represent the well-formed parse trees over a grammar. The predicate $\mathcal{G} \vdash T$ determines whether a tree $T$ is well-formed with respect to grammar $\mathcal{G}$. This is defined in terms of $\vdash T : T'$, which checks whether $T$ is a tree of type $T'$. Since a tree contains all type information explicitly in the form of productions, this can be checked without reference to the grammar. The well-formedness with respect to the grammar is then defined by checking that all productions in a tree are actually productions of the grammar.

The main constructor for trees is the function 'appl' that creates an *application* of a context-free production to a list of trees such that the types of the argument trees correspond to the symbols in the left-hand side of the production. As an example, consider the grammar

```
sorts E
syntax
  [a-z]    -> E
  [\+]     -> "+"
  E "+" E -> E {left}
```

The following ATerm is a well-formed parse tree over this grammar for the sentence `a+b`.

```
appl(prod([sort("E"),lit("+"),sort("E")],
          sort("E"), attrs([atr("left")])),
     [appl(prod([char-class([range(97,122)])],
                sort("E"),no-attrs),
           [97]),
      appl(prod([char-class([43])],
                lit("+"),no-attrs),
           [43]),
      appl(prod([char-class([range(97,122)])],
                sort("E"),no-attrs),
           [98])])
```

Observe that the main `appl` has the aterm encoding of the production `E "+" E -> E` {`left`} as first argument and as second argument a list of three trees with types that correspond to the arguments of that production. The numbers at the leafs of the trees denote the ASCII values of the characters, i.e., 97 denotes `a`, 43 denotes `+` and 98 denotes `b`.

Context-free grammars can be ambiguous, i.e., generate more than one tree for a single sentence. The constructor 'amb' is introduced to represent parse forests, i.e., compact representations of sets of parse trees. A term amb($[T_1, \ldots, T_n]$) represents the set of parse trees containing the terms $T_1, \ldots, T_n$ (which can again contain 'amb' nodes). For example, the string `E "*" E "+" E` is ambiguous with respect to the grammar

```
sorts E
syntax
  E "+" E  -> E
  E "*" E  -> E
```

and therefore has the following parse forest that represents the two possible parses, left-associative `(E "*" E) "+" E` and right associative `E "*" (E "+" E)`.

```
amb([appl(prod([sort("E"),lit("+"),sort("E")],
               sort("E"),no-attrs),
          [appl(prod([sort("E"),lit("*"),sort("E")],
                     sort("E"),no-attrs),
                [sort("E"),
                 lit("*"),
                 sort("E")]),
           lit("+"),
           sort("E")]),
      appl(prod([sort("E"),lit("*"),sort("E")],
               sort("E"),no-attrs),
          [sort("E"),
           lit("*"),
           appl(prod([sort("E"),lit("+"),sort("E")],
                     sort("E"),no-attrs),
                [sort("E"),
                 lit("+"),
                 sort("E")])])])
```

Note that in order to reduce the size of the term, the subtrees for `E` and `"*"` and `"+"` are symbols. The tree represents the parse tree for a sentential form.

The yield of a tree is the concatenation of the characters at the leafs of the tree. For instance, the yield of the first tree above is `[97] [43] [98]`, i.e., the list of characters `a+b`.

**module** Kernel-Sdf-Trees
**imports** Kernel-Sdf-ATerms[2.5.4] Kernel-Sdf-Projection[2.3.2]
**exports**
**context-free syntax**

| | |
|---|---|
| Grammar "⊢" ATerm | → Bool |
| "⊢" ATerm ":" ATerm | → Bool |
| yield "⟦" Grammar "⟧" "(" ATerm ")" | → Symbols |

| | |
|---|---|
| args(ATerm) | $\rightarrow$ ATermList |
| type(ATerm) | $\rightarrow$ ATerm |
| prods(ATerm) | $\rightarrow$ Productions |

**variables**

“*Prod*”[*0-9′*]∗ $\rightarrow$ ATerm
“*Res*”[*0-9′*]∗ $\rightarrow$ ATerm
“*Attrs*”[*0-9′*]∗ $\rightarrow$ ATerm
“*Args*”[*0-9′*]∗ $\rightarrow$ ATermList

**equations**

A term $T$ is a well-formed parse tree *over grammar* $\mathcal{G}$, if it is a well-formed tree and if all its productions are productions of $\mathcal{G}$.

[1] $$\frac{\vdash T : \mathrm{type}(T) \wedge \mathrm{prods}(T) \subseteq \mathrm{P}(\mathcal{G}) = \top}{\mathcal{G} \vdash T = \top}$$

Otherwise, the tree is not well-formed.

[2] $$\mathcal{G} \vdash T = \bot \quad \textbf{otherwise}$$

The function ‘prods’ gives the productions of a term, i.e., the list of all productions used in its applications.

$$
\begin{array}{lrl}
[3] & \mathrm{prods}(\mathrm{appl}(Prod, Args)) = & \mathrm{production}(Prod) +\!\!+ \mathrm{prods}(Args) \\
[4] & \mathrm{prods}(\mathrm{amb}(Args)) = & \mathrm{prods}(Args) \\
[5] & \mathrm{prods}([\,]) = & \\
[6] & \mathrm{prods}([T]) = & \mathrm{prods}(T) \\
[7] & \mathrm{prods}([T, Ts]) = & \mathrm{prods}(T) +\!\!+ \mathrm{prods}([Ts]) \\
[8] & \mathrm{prods}(T) = & \quad \textbf{otherwise}
\end{array}
$$

For the definition of $\mathcal{G} \vdash T$ we need several auxiliary functions on terms. The function ‘args’ gives the arguments of a production. The ‘type’ of a production is its result type. The ‘type’ of an application is the result type of its production.

$$
\begin{array}{lrl}
[9] & \mathrm{args}(\mathrm{prod}(Tl, Res, Attrs)) = & Tl \\
[10] & \mathrm{args}(\mathrm{appl}(Prod, Args)) = & Args \\
[11] & \mathrm{type}(\mathrm{prod}(Tl, Res, Attrs)) = & Res \\
[12] & \mathrm{type}(\mathrm{appl}(Prod, Args)) = & \mathrm{type}(Prod) \\
[13] & \mathrm{type}(\mathrm{amb}([Ts])) = & \mathrm{type}(\mathrm{first}([Ts]))
\end{array}
$$

An application is a well-formed term of type $T$ if the type of its production is $T$ and if its arguments are well-formed terms with types that correspond to the argument types of the production of the application.

[14] $$\frac{\mathrm{type}(Prod) = T,\ \vdash Args : \mathrm{args}(Prod) = \top}{\vdash \mathrm{appl}(Prod, Args) : T = \top}$$

A list of trees or symbols is well-formed if each element is well-formed.

$$
\begin{array}{lrl}
[15] & \vdash [\,] : [\,] = & \top \\
[16] & \vdash [T] : [T'] = & \vdash T : T' \\
[17] & \vdash [T, Ts] : [T', Ts'] = & \vdash T : T' \wedge \vdash [Ts] : [Ts']
\end{array}
$$

An ambiguous node is well-formed if all possibilities have the same type.

[18] $\vdash \text{amb}([T]) : T' = \vdash T : T'$

[19] $\vdash \text{amb}([T,\ Ts]) : T' = \vdash T : T' \wedge \vdash \text{amb}([Ts]) : T'$

A symbol term $T$ is a well-formed tree of type $T$. This is used to represent trees for sentential forms.

[20] $$\frac{\text{symbol}(T) = \text{symbol}(T')}{\vdash T : T' \ = \ \top}$$

The yield of a term is the concatenation of all its leaf symbols.

[21] $\text{yield}[\![\mathcal{G}]\!](\text{appl}(Prod,\ Tl)) = \text{yield}[\![\mathcal{G}]\!](Tl)$

[22] $\text{yield}[\![\mathcal{G}]\!](\text{amb}(Tl)) = \text{yield}[\![\mathcal{G}]\!](\text{first}(Tl))$

[23] $\text{yield}[\![\mathcal{G}]\!]([\,]) =$

[24] $\text{yield}[\![\mathcal{G}]\!]([T]) = \text{yield}[\![\mathcal{G}]\!](T)$

[25] $\text{yield}[\![\mathcal{G}]\!]([T,\ Ts]) = \text{yield}[\![\mathcal{G}]\!](T) +\!\!+ \ \text{yield}[\![\mathcal{G}]\!]([Ts])$

[26] $\text{yield}[\![\mathcal{G}]\!](T) = \text{symbol}(T)$ **otherwise**

### 2.5.6 Trees with Characters and Literals

In the previous section we defined character classes and literals as symbols in grammars. Since the meaning of literals is defined in terms of character classes by means of context-free productions, the definition of well-formedness of parse trees does not need to be extended for literals. However, for character classes we have to extend the definition such that a character is a tree with as type any character class that contains it, i.e., a parse tree is a well-formed tree of a character class type if it is a character (represented by a natural number) that is an element of the character class. For example, the term

```
appl(prod([sort("Id"),char-class([range(97,122)])],
          sort("Id"),no-attrs),
     [appl(prod([char-class([range(97,122)])],
                sort("Id"),no-attrs),
           [97]),
      98])
```

is a well-formed parse tree for the identifier `ab`. The definition of well-formedness and yield is extended as follows:

**module** CC-Sdf-Trees
**imports** CC-Sdf-ATerms$^{A.2.2}$ Kernel-Sdf-Trees$^{2.5.5}$
Character-Class-Normalization
**equations**

A character is represented by an integer. The characteristic functions for trees are extended for this new tree constructor. The type of a character is the character code itself. The yield of a character is a character class containing the single character. A character does not contain any productions.

[1] $\text{type}(n) = n$

[2] $\text{yield}[\![\mathcal{G}]\!](n) = [\text{character}(n)]$
[3] $\text{prods}(n) =$

A character code $n$ is a well-formed tree of type $T$ if $T$ represents a character class that contains the character corresponding to $n$.

[4]
$$\frac{\text{symbol}(T) = cc,\ \ \text{character}(n) \in cc = \top}{\vdash n : T \ = \ \top}$$

### 2.5.7 Cyclic Parse Forests

Some grammars generate infinitely many parse trees for a single string. For instance, the grammar

```
syntax
       -> S
   [a] -> S
   S S -> S
```

generates infinitely many trees for the string `a`. The collection of parse trees for strings over such grammars can be finitely represented by means of a cyclic parse forest (Billot and Lang, 1989, Rekers, 1992). However, in the term format defined here we have no provisions for cyclic forests. The parser for SDF2 in Visser (1997e) *does* generate a cyclic parse forest.

In ASF+SDF cyclic structures cannot be expressed in a natural way. This can be simulated by explicitly representing the pointer structure by means of a table of references and tags to represent these references. But this would complicate the entire specification. Since the application of SDF2 will be mainly to non-cyclic grammars we have not gone into the trouble of defining cyclic forests in ASF+SDF.

### 2.5.8 Equality of Trees

We define an equality predicate $\doteq$ on parse trees. At this point this comes down to syntactic equality. Later on we will extend the predicate such that trees that are not syntactically equal can be equal. This will be useful to abstract from certain details in parse trees. For instance, in §3.3 we will introduce parse trees containing layout. This is useful for applications that are aware of layout. But most applications will want to abstract from the specific layout in a tree and consider two trees equal up to layout. Another application is the equality of trees with associative operators such as list concatenation. The definition of the equality predicate below is intended to specify the details of such equality considerations. Furthermore, we define membership of a tree in a parse forest.

**module** Kernel-Sdf-Equality
**imports** Kernel-Sdf-Trees$^{2.5.5}$
**exports**
  **context-free syntax**
    ATerm "$\doteq$" ATerm $\to$ Bool
    ATerm "$\dot\in$" ATerm $\to$ Bool

**equations**

Equality of applications. If the productions are the same two applications are equal if the argument lists are.

[1] $$\frac{Args \doteq Args' = \top}{\mathrm{appl}(Prod,\ Args) \doteq \mathrm{appl}(Prod,\ Args') \ = \ \top}$$

Argument lists are equal if the elements are pairwise equal.

[2] $$[\,] \doteq [\,] = \top$$

[3] $$[T] \doteq [T'] = T \doteq T'$$

[4] $$[T,\ Ts] \doteq [T',\ Ts'] = T \doteq T' \wedge [Ts] \doteq [Ts']$$

An ambiguity node is equal to a tree if it all its possibilities are contained in the tree and vice versa.

[5] $$\frac{\mathrm{amb}([Ts]) \mathrel{\dot\in} T \wedge T \mathrel{\dot\in} \mathrm{amb}([Ts]) = \top}{\mathrm{amb}([Ts]) \doteq T \ = \ \top}$$

[6] $$\frac{\mathrm{amb}([Ts]) \mathrel{\dot\in} T \wedge T \mathrel{\dot\in} \mathrm{amb}([Ts]) = \top}{T \doteq \mathrm{amb}([Ts]) \ = \ \top}$$

If none of the cases above apply the terms are not equal.

[7] $$T_1 \doteq T_2 = \bot \quad \textbf{otherwise}$$

A tree is member of a parse forest (tree containing ambiguities) if it is contained in one of the possibilities of an ambiguity.

[8] $$\frac{T_1 \mathrel{\dot\in} T_2 = \top}{T_1 \mathrel{\dot\in} \mathrm{amb}([T_2]) \ = \ \top}$$

[9] $$\frac{T_1 \mathrel{\dot\in} T_2 \vee T_1 \mathrel{\dot\in} \mathrm{amb}([Ts]) = \top}{T_1 \mathrel{\dot\in} \mathrm{amb}([T_2,\ Ts]) \ = \ \top}$$

An ambiguity is contained in a forest if *all* its possibilities are contained in the forest.

[10] $$\frac{T_1 \mathrel{\dot\in} T_2 = \top}{\mathrm{amb}([T_1]) \mathrel{\dot\in} T_2 \ = \ \top}$$

[11] $$\frac{T_1 \mathrel{\dot\in} T_2 \wedge \mathrm{amb}([Ts]) \mathrel{\dot\in} T_2 = \top}{\mathrm{amb}([T_1,\ Ts]) \mathrel{\dot\in} T_2 \ = \ \top}$$

An application is contained in an application, if the arguments of the first are contained in the arguments of the second.

[12] $$\frac{Args_1 \mathrel{\dot\in} Args_2 = \top}{\mathrm{appl}(Prod,\ Args_1) \mathrel{\dot\in} \mathrm{appl}(Prod,\ Args_2) \ = \ \top}$$

Lists

[13] $$[\,] \mathrel{\dot\in} [\,] = \top$$

[14] $$[T] \mathrel{\dot{\in}} [T'] = T \mathrel{\dot{\in}} T'$$
[15] $$[T,\ Ts] \mathrel{\dot{\in}} [T',\ Ts'] = T \mathrel{\dot{\in}} T' \wedge [Ts] \mathrel{\dot{\in}} [Ts']$$

If none of the cases above apply membership does not hold

[16] $$T_1 \mathrel{\dot{\in}} T_2 = \perp \quad \textbf{otherwise}$$

# 3

# Disambiguation and Abbreviation

In this chapter we present features for disambiguation of ambiguous grammars and abbreviation of common patterns. Priorities are used to disambiguate ambiguous expression syntax, providing support for compact abstract syntax. Priorities are defined by means of an extension of the well-formedness predicate on parse trees. Regular expressions abbreviate common patterns of productions such as lists, optional constructs, alternatives, etc. Regular expressions are defined by generating the defining productions for each expression in the grammar. Lexical and context-free syntax sections separate the definition of tokens and phrases. These are integrated into a single context-free grammar by normalization such that no interference between the two levels is created. Furthermore, the definition of the placement of layout between tokens is handled by this normalization. Follow restrictions and reject productions are provided to express lexical disambiguation rules such as 'prefer longest match' and 'prefer literals'.

## 3.1 Priorities

Context-free grammars can be ambiguous. There are many methods for the disambiguation of context-free grammars. Most programming language oriented formalisms provide some kind of precedence based method. Here we adopt the method of disambiguation by associativity and priority as used in SDF. New with respect to the design of priorities in Heering *et al.* (1989) is (a) disambiguation of lexical syntax by lexical priorities, (b) a more uniform notation for priority declarations, and (c) derivation of productions from priority declarations, which provides a more compact notation by avoiding multiple declarations of productions. A feature not present in SDF2 is the abbreviation of productions in priority declarations by the list of literals of the left-hand side. (For example, `"*" > "+"` as an abbreviation of `E "*" E -> E > E "+" E -> E`.) The reason for this omission is the unclear semantics in combination with modularization. Also $<$ priority-chains are not included in SDF2 because these can also be expressed using $>$ chains.

We first define syntax, projection functions and normalization of priority declarations. In §3.1.4 we describe an extension of the well-formedness predicate on parse trees that characterizes parse trees without priority conflicts.

**Example 1.1** The following grammar defines priority and associativity relations over the syntax of expressions with unary negation and binary operators for exponentiation, multiplication, addition and subtraction. Note that, unlike in SDF, the syntax for arithmetic operators can be defined *inside* the priorities section.

```
syntax
  "(" E ")" -> E {bracket}
priorities
   "-" E   -> E
>  E "^" E -> E {right}
>  E "*" E -> E {left}
> {left:
   E "+" E -> E {assoc}
   E "-" E -> E {left}}
```

This grammar declares that unary $-$ has *higher priority* than ^, which has higher priority than binary $*$, which has higher priority than $+$ and binary $-$. The latter two are mutually left associative as declared by the *group associativity*. The bracket production declares that parentheses can also be used to disambiguate expressions. With this grammar the expression `E--E*E+E-E^E` should be interpreted as `((E-((-E)*E))+E)-(E^E)`. □

### 3.1.1 Syntax

The priorities section of a grammar defines the priority relation > on productions and the associativity relations 'left', 'right', 'assoc', and 'non-assoc'. A priority declaration is either a `>` chain or an associativity declaration. The objects of these declarations are single productions or groups of productions. A group can have an X-associativity, which declares the productions in the group to be mutually X-associative.

The 'bracket' attribute declares a production of the form $l\ \mathcal{A}\ r \rightarrow \mathcal{A}$ {bracket}, with 'l' ('r') denoting the syntax for the left- (right-) bracket, to be the identity function on $\mathcal{A}$. Such productions can be used to explicitly disambiguate some text or to indicate a different disambiguation than the one given by priority rules.

**module** Priority-Sdf-Syntax
**imports** Kernel-Sdf-Syntax[2.3.1]
**exports**
  **sorts** Associativity Group Priority Priorities
  **context-free syntax**

| | |
|---|---|
| "left" | → Associativity |
| "right" | → Associativity |
| "non-assoc" | → Associativity |
| "assoc" | → Associativity |
| "bracket" | → Attribute |
| Associativity | → Attribute |
| Production | → Group |
| "{" Productions "}" | → Group |
| "{" Associativity ":" Productions "}" | → Group |
| {Group ">"}+ | → Priority |
| Group Associativity Group | → Priority |
| {Priority ","}* | → Priorities |

“priorities” Priorities $\rightarrow$ Grammar

**variables**

“*g*”[*0-9′*]$*$ $\rightarrow$ Group
“*gg*$*$”[*0-9′*]$*$ $\rightarrow$ {Group “>”}$*$
“*gg*+”[*0-9′*]$*$ $\rightarrow$ {Group “>”}+
“*pr*”[*0-9′*]$*$ $\rightarrow$ Priority
“*pr*” “$*$”[*0-9′*]$*$ $\rightarrow$ {Priority “,”}$*$
“*pr*” “+”[*0-9′*]$*$ $\rightarrow$ {Priority “,”}+
“*as*”[*0-9′*]$*$ $\rightarrow$ Associativity

### 3.1.2 Projection

The projection function ‘Pr’ yields the list of all priority declarations of a grammar. The projection function ‘$\overline{\mathrm{Pr}}$’ yields the grammar without its priority declarations.

**module** Priority-Sdf-Projection
**imports** Priority-Sdf-Syntax$^{3.1.1}$ Kernel-Sdf-Projection$^{2.3.2}$
**exports**
**context-free syntax**
Priorities “++” Priorities $\rightarrow$ Priorities {**assoc**}
“Pr”(Grammar) $\rightarrow$ Priorities
“$\overline{\mathrm{Pr}}$”(Grammar) $\rightarrow$ Grammar
Priority “$\in$” Priorities $\rightarrow$ Bool
**equations**

Concatenation of priority declarations.

[1] $$pr_1^* \mathbin{+\!\!+} pr_2^* = pr_1^*, pr_2^*$$

The priorities and non-priorities of a grammar.

[2] $$\mathrm{Pr}(\text{priorities } pr^*) = pr^*$$
[3] $$\mathrm{Pr}(\mathcal{G}_1\ \mathcal{G}_2) = \mathrm{Pr}(\mathcal{G}_1) \mathbin{+\!\!+} \mathrm{Pr}(\mathcal{G}_2)$$
[4] $$\mathrm{Pr}(\mathcal{G}) = \quad \textbf{otherwise}$$
[5] $$\overline{\mathrm{Pr}}(\text{priorities } pr^*) = \emptyset$$
[6] $$\overline{\mathrm{Pr}}(\mathcal{G}_1\ \mathcal{G}_2) = \overline{\mathrm{Pr}}(\mathcal{G}_1)\ \overline{\mathrm{Pr}}(\mathcal{G}_2)$$
[7] $$\overline{\mathrm{Pr}}(\mathcal{G}) = \mathcal{G} \quad \textbf{otherwise}$$

Membership of a priority declaration. A pair is member of a declaration if the declaration contains a pair with similar productions. Recall from §2.3.2 that two productions are similar if they are the same except for their attributes, which may be different.

[8] $$\frac{p_1 \cong p_3 = \top,\ \ p_2 \cong p_4 = \top}{p_1 > p_2 \in pr_1^*, p_3 > p_4, pr_2^* \ = \ \top}$$

[9] $$\frac{p_1 \cong p_3 \wedge p_2 \cong p_4 \vee p_1 \cong p_4 \wedge p_2 \cong p_3 = \top}{p_1\ as\ p_2 \in pr_1^*, p_3\ as\ p_4, pr_2^* \ = \ \top}$$

[10] $$pr \in pr^* \ = \ \bot$$
**otherwise**

### 3.1.3 Normalization

The complex syntax for priority declarations can be expressed by means of only binary declarations for the $>$ relation and the associativity relations as follows: (1) Priority chains of the form $p_1 > \ldots > p_n$ are normalized to lists of simple priorities of the form $p_i > p_{i+1}$. This relation is closed transitively. (2) Associativity declarations in attributes and group associativities are expressed as binary associativity declarations of the form $p_1$ *as* $p_2$. (3) The productions that are mentioned in priorities sections are added to the syntax section of the grammar.

**Example 1.2** The normalization of the grammar in Example 1.1 on page 39 is:

```
syntax
  "-" E   -> E
  E "^" E -> E {right}
  E "*" E -> E {left}
  E "+" E -> E {assoc}
  E "-" E -> E {left}
priorities
  "-" E -> E           >     E "^" E -> E {right},
  E "^" E -> E {right} right E "^" E -> E {right},
  E "^" E -> E {right} >     E "*" E -> E {left},
  E "*" E -> E {left}  >     E "+" E -> E {assoc},
  E "*" E -> E {left}  >     E "-" E -> E {left},
  E "*" E -> E {left}  left  E "*" E -> E {left},
  E "+" E -> E {assoc} left  E "-" E -> E {left},
  E "+" E -> E {assoc} assoc E "+" E -> E {assoc},
  E "-" E -> E {left}  left  E "-" E -> E {left}
```

Observe that all productions mentioned in the priorities declaration are also declared as productions in the 'syntax' part. Furthermore, the $>$ chain is made into a binary relation, which is transitively closed. All associativity attributes are expressed by means of binary declarations. □

**module** Priority-Sdf-Normalization
**imports** Priority-Sdf-Syntax$^{3.1.1}$ Booleans Kernel-Sdf-Normalization$^{2.3.3}$
Priority-Sdf-Projection$^{3.1.2}$
**exports**
**context-free syntax**

| | |
|---|---|
| "p⟦" Grammar "⟧" | → Grammar |
| "assoc" "⟦" Grammar "⟧" | → Priorities |
| "assoc" "⟦" Productions "⟧" | → Priorities |
| "syntax" "⟦" Priorities "⟧" | → Grammar |
| "norm" "⟦" Priorities "⟧" | → Priorities |
| "trans" "⟦" Priorities "⟧" | → Priorities |

**equations**

The normalization function p⟦_⟧ extracts syntax information from priorities and priority information from syntax, normalizes the priorities declarations and takes the transitive closure.

[1] $$\frac{\mathrm{Pr}(\mathcal{G}) = pr_1^*,\ \mathrm{norm}[\![pr_1^*]\!] = pr_2^*,\ \overline{\mathrm{Pr}}(\mathcal{G})\ \mathrm{syntax}[\![pr_2^*]\!] = \mathcal{G}'}{\mathrm{p}[\![\mathcal{G}]\!]\ =\ \mathcal{G}'\ \mathrm{priorities\ trans}[\![pr_2^* +\!\!+\ \mathrm{assoc}[\![\mathcal{G}']\!]]\!]}$$

The function norm⟦_⟧ normalizes a priority declaration to a list of pairs of the form $p > p'$ or $p\ as\ p'$ by eliminating >-chains and {_} groups.

$$\text{[2]}\qquad \text{norm}[\![p> p']\!] = p> p'$$
$$\text{[3]}\qquad \text{norm}[\![p\ as\ p']\!] = p\ as\ p',\ p'\ as\ p$$

Each of the priority declarations in the list is normalized.

$$\text{[4]}\qquad \text{norm}[\![]\!] =$$
$$\text{[5]}\qquad \text{norm}[\![p]\!] =$$
$$\text{[6]}\qquad \text{norm}[\![pr_1^+,\ pr_2^+]\!] = \text{norm}[\![pr_1^+]\!] +\!\!+ \text{norm}[\![pr_2^+]\!]$$

A > chain is broken into binary > declarations. The transitive closure defined below ensures that $p_1 > p_3$ if $p_1 > p_2 > p_3$ was specified.

$$\text{[7]}\qquad \text{norm}[\![gg_1^+> gg_2^+> gg_3^+]\!] = \text{norm}[\![gg_1^+> gg_2^+,\ gg_2^+> gg_3^+]\!]$$

Groups and priority. A group is an abbreviation for a pointwise extension of the declared relation to the members of the group.

$$\text{[8]}\qquad \text{norm}[\![\{p\}]\!] =$$
$$\text{[9]}\qquad \text{norm}[\![\{\}> g]\!] =$$
$$\text{[10]}\qquad \text{norm}[\![g> \{\}]\!] =$$
$$\text{[11]}\qquad \text{norm}[\![\{p_1^+\ p_2^+\}> g]\!] = \text{norm}[\![\{p_1^+\}> g,\ \{p_2^+\}> g]\!]$$
$$\text{[12]}\qquad \text{norm}[\![g> \{p_1^+\ p_2^+\}]\!] = \text{norm}[\![g> \{p_1^+\},\ g> \{p_2^+\}]\!]$$
$$\text{[13]}\qquad \text{norm}[\![\{p\}> g]\!] = \text{norm}[\![p> g]\!]$$
$$\text{[14]}\qquad \text{norm}[\![g> \{p\}]\!] = \text{norm}[\![g> p]\!]$$
$$\text{[15]}\qquad \text{norm}[\![\{as : p^*\}> g]\!] = \text{norm}[\![\{as : p^*\},\ \{p^*\}> g]\!]$$
$$\text{[16]}\qquad \text{norm}[\![g> \{as : p^*\}]\!] = \text{norm}[\![\{as : p^*\},\ g> \{p^*\}]\!]$$

Groups and associativity.

$$\text{[17]}\qquad \text{norm}[\![\{\}\ as\ g]\!] = \text{norm}[\![g]\!]$$
$$\text{[18]}\qquad \text{norm}[\![\{p\ p^*\}\ as\ g]\!] = \text{norm}[\![p\ as\ g,\ \{p^*\}\ as\ g]\!]$$
$$\text{[19]}\qquad \text{norm}[\![g\ as\ \{\}]\!] = \text{norm}[\![g]\!]$$
$$\text{[20]}\qquad \text{norm}[\![g\ as\ \{p\ p^*\}]\!] = \text{norm}[\![g\ as\ p,\ g\ as\ \{p^*\}]\!]$$
$$\text{[21]}\qquad \text{norm}[\![\{as_1 : p^*\}\ as_2\ g]\!] = \text{norm}[\![\{as_1 : p^*\},\ \{p^*\}\ as_2\ g]\!]$$
$$\text{[22]}\qquad \text{norm}[\![g\ as_2\ \{as_1 : p^*\}]\!] = \text{norm}[\![\{as_1 : p^*\},\ g\ as_2\ \{p^*\}]\!]$$

Associativity groups are abbreviations. The members of an associativity group are mutually associative with respect to the declared relation. If the group contains a single production, it is taken to define the associativity for that production. Otherwise, the associativities are defined only *between* the members of the group and are not defined reflexively. This makes it possible, for instance, to have a production that is left-associative with respect to itself, but right-associative with respect to a group of other productions.

$$\text{[23]}\qquad \text{norm}[\![\{as : p\}]\!] = p\ as\ p$$
$$\text{[24]}\qquad \text{norm}[\![\{as : p_1\ p_2\}]\!] = p_1\ as\ p_2$$
$$\text{[25]}\qquad \text{norm}[\![\{as : p_1\ p_2\ p^+\}]\!] = \text{norm}[\![p_1\ as\ p_2,\ \{as : p_1\ p^+\},\ \{as : p_2\ p^+\}]\!]$$

The function trans⟦_⟧ takes the transitive closure of the > relation.

$$\text{[26]}\qquad \frac{pr_1^*,\ p_1> p_2,\ pr_2^* = pr^*,\ \ pr^* = pr_3^*,\ p_2> p_3,\ pr_4^*,\ \ p_1> p_3 \in pr^* \neq \top}{\text{trans}[\![pr^*]\!]\ =\ \text{trans}[\![p_1> p_3,\ pr^*]\!]}$$

[27] $\text{trans}[\![pr_1^*,\ pr,\ pr_2^*,\ pr,\ pr_3^*]\!] \ = \ \text{trans}[\![pr_1^*,\ pr,\ pr_2^*,\ pr_3^*]\!]$

[28] $\text{trans}[\![pr^*]\!] \ = \ pr^*$
**otherwise**

The function assoc⟦_⟧ derives associativity declarations from the productions of a grammar. Productions that have an attribute declaring them as left, right, or non-associative produce a declaration of that associativity in the priorities declaration.

[29] $\text{assoc}[\![\mathcal{G}]\!] = \text{assoc}[\![\text{P}(\mathcal{G})]\!]$
[30] $\text{assoc}[\![]\!] =$
[31] $\text{assoc}[\![p_1^+\ p_2^+]\!] = \text{assoc}[\![p_1^+]\!] +\!\!+ \text{assoc}[\![p_2^+]\!]$
[32] $\text{assoc}[\![p]\!] = p\ as\ p$ **when** $p = \alpha \to \mathcal{A}\ \{attr_1^*,\ as,\ attr_2^*\}$
[33] $\text{assoc}[\![p]\!] =$ **otherwise**

The function 'syntax' derives from a priorities declaration the list of all productions referred to in that declaration.

[34] $\text{syntax}[\![]\!] = \emptyset$
[35] $\text{syntax}[\![pr_1^+,\ pr_2^+]\!] = \text{syntax}[\![pr_1^+]\!]\ \text{syntax}[\![pr_2^+]\!]$
[36] $\text{syntax}[\![p_1 > p_2]\!] = \text{syntax}\ p_1\ p_2$
[37] $\text{syntax}[\![p_1\ as\ p_2]\!] = \text{syntax}\ p_1\ p_2$

Merging and ordering of grammars.

[38] $\text{priorities} = \emptyset$
[39] $\text{priorities}\ pr_1^* \diamond \text{priorities}\ pr_2^* = \langle \text{priorities}\ pr_1^*,\ pr_2^*,\ \emptyset \rangle$
[40] $\text{priorities}\ pr^* \diamond \text{syntax}\ p^* = \langle \text{syntax}\ p^*,\ \text{priorities}\ pr^* \rangle$

### 3.1.4 Parse Trees with Priority Conflicts

We extend the notion of well-formedness of parse trees to well-formedness over a grammar with priorities. A tree is well-formed if it is a well-formed context-free tree and if, moreover, it does not contain priority conflicts.

**module** Priority-Sdf-Trees
**imports** Kernel-Sdf-Trees[2.5.5] Priority-Sdf-Projection[3.1.2]
**exports**
  **context-free syntax**
    Grammar "$\vdash_{\text{prio}}$" ATerm → Bool
    conf "⟦" Priorities "⟧" "(" ATerm ")" → Bool
    rootconf "⟦" Priorities "⟧" "(" Production "," ATermList ")" → Bool
    left "⟦" Priorities "⟧" "(" Production "," ATerm ")" → Bool
    middle "⟦" Priorities "⟧" "(" Production "," ATermList ")" → Bool
    right "⟦" Priorities "⟧" "(" Production "," ATerm ")" → Bool
**equations**

We introduce an extension of the notion of well-formedness. A tree is well-formed with respect to the priorities in a grammar, if it is well-formed with respect to the grammar and does not contain a priority conflict.

$$[1] \qquad \frac{\mathcal{G} \vdash T = \top, \ \mathrm{conf}[\![\mathrm{Pr}(\mathcal{G})]\!](T) = \bot}{\mathcal{G} \vdash_{\mathrm{prio}} T \ = \ \top}$$

An application has a conflict if it has a root conflict or if any of its descendants has a conflict.

$$[2] \qquad \frac{\mathrm{rootconf}[\![pr^*]\!](\mathrm{production}(Prod),\ Args) \vee \mathrm{conf}[\![pr^*]\!](Args) = Bool}{\mathrm{conf}[\![pr^*]\!](\mathrm{appl}(Prod,\ Args)) \ = \ Bool}$$

For the other constructors, a tree has a conflict if any descendant has.

$$\begin{array}{ll}
[3] & \mathrm{conf}[\![pr^*]\!](\mathrm{amb}([T])) = \mathrm{conf}[\![pr^*]\!](T) \\
[4] & \mathrm{conf}[\![pr^*]\!](\mathrm{amb}([T,\ Ts])) = \mathrm{conf}[\![pr^*]\!](T) \vee \mathrm{conf}[\![pr^*]\!](\mathrm{amb}([Ts])) \\
[5] & \mathrm{conf}[\![pr^*]\!]([\ ]) = \bot \\
[6] & \mathrm{conf}[\![pr^*]\!]([T]) = \mathrm{conf}[\![pr^*]\!](T) \\
[7] & \mathrm{conf}[\![pr^*]\!]([T,\ Ts]) = \mathrm{conf}[\![pr^*]\!](T) \vee \mathrm{conf}[\![pr^*]\!]([Ts])
\end{array}$$

An application with no descendants does not have a conflict.

$$[8] \qquad \mathrm{rootconf}[\![pr^*]\!](p,\ [\ ]) = \bot$$

An application with more than one descendant has a root conflict if it has a left conflict, a middle conflict or a right conflict.

$$[9] \qquad \mathrm{rootconf}[\![pr^*]\!](p,\ [T,\ Ts]) = \mathrm{left}[\![pr^*]\!](p,\ T) \vee \mathrm{middle}[\![pr^*]\!](p,\ [Ts])$$

An injection, i.e., an application with only one child has a root conflict if its production has higher priority than its child's production.

$$[10] \qquad \frac{\mathrm{production}(Prod) = p_2,\ \ p_1 > p_2 \in pr^* = Bool}{\mathrm{rootconf}[\![pr^*]\!](p_1,\ [\mathrm{appl}(Prod,\ Args)]) \ = \ Bool}$$

A tree has a left conflict if the productions of root and left-most child are mutually right-associative or non-associative, or if the root production has higher priority than the child production.

$$[11] \qquad \frac{\begin{array}{c}\mathrm{production}(Prod) = p_2, \\ p_1 \ \mathrm{right}\ p_2 \in pr^* \vee p_1 \ \mathrm{non\text{-}assoc}\ p_2 \in pr^* \vee p_1 > p_2 \in pr^* = Bool\end{array}}{\mathrm{left}[\![pr^*]\!](p_1,\ \mathrm{appl}(Prod,\ Args)) \ = \ Bool}$$

If the left-most child is an ambiguitity node, the tree has a conflict if there is a left conflict with any of the possibilities of the ambiguity.

$$\begin{array}{ll}
[12] & \mathrm{left}[\![pr^*]\!](p_1,\ \mathrm{amb}([T])) = \mathrm{left}[\![pr^*]\!](p_1,\ T) \\
[13] & \mathrm{left}[\![pr^*]\!](p_1,\ \mathrm{amb}([T,\ Ts])) = \mathrm{left}[\![pr^*]\!](p_1,\ T) \vee \mathrm{left}[\![pr^*]\!](p_1,\ \mathrm{amb}([Ts]))
\end{array}$$

A tree has a middle conflict if the root production has higher priority than any of the middle child productions.

$$[14] \qquad \frac{\mathrm{production}(Prod) = p_2,\ \ p_1 > p_2 \in pr^* \vee \mathrm{middle}[\![pr^*]\!](p_1,\ [Ts]) = Bool}{\mathrm{middle}[\![pr^*]\!](p_1,\ [\mathrm{appl}(Prod,\ Args),\ Ts]) \ = \ Bool}$$

[15] $$\text{middle}[\![pr^*]\!](p, [T]) \ = \ \text{right}[\![pr^*]\!](p, T)$$

[16] $$\text{middle}[\![pr^*]\!](p_1, [\text{amb}([T]), \mathit{Ts}]) \ = \ \text{middle}[\![pr^*]\!](p_1, [T, \mathit{Ts}])$$

[17] $$\frac{\text{middle}[\![pr^*]\!](p_1, [T, \mathit{Ts}]) \vee \text{middle}[\![pr^*]\!](p_1, [\text{amb}([\mathit{Ts}]), \mathit{Ts}]) = \mathit{Bool}}{\text{middle}[\![pr^*]\!](p_1, [\text{amb}([T, \mathit{Ts}]), \mathit{Ts}]) \ = \ \mathit{Bool}}$$

A tree has a right conflict if the productions of root and right-most child are mutually left-associative, non-associative or associative (a synonym for left), or if the root production has higher priority than the child production.

[18] $$\frac{\begin{array}{c}\text{production}(\mathit{Prod}) = p_2, \\ p_1 \text{ left } p_2 \in pr^* \vee p_1 \text{ assoc } p_2 \in pr^* \\ \vee\, p_1 \text{ non-assoc } p_2 \in pr^* \vee p_1 > p_2 \in pr^* = \mathit{Bool}\end{array}}{\text{right}[\![pr^*]\!](p_1, \text{appl}(\mathit{Prod}, \mathit{Args})) \ = \ \mathit{Bool}}$$

The case of an ambiguity as right-most child.

[19] $$\text{right}[\![pr^*]\!](p_1, \text{amb}([T])) \ = \ \text{right}[\![pr^*]\!](p_1, T)$$

[20] $$\frac{\text{right}[\![pr^*]\!](p_1, T) \vee \text{right}[\![pr^*]\!](p_1, \text{amb}([\mathit{Ts}])) = \mathit{Bool}}{\text{right}[\![pr^*]\!](p_1, \text{amb}([T, \mathit{Ts}])) \ = \ \mathit{Bool}}$$

### 3.1.5 Discussion

Here we have described the requirements on parse trees that a parser should produce, i.e., not containing priority conflicts. There are various ways to implement this requirement. One possible scheme that is further discussed in Klint and Visser (1994) is to interpret the priority rules as a filter on parse forests that prunes the subtrees with conflicts. This scheme is used in the parser in the current ASF+SDF Meta-Environment (Heering *et al.*, 1989, Klint, 1993). An advantage of this approach is that disambiguation is decoupled from parsing and that other disambiguation filters could be added. The drawback of the approach is that the parse forest can become very large, which hampers efficiency. Therefore, applying the priority rules as early as possible in the parsing process will increase efficiency. A parser-generation time interpretation of priorities is described in Visser (1995). There the priorities are completely expressed in the parse tables produced by the parser generator. An implementation of this method is discussed in Visser (1997e).

*Other Disambiguation Methods* Disambiguation by priority conflicts is similar to the methods using precedences of Earley (1975) and Aho *et al.* (1975). The latter also describe a method for interpreting these rules in the parser generation process, which is less general than the one in Visser (1995). Disambiguation by priorities as defined in this section is based on the definition of priorities in Heering *et al.* (1989). In that definition a second interpretation of priorities is defined. Parse trees are interpreted as a multi-set of productions and the priorities are interpreted as an ordering of such multi-sets. This ordering is used to make a further selection of trees if the filtering by priority conflicts does not solve all ambiguities.

Subtree exclusion is a disambiguation method introduced by Thorup (1994) that works by specifying a finite set of partial parse trees that are forbidden as subtrees of parse trees yielded by the parser. This method allows a more fine tuned disambiguation than is achievable by the priority scheme. Examples are disambiguation of generic operators and internal arguments. Some problems can not be solved appropriately. The if-then-else ambiguity is solved in the same way as with priorities, which is not correct. In Klint and Visser (1994) these and several other disambiguation methods are studied in the framework of filters on parse forests.

*Brackets* Unparsing is complicated in the presence of priorities. When a parse tree is created by a semantics processor, a rewriter for instance, it might create a well-formed tree that does not satisfy the $\vdash_{\mathrm{prio}}$ predicate, i.e., contains a priority conflict. Such trees are semantically meaningful, but problematic when their yield is considered. Naively translating an abstract syntax tree to a string as described before might lead to a string that, when parsed, does not represent the same tree because it would contain conflicts. To force equivalence of tree and string, brackets should be introduced. In Van den Brand and Visser (1996) the rules for priority conflicts are used to place brackets when unparsing an abstract syntax tree.

## 3.2 Regular Expressions

Certain patterns of context-free productions occur again and again. Examples of such patterns are lists, lists with separators, optional constructs and alternative. For example, a list of one or more identifiers can be specified by the grammar

```
syntax
  Id                -> Id-List
  Id-List Id-List -> Id-List {left}
```

Here a list is defined in terms of two constructors, one for singleton lists and one for concatenation of lists.

Many formalisms provide shortcuts for such patterns by extending the language of context-free grammars with some collection of regular operators on symbols. For instance, BNF provides an alternative at the level of productions, i.e., a production has the form $\mathcal{A} := \mathcal{A}_0|\dots|\mathcal{A}_n$, where the symbol '|' has the meaning of *or*. Extended BNF (EBNF) is the canonical extension of BNF with regular operators. In one formulation, Wirth (1977) adds the operators $\{\mathcal{A}\}$ for iteration and $[\mathcal{A}]$ for optionality. Variations on this notation appear in Lee (1972) and Williams (1982). SDF provides iteration $\mathcal{A}*$ and $\mathcal{A}+$ and $\{\mathcal{A}\ L\}\oplus$ iteration for abbreviation of lists of $\mathcal{A}$s separated by a literal $L$.

In this section we give an extension of context-free productions by a set of regular operators on symbols. In all the approaches mentioned above regular operators are given a special treatment. New in our formulation is the treatment of regular operators as first class citizens. They are nothing but constructors of new symbols that spare the specifier the burden of having to invent new names. As a consequence, a regular expression can occur at all positions where a normal symbol can occur, in particular in the right-hand side of a production.

This approach is motivated by the following considerations: (1) It enables us to express the meaning of regular expressions by means of a normalization of the grammar that adds defining productions for each expression. (2) Our grammars function as signatures for algebraic specifications, where each production represents a function. If regular symbols can not be the result of functions, as is the case in SDF, we still have to define an auxiliary symbol to define a function that yields such a result. For example, suppose that we want to define a function `add` that adds an integer to each integer in a list of integers. In the syntax below we can write this as

```
add(Int, Int*) -> Int*
```

whereas in SDF we should introduce an auxiliary sort `IntList` to represent the result sort of this function.

### 3.2.1 Syntax

We consider the following operators:

— Empty: The symbol () represents the empty string

— Concatenation: The symbol $(\mathcal{A}_1 \ldots \mathcal{A}_n)$ with $n > 2$ denotes the concatenation $a_1 \ldots a_n$ of expressions $a_i$ of type $\mathcal{A}_i$.

— Alternative: The symbol $\mathcal{A}_1 | \ldots | \mathcal{A}_n$, with $n \geq 2$, denotes an expression $a$ of one of the types $\mathcal{A}_i$.

— Optional: The symbol $\mathcal{A}?$ is an optional $\mathcal{A}$

— Iteration: The symbol $\mathcal{A}*$ $(\mathcal{A}+)$ denotes a list $a_1 \ldots a_n$ with $n \geq 0$ $(n \geq 1)$ of expressions $a_i$ of type $\mathcal{A}$.

— Iteration with separator: The symbol $\{\mathcal{A}\ \mathcal{B}\}*$ $(\{\mathcal{A}\ \mathcal{B}\}+)$ denotes a list $a_1 b_1 \ldots b_{n-1} a_n$ with $n \geq 0$ $(n \geq 1)$ of expressions $a_i$ of type $\mathcal{A}$ separated by expressions $b_i$ of type $\mathcal{B}$. Observe that, unlike in SDF, in SDF2 there is no limitation of the symbols that can be used as separators. For example, `{Stat [\;]|[\n]}*` denotes lists of statements separated by semicolons *or* newlines.

— Constrained iteration: The symbol $\{\mathcal{A}\}n+$ with $n \geq 2$ denotes a list $a_1 \ldots a_n$ of $n$ or more expressions $a_i$ of type $\mathcal{A}$. Similarly for $\{\mathcal{A}\ \mathcal{B}\}n+$ with separator $\mathcal{B}$.

— Set expressions: The symbol Set$[\mathcal{A}]$ represents the syntax of set expressions of the form $\{a_1, \ldots, a_n\}$ with the $a_i$ expressions of type $\mathcal{A}$

— Product: The symbol $\mathcal{A}_1 \# \ldots \# \mathcal{A}_n$, with $n \geq 2$, denotes tuples $\langle a_1, \ldots, a_n \rangle$ of expressions $a_i$ of type $\mathcal{A}_i$.

— Functions: The symbol $(\mathcal{A}_1 \ldots \mathcal{A}_n \Rightarrow \mathcal{B})$, with $n \geq 0$, denotes function expressions $f$ that can be used in expressions $f(a_1 \ldots a_n)$ of type $\mathcal{B}$ with the $a_i$ expressions of type $\mathcal{A}_i$.

— Permutation: The symbol $\ll \mathcal{A}_1 \ldots \mathcal{A}_n \gg$ denotes expressions of the form $a_1 \ldots a_n$ such that for each $\mathcal{A}_i$ exactly one of the $a_j$ has type $\mathcal{A}_i$.

The syntax of these operators is defined in the following module. Observe that the empty symbol () and sequences $(\mathcal{A}_1 \ldots \mathcal{A}_n)$ are not defined using a single production "(" Symbol $*$ ")" $\rightarrow$ Symbol because parentheses around a single symbol are already used as brackets; see §2.1.

**module** Regular-Sdf-Syntax
**imports** Kernel-Sdf-Syntax$^{2.3.1}$ IntCon
**exports**
**context-free syntax**

| | | |
|---|---|---|
| "(" ")" | $\rightarrow$ Symbol | |
| "(" Symbol Symbol+ ")" | $\rightarrow$ Symbol | |
| Symbol "?" | $\rightarrow$ Symbol | |
| Symbol "+" | $\rightarrow$ Symbol | |
| Symbol "$*$" | $\rightarrow$ Symbol | |
| "{" Symbol Symbol "}" "+" | $\rightarrow$ Symbol | |
| "{" Symbol Symbol "}" "$*$" | $\rightarrow$ Symbol | |
| "{" Symbol "}" NatCon "+" | $\rightarrow$ Symbol | |
| "{" Symbol Symbol "}" NatCon "+" | $\rightarrow$ Symbol | |
| "Set" "[" Symbol "]" | $\rightarrow$ Symbol | |
| Symbol "#" Symbol | $\rightarrow$ Symbol | {**right**} |
| "(" Symbols "$\Rightarrow$" Symbol ")" | $\rightarrow$ Symbol | |
| Symbol "\|" Symbol | $\rightarrow$ Symbol | {**right**} |
| "$\ll$" Symbols "$\gg$" | $\rightarrow$ Symbol | |

**priorities**

{Symbol "?" $\rightarrow$ Symbol, Symbol "$*$" $\rightarrow$ Symbol, Symbol "+" $\rightarrow$ Symbol, Symbol NatCon "+" $\rightarrow$ Symbol} $>$ Symbol "#"Symbol $\rightarrow$ Symbol $>$ Symbol "|"Symbol $\rightarrow$ Symbol

### 3.2.2 Normalization

We define a normalization function r⟦_⟧ that for each regular expression that is used in the grammar introduces one or more productions that define its meaning. In this interpretation regular expressions form a shorthand for defining extra symbols and productions.

**Example 2.1** The following production defines a single production describing the structure of a block in a While program consisting of an optional declaration followed by a list of statements.

```
syntax
  "begin" (Decl ";")? {Stat ";"}+ "end" -> Stat
```

The normalization of this grammar is:

```
syntax
  "begin" (Decl ";")? {Stat ";"}+ "end" -> Stat
                                        -> (Decl ";")?
  (Decl ";")                            -> (Decl ";")?
  Decl ";"                              -> (Decl ";")
  Stat                                  -> {Stat ";"}+
  {Stat ";"}+ ";" {Stat ";"}+           -> {Stat ";"}+ {left}
```

```
    {Stat ";"}+ ";" {Stat ";"}*           -> {Stat ";"}+
    {Stat ";"}* ";" {Stat ";"}+           -> {Stat ";"}+
    {Stat ";"}* ";" {Stat ";"}*           -> {Stat ";"}* {left}
                                          -> {Stat ";"}*
    {Stat ";"}+                           -> {Stat ";"}*
  priorities
  {left :
    {Stat ";"}+ ";" {Stat ";"}+           -> {Stat ";"}+ {left}
    {Stat ";"}+ ";" {Stat ";"}*           -> {Stat ";"}+
    {Stat ";"}* ";" {Stat ";"}+           -> {Stat ";"}+
    {Stat ";"}* ";" {Stat ";"}*           -> {Stat ";"}* {left}
  }>
    {Stat ";"}+                           -> {Stat ";"}*
```

We see that the meaning of the operators is expressed by means of extra productions. Observe how regular expressions are used as target symbols of productions. □

**module** Regular-Sdf-Normalization
**imports** Regular-Sdf-Syntax[3.2.1] Priority-Sdf-Syntax[3.1.1]
Literals-Sdf-Syntax[2.4.3] Kernel-Sdf-Normalization[2.3.3]
**exports**
**context-free syntax**
"r⟦" Grammar "⟧" → Grammar
"r⟦" Symbols "⟧" → Grammar
alt(Symbol, Symbol) → Grammar
tup(Symbol) → Symbols
perm(Symbols) → Productions
perm3(Symbols, Symbols) → Productions
**equations**
The function r⟦_⟧ adds defining productions for each regular expresssion occurring in one of the productions of the grammar. Existing productions are not affected.

[1] $$\mathrm{r}[\![\mathcal{G}]\!] = \mathcal{G}\ \mathrm{r}[\![\alpha]\!] \quad \textbf{when}\ \{\alpha\} = \mathrm{symbols}(\mathcal{G})$$

Recall that the function 'symbols', defined in §2.3.2, gives the set of all symbols in a grammar. The function r⟦_⟧ generates a grammar for each of the regular expressions in the list of symbols.

[2] $$\mathrm{r}[\![\,]\!] = \emptyset$$

[3] $$\mathrm{r}[\![\alpha^+\ \beta^+]\!] = \mathrm{r}[\![\alpha^+]\!]\ \mathrm{r}[\![\beta^+]\!]$$

*Concatenation* The regular expression $(\alpha)$ is a *symbol* that abbreviates the *concatenation* of the symbols $\alpha$.

[4] $$\mathrm{r}[\![(\,)]\!] = \mathrm{syntax}\ \to (\,)$$

[5] $$\frac{p\ =\ \mathcal{A}\ \alpha^+ \to (\mathcal{A}\ \alpha^+)}{\mathrm{r}[\![(\mathcal{A}\ \alpha^+)]\!]\ =\ \mathrm{syntax}\ p\ \mathrm{r}[\![\mathcal{A}\ \alpha^+]\!]}$$

Note that $\mathrm{r}[\![\mathcal{A}\ \alpha^+]\!]$ recursively produces the productions for regular expressions in the list of symbols $\mathcal{A}\ \alpha^+$.

*Alternative* The *alternative* $\mathcal{A}|\mathcal{B}$ denotes either $\mathcal{A}$ or $\mathcal{B}$. We could thus define $\mathrm{r}[\![\mathcal{A}|\mathcal{B}]\!]$ to yield the productions $\mathcal{A} \to \mathcal{A}|\mathcal{B}$ and $\mathcal{B} \to \mathcal{A}|\mathcal{B}$. However, if one of the alternatives is again an alternative, an unnecessary chain $\mathcal{A} \to \mathcal{A}|\mathcal{B}$ and $\mathcal{A}|\mathcal{B} \to \mathcal{A}|\mathcal{B}|\mathcal{C}$ is created. We would rather have $\mathcal{A} \to \mathcal{A}|\mathcal{B}|\mathcal{C}$. Therefore, we define

[6] $$\mathrm{r}[\![\mathcal{A} \mid \mathcal{B}]\!] = \mathrm{alt}(\mathcal{A} \mid \mathcal{B}, \mathcal{A} \mid \mathcal{B})$$

where the function 'alt' unpacks the alternative until a symbol is reached that is not an alternative.

[7] $$\mathrm{alt}(\mathcal{B}_1 \mid \mathcal{B}_2, \mathcal{A}) = \mathrm{alt}(\mathcal{B}_1, \mathcal{A})\ \mathrm{alt}(\mathcal{B}_2, \mathcal{A})$$
[8] $$\mathrm{alt}(\mathcal{B}, \mathcal{A}) = \mathrm{syntax}\ \mathcal{B} \to \mathcal{A}\ \ \mathrm{r}[\![\mathcal{B}]\!] \quad \textbf{otherwise}$$

*Optional* The *optional construct* $\mathcal{A}?$ is either empty or $\mathcal{A}$.

[9] $$\frac{\begin{array}{rcl} p_1 & = & \to \mathcal{A}?\ , \\ p_2 & = & \mathcal{A} \to \mathcal{A}? \end{array}}{\mathrm{r}[\![\mathcal{A}?]\!] \ = \ \mathrm{syntax}\ p_1\ p_2\ \mathrm{r}[\![\mathcal{A}]\!]}$$

*Iteration* The *iteration* operator $\mathcal{A}+$ denotes lists of *one or more* $\mathcal{A}$'s, i.e., either $\mathcal{A}$ or $\mathcal{A}\ \mathcal{A}$ or $\mathcal{A}\ \mathcal{A}\ \mathcal{A}$ or .... The iteration $\mathcal{A}*$ denotes a list of *zero or more* $\mathcal{A}$'s, i.e., $\epsilon$ (empty) or $\mathcal{A}$ or $\mathcal{A}\ \mathcal{A}$ or $\mathcal{A}\ \mathcal{A}\ \mathcal{A}$ or ... There are several ways to define such lists with productions. It is not sufficient to define a list by means of the productions

$$\begin{array}{rcr} \to \mathcal{A}* & \qquad\qquad & \mathcal{A} \to \mathcal{A}+ \\ \mathcal{A}+ \to \mathcal{A}* & & \mathcal{A}+\mathcal{A} \to \mathcal{A}+ \end{array}$$

The symbols $\mathcal{A}*$ and $\mathcal{A}+$ can be the right-hand side of any production, i.e., lists can be the result of arbitrary functions. Therefore, an $\mathcal{A}*$ expression can also contain function calls and variables. For instance, if a grammar contains the production

```
yield(Tree) -> Symbol*
```

then `yield(T1) yield(T2)` should also be an expression of type `Symbol*` (with `T1` and `T2` expression of type `Tree`). We have the following rules for the composition of list expressions.

0. A single $\mathcal{A}$ is an $\mathcal{A}+$.
1. An $\mathcal{A}+$ followed by an $\mathcal{A}+$ is an $\mathcal{A}+$.
2. An $\mathcal{A}+$ followed by an $\mathcal{A}*$ is an $\mathcal{A}+$.
3. An $\mathcal{A}*$ followed by an $\mathcal{A}+$ is an $\mathcal{A}+$.
4. An $\mathcal{A}*$ followed by an $\mathcal{A}*$ is an $\mathcal{A}*$.
5. An $\mathcal{A}*$ can be empty.
6. An $\mathcal{A}+$ is an $\mathcal{A}*$.

Productions expressing these rules are generated by the following equation. The priorities section declares the concatenation operators to be mutually left-associative. The priority prevents that the empty production and the injection are used vacuously.

[10]
$$\frac{\begin{array}{rcl} p_0 & = & \mathcal{A} \to \mathcal{A}+\ , \\ p_1 & = & \mathcal{A}+\ \mathcal{A}+ \to \mathcal{A}+\ \{\text{left}\}, \\ p_2 & = & \mathcal{A}+\ \mathcal{A}* \to \mathcal{A}+\ , \\ p_3 & = & \mathcal{A}*\ \mathcal{A}+ \to \mathcal{A}+\ , \\ p_4 & = & \mathcal{A}*\ \mathcal{A}* \to \mathcal{A}*\ \{\text{left}\}, \\ p_5 & = & \to \mathcal{A}*\ , \\ p_6 & = & \mathcal{A}+ \to \mathcal{A}* \end{array}}{\begin{array}{rcl} \mathrm{r}[\![\mathcal{A}*]\!] & = & \text{syntax}\ p_0\ p_1\ p_2\ p_3\ p_4\ p_5\ p_6 \\ & & \text{priorities}\ \{\text{left} : p_1\ p_2\ p_3\ p_4\} > \{p_5\ p_6\}\ \mathrm{r}[\![\mathcal{A}]\!] \end{array}}$$

[11]
$$\mathrm{r}[\![\mathcal{A}+]\!] = \mathrm{r}[\![\mathcal{A}*]\!]$$

*Iteration with Separator* The *iteration with separator* operators $\{\mathcal{A}\ \mathcal{B}\}+$ and $\{\mathcal{A}\ \mathcal{B}\}*$ denote iteration of $\mathcal{A}$'s separated by $\mathcal{B}$'s. Their meaning is defined analogously to $\mathcal{A}+$ and $\mathcal{A}*$.

[12]
$$\frac{\begin{array}{rcl} p_0 & = & \mathcal{A} \to \{\mathcal{A}\ \mathcal{B}\}+\ , \\ p_1 & = & \{\mathcal{A}\ \mathcal{B}\}+\ \mathcal{B}\ \{\mathcal{A}\ \mathcal{B}\}+ \to \{\mathcal{A}\ \mathcal{B}\}+\ \{\text{left}\}, \\ p_2 & = & \{\mathcal{A}\ \mathcal{B}\}+\ \mathcal{B}\ \{\mathcal{A}\ \mathcal{B}\}* \to \{\mathcal{A}\ \mathcal{B}\}+\ , \\ p_3 & = & \{\mathcal{A}\ \mathcal{B}\}*\ \mathcal{B}\ \{\mathcal{A}\ \mathcal{B}\}+ \to \{\mathcal{A}\ \mathcal{B}\}+\ , \\ p_4 & = & \{\mathcal{A}\ \mathcal{B}\}*\ \mathcal{B}\ \{\mathcal{A}\ \mathcal{B}\}* \to \{\mathcal{A}\ \mathcal{B}\}*\ \{\text{left}\}, \\ p_5 & = & \to \{\mathcal{A}\ \mathcal{B}\}*\ , \\ p_6 & = & \{\mathcal{A}\ \mathcal{B}\}+ \to \{\mathcal{A}\ \mathcal{B}\}* \end{array}}{\begin{array}{rcl} \mathrm{r}[\![\{\mathcal{A}\ \mathcal{B}\}*]\!] & = & \text{syntax}\ p_0\ p_1\ p_2\ p_3\ p_4\ p_5\ p_6 \\ & & \text{priorities}\ \{\text{left} : p_1\ p_2\ p_3\ p_4\} > p_6\ \mathrm{r}[\![\mathcal{A}\ \mathcal{B}]\!] \end{array}}$$

[13]
$$\mathrm{r}[\![\{\mathcal{A}\ \mathcal{B}\}+]\!] = \mathrm{r}[\![\{\mathcal{A}\ \mathcal{B}\}*]\!]$$

*Constrained Iteration* The iteration operator $\{\mathcal{A}\}n+$ denotes the iteration of *at least* $n$ $\mathcal{A}$s. First of all we define that zero or more $\mathcal{A}$s corresponds to $*$ iteration and that one or more $\mathcal{A}$s corresponds to $+$ iteration. For integers $n \geq 2$ we define $\{\mathcal{A}\}n+$ in terms of $\{\mathcal{A}\}(n-1)+$, and eventually $\mathcal{A}+$, by productions of the form $\mathcal{A}\ \{\mathcal{A}\}(n-1)+ \to \{\mathcal{A}\}n+$.

[14]
$$\{\mathcal{A}\}\ 0+ = \mathcal{A}*$$

[15]
$$\{\mathcal{A}\}\ 1+ = \mathcal{A}+$$

[16]
$$\frac{n \geq 2\ =\ \top,\ \ n-1\ =\ n',\ \ p\ =\ \mathcal{A}\ \{\mathcal{A}\}\ n'+ \to \{\mathcal{A}\}\ n+}{\mathrm{r}[\![\{\mathcal{A}\}\ n+]\!]\ =\ \text{syntax}\ p\ \mathrm{r}[\![\{\mathcal{A}\}\ n'+]\!]}$$

Constrained iteration is defined similarly for lists with separators.

[17]
$$\{\mathcal{A}\ \mathcal{B}\}\ 0+ = \{\mathcal{A}\ \mathcal{B}\}*$$

[18]
$$\{\mathcal{A}\ \mathcal{B}\}\ 1+ = \{\mathcal{A}\ \mathcal{B}\}+$$

[19]
$$\frac{n \geq 2\ =\ \top,\ \ n-1\ =\ n',\ \ p\ =\ \mathcal{A}\ \mathcal{B}\ \{\mathcal{A}\ \mathcal{B}\}\ n'+ \to \{\mathcal{A}\ \mathcal{B}\}\ n+}{\mathrm{r}[\![\{\mathcal{A}\ \mathcal{B}\}\ n+]\!]\ =\ \text{syntax}\ p\ \mathrm{r}[\![\{\mathcal{A}\ \mathcal{B}\}\ n'+]\!]}$$

*Tuples* For the definition of functions that return a tuple of values, new sorts have to be invented. To give sensible types to tuples the notation $\mathcal{A}\#\mathcal{B}$ is introduced. A symbol $\mathcal{A}_1\#\ldots\#\mathcal{A}_n$ denotes a tuple of $\mathcal{A}_1\ldots\mathcal{A}_n$ expressions. A tuple is written as $\langle T_1,\ldots,T_n\rangle$, where the $T_i$ are expresions of type $\mathcal{A}_i$.

[20] $$\frac{\texttt{"<"} +\!\!+\ \mathrm{tup}(\mathcal{A}\ \#\ \mathcal{B})\ +\!\!+\ \texttt{">"}\ =\ \alpha,\ \ p\ =\ \alpha\rightarrow\mathcal{A}\ \#\ \mathcal{B}}{\mathrm{r}[\![\mathcal{A}\ \#\ \mathcal{B}]\!]\ =\ \mathrm{syntax}\ p\ \mathrm{r}[\![\alpha]\!]}$$

The auxiliary function $\mathrm{tup}[\![\_]\!]$ derives the syntax of the body of the tuple by separating the symbols by commas.

[21] $$\mathrm{tup}(\mathcal{A}\ \#\ \mathcal{B}) = \mathrm{tup}(\mathcal{A})\ +\!\!+\ \texttt{","}\ +\!\!+\ \mathrm{tup}(\mathcal{B})$$
[22] $$\mathrm{tup}(\mathcal{A}) = \mathcal{A}\quad \textbf{otherwise}$$

*Sets* The conventional notation for sets is a list of items between { and }. The operator $\mathrm{Set}[\mathcal{A}]$ generates this notation such that if $T_1\ldots T_n$ are expressions of type $\mathcal{A}$, then $\{T_1,\ldots,T_n\}$ is an expression of type $\mathrm{Set}[\mathcal{A}]$.

[23] $$\frac{\alpha\ =\ \texttt{"\{"}\ \{\mathcal{A}\ \texttt{","}\}*\ \texttt{"\}"},\ \ p\ =\ \alpha\rightarrow\mathrm{Set}[\mathcal{A}]}{\mathrm{r}[\![\mathrm{Set}[\mathcal{A}]]\!]\ =\ \mathrm{r}[\![\alpha]\!]\ \mathrm{syntax}\ p}$$

*Functions* Sometimes it is convenient to pass functions around as data. The operator $(\alpha\Rightarrow\mathcal{B})$ can be used to give a type to functions. It denotes the sort of functions from $\alpha$ to $\mathcal{B}$. The operator generates syntax for the prefix application of a function to an argument.

[24] $$\frac{p\ =\ (\alpha\Rightarrow\mathcal{B})\ \texttt{"("}\ \alpha\ \texttt{")"}\rightarrow\mathcal{B}}{\mathrm{r}[\![(\alpha\Rightarrow\mathcal{B})]\!]\ =\ \mathrm{syntax}\ p\ \mathrm{r}[\![\alpha\ \mathcal{B}]\!]}$$

*Permutation* The permutation symbol $\ll\alpha\gg$ denotes any concatenation of the symbols in $\alpha$, i.e., $\beta\rightarrow\ll\alpha\gg$ if $\beta$ is a permutation of $\alpha$.

[25] $$\mathrm{r}[\![\ll\alpha\gg]\!] = \mathrm{syntax}\ \mathrm{perm}(\alpha)\ \mathrm{r}[\![\alpha]\!]$$

The function 'perm' generates the productions for all permutations of a set of symbols. In case the permutation consists of two elements it generates the two productions directly. In case of more elements the function 'perm3' is used to generate permutations.

[26] $$\mathrm{perm}() = \ \rightarrow\ \ll\ \gg$$
[27] $$\mathrm{perm}(\mathcal{A}) = \mathcal{A}\rightarrow\ \ll\mathcal{A}\gg$$
[28] $$\mathrm{perm}(\mathcal{A}\ \mathcal{B}) = \mathcal{A}\ \mathcal{B}\rightarrow\ \ll\mathcal{A}\ \mathcal{B}\gg\ \ \mathcal{B}\ \mathcal{A}\rightarrow\ \ll\mathcal{A}\ \mathcal{B}\gg$$
[29] $$\mathrm{perm}(\alpha) = \mathrm{perm3}(,\alpha)\quad \textbf{when}\ \mid\alpha\mid\ >2\ =\ \top$$

For each symbol in the list a production is generated with that symbol first and a permutation of the other symbols following it.

[30] $$\mathrm{perm3}(\alpha,)\ =$$

[31] $$\frac{p\ =\ \mathcal{A}\ll\alpha\ \beta\gg\ \rightarrow\ \ll\alpha\ \mathcal{A}\ \beta\gg}{\mathrm{perm3}(\alpha,\ \mathcal{A}\ \beta)\ =\ p\ +\!\!+\ \mathrm{perm3}(\alpha\ \mathcal{A},\ \beta)\ +\!\!+\ \mathrm{perm}(\alpha\ \beta)}$$

It should be observed that this is not a very efficient way to implement permutation constructs. It should be adequate for permutations of 2 or 3 elements, though. What is needed in addition to the generation of these productions, is the normalization of the parse trees over these productions to a form that lists the elements in a fixed order such that semantic functions do not also have to deal with all permutations. Cameron (1993) describes an extension of LL(1) parsing for permutation operators. An alternative approach suggested by Cameron (1993) is the introduction of an intermediate symbol representing the union of the symbols in the permutation and a check after parsing that each symbol in the permutation is represented exactly once.

*Discussion* We could have handled several of these regular expressions by translating them to other regular expressions. For instance, optionality can be expressed by means of empty and alternative via the equation $\mathcal{A}? = ()|\mathcal{A}$. In the specification above we have chosen not follow this route. Except for a few cases involving constrained iteration.

### 3.2.3 Equality of Parse Trees with Lists

Since all new constructs are expressed by means of existing constructs—all regular expressions are expressed by means of generated context-free productions—there is no need to extend the definition of well-formedness of parse trees.

We do extend the definition of the equality of trees. This definition makes lists equal modulo associativity of the concatenation operators. It is the basis for matching modulo associativity. We give the equations that should be considered in matching, where a variable $a^+$ ($a^*$) ranges over all constructs of type $\mathcal{A}+$ ($\mathcal{A}*$) and $\epsilon_A$ denotes the tree constructed with $\rightarrow \mathcal{A}*$. Empty sublists are units for concatenation and can be removed.

$$\epsilon_A\ a^+ = a^+ \qquad\qquad a^+\ \epsilon_A = a^+$$
$$\epsilon_A\ a^* = a^* \qquad\qquad a^*\ \epsilon_A = a^*$$

Injections from $\mathcal{A}+$ into $\mathcal{A}*$ can be removed or lifted over concatenations.

$$[a_1^+ \rightarrow \mathcal{A}*]\ [a_1^+ \rightarrow \mathcal{A}*] = [(a_1^+\ a_2^+) \rightarrow \mathcal{A}*]$$
$$a_1^+\ [a_1^+ \rightarrow \mathcal{A}*] = a_1^+\ a_2^+$$
$$[a_1^+ \rightarrow \mathcal{A}*]\ a_1^+ = a_1^+\ a_2^+$$

Right-associative concatenations are equal to left-associative ones. Each of these expressions involves different concatenation operators.

$$a_1^+\ (a_2^+\ a_3^+) = (a_1^+\ a_2^+)\ a_3^+ \qquad\qquad a_1^*\ (a_2^+\ a_3^+) = (a_1^*\ a_2^+)\ a_3^+$$
$$a_1^*\ (a_2^*\ a_3^+) = (a_1^*\ a_2^*)\ a_3^+ \qquad\qquad a_1^*\ (a_2^+\ a_3^*) = (a_1^*\ a_2^+)\ a_3^*$$
$$a_1^+\ (a_2^*\ a_3^+) = (a_1^+\ a_2^*)\ a_3^+ \qquad\qquad a_1^+\ (a_2^*\ a_3^*) = (a_1^+\ a_2^*)\ a_3^*$$
$$a_1^+\ (a_2^+\ a_3^*) = (a_1^+\ a_2^+)\ a_3^* \qquad\qquad a_1^*\ (a_2^*\ a_3^*) = (a_1^*\ a_2^*)\ a_3^*$$

## 3.3 Lexical and Context-Free Syntax

The syntax of a programming language is usually divided into two levels: lexical syntax and context-free syntax. Lexical syntax is the syntax of the tokens, the words of the language, e.g., identifiers, numbers and keywords. Context-free syntax is the syntax of the sentences of a language, e.g., expressions, statements, type declarations and function definitions. The division affects both language definition and implementation. Conventionally lexical analysis is restricted to grammars that can be recognized by finite automata, whereas context-free analysis is implemented with push-down automata. Indeed, it is sometimes not clear whether the division is motivated by the implementation or by an inherent concept of lexical syntax.

In many formalisms the separation is even physical; lexical and context-free syntax are defined with completely different formalisms that are written in separate files. For instance, YACC and METAL use LEX to define lexical syntax. This means lexical definitions in the form of a number of regular expressions are defined in a separate file. Context-free and lexical definitions share a declaration of token symbols that constitutes the interface between the lexical and context-free level. The syntax definition formalism of PCCTS uses a lexical syntax similar to LEX, but provides a mechanism to include token definitions in the same file as the context-free syntax definition. In SDF lexical and context-free syntax are integrated in one formalism, but still uses different semantics for both. All these approaches have in common that the distinction between lexical and context-free syntax is identified with the distinction between regular and context-free grammars.

In SDF2 the inherent distinction between the two categories is that context-free symbols can be *separated by layout*, while lexical symbols cannot. Beyond that difference there is none. The exact same features should be available for the definition of lexical and context-free syntax.

New in this approach is that we provide a uniform notation for the definition of lexical and context-free syntax by means of context-free productions. Grammars for lexical and context-free syntax are normalized to the context-free grammars of the kernel. The distinction between lexical and context-free syntax is completely expressed in the resulting productions.

By treating lexical and context-free syntax identically, every extension that is defined for one is also applicable to the other. For instance, in §3.1 we defined priorities for disambiguation. In Heering *et al.* (1989) these are only defined for context-free syntax. As result of our approach we can also provide lexical disambiguation through priorities. Similarly the regular operators introduced in §3.2 can be used in the definition of both lexical and context-free syntax.

In addition to lexical syntax we also define variables. Variable schemes are used in the specification of the semantics of a language. We also introduce the notion of lexical variables that range over constructs introduced in lexical syntax grammars.

The extension in this section is called *Basic Sdf* because it covers the basic idea of the original SDF: integration of lexical and context-free syntax in one formalism.

**Example 3.1** The following definition introduces a simple expression language with variables and addition.

```
sorts Id Exp
lexical syntax
  [\ \t\n]    -> LAYOUT
```

```
    [a-z]+       -> Id
  context-free syntax
    Id           -> Exp
    Exp "+" Exp -> Exp {left}
  variables
    [i]   -> Id
    [xyz] -> Exp
```

The lexical syntax section defines the syntax of layout as spaces, tabs and newlines and identifiers as lists of one or more lowercase letters. The division in lexical and context-free syntax entails that whitespace can occur between expressions, but not between the letters of an identifier.

To illustrate the power of the integration of lexical and context-free syntax we can extend the layout convention above by introducing C-like comments consisting of a string of comment words between `/*` and `*/`, as follows:

```
  sorts ComWord Comment
  lexical syntax
    ~[\ \t\n\|\/\*]+ -> ComWord
  context-free syntax
    "/*" ComWord* "*/" -> Comment
    Comment            -> LAYOUT
```

Because the definition of comments is part of the context-free syntax, comment words can be separated by layout, including layout. This means that we have specified nested comments, which is useful when commenting out pieces of code already containing comment.

We can extend the defition of comments further to include syntactically correct expressions between bars as comment words.

```
  context-free syntax
    "|" Exp "|" -> ComWord
```

For instance, the text

```
   a + b /* an expression |x + y| denotes
            the addition of |x| and |y| */ + c
```

is a syntactically correct expression over the grammar above denoting the expression `a+b+c` with some comment after `b`. In the conventional setting of a separate scanner and parser this would require a call to the parser from the scanner. One application of syntactically correct program fragments in comments is in typesetting programs for documentation. The typesetting algorithms applied to the real program text can also be applied in typesetting the expressions in comments and crossreferences to program variables can be extended to variables occurring comments. □

### 3.3.1 Syntax

The grammar constructors 'lexical syntax' and 'context-free syntax' introduce the syntax of lexical constructs and context-free constructs, respectively. The grammar constructors 'variables' and 'lexical variables' introduce the syntax of variables over context-free symbols and variables over lexical symbols, respectively. The symbol constructors ⟨_-`LEX`⟩, ⟨_-`CF`⟩ and ⟨_-`VAR`⟩ are used to indicate lexical symbols, context-free symbols and variable symbols, respectively. The special symbol `LAYOUT` is used to define layout.

**module** Basic-Sdf-Syntax
**imports** Kernel-Sdf-Syntax[2.3.1]
**exports**
**context-free syntax**

| | |
|---|---|
| “lexical” “syntax” Productions | → Grammar |
| “context-free” “syntax” Productions | → Grammar |
| “variables” Productions | → Grammar |
| “lexical” “variables” Productions | → Grammar |
| “<” Symbol “-CF” “>” | → Symbol |
| “<” Symbol “-LEX” “>” | → Symbol |
| “<” Symbol “-VAR” “>” | → Symbol |
| “LAYOUT” | → Symbol |

### 3.3.2 Normalization

The normalization function defined below expresses the meaning of lexical and context-free syntax by merging them into a single grammar. To avoid interference between the two levels, the symbols in the lexical syntax are renamed into ⟨_-`LEX`⟩ symbols and the symbols in the context-free syntax are renamed into ⟨_-`CF`⟩ symbols. These ideas are illustrated in the following example.

**Example 3.2** The grammar in Example 3.1 is mapped to the following grammar in which lexical and context-free syntax have been merged.

```
sorts Id Exp ComWord Comment
syntax
  <[a-z]+-LEX>                    -> <Id-LEX>
  <Id-LEX>                        -> <Id-CF>
  <Id-CF>                         -> <Exp-CF>
  <Exp-CF> <LAYOUT?-CF> "+"
           <LAYOUT?-CF> <Exp-CF>  -> <Exp-CF> {left}
  [i]                             -> <<Id-CF>-VAR>
  <<Id-CF>-VAR>                   -> <Id-CF>
  [xyz]                           -> <<Exp-CF>-VAR>
  <<Exp-CF>-VAR>                  -> <Exp-CF>
  <~[\ \t\n\|\047\*]+-LEX>        -> <ComWord-LEX>
  <ComWord-LEX>                   -> <ComWord-CF>
  "|" <LAYOUT?-CF> <Exp-CF>
      <LAYOUT?-CF> "|"            -> <ComWord-CF>
  "/*" <LAYOUT?-CF> <ComWord*-CF>
       <LAYOUT?-CF> "*/"          -> <Comment-CF>
  [\ \t\n]                        -> <LAYOUT-LEX>
  <LAYOUT-LEX>                    -> <LAYOUT-CF>
  <Comment-CF>                    -> <LAYOUT-CF>
  <LAYOUT-CF> <LAYOUT-CF>         -> <LAYOUT-CF> {left}
```

```
                                   -> <LAYOUT?-CF>
<LAYOUT-CF>                        -> <LAYOUT?-CF>
```

The symbols in lexical productions are renamed into ⟨_-LEX⟩ symbols. The symbols in context-free productions are renamed into ⟨_-CF⟩ symbols. The connection between lexical and context-free syntax is made by an injection from each ⟨$\mathcal{A}$-LEX⟩ symbol into the corresponding ⟨$\mathcal{A}$-CF⟩ symbol. □

The following module makes these ideas formal by introducing the normalization function b⟦_⟧.

**module** Basic-Sdf-Normalization
**imports** Basic-Sdf-Syntax$^{3.3.1}$ Regular-Sdf-Syntax$^{3.2.1}$ Priority-Sdf-Syntax$^{3.1.1}$
Kernel-Sdf-Normalization$^{2.3.3}$
**exports**
**context-free syntax**

| | |
|---|---|
| "b⟦" Grammar "⟧" | → Grammar |
| "$\mathrm{b_{aux}}$⟦" Grammar "⟧" | → Grammar |
| "<" Symbols "-LEXs" ">" | → Symbols |
| "<" Production "-LEX" ">" | → Production |
| "<" Productions "-LEXs" ">" | → Productions |
| "<" Grammar "-LEX" ">" | → Grammar |
| "<" Symbols "-CFs" ">" | → Symbols |
| "<" Production "-CF" ">" | → Production |
| "<" Productions "-CFs" ">" | → Productions |
| "<" Grammar "-CF" ">" | → Grammar |
| "<" Productions "-VARs" ">" | → Productions |
| "<" Productions "-LEXVARs" ">" | → Productions |

**equations**

The normalization function b⟦_⟧ integrates lexical and context-free syntax. It applies the auxiliary function $\mathrm{b_{aux}}$ to each subgrammar of a grammar to transform lexical and context-free sections into normal production sections by renaming symbols and separating context-free symbols by ⟨LAYOUT?-CF⟩, which entails that two tokens can optionally be separated by ⟨LAYOUT?-CF⟩. Context-free layout is a list of lexical layout. Concatenation of layout is defined by the production added by the function b⟦_⟧.

[1] $$\frac{p = \langle\texttt{LAYOUT-CF}\rangle\ \langle\texttt{LAYOUT-CF}\rangle \to \langle\texttt{LAYOUT-CF}\rangle\ \{\text{left}\}}{\mathrm{b}[\![\mathcal{G}]\!]\ =\ \text{syntax}\ p\ \mathrm{b_{aux}}[\![\mathcal{G}]\!]}$$

The default rule declares that unless otherwise stated $\mathrm{b_{aux}}$ does not affect a grammar. Below we deal with the exceptions.

[2] $$\mathrm{b_{aux}}[\![\emptyset]\!] = \emptyset$$
[3] $$\mathrm{b_{aux}}[\![\mathcal{G}_1\ \mathcal{G}_2]\!] = \mathrm{b_{aux}}[\![\mathcal{G}_1]\!]\ \mathrm{b_{aux}}[\![\mathcal{G}_2]\!]$$
[4] $$\mathrm{b_{aux}}[\![\mathcal{G}]\!] = \mathcal{G}\quad \textbf{otherwise}$$

*Lexical Syntax* Lexical syntax grammars are translated to normal syntax grammars by encoding the symbols of the grammar to ⟨$\mathcal{A}$-LEX⟩ symbols. Furthermore, for each symbol appearing in a lexical syntax section an injection from the lexical into the context-free symbol is added.

[5] $\mathrm{b_{aux}}[\![\text{lexical syntax } p^*]\!] = \langle \text{syntax } p^*\texttt{-LEX}\rangle$

[6] $\langle \alpha\texttt{-LEXs}\rangle = \quad \textbf{when } \alpha =$
[7] $\langle \mathcal{A}\texttt{-LEXs}\rangle = \langle \mathcal{A}\texttt{-LEX}\rangle$
[8] $\langle \alpha^+\ \beta^+\texttt{-LEXs}\rangle = \langle \alpha^+\texttt{-LEXs}\rangle +\!\!+ \langle \beta^+\texttt{-LEXs}\rangle$
[9] $\langle \alpha \to \mathcal{A}\ \$\texttt{-LEX}\rangle = \langle \alpha\texttt{-LEXs}\rangle \to \langle \mathcal{A}\texttt{-LEX}\rangle\ \$$
[10] $\langle \alpha \to \mathcal{A}\ \$\texttt{-LEXs}\rangle = \langle \alpha \to \mathcal{A}\ \$\texttt{-LEX}\rangle\ \langle \mathcal{A}\texttt{-LEXs}\rangle \to \langle \mathcal{A}\texttt{-CF}\rangle$
[11] $\langle p^*\texttt{-LEXs}\rangle = \quad \textbf{when } p^* =$
[12] $\langle p_1^+\ p_2^+\texttt{-LEXs}\rangle = \langle p_1^+\texttt{-LEXs}\rangle +\!\!+ \langle p_2^+\texttt{-LEXs}\rangle$
[13] $\langle \text{syntax } p^*\texttt{-LEX}\rangle = \text{syntax } \langle p^*\texttt{-LEXs}\rangle$
[14] $\langle \mathcal{G}_1\ \mathcal{G}_2\texttt{-LEX}\rangle = \langle \mathcal{G}_1\texttt{-LEX}\rangle\ \langle \mathcal{G}_2\texttt{-LEX}\rangle$
[15] $\langle \emptyset\texttt{-LEX}\rangle = \emptyset$

*Context-free Syntax* Context-free syntax is treated similarly to lexical syntax. All symbols in the production are mapped to $\langle \mathcal{A}\texttt{-CF}\rangle$ symbols. The important difference is that each adjacent pair of symbols in the left-hand side of a production is separated by the symbol $\langle \texttt{LAYOUT?-CF}\rangle$.

[16] $\mathrm{b_{aux}}[\![\text{context-free syntax } p^*]\!] = \langle \text{syntax } p^*\texttt{-CF}\rangle$

[17] $\langle \alpha\texttt{-CFs}\rangle = \quad \textbf{when } \alpha =$
[18] $\langle \mathcal{A}\texttt{-CFs}\rangle = \langle \mathcal{A}\texttt{-CF}\rangle$
[19] $\langle \alpha^+\ \beta^+\texttt{-CFs}\rangle = \langle \alpha^+\texttt{-CFs}\rangle +\!\!+ \langle \texttt{LAYOUT?-CF}\rangle +\!\!+ \langle \beta^+\texttt{-CFs}\rangle$
[20] $\langle \alpha \to \mathcal{A}\ \$\texttt{-CF}\rangle = \langle \alpha\texttt{-CFs}\rangle \to \langle \mathcal{A}\texttt{-CF}\rangle\ \$$
[21] $\langle p\texttt{-CFs}\rangle = \langle p\texttt{-CF}\rangle$
[22] $\langle p^*\texttt{-CFs}\rangle = \quad \textbf{when } p^* =$
[23] $\langle p_1^+\ p_2^+\texttt{-CFs}\rangle = \langle p_1^+\texttt{-CFs}\rangle +\!\!+ \langle p_2^+\texttt{-CFs}\rangle$
[24] $\langle \text{syntax } p^*\texttt{-CF}\rangle = \text{syntax } \langle p^*\texttt{-CFs}\rangle$
[25] $\langle \mathcal{G}_1\ \mathcal{G}_2\texttt{-CF}\rangle = \langle \mathcal{G}_1\texttt{-CF}\rangle\ \langle \mathcal{G}_2\texttt{-CF}\rangle$
[26] $\langle \emptyset\texttt{-CF}\rangle = \emptyset$

*Variables* Variables and lexical variables grammars introduce tokens that have the status of variables. The symbol constructor $\langle \mathcal{A}\texttt{-VAR}\rangle$ is used to denote variables over the symbol $\mathcal{A}$. The left-hand sides of variable productions are interpreted as lexical syntax. The lexical value produced by such a left-hand side is given the type of a variable over the symbol in the right-hand side of the production. For each production in a variables grammar, two productions are generated. The first interprets the left-hand side of the production as a lexical pattern, i.e., the symbols on the left-hand side are lexical symbols and no layout between symbols can occur. The right-hand side is $\langle\langle \mathcal{A}\texttt{-CF}\rangle\texttt{-VAR}\rangle$ indicating that the pattern is a variable over the context-free symbols $\mathcal{A}$. The second production injects $\langle \mathcal{A}\texttt{-CF}\rangle$ variables into $\langle \mathcal{A}\texttt{-CF}\rangle$ such that a variable can occur whereever an $\langle \mathcal{A}\texttt{-CF}\rangle$ can occur.

[27] $\mathrm{b_{aux}}[\![\text{variables } p^*]\!] = \text{syntax } \langle p^*\texttt{-VARs}\rangle$
[28] $\langle \texttt{-VARs}\rangle =$
[29] $\langle p_1^+\ p_2^+\texttt{-VARs}\rangle = \langle p_1^+\texttt{-VARs}\rangle +\!\!+ \langle p_2^+\texttt{-VARs}\rangle$

[30] $$\langle \alpha \rightarrow \mathcal{A}\ \texttt{\$-VARs} \rangle \;=\; \langle \alpha\texttt{-LEXs} \rangle \rightarrow \langle\langle \mathcal{A}\texttt{-CF} \rangle\texttt{-VAR} \rangle\ \texttt{\$} \quad \langle\langle \mathcal{A}\texttt{-CF} \rangle\texttt{-VAR} \rangle \rightarrow \langle \mathcal{A}\texttt{-CF} \rangle$$

Lexical variables are treated similarly, but their result sort is the corresponding lexical sort.

[31] $$\text{baux}[\![\text{lexical variables } p^*]\!] = \text{syntax } \langle p^*\texttt{-LEXVARs} \rangle$$
[32] $$\langle \texttt{-LEXVARs} \rangle =$$
[33] $$\langle p_1^+\ p_2^+\texttt{-LEXVARs} \rangle = \langle p_1^+\texttt{-LEXVARs} \rangle +\!\!+ \langle p_2^+\texttt{-LEXVARs} \rangle$$

[34] $$\langle \alpha \rightarrow \mathcal{A}\ \texttt{\$-LEXVARs} \rangle \;=\; \langle \alpha\texttt{-LEXs} \rangle \rightarrow \langle\langle \mathcal{A}\texttt{-LEX} \rangle\texttt{-VAR} \rangle\ \texttt{\$} \quad \langle\langle \mathcal{A}\texttt{-LEX} \rangle\texttt{-VAR} \rangle \rightarrow \langle \mathcal{A}\texttt{-LEX} \rangle$$

*Ordering Grammars* The following equations specify the ordering of grammars, where the following order is obtained: lexical syntax, context-free syntax, lexical variables, and variables. We only show two of the equations, the other cases are similar.

[35] $$\frac{\mathcal{G}_1 = \text{context-free syntax } p_1^*,\;\; \mathcal{G}_2 = \text{context-free syntax } p_2^*}{\mathcal{G}_1 \diamond \mathcal{G}_2 \;=\; \langle \text{context-free syntax } p_1^*\ p_2^*, \emptyset \rangle}$$

[36] $$\frac{\mathcal{G}_1 = \text{context-free syntax } p_1^*,\;\; \mathcal{G}_2 = \text{lexical syntax } p_2^*}{\mathcal{G}_1 \diamond \mathcal{G}_2 \;=\; \langle \text{lexical syntax } p_2^*, \text{context-free syntax } p_1^* \rangle}$$

### 3.3.3 Parse Trees

Since we have expressed the meaning of lexical syntax and context-free syntax in terms of normal syntax productions, we do not have to extend the definition of parse trees, except for the encoding of symbols and grammars in the ATerm format. See §A.2 for the encoding and decoding of the newly introduced constructs. This entails that trees for lexical and context-free syntax have the same form. In particular, the structure assigned to lexical tokens by the grammar is retained in parse trees for tokens.

We will refine the equality predicate on trees such that layout is ignored. In considering whether two trees are equivalent it is likely that we do not want to consider layout. For this purpose it is not required to first translate a parse tree to an abstract syntax tree. It suffices to define two arbitrary layout trees as equivalent, as is done in the following extension of the equality predicate on trees.

**module** Basic-Sdf-Equality
**imports** Kernel-Sdf-Equality$^{2.5.8}$ Basic-Sdf-ATerms$^{A.2.2}$
Regular-Sdf-ATerms$^{A.2.2}$
**equations**

[1] $$\frac{\text{symbol}(\text{type}(T_1)) = \langle \texttt{LAYOUT-CF} \rangle,\;\; \text{symbol}(\text{type}(T_2)) = \langle \texttt{LAYOUT-CF} \rangle}{T_1 \doteq T_2 \;=\; \top}$$

[2] $$\frac{\text{symbol}(\text{type}(T_1)) = \langle \texttt{LAYOUT?-CF} \rangle,\;\; \text{symbol}(\text{type}(T_2)) = \langle \texttt{LAYOUT?-CF} \rangle}{T_1 \doteq T_2 \;=\; \top}$$

### 3.3.4 Discussion

*Lexical Layout* In some languages, such as `FORTRAN`, tokens can contain some kind of layout. In Heering *et al.* (1989) the symbol `IGNORE` is introduced for this purpose. This can be dealt with by separating the symbols in a lexical production by a *lexical layout* symbol just as this is done with context-free productions. This is not done in the current version because for most languages this is not necessary, but it is straightforward to add this feature to the normalization above.

*Implementation* A conventional implementation of parsers for lexical and context-free syntax is based on a separate scanner and parser. Such an implementation can be achieved for grammars as introduced here by separating productions for ⟨_-LEX⟩ and ⟨_-VAR⟩ symbols from productions for ⟨_-CF⟩ symbols and generating a scanner based on finite automata for the first set of productions and by generating a parser for the second set of productions based on push-down automata. Scanner and parser communicate through some shared buffer-like data-structure. A requirement for this approach is that the lexical productions form a regular grammar. This can be enforced by specifying static constraints on lexical productions.

The parser generator for SDF2 described in Visser (1997e) does not depend on a separate scanner. Instead 'lexical analysis', i.e., parsing according to the productions for ⟨_-LEX⟩ symbols, is incorporated in the parser. To cope with ambiguities and lookahead, generalized LR parsing is used. A similar approach is described by Salomon and Cormack (1989, 1995) under the name scannerless parsing using conventional LR techniques.

## 3.4 Restrictions

When a distinction is made between lexical and context-free syntax, lexical ambiguities have to be solved before tokens can be sent to the parser. This is usually done by applying rules such as 'prefer longest match', 'prefer keywords' and 'prefer variables'.

By removing this distinction, as we did in the previous section, lexical ambiguities can be dealt with in the same way as context-free ambiguities. For example, in §3.1 we defined disambiguation by priorities, which applies both to lexical and context-free syntax. Furthermore, many lexical ambiguities are solved by considering the context in which tokens occur. For instance, the well-known problem of distinguishing an occurrence of the subrange 1..10 from two consecutive occurrences of the real numbers 1. and .10 in Pascal is solved automatically, because ranges and reals do not occur in the same context in the grammar.

However, not all lexical ambiguities can be solved by context or by means of priorities. Some lexical ambiguities need to be solved by rules such as 'prefer longest match' and 'prefer literals'.

In this section we introduce two extensions of context-free grammars that are aimed at lexical disambiguation: *follow restrictions* and *reject productions*. A follow restriction $\mathcal{A} \not\to cc$ declares that the symbol $\mathcal{A}$ can not be followed by any character in the character class $cc$. A reject production $\alpha \to \mathcal{A}$ {reject} declares that any tree of type $\mathcal{A}$ should be rejected if there exists a tree with the same yield that has this reject production as root production. These constructs suffice for expressing most lexical disambiguation rules.

**Example 4.1** The definition of a simple expression language with nested comments in Example 3.1 contains two lexical ambiguities. First, the definition of lists of comment-words `ComWord*` is ambiguous. The string `abc` can be one comment word (a list of characters from the class `~[\ \t\n\|\/]`), but it can also be considered as a list of two comment-words `ab` and `c` or as `a` and `bc` or as `a` and `b` and `c`. We want to express that the longest possible comment-word should be selected. Second, the definition of identifiers and variables for identifiers and expressions overlap, i.e., `x` can be either an identifier or an expression of sort `Exp`. Here we want to express the rule 'prefer variables' that selects a variable over a lexical. These ambiguities are solved by the following rules:

```
lexical restrictions
  ComWord -/- ~[\ \t\n\|\/]
syntax
  <<Id-CF>-VAR>  -> <Id-CF> {reject}
  <<Exp-CF>-VAR> -> <Id-CF> {reject}
```

The first rule states that a comment-word should not be followed by any of the characters in `~[\ \t\n\|\/]`. This solves the problem because it rules out all parses, except the one in which `abc` is one word. The last two rules state that variables should be preferred over identifiers. □

### 3.4.1 Syntax

A follow restriction has the form $\mathcal{A}$-/-$cc$. Follow restrictions are declared in a grammar starting with the keyword 'restrictions' followed by a list of restrictions. A reject production is a normal production attributed with the attribute 'reject'.

**module** Restrictions-Sdf-Syntax<br>
**imports** CC-Sdf-Syntax[2.4.2]<br>
**exports**<br>
  **sorts** Restriction Restrictions<br>
  **context-free syntax**<br>
    Symbols "-/-" CharClass → Restriction<br>
    Restriction∗ → Restrictions<br>
    "restrictions" Restrictions → Grammar<br>
    "reject" → Attribute<br>
  **variables**<br>
    "*restr*"[*0-9′*]∗ → Restriction<br>
    "*restr∗*"[*0-9′*]∗ → Restriction∗<br>
    "*restr+*"[*0-9′*]∗ → Restriction+

### 3.4.2 Projection

The function 'R' gives the restrictions of a grammar. The function $\pi_{\mathcal{A}}$ looks up the restrictions for some symbol.

**module** Restrictions-Sdf-Projection

**imports** Restrictions-Sdf-Syntax$^{3.4.1}$
**exports**
**context-free syntax**
Restrictions "++" Restrictions → Restrictions {**right**}
"R"(Grammar) → Restrictions
$\pi$ "_" Symbol "(" Restrictions ")" → CharClass
**equations**
Concatenation of restrictions.

$$[1] \qquad \mathit{restr}_1^* +\!\!+\ \mathit{restr}_2^* = \mathit{restr}_1^*\ \mathit{restr}_2^*$$

The restrictions of a grammar.

$$[2] \qquad \mathrm{R}(\text{restrictions } \mathit{restr}^*) = \mathit{restr}^*$$
$$[3] \qquad \mathrm{R}(\mathcal{G}_1\ \mathcal{G}_2) = \mathrm{R}(\mathcal{G}_1) +\!\!+\ \mathrm{R}(\mathcal{G}_2)$$
$$[4] \qquad \mathrm{R}(\mathcal{G}) = \quad \textbf{otherwise}$$

The restrictions for a symbol.

$$[5] \qquad \pi_{\mathcal{A}}() = []$$
$$[6] \qquad \pi_{\mathcal{A}}(\ \not-\ cc\ \mathit{restr}^*) = \pi_{\mathcal{A}}(\mathit{restr}^*)$$
$$[7] \qquad \pi_{\mathcal{A}}(\mathcal{A}\ \alpha \not-\ cc\ \mathit{restr}^*) = cc \vee \pi_{\mathcal{A}}(\alpha \not-\ cc\ \mathit{restr}^*)$$
$$[8] \qquad \pi_{\mathcal{A}}(\mathcal{B}\ \alpha \not-\ cc\ \mathit{restr}^*) = \pi_{\mathcal{A}}(\alpha \not-\ cc\ \mathit{restr}^*) \quad \textbf{otherwise}$$

### 3.4.3 Normalization

No special normalization is needed for restrictions except the normal ordering and merging of grammars.

**module** Restrictions-Sdf-Normalization
**imports** Restrictions-Sdf-Syntax$^{3.4.1}$ CC-Sdf-Normalization$^{2.4.2}$
**equations**
Merging and ordering of grammars.

$$[1] \quad \text{restrictions } \mathit{restr}_1^* \diamond \text{restrictions } \mathit{restr}_2^* = \langle \text{restrictions } \mathit{restr}_1^*\ \mathit{restr}_2^*, \emptyset \rangle$$
$$[2] \quad \text{restrictions } \mathit{restr}^* \diamond \text{syntax } p^* = \langle \text{syntax } p^*, \text{restrictions } \mathit{restr}^* \rangle$$

### 3.4.4 Discussion

The disambiguation rules presented above are derived from similar rules introduced by Salomon and Cormack (1989). The adjacency restriction of Salomon and Cormack (1989) is more general. It has the form $\mathcal{A} \not- \mathcal{B}$ and declares that symbols $\mathcal{A}$ and $\mathcal{B}$ should not be adjacent. Since this may require arbitrary long lookahead, we have chosen for the simpler follow restrictions, which can be implemented by restricting the lookahead of productions. The implementation of reject productions in SGLR parsing described in Visser (1997e) is more general than the implementation based on noncanonical SLR(1) parsing of Salomon and Cormack (1989).

We have not presented the interpretation of follow restrictions and reject productions as disambiguation devices. Follow restrictions can be interpreted as an extension of the well-formedness predicate on parse trees. If a follow restriction applies to a symbol, for any tree with that symbol as type, the character immediately next to the right-most character of its yield should not be contained in the restriction. For a discussion of the semantics of reject productions see Visser (1997e).

In the current situation lexical disambiguation rules have to be invented by the user. In SDF lexical disambiguation is completely taken care of in the scanner by means of a number of heuristics. These heuristics do cause problems in a number of cases. Therefore, it is attractive to have complete control over lexical disambiguation as is provided by restrictions and reject productions introduced here. However, it would be desirable if for most cases the necessary restrictions could be derived automatically from the grammar. Although some schemes have been considered, it is not yet clear how the derivation rules should be defined.

# 4

# Renaming and Modularization

In this chapter we introduce a module mechanism for reusing parts of syntax definitions. In order to adapt imported modules to specific applications and to avoid name clashes, a renaming mechanism is provided that can be used to rename symbols and productions. The renaming mechanism is also used in the definition of symbol aliases that can be used to define abbreviated names for large regular expressions. Renamings are also used to define symbol parameterization of modules.

## 4.1 Renamings

In the previous sections we have presented a number of features that enable more concise definition of syntax than plain context-free grammars. The grammars that can be defined are long monolithic lists of productions. To promote reuse of grammars we will introduce in §4.3 a module layer on top of grammars, such that parts of a language definition can be reused in various other definitions. To make the opportunities for reuse even greater we introduce here a renaming operator on grammars. Renamings enable the adaptation of a generic grammar to specific needs by renaming sorts and productions. A renaming is either a symbol renaming $\mathcal{A} \Rightarrow \mathcal{B}$ that renames $\mathcal{A}$ to $\mathcal{B}$ or a production renaming $p_1 \Rightarrow p_2$ that renames $p_1$ to $p_2$. For example, the renaming

```
[Key => Var   Value => Term   Table => Subst
 lookup(Table, Key) -> Value
     => Subst "[" Var "]" -> Term ]
```

specifies the renaming of symbols `Key` and `Value` to `Var` and `Term`, respectively, and the renaming of the production `lookup(Table, Key) -> Value` to `Subst "[" Var "]" -> Term`.

Once we have defined renamings on grammars we can apply them in several situations: renaming of imported modules, symbol parameters of modules and symbol aliases. These will be the subject of the next sections.

### 4.1.1 Syntax

A renaming is a list of symbol renamings of the form $\mathcal{A} \Rightarrow \mathcal{B}$ and production renamings of the form $p_1 \Rightarrow p_2$.

**module** Renaming-Sdf-Syntax
**imports** Kernel-Sdf-Syntax[2.3.1]
**exports**
  **sorts** Renaming Renamings
  **context-free syntax**
    "[" Renaming∗ "]" → Renamings
    Symbol "⇒" Symbol → Renaming
    Production "⇒" Production → Renaming
  **variables**
    "$\rho$"[*0-9'*]∗ → Renamings
    "$\rho$""∗"[*0-9'*]∗ → Renaming∗
    "$\rho$""+"[*0-9'*]∗ → Renaming+

The only requirement on production renamings is that if $\alpha \rightarrow \mathcal{A} \Rightarrow \beta \rightarrow \mathcal{B}$ is a renaming, then $\alpha$ and $\beta$ should be similar, i.e., the non-terminal parts should correspond. This entails that production renamings can only be used to rename literals between the arguments—the 'syntax'—and not the order of the arguments.

### 4.1.2 Projection

We define two projection functions for looking up the value of a symbol or a production in a list of renamings.

**module** Renaming-Sdf-Projection
**imports** Renaming-Sdf-Syntax[4.1.1]
**exports**
  **context-free syntax**
    Renamings "++" Renamings → Renamings {**right**}
    "$\pi$" "_" Symbol "(" Renamings ")" → Symbol
    "$\pi$" "_" Production "(" Renamings ")" → Production
    "(" Symbols "⇒" Symbols ")" → Renamings
**equations**
Concatenation of renamings.

[1] $$[\rho_1^*] \mathbin{+\!\!+} [\rho_2^*] = [\rho_1^* \; \rho_2^*]$$

Looking up the renaming of a symbol in a list of renamings.

[2] $$\pi_{\mathcal{A}}([\mathcal{A} \Rightarrow \mathcal{B} \; \rho^*]) = \mathcal{B}$$
[3] $$\pi_{\mathcal{A}}([\mathcal{A}' \Rightarrow \mathcal{B} \; \rho^*]) = \pi_{\mathcal{A}}([\rho^*]) \quad \textbf{otherwise}$$
[4] $$\pi_{\mathcal{A}}([p \Rightarrow p' \; \rho^*]) = \pi_{\mathcal{A}}([\rho^*])$$
[5] $$\pi_{\mathcal{A}}([]) = \mathcal{A}$$

Looking up the renaming of a production in a list of renamings.

[6] $$\pi_p([]) = p$$
[7] $$\pi_p([p \Rightarrow p' \; \rho^*]) = p'$$
[8] $$\pi_{\alpha \,\rightarrow\, \mathcal{A} \,\$}([\alpha \rightarrow \mathcal{A} \; \Rightarrow \beta \rightarrow \mathcal{B} \;\; \rho^*]) = \beta \rightarrow \mathcal{B} \; \$$$
[9] $$\pi_p([p' \Rightarrow p'' \; \rho^*]) = \pi_p([\rho^*]) \quad \textbf{otherwise}$$
[10] $$\pi_p([\mathcal{A} \Rightarrow \mathcal{B} \; \rho^*]) = \pi_p([\rho^*])$$

Abbreviation of a renaming of a list of symbols into another list of symbols.

[11] $$( \Rightarrow ) = []$$
[12] $$(\alpha \Rightarrow ) = []$$
[13] $$( \Rightarrow \beta) = []$$
[14] $$(\mathcal{A}\ \alpha \Rightarrow \mathcal{B}\ \beta) = [\mathcal{A} \Rightarrow \mathcal{B}] +\!\!+ (\alpha \Rightarrow \beta)$$

This will be used for the instantiation of a list of formal parameters with a list of actual parameters.

### 4.1.3 Normalization

Now we can define the application of a renaming to a grammar. For each sort $\mathcal{S}$ we define an application function $(\mathcal{S})$ Renamings $\rightarrow \mathcal{S}$ that applies a renaming to constructs of sort $\mathcal{S}$. We start by defining the renaming of symbols and productions. The rest is mainly a distribution of the renamings function over the constructs building a grammar.

**module** Kernel-Sdf-Renaming
**imports** Renaming-Sdf-Projection[4.1.2] Kernel-Sdf-Projection[2.3.2]
**exports**
  **context-free syntax**
    "(" Symbol ")" Renamings → Symbol
    "[" Symbol "]" Renamings → Symbol
    "(" Symbols ")*" Renamings → Symbols
    "(" Production ")" Renamings → Production
    "(" Productions ")*" Renamings → Productions
    "(" Grammar ")" Renamings → Grammar
**equations**
Renaming a symbol. If the symbol is defined in the renaming it is replaced by its value in the renaming. Otherwise, the renaming is applied recursively to the subsymbols of the symbol, which is done by the function $[\_]\rho$.

[1] $$(\mathcal{A})\ \rho = \mathcal{B} \quad \textbf{when}\ \ \pi_{\mathcal{A}}(\rho) = \mathcal{B},\ \ \mathcal{A} \neq \mathcal{B}$$
[2] $$(\mathcal{A})\ \rho = [\mathcal{A}]\ \rho \quad \textbf{otherwise}$$

Renaming a production works similarly. If the production is defined in the renaming it is replaced by its value. Otherwise, its symbols are renamed.

[3] $$(p)\ \rho = p' \quad \textbf{when}\ \ \pi_p(\rho) = p',\ \ p' \neq p$$
[4] $$(\alpha \rightarrow \mathcal{A}\ \$)\ \rho = (\alpha\ )\!*\ \rho \rightarrow (\mathcal{A})\ \rho\ \$ \quad \textbf{otherwise}$$

For all other grammar constructs, renaming is a homomorphism that applies the renamings to symbols and productions contained in the structure.
Renaming lists of symbols.

[5] $$(\alpha\ )\!*\ \rho = \quad \textbf{when}\ \ \alpha =$$
[6] $$(\alpha^+\ \beta^+\ )\!*\ \rho = (\alpha^+\ )\!*\ \rho +\!\!+ (\beta^+\ )\!*\ \rho$$
[7] $$(\mathcal{A}\ )\!*\ \rho = (\mathcal{A})\ \rho$$

Renaming lists of productions.

[8] $$(p\ )\!*\ \rho = (p)\ \rho$$

[9] $$(p_1^* \ )* \ \rho = \quad \textbf{when} \ \ p_1^* =$$
[10] $$(p_1^+ \ p_2^+ \ )* \ \rho = (p_1^+ \ )* \ \rho +\!\!+ (p_2^+ \ )* \ \rho$$

Renaming grammars.

[11] $$(\emptyset) \ \rho = \emptyset$$
[12] $$(\mathcal{G}_1 \ \mathcal{G}_2) \ \rho = (\mathcal{G}_1) \ \rho \ (\mathcal{G}_2) \ \rho$$
[13] $$(\text{syntax} \ p^*) \ \rho = \text{syntax} \ (p^* \ )* \ \rho$$

The application of a renaming to a renaming denotes the composition of the renamings.

**module** Renaming-Sdf-Renaming
**imports** Kernel-Sdf-Renaming[4.1.3]
**exports**
  **context-free syntax**
    "(" Renamings ")" Renamings → Renamings
**equations**

A renaming $\rho_2$ applied to a renaming $\rho_1$ $(\rho_1)\rho_2$ denotes the composition of the renamings, i.e., $(x)(\rho_1)\rho_2 = ((x)\rho_1)\rho_2$. This can be expressed by means of a single renaming by renaming the targets of $\rho_1$ with $\rho_2$ and adding $\rho_2$ at the end of the list of renamings.

[1] $$([]) \ \rho = \rho$$
[2] $$([\mathcal{A} \Rightarrow \mathcal{B} \ \rho^*]) \ \rho = [\mathcal{A} \Rightarrow (\mathcal{B}) \ \rho] +\!\!+ ([\rho^*]) \ \rho$$
[3] $$([p \Rightarrow p' \ \rho^*]) \ \rho = [p \Rightarrow (p') \ \rho] +\!\!+ ([\rho^*]) \ \rho$$

For each of the extensions of the kernel we have to extend the renaming functions to the new constructors. See §A.3 for the specification of these extensions.

### 4.1.4 Renaming Trees

If well-formed trees exist over a grammar that is renamed, the trees have to be renamed as well, if they have to be reused in the same context as the renamed grammar. For example, if equations over a grammar are defined, the equations must be renamed as well. Therefore, we extend the definition of renaming to parse trees.

**module** Renaming-Sdf-Trees
**imports** Kernel-Sdf-Trees[2.5.5] CC-Sdf-Trees[2.5.6] Basic-Sdf-Trees
    Regular-Sdf-Trees Kernel-Sdf-Renaming[4.1.3]
    Literals-Sdf-Normalization[2.4.3]
**exports**
  **context-free syntax**
    "(" ATerm ")" Renamings → ATerm
    rnargs(ATermList, ATermList) → ATermList
    mktree(Literal) → ATerm
    chartrees(Symbols) → ATermList
**equations**

Renaming an application. If the production is defined in the renaming, rename the arguments and then rename the literals in the argument terms according to the new production.

$$[1] \quad \frac{\begin{array}{c}\text{aterm}((\text{production}(\mathit{Prod}))\ \rho) = \mathit{Prod}',\ \ \mathit{Prod}' \neq \mathit{Prod},\\ (\mathit{Args})\ \rho = \mathit{Args}',\ \ \text{rnargs}(\text{args}(\mathit{Prod}'),\ \mathit{Args}') = \mathit{Args}''\end{array}}{(\text{appl}(\mathit{Prod},\ \mathit{Args}))\ \rho\ =\ \text{appl}(\mathit{Prod}',\ \mathit{Args}'')}$$

If the production is not defined in the renaming, then only rename the arguments.

$$[2] \quad (\text{appl}(\mathit{Prod},\ \mathit{Args}))\ \rho = \text{appl}(\mathit{Prod},\ (\mathit{Args})\ \rho) \quad \textbf{otherwise}$$

Renaming is a homomorphism over the other tree constructors.

$$\begin{array}{ll}
[3] & (n)\ \rho = n\\
[4] & (\text{amb}(\mathit{Args}))\ \rho = \text{amb}((\mathit{Args})\ \rho)\\
[5] & (\mathit{Tl})\ \rho = [\,] \quad \textbf{when} \ \ \mathit{Tl} = [\,]\\
[6] & ([T])\ \rho = [(T)\ \rho]\\
[7] & ([T,\ \mathit{Ts}])\ \rho = [(T)\ \rho,\ \mathit{Ts}'] \quad \textbf{when} \ \ [\mathit{Ts}'] = ([\mathit{Ts}])\ \rho
\end{array}$$

Renaming the arguments.

$$[8] \quad \text{rnargs}([\,],\ [\,]) = [\,]$$

Insert literals of the new pattern.

$$[9] \quad \frac{\text{symbol}(\text{first}([\mathit{Ts}_1])) = L}{\text{rnargs}([\mathit{Ts}_1],\ \mathit{Tl}_2)\ =\ \text{mktree}(L) : \text{rnargs}(\text{rest}([\mathit{Ts}_1]),\ \mathit{Tl}_2)}$$

Skip literals of the old tree.

$$[10] \quad \frac{\text{symbol}(\text{type}(\text{first}([\mathit{Ts}_2]))) = L}{\text{rnargs}(\mathit{Tl}_1,\ [\mathit{Ts}_2])\ =\ \text{rnargs}(\mathit{Tl}_1,\ \text{rest}([\mathit{Ts}_2]))}$$

Copy layout from old tree to new tree if layout is requested in new pattern.

$$[11] \quad \frac{\begin{array}{c}\text{symbol}(\text{first}([\mathit{Ts}_1])) = \langle \texttt{LAYOUT?-CF} \rangle,\\ \text{symbol}(\text{type}(\text{first}([\mathit{Ts}_2]))) = \langle \texttt{LAYOUT?-CF} \rangle\end{array}}{\text{rnargs}([\mathit{Ts}_1],\ [\mathit{Ts}_2])\ =\ \text{first}([\mathit{Ts}_2]) : \text{rnargs}(\text{rest}([\mathit{Ts}_1]),\ \text{rest}([\mathit{Ts}_2]))}$$

Insert empty layout in the new tree.

$$[12] \quad \frac{\begin{array}{c}\text{symbol}(\text{first}([\mathit{Ts}_1])) = \langle \texttt{LAYOUT?-CF} \rangle,\\ \text{symbol}(\text{type}(\text{first}([\mathit{Tl}_2]))) \neq \langle \texttt{LAYOUT?-CF} \rangle\end{array}}{\begin{array}{rl}\text{rnargs}([\mathit{Ts}_1],\ \mathit{Tl}_2)\ =& \text{appl}(\text{aterm}(\ \to \langle \texttt{LAYOUT?-CF} \rangle\ ),\ [\,])\\ & : \text{rnargs}(\text{rest}([\mathit{Ts}_1]),\ \mathit{Tl}_2)\end{array}}$$

Skip layout of the old tree.

$$[13] \quad \frac{\begin{array}{c}\text{symbol}(\text{first}(\mathit{Tl}_1)) \neq \langle \texttt{LAYOUT?-CF} \rangle,\\ \text{symbol}(\text{type}(\text{first}([\mathit{Ts}_2]))) = \langle \texttt{LAYOUT?-CF} \rangle\end{array}}{\text{rnargs}(\mathit{Tl}_1,\ [\mathit{Ts}_2])\ =\ \text{rnargs}(\mathit{Tl}_1,\ \text{rest}([\mathit{Ts}_2]))}$$

In the other cases there is no layout or literal in either list. This means that it concerns an argument tree that should be copied from the old tree to the renamed tree.

$$[14] \quad \text{rnargs}([\mathit{Ts}_1], [\mathit{Ts}_2]) \; = \; \text{first}([\mathit{Ts}_2]) : \text{rnargs}(\text{rest}([\mathit{Ts}_1]), \text{rest}([\mathit{Ts}_2])) \quad \textbf{otherwise}$$

The function 'mktree' constructs a tree for a literal $L$, by constructing the production according to the definition in §2.4.3 and by generating the list of character codes.

$$[15] \quad \frac{\text{chars}(L) = \alpha,\ \text{aterm}(\alpha \to L\ ) = \mathit{Prod}}{\text{mktree}(L) \; = \; \text{appl}(\mathit{Prod}, \text{chartrees}(\alpha))}$$

From a list of singleton character classes generate a term list of integers representing the character codes.

$$[16] \quad \text{chartrees}() = [\,]$$
$$[17] \quad \text{chartrees}([\mathtt{c}]\ \alpha) = \text{int}(\mathtt{c}) : \text{chartrees}(\alpha)$$

### 4.1.5 Discussion

It would be desirable that renaming preserves well-formedness, i.e., if a tree $T$ is well-formed under some grammer $\mathcal{G}$, it should also be well-formed when renamed with some renaming $\rho$. That is, we want that

$$\mathcal{G} \vdash T \Rightarrow (\mathcal{G})\rho \vdash (T)\rho$$

In fact we would like that renaming preserves the structure defined by a grammar, i.e., renaming the trees generated by a grammar gives the same trees as those generated by the renamed grammar:

$$(\mathcal{T}[\![\mathcal{G}]\!])\rho = \mathcal{T}[\![(\mathcal{G})\rho]\!]$$

Unfortunately, this is not the case for all renamings. If the argument sorts of a production are renamed using a production renaming, but the sorts are not renamed independently, the arguments of an application with that production have the wrong type after renaming. For instance, the renaming

```
[E "+" E -> E => Set "&" Set -> Set]
```

will change the notation of addition on the sort `E` into a binary operator `&` on the sort `Set`. Other constructs for sort `E` will still have type `E` after renaming, including the arguments of the `&` operator, which will hence not be well-formed. It is sufficient to require that in such cases the corresponding symbol renamings are present as well, i.e., the renaming

```
[E => Set E "+" E -> E => Set "&" Set -> Set]
```

does preserve well-formedness.

Also the interaction between regular expressions and renamings spoils the preservation property. For instance, consider the renaming

```
[{Int ","}* => {Int ";"}*]
```

that is intended to rename lists of integers separated by commas into lists separated by semicolons. This will rename all symbols `{Int ","}*`, but it will not rename the concatenation operators for this sort. The renaming

```
[{Int ","}* => {Int ";"}*
   {Int ","}* "," {Int ","}* -> {Int ","}*
=> {Int ";"}* ";" {Int ";"}* -> {Int ";"}*
   {Int ","}+ "," {Int ","}* -> {Int ","}+
=> {Int ";"}+ ";" {Int ";"}* -> {Int ";"}+
   {Int ","}* "," {Int ","}+ -> {Int ","}+
=> {Int ";"}* ";" {Int ";"}+ -> {Int ";"}+
   {Int ","}+ "," {Int ","}+ -> {Int ","}+
=> {Int ";"}+ ";" {Int ";"}+ -> {Int ";"}+
]
```

is a well-formedness preserving renaming that does have the intended effect.

In all these cases correct renamings can be given that will preserve well-formedness and achieve the intended renaming, but these examples show that care has to be taken when writing down renamings. Ideally we would like to restrict the renamings such that the preservation property holds. It might also be possible to complete a renaming to guarantee well-formedness preservation as in the examples above. This is a matter for further study.

## 4.2 Aliases

The regular expressions introduced in §3.2 provide a way to concisely declare a number of productions without actually having to write them down. A problem with these regular expressions is that they can become rather large. This is a property that might make their use unattractive. Therefore, we introduce symbol aliases. An alias declaration introduces a short name for a complicated regular expression. All occurences of the alias are replaced by their meaning. For example, the declarations

```
aliases
  {Term ","}*            -> Terms
  {Var  ","}*            -> Vars
  Set[(Var "|->" Term)] -> Subst
```

introduce `Terms` and `Vars` as aliases for lists of `Term` and `Var`, respectively, and `Subst` as an alias for sets of pairs of variables and terms. This entails that all operations generated for list constructs also apply to `Terms` and `Vars` and all operations generated for sets apply to `Subst`.

Aliases are defined using the renamings of the previous section. An alias $\mathcal{A} \rightarrow \mathcal{B}$ induces a grammar renaming $[\mathcal{B} \Rightarrow \mathcal{A}]$, which is applied to the entire grammar. Why then introduce this extra feature if we already have renamings? Renamings apply to a fixed grammar. Only the grammar to which the renaming is applied, including all imported grammars, is affected. An alias is a renaming of a symbol that also affects all modules that import the alias.

### 4.2.1 Syntax

An *alias* grammar consists of a list of aliases of the form $\mathcal{A} \rightarrow \mathcal{B}$ that define the symbol $\mathcal{B}$ to be an alias of symbol $\mathcal{A}$.

**module** Alias-Sdf-Syntax
**imports** Kernel-Sdf-Syntax$^{2.3.1}$
**exports**
  **sorts** Alias Aliases
  **context-free syntax**
    "aliases" Aliases → Grammar
    Symbol "→" Symbol → Alias
    Alias∗ → Aliases
  **variables**
    "*al*"[*0-9′*]∗ → Alias
    "*al*∗"[*0-9′*]∗ → Alias∗
    "*al*+"[*0-9′*]∗ → Alias+

### 4.2.2 Projection

Concatenation of alias lists. Projection of the aliases and non-alias parts of a grammar.

**module** Alias-Sdf-Projection
**imports** Alias-Sdf-Syntax$^{4.2.1}$
**exports**
  **context-free syntax**
    Aliases "++" Aliases → Aliases {**right**}
    "Al"(Grammar) → Aliases
    "$\overline{\text{Al}}$"(Grammar) → Grammar
**equations**
The function 'Al' gives all alias declarations of a grammar, '$\overline{\text{Al}}$' the grammar without alias declarations.

[1] $$al_1^* +\!\!+\ al_2^* = al_1^*\ al_2^*$$
[2] $$\text{Al}(\text{aliases } al^*) = al^*$$
[3] $$\text{Al}(\mathcal{G}_1\ \mathcal{G}_2) = \text{Al}(\mathcal{G}_1) +\!\!+\ \text{Al}(\mathcal{G}_2)$$
[4] $$\text{Al}(\mathcal{G}) = \quad \textbf{otherwise}$$
[5] $$\overline{\text{Al}}(\text{aliases } al^*) = \emptyset$$
[6] $$\overline{\text{Al}}(\mathcal{G}_1\ \mathcal{G}_2) = \overline{\text{Al}}(\mathcal{G}_1)\ \overline{\text{Al}}(\mathcal{G}_2)$$
[7] $$\overline{\text{Al}}(\mathcal{G}) = \mathcal{G} \quad \textbf{otherwise}$$

### 4.2.3 Normalization

Aliases are defined by renaming all alias symbols to their defined meaning. The function a⟦_⟧ produces a renaming from the alias declarations in the grammar and applies it to the non-alias parts of the grammar. The alias declarations are then attached to the renamed grammar. This is done in order to keep the following modular property:

$$\text{a}[\![\mathcal{G}_1\ \mathcal{G}_2]\!] = \text{a}[\![\text{a}[\![\mathcal{G}_1]\!]\ \text{a}[\![\mathcal{G}_2]\!]]\!]$$

This entails that aliases can be replaced before flattening a module, after which the aliases are still part of the grammar and keep their forward renaming property.

**module** Alias-Sdf-Normalization
**imports** Alias-Sdf-Projection$^{4.2.2}$ Kernel-Sdf-Normalization$^{2.3.3}$
Kernel-Sdf-Renaming$^{4.1.3}$
**exports**
**context-free syntax**
"a⟦" Grammar "⟧" → Grammar
rn(Aliases) → Renamings
symbols(Aliases) → SymbolSet
**equations**
Replace all alias symbols by their definition by applying a renaming derived from the alias declarations to the non-alias parts of the grammar.

[1] $\mathrm{a}[\![\mathcal{G}]\!] = \text{aliases } al^{+}\ (\overline{\mathrm{Al}}(\mathcal{G}))\ \mathrm{rn}(al^{+}) \quad \textbf{when } \ \mathrm{Al}(\mathcal{G}) = al^{+}$
[2] $\mathrm{a}[\![\mathcal{G}]\!] = \mathcal{G} \quad \textbf{otherwise}$

Build a renaming from a list of aliases. The target $\mathcal{B}$ of the alias declaration $\mathcal{A} \to \mathcal{B}$ is renamed to the source $\mathcal{A}$.

[3] $\mathrm{rn}() = []$
[4] $\mathrm{rn}(\mathcal{A} \to \mathcal{B}\ al^{*}) = [\mathcal{B} \Rightarrow \mathcal{A}] +\!\!+ \mathrm{rn}(al^{*})$

The symbols occurring in an alias declaration.

[5] $\mathrm{symbols}(\text{aliases } al^{*}) = \mathrm{symbols}(al^{*})$
[6] $\mathrm{symbols}(al^{*}) = \{\} \quad \textbf{when } \ al^{*} =$
[7] $\mathrm{symbols}(\mathcal{A} \to \mathcal{B}\ al^{*}) = \{\mathcal{A}\ \mathcal{B}\} \cup \mathrm{symbols}(al^{*})$

Merging and ordering of grammars.

[8] $\text{aliases} = \emptyset$
[9] $\text{aliases } al_1^{*} \diamond \text{aliases } al_2^{*} = \langle \text{aliases } al_1^{*}\ al_2^{*}, \emptyset \rangle$
[10] $\text{syntax } p^{*} \diamond \text{aliases } al^{*} = \langle \text{aliases } al^{*}, \text{syntax } p^{*} \rangle$

Aliases themselves can also be subject to renamings.

**module** Alias-Sdf-Renaming
**imports** Kernel-Sdf-Renaming$^{4.1.3}$ Alias-Sdf-Projection$^{4.2.2}$
**exports**
**context-free syntax**
"(" Aliases ")a" Renamings → Aliases
**equations**
Renaming of aliases.

[1] $(\text{aliases } al^{*})\ \rho = \text{aliases } (al^{*}\ )\mathrm{a}\ \rho$
[2] $(al^{*}\ )\mathrm{a}\ \rho = \quad \textbf{when } \ al^{*} =$
[3] $(\mathcal{A} \to \mathcal{B}\ )\mathrm{a}\ \rho = (\mathcal{A})\ \rho \to (\mathcal{B})\ \rho$
[4] $(al_1^{+}\ al_2^{+}\ )\mathrm{a}\ \rho = (al_1^{+}\ )\mathrm{a}\ \rho +\!\!+ (al_2^{+}\ )\mathrm{a}\ \rho$

## 4.3 Modules

In this section we introduce a module framework for grammars to support management and reuse of parts of the grammar of a language. A modular definition consists of a list of named modules. Modules can be reused in other modules by means of imports. The body of a module is a list of exported and hidden grammars. Export and hiding provide a means to control what is visible from a module and what is local to that module. Hidden syntax is useful when the syntax definition formalism is coupled to a semantics formalism for the specification of the semantics of languages. Hidden syntax then plays the role of auxiliary functions. Since imports are abbreviations for grammars, an import can be hidden or exported. Modules can be parameterized by a list of symbols. An import can instantiate these parameters, although this is not required. Parameterization is an abbreviation for a renaming. When a module $M[\alpha]$ is imported as $M[\beta]$, the formal parameters $[\alpha]$ are renamed into the actual parameters $[\beta]$. An import can also be subject to a renaming of symbols and productions.

**Example 3.1 (Aliases and Renaming)** The following module defines the syntax of tables. A table is defined as an alias for a set of mappings from keys to values. The value assigned to a key can be looked up in a table using the access function `lookup`.

```
module Tables
exports
  sorts Key Value Table
  aliases
    Set[(Key "|->" Value)] -> Table
  context-free syntax
    lookup(Table, Key) -> Value
```

Below we transform tables into mappings from variables to terms, thus obtaining a representation for substitutions. This is achieved by renaming the sorts in module `Tables` such that variables become the keys and terms the values in tables.

```
module Substitutions
imports Terms
        Tables[Key => Var   Value => Term   Table => Subst
               lookup(Table, Key) -> Value
                   => Subst "[" Var "]" -> Term ]
exports
  context-free syntax
    Subst "(" Term ")" -> Term
```

The additional function applies a substitution to all variables in a term. □

**Example 3.2 (Map)** Using renaming a kind of polymorphic higher-order functions can be expressed. The following module defines a function that maps a function over the elements of a list. The function is defined for a given `A` and `B` that can be instantiated as needed.

```
module Map[A B]
```

```
exports
  sorts A B
  context-free syntax
    (A => B) "*" "(" A* ")" -> B*
```

The disadvantage of this kind of polymorphism is that for each instance of a polymorphic function, an explicit module import has to be done. □

**Example 3.3 (Parameterized Modules)** The following module defines the syntax of a list of conditional equations preceded by the keyword 'equations'. This is the syntax of the equations part of an ASF+SDF module, which is parameterized by the syntax of some language. For each sort, productions defining the syntax of equations over that sort are defined. Note the use of the constrained iteration operator to define the bar (`Implies`) between conditions and conclusion as at least 3 equal signs.

```
module Equations
exports
  sorts Tag TagId CondEquation Equation Implies
        Condition Equations
  lexical syntax
    {[\=]}3+       -> Implies
    [a-z0-9A-Z\-]+ -> TagId
  aliases
    {Condition ","}+ -> Conditions
  context-free syntax
    "equations" CondEquation*          -> Equations
    Tag (Conditions Implies)? Equation -> CondEquation
    Tag Equation "when" Conditions     -> CondEquation
    "[" TagId? "]"                     -> Tag
```

Next we define generic syntax for sorts `Equation` and `Condition` as follows:

```
module X-Equations[X]
exports
  sorts X Equation Condition
  context-free syntax
    X "="  X -> Equation
    X "="  X -> Condition
    X "!=" X -> Condition
```

To define the syntax of the equations part of a module $M$, the ASF+SDF Meta-Environment generates a module $M$-`Equations` that defines the syntax of these equations (Klint, 1993). This module imports the language independent syntax of equations and defines equations for the sorts declared in the module. With the parameterized module `X-Equations` we can express this by a module that contains an import for each declared sort. For instance, for `Boolean-Equations` we get the following module:

```
module Booleans-Equations
imports Booleans
        Equations
        X-Equations[Bool]
```

Observe that the sorts `Condition` and `Equation` are declared in two different modules. This is not problematic when these modules meet, because duplicate definitions are merged. □

### 4.3.1 Syntax

A modular syntax definition consists of a series of named module declarations. A module declaration consists of a list of sections, which are either exports or hiddens. A module name consists of a module identifier and an optional list of parameters. Module identifiers can contain slashes to enable the use of directory names in module names, e.g., `sdf/kernel/Syntax`. A module can import any number of other modules. An import consists of a module name with optionally a renaming applied to it. An import of a module $M$ denotes the grammar declared in module $M$. An import can be contained in one of the exports or hiddens sections. In the latter case all syntax imported through that module is hidden and thus not exported from the module. Imports can also occur at the start of a module, outside any exports or hiddens section. In this case the imports are exported.

**module** Modular-Sdf-Syntax
**imports** Kernel-Sdf-Syntax[2.3.1] Renaming-Sdf-Syntax[4.1.1]
**exports**
  **sorts** ModuleId ModuleName Import Imports Section Sections Module
    Definition ImpSection
  **lexical syntax**

| | | |
|---|---|---|
| `[A-Za-z0-9_\-]+` | → | ModuleWord |
| "/"ModuleWord | → | ModuleDir |
| ModuleWord ModuleDir+ | → | ModuleId |
| ModuleDir+ | → | ModuleId |

  **context-free syntax**

| | | | |
|---|---|---|---|
| Module∗ | → | Definition | |
| "module" ModuleName ImpSection∗ Sections | → | Module | |
| "exports" Grammar | → | Section | |
| "hiddens" Grammar | → | Section | |
| Section∗ | → | Sections | |
| ModuleId | → | ModuleName | |
| ModuleId "[" Symbols "]" | → | ModuleName | |
| id(ModuleName) | → | Attribute | |
| "imports" Imports | → | ImpSection | |
| ImpSection | → | Grammar | |
| Import∗ | → | Imports | |
| ModuleName | → | Import | |
| ModuleName Renamings | → | Import | |
| "(" Import ")" | → | Import | {**bracket**} |

  **variables**

| | | |
|---|---|---|
| "*Mid*"[*0-9′*]∗ | → | ModuleId |
| "*M*"[*0-9′*]∗ | → | ModuleName |
| "*s*"[*0-9′*]∗ | → | Section |
| "*s*""∗"[*0-9′*]∗ | → | Section∗ |
| "*s*""+"[*0-9′*]∗ | → | Section+ |
| "*m*"[*0-9′*]∗ | → | Module |

| | |
|---|---|
| "$m*$" [$0$-$9'$]$*$ | $\rightarrow$ Module$*$ |
| "$m+$" [$0$-$9'$]$*$ | $\rightarrow$ Module+ |
| "$d$" [$0$-$9'$]$*$ | $\rightarrow$ Definition |
| "i" [$0$-$9'$]$*$ | $\rightarrow$ Import |
| "i" "$*$" [$0$-$9'$]$*$ | $\rightarrow$ Import$*$ |
| "i" "+" [$0$-$9'$]$*$ | $\rightarrow$ Import+ |
| "$is$" [$0$-$9'$]$*$ | $\rightarrow$ ImpSection |
| "$is*$" [$0$-$9'$]$*$ | $\rightarrow$ ImpSection$*$ |

### 4.3.2 Projection

Projection functions: $\pi_M(d)$ yields the body of the module named $M$. 'Exp' yields the exported part of a module and 'Hid' yields the hidden part of a module.

**module** Modular-Sdf-Projection
**imports** Modular-Sdf-Syntax[4.3.1] Booleans Kernel-Sdf-Projection[2.3.2]
Modular-Sdf-Renaming[4.3.4]
**exports**
**context-free syntax**

| | | |
|---|---|---|
| Import "$\in$" Imports | $\rightarrow$ Bool | |
| Sections "++" Sections | $\rightarrow$ Sections | {**assoc**} |
| Imports "++" Imports | $\rightarrow$ Imports | {**assoc**} |
| "$\pi$" "_" ModuleName "(" Definition ")" | $\rightarrow$ Sections | |
| "Exp" (Sections) | $\rightarrow$ Grammar | |
| "Hid" (Sections) | $\rightarrow$ Grammar | |

**equations**

Membership of a list of imports.

[1] $$\mathsf{i} \in \mathsf{i}_1^* \; \mathsf{i} \; \mathsf{i}_2^* = \top$$

[2] $$\mathsf{i} \in \mathsf{i}^* = \bot \quad \textbf{otherwise}$$

Concatenation of section and imports lists.

[3] $$s_1^* +\!\!+ s_2^* = s_1^* \; s_2^*$$

[4] $$\mathsf{i}_1^* +\!\!+ \mathsf{i}_2^* = \mathsf{i}_1^* \; \mathsf{i}_2^*$$

Lookup of a module by its name in a list of modules. If a module name matches the module name searched for, its list of sections is yielded. If a parameterized module is imported without specifying any actual parameters, the parameters are left uninstantiated. If a list of actual paramters is given, these are used to rename the formal parameters into the actual parameters. The function (_ $\Rightarrow$ _) constructs a renaming from the formal parameters to the actual parameters of a parameterized module. If no modules are found the empty list of sections is yielded.

[5] $$\pi_M(\text{module } M \;\; s^* \; m^*) = s^* +\!\!+ \pi_M(m^*)$$

[6] $$\pi_{Mid}(\text{module } Mid[\alpha] \;\; s^* \; m^*) = s^* +\!\!+ \pi_{Mid}(m^*)$$

[7] $$\pi_{Mid[\beta]}(\text{module } Mid[\alpha] \;\; s^* \; m^*) = (s^*) \; (\alpha \Rightarrow \beta) +\!\!+ \pi_{Mid[\beta]}(m^*)$$

[8] $$\pi_M(\text{module } M' \;\; s^* \; m^*) = \pi_M(m^*) \quad \textbf{otherwise}$$

[9] $$\pi_M() =$$

Exported grammars from a list of sections.

[10] $\text{Exp}() = \emptyset$
[11] $\text{Exp}(s_1^+ \ s_2^+) = \text{Exp}(s_1^+) \ \text{Exp}(s_2^+)$
[12] $\text{Exp}(\text{exports } \mathcal{G}) = \mathcal{G}$
[13] $\text{Exp}(\text{hiddens } \mathcal{G}) = \emptyset$

Hidden grammars from a list of sections.

[14] $\text{Hid}() = \emptyset$
[15] $\text{Hid}(s_1^+ \ s_2^+) = \text{Hid}(s_1^+) \ \text{Hid}(s_2^+)$
[16] $\text{Hid}(\text{exports } \mathcal{G}) = \emptyset$
[17] $\text{Hid}(\text{hiddens } \mathcal{G}) = \mathcal{G}$

### 4.3.3 Normalization

We define the semantics of the modular constructs introduced above by means of a normalization function that yields the flattening of a module in a modular syntax definition by replacing each import by the body of the module it refers to.

Hidden productions are renamed by attaching the name of the hiding module. Since all productions occurring in a hiddens do not occur in another hiddens section (they should have been exported) it can never occur that two such renamed productions are imported into the same module. A consequence of production merging in this case is that an exported function becomes hidden if it is also in the hiddens part of the module.

We define the function $\text{m}[\![d]\!](M)$ that yields the grammar corresponding to module $M$ in the definition $d$.

**module** Modular-Sdf-Normalization
**imports** Modular-Sdf-Projection$^{4.3.2}$ Modular-Sdf-Renaming$^{4.3.4}$
  Kernel-Sdf-Normalization$^{2.3.3}$ Grammar-Projection
**exports**
  **context-free syntax**
    "m⟦" Definition "⟧" "(" ModuleName ")" → Grammar
    hide(ModuleName, Grammar) → Grammar
    hide(ModuleName, Productions) → Productions
**hiddens**
  **sorts** IG
  **context-free syntax**
    "<" Imports "," Grammar ">" → IG
    imp "⟦" Definition "⟧" "(" Imports "," Import ")" → IG
    ims "⟦" Definition "⟧" "(" Imports "," Imports ")" → IG
    gra "⟦" Definition "⟧" "(" Imports "," Grammar ")" → IG
**equations**
Normalization of order of grammars.

[1] $\text{imports} = \emptyset$
[2] $\text{imports } \mathsf{i}_1^* \diamond \text{imports } \mathsf{i}_2^* = \langle \text{imports } \mathsf{i}_1^* \ \mathsf{i}_2^*, \emptyset \rangle$
[3] $\mathcal{G} \diamond \text{imports } \mathsf{i}^* = \langle \text{imports } \mathsf{i}^*, \mathcal{G} \rangle$ **otherwise**

Normalization of module sections. Exports and hiddens sections can be merged.

[4] $$\text{module } M\ is^*\ is\ s^* = \text{module } M\ is^* \text{ exports } is\ s^*$$
[5] $$s_1^* \text{ exports } \mathcal{G}_1 \text{ exports } \mathcal{G}_2\ s_2^* = s_1^* \text{ exports } \mathcal{G}_1\ \mathcal{G}_2\ s_2^*$$
[6] $$s_1^* \text{ hiddens } \mathcal{G}_1 \text{ hiddens } \mathcal{G}_2\ s_2^* = s_1^* \text{ hiddens } \mathcal{G}_1\ \mathcal{G}_2\ s_2^*$$
[7] $$s_1^* \text{ hiddens } \mathcal{G}_1 \text{ exports } \mathcal{G}_2\ s_2^* = s_1^* \text{ exports } \mathcal{G}_2 \text{ hiddens } \mathcal{G}_1\ s_2^*$$

The semantics of a module named $M$ in a definition $d$ is expressed by $\mathrm{m}[\![d]\!](M)$ and is the composition of the exported and hidden grammars of module $M$ with all imports replaced by the exported grammars of the modules they refer to.

[8] $$\frac{\pi_M(d) = s^*,\ \ \mathrm{gra}[\![d]\!](,\mathrm{Hid}(s^*)) = \langle \mathsf{i}_1^*, \mathcal{G}_1\rangle,\ \ \mathrm{gra}[\![d]\!](\mathsf{i}_1^*, \mathrm{Exp}(s^*)) = \langle \mathsf{i}_2^*, \mathcal{G}_2\rangle}{\mathrm{m}[\![d]\!](M)\ =\ \mathcal{G}_2\ \mathrm{hide}(M, \mathcal{G}_1)}$$

The function 'hide' marks all productions in the hiddens part of a module with the module name by attaching the attribute $\mathrm{id}(M)$ to it.

[9] $$\mathrm{hide}(M, \emptyset) = \emptyset$$
[10] $$\mathrm{hide}(M, \mathcal{G}_1\ \mathcal{G}_2) = \mathrm{hide}(M, \mathcal{G}_1)\ \mathrm{hide}(M, \mathcal{G}_2)$$
[11] $$\mathrm{hide}(M, \text{syntax } p^*) = \text{syntax } \mathrm{hide}(M, p^*)$$
[12] $$\mathrm{hide}(M, \mathcal{G}) = \mathcal{G} \quad \textbf{otherwise}$$
[13] $$\mathrm{hide}(M, \alpha \to \mathcal{A}\ \$) = \alpha \to \mathcal{A}\ \$ \mathbin{+\!\!+} \{\mathrm{id}(M)\}$$
[14] $$\mathrm{hide}(M, ) =$$
[15] $$\mathrm{hide}(M, p_1^+\ p_2^+) = \mathrm{hide}(M, p_1^+) \mathbin{+\!\!+} \mathrm{hide}(M, p_2^+)$$

The function 'gra' expands all the imports in a grammar. It returns a structure $\langle \mathsf{i}^*, \mathcal{G}\rangle$, which denotes a flattened grammar with the list of imports $\mathsf{i}^*$ that were expanded to flatten the grammar. This list is passed on to the rest of the flattening process in order to prevent multiple imports of the same module. This is important in particular in the presence of cyclic imports.

[16] $$\frac{\mathrm{gra}[\![d]\!](\mathsf{i}_1^*, \mathcal{G}_1) = \langle \mathsf{i}_2^*, \mathcal{G}_1'\rangle,\ \ \mathrm{gra}[\![d]\!](\mathsf{i}_2^*, \mathcal{G}_2) = \langle \mathsf{i}_3^*, \mathcal{G}_2'\rangle}{\mathrm{gra}[\![d]\!](\mathsf{i}_1^*, \mathcal{G}_1\ \mathcal{G}_2)\ =\ \langle \mathsf{i}_3^*, \mathcal{G}_1'\ \mathcal{G}_2'\rangle}$$

[17] $$\mathrm{gra}[\![d]\!](\mathsf{i}_1^*, \text{imports } \mathsf{i}_2^*)\ =\ \mathrm{ims}[\![d]\!](\mathsf{i}_1^*, \mathsf{i}_2^*)$$

[18] $$\mathrm{gra}[\![d]\!](\mathsf{i}^*, \mathcal{G})\ =\ \langle \mathsf{i}^*, \mathcal{G}\rangle \quad \textbf{otherwise}$$

The function 'ims' yields the flattened grammars for a list of imports.

[19] $$\mathrm{ims}[\![d]\!](\mathsf{i}^*, )\ =\ \langle \mathsf{i}^*, \emptyset\rangle$$

[20] $$\frac{\mathrm{imp}[\![d]\!](\mathsf{i}_1^*, \mathsf{i}) = \langle \mathsf{i}_3^*, \mathcal{G}_1\rangle,\ \ \mathrm{ims}[\![d]\!](\mathsf{i}_3^*, \mathsf{i}_2^*) = \langle \mathsf{i}_4^*, \mathcal{G}_2\rangle}{\mathrm{ims}[\![d]\!](\mathsf{i}_1^*, \mathsf{i}\ \mathsf{i}_2^*)\ =\ \langle \mathsf{i}_4^*, \mathcal{G}_1\ \mathcal{G}_2\rangle}$$

The function 'imp' yields the flattened grammar associated with the exported grammar of an import. The first list of imports denotes the imports that are already expanded. If a module was already imported it is not imported again. This is a protection against cyclic imports.

[21] $$\frac{\mathcal{G} = \text{if } M \in \mathsf{i}^* \text{ then } \emptyset \text{ else } \mathrm{Exp}(\pi_M(d)) \text{ fi}}{\mathrm{imp}[\![d]\!](\mathsf{i}^*, M)\ =\ \mathrm{gra}[\![d]\!](\mathsf{i}^*\ M, \mathcal{G})}$$

[22] $$\frac{\mathcal{G} = \text{if } M\ \rho \in \mathsf{i}^* \text{ then } \emptyset \text{ else } (\mathrm{Exp}(\pi_M(d)))\ \rho \text{ fi}}{\mathrm{imp}[\![d]\!](\mathsf{i}^*, M\ \rho)\ =\ \mathrm{gra}[\![d]\!](\mathsf{i}^*\ M\ \rho, \mathcal{G})}$$

As we will see in the next section, the renaming $\rho$ that is applied to the exported part of the imported module $M$ in the last equation above is also applied to the imports of that module and hence is applied recursively to all modules imported via $M$.

### 4.3.4 Renaming

We extend the definition of renaming to renaming of module sections and imports. This includes the renaming of imports, and hence the renaming of renamings applied to imported modules.

**module** Modular-Sdf-Renaming
**imports** Renaming-Sdf-Renaming$^{4.1.3}$ Modular-Sdf-Syntax$^{4.3.1}$
Modular-Sdf-Projection$^{4.3.2}$
**exports**
**context-free syntax**
"(" Sections ")" Renamings → Sections
"(" Imports ")" Renamings → Imports
**equations**
Renaming sections.

[1] $$(s^*)\ \rho = \quad \textbf{when}\ \ s^* =$$
[2] $$(\text{exports } \mathcal{G})\ \rho = \text{exports } (\mathcal{G})\ \rho$$
[3] $$(\text{hiddens } \mathcal{G})\ \rho = \text{hiddens } (\mathcal{G})\ \rho$$
[4] $$(s_1^+\ s_2^+)\ \rho = (s_1^+)\ \rho +\!\!+ (s_2^+)\ \rho$$

Renaming a list of imports implies applying the renaming to all imported modules, i.e., attaching the renaming to each module name in the list of imports.

[5] $$(\text{imports } \mathsf{i}^*)\ \rho = \text{imports } (\mathsf{i}^*)\ \rho$$
[6] $$(M)\ \rho = M\ \rho$$
[7] $$(\mathsf{i}^*)\ \rho = \quad \textbf{when}\ \ \mathsf{i}^* =$$
[8] $$(\mathsf{i}_1^+\ \mathsf{i}_2^+)\ \rho = \mathsf{i}_3^+\ \mathsf{i}_4^+ \quad \textbf{when}\ \ \mathsf{i}_3^+ = (\mathsf{i}_1^+)\ \rho,\ \ \mathsf{i}_4^+ = (\mathsf{i}_2^+)\ \rho$$

If the imported module has already a renaming attached to it, the new renaming is applied to the first, yielding the composition of the two renamings.

[9] $$(M\ \rho_1)\ \rho_2 = M\ (\rho_1)\ \rho_2$$

### 4.3.5 Discussion

The modularization presented here is an extension with symbol parameters, import renamings and hidden imports of the modularization of ASF+SDF as implemented in the ASF+SDF Meta-Environment (Klint, 1993). The definition here is a pure 'textual' inclusion semantics of modularization. Hendriks (1991) describes both a textual normalization semantics and an incremental semantics for modular constructs without renamings and hidden imports. The incremental implementation of modularization in the Meta-Environment becomes complicated in the presence of renamings, since items are created on the fly and can no longer be associated with a module. We have not addressed the issue of incremental parser generation and modular parser generation in a setting with renamings.

We deviate from the original design of ASF in that we do not incorporate the 'origin rule' that forbids identification of names that originate from different modules (Bergstra *et al.*, 1989b). This style forbids to have two modules with partly overlapping signatures, e.g., both introducing the same sort or function, that are imported in the same module, even if the overlap is intentional. The definition here is completeley liberal in this respect. Productions that are imported via different routes are identified if they are the same.

In §4.1 we saw that renamings are not guaranteed to preserve well-formedness of trees. A further study of modular properties of grammars in the line of module algebra (Bergstra *et al.*, 1990) should give more insight into properties of good modularization. Some topics for study are: properties of trees and languages under renaming, ambiguity caused by union, interaction of regular expressions and renamings, modular properties of reject productions.

# 5

# The Syntax Definition Formalism SDF2

This chapter presents the assembly of the syntax definition formalism SDF2 from the features designed in previous chapters. This is mainly a matter of defining collecting modules that import the modules defined earlier. However, some features interfere. In some cases the normalization functions have to be extended to cover constructs introduced for other features. In other cases features have to be extended such that the orthogonality of another feature is maintained. The chapter concludes with a comparison of SDF2 to SDF and a discussion of anomalies and possible improvements to the formalism.

## 5.1 SDF2

Now we put the pieces together and define the syntax definition formalism SDF2, which is a generalization of SDF (Heering *et al.*, 1989). It covers all features available in SDF and adds several new ones. Furthermore, up to some small adaptations, SDF is textually (although not structurally) a subset of SDF2. This means that existing SDF definitions can be used almost literally as SDF2 definitions. The differences can be translated automatically by means of a migration tool.

The combination of features described earlier is achieved basically by combining them by means of imports into collecting modules. For each aspect of the definition, such as syntax, projection and normalization, a collecting module is defined. Here we show the collecting modules for the syntax and normalization of SDF2. The other modules can be found in Appendix A.4. Although we have tried to define features orthogonally, some interference between them is unavoidable. For instance, when we extend the syntax of symbols, normalization functions that deal with symbols are affected and have to be extended accordingly.

### 5.1.1 Syntax

The syntax of SDF2 is simply the collection of the syntax of all features introduced sofar. The syntax is extended with *lexical* and *context-free* priorities and restrictions, which arise as a result of the combination of Basic-Sdf with Priority-Sdf and Restriction-Sdf. The constructor 'definition' collects a list of modules into a single SDF definition.

The symbols ⟨`Start`⟩ and ⟨`START`⟩ serve to define grammars with a single start symbol. In the normalization below productions will be added such that ⟨START⟩ is the union of all sorts of the grammar. The symbol ⟨Start⟩ is used to describe files that consist of a string over the language of ⟨START⟩ followed by the end of file character.

In the Label extension a symbol can be labeled with a literal using the syntax $L : \mathcal{A}$. This extension is not further defined here. The priorities section is extended to deal with this extra symbol constructor.

**module** Sdf2-Syntax
**imports** Kernel-Sdf-Syntax$^{2.3.1}$ Basic-Sdf-Syntax$^{3.3.1}$ Modular-Sdf-Syntax$^{4.3.1}$
Regular-Sdf-Syntax$^{3.2.1}$ Priority-Sdf-Syntax$^{3.1.1}$ CC-Sdf-Syntax$^{2.4.2}$
Sorts-Sdf-Syntax$^{2.4.1}$ Literals-Sdf-Syntax$^{2.4.3}$ Label-Sdf-Syntax
Restrictions-Sdf-Syntax$^{3.4.1}$ Alias-Sdf-Syntax$^{4.2.1}$
**exports**
**sorts** SDF
**context-free syntax**

| | |
|---|---|
| "⟨START⟩" | → Symbol |
| "⟨Start⟩" | → Symbol |
| "lexical" "priorities" Priorities | → Grammar |
| "context-free" "priorities" Priorities | → Grammar |
| "lexical" "restrictions" Restrictions | → Grammar |
| "context-free" "restrictions" Restrictions | → Grammar |
| "definition" Definition | → SDF |

**priorities**
Symbol "|"Symbol → Symbol > Literal ":"Symbol → Symbol

### 5.1.2 Normalization

We define the normalization function that normalizes a syntax definition by applying the normalization functions of the individual features. Here we have to deal with interaction between the normalization functions for the separate features and the constructs added to the formalism in other features.

**module** Sdf2-Normalization
**imports** Sdf2-Syntax$^{5.1.1}$ Sdf2-Projection$^{A.4}$ Sdf2-Renaming$^{A.4}$
Basic-Sdf-Normalization$^{3.3.2}$ Modular-Sdf-Normalization$^{4.3.3}$
Priority-Sdf-Normalization$^{3.1.3}$ Regular-Sdf-Normalization$^{3.2.2}$
Literals-Sdf-Normalization$^{2.4.3}$ CC-Sdf-Normalization$^{2.4.2}$
Sorts-Sdf-Normalization$^{2.4.1}$ Sorts-Sdf-Projection$^{2.4.1}$
Restrictions-Sdf-Normalization$^{3.4.3}$ Alias-Sdf-Normalization$^{4.2.3}$
**exports**
**context-free syntax**

| | |
|---|---|
| normalize "⟦" SDF "⟧" "(" ModuleName "," Symbol ")" | → Grammar |
| topsorts "⟦" Grammar "⟧" "(" Symbol ")" | → Grammar |
| topsorts(Symbol, Symbols) | → Productions |

**equations**
The normalization of an SDF2 definition is defined by the following equation. The function 'normalize' is parameterized with a module name denoting the top module to be normalized and a sort denoting the topsort of the definition. The definition is normalized by first expanding module $M$ by means of function 'm'. Then the normalization functions 'b' (Basic), 'a' (Alias), 'r' (Regular), 'p'

(Priorities), 'l' (Literals) and 'k' (Kernel) are applied to the resulting grammar. The function 'topsorts', defined below, is used to add special productions for the top sorts of the definition and to remove productions not reachable from those top sorts.

[1] $$\frac{\mathrm{k}[\![\mathrm{l}[\![\mathrm{p}[\![\mathrm{r}[\![\mathrm{topsorts}[\![\mathrm{a}[\![\mathrm{b}[\![\mathrm{m}[\![d]\!](M)]\!]]\!]]\!](\mathcal{A})]\!]]\!]]\!]]\!] = \mathcal{G}}{\mathrm{normalize}[\![\mathrm{definition}\ d]\!](M, \mathcal{A}) \ = \ \mathrm{reachable}(\{\langle\mathtt{Start}\rangle\}, \mathcal{G})}$$

The function 'topsorts' adds a special production for the symbol $\langle\mathtt{Start}\rangle$, which declares that a text over the grammar is a string of sort $\mathcal{A}$ followed by the character representing the end of file. For each declared sort in the definition a production is added that defines that a text can be a string of that sort which starts and ends with layout.

[2] $$\frac{\mathcal{G}' = \mathrm{syntax}\ \mathcal{A}\ [\backslash\mathtt{EOF}] \rightarrow \langle\mathtt{Start}\rangle \ +\!\!+\ \mathrm{topsorts}(\mathcal{A}, \mathrm{S}(\mathcal{G}))}{\mathrm{topsorts}[\![\mathcal{G}]\!](\mathcal{A}) \ = \ \mathcal{G}'\ \mathcal{G}}$$

[3] $\mathrm{topsorts}(\mathcal{A}, \alpha^+\ \beta^+) = \mathrm{topsorts}(\mathcal{A}, \alpha^+) +\!\!+ \mathrm{topsorts}(\mathcal{A}, \beta^+)$

[4] $\mathrm{topsorts}(\mathcal{A}, ) =$

[5] $\mathrm{topsorts}(\mathcal{A}, \mathcal{B}) = \langle\mathtt{LAYOUT?\text{-}CF}\rangle\ \langle\mathcal{B}\mathtt{\text{-}CF}\rangle\ \langle\mathtt{LAYOUT?\text{-}CF}\rangle \rightarrow \mathcal{A}$

### 5.1.3 Interaction

Several of the normalization functions are underdefined, i.e., the full SDF2 formalism contains more constructors than the extension for which they have been defined. Therefore, we must extend these functions accordingly.

**exports**
  **context-free syntax**
    "<" Priorities "-LEXp" ">" → Priorities
    "<" Priorities "-CFp" ">" → Priorities
    "<" Restrictions "-LEX" ">" → Restrictions
    "<" Restrictions "-CF" ">" → Restrictions
**hiddens**
  **variables**
    "*L*" → Literal
**equations**

The normalization function for regular expressions must be extended to the symbol constructors added in other extensions. The first equations express that sorts, character classes, literals and the symbols `LAYOUT`, $\langle\mathtt{START}\rangle$ and $\langle\mathtt{Start}\rangle$ do not generate any productions.

| | | | |
|---|---|---|---|
| [6] | $\mathrm{r}[\![\mathcal{S}]\!] = \emptyset$ | [7] | $\mathrm{r}[\![\mathtt{LAYOUT}]\!] = \emptyset$ |
| [8] | $\mathrm{r}[\![cc]\!] = \emptyset$ | [9] | $\mathrm{r}[\![\langle\mathtt{START}\rangle]\!] = \emptyset$ |
| [10] | $\mathrm{r}[\![L]\!] = \emptyset$ | [11] | $\mathrm{r}[\![\langle\mathtt{Start}\rangle]\!] = \emptyset$ |

The following equations define that the productions generated for some symbol $\mathcal{A}$ should be transformed into productions for lexical (context-free) productions if a lexical (context-free) version of the symbol occurs. This entails that first the productions for $\mathcal{A}$ are generated by the recursive call and that these are transformed by the $\langle\_\mathtt{\text{-}LEX}\rangle$ ($\langle\_\mathtt{\text{-}CF}\rangle$) function.

[12] $$\mathrm{r}[\![\langle\mathcal{A}\mathtt{\text{-}LEX}\rangle]\!] = \langle\mathrm{r}[\![\mathcal{A}]\!]\mathtt{\text{-}LEX}\rangle$$

[13] $\mathrm{r}[\![\langle \mathcal{A}\texttt{-CF}\rangle]\!] = \langle \mathrm{r}[\![\mathcal{A}]\!]\texttt{-CF}\rangle$
[14] $\mathrm{r}[\![\langle \mathcal{A}\texttt{-VAR}\rangle]\!] = \mathrm{r}[\![\mathcal{A}]\!]$

This is an example of the context-sensitivity of the generation of productions from symbols. The meaning of $\langle \mathrm{Id} * \texttt{-CF}\rangle$ is different from that of $\langle \mathrm{Id} * \texttt{-LEX}\rangle$.

*Basic* Literals and character classes do not need the $\langle \_\texttt{-LEX}\rangle$ or $\langle \_\texttt{-CF}\rangle$ constructor, because they are lexical by definition

[15] $\langle cc\texttt{-LEX}\rangle = cc$ [16] $\langle cc\texttt{-CF}\rangle = cc$ [17] $\langle cc\texttt{-VAR}\rangle = cc$
[18] $\langle L\texttt{-LEX}\rangle = L$ [19] $\langle L\texttt{-CF}\rangle = L$ [20] $\langle L\texttt{-VAR}\rangle = L$

*Basic + Priorities* Equations for the normalization of lexical and context-free priorities that were added at the level of SDF2.

[21] $$\frac{\mathcal{G}_1 = \text{context-free priorities } pr^*,\ \ \mathcal{G}_2 = \text{context-free syntax } p^*}{\mathcal{G}_1 \diamond \mathcal{G}_2 \ = \ \langle \text{context-free syntax } p^*, \text{context-free priorities } pr^*\rangle}$$

[22] $$\frac{\mathcal{G}_1 = \text{context-free priorities } pr^*,\ \ \mathcal{G}_2 = \text{lexical syntax } p^*}{\mathcal{G}_1 \diamond \mathcal{G}_2 \ = \ \langle \text{lexical syntax } p^*, \text{context-free priorities } pr^*\rangle}$$

[23] $$\frac{\mathcal{G}_1 = \text{context-free priorities } pr_1^*,\ \ \mathcal{G}_2 = \text{context-free priorities } pr_2^*}{\mathcal{G}_1 \diamond \mathcal{G}_2 \ = \ \langle \text{context-free priorities } pr_1^*, pr_2^*, \emptyset\rangle}$$

Context-free priorities are priority declarations for context-free productions and are abbreviations of normal priorities in the same way that context-free syntax is an abbreviation for a certain style of normal syntax. The productions in the priorities sections are thus treated with the same $\langle \_\texttt{-CF}\rangle$ functions as context-free productions.

[24] $\mathrm{b_{aux}}[\![\text{context-free priorities } pr^*]\!] = \langle \text{priorities } pr^*\texttt{-CF}\rangle$
[25] $\langle \text{priorities } pr^*\texttt{-CF}\rangle = \text{priorities } \langle \mathrm{norm}[\![pr^*]\!]\texttt{-CFp}\rangle$
[26] $\langle pr^*\texttt{-CFp}\rangle = \quad \textbf{when} \ \ pr^* =$
[27] $\langle pr_1^+, pr_2^+\texttt{-CFp}\rangle = \langle pr_1^+\texttt{-CFp}\rangle +\!\!+ \langle pr_2^+\texttt{-CFp}\rangle$
[28] $\langle p_1 > p_2\texttt{-CFp}\rangle = \langle p_1\texttt{-CF}\rangle > \langle p_2\texttt{-CF}\rangle$
[29] $\langle p_1 \ as \ p_2\texttt{-CFp}\rangle = \langle p_1\texttt{-CF}\rangle \ as \ \langle p_2\texttt{-CF}\rangle$

Similarly for lexical priorities.

[30] $\mathrm{b_{aux}}[\![\text{lexical priorities } pr^*]\!] = \langle \text{priorities } pr^*\texttt{-LEX}\rangle$
[31] $\langle \text{priorities } pr^*\texttt{-LEX}\rangle = \text{priorities } \langle \mathrm{norm}[\![pr^*]\!]\texttt{-LEXp}\rangle$
[32] $\langle pr^*\texttt{-LEXp}\rangle = \quad \textbf{when} \ \ pr^* =$
[33] $\langle pr_1^+, pr_2^+\texttt{-LEXp}\rangle = \langle pr_1^+\texttt{-LEXp}\rangle +\!\!+ \langle pr_2^+\texttt{-LEXp}\rangle$
[34] $\langle p_1 > p_2\texttt{-LEXp}\rangle = \langle p_1\texttt{-LEX}\rangle > \langle p_2\texttt{-LEX}\rangle$
[35] $\langle p_1 \ as \ p_2\texttt{-LEXp}\rangle = \langle p_1\texttt{-LEX}\rangle \ as \ \langle p_2\texttt{-LEX}\rangle$

*Basic + Restrictions*

[36] $\mathrm{b_{aux}}[\![\text{lexical restrictions } restr^*]\!] = \text{restrictions } \langle restr^*\texttt{-LEX}\rangle$

[37] $\langle restr^*\texttt{-LEX}\rangle = \quad \textbf{when} \ \ restr^* =$
[38] $\langle restr_1^+ \ restr_2^+\texttt{-LEX}\rangle = \langle restr_1^+\texttt{-LEX}\rangle +\!\!+ \langle restr_2^+\texttt{-LEX}\rangle$

[39] $$\langle \alpha \not+ cc\text{-}\mathtt{LEX}\rangle = \langle \alpha\text{-}\mathtt{LEXs}\rangle \not+ cc$$

[40] $$\mathrm{b_{aux}}[\![\text{context-free restrictions } restr^*]\!] = \text{restrictions } \langle restr^*\text{-}\mathtt{CF}\rangle$$

[41] $$\langle restr^*\text{-}\mathtt{CF}\rangle = \quad \textbf{when } restr^* =$$
[42] $$\langle restr_1^+ \; restr_2^+\text{-}\mathtt{CF}\rangle = \langle restr_1^+\text{-}\mathtt{CF}\rangle +\!\!+ \langle restr_2^+\text{-}\mathtt{CF}\rangle$$
[43] $$\langle \not+ cc\text{-}\mathtt{CF}\rangle = \not+ cc$$
[44] $$\langle \mathcal{A}\ \alpha \not+ cc\text{-}\mathtt{CF}\rangle = \langle \mathcal{A}\text{-}\mathtt{CF}\rangle\ \beta \not+ cc \quad \textbf{when } \langle \alpha \not+ cc\text{-}\mathtt{CF}\rangle = \beta \not+ cc$$

*Labels*

[45] $$\mathrm{r}[\![L : \mathcal{A}]\!] = \mathrm{r}[\![\mathcal{A}]\!]$$
[46] $$\langle L : \mathcal{A}\text{-}\mathtt{LEX}\rangle = L : \langle \mathcal{A}\text{-}\mathtt{LEX}\rangle$$
[47] $$\langle L : \mathcal{A}\text{-}\mathtt{CF}\rangle = L : \langle \mathcal{A}\text{-}\mathtt{CF}\rangle$$
[48] $$\langle L : \mathcal{A}\text{-}\mathtt{VAR}\rangle = L : \langle \mathcal{A}\text{-}\mathtt{VAR}\rangle$$

*Hiding Productions*

[49] $$\mathrm{hide}(M, \text{context-free syntax } p^*) = \text{context-free syntax } \mathrm{hide}(M, p^*)$$
[50] $$\mathrm{hide}(M, \text{lexical syntax } p^*) = \text{lexical syntax } \mathrm{hide}(M, p^*)$$
[51] $$\mathrm{hide}(M, \text{variables } p^*) = \text{variables } \mathrm{hide}(M, p^*)$$
[52] $$\mathrm{hide}(M, \text{lexical variables } p^*) = \text{lexical variables } \mathrm{hide}(M, p^*)$$

*Aliases*

[53] $$\frac{\mathcal{G}_1 = \text{aliases } al^*,\ \ \mathcal{G}_2 = \text{sorts } \alpha}{\mathcal{G}_1 \diamond \mathcal{G}_2 \ = \ \langle \text{sorts } \alpha, \text{aliases } al^*\rangle}$$

## 5.2 Comparison to SDF

SDF2 was developed as a generalization of SDF (Heering *et al.*, 1989). We briefly list the differences between the two formalisms.

### 5.2.1 Semantics

SDF defines the semantics of a syntax definition by means of mappings to other formalisms. The lexical syntax is mapped to a regular grammar. The context-free syntax is mapped to a context-free grammar. From the entire definition a first-order many-sorted algebraic signature is derived. A parse tree for a string according to the grammar is translated to a term or abstract syntax tree over the signature. In SDF2 parse trees are defined by means of a well-formedness predicate on ATerms based directly on (the normal form of) a syntax definition. The strings of the language defined by a grammar are obtained via the function yield. No external formalism is used to define trees. In this way the notions of grammar and signature that were related via mappings in SDF are completely integrated in SDF2.

### 5.2.2 Lexical and Context-free Syntax

SDF integrates lexical syntax and context-free syntax in one formalism. However, this integration is only at the level of the formalism; on the level of the implementation these are separated. The lexical syntax is mapped to a regular grammar (hence the specification of the lexical syntax should also be regular). The context-free syntax is translated to a context-free grammar. In SDF2 the integration of lexical and context-free syntax is completed. All other features are orthogonal with respect to lexical and context-free syntax. For instance, character classes and regular expressions can be used in exactly the same way in lexical productions and context-free productions.

### 5.2.3 Lexical Disambiguation

SDF has several built-in lexical disambiguation rules that are applied to the token stream before tokens are passed to the parser. SDF2 has no built-in lexical disambiguation rules, but provides reject productions to express the prefer literals rule and follow restrictions to express longest match disambiguation.

### 5.2.4 Character Classes

In SDF the syntax of character classes is defined lexically. In SDF2 character classes are defined by means of context-free constructors. This makes the definition of normalization of character classes much easier. The differences with character classes in SDF are: numeric characters have a decimal interpretation instead of an octal interpretation, there is no syntactic limit to the range of numeric characters, all characters except letters and digits have to be escaped using a slash. In SDF2 character classes have a numeric interpretation, that is, each character class is normalized to an ordered and non-overlapping list of numeric characters and ranges of characters.

### 5.2.5 Lists

SDF only provides list sorts that can be used in the left-hand sides of productions. Furthermore, lists are not orthogonally defined. In the lexical syntax no iteration with separator is provided. In SDF only sorts can be used on the right-hand side of a production. This means that list sorts cannot be the result of functions. In order to define a function with a list as result, a new sort has to be introduced into which the list sort is injected. Furthermore, to concatenate the lists that result from a function a concatenation function should be defined.

SDF2 provides an expressive set of regular expressions that are treated as first-class citizens. Regular expressions can be used where ever any another symbol can be used. In general, all symbols that can be used in the left-hand side of a production can also be used as output symbols.

### 5.2.6 Priorities

The priorities declarations of SDF2 are the same as in SDF with the following exceptions: No abbreviations of productions in priorities are supported because of the problematic semantics in a setting with modules. No <-chains are provided. The implementation does not provide the multi-set filter interpretation of priorities.

### 5.2.7 Reuse

SDF does not provide renamings, module parameterization, hidden imports, and aliases.

## 5.3 Discussion and Concluding Remarks

We have presented the modular design of a family of syntax definition formalisms. The result is a uniform formalism for syntax definition designed for extensibility. A guiding principle in the design is the orthogonality of the features with respect to one another. As as consequence it is easy to replace a feature by a variant or to add a new feature without affecting the design and implementation of all other features.

### 5.3.1 Parser Generation

The direct motivation for this work was the specification of a parser generator for SDF. Many of the techniques presented in this paper were originally developed for the translation of SDF to intermediate languages like context-free and regular grammars as prescribed by the SDF reference manual (Heering *et al.*, 1989). Gradually it became clear that the difficulty of this project was due to the monolithic design of SDF. The features presented in this paper are combined in the formalism SDF2 that is intended to replace SDF. The specification of a parser generator for SDF2 was easier due to the uniform abstract syntax and elimination of cases by normalization. The tables generated by the generator are interpreted by the generic scannerless generalized-LR parser described in Visser (1997e).

### 5.3.2 Disambiguation

Priorities are interpreted as a well-formedness requirement on parse forests, which could be operationalized as a filter on parse forests as prescribed by Heering *et al.* (1989). This approach can be extended to other disambiguation methods as described in Klint and Visser (1994).

We have provided some features for disambiguation of ambiguous context-free grammars. There remain a large number of ambiguities that can not be solved with these mechanisms. Some more advanced disambiguation methods are described in Klint and Visser (1994). Here we list some ideas for improvements of the current scheme.

The priority relation $>$ on productions does not allow a distinction between the arguments of the productions to which it applies. In several cases it would be useful to restrict the relation to certain arguments. For instance, the priority declaration

```
T T -> T  >  "let" V "=" T "in" T -> T
```

does correctly forbid the usage of a `let` expression as the first argument of an application. However it also forbids the usage of `let` as the last argument of an application, for which there is no reason. An extension of the notation could be

```
T T -> T  {1}>  "let" V "=" T "in" T -> T
```

to declare the desired disambiguation. There would be no implementation problems with such an extension.

A case for non-standard disambiguation is in ambiguous equations. In §4.3 we gave as an example the specification of the syntax of conditional equations. It can occur that equations are ambiguous due to injections. If a symbol $\mathcal{A}$ is injected in $\mathcal{B}$, then an equation over two $\mathcal{A}$ expressions can be interpreted both as $\mathcal{A}$ and $\mathcal{B}$ equations. A possible interpretation of such an ambiguity is to take both possibilities. This is done in the definition of multi-level algebraic specifications in Visser (1996), where ambiguous equations can occur due to overloading of functions.

In the implementation of SDF in the ASF+SDF Meta-Environment an undocumented disambiguation method is used. As a simplification of the multi-set ordering, trees with fewer injections are preferred over trees with more injections. Such a method is needed to disambiguate conditions of equations. This method has not been implemented as part of the SDF2 tools, but can be added as a post-parse filter without problems.

We have defined follow restrictions and reject productions to express lexical disambiguation rules. We omitted the definition of these methods as an extension of the well-formedness predicate on parse trees. See Visser (1997e) for a discussion of the semantics of these methods and for a discussion of automatic lexical disambiguation.

### 5.3.3 Renaming

Modules associate a name with a grammar. Grammars can be combined by module imports. Export and hiding provide control over visibility of grammars. New with respect to the modularization of SDF are renamings and hidden imports. In the current definition of renaming productions, only the literal skeleton of the production can be changed, but the order of the arguments stays the same. Sometimes it is desirable to change the syntax of a production and also make a permutation of the arguments. A notation for such permutations should be devised by means of some kind of indexing. The problem with such a notation is that the current definition reuses the syntax of productions literally in the definition of renamings. Changing the syntax of productions will thus be applicable everywhere. The label facility (see below) could be used for this purpose. Unfortunately, renamings are not guaranteed to preserve well-formedness of parse trees. Further study is needed to find a set of sufficient requirements on renamings that do guarantee well-formedness.

### 5.3.4 Labels

A feature that has not discussed are labels. Labels are intended to be used as ‘field names’ of a record. For instance, consider the following production defining the syntax of assignments in an imperative language:

```
var : Var ":=" value : Exp -> Stat {cons(assign)}
```

The two arguments are labeled with `var` and `value`, respectively. From this information we can derive the following syntax for projection functions based on the field names:

```
Stat "." var   -> Var
Stat "." value -> Exp
```

This should be accompanied by the the defining equations for these functions.

### 5.3.5 Derived Syntax

Regular expressions are considered as name constructors that are used to make new names out of existing ones. A normalization function adds canonical productions defining the regular operators. For instance, $\mathcal{A}?$ denotes an optional $\mathcal{A}$ and is defined by the productions $\mathcal{A} \rightarrow \mathcal{A}?$ and $\rightarrow \mathcal{A}?$. However, there is no restriction on the use of these name constructors. Other defining productions can be added by the user. In the context of algebraic specification this means for instance that users can specify functions that have lists ($\mathcal{A}*$) as result.

Regular expressions are an example of derived syntax: Given some symbol or even production in the grammar, other productions are derived. Many other applications of derived syntax could be useful.

Sometimes it is useful to explicitly indicate empty constructs and injections. This could be accommodated by generating syntax for explicitly matching injection functions and $\epsilon$-functions, i.e., if `A -> B` is a production then also `"injA-B" (A) -> B` Similarly for $\epsilon$, if `-> A` a production, then also `"emptyA" -> A`. These constructors should of course match with their origins. This can be done by translating these functions internally to the real injection or $\epsilon$ function. See Dinesh (1995) for some interesting remarks on injections in ASF+SDF.

A structure editor provides facilities to manipulate sentential forms. This requires the specification of the syntax of symbol placeholders. For each symbol `A` that is not a literal add a production `"<A>" -> A`.

Another case of this kind is the generation of explicit type casts `S ":" "S" -> S {cast}` (like bracket attribute) to constrain the type of an overloaded entity. This would be similar to the `no-operator` attribute in SDF.

It would be even better to make syntax derivation user-definable by providing schemas such as discussed above.

### 5.3.6 Polymorphic Syntax

The definition of regular expressions by introducing new productions is an instance of second order quantification. The generalization of this approach to two-level grammars in Visser (1997d) provides the syntactic counterpart of the two-level specifications in Meinke (1992) and the multi-level specifications in Visser (1996). Generic productions are written as production schemata. The syntax of symbol constructors is described by means of a second level grammar.

### 5.3.7 Dynamic Syntax

Another open problem is the formal description of languages with an extensible syntax. Programs in such languages can contain grammars that define part of the syntax of the program itself. An example of extensible syntax is the syntax of equations in ASF+SDF. Several other instances exist, e.g., Cardelli *et al.* (1994), and Vittek (1994) (Elan). All these approaches treat the meta-language and object language differently. A formal approach to this problem would specify the syntax of the base language and the grammars it can specify and the lifting of these grammars to meta-level grammars.

### 5.3.8 Design Methodology

We have presented a large specification. We approached this using a rigorous modularization of the specification in a matrix of modules. For each feature the syntax and tools are described in separate modules. In this way it becomes feasible to flexibly include and exclude parts of a language definition. Some parts of the specification such as the ATerm encoding are not very interesting. It would be better if those parts could be generated using a simple rule.

The main technique we have applied is that of definition of features by normalization, i.e., transformation to a subset of the language. The great advantage of normalization is that many features can be provided to enhance the expressiveness of the language while defining the semantics of the formalism on a small set of kernel features in which the other features are expressed. Normalization has also its disadvantages. The semantics of various features is defined indirectly, which makes reasoning about them more troublesome. Furthermore, parse trees over a grammar use the normalized productions, which can look rather different than their origins. It would be desirable to use normalization equations rather than functions in order to be able to reason about equivalence of syntax definitions. The problem with such an approach is the lack of control over normalization. A solution could be the use of strategies such as described in Luttik and Visser (1997).

The modularization of the formalism and hence the modularization of the normalization in separate normalization functions for each feature made the specification of normalization feasible. A normalization function that would in one pass over the grammar normalize it would be a very complex. However, the modularization also hides the interaction between features. When defining a normalization function for an extension of the kernel, only those constructs introduced are normalized. The combination of features prompts the extension of the normalization function to new constructs. Often this is can be achieved by innocent distribution equations, but in some cases the interaction between features is more problematic. In particular, the interaction of renamings with other features needs more study.

Language design is a software engineering process. A language definition gets better developed if it is actually used in a prototype implementation. The parts of the specification of SDF2 that are used in the parser generator, i.e., the normalization, were developed on demand. Especially the fragment of SDF2 that corresponds to SDF was developed first, because most syntax definitions fed to the parser generator were converted SDF definitions. Other parts of the specification, such as well-formedness or equality that are not directly used in tools were developed later. But these parts are important because they define the correctness criteria for implementations. The well-formedness checker can be used to validate the output of a parser for a grammar. The equality checker can be used to validate a matching algorithm for terms.

The design approach we have used for SDF2 has led to an infrastructure for further study of syntax definition and experimentation with new features. It is indeed very easy to extend the specification in order to construct subsets or supersets of the formalism, or to replace a feature by a variant.

# A

# Auxiliary Modules for the Specification of SDF2

In this appendix we include several auxiliary modules used in the specification of SDF2.

## A.1 Literals

**module** Literals
**imports** Layout
**exports**
  **sorts** Literal
  **lexical syntax**
    "\\"~□ → EscChar
    "\\"[01][0-7][0-7] → EscChar
    ~[\000-\040"\\] → L-Char
    [␣\t\n] → L-Char
    EscChar → L-Char
    "\""L-Char∗"\"" → Literal
  **variables**
    "*L*"[*0-9′*]∗ → Literal

## A.2 ATerms

### A.2.1 Constructors

**module** Grammar-Tree-Constructors
**imports** ATerms[2.5.1]
**exports**
  **context-free syntax**
    "empty-grammar" → AFun
    "conc-grammars" → AFun

**module** CC-Sdf-Tree-Constructors
**imports** Kernel-Sdf-Tree-Constructors[2.5.2]
**exports**
  **context-free syntax**
    "char-class" → AFun

"range" → AFun

**module** Sorts-Sdf-Tree-Constructors
**imports** Kernel-Sdf-Tree-Constructors$^{2.5.2}$
**exports**
**context-free syntax**
"sort" → AFun

**module** Literals-Sdf-Tree-Constructors
**imports** Kernel-Sdf-Tree-Constructors$^{2.5.2}$
**exports**
**context-free syntax**
"lit" → AFun

**module** Regular-Sdf-Tree-Constructors
**imports** ATerms$^{2.5.1}$
**exports**
**context-free syntax**
"empty" → AFun
"seq" → AFun
"opt" → AFun
"iter" → AFun
"iter-star" → AFun
"iter-sep" → AFun
"iter-star-sep" → AFun
"iter-n" → AFun
"iter-sep-n" → AFun
"set" → AFun
"pair" → AFun
"func" → AFun
"alt" → AFun
"perm" → AFun

**module** Basic-Sdf-Tree-Constructors
**imports** Kernel-Sdf-Tree-Constructors$^{2.5.2}$
**exports**
**context-free syntax**
"lexical-syntax" → AFun
"context-free-syntax" → AFun
"variables" → AFun
"lexical-variables" → AFun
"cf" → AFun
"lex" → AFun
"varsym" → AFun
"layout" → AFun

### A.2.2 Encoding and Decoding

**module** Grammar-ATerms
**imports** Grammar-Tree-Constructors$^{A.2.1}$ Grammar-Syntax$^{2.2}$
**exports**
**context-free syntax**

aterm(Grammar) $\rightarrow$ ATerm
grammar(ATerm) $\rightarrow$ Grammar

**equations**

Encoding of grammars.

$$\text{[1]} \quad \text{aterm}(\emptyset) = \text{empty-grammar}$$
$$\text{[2]} \quad \text{aterm}(\mathcal{G}_1\ \mathcal{G}_2) = \text{conc-grammars}(\text{aterm}(\mathcal{G}_1),\ \text{aterm}(\mathcal{G}_2))$$

Decoding of grammars.

$$\text{[3]} \quad \text{grammar}(\text{empty-grammar}) = \emptyset$$
$$\text{[4]} \quad \text{grammar}(\text{conc-grammars}(T_1,\ T_2)) = \text{grammar}(T_1)\ \text{grammar}(T_2)$$

**module** CC-Sdf-ATerms
**imports** Kernel-Sdf-ATerms$^{2.5.4}$ CC-Sdf-Syntax$^{2.4.2}$
CC-Sdf-Tree-Constructors$^{A.2.1}$ Character-Arithmetic
**exports**
**context-free syntax**

atermlist(OptCharRanges) $\rightarrow$ ATermList
aterm(Character) $\rightarrow$ NatCon
ranges(ATermList) $\rightarrow$ CharRanges
range(ATerm) $\rightarrow$ CharRange
character(ATerm) $\rightarrow$ Character

**equations**

Encoding character classes.

$$\text{[1]} \quad \text{aterm}([cr^*]) = \text{char-class}(\text{atermlist}(cr^*))$$
$$\text{[2]} \quad \text{atermlist}(cr^*) = [\,] \quad \textbf{when} \quad cr^* =$$
$$\text{[3]} \quad \text{atermlist}(cr_1^+\ cr_2^+) = \text{atermlist}(cr_1^+) \mathbin{+\!\!+} \text{atermlist}(cr_2^+)$$
$$\text{[4]} \quad \text{atermlist}(cr) = [\text{aterm}(\mathsf{c})] \quad \textbf{when} \quad cr = \mathsf{c}$$
$$\text{[5]} \quad \text{atermlist}(\mathsf{c}_1 - \mathsf{c}_2) = [\text{range}(\text{aterm}(\mathsf{c}_1),\ \text{aterm}(\mathsf{c}_2))]$$
$$\text{[6]} \quad \text{aterm}(\mathsf{c}) = \text{int}(\mathsf{c})$$

Decoding character classes.

$$\text{[7]} \quad \text{symbol}(\text{char-class}([\,])) = []$$
$$\text{[8]} \quad \text{symbol}(\text{char-class}([Ts])) = [\text{ranges}([Ts])]$$
$$\text{[9]} \quad \text{ranges}([T]) = \text{range}(T)$$
$$\text{[10]} \quad \text{ranges}([T,\ Ts]) = \text{range}(T)\ \text{ranges}([Ts])$$
$$\text{[11]} \quad \text{range}(n) = \text{character}(n)$$
$$\text{[12]} \quad \text{range}(\text{range}(n_1,\ n_2)) = \text{character}(n_1) - \text{character}(n_2)$$
$$\text{[13]} \quad \text{character}(n) = \text{char}(n)$$
$$\text{[14]} \quad \text{symbol}(n) = [\text{char}(n)]$$

**module** Sorts-Sdf-ATerms
**imports** Kernel-Sdf-ATerms$^{2.5.4}$ Sorts-Sdf-Tree-Constructors$^{A.2.1}$
Sorts-Sdf-Syntax$^{2.4.1}$
**equations**

Encoding and decoding sorts.

$$\text{[1]} \quad \text{aterm}(\text{sort}(c^+)) = \text{sort}(\text{literal}(\texttt{"\\""}\ c^+\ \texttt{"\\""}))$$

[2] $\text{symbol}(\text{sort}(\text{literal}(\texttt{"\""}\ c^{+}\ \texttt{"\""}))) = \text{sort}(c^{+})$

**module** Literals-Sdf-ATerms
**imports** Kernel-Sdf-ATerms$^{2.5.4}$ Literals-Sdf-Tree-Constructors$^{A.2.1}$
Literals-Sdf-Syntax$^{2.4.3}$
**equations**
Encoding and decoding literals.

[1] $\text{aterm}(L) = \text{lit}(L)$
[2] $\text{symbol}(\text{lit}(L)) = L$

**module** Priority-Sdf-ATerms
**imports** Kernel-Sdf-ATerms$^{2.5.4}$ Priority-Sdf-Syntax$^{3.1.1}$
**equations**
Encoding attributes.

[1] $\text{aterm}(\text{left}) = \text{atr}(\texttt{"left"})$
[2] $\text{aterm}(\text{right}) = \text{atr}(\texttt{"right"})$
[3] $\text{aterm}(\text{bracket}) = \text{atr}(\texttt{"bracket"})$
[4] $\text{aterm}(\text{assoc}) = \text{atr}(\texttt{"assoc"})$
[5] $\text{aterm}(\text{non-assoc}) = \text{atr}(\texttt{"non-assoc"})$

Decoding attributes.

[6] $\text{attribute}(\text{atr}(\texttt{"left"})) = \text{left}$
[7] $\text{attribute}(\text{atr}(\texttt{"right"})) = \text{right}$
[8] $\text{attribute}(\text{atr}(\texttt{"bracket"})) = \text{bracket}$
[9] $\text{attribute}(\text{atr}(\texttt{"assoc"})) = \text{assoc}$
[10] $\text{attribute}(\text{atr}(\texttt{"non-assoc"})) = \text{non-assoc}$

**module** Regular-Sdf-ATerms
**imports** Regular-Sdf-Tree-Constructors$^{A.2.1}$ Kernel-Sdf-ATerms$^{2.5.4}$
Regular-Sdf-Syntax$^{3.2.1}$
**equations**
Encoding regular expressions.

[1] $\text{aterm}((\ )) = \text{empty}$
[2] $\text{aterm}((\mathcal{A}\ \alpha^{+})) = \text{seq}(\text{atermlist}(\mathcal{A}\ \alpha^{+}))$
[3] $\text{aterm}(\mathcal{A}?) = \text{opt}(\text{aterm}(\mathcal{A}))$
[4] $\text{aterm}(\mathcal{A}+) = \text{iter}(\text{aterm}(\mathcal{A}))$
[5] $\text{aterm}(\mathcal{A}*) = \text{iter-star}(\text{aterm}(\mathcal{A}))$
[6] $\text{aterm}(\{\mathcal{A}\ \mathcal{B}\}+) = \text{iter-sep}(\text{aterm}(\mathcal{A}), \text{aterm}(\mathcal{B}))$
[7] $\text{aterm}(\{\mathcal{A}\ \mathcal{B}\}*) = \text{iter-star-sep}(\text{aterm}(\mathcal{A}), \text{aterm}(\mathcal{B}))$
[8] $\text{aterm}(\{\mathcal{A}\}\ n+) = \text{iter-n}(\text{aterm}(\mathcal{A}), \text{con}(n))$
[9] $\text{aterm}(\{\mathcal{A}\ \mathcal{B}\}\ n+) = \text{iter-sep-n}(\text{aterm}(\mathcal{A}), \text{aterm}(\mathcal{B}), \text{con}(n))$
[10] $\text{aterm}(\text{Set}[\mathcal{A}]) = \text{set}(\text{aterm}(\mathcal{A}))$
[11] $\text{aterm}(\mathcal{A}\ \#\ \mathcal{B}) = \text{pair}(\text{aterm}(\mathcal{A}), \text{aterm}(\mathcal{B}))$

[12] $\text{aterm}((\alpha \Rightarrow \mathcal{B})) = \text{func}(\text{atermlist}(\alpha), \text{aterm}(\mathcal{B}))$
[13] $\text{aterm}(\mathcal{A} \mid \mathcal{B}) = \text{alt}(\text{aterm}(\mathcal{A}), \text{aterm}(\mathcal{B}))$
[14] $\text{aterm}(\ll \alpha \gg) = \text{perm}(\text{atermlist}(\alpha))$

Decoding regular expressions.

[15] $\text{symbol}(\text{empty}) = (\ )$
[16] $\text{symbol}(\text{seq}(Tl)) = (\mathcal{A}\ \alpha^{+}) \quad \textbf{when}\ \ \text{symbols}(Tl) = \mathcal{A}\ \alpha^{+}$
[17] $\text{symbol}(\text{opt}(T_1)) = \text{symbol}(T_1)?$
[18] $\text{symbol}(\text{iter}(T_1)) = \text{symbol}(T_1)+$
[19] $\text{symbol}(\text{iter-star}(T_1)) = \text{symbol}(T_1)*$
[20] $\text{symbol}(\text{iter-sep}(T_1,\ T_2)) = \{\text{symbol}(T_1)\ \text{symbol}(T_2)\}+$
[21] $\text{symbol}(\text{iter-star-sep}(T_1,\ T_2)) = \{\text{symbol}(T_1)\ \text{symbol}(T_2)\}*$
[22] $\text{symbol}(\text{iter-n}(T_1,\ n)) = \{\text{symbol}(T_1)\}\ n+$
[23] $\text{symbol}(\text{iter-sep-n}(T_1,\ T_2,\ n)) = \{\text{symbol}(T_1)\ \text{symbol}(T_2)\}\ n+$
[24] $\text{symbol}(\text{set}(T_1)) = \text{Set}[\text{symbol}(T_1)]$
[25] $\text{symbol}(\text{pair}(T_1,\ T_2)) = \text{symbol}(T_1)\ \#\ \text{symbol}(T_2)$
[26] $\text{symbol}(\text{func}(Tl_1,\ T_2)) = (\text{symbols}(Tl_1) \Rightarrow \text{symbol}(T_2))$
[27] $\text{symbol}(\text{alt}(T_1,\ T_2)) = \text{symbol}(T_1) \mid \text{symbol}(T_2)$
[28] $\text{symbol}(\text{perm}(Tl)) = \ll \text{symbols}(Tl) \gg$

**module** Basic-Sdf-ATerms
**imports** Basic-Sdf-Tree-Constructors$^{A.2.1}$ Basic-Sdf-Syntax$^{3.3.1}$
Kernel-Sdf-ATerms$^{2.5.4}$
**equations**
Encoding grammars.

[1] $\text{aterm}(\text{lexical syntax}\ p^*) = \text{lexical-syntax}(\text{atermlist}(p^*))$
[2] $\text{aterm}(\text{context-free syntax}\ p^*) = \text{context-free-syntax}(\text{atermlist}(p^*))$
[3] $\text{aterm}(\text{variables}\ p^*) = \text{variables}(\text{atermlist}(p^*))$
[4] $\text{aterm}(\text{lexical variables}\ p^*) = \text{lexical-variables}(\text{atermlist}(p^*))$

Encoding symbols.

[5] $\text{aterm}(\langle\mathcal{A}\texttt{-LEX}\rangle) = \text{lex}(\text{aterm}(\mathcal{A}))$
[6] $\text{aterm}(\langle\mathcal{A}\texttt{-CF}\rangle) = \text{cf}(\text{aterm}(\mathcal{A}))$
[7] $\text{aterm}(\langle\mathcal{A}\texttt{-VAR}\rangle) = \text{varsym}(\text{aterm}(\mathcal{A}))$
[8] $\text{aterm}(\texttt{LAYOUT}) = \text{layout}$

Decoding grammars.

[9] $\text{grammar}(\text{lexical-syntax}(Tl)) = \text{lexical syntax productions}(Tl)$
[10] $\text{grammar}(\text{context-free-syntax}(Tl)) = \text{context-free syntax productions}(Tl)$
[11] $\text{grammar}(\text{variables}(Tl)) = \text{variables productions}(Tl)$
[12] $\text{grammar}(\text{lexical-variables}(Tl)) = \text{lexical variables productions}(Tl)$

Decoding symbols.

[13] $\text{symbol}(\text{lex}(T)) = \langle\text{symbol}(T)\texttt{-LEX}\rangle$
[14] $\text{symbol}(\text{cf}(T)) = \langle\text{symbol}(T)\texttt{-CF}\rangle$
[15] $\text{symbol}(\text{varsym}(T)) = \langle\text{symbol}(T)\texttt{-VAR}\rangle$
[16] $\text{symbol}(\text{layout}) = \texttt{LAYOUT}$

## A.3 Renamings

**module** CC-Sdf-Renaming
**imports** Kernel-Sdf-Renaming$^{4.1.3}$ CC-Sdf-Syntax$^{2.4.2}$
**equations**
Renaming character classes.

[1] $$[cc]\ \rho = cc$$

**module** Literals-Sdf-Renaming
**imports** Kernel-Sdf-Renaming$^{4.1.3}$ Literals-Sdf-Syntax$^{2.4.3}$
**hiddens**
  **variables**
    "$L$" $\rightarrow$ Literal
**equations**
Renaming literals.

[1] $$[L]\ \rho = L$$

**module** Sorts-Sdf-Renaming
**imports** Sorts-Sdf-Syntax$^{2.4.1}$ Kernel-Sdf-Renaming$^{4.1.3}$
**equations**
Renaming sorts.

[1] $$[\mathcal{S}]\ \rho = \mathcal{S}$$
[2] $$(\text{sorts}\ \alpha)\ \rho = \text{sorts}\ (\alpha\ )*\ \rho$$

**module** Priority-Sdf-Renaming
**imports** Priority-Sdf-Projection$^{3.1.2}$ Basic-Sdf-Renaming$^{A.3}$
**exports**
  **context-free syntax**
    "(" Priorities ")" Renamings $\rightarrow$ Priorities
    "(" Group ")$_{\mathrm{G}}$" Renamings $\rightarrow$ Group
**equations**
Renaming symbols and productions in priority declarations.

[1] $$(\text{priorities}\ pr^*)\ \rho = \text{priorities}\ (pr^*)\ \rho$$

Lists of priorities.

[2] $$(pr^*)\ \rho = \quad \textbf{when}\ \ pr^* =$$
[3] $$(pr_1^+, pr_2^+)\ \rho = (pr_1^+)\ \rho +\!\!+ (pr_2^+)\ \rho$$

Associativity and priority declarations.

[4] $$(g_1\ as\ g_2)\ \rho = (g_1\ )_{\mathrm{G}}\ \rho\ as\ (g_2\ )_{\mathrm{G}}\ \rho$$
[5] $$(g_1 > g_2)\ \rho = (g_1\ )_{\mathrm{G}}\ \rho > (g_2\ )_{\mathrm{G}}\ \rho$$
[6] $$(g_1 > g_2 > gg^+)\ \rho = (g_1\ )_{\mathrm{G}}\ \rho > g_2' > gg'^+ \quad \textbf{when}\ \ g_2' > gg'^+ = (g_2 > gg^+)\ \rho$$

Groups.

[7] $(g\ )_{\mathrm{G}}\ \rho = p' \quad \textbf{when}\ \ g = p,\ \ p' = (p)\ \rho$
[8] $(\{p^*\}\ )_{\mathrm{G}}\ \rho = \{(p^*\ )*\ \rho\}$
[9] $(\{as : p^*\}\ )_{\mathrm{G}}\ \rho = \{as : (p^*\ )*\ \rho\}$

**module** Regular-Sdf-Renaming
**imports** Kernel-Sdf-Renaming$^{4.1.3}$ Regular-Sdf-Syntax$^{3.2.1}$
**equations**
Renaming symbols in regular expressions.

[1] $[(\ )]\ \rho = (\ )$
[2] $[(\mathcal{A}\ \alpha^+)]\ \rho = (\mathcal{B}\ \beta^+) \quad \textbf{when}\ \ (\mathcal{A}\ \alpha^+\ )*\ \rho = \mathcal{B}\ \beta^+$
[3] $[\mathcal{A}?]\ \rho = (\mathcal{A})\ \rho?$
[4] $[\mathcal{A}+]\ \rho = (\mathcal{A})\ \rho+$
[5] $[\mathcal{A}*]\ \rho = (\mathcal{A})\ \rho*$
[6] $[\{\mathcal{A}\ \mathcal{B}\}+]\ \rho = \{(\mathcal{A})\ \rho\ (\mathcal{B})\ \rho\}+$
[7] $[\{\mathcal{A}\ \mathcal{B}\}*]\ \rho = \{(\mathcal{A})\ \rho\ (\mathcal{B})\ \rho\}*$
[8] $[\{\mathcal{A}\}\ n+]\ \rho = \{(\mathcal{A})\ \rho\}\ n+$
[9] $[\{\mathcal{A}\ \mathcal{B}\}\ n+]\ \rho = \{(\mathcal{A})\ \rho\ (\mathcal{B})\ \rho\}\ n+$
[10] $[\mathrm{Set}[\mathcal{A}]]\ \rho = \mathrm{Set}[(\mathcal{A})\ \rho]$
[11] $[\mathcal{A}\ \#\ \mathcal{B}]\ \rho = (\mathcal{A})\ \rho\ \#\ (\mathcal{B})\ \rho$
[12] $[(\alpha \Rightarrow \mathcal{B})]\ \rho = ((\alpha\ )*\ \rho \Rightarrow (\mathcal{B})\ \rho)$
[13] $[\mathcal{A} \mid \mathcal{B}]\ \rho = (\mathcal{A})\ \rho \mid (\mathcal{B})\ \rho$

**module** Basic-Sdf-Renaming
**imports** Basic-Sdf-Normalization$^{3.3.2}$ Kernel-Sdf-Renaming$^{4.1.3}$
**exports**
**context-free syntax**
"<" Renamings "`-LEX`" ">" → Renamings
"<" Renamings "`-CF`" ">" → Renamings
**equations**
Renaming grammars.

[1] $(\text{context-free syntax}\ p^*)\ \rho = \text{context-free syntax}\ (p^*\ )*\ \rho$
[2] $(\text{lexical syntax}\ p^*)\ \rho = \text{lexical syntax}\ (p^*\ )*\ \rho$
[3] $(\text{variables}\ p^*)\ \rho = \text{variables}\ (p^*\ )*\ \rho$
[4] $(\text{lexical variables}\ p^*)\ \rho = \text{lexical variables}\ (p^*\ )*\ \rho$

Renaming symbols.

[5] $[\langle\mathcal{A}\texttt{-LEX}\rangle]\ \rho = \langle(\mathcal{A})\ \rho\texttt{-LEX}\rangle$
[6] $[\langle\mathcal{A}\texttt{-CF}\rangle]\ \rho = \langle(\mathcal{A})\ \rho\texttt{-CF}\rangle$
[7] $[\langle\mathcal{A}\texttt{-VAR}\rangle]\ \rho = \langle(\mathcal{A})\ \rho\texttt{-VAR}\rangle$
[8] $[\texttt{LAYOUT}]\ \rho = \texttt{LAYOUT}$

Applying $\langle\_\texttt{-LEX}\rangle$ to a renaming.

[9] $\langle[]\texttt{-LEX}\rangle = []$

[10] $\langle[\mathcal{A} \Rightarrow \mathcal{B}]\texttt{-LEX}\rangle = [\langle\mathcal{A}\texttt{-LEX}\rangle \Rightarrow \langle\mathcal{B}\texttt{-LEX}\rangle]$
[11] $\langle[p_1 \Rightarrow p_2]\texttt{-LEX}\rangle = [\langle p_1\texttt{-LEX}\rangle \Rightarrow \langle p_2\texttt{-LEX}\rangle]$
[12] $\langle[\rho_1^+ \ \rho_2^+]\texttt{-LEX}\rangle = \langle[\rho_1^+]\texttt{-LEX}\rangle + \langle[\rho_2^+]\texttt{-LEX}\rangle$

Applying $\langle\_\texttt{-CF}\rangle$ to a renaming.

[13] $\langle[]\texttt{-CF}\rangle = []$
[14] $\langle[\mathcal{A} \Rightarrow \mathcal{B}]\texttt{-CF}\rangle = [\langle\mathcal{A}\texttt{-CF}\rangle \Rightarrow \langle\mathcal{B}\texttt{-CF}\rangle]$
[15] $\langle[p_1 \Rightarrow p_2]\texttt{-CF}\rangle = [\langle p_1\texttt{-CF}\rangle \Rightarrow \langle p_2\texttt{-CF}\rangle]$
[16] $\langle[\rho_1^+ \ \rho_2^+]\texttt{-CF}\rangle = \langle[\rho_1^+]\texttt{-CF}\rangle + \langle[\rho_2^+]\texttt{-CF}\rangle$

**module** Restrictions-Sdf-Renaming
**imports** Restrictions-Sdf-Syntax[3.4.1] Kernel-Sdf-Renaming[4.1.3]
**exports**
**context-free syntax**
"(" Restrictions ")" Renamings → Restrictions
**equations**
Renaming restrictions.

[1] (restrictions $restr^*$) $\rho$ = restrictions ($restr^*$) $\rho$
[2] ($restr^*$) $\rho$ = **when** $restr^*$ =
[3] $(\alpha \not- cc\ restr_1^*)\ \rho = (\alpha\ )*\ \rho \not- cc\ restr_2^*$ **when** $restr_2^* = (restr_1^*)\ \rho$

## A.4 SDF2

**module** Sdf2-Projection
**imports** Kernel-Sdf-Projection[2.3.2] Sorts-Sdf-Projection[2.4.1] Priority-Sdf-Projection[3.1.2] Renaming-Sdf-Projection[4.1.2] Modular-Sdf-Projection[4.3.2] Alias-Sdf-Projection[4.2.2] Restrictions-Sdf-Projection[3.4.2]

**module** Sdf2-Renaming
**imports** Sdf2-Syntax[5.1.1] Kernel-Sdf-Renaming[4.1.3] Priority-Sdf-Renaming[A.3] Regular-Sdf-Renaming[A.3] Literals-Sdf-Renaming[A.3] CC-Sdf-Renaming[A.3] Basic-Sdf-Renaming[A.3] Sorts-Sdf-Renaming[A.3] Restrictions-Sdf-Renaming[A.3] Modular-Sdf-Renaming[4.3.4] Alias-Sdf-Renaming[4.2.3]
**equations**

[1] (context-free priorities $pr^*$) $\rho$ = context-free priorities ($pr^*$) $\rho$
[2] (lexical priorities $pr^*$) $\rho$ = lexical priorities ($pr^*$) $\rho$

[3] $[\langle\texttt{Start}\rangle]\ \rho = \langle\texttt{Start}\rangle$
[4] $[\langle\texttt{START}\rangle]\ \rho = \langle\texttt{START}\rangle$

**module** Sdf2-Tree-Constructors
**imports** Kernel-Sdf-Tree-Constructors[2.5.2] Basic-Sdf-Tree-Constructors[A.2.1]
Modular-Sdf-Tree-Constructors Regular-Sdf-Tree-Constructors[A.2.1]
Priority-Sdf-Tree-Constructors CC-Sdf-Tree-Constructors[A.2.1]
Sorts-Sdf-Tree-Constructors[A.2.1] Literals-Sdf-Tree-Constructors[A.2.1]

**module** Sdf2-ATerms
**imports** Sdf2-Tree-Constructors[A.4] Sdf2-Syntax[5.1.1] Kernel-Sdf-ATerms[2.5.4]
Basic-Sdf-ATerms[A.2.2] Modular-Sdf-ATerms Regular-Sdf-ATerms[A.2.2]
Priority-Sdf-ATerms[A.2.2] CC-Sdf-ATerms[A.2.2] Sorts-Sdf-ATerms[A.2.2]
Literals-Sdf-ATerms[A.2.2] Restrictions-Sdf-ATerms
**equations**

[1] $\mathrm{aterm}(\langle \texttt{START} \rangle) = \mathrm{sort}(\texttt{"<START>"})$
[2] $\mathrm{aterm}(\langle \texttt{Start} \rangle) = \mathrm{sort}(\texttt{"<Start>"})$

**module** Sdf2-Trees
**imports** Sdf2-ATerms[A.4] Sdf2-Syntax[5.1.1] Kernel-Sdf-Trees[2.5.5]
Priority-Sdf-Trees[3.1.4] CC-Sdf-Trees[2.5.6] Renaming-Sdf-Trees[4.1.4]

**module** Sdf2-Equality
**imports** Kernel-Sdf-Equality[2.5.8] Regular-Sdf-Equality Basic-Sdf-Equality[3.3.3]

# B

# Bibliography

Technical reports from the Programming Research Group of the University of Amsterdam can be obtained from `http://www.wins.uva.nl/research/prog/reports/reports.html`. Technical reports from CWI can be found at `http://www.cwi.nl/cwi/publications/reports/reports.html`.


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

Backus, J. W. (1959). The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *Proceedings of the International Conference on Information Processing*, pages 125–132, Paris. UNESCO.

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

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

Bergstra, J. A., Heering, J., and Klint, P. (1989b). The algebraic specification formalism ASF. In J. A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley. Chapter 1.

Bergstra, J. A., Heering, J., and Klint, P. (1990). Module algebra. *Journal of the ACM*, **37**(2), 335–372.

Billot, S. and Lang, B. (1989). The structure of shared forests in ambiguous parsing. In *Proceedings of the Twenty-Seventh Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Van den Brand, M. G. J. and Visser, E. (1996). Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, **5**(1), 1–41.

Van den Brand, M. G. J., Klint, P., Olivier, P., and Visser, E. (1997). ATerms: Representing structured data for exchange between heterogeneous tools. Technical report, Programming Research Group, University of Amsterdam.

Cameron, R. D. (1993). Extending context-free grammars with permutation phrases. *ACM Letters on Programming Languages and Systems*, **2**(1-4), 85–94.

Cardelli, L., Matthes, F., and Abadi, M. (1994). Extensible syntax with lexical scoping. SRC Research Report 121, Digital Systems Research Center, Palo Alto, California.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, **2**, 113–124.

Cordy, J. R. and Carmichael, I. H. (1993). *The TXL Programming Language. Syntax and Informal Semantics. Version 7*. Software Technology Laboratory, Department of Computer and Information Science, Queen's University at Kingston, Kinston, Canada, 7 edition.

Van Deursen, A., Heering, J., and Klint, P., editors (1996). *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore.

Dinesh, T. B. (1995). Injection misdemeanors. In M. G. J. v. d. Brand *et al.*, editors, *ASF+SDF'95. A Workshop on Generating Tools from Algebraic Specifications*, pages 255–270. Technical Report P9504, Programming Research Group, University of Amsterdam.

Earley, J. (1975). Ambiguity and precedence in syntax description. *Acta Informatica*, **4**(1), 183–192.

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

Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B. (1977). Initial algebra semantics and continuous algebras. *Journal of the ACM*, **24**(1), 68–95.

Gray, R. W., Heuring, V. P., Levi, S. P., Sloane, A. M., and Waite, W. M. (1992). Eli: A complete, flexible compiler construction system. *Communications of the ACM*, **35**, 121–131.

Grosch, J. (1990). Lalr - a generator for efficient parsers. *Software-Practice & Experience*, **20**, 1115–1135.

Hatcher, W. S. and Rus, T. (1976). Context-free algebras. *Journal of Cybernetics*, **6**, 65–76.

Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989). The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, **24**(11), 43–75.

Hendriks, P. R. H. (1991). *Implementation of Modular Algebraic Specifications*. Ph.D. thesis, University of Amsterdam.

Johnson, S. C. (1975). YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories, Murray Hill, N.J.

Kahn, G., Lang, B., Mélèse, B., and Morcos, E. (1983). METAL: A formalism to specify formalisms. *Science of Computer Programming*, **3**, 151–188.

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

Klint, P. and Visser, E. (1994). Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical Systems Theory*, **2**(2), 127–145. Correction in: Mathematical Systems Theory 5(1), pp. 95-96, Springer-Verlag, 1971.

Koster, C. H. A. (1971). Affix grammars. In J. E. L. Peck, editor, *Algol-68 Implementation*. North-Holland, Amsterdam.

Lee, J. A. N. (1972). The formal definition of the BASIC language. *Computer Journal*, **15**(1), 37–41.

Lesk, M. E. and Schmidt, E. (1986). *LEX — A lexical analyzer generator*. Bell Laboratories. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).

Luttik, B. and Visser, E. (1997). Specification of rewriting strategies. In A. Sellink, editor, *Second International Conference on the Theory and Practice of Algebraic Specification (ASF+SDF'97)*, Amsterdam, The Netherlands. Programming Research Group, University of Amsterdam.

Meinke, K. (1992). Equational specification of abstract types and combinators. In E. Boerger, G. Jaeger, H. K. Buening, and M. M. Richter, editors, *Computer Science Logic - CSL'91*, volume 626 of *Lecture Notes in Computer Science*, pages 257–271, Berlin. Springer-Verlag.

Mosses, P. D. (1992). *Action Semantics*. Cambridge University Press.

Naur, P. *et al.* (1960). Report on the algorithmic language ALGOL 60. *Communications of the ACM*, **3**(5), 299–314.

Parr, T. J. and Quong, R. W. (1994). Adding semantic and syntactic predicates to LL(k): pred-LL(k). In P. A. Fritzson, editor, *Compiler Construction, 5th International Conference, CC'94*, volume 786 of *LNCS*, pages 263–277, Edinburgh, U.K. Springer-Verlag.

Pereira, F. C. N. and Warren, D. H. D. (1980). Definite Clause Grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, **13**, 231–278.

Rekers, J. (1992). *Parser Generation for Interactive Environments*. Ph.D. thesis, University of Amsterdam. ftp://ftp.cwi.nl/pub/gipe/reports/Rek92.ps.Z.

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

Salomon, D. J. and Cormack, G. V. (1989). Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Notices*, **24**(7), 170–178.

Salomon, D. J. and Cormack, G. V. (1995). The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Technical Report 95/06, Department of Computer Science, University of Manitoba, Winnipeg, Canada.

Thorup, M. (1994). Controlled grammatic ambiguity. *ACM Transactions on Programming Languages and Systems*, **16**(3), 1024–1050.

Visser, E. (1995). A case study in optimizing parsing schemata by disambiguation filters. In S. Fischer and M. Trautwein, editors, *Proceedings Accolade95*, pages 153–167, Amsterdam. The Dutch Graduate School in Logic.

Visser, E. (1996). Multi-level specifications. In A. van Deursen, J. Heering, and P. Klint, editors, *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*, pages 105–196. World Scientific, Singapore.

Visser, E. (1997a). A case study in optimizing parsing schemata by disambiguation filters. In A. Nijholt, editor, *International Workshop on Parsing Technology IWPT'97*, Boston, USA. (To appear).

Visser, E. (1997b). Character classes. Technical Report P9708, Programming Research Group, University of Amsterdam.

Visser, E. (1997c). Executable specification of programming languages. (Draft).

Visser, E. (1997d). Polymorphic syntax definition. *Theoretical Computer Science*. (To appear).

Visser, E. (1997e). Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam.

Vittek, M. (1994). *ELAN: Un cadre logique pour le prototypage de language de programmation avec contraintes*. Ph.D. thesis, Université Henri Poincaré – Nancy I, Nancy, France.

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

Watt, D. A. (1977). The parsing problem for affix grammars. *Acta Informatica*, **8**(1), 1–20.

Williams, M. H. (1982). A flexible notation for syntactic definitions. *ACM Transactions on Programming Languages and Systems*, **4**(1), 113–119.

Wirth, N. (1977). What can we do about the unnecessary diversity of notation for syntactic definitions. *Communications of the ACM*, **20**(11), 822–823.

## Technical Reports of the Programming Research Group

**Note:** These reports can be obtained using the technical reports overview on our WWW site (URL `http://www.wins.uva.nl/research/prog/reports/`) or using anonymous ftp to `ftp.wins.uva.nl`, directory `pub/programming-research/reports/`.


[P9711] L. Moonen. *A Generic Architecture for Data Flow Analysis to Support Reverse Engineering.*

[P9710] B. Luttik and E. Visser. *Specification of Rewriting Strategies.*

[P9709] J.A. Bergstra and M.P.A. Sellink. *An Arithmetical Module for Rationals and Reals.*

[P9708] E. Visser. *Character Classes.*

[P9707] E. Visser. *Scannerless generalized-LR parsing.*

[P9706] E. Visser. *A family of syntax definition formalisms.*

[P9705] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Generation of Components for Software Renovation Factories from Context-free Grammars.*

[P9704] P.A. Olivier. *Debugging Distributed Applications Using a Coordination Architecture.*

[P9703] H.P. Korver and M.P.A. Sellink. *A Formal Axiomatization for Alphabet Reasoning with Parametrized Processes.*

[P9702] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Reengineering COBOL Software Implies Specification of the Underlying Dialects.*

[P9701] E. Visser. *Polymorphic Syntax Definition.*

[P9618] M.G.J. van den Brand, P. Klint, and C. verhoef. *Re-engineering needs Generic Programming Language Technology.*

[P9617] P.I. Manuel. *ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool.*

[P9616] P.H. Rodenburg. *A Complete System of Four-valued Logic.*

[P9615] S.P. Luttik and P.H. Rodenburg. *Transformations of Reduction Systems.*

[P9614] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Core Technologies for System Renovation.*

[P9613] L. Moonen. *Data Flow Analysis for Reverse Engineering.*

[P9612] J.A. Hillebrand. *Transforming an ASF+SDF Specification into a ToolBus Application.*

[P9611] M.P.A. Sellink. *On the conservativity of Leibniz Equality.*

[P9610] T.B. Dinesh and S.M. Üsküdarlı. *Specifying input and output of visual languages.*

[P9609] T.B. Dinesh and S.M. Üsküdarlı. *The VAS formalism in VASE.*

[P9608] J.A. Hillebrand. *A small language for the specification of Grid Protocols.*

[P9607] J.J. Brunekreef. *A transformation tool for pure Prolog programs: the algebraic specification.*

[P9606] E. Visser. *Solving type equations in multi-level specifications (preliminary version).*

[P9605] P.R. D'Argenio and C. Verhoef. *A general conservative extension theorem in process algebras with inequalities.*

[P9602b] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives (revised version of P9602).*

[P9604] E. Visser. *Multi-level specifications.*

[P9603] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Reverse engineering and system renovation: an annotated bibliography.*

[P9602] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives.*

[P9601] P.A. Olivier. *Embedded system simulation: testdriving the ToolBus.*

[P9512] J.J. Brunekreef. *TransLog, an interactive tool for transformation of logic programs.*

[P9511] J.A. Bergstra, J.A. Hillebrand, and A. Ponse. *Grid protocols based on synchronous communication: specification and correctness.*

[P9510] P.H. Rodenburg. *Termination and confluence in infinitary term rewriting.*

[P9509] J.A. Bergstra and Gh. Stefanescu. *Network algebra with demonic relation operators.*

[P9508] J.A. Bergstra, C.A. Middelburg, and Gh. Stefanescu. *Network algebra for synchronous and asynchronous dataflow.*

[P9507] E. Visser. *A case study in optimizing parsing schemata by disambiguation filters.*

[P9506] M.G.J. van den Brand and E. Visser. *Generation of formatters for context-free languages.*

[P9505] J.M.T. Romijn. *Automatic analysis of term rewriting systems: proving properties of term rewriting systems derived from* ASF+SDF *specifications.*