A Family of Syntax Definition Formalisms

Eelco Visser

Programming Research Group, University of Amsterdam,
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands
email: visser@fwi.uva.nl, http://adam.fwi.uva.nl/~visser/

Abstract. In this paper we design a syntax definition formalism as a family of formalisms. Starting with a small kernel, various features for syntax definition are designed orthogonally to each other. This provides a framework for constructing new formalisms by adapting and extending old ones. The formalism is developed with the algebraic specification formalism ASF+SDF. It provides the following features: lexical and context-free syntax, variables, disambiguation by priorities, regular expressions, character classes and modular definitions. New are the uniform treatment of lexical syntax, context-free syntax and variables, the treatment of regular expressions by normalization yielding abstract syntax without auxiliary sorts, regular expressions as result of productions and modules with hidden imports and renamings.

Key Words & Phrases: syntax definition formalism, language design, context-free grammar, context-free syntax, lexical syntax, priorities, regular expressions, formal language, parsing, abstract syntax, module, renaming, hidden imports

Note: Supported by the Dutch Organization for Scientific Research (NWO) under grant 612-317-420: Incremental parser generation and context-dependent disambiguation, a multi-disciplinary perspective.

1 Introduction

1.1 General

New programming, specification and special purpose languages are being developed continuously [C+94]. 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 in 1956 [Cho56]. A context-free grammar is a set of string rewrite rules of the form $\alpha \to \mathcal{A}$. A string w is 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 $w = \alpha_0 \to \alpha_1 \to \ldots \to \alpha_n = S$ and each step has the form $\alpha_i\beta_i\gamma_i \to \alpha_i\mathcal{B}_i\gamma_i$ where $\beta_i \to \mathcal{B}_i$ is a production in $\mathcal{G}$.

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) [Bac59, N+60], originally developed 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. Several standard notations for syntax definition have been proposed [Wir77, Wil82]. None of these has been convincing, instead a number of similar or overlapping formalisms exist.

Proceedings of ASF+SDF95. A workshop on Generating Tools from Algebraic Specifications. May 11 & 12, 1995, CWI, Amsterdam, M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman & E. Visser (eds.) Technical Report P9504, Programming Research Group, University of Amsterdam

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, interfacing between the formalism and its environment, e.g. mapping to abstract syntax. Furthermore, various extensions of context-free grammars are developed for attaching semantics to grammars: attribute grammars [Knu68], affix grammars [Kos71], and definite clause grammars [PW80]. Examples of formalisms built on one or more of these themes are: LEX [LS86], YACC [Joh75], PSG [BS86], Cigale [Voi86], synthesizer generator [RT89], METAL [Aus89], SDF [HHKR92], TXL [CC93] and PCCTS [PQ94]. An overview of available syntax definition tools is provided by [MR94].

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 intended ones. Syntax definition formalisms form no exception to this rule.

1.2 A Family of Formalisms

In this paper 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 applications 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 restricted set of features that are useful in many application areas, but in particular in the application of grammars as signatures for algebraic specifications of programming languages. The result is a complete syntax definition formalism SDF_1 that can be seen as an improvement of the formalism SDF. It incorporates several concepts and techniques introduced in [HHKR92] in a more consequent way.

We use the algebraic specification formalism ASF+SDF to formally specify the family of syntax definition formalisms. The development of ASF+SDF specifications is supported by an interactive environment [Kli93] that supports execution of specifications by interpretation as term rewrite systems.

In the next section we define strings of symbols, the abstract notion of a grammar and several possible interpretations of grammars. Sections 3 through 7 define several syntax definition features. An overview of these features is given at the end of the next section. Section 8 gives an overview of the appendices. Section 9 evaluates the design and draws some conclusions from this project.

2 Symbols and Grammars

2.1 Symbols

Syntax definitions define languages, i.e. sets of strings of *symbols*. Without specifying a concrete alphabet of symbols, we declare the sort Symbol. A string of symbols is a list of zero or more symbols. The function ++ concatenates strings. We will use this operator also as concatenation operator for other list sorts.

imports Layout$^{B.1}$
exports
  sorts Symbol Symbols
  context-free syntax
    Symbol* → Symbols
    Symbols "++" Symbols → Symbols {assoc}
  variables
    [$\mathcal{ABC}$][0-9']* → Symbol
    [$\alpha\beta\gamma$][0-9']* → Symbol*
    [$\alpha\beta\gamma$]"+"[0-9']* → Symbol+

(1) $\alpha \mathbin{+\!\!+} \beta = \alpha\ \beta$

Note that 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. In examples we will use the syntax for basic symbols of SDF_1: literals between double quotes, e.g., `"aliteral"`, sorts start with a capital, e.g., `ASort`, and character classes are written as [`a`−`z`]. The definition of literals, sorts and character classes can be found in appendices G, H, and I, respectively. It is important to note that we do not commit to any specific notation for things like sorts and literals. We design the family of formalisms in such a way that we can easily switch to another notation for literals without affecting the rest of the framework.

2.2 Grammars

All syntax definition formalisms we will specify consist of grammars. 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 $\emptyset$ representing the empty grammar.

imports Layout$^{B.1}$
exports
  sorts Grammar
  context-free syntax
    "$\emptyset$" → Grammar
    Grammar Grammar → Grammar {assoc}
    "(" Grammar ")" → Grammar {bracket}
  variables
    "$\mathcal{G}$"[0-9']* → Grammar

(1) $\emptyset\ \mathcal{G} = \mathcal{G}$

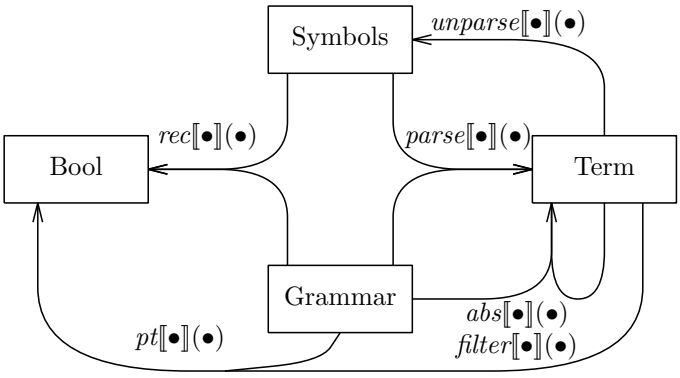


Figure 1: Grammar interpretation functions.

(2) $\mathcal{G}\ \emptyset = \mathcal{G}$
(3) $\mathcal{G}\ \mathcal{G} = \mathcal{G}$
(4) $\mathcal{G}_1\ (\mathcal{G}_2\ \mathcal{G}_3) = \mathcal{G}_1\ \mathcal{G}_2\ \mathcal{G}_3$

2.3 Interpretation

A grammar defines first and foremost a set of strings of symbols—a language. The relation between a grammar and the language it defines is specified by a *recognition* predicate on the universe of strings. Apart from this fundamental interpretation of a grammar, many other interpretations are useful. In this paper we will consider parsers, unparsers, filters, and abstraction functions.

```
imports Grammars-Syntax[2.2] Symbols[2.1] Booleans[B.2] ATerms[3.4]
exports
  context-free syntax
    rec "⟦" Grammar "⟧" "(" Symbols ")"      → BOOL
    pt "⟦" Grammar "⟧" "(" ATerm ")"         → BOOL
    yield "⟦" Grammar "⟧" "(" ATerm ")"      → Symbols
    parse "⟦" Grammar "⟧" "(" Symbols ")"    → ATerm
    filter "⟦" Grammar "⟧" "(" ATerm ")"     → ATerm
    abs "⟦" Grammar "⟧" "(" ATerm ")"        → ATerm
    unparse "⟦" Grammar "⟧" "(" ATerm ")"    → Symbols
```

Figure 1 summarizes the signature of grammar interpretation functions. (Throughout this paper we use • to indicate some unspecified argument of a function.) The sort Term covers parse trees, abstract syntax trees and sets of parse trees. We will use both 'term' and 'tree' to refer to tree-like structures. We briefly discuss the functions.

Recognition The recognition function $rec[\![\mathcal{G}]\!](\bullet)$ for a grammar $\mathcal{G}$ is a predicate on strings of symbols that characterizes the strings in the language $L(\mathcal{G})$ defined by $\mathcal{G}$.

Parse Trees The function $pt[\![\mathcal{G}]\!](\bullet)$ characterizes the terms that are correct parse trees for grammar $\mathcal{G}$.

Parsing A parser $parse[\![\mathcal{G}]\!](\bullet)$ is a function that maps strings in $L(\mathcal{G})$ to sets of parse trees. The inverse of this function is *yield* that maps parse trees to strings.

Filtering A filter function $\mathit{filter}[\![\mathcal{G}]\!](\bullet)$ selects a subset of a set of parse trees. Filters are used to model disambiguation. In [KV94] several properties and examples of filters are discussed.

Abstraction An abstraction function $\mathit{abs}[\![\mathcal{G}]\!](\bullet)$ maps parse trees to abstract syntax trees.

Unparsing An unparser $\mathit{unparse}[\![\mathcal{G}]\!](\bullet)$ maps abstract syntax trees to strings of symbols.

The following laws should be satisfied by any specification of these functions.

$$T \in \mathit{parse}[\![\mathcal{G}]\!](\alpha) \iff \mathit{pt}[\![\mathcal{G}]\!](T) \wedge \mathit{yield}[\![\mathcal{G}]\!](T) = \alpha$$

$$\mathit{rec}[\![\mathcal{G}]\!](\mathit{yield}[\![\mathcal{G}]\!](T)) \iff \mathit{pt}[\![\mathcal{G}]\!](T)$$

$$\mathit{filter}[\![\mathcal{G}]\!](\mathit{parse}[\![\mathcal{G}]\!](\alpha)) \subseteq \mathit{parse}[\![\mathcal{G}]\!](\alpha)$$

$$T \in \mathit{abs}[\![\mathcal{G}]\!](\mathit{parse}[\![\mathcal{G}]\!](\mathit{unparse}[\![\mathcal{G}]\!](T)))$$

2.4 Features

The rest of this paper is devoted to the specification of several common features for syntax definition. Figure 2 shows the import graph of the various extensions.

The specification of a feature starts with the definition of its syntax. Normalization defines how this syntax can be tranlated to a subset of the syntax. This is done either directly by equations on the syntax, e.g., the equation

$$p_1^*\ p\ p_2^*\ p\ p_3^* = p_1^*\ p\ p_2^*\ p_3^*$$

expresses that multiple occurences of a production p in a list of productions can be replaced by one occurence, or, in case such equations would lead to a non-terminating rewrite system, normalization is defined through a normalization function, e.g., the equation

$$\begin{array}{rll} r[\![\mathcal{A}?]\!] = & \texttt{syntax} & \\ & & \to \mathcal{A}? \\ & \mathcal{A} & \to \mathcal{A}? \end{array}$$

expresses that the optional construct $\mathcal{A}?$ generates two productions. For each feature all interpretation functions for recognition, parse trees, parsing, filtering, and abstract syntax are defined, in principle; a feature for which the syntax normalizes to the constructs of another formalism, can reuse the interpretation functions of that formalism. The various aspects of a feature are each defined in a separate module. Figure 3 shows a typical import graph for the cluster of modules defining a single feature.

We give an overview of the features.

Kernel SDF introduces context-free productions of the form $\alpha \to \mathcal{A}$. A kernel grammar is a set of productions. Based on this structure we define the interpretation functions for recognition, parse trees, parsing and abstract syntax trees. (Section 3)

Basic SDF provides a method to combine the grammars for lexical and context-free syntax in one grammar. A similar method is provided for the definition of syntax for variables. The interface between lexical and context-free syntax is defined by normalizing definitions to kernel grammars with the function $b[\![\bullet]\!]$. This normalization uses the new symbol constructors $\langle \bullet\texttt{-LEX} \rangle$, $\langle \bullet\texttt{-CF} \rangle$ and $\langle \bullet\texttt{-VAR} \rangle$. If $\mathcal{G}$ is a definition over Basic SDF, then a string α is parsed by $\mathit{parse}[\![b[\![\mathcal{G}]\!]]\!](\alpha)$. (Section 4)

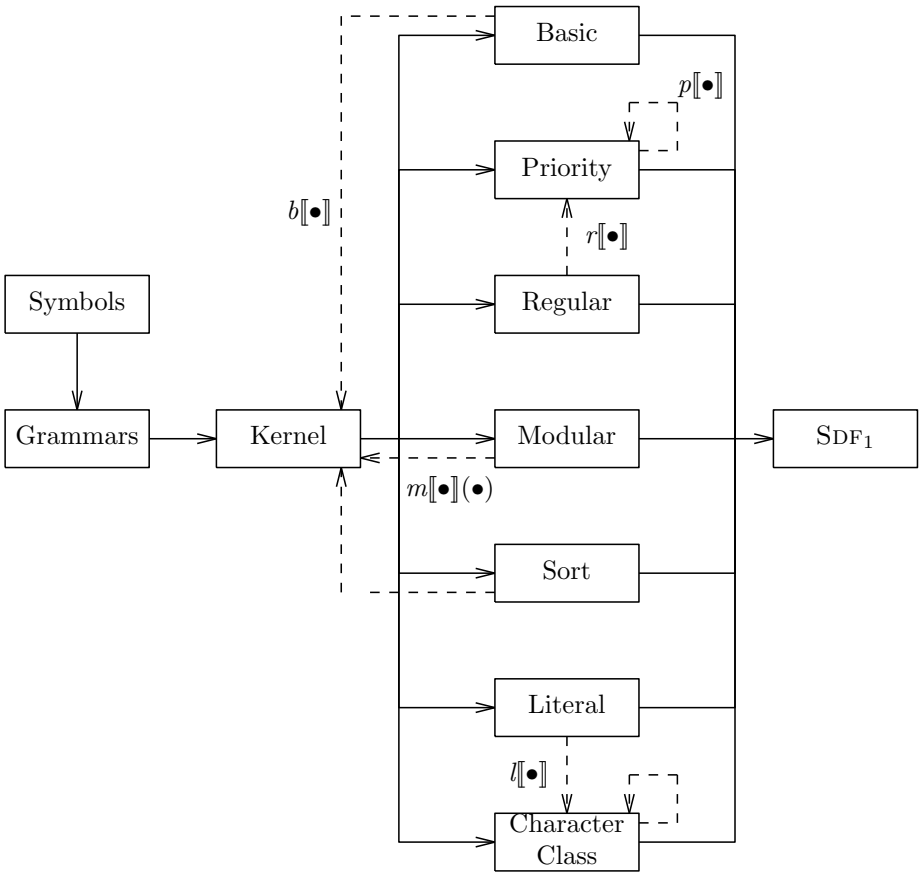


Figure 2: Import graph of a family of syntax definition formalisms. The formalism SDF_1 is the combination of all the features described in this paper. Normal arrows show the import relation. Dashed arrows show domain and range of normalization.

Priority SDF provides disambiguation by a priority and associativity relation on productions. The normalization function $p[\![\bullet]\!]$ derives productions from priority declarations and priority declarations from productions, so that no redundant declarations are needed. Syntax for abbreviation of chains of priorities is normalized by means of semantic equations. Priorities are interpreted by a filter on sets of parse trees that selects the minimal parse trees without priority or associativity conflicts. (Section 5)

Regular SDF provides abbreviations for iteration, optional constructs and alternatives by means of regular operators on symbols ($*$, $+$, ?, and $|$). The normalization function $r[\![\bullet]\!]$ adds productions for each symbol. (Section 6)

Modular SDF provides a structure of named modules on top of grammars. Modules consist of a series of exports and hiddens sections. Other modules can be imported in a module. Imports can be included either in exports or in hiddens sections. Renamings of symbols and productions can be applied to imported modules to prevent name clashes and to instantiate generic modules. The normalization function $m[\![\bullet]\!](\bullet)$ yields the grammar corresponding to a named module. (Section 7)

Literal SDF provides syntax for literals. The normalization function $l[\![\bullet]\!]$ defines

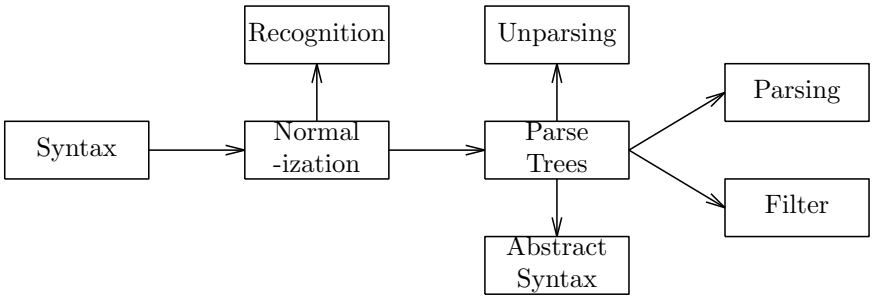


Figure 3: Import graph of a feature cluster.

literals by adding productions using character classes. (Section G)

Sort SDF provides syntax for sort names and declaration of sorts. Sorts are only used to check that no unintended symbols are used in productions. (Section H)

Character Class SDF provides a compact description of sets of characters. Normalization of character classes is expressed by semantic equations. (Section I)

SDF_1 combines all features above. A series of module declarations d is normalized by first extracting the flattened grammar for a module and then normalizing using the normalization functions of the other features, i.e., the grammar $\mathcal{G}$ corresponding to a module named M in a definition d is $\mathcal{G} = l[\![p[\![r[\![b[\![m[\![d]\!](M)]\!]]\!]]\!]]\!]$. (Section K)

3 Kernel SDF: Context-free Productions

3.1 Syntax

The basic structure of our grammars is the context-free production. A production $\alpha \to \mathcal{A}!$ is a tuple of a list of symbols α, a symbol $\mathcal{A}$, and an optional list of attributes ! (we use the exclamation mark '!' as variable for attributes). The function `syntax` maps a list of productions to a grammar. (To distinguish the object language from the meta language we write all tokens of the object language in a `typewriter` font.)

```
imports Symbols[2.1] Grammars-Syntax[2.2] Kernel-Sdf-Attributes[D.1]
exports
  sorts Production Productions
  context-free syntax
    Symbols "→" Symbol Attributes → Production
    Production*                   → Productions
    Productions "++" Productions  → Productions {right}
    "syntax" Productions          → Grammar
  variables
    [p][0-9']*      → Production
    [p]"*"[0-9']*   → Production*
    [p]"+"[0-9']*   → Production+
```

(1) $p_1^* \mathbin{+\!\!+} p_2^* = p_1^*\, p_2^*$

Conventionally, productions are written as $\mathcal{A} \to \alpha$ in context-free grammars to emphasize the generative view of grammars; they describe how to *generate* a string from a specific start symbol. The unconventional notation for productions that we

inherit from SDF emphasizes the *functional* view of productions when used in the context of algebraic specification. A production coincides with the declaration of name and type of a function. However, there is no fundamental difference with other notations for productions as in BNF, YACC [Joh75] or any of its variants. Indeed, we could define a version of SDF that uses the $\mathcal{A} ::= \alpha$ notation of BNF and define its meaning by translating to the notation used here.

3.2 Normalization

Composition of grammars consisting of sets of productions is equal to the composition of the sets. Lists of productions should be considered as sets. This entails that multiple occurences of the same production can be reduced to one occurence as is expressed by equation (4). Note that the property expressed by equation (4) is often not recognized by parser generators, resulting in reports of ambiguities.

imports Kernel-Sdf-Syntax[3.1]

(1) $\texttt{syntax} = \emptyset$

(2) $\texttt{syntax}\ p_1^*\ \texttt{syntax}\ p_2^* = \texttt{syntax}\ p_1^*\ p_2^*$

(3) $\mathcal{G}\ \texttt{syntax}\ p_1^*\ \texttt{syntax}\ p_2^* = \mathcal{G}\ \texttt{syntax}\ p_1^*\ p_2^*$

(4) $p_1^*\ p\ p_2^*\ p\ p_3^* = p_1^*\ p\ p_2^*\ p_3^*$

3.3 Recognition

The predicate $rec[\![\mathcal{G}]\!]$ characterizes the strings of the language defined by $\mathcal{G}$, i.e.,

$$L(\mathcal{G}) = \{\alpha \mid rec[\![\mathcal{G}]\!](\alpha) = \mathsf{true}\}$$

A set of productions describes a rewrite system on strings of symbols. A string α is member of the language $L(\mathcal{G})$ described by a grammar $\mathcal{G}$ if it can be rewritten to a symbol $\mathcal{A}$. Equation (4) expresses this rewrite property: a string $\alpha\beta\gamma$ is recognized if there exists a production $\beta \to \mathcal{A}$ and the string $\alpha\mathcal{A}\gamma$ is recognized.

imports Kernel-Sdf-Normalization[3.2] Booleans[B.2] Grammar-Interpretation[2.3]
exports
  context-free syntax
    rec "⟦" Productions "⟧" "(" Symbols ")" → BOOL
    rec "⟦" Productions "⟧" "(" Symbols "·" Symbols ")" → BOOL

(1) $rec[\![\texttt{syntax}\ p^*]\!](\alpha) = rec[\![p^*]\!](\alpha)$

(2) $rec[\![p^*]\!](\alpha) = rec[\![p^*]\!](\cdot\ \alpha)$

(3) $rec[\![p^*]\!](\mathcal{A}\ \cdot\ ) = \mathsf{true}$

(4) $$\frac{rec[\![p_1^*\ \beta \to \mathcal{A}\ !\ p_2^*]\!](\alpha\ \mathcal{A}\ \cdot\ \gamma) = \mathsf{true}}{rec[\![p_1^*\ \beta \to \mathcal{A}\ !\ p_2^*]\!](\alpha\ \beta\ \cdot\ \gamma) = \mathsf{true}}$$

(5) $$\frac{rec[\![p^*]\!](\alpha\ \mathcal{A}\ \cdot\ \gamma) = \mathsf{true}}{rec[\![p^*]\!](\alpha\ \cdot\ \mathcal{A}\ \gamma) = \mathsf{true}}$$

Productions are context-free because the context $\alpha\bullet\gamma$ of the string β has no influence on the rewrite step $\beta \to \mathcal{A}$. This entails that a rewrite step can take place at any point in a string. The auxiliary predicate controls the searching for substrings to be reduced by dividing the string in two parts. A substring is only reduced if it occurs

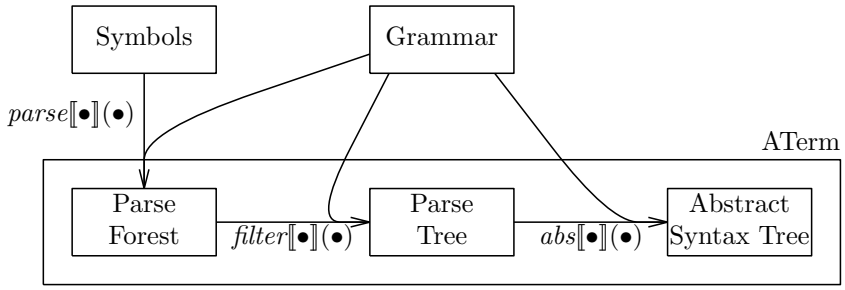


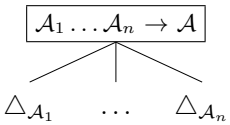
Figure 4: Signature of terms. A generic term datatype is used to model various forms of trees.

right before the dot and matches the left-hand side of a production [equation (4)]. By an application of equation (5) the dot can be shifted to the right. We could have done without the auxiliary predicate by removing the dot and equation (5). However, this techniques, known as shift-reduce backtrack recognition (see, e.g., [AU72]), cuts down the search space considerably and enables us to actually use this specification as a prototype.

Generally there are many ways to rewrite a string given a set of productions. The rewrite process can be infinite if there exist empty productions ($\rightarrow \mathcal{A}$) or cyclic productions ($\mathcal{A} \rightarrow \mathcal{B} \rightarrow \cdots \rightarrow \mathcal{A}$); empty productions even increase the size of the 'reduced' string. Although the specification has infinite reductions for cyclic or ϵ-grammars, for acyclic, ϵ-free grammars this specification is terminating and can be used as an executable prototype. It is not supposed to be efficiently executable, though. A whole array of techniques from parsing theory can be applied to optimize this specification.

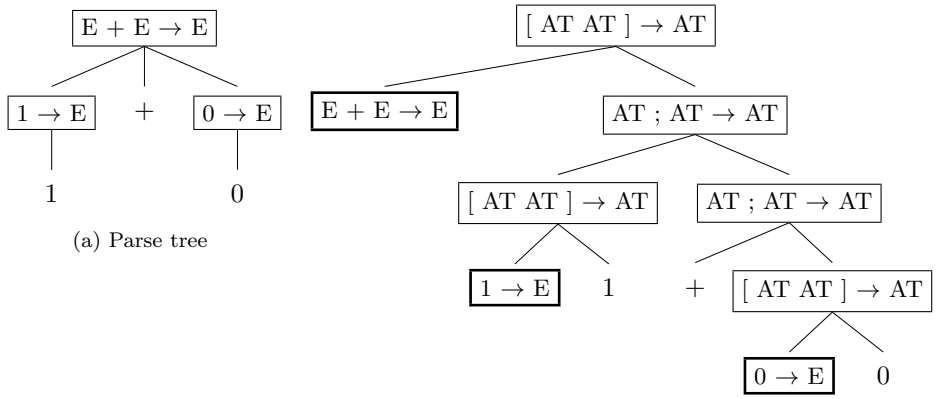
3.4 Terms

The recognition predicate in the previous section specifies which strings are generated by a set of productions. To analyze strings we want to use the structure assigned to strings by a grammar. This structure is defined in terms of trees in the following way. If for $1 \leq i \leq n$ $\triangle_{\mathcal{A}_i}$ are parse trees of type $\mathcal{A}_i$ and $\mathcal{A}_1 \ldots \mathcal{A}_n \rightarrow \mathcal{A}$ is a production, then the tree



is a parse tree of type $\mathcal{A}$. A complete example of a parse tree is shown in Figure 5(a). The language of AsFix terms [Kli94] is a generic representation for applicative, annotated terms. It was developed as a fixed representation language for Asf+Sdf. We will use this language to represent all kinds of tree-like structures we need, i.e., parse trees, parse forests (or sets of parse trees) and abstract syntax trees. Figure 4 shows how the sort of terms is divided into several classes of terms.

```
imports Literals^G.2
exports
  sorts ATerm
  context-free syntax
    Literal                  → ATerm
    "[" ATerm ATerm "]"      → ATerm
```



(a) Parse tree

(b) AsFix representation

Figure 5: Parse tree and its AsFix representation. The sort ATerm is abbreviated to AT.

```
    ATerm ";" ATerm      → ATerm  {right}
    nil                  → ATerm
    ATerm "/" ATerm      → ATerm  {left}
    "(" ATerm ")"        → ATerm  {bracket}
  variables
    T [0-9']* → ATerm
  priorities
    ATerm "/" ATerm → ATerm > ATerm ";" ATerm → ATerm
```

(1) $nil;\ T = T$

(2) $T;\ nil = T$

(3) $(T_1;\ T_2);\ T_3 = T_1;\ T_2;\ T_3$

This syntax should be interpreted as follows: $[T_1\ T_2]$ is the application of the term T_1 (a function) to the term T_2 (its argument), *nil* is the empty list, $T_1;T_2$ is concatenation of the lists or list items T_1 and T_2, T_1/T_2 is the annotation of T_1 with T_2.

3.5 Parse Trees

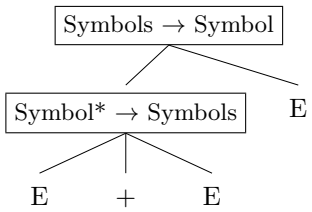
To represent parse trees as terms we extend the definition of terms in the previous section with productions and symbols.

```
imports ATerms[3.4] Kernel-Sdf-Normalization[3.2] Grammar-Interpretation[2.3]
exports
  context-free syntax
    Production → ATerm
    Symbol     → ATerm
```

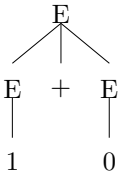
The example parse tree from the previous section can now be represented as

$$[E + E \to E\ [1 \to E\ 1]\ ;\ +\ ;\ [0 \to E\ 0]]$$

Figure 5(b) shows a pictorial representation of this term. In the official use of ATerms, productions and symbols are also encoded as terms using literal strings to indicate the primitive functions. We can easily provide equations that translate productions to such terms. However, for clarity and simplicity we use symbols and productions from our syntax definitions directly. Note that in Figure 5(b) a production like $\boxed{\text{E + E} \rightarrow \text{E}}$ actually denotes the structure



From now on we will assume term representations and parse trees to be the same and will depict them as in Figure 5(a). Conventional definitions of parse trees use symbols as labels of interior nodes. For instance, our example would look like



To check the consistency of such a tree the grammar that generated the tree is needed. With our approach this is not necessary; a parse tree is self-descriptive.

In our encoding productions are used as labels of interior nodes and symbols are used as leaves of trees. However, by declaring productions and symbols as terms, we can construct terms that are not representations of parse trees. For instance, the term $[E\ [E\ E + E \rightarrow E]]$ is a syntactically correct term but we do not want to consider it a representation of a parse tree. Therefore, we define a function *type* that characterizes the terms that we want to consider as legal parse trees by assigning a type to a term; only those terms that have a single symbol as type are legal parse trees.

```
context-free syntax
  type(ATerm) → Type
```

In this context we take symbols as basic types denoting the set of all trees that correspond to that symbol.

```
sorts  Type
context-free syntax
  Symbols    → Type
  Production → Type
```

The type assignment for terms is

(1) $type(p) = p$

(2) $type(\mathcal{A}) = \mathcal{A}$

(3) $type(nil) =$

(4) $type(T_1;\ T_2) = \alpha\ \beta$ when $type(T_1) = \alpha,\ type(T_2) = \beta$

(5) $type([T_1\ T_2]) = \mathcal{A}$ when $type(T_1) = \alpha \rightarrow \mathcal{A}\ !,\ type(T_2) = \alpha$

This assignment specifies that a tree can only be composed from symbols and productions using sequential composition $\bullet;\bullet$ and function application $[\bullet\ \bullet]$. These

constructs correspond to concatenation of list of symbols ($\alpha\beta$) and to productions ($\alpha \to \mathcal{A}$) on the type level. Since symbols can not be productions, higher-order application is excluded and the function T_1 in an application $[T_1\ T_2]$ should always be a production. Note the similarity of equation (5) with type checking in typed term construction systems like the simple theory of types [Chu40].

The function *type* checks that a parse tree is internally consistent. If we want to check that a parse tree is consistent *according to a particular grammar* we also have to check that all productions used as function in the tree also occur in the grammar. To this end the function Σ produces the productions (the signature) of a parse tree

context-free syntax
"Σ" "(" ATerm ")" $\to$ Productions

(6) $\Sigma(p) = p$
(7) $\Sigma([T_1\ T_2]) = \Sigma(T_1) +\!\!+ \Sigma(T_2)$
(8) $\Sigma(T_1;\ T_2) = \Sigma(T_1) +\!\!+ \Sigma(T_2)$
(9) $\Sigma(T) =$ otherwise

The predicate *pt* characterizes the correct parse trees over a grammar.

imports Booleans$^{B.2}$
exports
context-free syntax
pt "$[\![$" Productions "$]\!]$" "(" ATerm ")" $\to$ BOOL
Productions "$\subseteq$" Productions $\to$ BOOL

(10) $pt[\![\texttt{syntax}\ p^*]\!](T) = pt[\![p^*]\!](T)$
(11) $pt[\![p^*]\!](T) =$ true when $type(T) = \mathcal{A},\ \Sigma(T) \subseteq p^* =$ true
(12) $pt[\![p^*]\!](T) =$ false otherwise

(13) $\subseteq p^* =$ true
(14) $p\ p^* \subseteq p_1^*\ p\ p_2^* = p^* \subseteq p_1^*\ p\ p_2^*$

The *yield* of a parse tree is the concatenation of its leaves. The function is the inverse of the parse function that we define in the next section.

context-free syntax
yield(ATerm) $\to$ Symbols

(15) $yield(\mathcal{A}) = \mathcal{A}$
(16) $yield(nil) =$
(17) $yield(T_1;\ T_2) = yield(T_1) +\!\!+ yield(T_2)$
(18) $yield([T_1\ T_2]) = yield(T_2)$

3.6 Parsing

Given the definition of parse trees we can now define how parse trees are derived from strings of symbols. Below we give a definition for the function *parse* that interprets a grammar as a parser and that yields the list of all parse trees for a

string. We first declare an auxiliary function *list* that translates an associative list of terms to a term list built with the term concatentation operator $\bullet;\bullet$.

```
imports Kernel-Sdf-Trees[3.5] Grammar-Interpretation[2.3]
hiddens
  context-free syntax
    list(ATerm*) → ATerm
  variables
    "T" "*"[0-9']* → ATerm*
    "T" "+"[0-9']* → ATerm+
```

(1) $list() = nil$ (2) $list(T) = T$ (3) $list(T\ T^+) = T; list(T^+)$

A declarative specification of the parse function is:

$$parse[\![\mathcal{G}]\!](\alpha) = \begin{cases} nil & \text{when } rec[\![\mathcal{G}]\!](\alpha) = \mathsf{false} \\ T_1; \ldots; T_n & \text{when } rec[\![\mathcal{G}]\!](\alpha) = \mathsf{true}, pt[\![\mathcal{G}]\!](T_i), yield(T_i) = \alpha \end{cases}$$

More constructively, the interpretation of a grammar as a parser is defined in terms of the auxiliary functions

```
context-free syntax
  parse "[[" Productions "]]" "(" Productions ")"
  "(" ATerm* "·" Symbols ")"                          → ATerm
  parse "[[" Productions "]]" "(" ATerm* "·" Symbols ")" → ATerm
```

by

(4) $parse[\![\texttt{syntax}\ p^*]\!](\alpha) = parse[\![p^*]\!](\cdot\ \alpha)$

The functions maintain a parse configuration $T^* \cdot \beta$ consisting of an associative list of trees T^* corresponding to a prefix of the original string α and a suffix β of α. If the initial configuration is $\cdot\alpha$, then each following configuration $T_1 \ldots T_n \cdot \beta$ satisfies the property

$$pt[\![\mathcal{G}]\!](T_i) = \mathsf{true} \wedge yield(T_1) +\!\!+ \cdots +\!\!+ yield(T_n) +\!\!+ \beta = \alpha$$

The first function tries to rewrite some list of trees before the dot with some production in the set. If the type of the list of trees matches the left-hand side of a production, the list is replaced by a single tree with the list as arguments and the production as function. If the left-hand side of the production does not match any list of trees, parsing fails, i.e., returns an empty list of trees.

(5) $$\frac{list(T_2^*) = T_3,\ \ type(T_3) = \alpha}{parse[\![p^*]\!](\alpha \to \mathcal{A}\ !)(T_1^*\ T_2^* \cdot \beta) = parse[\![p^*]\!](T_1^*\ [\alpha \to \mathcal{A}\ !\ T_3] \cdot \beta)}$$

(6) $parse[\![p^*]\!](p)(T^* \cdot \alpha) = nil$ otherwise

If more than one production can be tried, they are tried in parallel, concatenating the resulting lists of trees. If no production can be tried parsing fails.

(7) $parse[\![p^*]\!](p_1^+\ p_2^+)(T^* \cdot \alpha) = parse[\![p^*]\!](p_1^+)(T^* \cdot \alpha); parse[\![p^*]\!](p_2^+)(T^* \cdot \alpha)$

(8) $parse[\![p^*]\!]()(T^* \cdot \alpha) = nil$

A configuration consisting of a single tree and an empty suffix is an accepting configuration and T is the result.

(9) $parse[\![p^*]\!](T \cdot\ ) = T$

If there is more than one tree in the configuration or the suffix is not empty, attempts are made to rewrite the configuration with the productions of the grammar and a new configuration is created by shifting a symbol from the suffix to the tree list.

(10) $parse[\![p^*]\!](T\ T^+ \cdot) = parse[\![p^*]\!](p^*)(T\ T^+ \cdot)$

(11) $parse[\![p^*]\!](T^* \cdot \mathcal{A}\ \alpha) = parse[\![p^*]\!](T^*\ \mathcal{A} \cdot \alpha);\ parse[\![p^*]\!](p^*)(T^* \cdot \mathcal{A}\ \alpha)$

This specification is a nonterminating term rewrite system if the grammar is cyclic or contains ϵ-productions. However, as a semantic description it is adequate. In the case of cycles or ϵ-productions the result of parsing is an infinite list of parse trees.

3.7 Abstract Syntax

McCarthy [McC62] introduced abstract syntax as a way to deal with programs by abstracting from concrete notation. This was achieved by functions to construct and deconstruct terms. In the initial algebra approach to semantics [GTWW77], the abstract syntax of a programming language is represented by a many sorted algebraic signature. The relation between a constructor function $f : S_1 \times \cdots \times S_n \to S$ that constructs a tree (or term) of type S from a tuple of trees of type $S_1 \times \cdots \times S_n$ and a selector function $g : S \to S_i$ that selects a single subtree from such a composition can be specified by the equation $g(f(s_1, \ldots, s_i, \ldots, s_n)) = s_i$. However, in [GTWW77] it is shown that all kinds of recursive constructions, including context-free grammars, are isomorphic to algebraic signatures. This entails that 'abstract syntax' as in [McC62] is no more abstract than context-free grammars.

A number of algebraic specification formalisms—OBJ, Aspegique, ASF+SDF—exploit this property by using signatures with distfix operators or even arbitrary context-free grammars instead of a prefix signature. A definition can be viewed as a context-free grammar or as an algebraic signature. The grammar view is used to generate parsers from a definition. The signature view describes the abstract syntax trees that are used by semantic tools. A mapping from parse trees to abstract syntax trees is used as interface between parser and semantic tool. In [HHKR92] these views are made explicit by translating an SDF definition to a contex-free grammar (BNF) and to a first-order algebraic signature and by providing a translation from parse trees to abstract syntax trees.

In this paper we do not give such translations but we directly interpret productions as grammar rules and as function declarations. This has the advantage that no external formalisms have to be defined and understood. Our definition of parse trees in this section is such that parse trees are close to abstract syntax trees. Productions are used as functions in these trees. Abstract syntax reflects our idea of syntactic equality; two strings with the same abstract syntax tree are equal. The only thing that is left to abstract is 'redundant' information that is not needed by semantic tools. Typical examples of such redundant information are the layout between tokens and the literals in a production, that are introduced in the next section and in appendix G, respectively.

Here we define the mapping from parse trees to abstract syntax trees by means of a predicate *abs* on types that characterizes the types of (sub)trees to be removed.

```
imports Kernel-Sdf-Trees[3.5]
exports
  context-free syntax
    abs "⟦" Grammar "⟧" "(" Symbol ")" → BOOL
```

The predicate *abs* determines which symbols act as sorts and which as terminals.

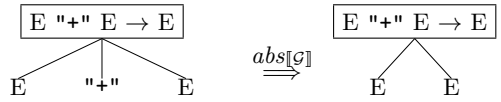
(1) $abs[\![\mathcal{G}]\!](\mathcal{A}) =$ `false` `otherwise`

Based on the *abs* predicate the function *abs* removes all subtrees from a parse tree that have a type that is not abstracted from according to the *abs* predicate.

$$(2)\quad \frac{type(T) = \mathcal{A},\ \ abs[\![\mathcal{G}]\!](\mathcal{A}) = \mathsf{true}}{abs[\![\mathcal{G}]\!](T) = nil}$$

$$\begin{array}{lllll}
(3) & abs[\![\mathcal{G}]\!](T_1;\ T_2) & = & abs[\![\mathcal{G}]\!](T_1);\ abs[\![\mathcal{G}]\!](T_2) & \mathsf{otherwise} \\
(4) & abs[\![\mathcal{G}]\!]([T_1\ T_2]) & = & [abs[\![\mathcal{G}]\!](T_2)\ abs[\![\mathcal{G}]\!](T_2)] & \mathsf{otherwise} \\
(5) & abs[\![\mathcal{G}]\!](\mathcal{A}) & = & \mathcal{A} & \mathsf{otherwise} \\
(6) & abs[\![\mathcal{G}]\!](p) & = & p & \mathsf{otherwise}
\end{array}$$

In the extension Literals SDF we introduce double quoted literals like `"+"`. By defining $abs[\![\mathcal{G}]\!](L) = \texttt{true}$, where L is a variable ranging over literals, we obtain an abstraction that has the following effect:



The abstract syntax trees that are produced by the function *abs* are characterized by the following abstract type function *atype*.

```
context-free syntax
  atype "⟦" Grammar "⟧" "(" ATerm ")" → Type
  nt "⟦" Grammar "⟧" "(" Symbols ")"  → Symbols
```

The function *nt* selects the 'sorts', i.e., the non-*abs* symbols, in a string of symbols.

$$\begin{array}{lllll}
(7) & nt[\![\mathcal{G}]\!]() & = & & \\
(8) & nt[\![\mathcal{G}]\!](\alpha^+\ \beta^+) & = & nt[\![\mathcal{G}]\!](\alpha^+) +\!\!+\ nt[\![\mathcal{G}]\!](\beta^+) & \\
(9) & nt[\![\mathcal{G}]\!](\mathcal{A}) & = & & \mathsf{when}\ \ abs[\![\mathcal{G}]\!](\mathcal{A}) = \mathsf{true} \\
(10) & nt[\![\mathcal{G}]\!](\mathcal{A}) & = & \mathcal{A} & \mathsf{when}\ \ abs[\![\mathcal{G}]\!](\mathcal{A}) = \mathsf{false}
\end{array}$$

The type $nt[\![\mathcal{G}]\!](\alpha) \to \mathcal{A}!$ is the 'abstract' type of the function $\alpha \to \mathcal{A}!$. The function *atype* returns the abstract type of a well formed abstract syntax tree.

$$(11)\quad atype[\![\mathcal{G}]\!](\mathcal{A}) = \mathcal{A}$$

$$(12)\quad atype[\![\mathcal{G}]\!](\alpha \to \mathcal{A}\ !) = nt[\![\mathcal{G}]\!](\alpha) \to \mathcal{A}\ !$$

$$(13)\quad atype[\![\mathcal{G}]\!](nil) =$$

$$(14)\quad \frac{atype[\![\mathcal{G}]\!](T_1) = \alpha \to \mathcal{A}\ ,\ \ atype[\![\mathcal{G}]\!](T_2) = \alpha}{atype[\![\mathcal{G}]\!]([T_1\ T_2]) = \mathcal{A}}$$

$$(15)\quad \frac{atype[\![\mathcal{G}]\!](T_1) = \alpha,\ \ atype[\![\mathcal{G}]\!](T_2) = \beta}{atype[\![\mathcal{G}]\!](T_1;\ T_2) = \alpha\ \beta}$$

3.8 Unparsing

An unparser maps abstract syntax trees to strings. A parse tree can be translated easily to its underlying string by concatening its leaves. For abstract syntax trees this is not so easy since some of the leaves are thrown away. Unparsing is used to display generated trees, for example in structure editing, and to 'beautify' the display of a text. Pretty printing and typesetting of programming languages is for example described in [Bra95, BV95].

Here we are not concerned with the æsthetics of unparsing, but only with (re)producing the underlying string from a tree. The function $\mathit{unparse}[\![\mathcal{G}]\!](\mathcal{A})$ produces a string for each symbol that is abstracted by the abstraction function from the previous section. The function $\mathit{unparse}[\![\mathcal{G}]\!](\alpha, T)$ produces the strings for the trees in a list T. For the abstract types it encounters in α, default strings are filled in.

```
imports Kernel-Sdf-Abstract-Syntax[3.7] Grammar-Interpretation[2.3]
exports
  context-free syntax
    unparse "[[" Grammar "]]" "(" Symbol ")"          -> Symbols
    unparse "[[" Grammar "]]" "(" Symbols "," ATerm ")" -> Symbols
```

$$\mathit{unparse}[\![\mathcal{G}]\!]([\alpha \to \mathcal{A}\ \ T]) \ = \ \mathit{unparse}[\![\mathcal{G}]\!](\alpha,\ T;\ \mathit{nil}) \tag{1}$$

$$\mathit{unparse}[\![\mathcal{G}]\!](,\ \mathit{nil}) = \tag{2}$$

$$\frac{\mathit{abs}[\![\mathcal{G}]\!](\mathcal{A}) = \mathsf{true}}{\mathit{unparse}[\![\mathcal{G}]\!](\mathcal{A}\ \alpha,\ T_1;\ T_2) = \mathit{unparse}[\![\mathcal{G}]\!](\mathcal{A}) +\!\!+ \mathit{unparse}[\![\mathcal{G}]\!](\alpha,\ T_1;\ T_2)} \tag{3}$$

$$\frac{\mathit{abs}[\![\mathcal{G}]\!](\mathcal{A}) = \mathsf{false}}{\mathit{unparse}[\![\mathcal{G}]\!](\mathcal{A}\ \alpha,\ T_1;\ T_2) = \mathit{unparse}[\![\mathcal{G}]\!](T_1) +\!\!+ \mathit{unparse}[\![\mathcal{G}]\!](\alpha,\ T_2)} \tag{4}$$

4 Basic SDF: Lexical Syntax and Variables

4.1 Syntax

The syntax of a programming language is often divided in 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. The parsing of lexical syntax is often implemented with finite automata, while the parsing of context-free syntax 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 this paper we adopt the view that the inherent distinction between the two categories is that context-free symbols can be separated by layout while lexical symbols can not. 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 the next section we define priorities for disambiguation. In [HHKR92] 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 section 6 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 specifications of semantics of languages. We also introduce the notion of lexical variables that range over constructs introduced in lexical syntax grammars.

imports Kernel-Sdf-Syntax$^{3.1}$
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

4.2 Normalization

The normalization function $b[\![\bullet]\!]$ translates all production grammars not constructed with `syntax` to such grammars. Figure 6 shows an example grammar and its normal form under application of this function.

imports Basic-Sdf-Syntax$^{4.1}$ Priority-Sdf-Syntax$^{5.1}$
exports
context-free syntax
"$b[\![$" Grammar "$]\!]$" → Grammar

(1) $b[\![\emptyset]\!] = \emptyset$
(2) $b[\![\mathcal{G}_1\ \mathcal{G}_2]\!] = b[\![\mathcal{G}_1]\!]\ b[\![\mathcal{G}_2]\!]$
(3) $b[\![\texttt{syntax}\ p^*]\!] = \texttt{syntax}\ p^*$

Lexical syntax grammars are translated to normal syntax grammars by encoding the symbols of the grammar to $\langle \mathcal{A}\texttt{-LEX} \rangle$ symbols.

context-free syntax
"<" Symbols "`-LEX`" ">" → Symbols
"<" Production "`-LEX`" ">" → Production
"<" Productions "`-LEX`" ">" → Productions
"<" Grammar "`-LEX`" ">" → Grammar
priorities
"<"Symbol "`-LEX`" ">" → Symbol > "<"Symbols "`-LEX`" ">" → Symbols,
"<"Production "`-LEX`" ">" → Production >
"<"Productions "`-LEX`" ">" → Productions

```
lexical syntax
  [\ \t\n] -> LAYOUT
  [a-z] [0-9a-z]* -> V
context-free syntax
  V       -> E
  E "+" E -> E {assoc}
variables
  [E-F] [\'0-9]* -> E
  [V] [\'0-9]*   -> V
```

(a) Original

```
syntax
  [\ \t\n]                          -> <LAYOUT-LEX>
  <LAYOUT-LEX>                      -> <LAYOUT-CF>
  <LAYOUT-LEX>                      -> LAYOUT
  [a-z] <[0-9a-z]*-LEX>             -> <V-LEX>
  <V-LEX>                           -> <V-CF>
  <<V-CF>-VAR>                      -> <V-CF>
  <V-CF>                            -> <E-CF>
  <E-CF> LAYOUT "+" LAYOUT <E-CF> -> <E-CF> {assoc}
  <<E-CF>-VAR>                      -> <E-CF>
  [E-F] <[\'0-9]*-LEX>              -> <<E-CF>-VAR>
  [V] <[\'0-9]*-LEX>                -> <<V-CF>-VAR>
priorities
   LAYOUT LAYOUT -> LAYOUT {assoc} >
                 -> LAYOUT
```

(b) Normal Form

Figure 6: A Basic SDF grammar and its normal form.

Transform all symbols in a string to $\langle$-LEX$\rangle$ symbols.

(4) $\langle \alpha\text{-LEX} \rangle = \quad$ when $\alpha =$

(5) $\langle \alpha^+\text{-LEX} \rangle = \langle \mathcal{A}\text{-LEX} \rangle \quad$ when $\alpha^+ = \mathcal{A}$

(6) $\langle \alpha^+\ \beta^+\text{-LEX} \rangle = \langle \alpha^+\text{-LEX} \rangle +\!\!+ \langle \beta^+\text{-LEX} \rangle$

A production is lexicalized by encoding left-hand side and right-hand side.

(7) $\langle \alpha \to \mathcal{A}\ !\text{-LEX} \rangle = \langle \alpha\text{-LEX} \rangle \to \langle \mathcal{A}\text{-LEX} \rangle\ !$

Translating list of productions: for each lexical production an injection from the lexical into the context-free symbol is added.

(8) $\langle p^+\text{-LEX} \rangle = \langle \alpha \to \mathcal{A}\ !\text{-LEX} \rangle\ \langle \mathcal{A}\text{-LEX} \rangle \to \langle \mathcal{A}\text{-CF} \rangle \quad$ when $p^+ = \alpha \to \mathcal{A}\ !$

(9) $\langle p^*\text{-LEX} \rangle = \quad$ when $p^* =$

(10) $\langle p_1^+\ p_2^+\text{-LEX} \rangle = \langle p_1^+\text{-LEX} \rangle +\!\!+ \langle p_2^+\text{-LEX} \rangle$

Finally, a lexical grammar is equal to a syntax grammar with lexicalized productions.

(11) $\langle \texttt{syntax}\ p^*\text{-LEX} \rangle = \texttt{syntax}\ \langle p^*\text{-LEX} \rangle$

(12) $\langle \mathcal{G}_1\ \mathcal{G}_2\text{-LEX} \rangle = \langle \mathcal{G}_1\text{-LEX} \rangle\ \langle \mathcal{G}_2\text{-LEX} \rangle$

(13) $\langle \emptyset\text{-LEX} \rangle = \emptyset$

(14) $b[\![\texttt{lexical syntax}\ p^*]\!] = \texttt{syntax}\ \langle p^*\text{-LEX} \rangle$

Context-free syntax is treated similarly to lexical syntax; all symbols in the production are mapped to $\langle \mathcal{A}$-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 LAYOUT.

context-free syntax

"<" Symbols "-CF" ">" → Symbols
"<" Production "-CF" ">" → Production
"<" Productions "-CF" ">" → Productions
"<" Grammar "-CF" ">" → Grammar
priorities
"<"Symbol "-CF"">" → Symbol > "<"Symbols "-CF"">" → Symbols,
"<"Production "-CF"">" → Production >
"<"Productions "-CF"">" → Productions

(15) $\langle \alpha\text{-CF} \rangle = \quad$ when $\alpha =$

(16) $\langle \alpha^+\text{-CF} \rangle = \langle \mathcal{A}\text{-CF} \rangle \quad$ when $\alpha^+ = \mathcal{A}$

(17) $\langle \alpha^+\ \beta^+\text{-CF} \rangle = \langle \alpha^+\text{-CF} \rangle +\!\!+ \texttt{LAYOUT} +\!\!+ \langle \beta^+\text{-CF} \rangle$

(18) $\langle \alpha \to \mathcal{A}\ !\text{-CF} \rangle = \langle \alpha\text{-CF} \rangle \to \langle \mathcal{A}\text{-CF} \rangle\ !$

(19) $\langle p^+\text{-CF} \rangle = \langle p\text{-CF} \rangle \quad$ when $p^+ = p$

(20) $\langle p^*\text{-CF} \rangle = \quad$ when $p^* =$

(21) $\langle p_1^+\ p_2^+\text{-CF} \rangle = \langle p_1^+\text{-CF} \rangle +\!\!+ \langle p_2^+\text{-CF} \rangle$

(22) $\langle \texttt{syntax}\ p^*\text{-CF} \rangle = \texttt{syntax}\ \langle p^*\text{-CF} \rangle$

(23) $\langle \mathcal{G}_1\ \mathcal{G}_2\text{-CF} \rangle = \langle \mathcal{G}_1\text{-CF} \rangle\ \langle \mathcal{G}_2\text{-CF} \rangle$

(24) $\langle \emptyset\text{-CF} \rangle = \emptyset$

Productions defining the symbol LAYOUT are added to the grammar produced for a context-free syntax grammar.

(25) $b[\![\texttt{context-free syntax}\ p^*]\!] =$
$\texttt{syntax}\ \langle p^*\text{-CF} \rangle$
$\texttt{priorities LAYOUT LAYOUT} \to \texttt{LAYOUT}\ \{\texttt{assoc}\}>$
$\to \texttt{LAYOUT}$
$\texttt{syntax}\ \langle \texttt{LAYOUT-LEX} \rangle \to \texttt{LAYOUT}$

Variables and lexical variables grammars introduce tokens that have the status of variables. The symbol constructor $\langle \mathcal{A}\text{-VAR} \rangle$ 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.

(26) $\texttt{variables} = \emptyset$

(27) $b[\![\texttt{variables}\ p_1^+\ p_2^+]\!] = b[\![\texttt{variables}\ p_1^+]\!]\ b[\![\texttt{variables}\ p_2^+]\!]$

(28) $b[\![\texttt{variables}\ \alpha \to \mathcal{A}\ !]\!] = \texttt{syntax}\ \langle \alpha\text{-LEX} \rangle \to \langle \langle \mathcal{A}\text{-CF} \rangle\text{-VAR} \rangle\ !$
$\langle \langle \mathcal{A}\text{-CF} \rangle\text{-VAR} \rangle \to \langle \mathcal{A}\text{-CF} \rangle$

(29) $\texttt{lexical variables} = \emptyset$

(30) $b[\![\texttt{lexical variables}\ p_1^+\ p_2^+]\!] = b[\![\texttt{lexical variables}\ p_1^+]\!]$
$b[\![\texttt{lexical variables}\ p_2^+]\!]$

(31) $b[\![\texttt{lexical variables}\ \alpha \to \mathcal{A}\ !]\!] = \texttt{syntax}\ \langle \alpha\text{-LEX} \rangle \to \langle \langle \mathcal{A}\text{-LEX} \rangle\text{-VAR} \rangle\ !$
$\langle \langle \mathcal{A}\text{-LEX} \rangle\text{-VAR} \rangle \to \langle \mathcal{A}\text{-LEX} \rangle$

Lexical Layout In some languages tokens can contain some kind of layout. In [HHKR92] the symbol `IGNORE` is introduced for this purpose. This can be dealt with by separating the the symbols in a lexical production by a *lexical layout* symbol just as this is done with context-free productions.

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 DFA technology for the first set of productions and by generating a parser for the second set of productions based on PDA technology. 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 a static constraint on lexical grammars.

The parser generator for SDF$_1$ described in [Vis95b] 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 techniques are used. A similar approach is described in [SC89] under the name scannerless parsing.

Lexical Disambiguation Lexical ambiguities are normally solved by rules like 'prefer longest match' or 'prefer keywords'. Such disambiguation heuristics are closely coupled to the duality in conventional implementations; lexical ambiguities have to be solved before tokens can be sent to the parser. By removing the duality, lexical ambiguities can be dealt with in the same way as context-free ambiguities. For example, in the next section we define 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.

5 Priority SDF: Disambiguation

5.1 Syntax

There are many disambiguation 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 of SDF. New with respect to the design of priorities in [HHKR92] 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 — this provides a more compact notation by avoiding multiple declarations of productions. We also give a specification of disambiguation by priorities by means of a filter on sets of parse trees.

```
imports Kernel-Sdf-Syntax3.1
exports
  sorts Associativity Group Priority Priorities
  context-free syntax
    "left"                          → Associativity
    "right"                         → Associativity
    "non-assoc"                     → Associativity
    "assoc"                         → Associativity

    Associativity                   → Attribute
```

```
  Production                                  → Group
  "{" Productions "}"                         → Group
  "{" Associativity ":" Productions "}"       → Group

  {Group ">"}+                                → Priority
  Group Associativity Group                   → Priority

  {Priority ","}*                             → Priorities
  Priorities "++" Priorities                  → Priorities      {assoc}

  "priorities" Priorities                     → Grammar
  "lexical" "priorities" Priorities           → Grammar
  "context-free" "priorities" Priorities      → Grammar
variables
  [g][0-9']*          → Group
  [g]"*"[0-9']*       → {Group ">"}*
  [g]"+"[0-9']*       → {Group ">"}+
  "pr"[0-9']*         → Priority
  "pr""*"[0-9']*      → {Priority ","}*
  "pr""+"[0-9']*      → {Priority ","}+
  "as"[0-9']*         → Associativity
```

(1) $pr_1^* \mathbin{+\!\!+} pr_2^* = pr_1^*, pr_2^*$

5.2 Normalization

The normalization of grammars with priorities consists of the following kinds of transformations: Lists of priorities with chains and groups are normalized to lists of simple priorities of the form $p_1 > p_2$ or $p_1\ as\ p_2$. Lexical and context-free priorities are translated to plain priorities, where the same techniques as in the previous section are applied. Composition of priority grammars is the same as union of the priority declarations. Finally, the normalization function p derives syntax rules from priority declarations and vice versa. We only show the specification of the function p. Figure 7 shows original and normal form of a Priority SDF definition.

```
imports Priority-Sdf-Syntax[5.1] Basic-Sdf-Normalization[4.2]
exports
  context-free syntax
    "p⟦" Grammar "⟧"          → Grammar
    assoc "(" Productions ")" → Priorities
```

The interpretation function p extracts syntax information from priorities and priority information from syntax.

(1) $p[\![\mathcal{G}_1\ \mathcal{G}_2]\!] = p[\![\mathcal{G}_1]\!]\ p[\![\mathcal{G}_2]\!]$

(2) $p[\![\texttt{priorities}\ pr_1^+, pr_2^+]\!] = p[\![\texttt{priorities}\ pr_1^+]\!]\ p[\![\texttt{priorities}\ pr_2^+]\!]$

(3) $p[\![\texttt{syntax}\ p^*]\!] = \texttt{syntax}\ p^*\ \texttt{priorities assoc}(p^*)$

(4) $p[\![\texttt{priorities}\ p_1 > p_2]\!] = \texttt{syntax}\ p_1\ p_2$
$\texttt{priorities}\ p_1 > p_2$
$\texttt{priorities assoc}(p_1\ p_2)$

(5) $p[\![\texttt{priorities}\ p_1\ as\ p_2]\!] = \texttt{syntax}\ p_1\ p_2$
$\texttt{priorities}\ p_1\ as\ p_2$
$\texttt{priorities assoc}(p_1\ p_2)$

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

(a) Original

```
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}
```

(b) Normal Form

Figure 7: A Priority SDF grammar and its normal form.

(6) $p[\![\texttt{priorities}\ pr^*]\!] = p[\![\texttt{priorities}\ [\![pr^*]\!]]\!]$ otherwise

(7) $p[\![\texttt{context-free priorities}\ pr^*]\!] = p[\![\texttt{priorities}\ \langle pr^*\texttt{-CF}\rangle]\!]$

(8) $p[\![\texttt{lexical priorities}\ pr^*]\!] = p[\![\texttt{priorities}\ \langle pr^*\texttt{-LEX}\rangle]\!]$

(9) $p[\![\mathcal{G}]\!] = \mathcal{G}$ otherwise

where the function *assoc* is defined as

(10) $\texttt{assoc}() =$

(11) $\texttt{assoc}(p_1^+\ p_2^+) = \texttt{assoc}(p_1^+) +\!\!+ \texttt{assoc}(p_2^+)$

(12) $\texttt{assoc}(p) = p\ as\ p$ when $p = \alpha \to \mathcal{A}\ \{attr_1^*,\ as,\ attr_2^*\}$

(13) $\texttt{assoc}(p) =$ otherwise

5.3 Filter

A priority declaration is interpreted as a filter on a set of parse trees.

imports Priority-Sdf-Syntax[5.1] Priority-Sdf-Operations[F.1] Kernel-Sdf-Trees[3.5]
exports
  context-free syntax
    cnf "[\[" Priorities "]\]" "(" ATerm ")" → BOOL
    filter "[\[" Priorities "]\]" "(" ATerm ")" → ATerm

The priority filter selects those trees from a list of trees that have no priority conflict.

(1) $filter[\![pr^*]\!](T_1;\ T_2) = filter[\![pr^*]\!](T_1);\ filter[\![pr^*]\!](T_2)$

(2) $$\frac{cnf[\![pr^*]\!]([T_1\ T_2]) = \mathsf{false}}{filter[\![pr^*]\!]([T_1\ T_2]) = [T_1\ T_2]}$$

(3) $filter[\![pr^*]\!](T) = nil$ otherwise

A tree has a conflict if it has a root conflict at one of its nodes.

$$
\begin{array}{lll}
(4) & \mathit{cnf}[\![pr^*]\!]([p\ T]) & = \mathit{r\text{-}cnf}[\![pr^*]\!]([p\ T]) \text{ or } \mathit{cnf}[\![pr^*]\!](T) \\
(5) & \mathit{cnf}[\![pr^*]\!](T_1;\ T_2) & = \mathit{cnf}[\![pr^*]\!](T_1) \text{ or } \mathit{cnf}[\![pr^*]\!](T_2) \\
(6) & \mathit{cnf}[\![pr^*]\!](\mathit{nil}) & = \mathsf{false} \\
(7) & \mathit{cnf}[\![pr^*]\!](\mathcal{A}) & = \mathsf{false}
\end{array}
$$

```
hiddens
  sorts ATerms
  context-free syntax
    ATerm*                                  → ATerms
    "σ" "(" ATerm ")"                       → ATerms
    r-cnf "⟦" Priorities "⟧" "(" ATerm ")"  → BOOL
  variables
    [T]"*"[0-9']* → ATerm*
```

The function σ maps a list of argument trees to a list of the function labels of those trees. It looks through injection functions.

$$
\begin{array}{lll}
(8) & \sigma([\mathcal{B} \to \mathcal{A}\ \ T_2]) & = \sigma(T_2) \\
(9) & \sigma([p\ T_2]) & = p \qquad \mathsf{otherwise} \\
(10) & \sigma(\mathcal{A}) & = \mathcal{A} \\
(11) & \sigma(T_1;\ T_2) & = T_1^*\ T_2^* \quad \mathsf{when}\ \ T_1^* = \sigma(T_1),\ \ T_2^* = \sigma(T_2)
\end{array}
$$

A tree with function label p has a root conflict if one its descendants has a priority conflict with respect to p. A left child may not be right- or non-associative with respect to its parent. Likewise a right child may not be left- or non-associative to its parent. Furthermore, associative functions are treated as if left associative. None of the descendants may have lower priority than p.

$$
(12)\quad \frac{\sigma(T) = p_2\ T^*,\ \ p_1\ \texttt{right}\ p_2 \in pr^* \text{ or } p_1\ \texttt{non-assoc}\ p_2 \in pr^* = \mathsf{true}}{\mathit{r\text{-}cnf}[\![pr^*]\!]([p_1\ T]) = \mathsf{true}}
$$

$$
(13)\quad \frac{\sigma(T) = T_1^*\ p_2\ T_2^*,\ \ p_1 > p_2 \in pr^* = \mathsf{true}}{\mathit{r\text{-}cnf}[\![pr^*]\!]([p_1\ T]) = \mathsf{true}}
$$

$$
(14)\quad \frac{\begin{array}{c}\sigma(T) = T^*\ p_2, \\ p_1\ \texttt{left}\ p_2 \in pr^* \text{ or } p_1\ \texttt{assoc}\ p_2 \in pr^* \text{ or } p_1\ \texttt{non-assoc}\ p_2 \in pr^* = \mathsf{true}\end{array}}{\mathit{r\text{-}cnf}[\![pr^*]\!]([p_1\ T]) = \mathsf{true}}
$$

$$
(15)\quad \mathit{r\text{-}cnf}[\![pr^*]\!](T) = \mathsf{false} \qquad \mathsf{otherwise}
$$

The predicate $pr \in pr^*$ for tests whether a priority rule is contained in a set of priority rules (see appendix F.1). If the descendant is a symbol no conflict arises.

Multiset Order After disambiguation by the conflict filter a second filter is applied. This filter selects the smallest tree with respect to the multiset order on parse trees induced by the priority relation. See [KV94] for a definition of this filter.

Implementation The specification of a disambiguation method by a filter that selects the intended trees after parsing forces independence of disambiguation from parsing. However, application of disambiguation rules during might lead to a considerable efficiency increase. An efficient parser-generation time implementation of disambiguation by priorities is described in [Vis95a]

Unparsing Unparsing is complicated in the presence of priorities. Naievely translating an abstract syntax tree to a string as described before might lead to a string that, when parsed and filtered, does not represent the same tree. To force equivalence of tree and string, brackets are used. In [BV95] the rules for priority conflicts are used to place brackets when unparsing an abstract syntax tree.

Other Disambiguation Methods Disambiguation by priorities as defined in this section originally defined in [HHKR92]. Disambiguation by priority conflicts is similar to the methods using precedences as in [Ear75, AJU75]. In [KV94] several disambiguation methods are defined using filters.

Subtree exclusion [Tho94] is a disambiguation method 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 wrong.

6 Regular SDF: Regular Expressions

6.1 Syntax

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 $\mathcal{A}$'s can be specified by the productions $\mathcal{A} \rightarrow \mathcal{L}A$ and $\mathcal{L}A\ \mathcal{A} \rightarrow \mathcal{L}A$, where the auxiliary symbol $\mathcal{L}A$ stands for 'list of $\mathcal{A}$'s'.

Many formalisms provide shortcuts for these patterns by extending the language of context-free grammars with some collection of regular operators on symbols. For instance, BNF provides alternative at the level of productions, i.e. a production has the form $\mathcal{A} := \mathcal{A}_0 | \ldots | \mathcal{A}_n$, where | has the meaning of *or*. Extended BNF (EBNF) is the canonical extension of BNF with regular operators. In one formulation, Wirth [Wir77] adds the operators $\{\mathcal{A}\}$ for iteration and $[\mathcal{A}]$ for optionality. Variations on this notation appear in [Lee72, Wil82]. SDF provides Kleene 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 everywhere a normal symbol can occur, also 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 some integer to each integer in a list of integers. In the syntax below we can write this as `"add" "(" Int "," Int* ")"` $\rightarrow$ `Int*`, whereas in SDF we should introduce an auxiliary sort `IntList` to represent the result sort of this function. In the design of SDF this was motivated by the cost of implementation of associative matching against list matching.

We consider the following operators.

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

(a) Original

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

(b) Normal Form

Figure 8: A Regular SDF grammar and its normal form.

```
imports Kernel-Sdf-Syntax[3.1]
exports
  context-free syntax
    "(" Symbols ")"                 -> Symbol
    Symbol "|" Symbol               -> Symbol  {right}
    Symbol "?"                      -> Symbol
    Symbol "*"                      -> Symbol
    Symbol "+"                      -> Symbol
    "{" Symbol Symbol "}" "*"       -> Symbol
    "{" Symbol Symbol "}" "+"       -> Symbol
    "(|" Symbol "|)"                -> Symbol  {bracket}
  priorities
    {Symbol "?" -> Symbol,                          Symbol "*" -> Symbol,
    Symbol "+" -> Symbol} > Symbol "|"Symbol -> Symbol
```

Informally, the operators have the following meaning: (α) represents concatenation, $\mathcal{A}|\mathcal{B}$ is either $\mathcal{A}$ or $\mathcal{B}$, $\mathcal{A}?$ is an optional $\mathcal{A}$, $\mathcal{A}*$ ($\mathcal{A}+$) is a list of zero (one) or more $\mathcal{A}$'s, and $\{\mathcal{A}\ \mathcal{B}\}*$ ($\{\mathcal{A}\ \mathcal{B}\}+$) is a list of zero (one) or more $\mathcal{A}$'s separated by $\mathcal{B}$'s.

6.2 Normalization

We define a normalization function $r[\![\bullet]\!]$ that introduces for each regular expression that appears in some production one or more productions that define its meaning. In this interpretation regular expressions form a shorthand for defining extra symbols. Figure 8 shows an example grammar with regular expressions and its normal form.

```
imports Regular-Sdf-Syntax[6.1] Priority-Sdf-Syntax[5.1]
        Kernel-Sdf-Normalization[3.2] Kernel-Sdf-Symbols[D.2]
exports
  context-free syntax
    "r[[" Grammar "]]" -> Grammar
    "r[[" Symbols "]]" -> Grammar
```

The function $r[\![\bullet]\!]$ adds defining productions for each regular expresssion occurring in one of the productions of the grammar. Existing productions are not changed.

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

The function *symbols* defined in module Kernel-SDF-Symbols gives the set of all symbols in a grammar.

(2) $r[\![]\!] = \emptyset$

(3) $r[\![\alpha^+\ \beta^+]\!] = r[\![\alpha^+]\!]\ r[\![\beta^+]\!]$

The regular expression (α) is a *symbol* that abbreviates *concatentation*.

(4) $r[\![(\alpha)]\!] = r[\![\alpha]\!]$
$\quad$ `syntax` $\alpha \rightarrow (\alpha)$

An example of the use is the symbol (`Decl` ";") in Figure 8. Note that $r[\![\alpha]\!]$ recursively produces the productions for regular expressions in the list of symbols α. As an exception we define the concatation of a single symbol to be equal to that symbol.

(5) $(\mathcal{A}) = \mathcal{A}$

This permits the use of $(\bullet)$ as bracket symbols without considering it as a new symbol.

The *alternative* $\mathcal{A}|\mathcal{B}$ denotes either $\mathcal{A}$ or $\mathcal{B}$. We could thus define $r[\![\mathcal{A}|\mathcal{B}]\!]$ to yield the productions $\mathcal{A} \rightarrow \mathcal{A}|\mathcal{B}$ and $\mathcal{B} \rightarrow \mathcal{A}|\mathcal{B}$. However, if one of the alternatives is again an alternative, an unnecessary chain like

$$\mathcal{A} \rightarrow \mathcal{A}|\mathcal{B} \rightarrow \mathcal{A}|\mathcal{B}|\mathcal{C}$$

is created. To avoid such chains we define

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

where the function *alt* unpacks the alternative until a regular expression is reached that is not an alternative.

```
hiddens
  context-free syntax
    "alt[[" Symbol "," Symbol "]]" → Grammar
```

(7) $\mathit{alt}[\![\mathcal{A}, \mathcal{B}_1 \mid \mathcal{B}_2]\!] = \mathit{alt}[\![\mathcal{A}, \mathcal{B}_1]\!]\ \mathit{alt}[\![\mathcal{A}, \mathcal{B}_2]\!]$

(8) $\mathit{alt}[\![\mathcal{A}, \mathcal{B}]\!] = r[\![\mathcal{B}]\!]$ `syntax` $\mathcal{B} \rightarrow \mathcal{A}$ otherwise

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

(9) $r[\![\mathcal{A}?]\!] = r[\![\mathcal{A}]\!]$
$\quad$ `syntax` $\rightarrow \mathcal{A}?$
$\quad\quad \mathcal{A} \rightarrow \mathcal{A}?$

The *iteration* operator $\mathcal{A}+$ denotes a list 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 .... There are several ways to define such lists with productions. We use an injection of $\mathcal{A}$ into $\mathcal{A}+$ and an associative concatenation operator of $\mathcal{A}+$'s.

(10) $r[\![\mathcal{A}+]\!] = r[\![\mathcal{A}]\!]$
$\quad$ `syntax` $\mathcal{A} \rightarrow \mathcal{A}+$
$\quad\quad \mathcal{A}+\ \mathcal{A}+ \rightarrow \mathcal{A}+$ {`assoc`}

We can not use the more common solution $\mathcal{A}+\ \mathcal{A} \rightarrow \mathcal{A}+$ because productions $\gamma \rightarrow \mathcal{A}+$, other than the ones added here, might exist; this requires that $\mathcal{A}+\ \gamma$ be

recognized as $\mathcal{A}+$. Furthermore, this syntax gives a more 'abstract' syntax; lists are built from singleton lists by concatenation.

An iteration $\mathcal{A}*$ denotes a list of *zero or more* $\mathcal{A}$'s, i.e., ϵ (empty) or $\mathcal{A}$ or $\mathcal{A}\ \mathcal{A}$ or $\mathcal{A}\ \mathcal{A}\ \mathcal{A}$ or .... Thus analogously to + iteration we could introduce the productions

$$\begin{aligned}
&\rightarrow \mathcal{A}* \\
&\mathcal{A} \rightarrow \mathcal{A}* \\
&\mathcal{A} *\ \ \mathcal{A}* \rightarrow \mathcal{A} * \{\texttt{assoc}\}
\end{aligned}$$

However, this is not adequate if the grammar also introduces the symbol $\mathcal{A}+$, because $\mathcal{A}+$ is included in $\mathcal{A}*$. In particular if we have some function, other than the ones above, that produces an $\mathcal{A}+$ we want to include it in or combine it with an $\mathcal{A}*$, which would be impossible with the productions given above. Therefore we define $\mathcal{A}*$ as an extension to $\mathcal{A}+$. We use priorities to disambiguate the various concatenation operators.

$$\begin{aligned}
(11)\quad & r[\![\mathcal{A}*]\!] = r[\![\mathcal{A}+]\!] \\
&\quad \texttt{priorities}\ \mathcal{A}+ \rightarrow \mathcal{A}* > \\
&\qquad\qquad \mathcal{A}+\ \mathcal{A}+ \rightarrow \mathcal{A}+\ \{\texttt{assoc}\}> \\
&\qquad\qquad \{\texttt{assoc} : \mathcal{A}+\ \mathcal{A}* \rightarrow \mathcal{A}+ \\
&\qquad\qquad\qquad\quad \mathcal{A}*\ \mathcal{A}+ \rightarrow \mathcal{A}+\ \}> \\
&\qquad\qquad \mathcal{A}*\ \mathcal{A}* \rightarrow \mathcal{A}*\ \{\texttt{assoc}\}> \\
&\qquad\qquad \rightarrow \mathcal{A}*
\end{aligned}$$

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 $\mathcal{A}*$.

$$\begin{aligned}
(12)\quad & r[\![\{\mathcal{A}\ \mathcal{B}\}+]\!] = r[\![\mathcal{A}]\!]\ r[\![\mathcal{B}]\!] \\
&\quad \texttt{syntax}\ \mathcal{A} \rightarrow \{\mathcal{A}\ \mathcal{B}\}+ \\
&\qquad\qquad \{\mathcal{A}\ \mathcal{B}\}+\ \mathcal{B}\ \{\mathcal{A}\ \mathcal{B}\}+ \rightarrow \{\mathcal{A}\ \mathcal{B}\}+\ \{\texttt{assoc}\}
\end{aligned}$$

$$\begin{aligned}
(13)\quad & r[\![\{\mathcal{A}\ \mathcal{B}\}*]\!] = r[\![\{\mathcal{A}\ \mathcal{B}\}+]\!] \\
&\quad \texttt{priorities}\ \{\mathcal{A}\ \mathcal{B}\}+ \rightarrow \{\mathcal{A}\ \mathcal{B}\}* > \\
&\qquad\qquad \{\mathcal{A}\ \mathcal{B}\}+\ \mathcal{B}\ \{\mathcal{A}\ \mathcal{B}\}+ \rightarrow \{\mathcal{A}\ \mathcal{B}\}+\ \{\texttt{assoc}\}> \\
&\qquad\qquad \{\texttt{assoc} : \{\mathcal{A}\ \mathcal{B}\}+\ \mathcal{B}\ \{\mathcal{A}\ \mathcal{B}\}* \rightarrow \{\mathcal{A}\ \mathcal{B}\}+ \\
&\qquad\qquad\qquad\quad \{\mathcal{A}\ \mathcal{B}\}*\ \mathcal{B}\ \{\mathcal{A}\ \mathcal{B}\}+ \rightarrow \{\mathcal{A}\ \mathcal{B}\}+\ \}> \\
&\qquad\qquad \{\mathcal{A}\ \mathcal{B}\}*\ \mathcal{B}\ \{\mathcal{A}\ \mathcal{B}\}* \rightarrow \{\mathcal{A}\ \mathcal{B}\}*\ \{\texttt{assoc}\}> \\
&\qquad\qquad \rightarrow \{\mathcal{A}\ \mathcal{B}\}*
\end{aligned}$$

Direct Interpretation Instead of defining the interpretation of regular expressions in terms of context-free productions, we could also adapt the recognition and parsing functions to account for regular expressions. The definitions of the regular operators as shown here would then be built in the rules of the interpretation functions. This has the advantage that the grammar does not explode. But this is almost no advantage because instead the specification of the interpretation functions explodes; every tool has to be aware of regular operators while the complexity or efficiency of interpretation is not reduced.

6.3 Abstract Syntax

Through the normalization of the previous section we have defined regular operators in terms of normal context-free productions. Normalized grammars use regular expressions only as generators of symbol names, no extra meaning is attached to them anymore. Therefore, all the interpretation functions that we defined for context-free

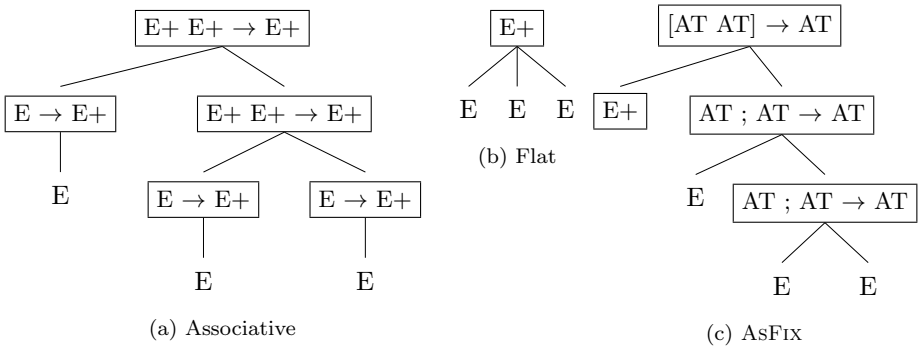


Figure 9: Various abstract syntax representations for lists.

productions are also applicable to the normalized Regular Sdf grammars, including parse and abstract syntax trees.

The only problem lies in matching terms. The associativity of the concatenation operators implies that matching of terms with constructs built over productions added here should be done modulo associativity. Furthermore, variables over the sort $\mathcal{A}*$ that are instantiated with an empty list should be removed from the list. This means that matching should be done modulo the following equations:

$$
\begin{array}{rcl}
(a_1^+\ a_2^+)\ a_3^+ & = & a_1^+\ (a_2^+\ a_3^+) \\
(a_1^+\ a_2^*)\ a_3^+ & = & a_1^+\ (a_2^*\ a_3^+) \\
(a_1^*\ a_2^+)\ a_3^+ & = & a_1^*\ (a_2^+\ a_3^+) \\
(a_1^*\ a_2^*)\ a_3^+ & = & a_1^*\ (a_2^*\ a_3^+) \\
(a_1^+\ a_2^+)\ a_3^* & = & a_1^+\ (a_2^+\ a_3^*) \\
(a_1^+\ a_2^*)\ a_3^* & = & a_1^+\ (a_2^*\ a_3^*) \\
(a_1^*\ a_2^+)\ a_3^* & = & a_1^*\ (a_2^+\ a_3^*) \\
(a_1^*\ a_2^*)\ a_3^* & = & a_1^*\ (a_2^*\ a_3^*)
\end{array}
\qquad
\begin{array}{rcl}
\epsilon_{\mathcal{A}*}\ a^+ & = & a^+ \\
a^+\ \epsilon_{\mathcal{A}*} & = & a^+ \\
\epsilon_{\mathcal{A}*}\ a^* & = & a^* \\
a^*\ \epsilon_{\mathcal{A}*} & = & a^*
\end{array}
$$

Where a variable a^+ (a^*) ranges over all constructs of type $\mathcal{A}+$ ($\mathcal{A}*$) and $\epsilon_{\mathcal{A}*}$ denotes the term $[\rightarrow \mathcal{A} *\ \mathit{nil}]$.

An alternative abstract syntax mapping is possible that interprets iterators as list constructors, reusing the list constructor of the underlying tree datatype instead of representing lists by concatenation operators. Figure 9 shows three representations of abstract syntax trees for the symbol $\mathcal{A}+$. The flat representation in Figure 9(b) is the conceptual representation of an abstract syntax tree. Figure 9(a) shows the representation for this tree with associative concatenation operators and Figure 9(c) shows its AsFix representation. This picture shows that there is not a great difference between one or the other representation, as long as associativity is regarded that enables one to apply the conceptual *flat* image. However, for the representation in Figure 9(c) an alternative typing scheme is needed that treats list symbols like $\mathcal{A}+$ as functions with input type $\mathcal{A} \ldots \mathcal{A}$, for an arbitrary number of $\mathcal{A}$'s and with output type $\mathcal{A}+$. This is a considerable complication, that permeates to all other interpretation functions that deal with parse or abstract syntax trees. Therefore, we keep to the associative operators defined by the derived productions.

Since the general case of rewriting modulo associativity is expensive, the current Asf+Sdf system only supports list matching [Hen91, Eke92]. This approach can still be used for equations that defines functions over lists, because the entire list is covered by the pattern and thus a flat representation can be used.

6.4 Generalization

The extension described in this section, actually introduces a set of new type constructors that are associated with characteristic functions. For instance, the operator + constructs a new type $\mathcal{A}+$ from an arbitrary type $\mathcal{A}$ and associated with this new type are the functions $\mathcal{A} \rightarrow \mathcal{A}+$ and $\mathcal{A}+\ \mathcal{A}+ \rightarrow \mathcal{A}+$. This approach can be applied to any number of other types. For example, we might introduce the operator $\mathcal{A}\#\mathcal{B}$ for the product of $\mathcal{A}$ and $\mathcal{B}$ and define it by extending the r function by

$$r[\![\mathcal{A}\#\mathcal{B}]\!] = \texttt{"<"}\mathcal{A}\texttt{","}\mathcal{B}\texttt{">"} \rightarrow \mathcal{A}\#\mathcal{B}$$

and use it as

```
context-free syntax
  "eval" Exp # Env -> Val # Env
```

The approach can be generalized by including the extension mechanism—implied rules for certain symbol constructors—in the formalism itself. This can be achieved by introducing variables that range over symbols. The grammar schemas that are now embedded in the definition of the function $r[\![\bullet]\!]$ can be written directly in a grammar. For instance, if we use `'A` as variable ranging over symbols we could define

```
context-free syntax
  'A      -> 'A+
  'A+ 'A+ -> 'A+ {assoc}
```

to express the semantics of $\bullet+$.

Another example of a constructor that might be added is the function type $(\bullet \Rightarrow \bullet)$. An example of its use is in the following grammar:

```
context-free syntax
  ('A => 'B) 'A -> 'B
  "sqr"         -> (Int => Int)
  "map"         -> (('A => B) => ('A* => 'B*))
```

With this syntax `"map" "sqr"` is a correct string with type `(Int* => Int*)`. Such a type system is the basis of combinatory algebraic specification [Vis93]. Note that this $\Rightarrow$ operator is similar to the / of the Lambek calculus [Lam58].

In addition to symbol variables we need to extend the formalism with a way to introduce new symbol constructors. This leads to the grammatical analogue of a leveled algebraic signature [Mei91], i.e., a leveled grammar formalism—*type grammars* in [Vis95c]. This formalism resembles definite clause grammars [PW80]. However, for the restricted use that we sketch here the formalism might be somewhat weaker such that implementation is feasible.

7 Modular SDF: Modules and Renamings

7.1 Syntax

Modularity constructs are provided by many specification and programming languages to break programs down in manageable parts and to make such parts reusable in many programs. In this section we define simple notions of modularity for syntax definition.

Seen from the perspective of modularization, syntax definition is not different from other formalisms. Maybe due to this fact, few modular syntax definition formalisms exist. An example is SDF as implemented in the ASF+SDF meta-environment

[Kli93]; the modularization is a subset of ASF's modularization. However, attaching a name to some entity and later referring to it by that name is not unique for modularization either, e.g., replacing an import with the body of the corresponding module is similar to function application or to rewriting in context-free grammars.

New with respect to the modularization constructs as provided by ASF [BHK89] and ASF+SDF are hidden imports. We define renamings of sorts and productions. Although renamings are part of ASF, they were never incorporated in SDF. Hendriks [Hen91] describes both a textual normalization semantics and an incremental semantics for modular constructs without renamings and hidden imports.

We deviate from ASF in that we do not incorporate the 'origin rule' that forbids identification of names that originate from different modules. '[T]he origin rule enforces a certain modularization of ASF specifications' [BHK89]. This style forbids to have two modules with partly overlapping signatures, e.g. both introducing the same sort, that are imported in the same module, even if the overlap is intentional.

We feel that these deviations are improvements of the modularization of ASF and ASF+SDF. They reflect our experience with the modularity constructs of ASF+SDF.

A modular syntax definition consists of a series of named *module* declarations. A module declaration consists of a list of section which can be grammars.

```
imports Kernel-Sdf-Syntax[3.1]
exports
  sorts ModuleName Section Sections Module Definition
  lexical syntax
    [A-Za-z][A-Za-z0-9\-]* → ModuleName
  context-free syntax
    Grammar                          → Section
    Section*                         → Sections
    "module" ModuleName Sections     → Module
    Module*                          → Definition
  variables
    [MN][0-9']*     → ModuleName
    "s"[0-9']*      → Section
    "s""*"[0-9']*   → Section*
    "s""+"[0-9']*   → Section+
    [m][0-9']*      → Module
    [m]"*"[0-9']*   → Module*
    "d"[0-9']*      → Definition
```

A module can *import* any numbers of other modules. An import of a module M denotes the grammar declared in module M.

```
  sorts Import Imports
  context-free syntax
    "imports" Imports → Grammar
    Import*           → Imports
    ModuleName        → Import
    "(" Import ")"    → Import     {bracket}
  variables
    "i"[0-9']*      → Import
    "i""*"[0-9']*   → Import*
    "i""+"[0-9']*   → Import+
```

Renamings enable the adaptation of a generic module to specific needs by renaming sorts and productions. Syntax and semantics of renamings are defined in appendix J. A renaming is either a sort 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. The application of a renaming ρ to some construct is written $(\bullet)\rho$.

```
imports Kernel-Sdf-Renamings[J.1]
exports
  context-free syntax
    Import Renamings → Import
  priorities
    Import Renamings → Import > Renamings Renamings → Renamings
```

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.

```
exports
  context-free syntax
    "exports" Grammar → Section
    "hiddens" Grammar → Section
```

We do not provide a parameter mechanism since we believe that renamings make parameters redundant. Modularization in this way is not enough; to structure large specifications we need some method to define clusters of modules. It might also be useful to have named submodules: named export or hiddens sections can be referenced in imports to obtain a selective import of a part of a module.

Semantics Context-free grammars do not have a compositional semantics, i.e., $L(\mathcal{G}1) \cup L(\mathcal{G}_2) \subseteq L(\mathcal{G}_1\mathcal{G}_2)$, where the inclusion is possibly strict. In terms of the recognition predicate

$$rec[\![p_1^* p_2^*]\!](\alpha) \not\Rightarrow rec[\![p_1^*]\!](\alpha) \vee rec[\![p_2^*]\!](\alpha)$$

This implies that parsers can not be composed from parsers for smaller languages/grammars. All kinds of properties like LR(k) etc. are not closed under union. Hendriks [Hen91] defines a compositional semantics for context-free grammars that adds all derived productions to a grammar. However, the composition of two such closed grammars needs still to be closed under that operation to account for the interference between the two grammars. Unfortunately, no matter what semantics is taken for context-free grammars, it will not express modularity in an appropriate way, i.e., without reconsidering all information in the grammars that are composed. There might be some condition on grammars that allows a compositional semantics; this condition forbids interference between the grammars and will probably yield a formalism that is too constrained.

7.2 Normalization

We define the function $m[\![d]\!](M)$ that yields the grammar corresponding to module M in the definition d. Figure 10 shows an example Modular SDF grammar and its normal form.

```
imports Modular-Sdf-Syntax[7.1] Modular-Sdf-Renamings[J.4]
```

Renaming of a renamed import is equal to renaming the import with the composition of the renamings.

(1) $\mathsf{i}\ \rho_1\ \rho_2\ =\ \mathsf{i}\ (\rho_1\ \rho_2)$

```
module Tables
  exports
    sorts Key Val Table
    syntax
      "[" {(Key "->" Val) ","}* "]" -> Table
      "lookup" Key "in" Table       -> Val

module Symbol-Tables
  exports
    imports
      Tables
        ["lookup" Key "in" Table -> Val
               => "type" "of" Symbol "in" SymbolTable -> Type
         Key   => Symbol
         Val   => Type
         Table => SymbolTable]
```

(a) Original

```
sorts Symbol Type SymbolTable
syntax
  "[" {(Symbol "->" Type) ","}* "]"   -> SymbolTable
  "type" "of" Symbol "in" SymbolTable -> Type
```

(b) Normal Form

Figure 10: A Modular SDF grammar and its normal form.

Normalization of order of grammars

(2) $\texttt{imports} = \emptyset$

(3) $\texttt{imports}\ \mathsf{i}_1^*\ \texttt{imports}\ \mathsf{i}_2^* = \texttt{imports}\ \mathsf{i}_1^*\ \mathsf{i}_2^*$

(4) $\mathcal{G}\ \texttt{imports}\ \mathsf{i}_1^*\ \texttt{imports}\ \mathsf{i}_2^* = \texttt{imports}\ \mathsf{i}_1^*\ \mathsf{i}_2^*\ \mathcal{G}$

(5) $\mathcal{G}\ \texttt{imports}\ \mathsf{i}^* = \texttt{imports}\ \mathsf{i}^*\ \mathcal{G}$ otherwise

Normalization of module sections

(6) $s_1^*\ \texttt{exports}\ \mathcal{G}_1\ \texttt{exports}\ \mathcal{G}_2\ s_2^* = s_1^*\ \texttt{exports}\ \mathcal{G}_1\ \mathcal{G}_2\ s_2^*$

(7) $s_1^*\ \texttt{hiddens}\ \mathcal{G}_1\ \texttt{hiddens}\ \mathcal{G}_2\ s_2^* = s_1^*\ \texttt{hiddens}\ \mathcal{G}_1\ \mathcal{G}_2\ s_2^*$

(8) $s_1^*\ \texttt{hiddens}\ \mathcal{G}_1\ \texttt{exports}\ \mathcal{G}_2\ s_2^* = s_1^*\ \texttt{exports}\ \mathcal{G}_2\ \texttt{hiddens}\ \mathcal{G}_1\ s_2^*$

(9) $\texttt{module}\ M\ s_1^*\ \mathcal{G}\ s_2^* = \texttt{module}\ M\ s_1^*\ \texttt{exports}\ \mathcal{G}\ s_2^*$

Projection functions. $[\![d]\!](N)$ yields the grammar corresponding to the module named N. *exp* yields the exported part of a module and *hid* yields the hidden part of a module.

```
hiddens
  context-free syntax
    "⟦" Definition "⟧" "(" ModuleName ")" → Sections
    exp(Sections)                         → Grammar
    hid(Sections)                         → Grammar
```

(10) $[\![m_1^*\ \texttt{module}\ N\ s_1^*\ m_2^*]\!](N) = s_1^*\ s_2^*$ when $s_2^* = [\![m_1^*\ m_2^*]\!](N)$

(11) $[\![d]\!](N) = \text{otherwise}$

(12) $exp() = \emptyset$
(13) $hid() = \emptyset$
(14) $exp(s_1^+\ s_2^+) = exp(s_1^+)\ exp(s_2^+)$
(15) $hid(s_1^+\ s_2^+) = hid(s_1^+)\ hid(s_2^+)$
(16) $exp(\texttt{exports}\ \mathcal{G}) = \mathcal{G}$
(17) $hid(\texttt{exports}\ \mathcal{G}) = \emptyset$
(18) $exp(\texttt{hiddens}\ \mathcal{G}) = \emptyset$
(19) $hid(\texttt{hiddens}\ \mathcal{G}) = \mathcal{G}$

The semantics of a module named M in a definition d is expressed by $m[\![d]\!](M)$ 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.

```
exports
  context-free syntax
    "m[[" Definition "]]" "(" ModuleName ")" → Grammar
hiddens
  sorts IG
  context-free syntax
    "<" Imports "," Grammar ">"                          → IG
    imp "[[" Definition "]]" "(" Imports "," Import ")"  → IG
    ims "[[" Definition "]]" "(" Imports "," Imports ")" → IG
    gra "[[" Definition "]]" "(" Imports "," Grammar ")" → IG
```

$$(20)\quad \frac{[\![d]\!](M) = s^*,\ \ gra[\![d]\!](,\ exp(s^*)\ hid(s^*)) = \langle \mathsf{i}^*,\ \mathcal{G}\rangle}{m[\![d]\!](M) = \mathcal{G}}$$

The function *gra* expands all the imports in a grammar. A structure $\langle \mathsf{i}^*, \mathcal{G}\rangle$ denotes a flattened grammar with the list of imports i^* that were expanded to flatten the grammar. It prevents the flattening function from looping in case of cyclic imports.

$$(21)\quad \frac{gra[\![d]\!](\mathsf{i}_1^*,\ \mathcal{G}_1) = \langle \mathsf{i}_2^*,\ \mathcal{G}_1{}'\rangle,\ \ gra[\![d]\!](\mathsf{i}_2^*,\ \mathcal{G}_2) = \langle \mathsf{i}_3^*,\ \mathcal{G}_2{}'\rangle}{gra[\![d]\!](\mathsf{i}_1^*,\ \mathcal{G}_1\ \mathcal{G}_2) = \langle \mathsf{i}_3^*,\ \mathcal{G}_1{}'\ \mathcal{G}_2{}'\rangle}$$

(22) $gra[\![d]\!](\mathsf{i}_1^*,\ \texttt{imports}\ \mathsf{i}_2^*) = ims[\![d]\!](\mathsf{i}_1^*,\ \mathsf{i}_2^*)$
(23) $gra[\![d]\!](\mathsf{i}^*,\ \mathcal{G}) = \langle \mathsf{i}^*,\ \mathcal{G}\rangle \quad \text{otherwise}$

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

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

$$(25)\quad \frac{imp[\![d]\!](\mathsf{i}_1^*,\ \mathsf{i}) = \langle \mathsf{i}_3^*,\ \mathcal{G}_1\rangle,\ \ ims[\![d]\!](\mathsf{i}_3^*,\ \mathsf{i}_2^*) = \langle \mathsf{i}_4^*,\ \mathcal{G}_2\rangle}{ims[\![d]\!](\mathsf{i}_1^*,\ \mathsf{i}\ \mathsf{i}_2^*) = \langle \mathsf{i}_4^*,\ \mathcal{G}_1\ \mathcal{G}_2\rangle}$$

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.

(26) $imp[\![d]\!](\mathsf{i}_1^*\ \mathsf{i}\ \mathsf{i}_2^*,\ \mathsf{i}) = \langle \mathsf{i}_1^*\ \mathsf{i}\ \mathsf{i}_2^*,\ \emptyset\rangle$

$$(27)\quad \frac{[\![d]\!](M) = s^*}{imp[\![d]\!](\mathsf{i}^*,\ M) = gra[\![d]\!](\mathsf{i}^*\ M,\ exp(s^*))} \quad \text{otherwise}$$

$$(28)\quad \frac{[\![d]\!](M) = s^*,\ \ \mathcal{G} = (exp(s^*))\ \rho}{imp[\![d]\!](\mathsf{i}^*,\ M\ \rho) = gra[\![d]\!](\mathsf{i}^*\ M\ \rho,\ \mathcal{G})} \quad \text{otherwise}$$

8 Overview of Appendices

The appendices to this paper define the extensions and details of extensions that were not yet treated. The appendices are available electronically via the WWW page of the workshop: http://www.cwi.nl/~gipe/asf+sdf95/.

A Examples Syntax definition of Pico and SDF$_1$ in SDF$_1$.

B Library Modules Layout and Booleans.

C Symbols Symbol sets

D Kernel SDF Symbols used in a grammar

E Basic SDF Abstract syntax

F Priority SDF Membership of priorities and associativity declaration.

G Literals SDF Syntax and normalization for literals. A literal like `"then"` is defined by adding a production [\116] [\104] [\101] [\110] → `"then"`.

H Sorts SDF Syntax for sort identifiers and declaration of sorts.

I Character Class SDF Syntax and normalization of character classes.

J Renamings For each extension renaming of symbols and productions over the syntax introduced in the extension is defined.

K SDF$_1$ Modules for syntax, normalization, parsing etc. of SDF$_1$. The modules mainly import earlier defined modules. Furthermore, an interactive environment for SDF$_1$ is defined, that contains tools for applying various normalizations and interpretations.

9 Conclusions

In this paper we 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. This entails that it is easier to replace a feature by a variant or to add a new feature without affecting the design of all other features.

We developed several features for syntax definition that are applicable in many areas. All features extend a kernel that introduces context-free productions.

The distinction between lexical and context-free syntax is not presented as a distinction between regular and context-free grammars. Instead, lexical and context-free syntax are defined both by context-free productions. The only difference between the two levels is that symbols in context-free syntax can be separated by layout. Interference between the levels is avoided by renaming the sorts. All features and interpretations available for lexical syntax are also available for context-free syntax and vice versa. This holds in particular for parsing and disambiguation of lexical syntax that are conventionally done by separate methods.

Priorities are interpreted as a filter on sets of parse trees as prescribed by [HHKR92]. This approach can be extended to other disambiguation methods [KV94].

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

to 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.

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.

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 reference manual [HHKR92]. 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 SDF_1 that is intended to replace SDF in the new architecture. A first specification of a parser generator for SDF_1 was a considerably easier task, due to the uniform abstract syntax and elimination of cases by normalization (and also due to the earlier experience). A complete implementation is expected soon [Vis95b]. This implementation will give more experience with scannerless parsing.

This paper introduces only a small family of syntax definition formalisms. We plan to investigate new features for the description of linguistic phenomena of programming languages for which no or poor formal support in syntax definition formalisms exist; for instance, the syntax of extensible languages (like ASF+SDF), mildly context-sensitive phenomena on lexical level (e.g. `\verb` construct in LaTeX) and disambiguation on the basis of type information. The generalization of the regular expression approach to type grammars [Vis95c] provides the syntactic analogue of the leveled signatures of [Mei91]. Another challenge is the application of the features developed here in application areas other than programming language design.

The ASF+SDF formalism and meta-environment are a considerable improvement over the techniques that were available when SDF was originally designed. This becomes clear when we compare the design of SDF in [HHKR92] with the design of SDF_1 in this paper. The use of ASF+SDF provides fast feedback and makes experiments feasible. However, it remains difficult to define a clean family of languages and it is still impossible to create new formalisms by specifying its features by their catalogue number. Features tend to interfere and the complexity of a formalism is higher than the sum of the complexities of its features (see also [BB89]). The specification of large formalisms in a compositional manner remains a challenge and tools that support such specifications can always be improved.

Acknowledgements The author thanks Wilco Koorn and Jan Rekers for their SDF improvement shopping lists and Jan Rekers and Paul Klint for their comments on drafts of this paper.

References

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

[AU72] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling - Volume I: Parsing.* Series in Automatic Computation. Prentice-Hall, 1972.

[Aus89] D. Austry. *The Metal Environment Guide.* Sophia-Antipolis, 1989. In: The CENTAUR Documentation - Version 0.9, Volume I.

[Bac59] J. W. Backus. 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, June 15–20 1959. UNESCO.

[BB89] J. C. M. Baeten and J. A. Bergstra. Design of a specification language by abstract syntax engineering. In J. A. Bergstra and L. M. G. Feijs, editors, *Algebraic Methods II: Theory, Tools and Applications*, volume 490 of *LNCS*, pages 363–394. Springer-Verlag, 1989.

[BHK89] J. A. Bergstra, J. Heering, and P. Klint. 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, 1989. Chapter 1.

[Bra95] M. G. J. van den Brand. Pretty printing in the ASF+SDF meta-environment: Past, present and future. In M. G. J. van den Brand et al., editors, *Proc. ASF+SDF'95*. Technical Report P9504, Programming Research Group, University of Amsterdam, May 1995.

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

[BV95] Mark van den Brand and Eelco Visser. Generation of formatters for context-free languages. Technical report, Programming Research Group, University of Amsterdam, 1995. In preparation.

[C+94] Robert Cartwright et al., editors. *Report of a Workshop on Future Directions in Programming Languages and Compilers*, May 31 1994.

[CC93] R. Cordy, James and Ian H. Carmichael. *The TXL Programming Language. Syntax and Informal Semantics. Version 7*. Software Technology Laboratory, Dept. of Comp. and Inf. Science, Queen's University at Kingston, Kinston, Canada, 7 edition, June 1993.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[Chu40] A. Church. A formulation of a Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Ear75] J. Earley. Ambiguity and precedence in syntax description. *Acta Informatica*, 4(1):183–192, 1975.

[Eke92] S. M. Eker. Associative matching for linear terms. Report CS-R9224, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992.

[GTWW77] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, 1977.

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

[HHKR92] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. *The syntax definition formalism SDF — Reference Manual*, 1992. Version 6 December 1992. Earlier version in *SIGPLAN Notices*, 24(11):43-75, 1989. Available as ftp://ftp.cwi.nl /pub/gipe/reports/ SDFManual.ps.Z.

[Joh75] S. C. Johnson. YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories, Murray Hill, N.J., 1975.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[Kli94] Paul Klint. Design of a fixed exchange format for ASF+SDF. Unpublished technical report, July 19 1994.

[Knu64] Donald E. Knuth. Backus Normal Form vs. Backus Naur Form. *Communications of the ACM*, 7(12):735–736, December 1964.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Correction in: Mathematical Systems Theory 5(1), pp. 95-96, Springer-Verlag, 1971.

[Kos71] C. H. A. Koster. Affix grammars. In J. E. L. Peck, editor, *Algol-68 Implementation*. North-Holland, Amsterdam, 1971.

[KV94] Paul Klint and Eelco Visser. 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, October 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano. Also as TR P9426, Programming Research Group, University of Amsterdam, ftp://ftp.fwi.uva.nl /pub/programming-research/ reports/1994/ P9426.ps.Z.

[Lam58] Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170, March 1958.

[Lee72] J. A. N. Lee. The formal definition of the BASIC language. *Computer Journal*, 15(1):37–41, February 1972.

[LS86] M. E. Lesk and E. Schmidt. *LEX — A lexical analyzer generator*. Bell Laboratories, 1986. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).

[McC62] J. McCarthy. Towards a mathematical science of computation. In *Information Processing 62, Proc. IFIP Congress 62*, pages 21–28. North-Holland, 1962.

[Mei91] K. Meinke. Equational specification of abstract types and combinators. Technical Report CSR-11-91, University College of Swansea, Swansea, 1991. To appear in Proc. Computer Science Logic 1991.

[MR94] David Muir Sharnoff and Steven Allen Robenalt. Catalog of free compilers and interpreters. http://remarque.berkeley.edu/∼muir/free-compilers/, 1994. Electronically distributed list of freely available compilers, including a list of syntax definition tools (see CATEGORY/compiler-1.html).

[N+60] P. Naur et al. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, May 1960.

[PQ94] Terence J. Parr and Russel W. Quong. Adding semantic and syntactic predicates to LL(k): pred-LL(k). In Peter A. Fritzson, editor, *Compiler Construction, 5th International Conference, CC'94*, volume 786 of *LNCS*, pages 263–277, Edinburgh, U.K., April 1994. Springer-Verlag.

[PW80] F. C. N. Pereira and D. H. D. Warren. Definite Clause Grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

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

[SC89] Daniel J. Salomon and Gordon V. Cormack. Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Notices*, 24(7):170–178, 1989.

[Tho94] Mikkel Thorup. Controlled grammatic ambiguity. *ACM Transactions on Programming Languages and Systems*, 16(3):1024–1050, May 1994.

[Vis93] Eelco Visser. Combinatory algebraic specification & compilation of list matching. Mastersthesis, University of Amsterdam, Amsterdam, June 1993. Available as ftp://ftp.cwi.nl /pub/gipe/reports/ Vis93.CAS-LM.ps.Z.

[Vis95a] Eelco Visser. Optimizing parsing schemata by disambiguation filters. In *Proc. Accolade95*. The Dutch Graduate School in Logic, March 1995. To appear.

[Vis95b] Eelco Visser. A parser generator for SDF_1. Technical report, Programming Research Group, University of Amsterdam, 1995. In preparation.

[Vis95c] Eelco Visser. Type grammars. Technical report, Programming Research Group, University of Amsterdam, 1995. In preparation.

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

[Wil82] M. H. Williams. A flexible notation for syntactic definitions. *ACM Transactions on Programming Languages and Systems*, 4(1):113–119, 1982.

[Wir77] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions. *Communications of the ACM*, 20(11):822–823, November 1977.