Chapter 2

Preliminaries

This chapter provides basic notations, definitions and properties needed throughout this book. Whenever possible the notations and definitions are used as they appear in the literature [Comon *et al.* (2008); Engelfriet (1974); Hopcroft *et al.* (2001); Aho *et al.* (1986)]. This chapter can be skipped and referred to when necessary.

2.1 Basic Definitions and Notations

The formal language theory is used to study templates and describe their syntax. This section provides definition for common concepts of formal languages. The definitions are based on automata and language theory:

- A *symbol* is a syntactic entity without any meaning.
- An *alphabet* is a finite, non-empty set of symbols.
- The *rank* (or arity) of a symbol is the number of children.
- A *ranked alphabet* is a pair of an alphabet and ranking functions, where the rank function maps a symbol in the alphabet to a single rank.
- A *string* is a finite sequence of symbols from the given alphabet.
- A *language* is the set of all strings belonging to an alphabet, including the empty string.
- A *terminal* symbol is a symbol from which sentences are formed and it occurs literally in a sentence, i.e. terminal symbols are elements of the alphabet.
- A *nonterminal* symbol is a variable representing a sequence of symbols and it can replace a string of terminal symbols or a string existing of a combination of terminal and nonterminal symbols.

Next to strings, the concept of *tree* is used throughout the book. The trees considered here are finite (finite number of nodes and branches), directed (top-down), rooted (there is one

node, the root, with no branches entering it), ordered (the children of a node are ordered left to right) and labeled (the nodes are labeled with symbols from a given alphabet) [Engelfriet (1974)]. The following terminology is be used:

- A *leaf* is a node with rank 0.
- The *top* of a tree is its root.
- A *path* through a tree is a sequence of nodes connected by branches ("leading downwards").
- A *subtree* of a tree is a tree determined by a node together with all (the subtrees of) its children.

The following list presents the naming convention and basic definitions used throughout the book.

- i, j and r are used for integer variables.
- Σ is used to denote an alphabet. For example, the binary alphabet $\Sigma = \{0,1\}$, and the set of all lower-case letters $\Sigma = \{a,b,\ldots,z\}$.
- Σ^* denotes all strings over an alphabet Σ.
- σ and c for an alphabet symbol.
- N for nonterminal alphabets.
- n for a nonterminal symbol of N.
- y for sequences of alphabet symbols combined with nonterminal symbols (i.e. strings, elements of $(N \cup \Sigma)^*$).
- z for an alphabet symbol or a nonterminal symbol ($z \in (N \cup \Sigma)$).
- ε is used for the empty string or null value.
- $\mathscr{L}$ for languages. A language contains all sentences defined by Σ^*.
- s for a sentence of a language $\mathscr{L}$ defined by Σ^*.
- r denotes the rank of a symbol and $r \in \mathbb{N}_0$. It is defined by the ranking function $r_\sigma = rank(\sigma)$ where $\sigma \in \Sigma$. Each symbol in a ranked alphabet (see Definition 2.1.1) has a unique rank.
- Σ_r for the set of symbols of rank r.
- $Tr(\Sigma)$ denotes the set of trees over a ranked alphabet Σ, i.e. Σ including a set of ranking functions over Σ.
- t for a tree (see Definition 2.1.2).
- a for an alphabet symbol with rank 0 ($a \in \Sigma_0$).
- f for an alphabet symbol with rank greater than 0 ($f \in \Sigma_r$, where $r > 0$).

- X is a set of symbols called variables, where it is assumed that the sets X and Σ_0 are disjoint.
- x is a variable $x \in X$ and is not used for integer values.
- G for grammars.

Definition 2.1.1 (Ranked alphabet). An alphabet Σ is said to be ranked if for each non-negative integer r a subset Σ_r of Σ is specified, such that Σ_r is nonempty for a finite number of r's only, and such that $\Sigma = \bigcup_{r \geqslant 0} \Sigma_r$. If $\sigma \in \Sigma_r$, then σ has rank r.

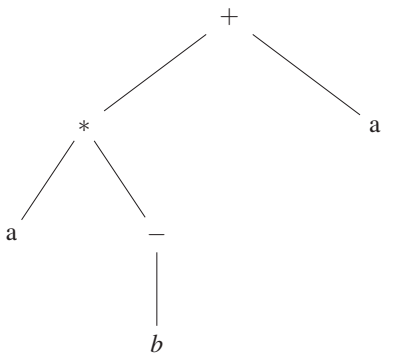
Example 2.1.1 (Ranked alphabet). The alphabet $\Sigma = \{a, b, +, -, *\}$ is converted to a ranked alphabet by specifying $\Sigma_0 = \{a, b\}$, $\Sigma_1 = \{-\}$ and $\Sigma_2 = \{+, *\}$.

Definition 2.1.2 (Tree). Given a ranked alphabet Σ, the set of trees over Σ, denoted by $Tr(\Sigma)$ is the language over the alphabet $\Sigma \cup \{[\,,\,]\}$, where $\Sigma \cap \{[\,,\,]\} = \emptyset$, defined inductively as follows.

(1) If $\sigma \in \Sigma_0$, then $\sigma \in Tr(\Sigma)$.
(2) For $r \geqslant 1$, if $\sigma \in \Sigma_r$ and $t_1, \ldots, t_r \in Tr(\Sigma)$, then $\sigma[t_1 \ldots t_r] \in Tr(\Sigma)$.

Example 2.1.2 (Tree). Consider the ranked alphabet of Example 2.1.1. Then $+[*[a-[b]]a]$ is a tree of this alphabet. This tree can be visualized as:

$+[*[a-[b]]a] =$



Which on its turn represents the concrete expression $(a*(-b))+a$.

Definition 2.1.3 (Linear tree). A tree may contain variables, i.e. placeholders for subtrees. A tree $t \in Tr(\Sigma \cup X)$ is linear when each variable is at most used once in t.

Definition 2.1.4 (Substitution). A substitution (respectively a ground substitution) m is a mapping from X into $Tr(\Sigma \cup X)$ (respectively into $Tr(\Sigma)$) where there are only finitely many variables not mapped to themselves. The domain of a substitution m is the subset of

variables $x \in X$ such that $m(x) \neq x$. The substitution $\{x_1 \leftarrow t_1, \ldots, x_r \leftarrow t_r\}$ maps $x_i \in X$ on $t_i \in Tr(\Sigma \cup X)$, for every index $1 \leq i \leq r$. A substitution is ground when all terms $t_1, \ldots, t_r$ are ground terms, that is, when the terms do not contain variables.
Substitutions can be extended to $Tr(\Sigma \cup X)$ in such a way that:

$$\forall f \in \Sigma_r, \forall t_1, \ldots, t_r \in Tr(\Sigma \cup X) \;\; m(f(t_1, \ldots, t_r)) = f(m(t_1), \ldots, m(t_r)).$$

Example 2.1.3 (Substitution). Let $\Sigma = \{f(\ ,\ ,\ ), g(\ ,\ ), a, b\}$ and $X = \{x_1, x_2\}$. Consider the term $t = f(x_1, x_1, x_2)$. Consider the ground substitution $m = \{x_1 \leftarrow a, x_2 \leftarrow g(b,b)\}$ and the non-ground substitution $m' = \{x_1 \leftarrow x_2, x_2 \leftarrow b\}$. Then $m(t) = t\{x_1 \leftarrow a, x_2 \leftarrow g(b,b)\} = f(a, a, g(b,b))$ and $m'(t) = t\{x_1 \leftarrow x_2, x_2 \leftarrow b\} = f(x_2, x_2, b)$.

Definition 2.1.5 (Tree homomorphism). Let Σ and Σ' be two, not necessarily disjoint, ranked alphabets. For each $r > 0$ such that Σ contains a symbol of rank r, a set of variables $X_r = \{x_1, \ldots, x_r\}$ disjoint from Σ and Σ' is defined.
Let h_Σ be a mapping which, with $f \in \Sigma$ of rank r, associates a term $t_f \in Tr(\Sigma', X_r)$. The tree homomorphism $h : Tr(\Sigma) \rightarrow Tr(\Sigma')$ is determined by h_Σ as follows:

- $h(a) = t_a \in Tr(\Sigma')$ for each $a \in \Sigma$ of rank 0,
- $h(f(t_1, \ldots, t_n)) = t_f\{x_1 \leftarrow h(t_1), \ldots, x_r \leftarrow h(t_r)\}$
  where $t_f\{x_1 \leftarrow h(t_1), \ldots, x_r \leftarrow h(t_r)\}$ is the result of applying the substitution $\{x_1 \leftarrow h(t_1), \ldots, x_r \leftarrow h(t_r)\}$ to the term t_f.

h_Σ is called a *linear tree homomorphism* when no t_f contains two occurrences of the same x_r. Thus a linear tree homomorphism cannot copy trees.

Example 2.1.4 (Tree homomorphism). Let $\Sigma = \{g(\ ,\ ,\ ), a, b\}$ and $\Sigma' = \{f(\ ,\ ), a, b\}$. Consider the tree homomorphism h determined by h_Σ defined by: $h_\Sigma(g) = f(x_1, f(x_2, x_3)), h_\Sigma(a) = a, h_\Sigma(b) = b$. For instance: If $t = g(a, g(b,b,b), a)$, then $h(t) = f(a, f(f(b, f(b,b)), a))$.

2.2 Context-free Grammars

This book will focus on the generation of sentences of languages aimed to express programs executed or interpreted by a computer. The rules for constructing valid sentences of these languages can be specified by context-free grammars. The *syntax*[1] of a language is its valid

[1] The syntax rules do not specify the meaning of a sentence; as a result a syntactical correct sentence can be nonsense.

set of sentences. Compilers or interpreters for most programming languages are based on LL or LR parsers. LL or LR parsers can handle a subset of the context-free grammars, which implies that these programming languages are context-free languages. A context-free language $\mathscr{L}(G)$ is specified by a context-free grammar G. A sentence belonging to the set of sentences specified by a context-free grammar is called a *well-formed* sentence. The context-free grammar is defined as follows [Hartmanis (1967)]:

Definition 2.2.1 (Context-free grammar). A context-free grammar (CFG) is a four-tuple $\langle \Sigma, N, S, Prods \rangle$ where

- Σ is a finite set of terminal symbols, i.e. the alphabet.
- N is a finite set of nonterminal symbols and $N \cap \Sigma = \emptyset$.
- S is the start symbol, or axiom, and $S \in N$.
- $Prods$ is a finite set of production rules of the form $n \to y$ where $n \in N$ and $y \in (N \cup \Sigma)*$.

Each context-free grammar G_{cfg} can be transformed into a Chomsky normal form without changing the language generated by that grammar [Hotz (1980)]. A context-free grammar of the Chomsky normal form only contains rules of the forms:

(1) $n \to \varepsilon$, where $n \in N$ and A is the start symbol;
(2) $n \to s$, where $n \in N$ and $s \in \Sigma^*$;
(3) $n \to n_1 n_2$, where $n,\ n_1,\ n_2 \in N$.

Example 2.2.1 shows a context-free grammar definition for a language based on boolean algebra.

Example 2.2.1 (Context-free grammar). Let G_{bool} be a context-free grammar with, $\Sigma = \{\text{␣}, \backslash\text{n}, \tilde{}, \&, |, (,), true, false\}$, nonterminals $N = \{\text{E}, \text{T}, \text{F}, \text{L}, \text{C}\}$, start symbol $S = \text{E}$ and rules

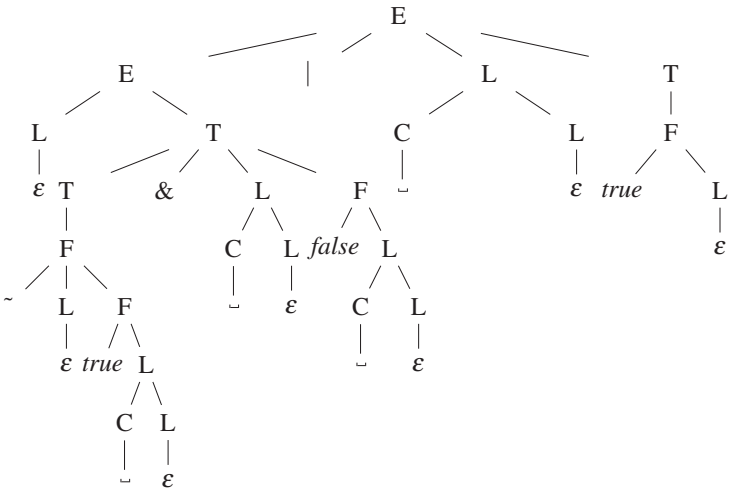
$$Prods = \left\{ \begin{array}{lll} \text{E} \to \text{L T}, & \text{F} \to \text{“}\tilde{}\text{”\ L F}, & \text{L} \to \text{C L}, \\ \text{E} \to \text{E “|” L T}, & \text{F} \to \text{“(” L E “)” L}, & \text{L} \to \varepsilon, \\ \text{T} \to \text{F}, & \text{F} \to \text{“}true\text{” L}, & \text{C} \to \text{“␣”}, \\ \text{T} \to \text{T “\&” L F}, & \text{F} \to \text{“}false\text{” L}, & \text{C} \to \text{“}\backslash\text{n”} \end{array} \right\}$$

The layout syntax is defined by the production rules for the nonterminal L. We assume that this layout nonterminal is inserted after every terminal symbol in the grammar and before the start nonterminal, in the case of the first production rule after nonterminal E [Johnstone *et al.* (2011)].

A context-free grammar defines a set of sentences, i.e. the language $\mathscr{L}(G)$, where for each $s \in \Sigma$ a derivation exists $S \overset{*}{\underset{G}{\Longrightarrow}} s$. If a sentence belongs to $\mathscr{L}(G)$, a parse tree can be constructed using the grammar. This tree is derived by applying the production rules of the grammar to construct the sentence and it is called the *parse tree*. Example 2.2.2 shows a parse tree derived from a sentence of $\mathscr{L}(G_{\text{bool}})$.

Example 2.2.2 (Parse tree). Let s be `~true & false | true`, the parse tree of s using the grammar G is:

$d_G(s) =$



A parse tree represents the hierarchical structure of the sentence expressed by the production rules of its grammar. Normally such parse trees are automatically constructed from a given sentence when a parser is used based on the grammar. A parser can, for instance, use algorithms like LL [Aho *et al.* (1989)] and LR [Knuth (1965)].
The parse tree contains all necessary information to restore the original sentence. Consider the parse tree of Example 2.2.2, and read the leaves from left to right, the original sentence is visible. The *yield* function reconstructs the original string of a parse tree. It traverses a parse tree in order to compute the original sentence from it by concatenating the leaves (taking the leaf symbols as letters) from left to right.

Definition 2.2.2 (Yield). The *yield* function is defined by the following two rules:

- $yield(a) = a$ if $a \in \Sigma_0$;
- $yield(f(t_1,\ldots,t_r)) = yield(t_1)\cdot\ldots\cdot yield(t_r)$ if $f \in \Sigma_r$ and $t_i \in Tr(\Sigma \cup N)$, where $\cdot$ denotes the string concatenation.

2.3 Regular Tree Grammars

A *regular tree language* is a set of trees generated by a *regular tree grammar*. The definition of regular tree grammars is:

Definition 2.3.1 (Regular tree grammar). A regular tree grammar (RTG) is a four-tuple $\langle \Sigma, N, S, Prods \rangle$, where:

- Σ is a finite set of terminal symbols with rank $r \geq 0$.
- N is a finite set of nonterminal symbols with rank $r = 0$ and $N \cap \Sigma = \emptyset$.
- S is a start symbol and $S \in N$.
- $Prods$ is a finite set of production rules of the form $n \to t$, where $n \in N$ and $t \in Tr(\Sigma \cup N)$.

Example 2.3.1 shows a regular tree grammar (taken from [Cleophas (2008)]).

Example 2.3.1 (Regular tree grammar). Let G be the regular tree grammar with $\Sigma = \{a(\ ,\ ), b(\ ), c\}$, nonterminals $N = \{E, W\}$, start symbol E, and rules

$$
\begin{aligned}
Prods = \{ & \\
& \mathrm{E} \to \mathrm{W}, \\
& \mathrm{W} \to \mathrm{b(W)}, \\
& \mathrm{W} \to \mathrm{b(a(c,c))} \\
\} &
\end{aligned}
$$

The language of this grammar is

$$\mathscr{L}(G_{rtg}) = \{b(a(c,c)), b(b(a(c,c))), b(b(b(a(c,c)))), \ldots\}.$$

The parse steps of the term $b(b(a(c,c)))$ are $E \Rightarrow W \Rightarrow b(W) \Rightarrow b(b(a(c,c)))$, where $\Rightarrow$ is a derivation step.

Regular tree languages have a number of properties [Cleophas (2008)], the one being important for this book is *recognizability* of regular tree languages. Recognizable tree languages are the languages recognized by a finite tree automaton. Regular tree languages are recognizable by (non)-deterministic bottom up finite tree automata and non-deterministic top-down tree automata [Comon *et al.* (2008)]. The set of languages recognizable by deterministic top-down tree automata is limited to the class of path-closed tree languages [Virágh (1981)], a subset of regular tree languages (see Section 3.3).

2.4 Relations between CFL and RTL

A number of relations can be defined between context-free languages and regular tree languages [Cleophas (2008)]. Amongst others, a tree can be represented as a term. These terms can be parsed using a context-free grammar, since printing a (sub)tree to text does not depend on the sibling nodes of that (sub)tree. This context-free grammar of the used term representation is given in Figure 2.4. For example the tree of Example 2.2.1 can be represented by the term

$$
\begin{aligned}
&E(\\
&E(L(\varepsilon),\\
&T(T(F(\tilde{},L(\varepsilon),F(true,L(C(\lrcorner),L(\varepsilon))))),\\
&\&,L(C(\lrcorner),L(\varepsilon)),F(false,L(C(\lrcorner),L(\varepsilon))))),\\
&|,L(C(\lrcorner),L(\varepsilon)),T(F(true,L(\varepsilon)))\\
&)
\end{aligned}
$$

The goal of this book is the formal definition of code generators based on templates. For this purpose, the relation between regular tree languages and the parse trees of context-free languages is relevant. The *parse* function takes a string and a grammar and returns the parse tree of that string when the string can be produced by that grammar. The way parse algorithms create a parse tree shows regularity, which suggests that the parse trees are indeed regular. A proof that the set of parse trees of a context-free grammar is a regular tree language can be found in [Comon *et al.* (2008)]. The following definition shows the derivation of the regular tree grammar $\mathscr{L}(G_{\text{pt}})$ defining the set of parse trees of a context-free grammar $\mathscr{L}(G_{cfg})$.

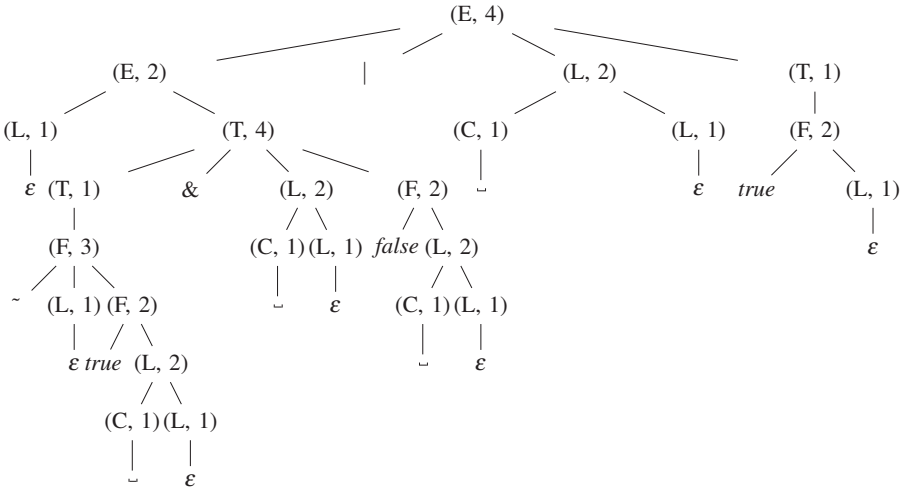
Definition 2.4.1 (Regular tree grammar for parse trees). Let $G_{cfg} = \langle \Sigma, N, S, Prods \rangle$ be a context-free grammar. The regular tree grammar $G_{\text{pt}} = \langle \Sigma', N', S', Prods' \rangle$ defining the parse trees of G_{cfg} is derived by the following rules:

- The start symbol of both grammars is equal: $S = S'$,
- The set of nonterminals of both grammars is equal: $N = N'$,
- The alphabet of G_{pt} is derived by the following rule: $\Sigma' = \Sigma \cup \{\varepsilon\} \cup \{(n,r) \mid n \in N,\ \exists n \to y \in Prods \text{ with } r \text{ equal to the number of symbols of } y\}$. In parse trees, a symbol can normally have a different number of children, when alternative production rules have a different pattern length. In tree languages a symbol must have a fixed rank, so

the symbol (n,r) is introduced for each $n \in N$ such that there is a rule $n \to y$ where y has r symbols.

- The set of productions $Prods'$ of G_{pt} is derived by the following rules:
  if $n \to \varepsilon \in Prods$ then $n \to (n,0)(\varepsilon) \in Prods'$.
  if $(n \to n_1 \dots n_r) \in Prods$ then $n \to (n,r)(n_1,\dots,n_r) \in Prods'$.

Example 2.4.1 (Regular tree grammar for parse trees). Since normally in parse trees a symbol can have different number of children, an updated version of the parse tree displayed in Example 2.2.2 is given: $d_G(s) =$



Using the definition given above, the G_{pt} defining the language of parse trees of G_{bool} can be derived. The result of the derivation is a regular tree grammar G_{pt} with, $\Sigma = \{\text{␣}, \backslash\text{n}, \tilde{}, \&, |, (,), true, false\}$, nonterminals $N = \{(\text{E},4), (\text{E},2), (\text{T},4), (\text{T},1), (\text{F},5), (\text{F},3), (\text{F},2), (\text{L},2), (\text{L},1), (\text{C},1)\}$, start symbol $S = \text{E}$ and rules

$$Prods = \left\{ \begin{array}{lll} (\text{E},2) \to \text{L T}, & (\text{F},3) \to \text{“}\tilde{}\text{”} \text{ L F}, & (\text{L},2) \to \text{C L}, \\ (\text{E},4) \to \text{E “|” L T}, & (\text{F},5) \to \text{“(” L E “)” L}, & (\text{L},1) \to \varepsilon, \\ (\text{T},1) \to \text{F}, & (\text{F},2) \to \text{“}true\text{” L}, & (\text{C},1) \to \text{“␣”}, \\ (\text{T},4) \to \text{T “\&” L F}, & (\text{F},2) \to \text{“}false\text{” L}, & (\text{C},1) \to \text{“}\backslash\text{n”} \end{array} \right\}$$

The following statements hold for context-free grammars and regular tree grammars:

- $\mathscr{L}(G_{\text{pt}}) = parse(\mathscr{L}(G_{cfg}))$
- $\mathscr{L}(G_{cfg}) = yield(\mathscr{L}(G_{\text{pt}}))$

Hence, also

- $\mathscr{L}(G_{cfg}) = yield(parse(\mathscr{L}(G_{cfg})))$
- $\mathscr{L}(G_{\text{pt}}) = parse(yield(\mathscr{L}(G_{\text{pt}})))$

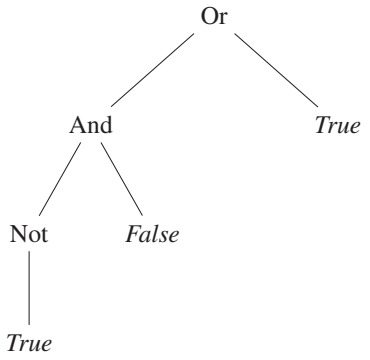
Sentences can be mapped to a parse tree and back to the original sentence. This is a result of the fact that the *parse* function does not throw away information, but it builds a tree with the original sentence distributed over its leaves.

2.5 Abstract Syntax Trees

The set of abstract syntax trees of a language is called the abstract syntax and this abstract syntax is defined by a regular tree grammar. These abstract syntax trees are considered as the abstract representation of well-formed sentences [Donzeau-Gouge *et al.* (1984)]. The abstract syntax representation of a sentence is unique, while the textual representation is usually cluttered with optional and semantically irrelevant details such as blanks and line feeds. These optional and semantically irrelevant details are called *syntactic sugar*.
The *abstract syntax tree* is a representation of a sentence without superfluous nodes, such as nodes corresponding to keywords and *Chain rules* [Koorn (1994)]. A chain rule is a grammar rule of the form $n_1 \rightarrow n_2$, where both n_1 and n_2 are nonterminals.

Example 2.5.1 (Abstract syntax tree). An example of an abstract syntax tree of the sentence s given in Example 2.2.2 is:

$AST(s) =$



It can also be represented as a term

$$AST(s) = Or(And(Not(True), False), True).$$

```
begin declare input : natural,
      output : natural,
      repnr : natural,
      rep : natural;
input := 14;
output := 1;
while input − 1 do
    rep := output;
    repnr := input;
    while repnr − 1 do
       output := output + rep;
       repnr := repnr − 1
    od;
    input := input − 1
od
end
```

Fig. 2.1 A PICO program.

2.6 Used Languages and Formalisms

This section discusses the syntax of the formalisms used throughout the book. The language PICO is also presented here. PICO is used for illustrating purposes.

2.6.1 *The PICO Language*

The goal of PICO [Bergstra *et al.* (1989)] is to have a simple language, large enough to illustrate the concepts of parsing, type checking and evaluation. Informal, the PICO language is the language of while-programs. The main features of PICO are:

- Two types: natural numbers and strings.
- Variables must be declared in a separate section.
- Expressions can be made of constants, variables, addition, subtraction and concatenation.
- Statements: assignment, if-then-else and while-do.

A PICO program consists of declarations followed by statements. Variables must be declared before they can be used in the program. Statements and expressions can be used in the body of the program. An example PICO program that computes the factorial function is given in Figure 2.1[2].

[2]Example borrowed from http://www.meta-environment.org/doc/books/extraction-transformation/language-definitions/language-definitions.html (accessed on December 18, 2011)

2.6.2 *Syntax Definition Formalism*

Template grammars, as presented in Chapter 5, can easily become ambiguous and dealing with ambiguities is a primary requirement for parsing templates. The Scannerless Generalized LR (SGLR) algorithm, and its implementation the SGLR parser [Visser (1997)], can deal with these ambiguities. Grammars for the SGLR parser are defined using the Syntax Definition Formalism (SDF) [Heering *et al.* (1989)], which is the main reason for using SDF in this book.

In contrast with other parser algorithms, such as LL or LALR, and their used BNF-like [Backus *et al.* (1960)] grammar formalisms, SDF supports the complete class of context-free grammars. This enables the support for modular grammar definitions. Pieces of grammar can be embedded in modules and imported by other modules. Modules may have formal symbol parameters, which can be bound by actual symbols using imports. The syntax of module parameters is: `module <ModuleName> [ <Symbol>+ ]`. When the module is imported, all occurences of the formal parameters will be substituted by the actual parameters The modularity enables combining and reusing of grammars.

The core of an SDF module consists of the elements of the mathematically four-tuple definition of a context-free grammar as defined in Section 2.2. In SDF nonterminals are called *sorts* and declared after the similar keyword `sorts`. *Symbols* is the global name for literals, sorts and character classes and form the elementary building blocks of SDF syntax rules. Start symbols are declared after the keyword `context-free start-symbols`. Production rules are declared in sections `context-free syntax` and `lexical syntax`. The *productions rules* contain a *syntactical pattern* at the left-hand side and a resulting sort at the right-hand side. This left-hand side pattern is based on a combination of symbols, i.e. terminals in combination with nonterminals. Symbols can be declared as optional via a postfix question mark. In the `context-free syntax` section a `LAYOUT` sort is automatically injected between every symbol in the left-hand side of a production rule. The `LAYOUT` sort is an SDF/SGLR embedded sort for white spaces and line feeds. This mechanism differs from the earlier presented approach, where the layout nonterminal should be present explicitly in the production rules. To illustrate SDF, the SDF module shown in Figure 2.2 defines the PICO language.

SDF also supports concise declaration of associative lists. A list is declared by its elements and a postfix operator `*` or `+`, with the respectively meaning of at least zero times or at least one time. Lists may contain a separator, which are declared via the pattern `{Symbol Literal}*`, where `Symbol` defines the syntax of the elements and `Literal` de-

```
module languages/pico/syntax/Pico

imports basic/NatCon
imports basic/StrCon
imports basic/Whitespace

hiddens
  context-free start-symbols
    PROGRAM

exports
  sorts PROGRAM DECLS ID-TYPE STATEMENT EXP TYPE PICO-ID

  context-free syntax
    "begin" DECLS {STATEMENT";" }* "end"
                        -> PROGRAM {cons("program")}
    "declare" {ID-TYPE "," }* ";"
                        -> DECLS {cons("decls")}
    PICO-ID ":" TYPE  -> ID-TYPE {cons("decl")}

    PICO-ID ":=" EXP  -> STATEMENT {cons("assignment")}
    "if" EXP "then" {STATEMENT ";" }*
             "else" {STATEMENT ";" }* "fi"
                        -> STATEMENT {cons("if")}
    "while" EXP "do" {STATEMENT ";" }* "od"
                        -> STATEMENT {cons("while")}

    PICO-ID             -> EXP {cons("id")}
    NatCon              -> EXP {cons("natcon")}
    StrCon              -> EXP {cons("strcon")}
    EXP "+" EXP         -> EXP {cons("add")}
    EXP "-" EXP         -> EXP {cons("sub")}
    EXP "||" EXP        -> EXP {cons("concat")}
    "(" EXP ")"         -> EXP {cons("bracket")}

    "natural"           -> TYPE {cons("natural")}
    "string"            -> TYPE {cons("string")}

  lexical syntax
    [a-z] [a-z0-9]*   -> PICO-ID  {cons("picoid")}

  lexical restrictions
    PICO-ID -/- [a-z0-9]
```

Fig. 2.2 The PICO grammar in SDF.

fines the separator syntax, for example: `{STATEMENT ";" }*`. This kind of lists are called *separated lists*.

The production rules can be annotated with a list of properties between curling brackets at the right-hand side of the rule. The parser includes these annotations in the parse tree at

the node produced by the production rule. Tools processing the parse tree can use these annotations.

For example, an abstract syntax for an SDF grammar can be specified using annotations. The label of the abstract syntax representation of a production rule is assigned by a so-called constructor value. This constructor value is used during desugaring to instantiate the nodes of the abstract syntax tree. The constructor is declared via a `cons` value. SDF requires that a constructor is unique for a given sort, and in that way suffices the first uniqueness requirement of the *desugar* function of Definition 3.1.1. It does not require that a constructor is only used for a fixed rank and thus SDF does not satisfy the requirements to generate a legal regular tree language. Production rules can also annotated with the keyword `reject`. The `reject` annotation specifies that strings specified by the rule is rejected for that nonterminal. Rejects should only used for nonterminals defining lexical syntax. The rejects are used to specify the lexical disambiguation rule "prefer keywords".

Besides these core features of SDF, it supports modularization of grammar definitions. Every grammar definition file is declared as a module with a name, which can be imported by other modules. Modules are imported via the `imports` keyword followed by the name(s) of imported modules. Sections of a grammar module can be declared hidden or visible via the keywords `hiddens` and `exports` to prevent unexpected collisions between grammar modules result in undesired ambiguities. Exported sections are visible in the entire grammar, while hidden sections are only visible in the local grammar module. Although the namespace of a nonterminal is global, adding a new alternative to a nonterminal, which is defined in an imported module, does not change the recognized language of imported module. This is because per module an LR parse table is generated, based on the module dependency graph. SDF also provides syntax to define priorities and associativity to express disambiguation rules in a grammar.

Considering again the SDF module shown in Figure 2.2. The nonterminals and production rules are declared in the exported section. The start symbol is `PROGRAM`, which is the root sort for a PICO program. In the PICO module the start symbol is declared hidden to prevent automatic propagation to modules importing this grammar. The annotation feature of SDF is also used in the PICO module to specify the abstract syntax tree. The definition for the sorts `NatCon` and `StrCon`, and a module defining white space (spaces, tabs, and new lines) are imported.

The grammar of Figure 2.2 is used to parse PICO programs, like Figure 2.1. The abstract syntax tree, result of parsing the program and desugaring the parse tree is shown in Fig-

```
program(
  decls([
    decl( "input", natural ),
    decl( "output", natural ),
    decl( "repnr", natural ),
    decl( "rep", natural )
  ]),
  [
    assignment( "input", natcon( 14 ) ),
    assignment( "output", natcon( 1 ) ),
    while( sub( id( "input" ), natcon( 1 ) ),[
      assignment( "rep", id( "output" ) ),
      assignment( "repnr", id( "input" ) ),
      while( sub( id( "repnr" ), natcon ( 1 ) ),[
        assignment( "output", add( id( "output" ),
                                   id( "rep" ) ) ),
        assignment( "repnr", sub( id( "repnr" ),
                                  natcon( 1 ) ) )
      ]),
      assignment( "input", sub( id( "input" ),
                                natcon( 1 ) ) )
    ])
  ]
)
```

Fig. 2.3 Abstract syntax tree of PICO program of Figure 2.1.

ure 2.3. The tree is displayed in the ATerm format, to be discussed in Section 2.6.3.

2.6.3 *ATerms*

The syntax for terms used in this book is based on a subset of the ATerms syntax [van den Brand *et al.* (2000)]. ATerms have support for lists, which are not directly supported by the presented regular tree grammars. Lists must be binary trees to stay fully compatible with the regular tree grammars. The serialized term notation of the list is only a shorthand notation for these binary trees, i.e. the list

```
[ "a", "b", "c" ]
```

has the internal representation

```
[ "a", [ "b", [ "c" , []]]].
```

Since the lists of ATerms are internally stored as binary trees, where the left branch is the element and right branch the list or empty list, the use of ATerms meets this requirement. The subset of the ATerm language is defined by the SDF definition of Figure 2.4.

```
module ATerms

imports StrCon
        IdCon

exports
  sorts AFun ATerm

  context-free syntax
        StrCon  -> AFun
        IdCon   -> AFun
        AFun                                -> ATerm
        AFun "(" {ATerm ","}+ ")"           -> ATerm
        "[" {ATerm ","}* "]"                -> ATerm
```

Fig. 2.4 Subset of ATerm syntax used in this book.

The `IdCon` and `StrCon` are respectively defined as the following character classes `[A-Za-z][A-Za-z\-]*`[3] and `["]~[\0-\31\n\t\"\\]*["]`.

[3]The original character class for IdCon allows numeric symbols in the tail. These characters are not allowed to prevent ambiguities in the tree path queries presented later on.