Top–down Parsing with Simultaneous Evaluation of Noncircular Attribute Grammars

Thomas Noll
Lehrstuhl für Informatik II
RWTH Aachen
Ahornstraße 55
D–52056 Aachen
Germany
noll@zeus.informatik.rwth-aachen.de

Heiko Vogler
Abt. Theoretische Informatik
Universität Ulm
Oberer Eselsberg
D–89069 Ulm
Germany
vogler@informatik.uni-ulm.de

Aachener Informatik–Berichte Nr. 92–14

Abstract. This paper introduces a machinery called attributed top-down parsing automaton which performs top–down parsing of strings and, simultaneously, the evaluation of arbitrary noncircular attribute grammars. The strategy of the machinery is based on a single depth–first left–to–right traversal over the syntax tree. There is no need to traverse parts of the syntax tree more than once, and hence, the syntax tree itself does not have to be maintained.

Attribute values are stored in a graph component, and values of attributes which are needed but not yet computed are represented by particular nodes. Values of attributes which refer to such uncomputed attributes are represented by trees over operation symbols in which pointers to the particular nodes at their leaves are maintained. Whenever eventually the needed attribute value is computed, it is glued into the graph at the appropriate nodes.

1 Introduction

Attribute grammars are an extension of context–free grammars. They were devised by Knuth in his seminal paper [16, 17] as a formalism to specify the semantics of a context–free language along with its syntax. Since then, attribute grammars were applied by computer scientists in many investigations, but in particular they have proved their appropriateness in the area of compiling programming languages. The reader is referred to [8] for a survey on the theoretical aspects of attribute grammars, for a collection of software systems which are based on attribute grammars, and for an extensive bibliography. In [7] and [4], an overwiew of current research trends in the area of attribute grammars is given. In the sequel we will only consider noncircular attribute grammars.

Considering the transformational approach [11], an attribute grammar is a descriptive device which specifies a transformation from the set of syntax trees of strings which are generated by the underlying context–free grammar G_0, into a set A of semantic values. In order to compute the semantics of a string $w \in L(G_0)$, two steps have to be performed:

(i) the parsing of w according to G_0; this yields a syntax tree s_w of w, and

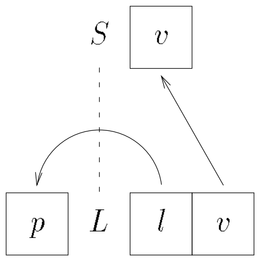
(ii) the evaluation of the designated synthesized attribute σ_0 at the root of s_w.

Then the value of σ_0 at the root of s_w is the semantic value of w. Actually, here we are only interested in the semantic value of w and not in the values of every attribute occurrence in s_w. In the sequel we will restrict ourselves to top–down parsing.

On one hand, many different parsing techniques have been investigated (cf. [1]). On the other hand, attribute evaluation algorithms are known, which coincide to differently powerful subclasses of the whole class of attribute grammars (cf. [10]). Now the question arises whether it is possible to interleave the two steps, i.e., to parse the given input string and to compute its semantic value simultaneously. The advantage of this combination is the possibility of saving storage space, because there is no need to keep the syntax tree in the storage.

On first glance, the combination of parsing and attribute evaluation seems to be impossible because of the following two contrary aspects (called "counter–one–pass features" in [13]). On one hand, top–down parsing of strings determines the syntax trees from the left to the right (and from the root to the leaves). On the other hand, it may happen that the value of an inherited attribute occurrence at a node x of the syntax tree depends on the value of a synthesized attribute occurrence at a node y which is located to the right of x. Thus, the part of the syntax tree starting with y as root is not yet known, and hence, the synthesized attributes of y are not yet computed. Even for $x = y$ these contrary aspects may occur. Let us give an example to illustrate this situation.

Example 1.1 We consider an attribute grammar G which computes the decimal value of a binary numeral. It is a slight modification of an example in [16]. Among others, the underlying context–free grammar contains a start production $S \to L$. The nonterminal symbol L which represents bit lists is associated with three attributes: the inherited attribute p holds the position of the leading bit within the list, assuming position 0 for the rightmost one. The attribute p depends on the synthesized attribute l which denotes the length of the list. The position information is transferred to each of the single bits



$$\langle p, 1\rangle = dec(\langle l, 1\rangle)$$
$$\langle v, \varepsilon\rangle = \langle v, 1\rangle$$

Figure 1: Dependency graph of $S \to L$.

of the list such that their value can be computed individually. Afterwards all values are added up in the value attribute v of L. The situation concerning the start production is illustrated by Figure 1. In the specification of the semantic rules, occurrences of nonterminal symbols are associated with their position within the production. The position of the left hand side symbol is denoted by the empty word ε.

If the top–down parser expands the nonterminal symbol L of the start production $S \to L$, then it has to evaluate the inherited attribute p of L by decrementing the value of the length attribute l of L. But l is not yet known as the parser has not yet built up the subtree with root label L. Hence, this is an instance of the general situation in which both nodes x and y represent the same node of the syntax tree. In Figure 5, a complete syntax tree with attribution is shown. $\diamond$

In general, the problem of required but not yet computed synthesized attributes disappears if the attribute grammar has the *only–S* property, i.e. if it does not contain inherited attributes. The same holds if we consider *L–attributed grammars* [5] only. Roughly speaking, in such grammars the dependencies between occurrences of attributes always show from left to right, and hence the dependencies are compatible with the scanning and parsing of the input string. But L–attributed grammars do not have much expressive power.

Is it possible to combine top–down parsing and attribute evaluation for more powerful subclasses of attribute grammars? At the time being, we know three techniques which answer the question positively, and in each case except the last one, the combination algorithm even applies to arbitrary noncircular attribute grammars.

The first technique [17] solves the combination problem in a very drastical way: Given an attribute grammar G which computes the function f from the set of syntax trees to the set A of semantic values. Construct an attribute grammar G_{id} with one attribute σ only; σ is synthesized and it may take syntax trees as well as elements of A as semantic values. During top–down parsing, at every node x, the syntax tree which corresponds to x is synthesized in σ. At the root of the syntax tree, additionally the function f is applied to the complete syntax tree. Thus, the whole semantic evaluation is shifted into the semantic domain of the attribute grammar by adding f explicitely as a semantic operation. Clearly, this technique has no practical meaning.

The second technique [6] solves the problem in a more realistic way: Let G be given as in the discussion of the first technique. The attribute grammar G' is obtained from G by dropping all the inherited attributes of G. The carrier set A of the semantic domain of G is lifted to the set $Ops(A) = [A^k \to A]$ of operations on A where k is the number of inherited attributes of G. Intuitively, for the synthesized attribute σ and node x, G' computes the function $f_{(\sigma,x)} \in Ops(A)$ which reflects the functional dependency of the value of σ at x from the values of the inherited attributes at x with respect to G. Then, during attribute evaluation by G', the functions (i.e. attribute values) are composed; since G is noncircular, eventually a constant function is computed which represents a value in A. Since G' is an attribute grammar with synthesized attributes only, the combination of parsing and attribute evaluation becomes trivial again. The disadvantage of this technique is the fact that, even if the underlying grammar is L–attributed and hence, attribute dependencies are compatible with parsing, a function is computed for every node x of the parse tree rather than a value.

Finally, the third technique applies to the class of *pseudo–L attribute grammars* as defined in [10]. The attribute evaluation algorithm is based on a depth–first left–to–right traversal over the syntax tree. With respect to the local attribute dependencies, it tries to evaluate as many attribute occurrences as possible. If the algorithm returns to a node y and it has computed the value of a synthesized attribute occurrence at y which was needed for the evaluation of an inherited attribute occurrence at node x and either $x = y$ or x occurs to the left of y, then the algorithm will traverse again the subtree of the syntax tree with root node x. Thus, it it necessary to store (parts of) the syntax tree during attribute evaluation.

In this paper, we introduce a more efficient method of combining top–down parsing and attribute evaluation, and we will develop our solution in two steps. In the first step, for every noncircular attribute grammar, we will construct an attribute evaluation algorithm called $\underline{eval}$, which is compatible with scanning and top–down parsing of input strings. In the second step, we will equip the usual top–down parsing automaton with the facilities needed to evaluate attribute values according to $\underline{eval}$.

More precisely, our attribute evaluation algorithm $\underline{eval}$ takes the syntax tree s_w of a given string w as input, and it performs a single depth–first left–to–right traversal over s_w. In contrast to the approach of pseudo–L attribute grammars, the evaluator computes a value for *every* attribute occurrence at the current node. Clearly, if an inherited attribute occurrence $\langle\iota, x\rangle$ at node x depends on a synthesized attribute occurrence $\langle\sigma, y\rangle$ at node y, and if $x = y$ or y occurs to the right of x, then the value of $\langle\iota, x\rangle$ can only be an approximation $t_{\langle\iota,x\rangle}$ of its final value (cf. Figure 2). We will call such intermediate values *schematic approximations*, because they are represented by trees over the set Ω of operation symbols and the set of attribute occurrences of s_w viewed as nullary symbols. (In Figure 1, the algorithm computes on first visit to the first son of S the schematic approximation $dec(\langle l, 1\rangle)$ as value for the attribute occurrence $\langle p, 1\rangle$.)

Now assume that $\underline{eval}$ has returned to node x having parsed the frontier of the subtree $sub(s_w, x)$ of s_w with root node x and having computed a value $t_{\langle\sigma',x\rangle}$ for the attribute occurrence $\langle\sigma', x\rangle$. If we assume that with respect to $sub(s_w, x)$, $\langle\sigma', x\rangle$ depends on $\langle\iota, x\rangle$, then $t_{\langle\sigma',x\rangle}$ is a schematic approximation which contains the attribute occurrence $\langle\sigma, y\rangle$. Next, $\underline{eval}$ visits the younger brothers of x and eventually it visits the node y. After having parsed the subtree $sub(s_w, y)$ and having computed a value $t_{\langle\sigma,y\rangle}$ for the synthesized attribute occurrence $\langle\sigma, y\rangle$, $\underline{eval}$ can refine the approximations of attribute occurrences to the left. In particular, it can refine the schematic approximation of $\langle\sigma', x\rangle$ to be the tree $t_{\langle\sigma',x\rangle}[\langle\sigma, y\rangle/t_{\langle\sigma,y\rangle}]$ which is obtained from $t_{\langle\sigma',x\rangle}$ by replacing every occurrence of $\langle\sigma, y\rangle$ by $t_{\langle\sigma,y\rangle}$.

This refinement may lead to a ground term over the set Ω of operation symbols;

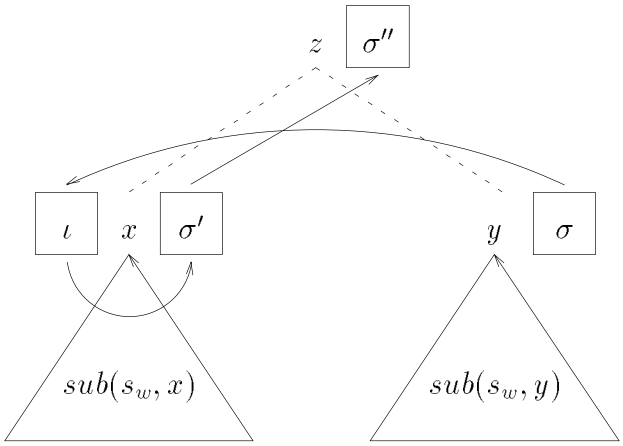


Figure 2: General example.

by applying the unique homomorphism $h : \mathcal{T}_\Omega \to \mathcal{A}$ from the initial term algebra $\mathcal{T}_\Omega$ to the semantic domain $\mathcal{A}$ of the attribute grammar, values in the carrier set A of $\mathcal{A}$ are obtained. If, however, $t_{\langle\sigma,y\rangle}$ is also just a schematic approximation, then the result of the substitution is again a schematic approximation. But eventually, at the root of s_w, ground terms are computed, because the attribute grammar is noncircular. We note that, if the attribute grammar is L, i.e. all dependencies show from left to right, then our attribute evaluation algorithm will compute immediately, for every attribute occurrence, its final value in $\mathcal{A}$. We also note that every subtree has to be visited at most once (in fact, exactly once). Thus, our approach is more efficient than the second and third technique discussed above.

We note that in [21] a very similar approach has been suggested; this is discussed in Section 7.

In the second step of our development, we construct a machinery which performs both parsing of a given input string w and the computation of the semantic value of w according to the evaluation algorithm constructed in the first step. The machinery, called *attributed top–down parsing automaton*, works deterministically for context–free grammars which are LL(k). In this paper we will restrict ourselves to $k = 1$, but the technique can be extended straightforwardly to any other k. The attributed top–down parsing automaton is an extension of the usual top–down parsing automaton and it is similar to the attributed pushdown machine [18]; however, the main additional component is a graph storage in which schematic approximations, ground terms, and semantic values of $\mathcal{A}$ are stored and updated. In order not to be bothered with pure evaluations in the semantic domain $\mathcal{A}$ of the attribute grammar and, in particular, with the transformations of ground terms into values of $\mathcal{A}$ by means of the unique homomorphism h, we will consider in our investigation attribute grammars only which have the initial term algebra $\mathcal{T}_\Omega$ as semantic domain.

How does the automaton store and update schematic approximations? Whenever the automaton creates a new schematic approximation t with an attribute occurrence $\langle\sigma, y\rangle$ in its frontier, an additional application node is created which contains a pointer to the graph representation of t and a pointer pt to the leaf labeled by $\langle\sigma, y\rangle$. In fact, using the well–known sharing technique, it suffices to represent $\langle\sigma, y\rangle$ only once (cf.

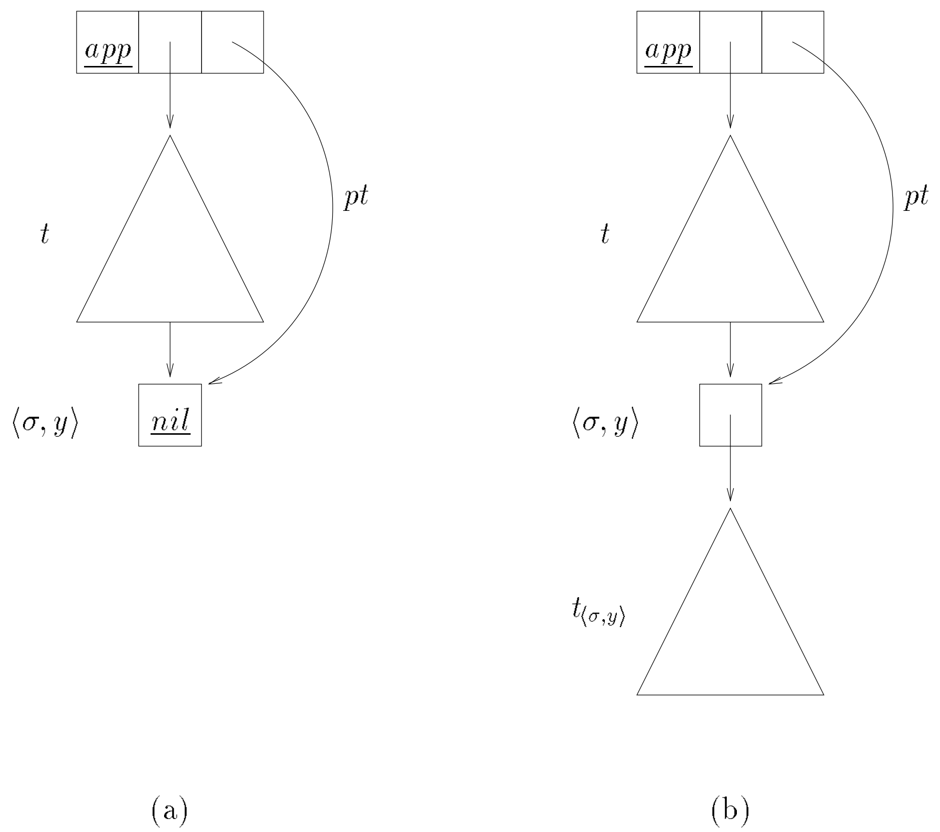


Figure 3: Representation and refinement of schematic approximations.

Figure 3(a)). If in a later stage of the automaton, a schematic approximation $t_{\langle\sigma,y\rangle}$ of the value of $\langle\sigma, y\rangle$ is computed, then the automaton just stores the address of the root of $t_{\langle\sigma,y\rangle}$ into the node referenced by pt (cf. Figure 3(b)).

Since the attribute evaluation algorithm which is implemented in the attributed top–down parsing automaton, computes a schematic approximation for every attribute occurrence at the current node, and since pointers to needed, but not yet computed attribute occurrences are maintained until refinements are computed, there is no need to call the attribute evaluator more than once at any node of the syntax tree s_w. Thus, the automaton does not have to store parts of s_w.

In fact, we have constructed the attributed top–down parsing automaton in a formal style (although we are not going to prove the correctness of this automaton with respect to the combination problem). The reason for having a formal construction is the fact that this has opened up a direct and obvious way for the systematic development of a test implementation. In Section 6 this test implementation is discussed briefly.

This paper is organized as follows. Section 2 provides the basic notions of context–free grammars, pushdown automata, and top–down parsing automata for LL(1) grammars. Although these topics are rather standard, we would like to advice the reader to glance at least at Section 2.2. The reason is that we present the top–down parsing automaton in a formalism which is slightly different from the usual one but more appropriate for an extension to attribute evaluation. In Section 3, we collect the definitions concerning attribute grammars. In Section 4, we introduce our attribute evaluation al-

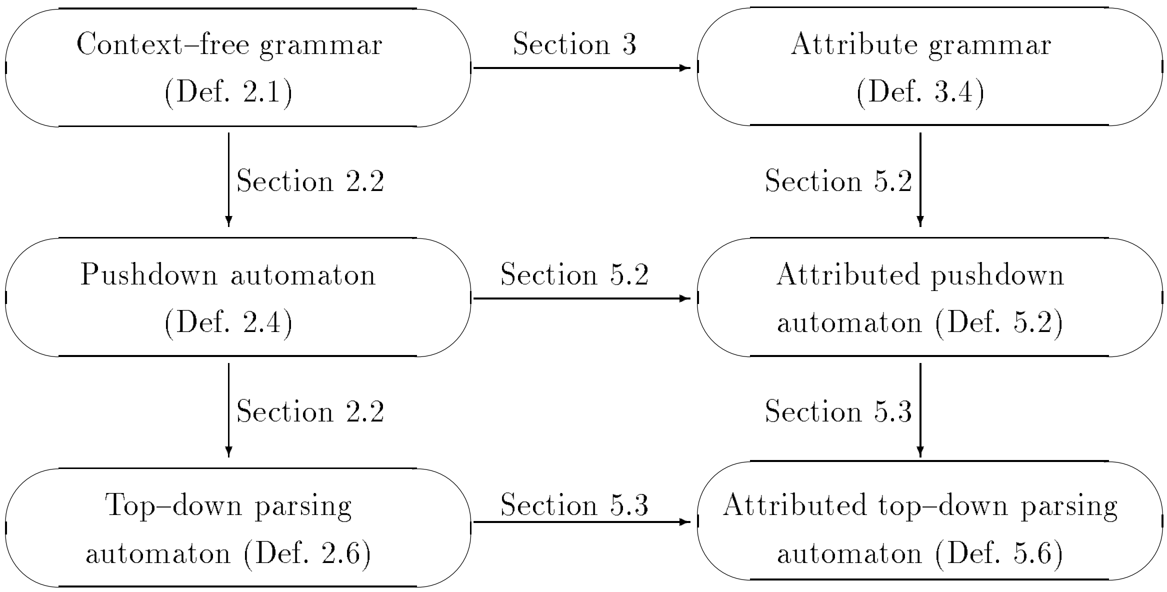


Figure 4: Survey.

gorithm. In Section 5 we extend the concept of pushdown automaton to the concept of attributed pushdown automaton. In the same way as top–down parsing automata are special instances of pushdown automata, we instantiate attributed pushdown automata to attributed top–down parsing automata. Figure 4 gives a survey of the interrelations. In Section 6 we discuss our test implementation of the attributed top–down parsing automaton. Finally, in Section 7 we take a look at the problem of combining parsing and attribute evaluation from the point of view of logic programming and discuss the relationship to our approach.

2 Context–free grammars and top–down parsing

2.1 Context–free grammars

Context–free grammars play a major rôle in the description of formal languages. They supply the syntactic base of the attribute grammar formalism. We introduce some basic concepts following mainly [1].

Definition 2.1 (Context–free grammar)
A context–free grammar

$$G_0 = (N, \Sigma, \pi, P, S)$$

consists of a *nonterminal alphabet* N, a *terminal alphabet* Σ disjoint from N, a bijective mapping $\pi : \{1, \ldots, |P|\} \rightarrow P$, a finite set $P \subset \{A \rightarrow \alpha \mid A \in N, \alpha \in (N \cup \Sigma)^*\}$ of *productions* and a designated *start symbol* $S \in N$. G_0 is called *reduced* if for each $A \in N$ there exist $\alpha, \beta \in (N \cup \Sigma)^*$ and $w \in \Sigma^*$ such that $S \Rightarrow^* \alpha\, A\, \beta \Rightarrow^* w$ where $\Rightarrow$ denotes the derivation relation of G_0. G_0 is called *start–separated* if the start symbol does not appear on the right hand side of any production. $L(G_0) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$

denotes the *language generated by* G_0. A formal language which is generated by a context–free grammar is called *context–free*. Two context–free grammars G_0 and G_1 are called *equivalent* if $L(G_0) = L(G_1)$. $\diamond$

In the sequel we assume that context–free grammars are reduced and start–separated. This can be achieved by the usual transformations.

We will use trees to represent derivations of context–free grammars. Tree nodes are specified by means of the well–known Dewey notation, i.e. a tree node x is a string $i_1.i_2 \ldots i_n$ with $i_j > 0$. Intuitively, the Dewey notation of a node x indicates the path from the root of the tree to x. Thus, the root itself is denoted by ε.

Definition 2.2 (Syntax tree)
Let $G_0 = (N, \Sigma, \pi, P, S)$ be a context–free grammar. A *syntax tree* of G_0 is a finite tree s whose nodes are labeled by symbols from $N \cup \Sigma$ such that the following conditions hold: The root ε of s is labeled by S, and for each inner node x there is a production $p = X_0 \to X_1 \ldots X_n \in P$ ($X_0 \in N$, $X_i \in N \cup \Sigma$) such that x is labeled by X_0, has n successors $x.1, \ldots, x.n$, and for every $i \in \{1, \ldots, n\}$, $x.i$ is labeled by X_i. In this case we say that p *applies at* x. The set of all syntax trees of G_0 is denoted by T_{G_0}. $\diamond$

2.2 Top–down parsing

The *parser* is the compiler part which is dedicated to the syntactic analysis of the token stream received from the *scanner*. This process is also called *parsing*. Parsing is done by pushdown automata. Each context–free language can be analyzed by a pushdown automaton with one state only. This statement is justified by the following informal construction of a pushdown automaton which supplies the foundation of both (nondeterministic) *top–down parsing* and deterministic *LL parsing* .

Let $G_0 = (N, \Sigma, \pi, P, S)$ be a context–free grammar. Note that G_0 is supposed to be reduced and start–separated. If a nonterminal X_0 lies on top of the pushdown, then the parsing automaton nondeterministically selects a production $p = X_0 \to X_1 \ldots X_n \in P$, pops X_0 from the pushdown, and pushes the symbols $X_n, \ldots, X_1$ one by one. This transition simulates the application of p and is called *expansion* of p. If a terminal symbol a lies on top of the pushdown, then it is compared with the next symbol on the input tape. If both symbols correspond, then a is popped from the pushdown. This transition is called *match transition*. If the pushdown is empty, then the computation stops.

As we can see immediately, this parsing method realizes a depth–first left–to–right traversal of the (virtual) syntax tree. But it has some additional properties which will turn out to be disadvantageous in connexion with evaluation of attribute grammars:

- It works nondeterministically.
- When expanding a production, its right hand side is pushed symbol by symbol. Thus, the pushdown gives no explicit information about which production is analyzed at this moment.
- In particular, the complete recognition of its right hand side cannot be realized.

We will solve these problems in the following way (also cf. the construction in Lemma 6.1 of [12]):

- We will restrict the class of context–free grammars which can be handled such that deterministic parsing is possible. Here we choose LL(1) grammars.
- In the pushdown, we do not store nonterminals and terminals, but we store LR(0)–items. These are known from *bottom–up* (or: *LR*) parsing and they contain the required information:
  - the identification of the production which is analyzed at this moment,
  - the specification of the suffix of its right hand side which has not yet been parsed, and thus, in particular,
  - the information about complete recognition of its right hand side.

Definition 2.3 (LR(0) item)
Let $G_0 = (N, \Sigma, \pi, P, S)$ be a context–free grammar. For any production $A \to \alpha\,\alpha' \in P$, $[A \to \alpha.\alpha']$ is called an *LR(0) item* of G_0. The set of all LR(0) items of G_0 is denoted by $LR(0)(G_0)$. $\diamond$

Obviously, $LR(0)(G_0)$ is finite. An LR(0) item $[A \to \alpha.\alpha'] \in LR(0)(G_0)$ on top of the pushdown of the top–down parsing automaton has the following meaning:

- Production $A \to \alpha\,\alpha' \in P$ is currently analyzed.
- The part of the input which was derived from the sentential form α, has already been accepted. If $\alpha = \varepsilon$, then the previously executed transition was an expansion.
- A prefix of the current input has to be parsed according to the sentential form α'. If $\alpha' = \varepsilon$, the production $A \to \alpha$ has been completely recognized.

The choice of the transition which the automaton has to execute next, is essentially determined by the LR(0) item $[A \to \alpha.\alpha'] \in LR(0)(G_0)$ on top of the pushdown:

- If $\alpha' = B\,\alpha''$ with $B \in N$, then the automaton selects an appropriate production $B \to \beta$. Thereafter, it puts the corresponding LR(0) item $[B \to .\beta]$ on top of the pushdown by means of a <u>*push*</u> operation.
- If $\alpha' = a\,\alpha''$ with $a \in \Sigma$, then the automaton compares the terminal symbol a to the next input symbol. If both correspond, then the LR(0) item on top of the pushdown is modified to $[A \to \alpha\,a.\alpha'']$ by a <u>*mod*</u> operation.
- If $\alpha' = \varepsilon$, a <u>*pop*</u> operation both removes the LR(0) item $[A \to \alpha.]$ on top of the pushdown and changes the item $[B \to \beta.A\,\beta']$ below to $[B \to \beta\,A.\beta']$. (For this reason, the transition function has to take notice of the upper two pushdown entries.)

After this informal introduction, we describe in greater detail the concepts of pushdown automaton, LL(1) grammar, and top–down parsing automaton. (Recall the overview in Figure 4.)

The pushdown automaton presented in the following definition is able to read the topmost two symbols of the pushdown. Moreover, the <u>*pop*</u> operation performs the deletion of the topmost symbol and, afterwards, it modifies the current topmost symbol. It is obvious how to construct for a pushdown automaton of our type an equivalent pushdown automaton of the usual type.

Definition 2.4 (Pushdown automaton)
A *pushdown automaton*

$$\mathcal{A}_0 = (Q, \Sigma, \Gamma, \delta, q^0, \gamma_0, F)$$

consists of a finite set Q of *states*, an *input alphabet* Σ, a *pushdown alphabet* Γ, a *transition function* $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^2 \to \wp(Q \times \{\underline{mod}, \underline{pop}, \underline{push}\} \times \Gamma)$ (where $\wp$ denotes the power set operator), a *start state* $q^0 \in Q$, a *pushdown bottom symbol* $\gamma_0 \in \Gamma$ and a subset $F \subset Q$ of *final states*. The set of *instantaneous descriptions* of $\mathcal{A}_0$ is the cartesian product

$$ID_{\mathcal{A}_0} = Q \times \Sigma^* \times \Gamma^*.$$

The *transition relation* of $\mathcal{A}_0$

$$\vdash_{\mathcal{A}_0} \subset ID_{\mathcal{A}_0} \times ID_{\mathcal{A}_0}$$

is given by: If $(q', op, \gamma') \in \delta(q, x, \gamma_1\gamma_2)$ with $q, q' \in Q$, $x \in \Sigma \cup \{\varepsilon\}$, $\gamma_1, \gamma_2, \gamma' \in \Gamma$ and $op \in \{\underline{mod}, \underline{pop}, \underline{push}\}$, then for every $w \in \Sigma^*$, and every $\xi \in \Gamma^*$

$$(q, x\,w, \gamma_1\gamma_2\xi) \vdash_{\mathcal{A}_0} (q', w, \xi')$$

where

$$\xi' = \begin{cases} \gamma'\gamma_2\xi & \text{if } op = \underline{mod} \\ \gamma'\xi & \text{if } op = \underline{pop} \\ \gamma'\gamma_1\gamma_2\xi & \text{if } op = \underline{push}. \end{cases}$$

The *language accepted by* $\mathcal{A}_0$ is the set

$$L(\mathcal{A}_0) = \{w \in \Sigma^* \mid \text{there are } q^f \in F, \gamma \in \Gamma \text{ such that } (q^0, w, \gamma_0\gamma_0) \vdash^*_{\mathcal{A}_0} (q^f, \varepsilon, \gamma)\}.$$

A pushdown automaton $\mathcal{A}_0$ is called *deterministic* if for every $q \in Q$, and every $\gamma_1, \gamma_2 \in \Gamma$ either

(i) $|\delta(q, \varepsilon, \gamma_1\gamma_2)| = 0$, and for every $a \in \Sigma : |\delta(q, a, \gamma_1\gamma_2)| \leq 1$ or

(ii) $|\delta(q, \varepsilon, \gamma_1\gamma_2)| = 1$, and for every $a \in \Sigma : |\delta(q, a, \gamma_1\gamma_2)| = 0$. $\diamond$

Given a context–free grammar G_0, we now want to construct a deterministic pushdown automaton which accepts $L(G_0)$, called top–down parsing automaton. Its determinism will be achieved by giving him the capability to look ahead one character on the input tape. Because the input alphabet is finite, the corresponding informations can be stored in the finite control of the automaton. Furthermore, we append a special end marker \$ to every input string. The class of context–free grammars which can be parsed top–down in a deterministic way under these assumptions is well known as LL(1).

Definition 2.5 (LL(1) grammar)
Let $G_0 = (N, \Sigma, \pi, P, S)$ be a context–free grammar. To each production $p = A \to \alpha \in P$, we assign the *look–ahead set*

$$\begin{aligned} \underline{la}(p) \;=\; & \{x \in \Sigma \cup \{\$\} \mid \text{there are } w \in \Sigma^*, \beta, \beta' \in (N \cup \Sigma \cup \{\$\})^* \text{ such that} \\ & S\,\$ \Rightarrow^*_l w\,A\beta \Rightarrow_l w\,\alpha\,\beta \Rightarrow^*_l w\,x\,\beta'\} \end{aligned}$$

where $\Rightarrow_l$ denotes the leftmost derivation relation of G_0 in which at every step the leftmost nonterminal symbol of a sentential form is derived. G_0 is called an *LL(1)* *grammar* if for every nonterminal $A \in N$, and for every pair of productions $A \to \alpha$ and $A \to \beta$ in P the following condition is true:

$$\underline{la}(A \to \alpha) \cap \underline{la}(A \to \beta) = \emptyset.$$

The set of all LL(1) grammars is denoted by $LL(1)$. $\diamond$

There are algorithms which try to transform context–free grammars into equivalent LL(1) grammars such as *elimination of left recursion* and *left factoring* (cf. [1]). However, it is well known that there are deterministic context–free languages which cannot be generated by an LL(k) grammar for arbitrary $k \in \mathbb{N}$ where $\mathbb{N}$ denotes the set of natural numbers including zero. Moreover, one has to keep in mind that such transformations preserve the equivalence of grammars but not the syntactic structure of the generated strings upon which their semantics is defined. Thus, similar transformations of attribute grammars are required. In [2], left factoring is applied to attribute grammars. In [1], elimination of left recursion in attribute grammars is discussed.

Now, we formalize the construction of the top–down parsing automaton of G_0. Later this automaton will be extended to the attributed top–down parsing automaton for an attribute grammar with G_0 as underlying context–free grammar (cf. Definition 5.6).

Definition 2.6 (Top–down parsing automaton)
Let $G_0 = (N, \Sigma, \pi, P, S)$ be a context–free grammar. The *top–down parsing automaton* of G_0 is the pushdown automaton

$$TDA(G_0) = (Q, \Delta, \Gamma, \delta, q^0, \gamma_0, F)$$

where:

- $Q = \{q^0, q^f\} \cup \{q_a \mid a \in \Sigma \cup \{\$\}\}$,
- $\Delta = \Sigma \cup \{\$\}$,
- $\Gamma = LR(0)(G_0) \cup \{[\to .S], [\to S.]\}$,
- $\delta : Q \times (\Sigma \cup \{\$, \varepsilon\}) \times \Gamma^2 \to \wp(Q \times \{\underline{mod}, \underline{pop}, \underline{push}\} \times \Gamma)$ where
  - (i) Initiation of look–ahead:
    $\delta(q^0, a, [\to .S][\to .S]) = \{(q_a, \underline{mod}, [\to .S])\}$
    for every $a \in \Sigma \cup \{\$\}$
  - (ii) Expansion of start productions:
    $\delta(q_a, \varepsilon, [\to .S][\to .S]) \ni (q_a, \underline{push}, [S \to .\alpha])$
    for every $S \to \alpha \in P$, and $a \in \underline{la}(S \to \alpha)$
  - (iii) Expansion of non–start productions:
    $\delta(q_a, \varepsilon, [A \to \alpha.B\, \alpha']\gamma) \ni (q_a, \underline{push}, [B \to .\beta])$
    for every $B \in N$, $A \to \alpha\, B\, \alpha', B \to \beta \in P$, $\gamma \in \Gamma$, and $a \in \underline{la}(B \to \beta)$
  - (iv) Terminal symbol match:
    $\delta(q_a, b, [A \to \alpha.a\, \alpha']\gamma) = \{(q_b, \underline{mod}, [A \to \alpha\, a.\alpha'])\}$
    for every $a \in \Sigma$, $A \to \alpha\, a\, \alpha' \in P$, $\gamma \in \Gamma$, and $b \in \Sigma \cup \{\$\}$
  - (v) Reduction of non–start productions:
    $\delta(q_a, \varepsilon, [B \to \beta.][A \to \alpha.B\, \alpha']) = \{(q_a, \underline{pop}, [A \to \alpha\, B.\alpha'])\}$
    for every $a \in \Sigma \cup \{\$\}$, and $B \to \beta, A \to \alpha\, B\, \alpha' \in P$

(vi) Reduction of start productions:
$\delta(q_\$, \varepsilon, [S \to \alpha.][\to .S]) = \{(q_\$, \underline{pop}, [\to S.])\}$
for every $S \to \alpha \in P$

(vii) Final transition:
$\delta(q_\$, \varepsilon, [\to S.][\to .S]) = \{(q^f, \underline{pop}, [\to S.])\}$

(viii) In all remaining cases:
$\delta(q, x, \gamma_1\gamma_2) = \emptyset$

- $\gamma_0 = [\to .S]$ and
- $F = \{q^f\}$. $\diamond$

The following propositions are well known from the theory of LL grammars.

Lemma 2.1 For every context–free grammar G_0, $TDA(G_0)$ is deterministic iff G_0 is an LL(1) grammar. $\diamond$

Theorem 2.2 For every context–free grammar G_0, the following equivalence holds: $w \in L(G_0)$ iff $w\$ \in L(TDA(G_0))$. $\diamond$

Example 2.1 The attribute grammar which has been sketched in Section 1 is based on the following LL(1) grammar:

$$\begin{aligned} G_0 &= (\{S, L, B\}, \{0, 1\}, \pi, \{\pi_1 : S \to L, \pi_2 : L \to B\, L, \pi_3 : L \to \varepsilon, \\ &\quad \pi_4 : B \to 0, \pi_5 : B \to 1\}, S). \end{aligned}$$

Its top–down parsing automaton is given by

$$TDA(G_0) = (Q, \Delta, \Gamma, \delta, q^0, \gamma_0, F)$$

with the components

- $Q = \{q^0, q^f, q_0, q_1, q_\$\}$,
- $\Delta = \{0, 1, \$\}$,
- $\Gamma = \{[S \to .L], [S \to L.], \ldots, [B \to 1.], [\to .S], [\to S.]\}$,
- $\delta : Q \times \{0, 1, \$, \varepsilon\} \times \Gamma^2 \to \wp(Q \times \{\underline{mod}, \underline{pop}, \underline{push}\} \times \Gamma)$ where

(i) Initiation of look–ahead:
$\delta(q^0, a, [\to .S][\to .S]) \;=\; \{(q_a, \underline{mod}, [\to .S])\}$
for every $a \in \{0, 1, \$\}$

(ii) Expansion of start productions:
$\delta(q_a, \varepsilon, [\to .S][\to .S]) \;=\; \{(q_a, \underline{push}, [S \to .L])\}$
for every $a \in \{0, 1, \$\}$

(iii) Expansion of non–start productions:

$$\begin{aligned} \delta(q_a, \varepsilon, [S \to .L]\gamma) &= \{(q_a, \underline{push}, [L \to .B\, L])\} \\ \delta(q_\$, \varepsilon, [S \to .L]\gamma) &= \{(q_\$, \underline{push}, [L \to .])\} \\ \delta(q_a, \varepsilon, [L \to .B\, L]\gamma) &= \{(q_a, \underline{push}, [B \to .a])\} \\ \delta(q_a, \varepsilon, [L \to B.L]\gamma) &= \{(q_a, \underline{push}, [L \to .B\, L])\} \\ \delta(q_\$, \varepsilon, [L \to B.L]\gamma) &= \{(q_\$, \underline{push}, [L \to .])\} \end{aligned}$$

for every $a \in \{0, 1\}$, and $\gamma \in \Gamma$

(iv) Terminal symbol match:

$$\delta(q_0, a, [B \to .0]\gamma) \;=\; \{(q_a, \underline{mod}, [B \to 0.])\}$$
$$\delta(q_1, a, [B \to .1]\gamma) \;=\; \{(q_a, \underline{mod}, [B \to 1.])\}$$

for every $a \in \{0, 1, \$\}$, and $\gamma \in \Gamma$

(v) Reduction of non–start productions:

$$\begin{aligned}
\delta(q_a, \varepsilon, [L \to B\,L.][S \to .L]) &= \{(q_a, \underline{pop}, [S \to L.])\}\\
\delta(q_a, \varepsilon, [L \to B\,L.][L \to B.L]) &= \{(q_a, \underline{pop}, [L \to B\,L.])\}\\
\delta(q_a, \varepsilon, [L \to .][S \to .L]) &= \{(q_a, \underline{pop}, [S \to L.])\}\\
\delta(q_a, \varepsilon, [L \to .][L \to B.L]) &= \{(q_a, \underline{pop}, [L \to B\,L.])\}\\
\delta(q_a, \varepsilon, [B \to 0.][L \to .B\,L]) &= \{(q_a, \underline{pop}, [L \to B.L])\}\\
\delta(q_a, \varepsilon, [B \to 1.][L \to .B\,L]) &= \{(q_a, \underline{pop}, [L \to B.L])\}
\end{aligned}$$

for every $a \in \{0, 1, \$\}$

(vi) Reduction of start productions:

$$\delta(q_\$, \varepsilon, [S \to L.][\to .S]) \;=\; \{(q_\$, \underline{pop}, [\to S.])\}$$

(vii) Final transition:

$$\delta(q_\$, \varepsilon, [\to S.][\to .S]) \;=\; \{(q^f, \underline{pop}, [\to S.])\}$$

(viii) In all remaining cases:

$$\delta(q, x, \gamma_1\gamma_2) \;=\; \emptyset$$

- $\gamma_0 = [\to .S]$ and
- $F = \{q^f\}$. $\diamond$

3 Attribute grammars

This section is dedicated to the definition of attribute grammars and their semantics. First of all, we introduce basic notions from universal algebra which, together with the context–free grammar, supply the foundation of attribute grammars.

3.1 Universal algebra

In the scope of our paper it suffices to consider only homogeneous, i.e. single–sorted, algebras.

Definition 3.1 (Algebra)
A *set of operation symbols* is a (possibly infinite) countable set Ω in which with every symbol $f \in \Omega$ a natural number is associated. This number is called the *arity* of f. For every $n \in \mathbb{N}$, $\Omega^{(n)}$ denotes the set of all symbols of arity n; the relationship $f \in \Omega^{(n)}$ is indicated by $f^{(n)}$. For every set A and for every $n \in \mathbb{N}$, $Ops^{(n)}(A) = \{f \mid f : A^n \to A\}$ denotes the set of all *operations of arity* n on A; we abbreviate $\bigcup_{n\in\mathbb{N}} Ops^{(n)}(A)$ by $Ops(A)$. Moreover, if $\varphi : \Omega \to Ops(A)$ such that $\varphi(\Omega^{(n)}) \subset Ops^{(n)}(A)$, then $\mathcal{A} = (A, \varphi)$ is called an Ω*–algebra* with *carrier set* A and *interpretation* φ. $\diamond$

Definition 3.2 (Homomorphism)
Let Ω be a set of operation symbols, and let $\mathcal{A} = (A, \varphi)$ and $\mathcal{B} = (B, \psi)$ be two Ω–algebras. A mapping $h : A \to B$ is called a *homomorphism* if for every $n \in \mathbb{N}$, $f \in \Omega^{(n)}$, and $a_1, \ldots, a_n \in A$, the equation

$$h(\varphi(f)(a_1, \ldots, a_n)) = \psi(f)(h(a_1), \ldots, h(a_n))$$

holds. We also write $h : \mathcal{A} \to \mathcal{B}$. If $n = 0$, then the above equation reduces to $h(\varphi(f)) = \psi(f)$. $\diamond$

Definition 3.3 (Term algebra)
For every set of operation symbols Ω and for every (arbitrary) set U, $T_\Omega(U)$ denotes the set of all finite, well–formed *Ω–terms* (*Ω–trees*) in which leaves can be labeled by elements of U. Let X be a countable set of *variables*. The *Ω–term algebra $\mathcal{T}_\Omega(X)$ generated by X* is the algebra

$$\mathcal{T}_\Omega(X) = (T_\Omega(X), \varphi_T)$$

where

$$\varphi_T(f)(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$$

for every $n \in \mathbb{N}$, $f \in \Omega^{(n)}$, and $t_1, \ldots, t_n \in T_\Omega(X)$. $\mathcal{T}_\Omega(X)$ is freely generated by X, i.e. for every Ω–algebra $\mathcal{A} = (A, \varphi)$ and every assignment $\underline{val} : X \to A$, there is exactly one homomorphism $\widehat{\underline{val}} : \mathcal{T}_\Omega(X) \to \mathcal{A}$ such that $\widehat{\underline{val}}|_X = \underline{val}$. This property uniquely determines $\mathcal{T}_\Omega(\emptyset)$ up to isomorphism. $\mathcal{T}_\Omega(\emptyset)$ is called *initial* in the class of Ω–algebras, and it is denoted by

$$\mathcal{T}_\Omega = (T_\Omega, \varphi_T).$$

Given an Ω–algebra $\mathcal{A} = (A, \varphi)$, an assignment $\underline{val} : X \to A$, and a term $t \in T_\Omega(X)$. The *argument list* $\underline{arg}(t)$ is the duplicate–free list of variables which occur in a left–to–right scan over the frontier of t, i.e. $\underline{arg} : T_\Omega(X) \to X^*$ is given by

$$\underline{arg} = \underline{nodup} \circ \underline{var},$$

where $\underline{var} : T_\Omega(X) \to X^*$ and $\underline{nodup} : X^* \to X^*$ are defined inductively by

$$\underline{var}(t) = \begin{cases} x & \text{if } t = x \in X \\ \underline{var}(t_1) \ldots \underline{var}(t_n) & \text{if } t = f(t_1, \ldots, t_n), n \in \mathbb{N}, \end{cases}$$

and

$$\underline{nodup}(w) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \underline{nodup}(w')\, x & \text{if } w = w'\, x \text{ and } x \text{ does not occur in } w' \\ \underline{nodup}(w') & \text{if } w = w'\, x \text{ and } x \text{ occurs in } w'. \end{cases}$$

Let $t \in T_\Omega(X)$ and $\underline{arg}(t) = x_1 \ldots x_n$. Then the mapping $\underline{derop}(t) : A^n \to A$, called the *derived operation* of t in $\mathcal{A}$, is defined as follows: For every $a_1, \ldots, a_n \in A$, $\underline{derop}(t)(a_1, \ldots, a_n) = \widehat{\underline{val}}(t)$ where $\underline{val} : X \to A$ with $\underline{val}(x_i) = a_i$ for every $i \in \{1, \ldots, n\}$. $\diamond$

3.2 Definition of attribute grammars

Now we extend context–free grammars to attribute grammars in two steps. First, we augment them by adding an attribute scheme which specifies attributes and semantic rules. Second, we add an appropriate algebra in which the operation symbols occurring in the attribute scheme can be interpreted. Recall that we only consider reduced and start–separated context–free grammars.

Definition 3.4 (Attribute grammar)
Let $G_0 = (N, \Sigma, \pi, P, S)$ be a context–free grammar, and let Ω be a set of operation symbols. Let Inh and Syn be two finite, disjoint sets, and let $Att = Inh \cup Syn$. With each nonterminal symbol $A \in N$, there is associated a set $\underline{att}(A)$ of *attributes* which is partitioned into two sets $\underline{inh}(A) \subset Inh$ and $\underline{syn}(A) \subset Syn$ of *inherited* and *synthesized attributes*, respectively. Let the start symbol S have a designated *meaning attribute* $\sigma_0 \in \underline{syn}(S)$ and no inherited attributes, i.e. $\underline{inh}(S) = \emptyset$. To every production $p = A_0 \to w_0\, A_1\, w_1 \ldots A_n\, w_n \in P$ with $A_0, A_1, \ldots, A_n \in N$ and $w_0, \ldots, w_n \in \Sigma^*$, we assign the set of *inside attributes occurrences*

$$\underline{in}(p) = \{\langle \vartheta, i\rangle \mid \vartheta \in \underline{syn}(A_0), i = \varepsilon \text{ or } \vartheta \in \underline{inh}(A_i), 1 \leq i \leq n\}$$

and the set of *outside attributes occurrences*

$$\underline{out}(p) = \{\langle \vartheta, i\rangle \mid \vartheta \in \underline{inh}(A_0), i = \varepsilon \text{ or } \vartheta \in \underline{syn}(A_i), 1 \leq i \leq n\}.$$

For abbreviation, we define $\underline{att}(p) = \underline{in}(p) \cup \underline{out}(p)$ to be the set of *attribute occurrences* of p, and we define $\underline{att}(P) = \bigcup_{p \in P} \underline{att}(p)$. For each production $p \in P$, let $R_p : \underline{in}(p) \to T_\Omega(\underline{out}(p))$ be a mapping which assigns to every inside attribute a *semantic rule* (more exactly, the right hand side of a semantic rule). Furthermore, let $R = (R_p)_{p \in P}$. Then we call the tuple

$$\mathcal{B} = (\Omega, \underline{inh}, \underline{syn}, \sigma_0, R)$$

an *attribute scheme* for G_0. If $\mathcal{A}$ is an Ω–algebra, then

$$G = (G_0, \mathcal{B}, \mathcal{A})$$

is called an *attribute grammar with underlying context–free grammar* G_0. $\diamond$

The notion of inside and outside attributes is accepted from [10]. In our definition of attribute grammars, semantic rules determine inside attribute values in terms of outside attribute values, establishing a partition of the attribute occurrences and thus preventing circular attribute dependencies within single productions. This property is sometimes called *(Bochmann) normal form* and can be presumed without loss of generality, as it is shown in [11].

The semantics of an attribute grammar can be defined as follows. For every terminal string w of the underlying context–free grammar, we construct its syntax tree s_w, assign a storage cell to every occurrence of an attribute at every node of s_w, and apply the semantic rules within the context of this tree. Remember that we represent tree nodes using the Dewey notation.

Definition 3.5 (Attributed syntax tree)
Let $G = (G_0, \mathcal{B}, \mathcal{A})$ be an attribute grammar with underlying context–free grammar $G_0 = (N, \Sigma, \pi, P, S)$, attribute scheme $\mathcal{B} = (\Omega, \underline{inh}, \underline{syn}, \sigma_0, R)$, and Ω–algebra $\mathcal{A} = (A, \varphi)$. Let $s \in T_{G_0}$ be a syntax tree of G_0, and let x be a node of s labeled by $A \in N$. The sets $\underline{inh}(x) = \{\langle \iota, x\rangle \mid \iota \in \underline{inh}(A)\}$ and $\underline{syn}(x) = \{\langle \sigma, x\rangle \mid \sigma \in \underline{syn}(A)\}$ are called *set of inherited attribute occurrences* and *set of synthesized attribute occurrences* of x, respectively. The sets of *attribute occurrences* of x and of s are abbreviated by $\underline{att}(x) = \underline{inh}(x) \cup \underline{syn}(x)$ and $\underline{att}(s) = \bigcup\{\underline{att}(y) | y \text{ node of } s\}$, respectively. Assume that production $p \in P$ applies at x and that $\langle \vartheta, i\rangle \in \underline{in}(p)$ is an inside attribute occurrence

of p. Then $\langle\vartheta, x.i\rangle = R_s(\langle\vartheta, x.i\rangle)$ is the *semantic equation* for $\langle\vartheta, x.i\rangle \in \underline{att}(s)$ where $R_s(\langle\vartheta, x.i\rangle)$ is obtained from $R_p(\langle\vartheta, i\rangle)$ by replacing every attribute occurrence $\langle\theta, j\rangle$ of p by the corresponding attribute occurrence $\langle\theta, x.j\rangle$ of s. An *attribution* is a mapping $\underline{val} : \underline{att}(s) \to A$ which satisfies every semantic equation, i.e. for every $\langle\vartheta, x\rangle \in \underline{att}(s)$ the equation $\underline{val}(\langle\vartheta, x\rangle) = \widehat{\underline{val}}(R_s(\langle\vartheta, x\rangle))$ holds where $\widehat{\underline{val}}$ is the unique homomorphism which extends $\underline{val}$. The tuple $(s, \underline{val})$ is called an *attributed syntax tree* of G. $\diamond$

Since we are only interested in the value of the meaning attribute at the root (i.e. the *transformational approach* [11]), we associate with an attribute grammar a string–to–value translation.

Definition 3.6 (String–to–value translation)
Let $G = (G_0, \mathcal{B}, \mathcal{A})$ be an attribute grammar with underlying context–free grammar $G_0 = (N, \Sigma, \pi, P, S)$, attribute scheme $\mathcal{B} = (\Omega, \underline{inh}, \underline{syn}, \sigma_0, R)$, and Ω–algebra $\mathcal{A} = (A, \varphi)$. The *string–to–value translation* of G is the set

$$\tau_G = \{(w, \underline{val}(\langle\sigma_0, \varepsilon\rangle)) \in L(G_0) \times A \mid (s, \underline{val}) \text{ attributed syntax tree of } w\}. \quad \diamond$$

From the definition of an attribute scheme it follows that every attribute occurrence of a syntax tree appears on the left hand side of exactly one semantic equation. However, this does not imply that every attribute value is uniquely defined, because circularities may occur.

Definition 3.7 (Attribute dependencies)
Let $G_0 = (N, \Sigma, \pi, P, S)$ be a context–free grammar, and let $\mathcal{B} = (\Omega, \underline{inh}, \underline{syn}, \sigma_0, R)$ be an attribute scheme. Let $p \in P$ be a production of G_0. For every inside attribute $\langle\vartheta, i\rangle \in \underline{in}(p)$ and every outside attribute $\langle\theta, j\rangle \in \underline{out}(p)$ which occurs in $R_p(\langle\vartheta, i\rangle)$, we say that $\langle\vartheta, i\rangle$ *depends on* $\langle\theta, j\rangle$. The *dependency set* of $\langle\vartheta, i\rangle$ is the set $D_p(\langle\vartheta, i\rangle) \subset \underline{out}(p)$ of all outside attributes which $\langle\vartheta, i\rangle$ depends upon. The *dependency graph* DG_p of p is the tuple (DV_p, DE_p) where $DV_p = \underline{att}(p)$ is the set of vertices and $DE_p = \{(\langle\theta, j\rangle, \langle\vartheta, i\rangle) \in \underline{out}(p) \times \underline{in}(p) \mid \langle\theta, j\rangle \in D_p(\langle\vartheta, i\rangle)\}$ is the set of edges. In an analogous way, we define the *dependency graph* DG_s of a syntax tree $s \in T_{G_0}$. An attribute grammar G with underlying context–free grammar G_0 and attribute scheme $\mathcal{B}$ is called *noncircular* if for every syntax tree $s \in T_{G_0}$, its dependency graph DG_s is acyclic. $\diamond$

Noncircular attribute grammars (also known as *well–defined*) admit the evaluation of every attribute occurrence in a syntax tree; the evaluation can be based on a *topological sort* of its dependency graph, thus defining every attribute value uniquely. In the case of a noncircular attribute grammar for which the underlying context–free grammar G_0 is unambiguous (e.g. an LL(1) grammar), its string–to–value translation obviously can be regarded as a mapping

$$\tau_G : L(G_0) \to A$$

with $\tau_G(w) = \underline{val}(\langle\sigma_0, \varepsilon\rangle)$ where $(s, \underline{val})$ is the unique attributed syntax tree of w.

Example 3.1 The attribute grammar which was mentioned in the introduction is based on the following context–free grammar:

$$\begin{aligned} G_0 \;=\; & (\{S, L, B\}, \{0, 1\}, \pi, \{\pi_1 : S \to L, \pi_2 : L \to B\,L, \pi_3 : L \to \varepsilon, \\ & \pi_4 : B \to 0, \pi_5 : B \to 1\}, S). \end{aligned}$$

The attribute scheme

$$\mathcal{B} = (\Omega, \underline{inh}, \underline{syn}, \sigma_0, R)$$

is given by

$$\begin{aligned}
\Omega &= \{zero^{(0)}, dec^{(1)}, inc^{(1)}, exp^{(1)}, add^{(2)}\},\\
\underline{inh}(S) &= \emptyset,\\
\underline{inh}(L) &= \underline{inh}(B) = \{p\},\\
\underline{syn}(S) &= \underline{syn}(B) = \{v\},\\
\underline{syn}(L) &= \{l, v\},\\
\sigma_0 &= v.
\end{aligned}$$

For every production $p \in P$, we represent R_p in the usual way as a set of semantic rules.

$$\begin{aligned}
R_{\pi_1} &= \{\langle p,1\rangle = dec(\langle l,1\rangle),\ \langle v,\varepsilon\rangle = \langle v,1\rangle\},\\
R_{\pi_2} &= \{\langle p,1\rangle = \langle p,\varepsilon\rangle, \qquad \langle p,2\rangle = dec(\langle p,\varepsilon\rangle),\\
&\qquad \langle l,\varepsilon\rangle = inc(\langle l,2\rangle),\ \langle v,\varepsilon\rangle = add(\langle v,1\rangle,\langle v,2\rangle)\},\\
R_{\pi_3} &= \{\langle l,\varepsilon\rangle = zero, \qquad \langle v,\varepsilon\rangle = zero\},\\
R_{\pi_4} &= \{\langle v,\varepsilon\rangle = zero\}, \qquad \text{and}\\
R_{\pi_5} &= \{\langle v,\varepsilon\rangle = exp(\langle p,\varepsilon\rangle)\}.
\end{aligned}$$

We complete the definition of the attribute grammar

$$G = (G_0, \mathcal{B}, \mathcal{A})$$

by adding the Ω–algebra $\mathcal{A} = (A, \varphi)$ which reflects the intuitive meaning of all operation symbols, based on the carrier set $A = \mathbb{Z}$ of integer numbers. For every $x, y \in \mathbb{Z}$, we let

$$\begin{aligned}
\varphi(zero)() &= 0, & \varphi(exp)(x) &= 2^x,\\
\varphi(dec)(x) &= x - 1, & \varphi(add)(x,y) &= x + y.\\
\varphi(inc)(x) &= x + 1,
\end{aligned}$$

Figure 5 illustrates an attributed syntax tree of G together with its dependency graph. Note that production $L \to \varepsilon$ applies at the rightmost occurrence of L. As one can easily see, the value of the meaning attribute v at the root corresponds to the decimal value of the binary numeral which constitutes the front of the tree. $\diamond$

Note that attribute dependencies are defined for attribute schemes without regarding concrete algebras. Thus, when defining the notion of dependency, we consider the "worst case" in the sense that we view every basic operation as being strict in each of its arguments. In fact, in the initial term algebra $\mathcal{T}_\Omega$ over the set Ω of operation symbols every basic operation is strict. Recall from the introduction that, for technical convenience, we will consider in our investigation about the combination of top–down parsing and attribute evaluation only those attribute grammars in which the semantic domain is an initial term algebra. Thus, our choice of semantic domain fits to the handling of strictness of basic operations.

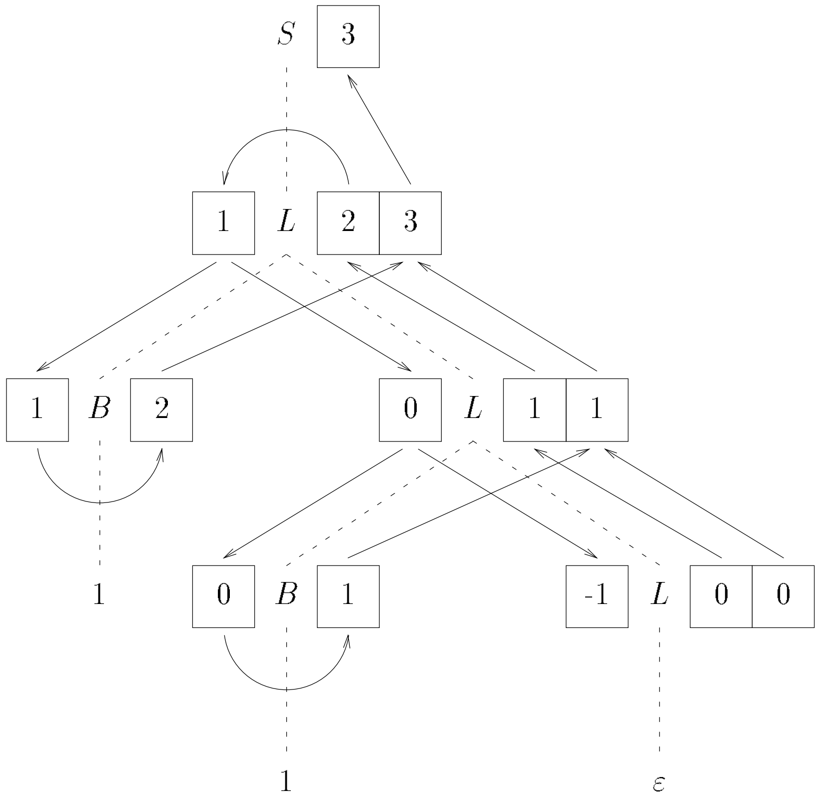


Figure 5: An attributed syntax tree and its dependency graph.

4 Attribute evaluation during depth–first left–to–right tree traversal

As discussed in the introduction, we abstract from the parsing problem in this section. We will construct an attribute evaluation algorithm $\underline{eval}$ which, for every noncircular attribute grammar $G = (G_0, \mathcal{B}, \mathcal{T}_\Omega)$ and every parse tree s, computes the value of the designated synthesized attribute at the root of s. Moreover, $\underline{eval}$ is compatible with the scanning and parsing of input strings such that it can easily be integrated into a usual top–down parsing automaton for performing both top–down parsing and attribute evaluation simultaneously (cf. Section 5).

The algorithm $\underline{eval}$ is a simple refinement of the well–known attribute evaluation algorithm $\underline{L\text{–}eval}$ which is tailor–made for L–attributed grammars (cf. e.g. P2 of [10]). $\underline{L\text{–}eval}$ consists of a recursive procedure with one parameter of type "node of syntax tree" and it performs a depth–first left–to–right traversal over the syntax tree. Figure 6 recalls the algorithm in our notational framework. Here $\widehat{\underline{val}}$ is the unique homomorphism from $\mathcal{T}_\Omega(att(s))$ to $\mathcal{T}_\Omega$ induced by $\underline{val} : att(s) \to T_\Omega$.

Let us now discuss the problems which occur when using $\underline{L\text{–}eval}$ for the evaluation of attributes of an arbitrary noncircular attribute grammar which is not L. We consider the production $p = A \to BCD$ of some context–free grammar and we assume that it applies at some node x of the syntax tree s_w of w. Let us assume that every nonterminal symbol is associated with exactly one inherited attribute ι and with exactly one synthesized attribute σ. Let us further assume that the semantic rule of $\langle\iota, 2\rangle$ induces the dependency

procedure *L–eval* $(x : node)$;
  begin
    let $p = A_0 \to w_0\, A_1\, w_1 \ldots A_n\, w_n$ be the production which applies at x;
    (* Process every successor of x *)
    for $i = 1$ **to** n **do**
      (* Evaluate inherited attributes of $x.i$ *)
      for every $\iota \in \underline{inh}(A_i)$ **do**
        $\underline{val}(\langle \iota, x.i\rangle) = \widehat{\underline{val}}(R_s(\langle \iota, x.i\rangle))$
      end;
      (* Visit ith subtree *)
      L–eval $(x.i)$;
    end;
    (* Evaluate synthesized attributes of x *)
    for every $\sigma \in \underline{syn}(A_0)$ **do**
      $\underline{val}(\langle \sigma, x\rangle) = \widehat{\underline{val}}(R_s(\langle \sigma, x\rangle))$
    end;
  end *L–eval*.

Figure 6: Attribute evaluation algorithm *L–eval*.

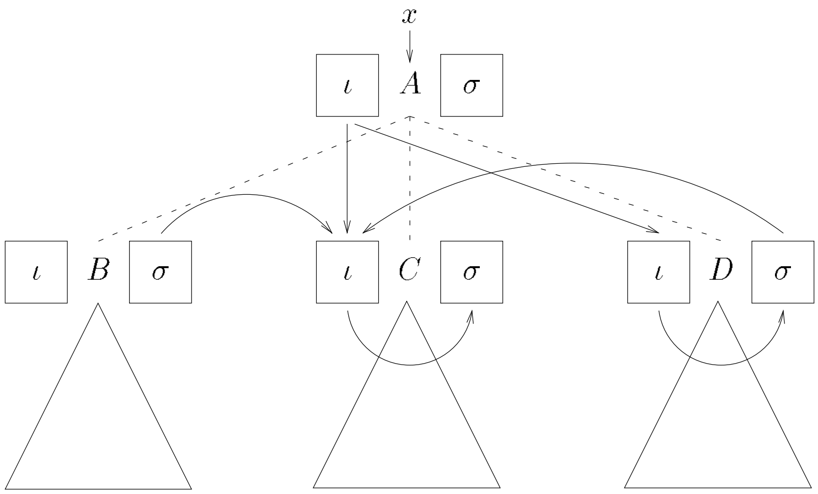


Figure 7: Attribute dependencies of $\langle \iota, 2\rangle$.

set $D_p(\langle \iota, 2\rangle) = \{\langle \iota, \varepsilon\rangle, \langle \sigma, 1\rangle, \langle \sigma, 3\rangle\}$ as it is illustrated in Figure 7.

Before the first visit of *L–eval* to node x, the value $t_{\langle \iota, x\rangle}$ of $\langle \iota, x\rangle$ is already computed. Then *L–eval* computes the inherited attribute of the first son $x.1$ of x, i.e. the attribute occurrence $\langle \iota, x.1\rangle$, and recursively calls itself to node $x.1$. If *L–eval* returns to x, then the value $t_{\langle \sigma, x.1\rangle}$ of $\langle \sigma, x.1\rangle$ is known, and *L–eval* tries to compute the value of the inherited attribute at $x.2$, i.e. the attribute occurrence $\langle \iota, x.2\rangle$. However, this is not possible, because this value depends on $\langle \sigma, x.3\rangle$ which is not yet known. Then we say that $\langle \iota, 2\rangle$ is an open reference of $\langle \sigma, 3\rangle$ in p, and we say that 2 is an open reference index of $\langle \sigma, 3\rangle$.

Definition 4.1 (Open reference, open reference index)
Let $G = (G_0, \mathcal{B}, \mathcal{T}_\Omega)$ be a noncircular attribute grammar with underlying context–free grammar $G_0 = (N, \Sigma, \pi, P, S)$, and attribute scheme $\mathcal{B} = (\Omega, \underline{inh}, \underline{syn}, \sigma_0, R)$. Let $p \in P$ be a production, and let $\langle \sigma, i\rangle \in \underline{out}(p)$ be a synthesized outside attribute occurrence of p. The set of *open references* of $\langle \sigma, i\rangle$ in p is defined by

$$O_p(\langle \sigma, i\rangle) = \{\langle \iota, j\rangle \in \underline{in}(p) \mid \langle \sigma, i\rangle \in D_p(\langle \iota, j\rangle), 1 \le j \le i\}.$$

The set of *open reference indices* of $\langle \sigma, i\rangle$ in p is defined by

$$I_p(\langle \sigma, i\rangle) = \{j \mid \langle \sigma, i\rangle \in D_p(\langle \iota, j\rangle) \text{ for some } \langle \iota, j\rangle \in \underline{in}(p) \text{ with } 1 \le j \le i\}. \quad \diamond$$

Clearly, for every production p, open references and open reference indices can be detected by means of a simple static analysis. In our example, we have to deal with the following problem: which value should be associated with $\langle \iota, x.2\rangle$? Assume that the semantic rule of $\langle \iota, 2\rangle$ is

$$\langle \iota, 2\rangle = f(\langle \iota, \varepsilon\rangle, g(\langle \sigma, 1\rangle, \langle \sigma, 3\rangle), \langle \sigma, 3\rangle)$$

where f and g are operation symbols with rank 3 and 2, respectively. Our algorithm $\underline{eval}$ implements the following solution: It associates with $\langle \iota, x.2\rangle$ the term

$$t_{\langle \iota, x.2\rangle} = f(t_{\langle \iota, x\rangle}, g(t_{\langle \sigma, x.1\rangle}, \langle \sigma, x.3\rangle), \langle \sigma, x.3\rangle).$$

That is, the final value of $\langle \iota, x.2\rangle$ is only approximated, because there are still occurrences of $\langle \sigma, x.3\rangle$ in the term; we also call $t_{\langle \iota, x.2\rangle}$ a schematic approximation. Then, $\underline{eval}$ is called recursively to node $x.2$ and after returning to x it has computed the value $t_{\langle \sigma, x.2\rangle}$ of the attribute occurrence $\langle \sigma, x.2\rangle$. Let us assume that $\langle \sigma, x.2\rangle$ depends on $\langle \iota, x.2\rangle$, then $t_{\langle \sigma, x.2\rangle}$ contains the tree $t_{\langle \iota, x.2\rangle}$ as a subtree, and hence it contains $\langle \sigma, x.3\rangle$ as a leaf. Now assume that there is a semantic rule

$$\langle \iota, 3\rangle = h(\langle \iota, \varepsilon\rangle)$$

in G where h is a unary operation symbol. Then, in the usual way, $\underline{eval}$ can associate the value $h(t_{\langle \iota, x\rangle})$ with $\langle \iota, x.3\rangle$. Eventually, $\underline{eval}$ is called recursively to node $x.3$ and, after returning to x, the value $t_{\langle \sigma, x.3\rangle}$ of $\langle \sigma, x.3\rangle$ is known.

At this point, the algorithm can further approximate synthesized attributes at open reference indices of $\langle \sigma, 3\rangle$. In particular, the value $t_{\langle \sigma, x.2\rangle}$ can be further approximated by replacing occurrence of $\langle \sigma, x.3\rangle$ by $t_{\langle \sigma, x.3\rangle}$. Finally, $\underline{eval}$ can compute the value of the synthesized attribute occurrence at x and finish the recursive call to x.

Note that the computation of the ground term for the designated synthesized attribute σ_0 at the root of the syntax tree may involve several approximation steps. This phenomenon is illustrated in Figure 8 which shows a syntax tree s and its dependency graph of some attribute grammar which is not specified here. We note however that, since the attribute grammars which we consider are assumed to be noncircular, every open reference which is involved in the computation of σ_0 at the root of s will be resolved if $\underline{eval}$ returns to the root of s.

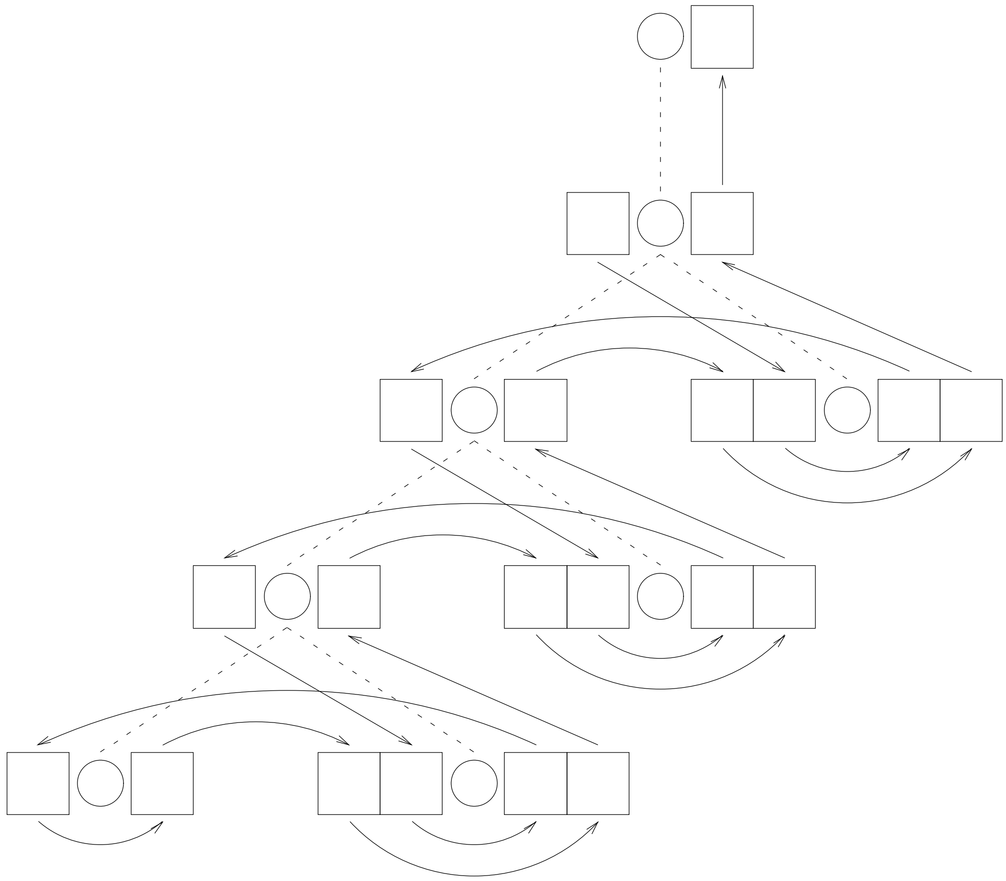


Figure 8: Repeated approximation during depth–first left–to–right tree traversal.

Obviously, our algorithm $\underline{eval}$ assigns to every attribute occurrence either a ground term, i.e., an element in T_Ω, or a schematic approximation, i.e., an element in $T_\Omega(att(s)) \setminus T_\Omega$; the latter terms are also called *functional terms* because they induce derived operations (cf. Section 3). Hence, $\underline{eval}$ deals with mappings of type $att(s) \to T_\Omega(att(s))$ which we call *intermediate attributions*, and it begins with the *start attribution* $\underline{val}_\emptyset$ which is the intermediate attribution with $\underline{val}_\emptyset(\langle\vartheta, s\rangle) = \langle\vartheta, s\rangle$ for every $\langle\vartheta, s\rangle \in att(s)$, i.e., $\underline{val}_\emptyset$ is the identity. Intuitively, $\underline{val}_\emptyset$ comprises the worst possible approximation of every attribute occurrence. Then, during the traversal through s, $\underline{eval}$ computes new intermediate attributions and, for every attribute occurrence $\langle\vartheta, x\rangle \in att(s)$, we can distinguish three cases:

- $\underline{val}(\langle\vartheta, x\rangle) = \langle\vartheta, x\rangle$: $\underline{eval}$ did not start yet to compute some value for $\langle\vartheta, x\rangle$. Note that the equation really characterizes this situation, because we are dealing with noncircular attribute grammars.

- $\underline{val}(\langle\vartheta, x\rangle) \in T_\Omega(\underline{att}(s)) \setminus (T_\Omega \cup \{\langle\vartheta, x\rangle\})$, i.e., $\underline{val}(\langle\vartheta, x\rangle)$ is a functional term which is not equal to $\langle\vartheta, x\rangle$.

- $\underline{val}(\langle\vartheta, x\rangle) \in T_\Omega$, i.e., $\underline{val}(\langle\vartheta, x\rangle)$ is a ground term. This represents the final value in the semantic domain.

procedure $\underline{eval}\,(x : node)$;

  begin
    let $p = A_0 \rightarrow w_0\, A_1\, w_1 \ldots A_n\, w_n$ be the production which applies at x;

    (* Process every successor of x *)
    for $i = 1$ **to** n **do**
      (* Evaluate inherited attributes of $x.i$ *)
      for every $\iota \in \underline{inh}(A_i)$ **do**
        $\underline{val}(\langle\iota, x.i\rangle) = \widehat{\underline{val}}(R_s(\langle\iota, x.i\rangle))$
      end;
      (* Visit ith subtree *)
      $\underline{eval}\,(x.i)$;
      (* Refine approximations at open reference indices *)
      for every $\sigma \in \underline{syn}(A_i)$ **do**
        for every $j \in I_p(\langle\sigma, i\rangle)$ **do**
          for every $\vartheta \in \underline{syn}(A_j)$ **do**
            $\underline{val}(\langle\vartheta, x.j\rangle) = \widehat{\underline{val}}(\underline{val}(\langle\vartheta, x.j\rangle))$
          end;
        end;
      end;
    end;

    (* Evaluate synthesized attributes of x *)
    for every $\sigma \in \underline{syn}(A_0)$ **do**
      $\underline{val}(\langle\sigma, x\rangle) = \widehat{\underline{val}}(R_s(\langle\sigma, x\rangle))$
    end;

  end $\underline{eval}$.

Figure 9: Attribute evaluation algorithm $\underline{eval}$.

Eventually, when returning to the root of s, an intermediate attribution $\underline{val}_{fin}$ is computed such that $\underline{val}_{fin}(\langle\vartheta, x\rangle)$ is a ground term for every $\langle\vartheta, x\rangle \in att(s)$.

The algorithm $\underline{eval}$ is obtained from $\underline{L\text{–}eval}$ by inserting behind the recursive call to the ith son a program piece which refines the approximation of synthesized attributes at open reference indices of the ith son. Figure 9 shows the algorithm $\underline{eval}$.

5 The parsing–evaluating automaton

In Section 2 we have recalled the concepts of context–free grammar, pushdown automaton, and top–down parsing automaton. The latter automaton model serves for the deterministic parsing of LL(1) grammars.

In Section 3 we have attached an attribute scheme and an algebra to context–free grammars thereby defining attribute grammars. Here we extend in a similar way the pushdown automaton, and thus, in particular, the top–down parsing automaton, in order to deal with attribute evaluation. The addition to the top–down parsing automaton yields the desired automaton, called the attributed top–down parsing automaton, which deterministically performs top–down parsing and full attribute evaluation for every noncircular attribute grammar with underlying LL(1) grammar and with an initial term algebra as semantic domain. It implements the attribute evaluation algorithm $\underline{eval}$ as described in Section 4 without storing the syntax tree explicitly.

First of all, however, we have to consider the problem of how to represent attribute values. The basic idea is to represent (ground or functional) terms by appropriate graphs which contain special nodes supporting the management of open references.

5.1 Representation of terms by graphs

Definition 5.1 (Representing graph)
Let $\Omega = \bigcup_{n\in\mathbb{N}} \Omega^{(n)}$ be a set of operation symbols. An Ω*–graph* is a labeled, ordered, directed, and acyclic graph

$$g = (V, \lambda, \underline{succ})$$

where V is the finite set of nodes (vertices) which is a subset of a countably infinite universe U of nodes, $\lambda : V \to \Omega$ denotes the labeling function, and $\underline{succ} : V \to V^*$ is the successor function such that $\lambda(x) \in \Omega^{(n)}$ iff $|\underline{succ}(x)| = n$. If $\underline{succ}(x) = x_1 \dots x_n$, then we abbreviate $\underline{succ}_i(x) = x_i$ for $i \in \{1, \dots, n\}$. The set of all Ω–graphs is denoted by DAG_Ω. An Ω–graph $g \in DAG_\Omega$ is said to *represent* the Ω–term $t = f(t_1, ..., t_n) \in T_\Omega$ ($n \in \mathbb{N}$) if there is an f–labeled node x of g with no predecessor (called a *root*) such that for every $i \in \{1, \dots, n\}$, the subgraph with root $\underline{succ}_i(x)$ (denoted by $g[\underline{succ}_i(x)]$) represents t_i. $\diamond$

So far, it is clear how to represent a ground term which has been computed as the final value of an attribute. But how do we represent functional terms by graphs? We recall the situation where such terms can occur: During the visit of a node x of the syntax tree with successors $x.1, \dots, x.n$, the evaluator may encounter open references when computing the value of any inherited attribute occurrence $\langle\iota, x.i\rangle \in \underline{att}(s)$ where $i \in \{1, \dots, n\}$.

This representation problem is solved as follows. Let p be the production which applies at x, and let $t = R_p(\langle\iota, i\rangle) \in T_\Omega(\underline{out}(p))$ be the right hand side of the semantic rule for $\langle\iota, x.i\rangle$. From the argument list $\underline{arg}(t)$ of t, we can extract all attribute occurrences for which $\langle\iota, x.i\rangle$ is an open reference, by applying a filter operation: For every set X and predicate $b : X \to \{true, false\}$, the mapping $\underline{filter}_b : X^* \to X^*$ is given inductively by:

$$\underline{filter}_b(w) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x\ \underline{filter}_b(w') & \text{if } w = x\ w', b(x) = true \\ \underline{filter}_b(w') & \text{if } w = x\ w', b(x) = false \end{cases}$$

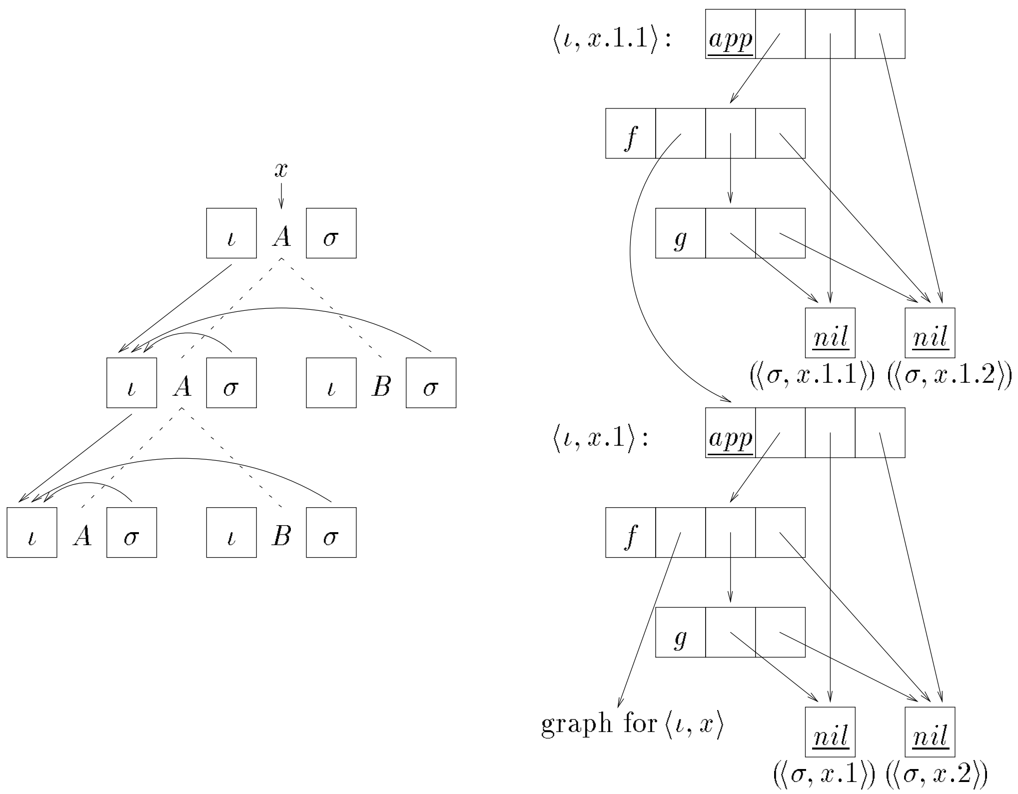


Figure 10: Representation of open references.

Then we define $\underline{arg}_i(t) = \underline{filter}_{b_i}(\underline{arg}(t))$ where $b_i(\langle\vartheta, j\rangle) = true$ for $\langle\vartheta, j\rangle \in \underline{out}(p)$ iff $j \geq i$. Note that $\underline{arg}_i(t)$ contains an attribute occurrence at most once. If $|\underline{arg}_i(t)| = k$, then $\langle\iota, i\rangle$ has to be associated with a function depending on all k attribute occurrences in this list. For this function, a graph is constructed in which every node representing an argument $\langle\sigma, j\rangle$ is labeled by a designated constant operation symbol $\underline{nil}$. Later, this $\underline{nil}$ will be replaced by a unary symbol $\underline{ref}$, and eventually, the successor position of $\underline{ref}$ represents the value of this synthesized attribute occurrence. Moreover, there is an additional root node which is labeled by an operation symbol $\underline{app}$ of arity $k + 1$. Its first successor is the root of the graph representation of t and the other successors are the representations of the elements in $\underline{arg}_i(t)$.

Example 5.1 Let $A \to A\,B$ be a production of an attribute grammar. With every nonterminal symbol there is associated one inherited attribute ι and one synthesized attribute σ. Let the semantic rule for $\langle\iota, 1\rangle$ be specified as follows:

$$\langle\iota, 1\rangle = f(\langle\iota, \varepsilon\rangle, g(\langle\sigma, 1\rangle, \langle\sigma, 2\rangle), \langle\sigma, 2\rangle).$$

Then, $\underline{arg}_1(R_{A\to A\,B}(\langle\iota, 1\rangle)) = \langle\sigma, 1\rangle\langle\sigma, 2\rangle$. Figure 10 depicts the attribute dependency graph of a syntax tree with root node x where $A \to A\,B$ applies twice and the graph which represents $\langle\iota, x.1.1\rangle$ and $\langle\iota, x.1\rangle$. $\diamond$

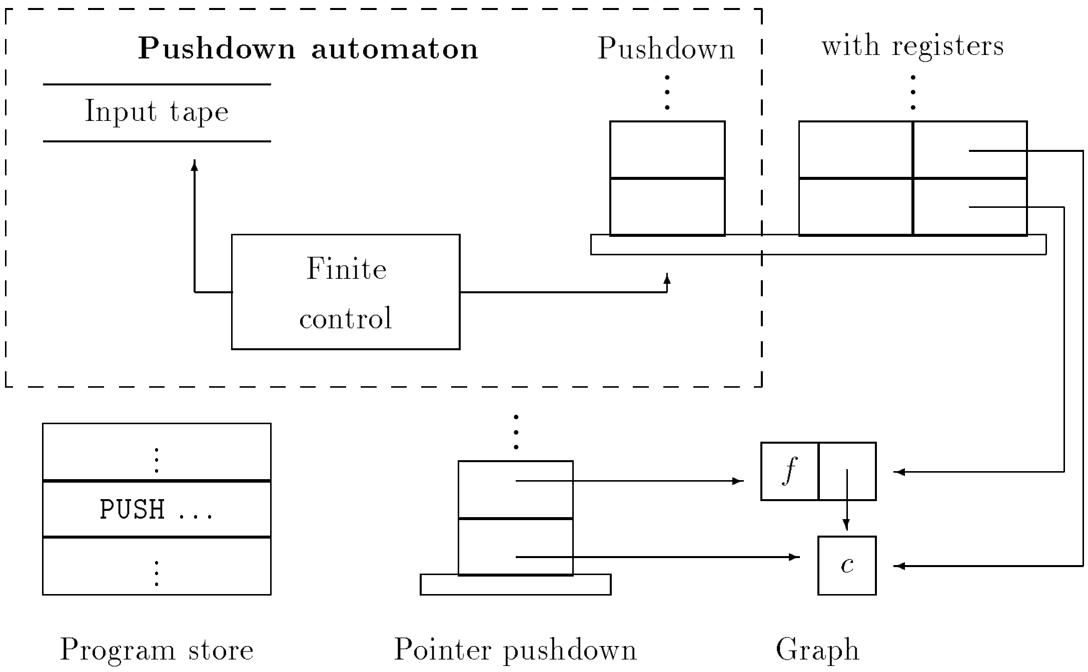


Figure 11: Organization of attributed pushdown automata.

5.2 Attributed pushdown automata

In Section 2 we have introduced the general concept of pushdown automaton, and we have refined this concept in order to handle top–down parsing of context–free grammars; this led to the notion of top–down parsing automata. In Section 3, context–free grammars have been extended to attribute grammars. In the same way we are now going to extend the pushdown automata by giving them the capability of computing attribute values and representing them by graphs. The resulting attributed pushdown automata will be used as attributed top–down parsing automata to compute the string–to–value translation for any noncircular attribute grammar with underlying LL(1) grammar.

A pushdown automaton consists of the following components:

- An input tape,
- a pushdown and
- a finite control.

This concept is extended to attributed pushdown automaton

- by adding a graph for the representation of attribute values,
- by associating with every pushdown entry a set of registers each of which contains a pointer to a subgraph of the graph,
- by adding a program store containing instructions which manipulate the graph as well as the registers of the pushdown, and
- by adding a pointer pushdown which is used for intermediate computations.

Figure 11 summarizes the structure of attributed pushdown automata.

Definition 5.2 (Attributed pushdown automaton)
An *attributed pushdown automaton*

$$\mathcal{A} = (\mathcal{A}_0, \Omega, \underline{act}, REG, \rho_0)$$

consists of a pushdown automaton $\mathcal{A}_0 = (Q, \Sigma, \Gamma, \delta, q^0, \gamma_0, F)$, a set of operation symbols Ω disjoint from $\{\underline{nil}^{(0)}, \underline{ref}^{(1)}\}$ and $\{\underline{app}^{(n)} \mid n \geq 2\}$, an *assignment of programs* $\underline{act} : \Gamma^2 \to PGM$, a finite set REG of *register names*, and an *output register* $\rho_0 \in REG$. The set PGM of *programs* is given by $PGM = (CMD\,\{;\})^*$ where the instruction set CMD is decomposed into the set of *register instructions*

$$\begin{aligned} CMD_{Reg} \;=\; & \{\mathtt{COPY}(i) \mid i \in \{1,2\}\} \cup \{\mathtt{PUSH}(\rho, i) \mid \rho \in REG, i \in \{1,2,3\}\} \cup \\ & \{\mathtt{TOP}(\rho) \mid \rho \in REG\}, \end{aligned}$$

and the set of *graph instructions*

$$\begin{aligned} CMD_{Graph} \;=\; & \{\mathtt{JOIN}(k) \mid k \in \mathbb{N}\} \cup \{\mathtt{MKAPP}(n) \mid n \in \mathbb{N}\} \cup \{\mathtt{MKNIL}\} \cup \\ & \{\mathtt{MKNODE}(f) \mid f \in \Omega\} \cup \{\mathtt{MKREF}(k) \mid k \in \mathbb{N}\} \cup \\ & \{\mathtt{SUCC}(k) \mid k \in \mathbb{N}\} \cup \{\mathtt{TOPCON}(n) \mid n \in \mathbb{N}\}. \end{aligned}$$

An attributed pushdown automaton is called *deterministic* if its underlying pushdown automaton is deterministic. The set of *pointer pushdowns* is the set

$$PPD = U^*,$$

and the set of *register assignments* of $\mathcal{A}$ is the set

$$ASS_{\mathcal{A}} = \{\underline{ass} \mid \underline{ass} : REG \longrightarrow U\}.$$

Recall that U is the universe of graph nodes. The set of *instantaneous descriptions* of $\mathcal{A}$ is the cartesian product

$$ID_{\mathcal{A}} = Q \times \Sigma^* \times (\Gamma \times ASS_{\mathcal{A}})^* \times DAG_{\Omega'}$$

where Ω' is the extension of Ω defined as follows:

$$\Omega^{(n)'} = \begin{cases} \Omega^{(0)} \cup \{\underline{nil}^{(0)}\} & \text{if } n = 0 \\ \Omega^{(1)} \cup \{\underline{ref}^{(1)}\} & \text{if } n = 1 \\ \Omega^{(n)} \cup \{\underline{app}^{(n)}\} & \text{if } n \geq 2. \end{cases}$$

The *transition relation* of $\mathcal{A}$

$$\vdash_{\mathcal{A}} \subset ID_{\mathcal{A}} \times ID_{\mathcal{A}}$$

is defined as follows: If $(q', op, \gamma') \in \delta(q, x, \gamma_1\gamma_2)$ with $q, q' \in Q$, $x \in \Sigma \cup \{\varepsilon\}$, $\gamma_1, \gamma_2, \gamma' \in \Gamma$, $op \in \{\underline{mod}, \underline{pop}, \underline{push}\}$, then for every $w \in \Sigma^*$, $\underline{ass}_1, \underline{ass}_2 \in ASS_{\mathcal{A}}$, $\xi \in (\Gamma \times ASS_{\mathcal{A}})^*$, and $g \in DAG_{\Omega'}$

$$(q, x\,w, (\gamma_1, \underline{ass}_1)(\gamma_2, \underline{ass}_2)\xi, g) \vdash_{\mathcal{A}} (q', w, \xi', g')$$

where

$$\xi' = \begin{cases} (\gamma', \underline{ass}')(\gamma_2, \underline{ass}_2)\xi & \text{if } op = \underline{mod} \\ (\gamma', \underline{ass}')\xi & \text{if } op = \underline{pop} \\ (\gamma', \underline{ass}')(\gamma_1, \underline{ass}_1)(\gamma_2, \underline{ass}_2)\xi & \text{if } op = \underline{push} \end{cases}$$

and

$$(\underline{ass}', g') = \mathcal{P}_{\mathcal{A}}[\![\underline{act}(\gamma_1\gamma_2)]\!](\underline{ass}_1, \underline{ass}_2, g). \quad \diamond$$

As one can see, transitions of an attributed pushdown automaton are performed in dependency of the present state, the current input symbol and the upper two pushdown entries. The transition function of the underlying pushdown automaton determines the next state as well as the kind of pushdown modification. Furthermore, the program selected by the upper pushdown entries computes the register assignment of the new top of pushdown, basing on the graph. For storing graph pointers, it makes use of the pointer pushdown which is empty at the beginning of program execution.

Next we will define in a bottom–up fashion the semantics of attributed pushdown automata ending up with the definition of the translation computed by an attributed pushdown automaton. We start with the semantics of register instructions and graph instructions and continue with the semantics of programs which was used in the definition of the transition relation.

Definition 5.3 (Instruction semantics)
Let $\mathcal{A} = (\mathcal{A}_0, \Omega, \underline{act}, REG, \rho_0)$ be an attributed pushdown automaton. The *instruction semantics* of $\mathcal{A}$ is the partial function

$$\mathcal{C}_\mathcal{A} : CMD \dashrightarrow (ASS^3_\mathcal{A} \times DAG_{\Omega'} \times PPD \dashrightarrow ASS^3_\mathcal{A} \times DAG_{\Omega'} \times PPD)$$

which, for every $\underline{ass}_1, \underline{ass}_2, \underline{ass}_3 \in ASS_\mathcal{A}$, $g = (V, \lambda, \underline{succ}) \in DAG_{\Omega'}$, $x, y, y_1, \ldots, y_n \in V$, $x' \in U \setminus V$, $\xi \in PPD$, $i \in \{1, 2, 3\}$, $\rho \in REG$, $f \in \Omega$, and $k, n \in \mathbb{N}$, is defined as follows:

$$\begin{aligned}
&\mathcal{C}_\mathcal{A}[\![\texttt{COPY}(i)]\!](\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g, \xi) = (\underline{ass}_1, \underline{ass}_2, \underline{ass}_i, g, \xi),\\
&\mathcal{C}_\mathcal{A}[\![\texttt{JOIN}(k)]\!](\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g, x\ y\ \xi) = (\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g', y\ \xi),\\
&\qquad \text{where } g' = (V, \lambda, \underline{succ}') \text{ with } \underline{succ}' = \underline{succ}[x/y_1 \ldots y_{k-1}\ y\ y_{k+1} \ldots y_n]\\
&\qquad \text{where } \underline{succ}(x) = y_1 \ldots y_n,\\
&\mathcal{C}_\mathcal{A}[\![\texttt{MKAPP}(n)]\!](\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g, \xi) = (\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g', x'\ \xi),\\
&\qquad \text{where } g' = (V', \lambda', \underline{succ}) \text{ with } V' = V \cup \{x'\}, \text{ and } \lambda' = \lambda[x'/\underline{app}^{(n)}],\\
&\mathcal{C}_\mathcal{A}[\![\texttt{MKNIL}]\!](\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g, \xi) = (\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g', x'\ \xi),\\
&\qquad \text{where } g' = (V', \lambda', \underline{succ}) \text{ with } V' = V \cup \{x'\}, \text{ and } \lambda' = \lambda[x'/\underline{nil}^{(0)}],\\
&\mathcal{C}_\mathcal{A}[\![\texttt{MKNODE}(f)]\!](\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g, \xi) = (\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g', x'\ \xi),\\
&\qquad \text{where } g' = (V', \lambda', \underline{succ}) \text{ with } V' = V \cup \{x'\}, \text{ and } \lambda' = \lambda[x'/f],\\
&\mathcal{C}_\mathcal{A}[\![\texttt{MKREF}(k)]\!](\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g, x\ y\ \xi) = (\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g', \xi),\\
&\qquad \text{where } g' = (V, \lambda', \underline{succ}') \text{ with } \lambda' = \lambda[\underline{succ}_k(x)/\underline{ref}],\\
&\qquad \text{and } \underline{succ}' = \underline{succ}[\underline{succ}_k(x)/y],\\
&\mathcal{C}_\mathcal{A}[\![\texttt{PUSH}(\rho, i)]\!](\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g, \xi) = (\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g, \underline{ass}_i(\rho)\ \xi),\\
&\mathcal{C}_\mathcal{A}[\![\texttt{SUCC}(k)]\!](\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g, x\ \xi) = (\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g, \underline{succ}_k(x)\ \xi),\\
&\mathcal{C}_\mathcal{A}[\![\texttt{TOP}(\rho)]\!](\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g, x\ \xi) = (\underline{ass}_1, \underline{ass}_2, \underline{ass}_3[\rho/x], g, \xi), \text{ and}\\
&\mathcal{C}_\mathcal{A}[\![\texttt{TOPCON}(n)]\!](\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g, x\ y_n \ldots y_1\ \xi) = (\underline{ass}_1, \underline{ass}_2, \underline{ass}_3, g', y\ \xi)\\
&\qquad \text{where } g' = (V, \lambda, \underline{succ}') \text{ with } \underline{succ}' = \underline{succ}[x/y_1 \ldots y_n]. \quad \diamond
\end{aligned}$$

Definition 5.4 (Program semantics)
Let $\mathcal{A} = (\mathcal{A}_0, \Omega, \underline{act}, REG, \rho_0)$ be an attributed pushdown automaton. The *program*

semantics of $\mathcal{A}$ is the partial function

$$\mathcal{P}_{\mathcal{A}} : PGM \dashrightarrow (ASS^2_{\mathcal{A}} \times DAG_{\Omega'} \dashrightarrow ASS_{\mathcal{A}} \times DAG_{\Omega'})$$

which is given, for every program $pgm \in PGM$, by

$$\mathcal{P}_{\mathcal{A}}[\![pgm]\!] = \underline{output}_{\mathcal{A}} \circ \mathcal{I}_{\mathcal{A}}[\![pgm]\!] \circ \underline{input}_{\mathcal{A}}$$

where the *input mapping*

$$\underline{input}_{\mathcal{A}} : ASS^2_{\mathcal{A}} \times DAG_{\Omega'} \to ASS^3_{\mathcal{A}} \times DAG_{\Omega'} \times PPD$$

is defined by

$$\underline{input}_{\mathcal{A}}(\underline{ass}_1, \underline{ass}_2, g) = (\underline{ass}_1, \underline{ass}_2, \underline{ass}_\emptyset, g, \varepsilon)$$

for every $\underline{ass}_1, \underline{ass}_2 \in ASS_{\mathcal{A}}$, and $g \in DAG_{\Omega}$ where $\underline{ass}_\emptyset(\rho)$ is undefined for every $\rho \in REG$. The *iteration semantics* of a program is the partial function

$$\mathcal{I}_{\mathcal{A}} : PGM \dashrightarrow (ASS^3_{\mathcal{A}} \times DAG_{\Omega'} \times PPD \dashrightarrow ASS^3_{\mathcal{A}} \times DAG_{\Omega'} \times PPD)$$

which is defined by

$$\begin{aligned} \mathcal{I}_{\mathcal{A}}[\![\varepsilon]\!](\underline{ass}_1, \underline{ass}_2, \underline{ass}, g, \xi) &= (\underline{ass}_1, \underline{ass}_2, \underline{ass}, g, \xi) \qquad \text{and} \\ \mathcal{I}_{\mathcal{A}}[\![C; pgm]\!](\underline{ass}_1, \underline{ass}_2, \underline{ass}, g, \xi) &= \mathcal{I}_{\mathcal{A}}[\![pgm]\!](\mathcal{A}_M[\![C]\!](\underline{ass}_1, \underline{ass}_2, \underline{ass}, g, \xi)). \end{aligned}$$

for every $\underline{ass}_1, \underline{ass}_2, \underline{ass} \in ASS_{\mathcal{A}}$, $g \in DAG_{\Omega'}$, $\xi \in PPD$, $C \in CMD$, and $pgm \in PGM$. Furthermore, let the *output mapping*

$$\underline{output}_{\mathcal{A}} : ASS^3_{\mathcal{A}} \times DAG_{\Omega'} \times PPD \to ASS_{\mathcal{A}} \times DAG_{\Omega'}$$

be given by

$$\underline{output}_{\mathcal{A}}(\underline{ass}_1, \underline{ass}_2, \underline{ass}, g, \xi) = (\underline{ass}, g)$$

for every $\underline{ass}_1, \underline{ass}_2, \underline{ass} \in ASS_{\mathcal{A}}$, $g \in DAG_{\Omega'}$, and $\xi \in PPD$. $\Diamond$

Definition 5.5 (Translation of an attributed pushdown automaton)
Let $\mathcal{A} = (\mathcal{A}_0, \Omega, \underline{act}, REG, \rho_0)$ be an attributed pushdown automaton with underlying pushdown automaton $\mathcal{A}_0 = (Q, \Sigma, \Gamma, \delta, q^0, \gamma_0, F)$. The *translation computed by* $\mathcal{A}$ is defined by

$$\begin{aligned} \tau_{\mathcal{A}} \;=\; & \{(w, g[\underline{ass}(\rho_0)]) \in \Sigma^* \times DAG_{\Omega'} \mid \text{ there are } q^f \in F \text{ and } \gamma \in \Gamma \text{ such that} \\ & (q^0, w, (\gamma_0, \underline{ass}_\emptyset)(\gamma_0, \underline{ass}_\emptyset), g_\emptyset) \vdash_{\mathcal{A}} (q^f, \varepsilon, (\gamma, \underline{ass}), g)\} \end{aligned}$$

where $g_\emptyset$ denotes the empty Ω'–graph. $\Diamond$

If $\mathcal{A}$ is a deterministic attributed pushdown automaton, its translation may be regarded as a partial function:

$$\tau_{\mathcal{A}} : \Sigma^* \dashrightarrow DAG_{\Omega'}.$$

Note that the pointer pushdown only appears as an intermediate storage; it does not occur in the instantaneous descriptions.

```
(* Resolve open references *)
for every σ ∈ syn(A_i) do
   for every ⟨ι, j⟩ ∈ O_p(⟨σ, i⟩) do
      val(⟨ι, x.j⟩) = \widehat{val}(val(⟨ι, x.j⟩))
   end;
end;
```

Figure 12: Modification of $\underline{eval}$.

5.3 Attributed top–down parsing automata

For every given noncircular attribute grammar G with underlying LL(1) grammar G_0, we now want to construct a deterministic attributed pushdown automaton which parses an input string $w\$$ where $w \in L(G_0)$ and simultaneously evaluates the meaning attribute at the root of the corresponding syntax tree according to the algorithm $\underline{eval}$ of Section 4. This automaton will be called an attributed top–down parsing automaton and will be denoted by $ATDA(G)$. As we have seen in Section 2, the LL(1) property of G_0 guarantees the determinism of $ATDA(G)$. We will slightly deviate from the algorithm $\underline{eval}$ as shown in Figure 9 in the sense that we do not refine approximations of synthesized attributes at open reference indices. Rather we resolve open references, i.e., we recompute schematic approximations of inherited attributes at open reference indices. The automaton is constructed in such a way that the pointer to the appropriate application node is available in this situation. Trying to follow the algorithm of Figure 6 would result in the repeated insertion of pieces of trees between the application node and its first son. By resolving open references, the pieces of trees which emerge during the tree traversal, can easily be built on top of the application node. Thus, the attributed top–down parsing automaton implements the algorithm $\underline{eval}$ in which the **for** statement below the label (* Refine approximations at open reference indices *) is replaced by the program piece which is shown in Figure 12.

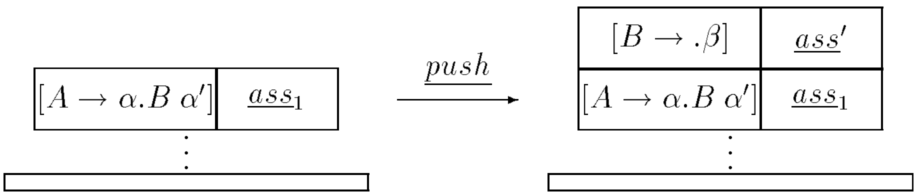
To define $ATDA(G) = (\mathcal{A}_0, \Omega, \underline{act}, REG, \rho_0)$, we have to construct its components in dependency of the given attribute grammar G.

- The parsing part has already been investigated: As the underlying pushdown automaton $\mathcal{A}_0$, we choose the top–down parsing automaton $TDA(G_0)$.
- The set Ω of operation symbols which label the graph's nodes, is exactly the set of operation symbols used in the specification of the semantic rules.
- Since the registers receive pointers to attribute values, the set REG of register names equals the set of all attribute occurrences of the productions.
- In particular, we identify the output register ρ_0 with the meaning attribute occurrence $\langle\sigma_0, \varepsilon\rangle$ of the start symbol.
- Next we have to construct an assignment $\underline{act}$ of programs, i.e., depending on the upper two LR(0) items on the pushdown, we have to specify the program which is to be executed. We distinguish between the same cases as in the construction of the top–down parsing automaton (cf. Section 2):

(i) Initiation of look–ahead:
According to the definition of attribute grammars, the start symbol has no inherited attributes. For this reason, no action is necessary; that is, $\underline{act}([\rightarrow .S][\rightarrow .S]) = \varepsilon$.

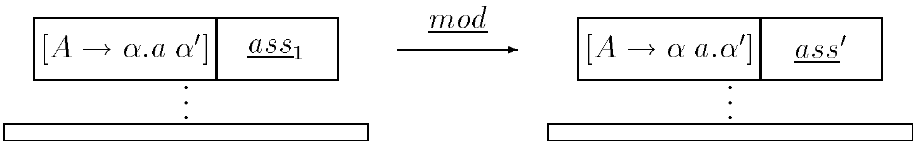
(ii) Expansion of start productions:
In analogy with case (i).

(iii) Expansion of non–start productions:



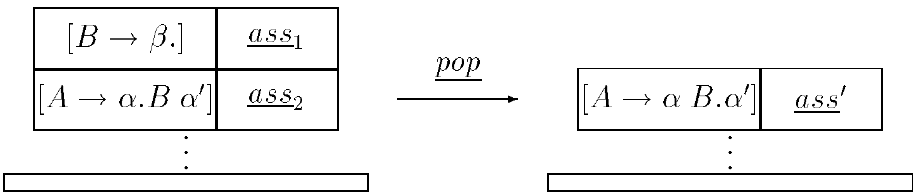
Assume that $i = |\underline{filter}_{\in N}(\alpha)| + 1$, then, in order to evaluate every inherited attribute $\iota \in \underline{inh}(B)$, we compute $\langle \iota, i\rangle$ using the semantic rule $\langle \iota, i\rangle = R_{A\rightarrow\alpha\, B\, \alpha'}(\langle \iota, i\rangle)$. The resulting ground or functional term is represented as a subgraph of the graph, and a pointer to its root is stored in the new register assignment $\underline{ass}'$ as $\underline{ass}'(\langle \iota, \varepsilon\rangle)$. Open references are handled by creation of an $\underline{app}$ node as described in Section 5.1.

(iv) Terminal symbol match:



Because terminal symbols do not have attributes, we only have to take over the register assignment, i.e., $\underline{ass}' = \underline{ass}_1$.

(v) Reduction of non–start productions:



Computing the new register assignment $\underline{ass}'$ involves four steps:

(1) Every attribute value of production $A \rightarrow \alpha B \alpha'$ which has been computed already and which has been stored in $\underline{ass}_2$, is copied into $\underline{ass}'$.

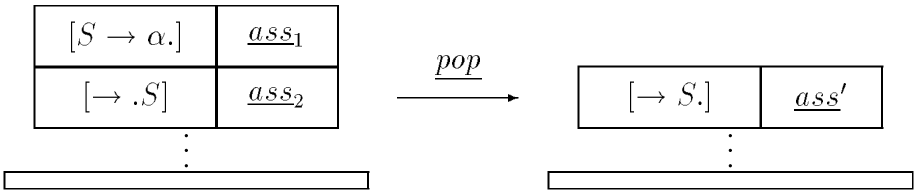
(2) Since the inherited attribute values of B stored in $\underline{ass}_1$ still may exhibit open references, they are also transferrred into $\underline{ass}'$.

(3) Every synthesized attribute $\sigma \in \underline{syn}(B)$ is evaluated using the semantic rule $\langle \sigma, \varepsilon\rangle = R_{B\rightarrow\beta}(\langle \sigma, \varepsilon\rangle)$. At this point, open references can not occur.

Afterwards, the attribute value is stored as $\underline{ass}'(\langle\sigma, i\rangle)$ where we again assume that $i = |\underline{filter}_{\in N}(\alpha)| + 1$.

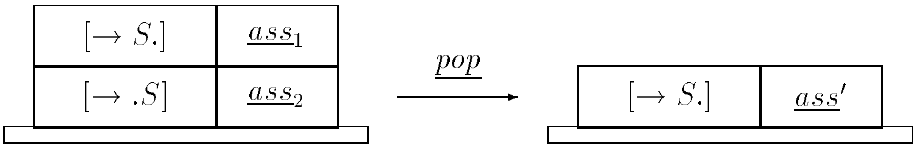
(4) The value of every synthesized attribute $\sigma \in \underline{syn}(B)$ (which is known now) has to be substituted, if necessary, at the corresponding argument position of every open reference $\langle\iota, j\rangle \in O_{A\to\alpha\, B\, \alpha'}(\langle\sigma, i\rangle)$.

(vi) Reduction of start productions:



Since we are only interested in the value of the meaning attribute σ_0, we compute its value using the semantic rule $\langle\sigma_0, \varepsilon\rangle = R_{S\to\alpha}(\langle\sigma_0, \varepsilon\rangle)$, and we store it as $\underline{ass}'(\langle\sigma_0, \varepsilon\rangle)$.

(vii) Final transition:



In this case, we only have to copy $\underline{ass}_1(\langle\sigma_0, \varepsilon\rangle)$, the meaning attribute value, into $\underline{ass}'(\langle\sigma_0, \varepsilon\rangle)$.

(viii) In all remaining cases:
No action is necessary because the top–down parsing automaton performs no transition.

According to this informal explanation, we are now going to construct an attributed pushdown automaton for every noncircular attribute grammar with underlying LL(1) grammar. The code generation is done by appropriate compilation schemata which build up the programs. Program pieces are joined by means of a concatenation operator $\bigodot$ which is defined as follows: For every finite ordered index set $I = \{i_1, \ldots, i_n\}$ and every I–indexed sequence $(w_i)_{i\in I}$ of words over a given alphabet,

$$\bigodot_{i\in I} w_i = w_{i_1} \ldots w_{i_n}.$$

If the ordering of I is not given explicitly, it can be chosen arbitrarily.

Definition 5.6 (Attributed top–down parsing automaton)
Let $G = (G_0, \mathcal{B}, \mathcal{T}_\Omega)$ be a noncircular attribute grammar with underlying LL(1) grammar G_0, and attribute scheme $\mathcal{B} = (\Omega, \underline{inh}, \underline{syn}, \sigma_0, R)$. The *attributed top–down parsing automaton* of G is the attributed pushdown automaton

$$ATDA(G) = (\mathcal{A}_0, \Omega, \underline{act}, REG, \rho_0)$$

which is given by:

- $\mathcal{A}_0 = TDA(G_0)$,
- $\underline{act} : \Gamma^2 \to PGM$ where

  (iii) Expansion of non–start productions:
  $\underline{act}([A \to \alpha.B\,\alpha']\gamma) = \underline{InhAttr}(A \to \alpha\ B\ \alpha', |\underline{filter}_{\in N}(\alpha)| + 1)$
  for every $B \in N$, $A \to \alpha\ B\ \alpha' \in P$, and $\gamma \in \Gamma$

  (iv) Terminal symbol match:
  $\underline{act}([A \to \alpha.a\,\alpha']\gamma) = \mathtt{COPY}(1);$
  for every $a \in \Sigma$, $A \to \alpha\ a\ \alpha' \in P$, and $\gamma \in \Gamma$

  (v) Reduction of non–start productions:
  $\underline{act}([B \to \beta.][A \to \alpha.B\,\alpha']) =$
  $\mathtt{COPY}(2); \underline{InhCopy}(B, i)$
  $\underline{SynAttr}(B \to \beta, i)\ \underline{OpenRef}(A \to \alpha\ B\alpha', i)$
  for every $B \to \beta, A \to \alpha\ B\ \alpha' \in P$ where $i = |\underline{filter}_{\in N}(\alpha)| + 1$

  (vi) Reduction of start productions:
  $\underline{act}([S \to \alpha.][\to .S]) = \underline{ExpTrans}(R_{S \to \alpha}(\langle\sigma_0, \varepsilon\rangle), \langle\sigma_0, \varepsilon\rangle, \emptyset)\ \mathtt{TOP}(\langle\sigma_0, \varepsilon\rangle);$
  for every $S \to \alpha \in P$

  (vii) Final transition:
  $\underline{act}([\to S.][\to .S]) = \mathtt{PUSH}(\langle\sigma_0, \varepsilon\rangle, 1); \mathtt{TOP}(\langle\sigma_0, \varepsilon\rangle);$

  — In all remaining cases:
  $\underline{act}(\gamma_1\gamma_2) = \varepsilon$
- $REG = \underline{att}(P)$, and
- $\rho_0 = \langle\sigma_0, \varepsilon\rangle$. $\diamond$

The compilation scheme $\underline{InhAttr}$ generates code which evaluates the inherited attribute occurrences of an expanded nonterminal symbol.

Scheme 5.1 (Evaluation of inherited attribute occurrences)
The compilation scheme

$$\underline{InhAttr} : P \times \mathbb{N} \dashrightarrow PGM$$

is given by

$$\underline{InhAttr}(p, i) = \bigodot_{\iota \in \underline{inh}(A_i)} \underline{RuleTrans}(p, \langle\iota, i\rangle)$$

for every $p = A_0 \to w_0\ A_1\ w_1 \dots A_n\ w_n \in P$, $i \in \{1, \dots, n\}$. $\diamond$

$\underline{InhAttr}$ uses the scheme $\underline{RuleTrans}$ which compiles a semantic rule and which, in its turn, calls $\underline{ExpTrans}$ to process right hand sides. In this situation, we have to deal with open references.

Scheme 5.2 (Application of semantic rules)
The compilation scheme

$$\underline{RuleTrans} : P \times \underline{att}(P) \dashrightarrow PGM$$

is given by

$\underline{RuleTrans}(p, \langle\vartheta, i\rangle) =$

if $i > max\{j \mid \langle\theta, j\rangle \in D_p(\langle\vartheta, i\rangle)\}$ **then**

(* No open reference *)

$\underline{ExpTrans}(t, \langle\vartheta, i\rangle, \emptyset)$ $\mathtt{TOP}(\langle\vartheta, \varepsilon\rangle)$;

else

(* At least one open reference *)

$\mathtt{MKAPP}(|\underline{arg}_i(t)| + 1)$; $\mathtt{TOP}(\langle\vartheta, \varepsilon\rangle)$;

$\underline{ExpTrans}(t, \langle\vartheta, i\rangle, \emptyset)$ $\mathtt{PUSH}(\langle\vartheta, \varepsilon\rangle, 3)$; $\mathtt{JOIN}(1)$;

endif

for every $p \in P$, $\langle\vartheta, i\rangle \in \underline{in}(p)$, and for $t = R_p(\langle\vartheta, i\rangle)$. $\diamond$

When compiling the right hand side of a semantic rule, we have to keep track of open references. For this purpose, $\underline{ExpTrans}$ is provided with a third parameter A being the set of all attribute occurrences for which we have already created a $\underline{nil}$ node. If we encounter such an attribute occurrence, then we only have to push a pointer to the corresponding successor of the $\underline{app}$ node. The successor position is given by the partial function

$$\underline{argpos}_p : \underline{att}(p) \times \underline{att}(p) \dashrightarrow \mathbb{N}$$

which is defined by $\underline{argpos}_p(\langle\theta, j\rangle, \langle\vartheta, i\rangle) = k$ iff $\underline{arg}_i(R_p(\langle\vartheta, i\rangle)) = w\,\langle\theta, j\rangle\,w'$ such that $k = |w| + 1$.

Scheme 5.3 (Compilation of right hand sides)
The compilation scheme

$$\underline{ExpTrans} : T_\Omega(\underline{att}(P)) \times \underline{att}(P) \times \wp(\underline{att}(P)) \dashrightarrow PGM$$

is given by

$\underline{ExpTrans}(c, \langle\vartheta, i\rangle, A) =$

$\mathtt{MKNODE}(c)$;

$\underline{ExpTrans}(f(t_1, \ldots, t_n), \langle\vartheta, i\rangle, A) =$

$$\bigodot_{j=1}^{n} \underline{ExpTrans}\left(t_j, \langle\vartheta, i\rangle, A \cup \bigcup_{k=1}^{i-1} \underline{Arg}_i(t_k)\right) \ \mathtt{MKNODE}(f);\mathtt{TOPCON}(n);$$

$\underline{ExpTrans}(\langle\theta, j\rangle, \langle\vartheta, i\rangle, A) =$

if $i = \varepsilon$ **or** $j = \varepsilon$ **or** $i > j$ **then**

(* Value is known *)

$\mathtt{PUSH}(\langle\theta, j\rangle, 1)$;

elsif $\langle\theta, j\rangle \in A$ **then**

(* $\underline{nil}$ node has been created *)

$\mathtt{PUSH}(\langle\vartheta, \varepsilon\rangle, 3)$; $\mathtt{SUCC}(\underline{argpos}_p(\langle\theta, j\rangle, \langle\vartheta, i\rangle) + 1)$;

else

(* Value is unknown *)

$\mathtt{MKNIL}$; $\mathtt{PUSH}(\langle\vartheta, \varepsilon\rangle, 3)$; $\mathtt{JOIN}(|A| + 2)$;

endif

for every $c \in \Omega^{(0)}$, $n \geq 1$, $f \in \Omega^{(n)}$, $t_1, \ldots, t_n \in T_\Omega(\underline{att}(P))$, $\langle \vartheta, i \rangle, \langle \theta, j \rangle \in \underline{att}(P)$, and $A \subset \underline{att}(P)$. For every $t \in T_\Omega(\underline{att}(P))$, and every $i \in \mathbb{N}$, $\underline{Arg}_i(t)$ denotes the set of all elements of the corresponding argument list $\underline{arg}_i(t)$. $\diamond$

When reducing a non–start production, the code sequence generated by $\underline{InhCopy}$ copies all inherited attribute values of the reduced nonterminal symbol because these may be required later again.

Scheme 5.4 (Copying inherited attribute values)
The compilation scheme

$$\underline{InhCopy} : N \times \mathbb{N} \dashrightarrow PGM$$

is given by

$$\underline{InhCopy}(B, i) = \bigodot_{\iota \in \underline{inh}(B)} (\texttt{PUSH}(\langle \iota, \varepsilon \rangle, 1); \texttt{TOP}(\langle \iota, i \rangle);)$$

for every $B \in N$, $i \in \mathbb{N}$. $\diamond$

After that, the synthesized attributes are evaluated ($\underline{SynAttr}$) and open references are resolved ($\underline{OpenRef}$), if necessary.

Scheme 5.5 (Evaluation of synthesized attributes)
The compilation scheme

$$\underline{SynAttr} : P \times \mathbb{N} \dashrightarrow PGM$$

is given by

$$\underline{SynAttr}(B \to \beta, i) = \bigodot_{\sigma \in \underline{syn}(B)} (\underline{ExpTrans}(R_{B \to \beta}(\langle \sigma, \varepsilon \rangle), \langle \sigma, \varepsilon \rangle, \emptyset)\ \texttt{TOP}(\langle \sigma, i \rangle);)$$

for every $B \to \beta \in P$, $i \in \mathbb{N}$. $\diamond$

Scheme 5.6 (Resolving open references)
The compilation scheme

$$\underline{OpenRef} : P \times \mathbb{N} \dashrightarrow PGM$$

is given by

$$\begin{aligned} \underline{OpenRef}(p, i) = \bigodot_{\sigma \in \underline{syn}(A_i)} \bigodot_{\langle \iota, j \rangle \in O_p(\langle \sigma, i \rangle)} & (\texttt{PUSH}(\langle \sigma, i \rangle, 3); \texttt{PUSH}(\langle \iota, j \rangle, 3); \\ & \texttt{MKREF}(\underline{argpos}_p(\langle \sigma, i \rangle, \langle \iota, j \rangle) + 1);) \end{aligned}$$

for every $p = A_0 \to w_0\, A_1\, w_1 \ldots A_n\, w_n \in P$, $i \in \{1, \ldots, n\}$. $\diamond$

Now, we are able to verify the construction of our attributed top–down parsing automaton by comparing its translation relation to the string–to–value translation of the attribute grammar.

Theorem 5.1 (Correctness of construction)
For any noncircular attribute grammar $G = (G_0, \mathcal{B}, \mathcal{T}_\Omega)$ with underlying LL(1) grammar G_0, the following equation holds:

$$\tau_G = \underline{term} \circ \tau_{ATDA(G)}$$

where

$$\underline{term} : DAG_{\Omega'} \dashrightarrow T_\Omega$$

is given by

$$\underline{term}(g) = \begin{cases} \underline{term}(g[\underline{succ}_1(r)]) & \text{if } \lambda(r) \in \{\underline{app}, \underline{ref}\} \\ f(\underline{term}(g[\underline{succ}_1(r)]), \ldots, \underline{term}(g[\underline{succ}_n(r)])) & \text{if } \lambda(r) = f \in \Omega^{(n)} \\ & \quad \text{and } n \in \mathbb{N} \end{cases}$$

for every Ω'–graph $g = (V, \lambda, \underline{succ}) \in DAG_{\Omega'}$ with exactly one root r. $\diamond$

Example 5.2 For the attribute grammar G described in Section 3, the attributed top–down parsing automaton $ATDA(G) = (\mathcal{A}_0, \Omega, \underline{act}, REG, \rho_0)$ looks as follows:

- $\mathcal{A}_0 = TDA(G_0)$ (cf. Section 2),
- $\Omega = \{zero^{(0)}, dec^{(1)}, inc^{(1)}, exp^{(1)}, add^{(2)}\}$,
- $\underline{act} : \Gamma^2 \to PGM$ where

  (iii) Expansion of non–start productions:

  $\underline{act}([S \to .L]\gamma)$
  $= \underline{InhAttr}(S \to L, 1)$
  $= \underline{RuleTrans}(S \to L, \langle p, 1\rangle)$
  $=$ $\texttt{MKAPP}(2); \texttt{TOP}(\langle p, \varepsilon\rangle);$
  $\underline{ExpTrans}(dec(\langle l, 1\rangle), \langle p, 1\rangle, \emptyset)$
  $\texttt{PUSH}(\langle p, \varepsilon\rangle, 3); \texttt{JOIN}(1);$
  $=$ $\texttt{MKAPP}(2); \texttt{TOP}(\langle p, \varepsilon\rangle);$
  $\texttt{MKNIL}; \texttt{PUSH}(\langle p, \varepsilon\rangle, 3); \texttt{JOIN}(2); \texttt{MKNODE}(dec); \texttt{TOPCON}(1);$
  $\texttt{PUSH}(\langle p, \varepsilon\rangle, 3); \texttt{JOIN}(1);$

  $\underline{act}([L \to .B\ L]\gamma)$
  $= \underline{InhAttr}(L \to B\ L, 1)$
  $= \underline{RuleTrans}(L \to B\ L, \langle p, 1\rangle)$
  $= \underline{ExpTrans}(\langle p, \varepsilon\rangle, \langle p, 1\rangle, \emptyset)\ \texttt{TOP}(\langle p, \varepsilon\rangle);$
  $= \texttt{PUSH}(\langle p, \varepsilon\rangle, 1); \texttt{TOP}(\langle p, \varepsilon\rangle);$

  $\underline{act}([L \to B.L]\gamma)$
  $= \texttt{PUSH}(\langle p, \varepsilon\rangle, 1); \texttt{MKNODE}(dec); \texttt{TOPCON}(1); \texttt{TOP}(\langle p, \varepsilon\rangle);$

  for every $\gamma \in \Gamma$

  (iv) Terminal symbol match:

  $\underline{act}([B \to .0]\gamma) = \texttt{COPY}(1);$
  $\underline{act}([B \to .1]\gamma) = \texttt{COPY}(1);$

  for every $\gamma \in \Gamma$

  (v) Reduction of non–start productions:

  $\underline{act}([L \to B\ L.][S \to .L])$

$= \mathtt{COPY}(2);\, \underline{InhCopy}(L,1)\ \underline{SynAttr}(L \to B\ L, 1)$
$\underline{OpenRef}(S \to L, 1)$
$= \mathtt{COPY}(2);\, \mathtt{PUSH}(\langle p,\varepsilon\rangle, 1);\, \mathtt{TOP}(\langle p,1\rangle);$
$\underline{ExpTrans}(inc(\langle l,2\rangle), \langle l,\varepsilon\rangle, \emptyset)\ \mathtt{TOP}(\langle l,1\rangle);$
$\underline{ExpTrans}(add(\langle v,1\rangle, \langle v,2\rangle), \langle v,\varepsilon\rangle, \emptyset)\ \mathtt{TOP}(\langle v,1\rangle);$
$\mathtt{PUSH}(\langle l,1\rangle, 3);\, \mathtt{PUSH}(\langle p,1\rangle, 3);\, \mathtt{MKREF}(2);$
$= \mathtt{COPY}(2);\, \mathtt{PUSH}(\langle p,\varepsilon\rangle, 1);\, \mathtt{TOP}(\langle p,1\rangle);$
$\mathtt{PUSH}(\langle l,2\rangle, 1);\, \mathtt{MKNODE}(inc);\, \mathtt{TOPCON}(1);\, \mathtt{TOP}(\langle l,1\rangle);$
$\mathtt{PUSH}(\langle v,1\rangle, 1);\, \mathtt{PUSH}(\langle v,2\rangle, 1);$
$\mathtt{MKNODE}(add);\, \mathtt{TOPCON}(2);\, \mathtt{TOP}(\langle v,1\rangle);$
$\mathtt{PUSH}(\langle l,1\rangle, 3);\, \mathtt{PUSH}(\langle p,1\rangle, 3);\, \mathtt{MKREF}(2);$

$\underline{act}([L \to B\ L.][L \to B.L])$
$= \mathtt{COPY}(2);\, \mathtt{PUSH}(\langle p,\varepsilon\rangle, 1);\, \mathtt{TOP}(\langle p,2\rangle);$
$\mathtt{PUSH}(\langle l,2\rangle, 1);\, \mathtt{MKNODE}(inc);\, \mathtt{TOPCON}(1);\, \mathtt{TOP}(\langle l,2\rangle);$
$\mathtt{PUSH}(\langle v,1\rangle, 1);\, \mathtt{PUSH}(\langle v,2\rangle, 1);$
$\mathtt{MKNODE}(add);\, \mathtt{TOPCON}(2);\, \mathtt{TOP}(\langle v,2\rangle);$

$\underline{act}([L \to .][S \to .L])$
$= \mathtt{COPY}(2);\, \mathtt{PUSH}(\langle p,\varepsilon\rangle, 1);\, \mathtt{TOP}(\langle p,1\rangle);$
$\mathtt{MKNODE}(zero);\, \mathtt{TOP}(\langle l,1\rangle);\, \mathtt{MKNODE}(zero);\, \mathtt{TOP}(\langle v,1\rangle);$
$\mathtt{PUSH}(\langle l,1\rangle, 3);\, \mathtt{PUSH}(\langle p,1\rangle, 3);\, \mathtt{MKREF}(2);$

$\underline{act}([L \to .][L \to B.L])$
$= \mathtt{COPY}(2);\, \mathtt{PUSH}(\langle p,\varepsilon\rangle, 1);\, \mathtt{TOP}(\langle p,2\rangle);$
$\mathtt{MKNODE}(zero);\, \mathtt{TOP}(\langle l,2\rangle);\, \mathtt{MKNODE}(zero);\, \mathtt{TOP}(\langle v,2\rangle);$

$\underline{act}([B \to 0.][L \to .B\ L])$
$= \mathtt{COPY}(2);\, \mathtt{PUSH}(\langle p,\varepsilon\rangle, 1);\, \mathtt{TOP}(\langle p,1\rangle);$
$\mathtt{MKNODE}(zero);\, \mathtt{TOP}(\langle v,1\rangle);$

$\underline{act}([B \to 1.][L \to .B\ L])$
$= \mathtt{COPY}(2);\, \mathtt{PUSH}(\langle p,\varepsilon\rangle, 1);\, \mathtt{TOP}(\langle p,1\rangle);$
$\mathtt{PUSH}(\langle b,\varepsilon\rangle, 1);\, \mathtt{PUSH}(\langle p,\varepsilon\rangle, 1);\, \mathtt{MKNODE}(exp);$
$\mathtt{TOPCON}(2);\, \mathtt{TOP}(\langle v,1\rangle);$

(vi) Reduction of start productions:
$\underline{act}([S \to L.][\to .S])$
$= \underline{ExpTrans}(\langle v,1\rangle, \langle v,\varepsilon\rangle, \emptyset)\ \mathtt{TOP}(\langle v,\varepsilon\rangle);$
$= \mathtt{PUSH}(\langle v,1\rangle, 1);\, \mathtt{TOP}(\langle v,\varepsilon\rangle);$

(vii) Final transition:
$\underline{act}([\to S.][\to .S]) = \mathtt{PUSH}(\langle v,\varepsilon\rangle, 1);\, \mathtt{TOP}(\langle v,\varepsilon\rangle);$

- $REG = \underline{att}(P) = \{\langle \vartheta, i\rangle \mid \vartheta \in \{p,l,v\}, i \in \{\varepsilon,1,2\}\}$, and
- $\rho_0 = \langle \sigma_0, \varepsilon\rangle = \langle v, \varepsilon\rangle$.

Figures 13 to 15 illustrate the computation of the string–to–value translation for the input string 1$. The pushdown grows downwards, and the registers (attribute occurrences) of each pushdown entry are represented by the corresponding attributed production where recent evaluations result in adding a pointer to the graph. $\diamond$

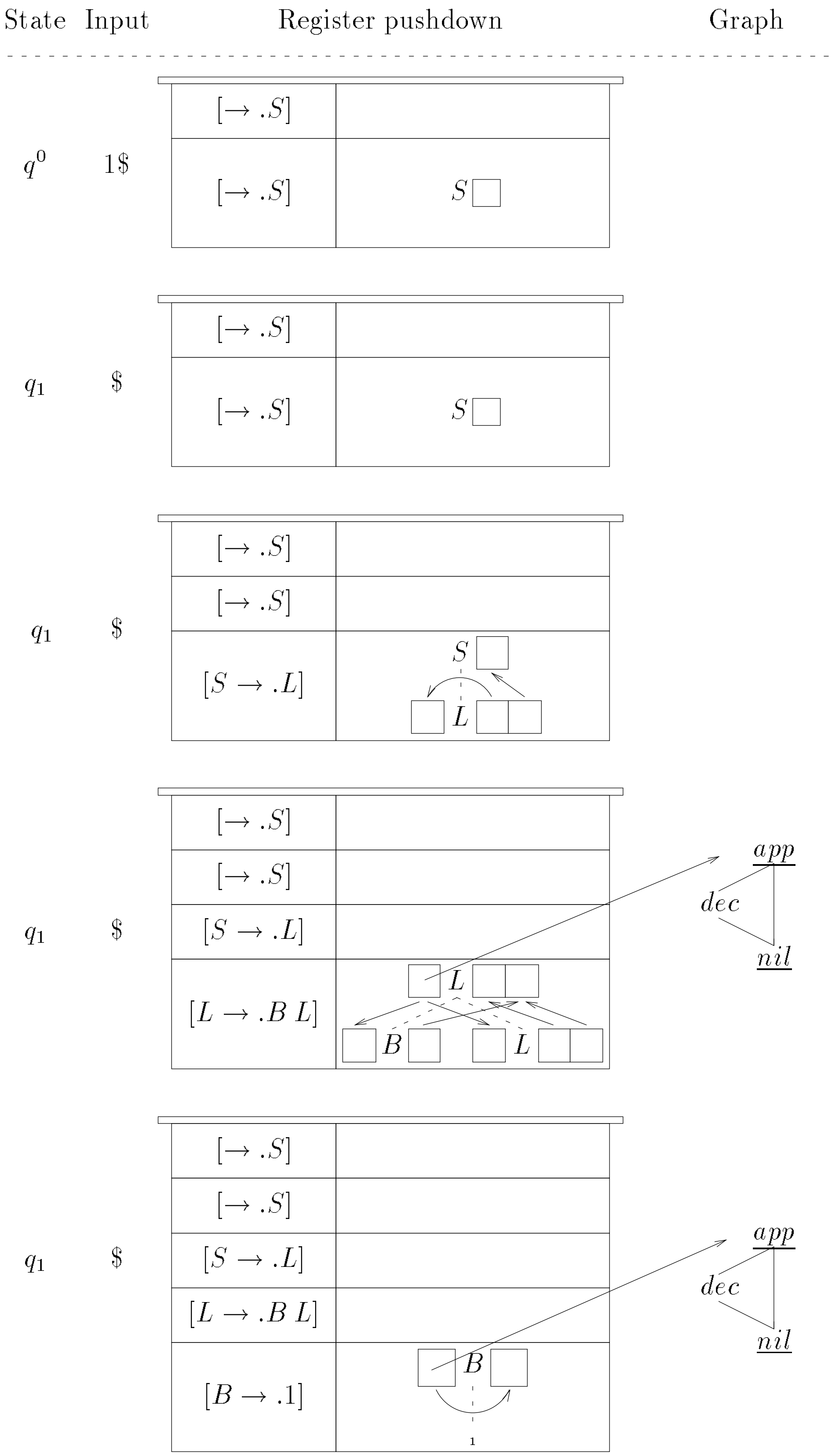


Figure 13: Computation protocol of $ATDA(G)$.

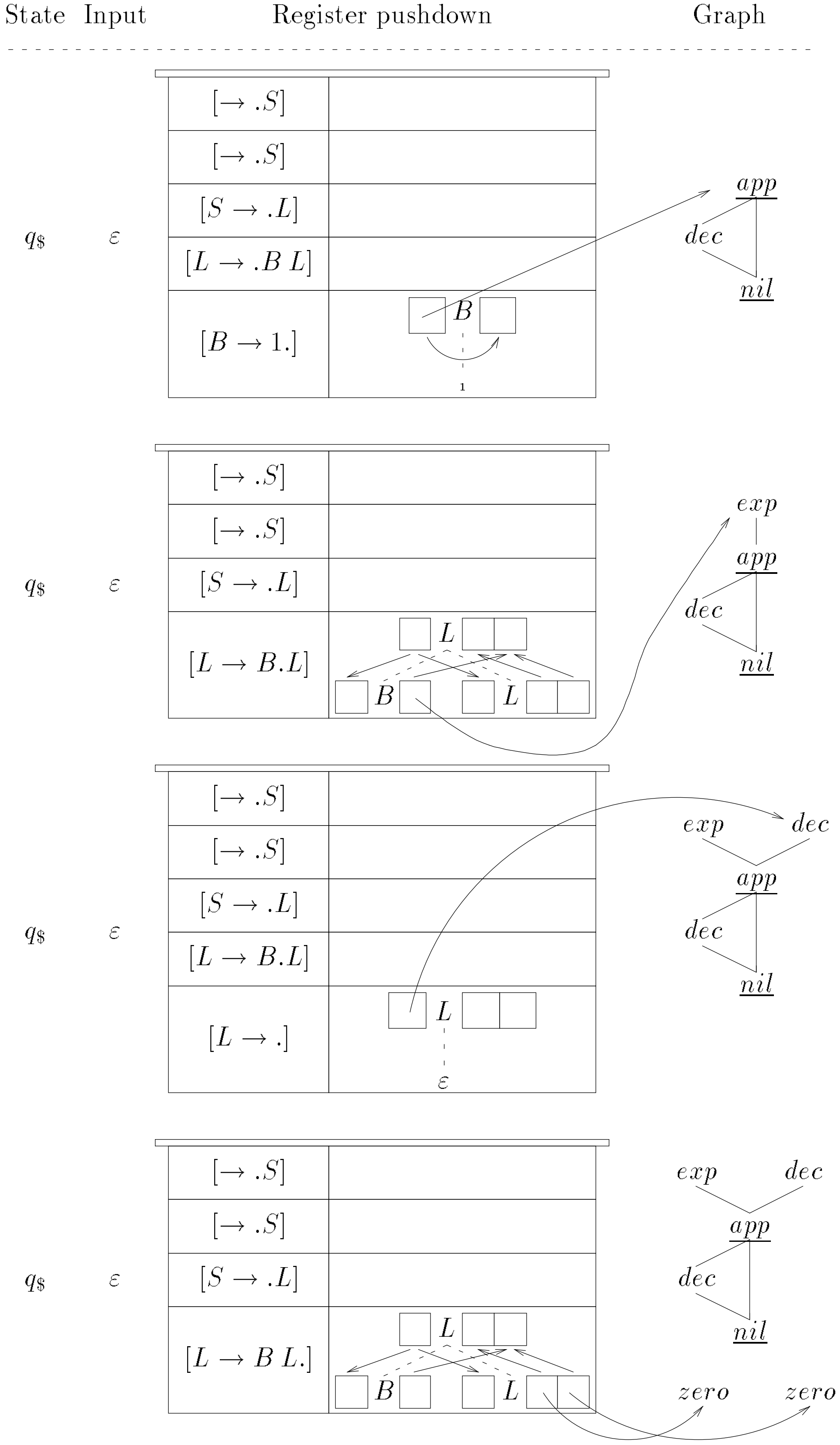


Figure 14: Computation protocol of $ATDA(G)$ (continued).

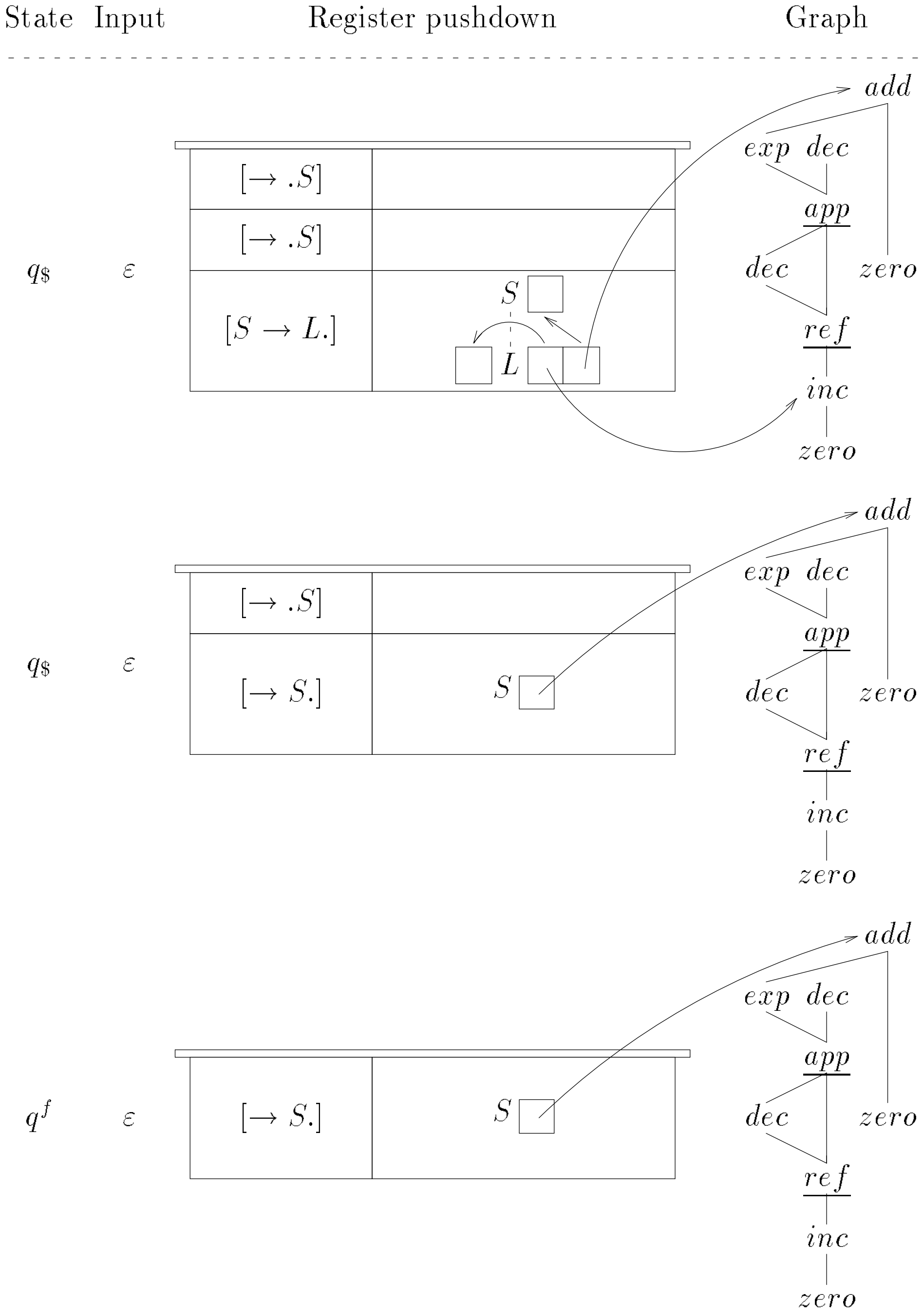


Figure 15: Computation protocol of $ATDA(G)$ (continued).

5.4 Space consumption

From the point of view of memory requirements, our approach has the advantage that no syntax–tree has to be stored during parsing and attribute evaluation. However, it makes use of a graph representation for semantic values; every application of an operation symbol results in the creation of a corresponding graph node. Furthermore, $\underline{nil}$ and $\underline{ref}$ nodes have been introduced for the management of incomplete values.

In comparison to syntax–tree based solutions, one can state the following: On one hand, a frequent use of operation symbols in the semantic rules may cause the graph to require as much space as the corresponding syntax tree decorated with attribute values. On the other hand, many semantic rules occurring in practical attribute grammars are generally so–called *copy rules* which only have an attribute occurrence on their right hand side. This kind of rule is handled very space–efficiently by our method since only the pointer has to be copied from the right hand side to the left hand side attribute without doubling the semantic value. Furthermore, the memory space allocated to the right hand side attribute will be disposed later if the corresponding syntactic production has completely been analyzed. Thus, the stack technique of the top–down parser approximates the *lifetime* of the attribute occurrences (cf. [14]).

6 Implementation

In order to demonstrate the practical applicability of our approach, we have developed a test implementation on a PC platform. The compiler reads the specification of an attribute grammar (productions, attributes, semantic rules) given in a predefined syntax, and it creates a MODULA–2 source program which simulates the corresponding attributed top–down parsing automaton.

Here, we refrain from a description of the program's usage. Instead, we exemplify that the formal construction of the attributed top–down parsing automaton in Sections 5.2 and 5.3 made it possible to implement the compiler in a systematical way.

The code for the attributed top–down parsing automaton generated from an attribute grammar specification consists of fixed parts and of variant (i.e. grammar–dependent) parts. The former comprise basic data structures and the implementation of the machine instructions. As an example, we consider an extract of the structures which encodes the instantaneous descriptions of the attributed top–down parsing automaton (cf. Definition 5.6):

```
InstDesc = RECORD                      (* Instantaneous description *)
  q: State;
  w: TerminalString;
  s: PushDown;
END;
State = CHAR;                          (* Current state *)
TerminalString = ARRAY [1..80] OF CHAR;  (* Input string *)
PushDown = POINTER TO PDEntry;         (* Parsing stack =  *)
PDEntry = RECORD                       (* list of          *)
  I: LR0Item;                          (* LR(0) items +    *)
  ass: AttrAss;                        (* attribute values *)
  next: PushDown;
```

```
END;
LR0Item = RECORD                     (* LR(0) item =            *)
  p: ProdNo;                         (* number of production +  *)
  i: ProdIndex;                      (* index within production *)
END;
AttrAss = POINTER TO AttrAssEntry;   (* Attribute value assignment = *)
AttrAssEntry = RECORD                (* list of                      *)
  ao: AttrOcc;                       (* attribute occurrence/        *)
  g: Graph;                          (* value pairs                  *)
  next: AttrAss;
END;
AttrOcc = RECORD                     (* Attribute occurrence =     *)
  att: Attribute;                    (* attribute name +           *)
  pos: ProdIndex;                    (* position within production *)
END;
Graph = POINTER TO GraphNode;        (* Graph in heap *)
GraphNode = RECORD
  lab: OpSymbol;                     (* Operation symbol *)
  succ: ARRAY Rank OF Graph;         (* List of successors *)
END;
```

A second example of a grammar–independent part is the following implementation of the machine instructions `TOP` and `MKNODE` (cf. Definition 5.3):

```
(* ppd is a global variable pointing to the global pointer pushdown *)
PROCEDURE TOP (ao: AttrOcc);
  VAR ass: AttrAss; ppd1: PointerPushDown;
  BEGIN
    (* Allocate new assignment entry for attribute occurrence ao *)
    NEW (ass); ass^.ao := ao; ass^.g := ppd^.g;
    (* Insert at head of list *)
    ass^.next := ass[3]; ass[3] := ass;
    (* Pop pointer pushdown *)
    ppd1 := ppd^.next; DISPOSE (ppd); ppd := ppd1;
  END TOP;
PROCEDURE MKNODE (f: OpSymbol);
  VAR ppd1: PointerPushDown; g: Graph;
  BEGIN
    (* Allocate new graph node *)
    NEW (g); g^.lab := f;
    (* Push this node onto the pointer pushdown *)
    NEW (ppd1); ppd1^.g := g; ppd1^.next := ppd; ppd := ppd1;
  END MKNODE;
```

The variant parts of the code comprise attribute and operation symbol tables as well as encodings of the transition function δ and of the assignment of programs *act*. The latter are generated by means of the compilation schemes described in Section 5.3. As an example, we consider the compiler's source code of the compilation scheme $\underline{ExpTrans}$ (cf. Scheme 5.3):

```
PROCEDURE ExpTrans (f: File; t: Term; ao: AttrOcc; pos: CARDINAL);
  VAR
    i, pos1: CARDINAL;
  BEGIN
    (* Consider the top-level symbol of t *)
    CASE GetTag (t^.ref) OF
    | op:
      (* Operation symbol *)
      IF GetRank (t^.ref) = 0 THEN
        (* Constant: emit MKNODE instruction *)
        WrStr (f, "MKNODE ("); WrCard (f, t^.ref, 1);
        WrStr (f, ");");
      ELSE
        (* Non-constant: emit MKNODE and TOPCON instructions *)
        pos1 := pos;
        FOR i := 1 TO GetRank (t^.ref) DO
          ExpTrans (f, t^.succ[i], ao, pos1);
          pos1 := pos1 + VarNo (t^.succ[i]);
        END;
        WrStr (f, "MKNODE (");  WrCard (f, t^.ref, 1);
        WrStr (f, "); TOPCON ("); WrCard (f, GetRank (t^.ref), 1);
        WrStr (f, ");");
      END;
    | att:
      (* Outside attribute *)
      IF (ao.pos = 0) OR (ao.pos > t^.pos) THEN
        (* Value is known: emit PUSH instruction *)
        WrStr (f, "PUSH ("); WriteReg (f, t^.ref, t^.pos);
        WrStr (f, ", 1);");
      ELSE
        (* Value is unknown: create nil node *)
        WrStr (f, "MKNIL; PUSH ("); WriteReg (f, ao.ref, 0);
        WrStr (f, ", 3); JOIN ("); WrCard (f, pos, 1); WrStr (f, ");");
      END;
    END;
  END ExpTrans;
```

As one can see, there is a direct correspondence between the formal description and the Modula–2 source code.

7 From the point of view of logic programming

In this section we want relate the problem of combining parsing and attribute evaluation to the concept of logic programming. There are two ways to proceed: first, embed the combination problem into the world of logic programming and use the well–known evaluation machineries, and second, start from attribute grammars and enrich this concept by logical variables. Let us briefly discuss both approaches.

It has been shown in [9] how an attribute grammar can be transformed into a logic program, more precisely, into a definite program [19]. Let us recall the productions and the semantic rules of our example attribute grammar G (cf. Example 2.1 and Example 3.1) and apply this transformation to G.

$\pi_1 : S \to L$	$\langle p,1\rangle = dec(\langle l,1\rangle)$ $\langle v,\varepsilon\rangle = \langle v,1\rangle$	$\pi_3 : L \to \varepsilon$	$\langle l,\varepsilon\rangle = zero$ $\langle v,\varepsilon\rangle = zero$
$\pi_2 : L \to B\ L$	$\langle p,1\rangle = \langle p,\varepsilon\rangle$ $\langle p,2\rangle = dec(\langle p,\varepsilon\rangle)$ $\langle l,\varepsilon\rangle = inc(\langle l,2\rangle)$ $\langle v,\varepsilon\rangle = add(\langle v,1\rangle,\langle v,2\rangle)$	$\pi_4 : B \to 0$ $\pi_5 : B \to 1$	$\langle v,\varepsilon\rangle = zero$ $\langle v,\varepsilon\rangle = exp(\langle p,\varepsilon\rangle)$

In the transformation of G into a definite program, every nonterminal A of the context–free grammar is viewed as a predicate A; the rank of the predicate A is equal to the number of attributes associated to the nonterminal A. The transformation proceeds production by production. Let us illustrate this transformation by applying three steps to production π_2.

In the first step, the production is changed into a clause (i.e., the arrow is reversed) and to every nonterminal an argument list is added which contains as many argument positions as there are attributes associated; thus $L \to B\ L$ is transformed into

$$L(-,-,-) \leftarrow B(-,-)\ L(-,-,-).$$

In the second step, the outside attribute occurrences are put into the corresponding places:

$$L(p,-,-) \leftarrow B(-,v)\ L(-,l,v').$$

In the third step, the right–hand sides of the semantic rules for inside attribute occurrences are put into the corresponding places:

$$L(p,inc(l),add(v,v')) \leftarrow B(p,v)\ L(dec(p),l,v').$$

In order to take care of the problem under concern, i.e., combination of parsing and attribute evaluation, we enrich every predicate by one more argument by means of which the parsing of an input string w can be described. The string w is represented by a comb which grows to the right, e.g., the string 11 is represented by the comb $\sigma(1,\sigma(1,\alpha))$ where σ and α are additional operation symbols in Ω of rank 2 and 0, respectively. Thus, the production π_2 is transformed into the following clause where b and w are considered as variables:

$$L(p,inc(l),add(v,v'),\sigma(b,w)) \leftarrow B(p,v,b)\ L(dec(p),l,v',w).$$

In total we obtain the following definite program $DP(G)$:

$$\begin{array}{lcl}
S(v,w) & \leftarrow & L(dec(l),l,v,w) \\
L(p,inc(l),add(v,v'),\sigma(b,w)) & \leftarrow & B(p,v,b)\ L(dec(p),l,v',w) \\
L(p,zero,zero,\alpha) & \leftarrow & \varepsilon \\
B(p,zero,0) & \leftarrow & \varepsilon \\
B(p,exp(p),1) & \leftarrow & \varepsilon
\end{array}$$

Next we show an SLD–refutation starting from the definite goal

$$S(\langle v,\varepsilon\rangle,\sigma(1,\sigma(1,\alpha)))$$

where free variables occurring in goals correspond to the attribute occurrences in the syntax tree of the input word 11. We always select the leftmost literal fo the current definite goal for further resolution. For every resolution step we have indicated the most general unifier of the selected literal and the right-hand side of the appropriate clause; moreover, we show the schematic approximation $t_{\langle v,\varepsilon\rangle}$ of the free variable $\langle v,\varepsilon\rangle$ in the initial definite goal.

$$\leftarrow\ S(\langle v,\varepsilon\rangle,\sigma(1,\sigma(1,\alpha)))$$

$$\begin{array}{rcl}
v & \rightarrow & \langle v,1\rangle, \\
w & \rightarrow & \sigma(1,\sigma(1,\alpha)), \\
l & \rightarrow & \langle l,1\rangle
\end{array}$$

$$\leftarrow\ L(dec(\langle l,1\rangle),\langle l,1\rangle,\langle v,1\rangle,\sigma(1,\sigma(1,\alpha)))$$

$$\begin{array}{rcl}
p & \rightarrow & dec(inc(\langle l,12\rangle)),\langle l,1\rangle \rightarrow inc(\langle l,12\rangle), \\
l & \rightarrow & \langle l,12\rangle, \\
v & \rightarrow & \langle v,11\rangle, v' \rightarrow \langle v,12\rangle, \langle v,1\rangle \rightarrow add(\langle v,11\rangle,\langle v,12\rangle), \\
b & \rightarrow & 1, w \rightarrow \sigma(1,\alpha), \\
t_{\langle v,\varepsilon\rangle} & = & \langle v,1\rangle
\end{array}$$

$$\leftarrow\ B(dec(inc(\langle l,12\rangle)),\langle v,11\rangle,1),\ L(dec^2(inc(\langle l,12\rangle)),\langle l,12\rangle,\langle v,12\rangle,\sigma(1,\alpha))$$

$$\begin{array}{rcl}
p & \rightarrow & dec(inc(\langle l,12\rangle)), \\
\langle v,11\rangle & \rightarrow & exp(dec(inc(\langle l,12\rangle))), \\
t_{\langle v,\varepsilon\rangle} & = & add(\langle v,11\rangle,\langle v,12\rangle)
\end{array}$$

$$\leftarrow\ L(dec^2(inc(\langle l,12\rangle)),\langle l,12\rangle,\langle v,12\rangle,\sigma(1,\alpha))$$

$$\begin{array}{rcl}
p & \rightarrow & dec^2(inc^2(\langle l,122\rangle)),\langle l,12\rangle \rightarrow inc(\langle l,122\rangle), \\
l & \rightarrow & \langle l,122\rangle, \\
v & \rightarrow & \langle v,121\rangle, v' \rightarrow \langle v,122\rangle, \langle v,12\rangle \rightarrow add(\langle v,121\rangle,\langle v,122\rangle), \\
b & \rightarrow & 1, w \rightarrow \varepsilon, \\
t_{\langle v,\varepsilon\rangle} & = & add(exp(dec(inc(\langle l,1\rangle))),\langle v,12\rangle)
\end{array}$$

$$\leftarrow\ B(dec^2(inc^2(\langle l,122\rangle)),\langle v,121\rangle,1),\ L(dec^3(inc^2(\langle l,122\rangle)),\langle l,122\rangle,\langle v,12\rangle,\varepsilon)$$

$$\begin{array}{rcl}
p & \rightarrow & dec^2(inc^2(\langle l,122\rangle)), \\
\langle v,121\rangle & \rightarrow & exp(dec^2(inc^2(\langle l,122\rangle))), \\
t_{\langle v,\varepsilon\rangle} & = & add(exp(dec(inc(\langle l,1\rangle))),add(\langle v,121\rangle,\langle v,122\rangle))
\end{array}$$

$$\leftarrow\ L(dec^3(inc^2(\langle l,122\rangle)),\langle l,122\rangle,\langle v,122\rangle,\varepsilon)$$

$$
\begin{array}{rcl}
p & \rightarrow & dec^3(inc^2(zero)), \\
\langle l, 122\rangle & \rightarrow & zero, \\
\langle v, 122\rangle & \rightarrow & zero, \\
t_{\langle v,\varepsilon\rangle} & = & add(exp(dec(inc(\langle l,1\rangle))), add(exp(dec^2(inc^2(\langle l,1\rangle))), \langle v,122\rangle))
\end{array}
$$

$$\leftarrow \quad \varepsilon$$

$$t_{\langle v,\varepsilon\rangle} = add(exp(dec(inc^2(zero))), add(exp(dec^2(inc^2(zero))), zero))$$

Thus, the literal $S(t_{\langle v,\varepsilon\rangle}, \sigma(1, \sigma(1, \alpha)))$ is a consequence of the definite program $DP(G)$.

Actually, this computation is deterministic in the sense that to every literal exactly one definite program clause is applicable. The determinism is due to the fact that the underlying context–free grammar is $LL(1)$ and to the way in which we have integrated parsing into the definite program.

Clearly, one could translate the definite program $DP(G)$ into code for some kind of a Warren–abstract–machine [23] and then start the machine with the (tree representation of the) input string w. The machine would compute the value of the attribute v at the root of the syntax tree of w, and thus, it would solve the combination problem. However, this approach is too inefficient, because the Warren–abstract–machine would follow its indexing scheme (using `TRY-ME-ELSE`, `RETRY-ME-ELSE`, and `TRUST-ME-ELSE-FAIL` commands) without using the fact that the parsing and hence, the choice of definite clauses can be done deterministically. Let us now briefly consider the second way to relate the combination problem to logic programming; this is due to J. Paakki.

In [21] a new formalism called *logical one–pass attribute grammar* has been introduced which "makes it possible to evaluate even counter–one–pass attributes during parsing" (p. 204, [21]). The idea is to start from the usual, well–known concept of attribute grammar and to introduce logical attributes to deal with incomplete semantic information. The power of this approach is the fact that, in contrast to classical evaluation schemes of attribute grammars, logical attributes do not have to receive their final value immediately during evaluation, rather they may contain such incomplete semantic information which is updated (perhaps in several steps) until the final value is computed at the end of the whole evaluation process. Paakki suggests an evaluation scheme for logical one–pass attribute grammars which is a refinement of the usual evaluation scheme for L–attributed grammars. As implementation vehicle, Paakki developed a system called PROFIT (PROlog dialect For Implementing Translators) [20] which is "currently translated into Prolog" (p. 216, [21]).

Obviously, the approach of Paakki and our algorithm presented in Section 4 are essentially the same. The concept of incomplete semantic information corresponds to our concept of schematic approximation, and the idea of the refinement of the evaluation scheme for L–attributed grammars is the same in both approaches. However, in the present paper we formalize a concrete abstract machine (the attributed top–down parsing automaton in Section 5 which is based on a deterministic pushdown automaton, for the implementation of this refinement, whereas Paakki implements his evaluation scheme on a WAM–like abstract machine.

8 Conclusions

Attribute grammars are a useful and intuitively appealing method for specifying the semantics of context–free languages. We have presented an algorithm which is able to evaluate all attribute occurrences of a syntax tree during a single top–down left–to–right treewalk. This algorithm was implemented by extending the top–down parsing automaton of the underlying context–free grammar to a parsing–evaluating automaton, called attributed top–down parsing automaton, which performs both parsing and attribute evaluation simultaneously.

There are some optimizations and extensions of our approach which one can think of in order to improve both efficiency and computing power:

- The present version of our algorithm computes the value of every attribute occurrence of the current production. Instead, we could confine ourselves to evaluate only the *useful* attribute occurrences, i.e. those whose values contribute to the value of the meaning attribute at the root of the tree. In [22], this optimization has been formalized for the Kennedy–Warren algorithm [15] and it has been called the *output–oriented approach*.

  Note that the set of all useful attribute occurrences of the syntax tree can not be determined statically at compile–time because it depends on the composition of the tree. Instead, only an upper–bound estimation is possible.

- In order to augment the set of all context–free languages that can be parsed with our method, one could take into consideration to use *bottom–up* (or: *LR*) parsing. (Recall that the set of all languages generated by LL grammars is properly contained in the set of all languages generated by LR(1) grammars.) In [3], one can find a survey of the various subclasses of L–attributed grammars for LR parsing.

Acknowledgements

The authors are grateful to one of the referees who pointed out the connections between attribute grammars and logic programming.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison–Wesley, 1986.

[2] Rieks op den Akker. Deterministic Parsing of Attribute Grammars, Part I: Top–Down Strategies. memorandum I F-86-19, Onderafdeling der Informatica, Technische Hogeschool Twente, 1986.

[3] Rieks op den Akker, Bořivoj Melichar, and Jorma Tarhio. The Hierarchy of LR–Attributed Grammars. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*, pages 13–28. Springer–Verlag, September 1990.

[4] Henk Alblas and Bořivoj Melichar, editors. *Attribute Grammars, Applications and Systems (SAGA)*, volume 545 of *Lecture Notes in Computer Science*. Springer–Verlag, June 1991.

[5] Gregor V. Bochmann. Semantic Evaluation From Left to Right. *Communications of the ACM*, 19(2):55–62, February 1976.

[6] Laurian M. Chirica and David F. Martin. An Order-Algebraic Definition of Knuthian Semantics. *Mathematical Systems Theory*, 13(1):1–27, 1979.

[7] Pierre Deransart and Martin Jourdan, editors. *Attribute Grammars and Their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*. Springer–Verlag, September 1990.

[8] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*, volume 323 of *Lecture Notes in Computer Science*. Springer–Verlag, August 1988.

[9] Pierre Deransart and Jan Maluszynski. Relating logic programs and attribute grammars. *Journal of Logic Programming*, 2(2):119–155, 1985.

[10] Joost Engelfriet. Attribute Grammars: Attribute Evaluation Methods. In Bernard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 103–138. Cambridge University Press, 1984.

[11] Joost Engelfriet and Gilberto Filè. The Formal Power of One–Visit Attribute Grammars. *Acta Informatica*, 16(3):275–302, 1981.

[12] Joost Engelfriet and Heiko Vogler. Pushdown machines for the macro tree transducer. *Theoretical Computer Science*, 42:251–367, 1986.

[13] R. Giegerich and R. Wilhelm. Counter–one–pass features in one–pass compilation: a formalisation using attribute grammars. *Information Processing Letters*, 7:279–284, 1978.

[14] U. Kastens. Lifetime analysis for attributes. *Acta Informatica*, 24:633–651, 1987.

[15] Ken Kennedy and Scott K. Warren. Automatic Generation of Efficient Evaluators for Attribute Grammars. In *3rd ACM POPL*, pages 32–49. ACM, January 1976.

[16] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[17] Donald E. Knuth. Semantics of Context-Free Languages: Correction. *Mathematical Systems Theory*, 5(1):95–96, June 1971.

[18] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. Attributed translations. *Journal of Computer and System Sciences*, 9(3):279–307, December 1974.

[19] J.W. Lloyd. *Foundations of Logic Programming*. Springer–Verlag, 1987.

[20] J. Paakki. A Prolog-based Compiler Writing Tool. In *Proc. of the Workshop of Compiler Compilers and High Speed Compilation, Berlin*, 1988. Report 3/1989, Akademie der Wissenschaften der DDR.

[21] J. Paakki. A logic–based modification of attribute grammars for practical compiler writing. In *Proc. of the 7th Int. Conference on Logic Programming*, pages 203–217, 1990.

[22] Mikko Saarinen. On Constructing Efficient Evaluators for Attribute Grammars. In G. Ausiello and C. Böhm, editors, *5th ICALP*, volume 62 of *Lecture Notes in Computer Science*, pages 382–397. Springer–Verlag, July 1978.

[23] D.H.D. Warren. An abstract PROLOG instrution machine. Technical report, SRI International, 1983. Technical Report No. 309.