中華民國八十四年全國計算機會議論文集
Proceedings of National Computer Symposium 1995

剖析時屬性計算

Evaluating Attributes During Parsing

歐陽士庭　吳培基　王豐堅
Sting Ouyang　Pei-Chi Wu　Feng-Jian Wang

國立交通大學資訊工程研究所
Department of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, Taiwan, R.O.C.
{stoyang, pcwu, fjwang}@csie.nctu.edu.tw

摘　要

屬性文法產生的編譯器往往效率不及手寫的編譯器。其原因之一是大部份屬性計算皆在剖析完成後才開始。本文提出將屬性計算的程式碼插入剖析動作中的演算法。首先我們提出將語法樹建構序列(tree construction sequence)和遊歷序列(visit sequence)合併在一起的演算法。由此演算法產生的序列可直接配合LL(*k*)剖析使用。對於LR(*k*)語法，由此產生的序列，還需經過額外的步驟，以排入適當的空閒位置(free position)。

關鍵字：屬性文法、遊歷導向的屬性計算(visit-oriented attribute evaluators)、剖析器產生器。

ABSTRACT

Compilers automatically generated from attribute grammars (AGs) are usually slower than hand-coded ones. One of the reasons is that most attribute evaluators are invoked after parsing. This paper proposes two algorithms to insert attribute evaluation codes of AGs into parsing actions. The first algorithm combines tree construction sequences (parsing actions) and attribute evaluation sequences (visit sequences of AGs) into a *parse-time visit sequence*. The resulting sequence can be directly used in top-down parsing. For bottom-up parsing, another algorithm introduces additional steps to schedule the visit sequence according to the *free* positions in the grammar.

Keywords: attribute grammars, visit-oriented attribute evaluators, parser generators.

1 Introduction

Attribute grammars (AGs) [Alb91] are a formal method for compiler specification. AGs associate program semantics with syntax by attaching a set of attributes to the symbols of a context-free grammar (CFG). Attribute values are defined using attribution rules associated with the productions of the CFG. *Attribute evaluators* evaluate attribution rules to get the semantics of the program. There are two categories of evaluators: *dynamic* and *static*. A dynamic evaluator determines the evaluation order of attributes by a topological sort on an attribute dependency graph. The evaluation order in a static evaluator is pre-determined, so there is no need to perform dependency analyses at runtime. Static evaluators take less computing time and memory storage than dynamic ones.

Using compiler generating tools such as AG generators is a fast and easy way to write a compiler. However, a compiler automatically generated is usually slower than a hand-coded one. One of the reasons is that most static attribute evaluators are invoked after parsing, and the whole parse tree for a program is constructed even if the program can be analyzed in one pass. By contrast, a hand-coded compiler may perform semantic actions during parsing. Such parse-time actions can reduce memory storage and compiling time, because the number of tree traversals can be reduced by one and the subtrees evaluated can be freed early.

There are several methods for static parse-time attribute evaluation. Lewis, Rosenkrantz, and Stearns [LRS74] introduced the idea of *L-attributed AGs*, where inherited and synthesized attributes are evaluated through one left-to-right traversal over the parse tree. L-attributed AGs can be easily applied to top-down parsing. In LR parsing, because the parse tree is constructed bottom up, it is difficult to evaluate the inherited attributes of a symbol before the symbol is reduced. Watt [Wat77] explored a method to evaluate both synthesized and inherited attributes during LR parsing for L-attributed AGs. The method stores the attributes in an attribute stack and adds new ε-productions for maintaining

attribute stack. Unfortunately, adding these productions to an LR(k) grammar may make the grammar not LR(k) anymore. Purdom and Brown [PuB80] presented an algorithm to find all *free* positions, where new ε-productions and semantic actions can be added without changing the grammar class. Kastens [Kas91] presented the idea of parse-time attribute evaluation, both top-down and bottom-up cases. He proposed a two-phase evaluation method for bottom-up parsing: 1) evaluating L-attributed AG at parse time, and 2) applying a common visit-oriented evaluator for the rest of AG.

This paper presents two algorithms to insert visit sequences of AGs into parsing actions. The first algorithm combines tree construction sequences (parsing actions) and attribute evaluation sequences (visit sequences of AGs) into a *parse-time visit sequence*. The resulting sequence can be directly used in top-down parsing. For bottom-up parsing, another algorithm introduces additional steps to schedule the visit sequence according to the free positions in the grammar.

The rest of this paper is organized as follows. Section 2 presents the definitions of AGs and visit-oriented attribute evaluators. Section 3 presents the algorithms to schedule the visit sequences of an AG in parsing actions. Section 4 gives an example. Section 5 concludes this paper.

2 Attribute Grammars and Visit-Oriented Attribute Evaluators

2.1 Attribute Grammars

An attribute grammar (AG) augments the context-free grammar $G = (N, T, P, S)$ by associating a set of *attributes* to each symbol of G and adding attribution rules to each production of G. The set of attributes for a symbol X is denoted $A(X)$. $A(X)$ consists of two disjoint finite sets: the *inherited* attributes $I(X)$ and the *synthesized* attributes $S(X)$, i.e., $I(X) \cap S(X) = \varnothing$ and $A(X) = I(X) \cup S(X)$. $R(p)$ is a set of dependency rules associated with production rule p in P. Let p: $X_0 \rightarrow X_1 \ldots X_{np}$. An *attribute occurrence* of p is a tuple (i, a), $a \in A(X_i)$, $0 \le i \le n_p$. A *dependency rule* is a dependency on two attribute occurrences:

$(i_1, a_1) \leftarrow (i_2, a_2)$;

where $(i_1, a_1) \in DO(p)$ (*defined occurrences*) and $(i_2, a_2) \in UO(p)$ (*used occurrences*) [Alb91a],

$DO(p) = \{(0, a) \mid a \in S(X_0)\} \cup \{(i, a) \mid a \in I(X_i), 1 \le i \le n_p\}$,

$UO(p) = \{(i, a) \mid a \in A(X_i), 0 \le i \le n_p\}$. The notation '$\leftarrow$' means "*depends on*".

Figure 1 shows a context-free grammar, which represents the assignment statement part of a

S → id = E	**if** id.type = *int* **then** E.exp_type = *int* **else** E.exp_type = *unspecified* **endif**
$E_0 \rightarrow E_1 + T$	E_1.exp_type = E_0.exp_type; T.exp_type = E_0.exp_type; **if** E_1.act_type = *real* **then** E_0.act_type = *real* **else** E_0.act_type = T.act_type **endif**
E → T	T.exp_type = E.exp_type; E.act_type = T.act_type
$T_0 \rightarrow P$ op T_1	**if** (T_0.exp_type = *int*) **or** (op.oper = *) **then** P.exp_type = *int*; T_1.exp_type = *int* **else** P.exp_type = *unspecified*; T_1.exp_type = *unspecified* **endif** **case** op.oper **of** * : **if** P.act_type = *real* **then** T_0.act_type = real **else** T_0.act_type = T_1.act_type / : **if** T_0.exp_type = *int* **then** *ERROR()*; T_0.act_type = *int* **else** T_0.act_type = *real* **endif**
T → P	P.exp_type = T.exp_type; T.act_type = P.act_type
P → (E)	E.exp_type = P.exp_type; P.act_type = E.act_type
P → id	**if** (P.exp_type = *int*) **and** (id.type = *real*) **then** *ERROR()*; P.act_type = *int* **else** P.act_type = id.type **endif**
op → *	op.oper = *
op → /	op.oper = /

Figure 1. An AG for expressions.

programming language, together with the attribution rules handling type checking. This example is a modification of the AG in [Alb91a, Example 3.2]. The attribute `exp_type`, which stores the expected type, is an inherited attribute. The attribute `act_type`, which stores the actual type, is a synthesized attribute. Terminal symbols `id` and `op` contain synthesized attributes, `type` and `oper`, respectively. When the expected type does not match the actual type, an error message is prompted by calling *ERROR()*.

Figure 2 shows the parse tree and the evaluation flow for an example piece of code: `I1 = I2 * I3`, where all `id`s are *int*. Associated with each node (grammar symbol) in the tree, the lefthand side is an inherited attribute and the righthand side is a synthesized attribute. The values of inherited attributes propagate top down; the values of synthesized attributes propagate bottom up.

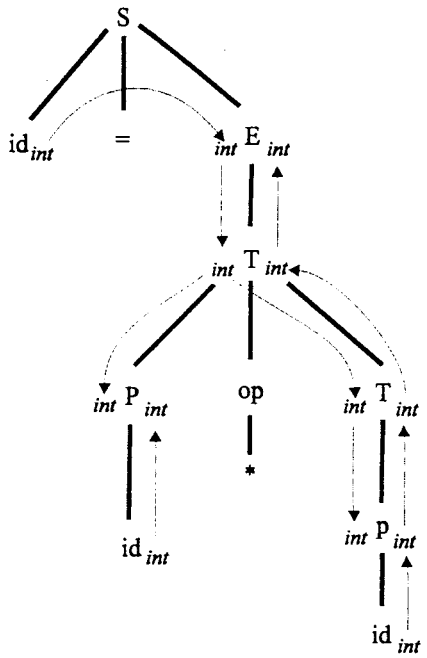

Figure 2. A parse tree decorated with attributes.

2.2 Visit-Oriented Attribute Evaluators

In a production p, the value of the attribute occurrence (i, a) is *available* if the values of all attribute occurrences (j, b) that (i, a) depends on are available. An attribute evaluator analyzes the attribute dependencies to choose a tree-walk strategy. Here we concentrate on the *visit-oriented* strategy, which evaluates the class of linear-ordered AGs [Alb91b]. A detailed discussion on visit-oriented attribute evaluators can be found in [Kas91].

A visit-oriented attribute evaluator has an evaluation pattern called *visit sequences* [Kas80] for each production. A visit sequence vs_p of production p is a sequence $v_1, ..., v_m$, where each v_k, $1 \le k \le m$, is one of the following operations (*comp*, *visit*, *leave*):

1. $v_k = comp_k$	define attributes of the context by using some other attributes.
2. $v_k = visit(i, j)$	visit the i-th child ($i \ge 1$) for the j-th time ($j \ge 1$).
3. $v_k = leave(j)$	return to the parent's context for the j-th time ($j \ge 1$).

According to attribute dependencies, the attributes of a symbol X_0 can be partitioned and ordered as a number of attribute subsets: $<I_1(X), S_1(X), ..., I_m(X), S_m(X)>$. In a production p: $X_0 \rightarrow X_1 ... X_n$, the visit sequence vs_p of p are constructed based on the attribute partitions of symbols X_0, X_1, ..., X_n. The control moves between adjacent contexts by *visit* and *leave* operations, after the attribute evaluator starts to work. When the control visits a child, some of the child's inherited attributes are computed. When the control returns to the parent, some of the parent's synthesized attributes are computed. Figure 3 shows the interaction between two visit sequences. These visit sequences fit together for the parse tree, and a complete tree walk is performed.

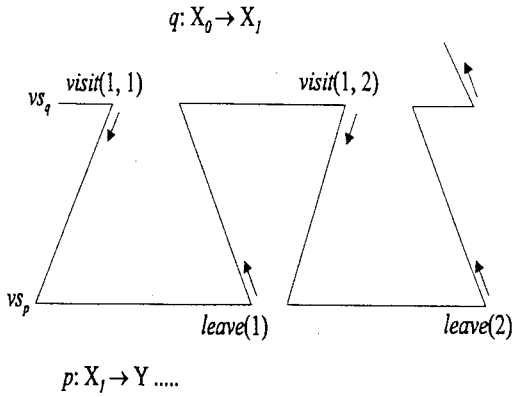

Figure 3. Interaction between two visit sequences

3 Parse-Time Visit Sequences

This section discusses parse-time attribute evaluation in both top-down (LL grammars) and bottom-up (LR grammars) cases. The difference between these two is that not all positions in LR grammars are free to add attribute evaluation codes. Cares should be taken to deal with the positions that are not free.

3.1 Combine Tree Construction Sequence with Visit sequences

The visit sequences described in Section 2 assume that a tree walk for attribute evaluation is performed after the parse tree is completely built. If the evaluation works while the tree is being built, the number of visits may be reduced and the

$E_0 \rightarrow$		E_1	+	T	
	$comp_1$: E_1.exp_type = E_0.exp_type, T.exp_type = E_0.exp_type,	*construct*(1), *visit*(1, 1),		*construct*(2), *visit*(2, 1),	$comp_4$: **if** E_1.act_type = *real* **then** E_0.act_type = *real* **else** E_0.act_type = T.act_type

Figure 4. A production and its parse-time visit sequence

(evaluated) tree nodes can be freed immediately. Thus, parse-time evaluation takes less time and space than than after-parse evaluation.

Consider the production p: $X_0 \rightarrow X_1 \ldots X_n$. The tree construction sequence for production p is:

construct(*1*), ..., *construct*(*n*),	where *construct*(*i*) denotes constructing the subtree rooted with X_i.

Embedding the visit sequence into the tree construction sequence allows evaluating attributes during parsing. For a visit sequence of production p, only the first visit can be embedded into p's tree construction sequence, because LL and LR parsing techniques are one pass. p's tree construction sequence and first visit of visit sequence is thus called p's *parse-time visit sequence*. The parse-time visit sequence can be further divided into n+1 *sections*, separated by *construct*(*i*) operations, and labeled as 0 to n.

The following are the constraints for inserting a visit sequence into a tree construction sequence:

1. The order of operations of the visit sequence cannot be altered.
2. A *visit*(*i*, 1) cannot precede a *construct*(*i*).

Algorithm 1 inserts the visit sequence into the tree construction sequence.

Algorithm 1. *Embed tree construction sequence and visit sequence.*
Input. A tree construction sequence and *VS*, the first partition of visit sequence of production p: $X_0 \rightarrow X_1 \ldots X_n$.
Output. A parse-time visit sequence *VS'*.
Begin
Let *VS* = [vs_0, *visit*(i_1, 1), vs_1, *visit*(i_2, 1), vs_2,..., *visit*(i_m, 1), vs_m, *leave*(1)], where
$1 \le i_1 < i_2 < \ldots < i_m \le n$.
VS' := [vs_0, *construct*(1..i_1), *visit*(i_1, 1), vs_1, *construct*(i_1+1..i_2), *visit*(i_2, 1), vs_2, ...,
construct(i_{m-1}+1..i_m), *visit*(i_m, 1), vs_m, *leave*(1)];
return *VS'*;
End of Algorithm

The vs_j, $0 \le j \le m$, are sub-sequences of *VS* that are separated by *visit*(i_1, 1), ..., *visit*(i_m, 1). These sub-sequences may contain other visits that do not follow the monotonic increasing order: $1 \le i_1 < i_2 < \ldots < i_m \le n$. Since Algorithm 1 retains the order of visit sequence, the first constraint is preserved. The *visit*(i_k, 1) operations are added after *construct*(i_k), that is, the second constraint is preserved, too. The notation *construct*(*i* .. *j*) represents the sequence [*construct*(*i*), *construct*(*i*+1), ..., *construct*(*j*)].

The embedded visit sequence of production $E_0 \rightarrow E_1$ + T in Figure 1 is shown in Figure 4. The computation $comp_1$ defines the inherited attributes (`exp_type`) of E_1 and T; $comp_4$ defines the synthesized attributes (`act_type`) of E_0. The sequence [*construct*(1), *visit*(1, 1)] is mapped into E_1; the sequence [*construct*(2), *visit*(2, 1)] is mapped into T.

3.2 Schedule Parse-Time Visit sequences in Free Positions

A sequence of *comp* and *visit* operations can be inserted between *construct*(*j*) and *construct*(*j*+1), only if the *j*+1-th position of production p is free. Algorithm 1 assumes that every positions in the grammar are free; however, this is not true for LR grammars. One way to solve this is to move the visit sequence that is inserted into a non-free (*forbidden*) position to the next free position, as shown in Algorithm 2.1.

Algorithm 2.1 *Adjust visit sequence for free positions.*
Input. Output of Algorithm 1 and the position information of p.
Output. The visit sequence vs_p that contains no operations in forbidden positions.
Begin
Let p: $X_0 \rightarrow X_1 \ldots X_n$.
for i := 0 to n **do**
if position i is forbidden and section i of vs_p is not empty
then move the operations in section i to section i+1;
End of Algorithm

With Algorithm 2.1, each production is associated with a corresponding parse-time visit sequence, where all operations are scheduled in free positions. One problem left is that the used inherited attributes of a symbol may be unavailable when parsing the symbol. This is because the computation of these inherited

attributes is delayed to the next free position by Algorithm 2.1, and this delay may make the computation unable to be invoked during parsing.

A production p: $X_0 \rightarrow X_1 \ldots X_n$ is *parse-time evaluable* if (1) the attribute occurrences in the parse-time visit sequence of p do not depend on X_0's inherited attributes; or (2) in any production q where X_0 is the j-th righthand side symbol of q, q is parse-time evaluable, and the used inherited attributes of X_0 are available before *construct*(j). Algorithm 2.2 tests whether a production is parse-time evaluable.

Algorithm 2.2. *Output parse-time evaluable productions.*
Input. A grammar G and parse-time visit sequence for each production.
Output. A set of productions that are parse-time evaluable.
Begin
P := the set of productions of G;
$Q := \varnothing$; { initialization for parse-time evaluable productions }
for each p: $X_0 \rightarrow X_1 \ldots X_n$ in P **do**
if the inherited attributes of X_0 are not used in the parse-time visit sequence
then move p from P to Q;
repeat
for each p: $X_0 \rightarrow X_1 \ldots X_n$ in P **do**
avail := TRUE;
for every occurrence of X_0 in q: $Y \rightarrow \ldots X_0 \ldots$ **do**
if the used inherited attributes of X_0 in the parse-time visit sequence
of p are not available before *construct*(X_0) in vs_q
then *avail* := FALSE;
if *avail* **then** move p from P to Q;
until no production can move from P to Q;
for each q: $X_0 \rightarrow X_1 \ldots X_n$ in Q **do**
if there is a p: $X_0 \rightarrow Y_1 \ldots Y_n$ in P **then** move q from Q to P;
End of Algorithm

S → id = E	*visit*(1, 1); **if** id.type = *int* **then** E.exp_type = *int* **else** E.exp_type = *unspecified* **endif** *visit*(2, 1); *visit*(3, 1); *leave*(1);
$E_0 \rightarrow$ E_1 + T	E_1.exp_type = E_0.exp_type; T.exp_type = E_0.exp_type; *visit*(1, 1); *visit*(2, 1); *visit*(3, 1); **if** E_1.act_type = *real* **then** E_0.act_type = *real* **else** E_0.act_type = T.act_type **endif** *leave*(1);
E → T	T.exp_type = E.exp_type; *visit*(1, 1); E.act_type = T.act_type; *leave*(1);
$T_0 \rightarrow$ P op T_1	*visit*(2, 1); **if** (T_0.exp_type = *int*) **or** (op.oper = *) **then** P.exp_type = *int*; T_1.exp_type = *int* **else** P.exp_type = *unspecified*; T_1.exp_type = *unspecified* **endif** *visit*(1, 1); *visit*(3, 1); **case** op.oper **of** * : **if** P.act_type = *real* **then** T_0.act_type = real **else** T_0.act_type = T_1.act_type **endif** / : **if** T_0.exp_type = *int* **then** *ERROR()*; T_0.act_type = *int* **else** T_0.act_type = *real* **endif** *leave*(1);
T → P	P.exp_type = T.exp_type; *visit*(1, 1); T.act_type = P.act_type; *leave*(1);
P → (E)	E.exp_type = P.exp_type; *visit*(1, 1); *visit*(2, 1); *visit*(3, 1); P.act_type = E.act_type; *leave*(1);
P → id	*visit*(1, 1); **if** (P.exp_type = *int*) **and** (id.type = *real*) **then** *ERROR()*; P.act_type = *int*; **else** P.act_type = id.type **endif** *leave*(1);
op → *	op.oper = *; *leave*(1);
op → /	op.oper = /; *leave*(1);

Figure 5. The visit sequences of the AG in Figure 1.

4. An Example

This section presents an example to show how the algorithms in Section 3 work. Figure 5 shows the visit sequences of the AG in Figure 1.

S → id = E	*construct*(1); *visit*(1, 1); **if** id.type = *int* **then** E.exp_type = *int* **else** E.exp_type = *unspecified* **endif** *construct*(2); *visit*(2, 1); *construct*(3); *visit*(3, 1); *leave*(1);
E_0 → E_1 + T	E_1.exp_type = E_0.exp_type; T.exp_type = E_0.exp_type; *construct*(1); *visit*(1, 1); *construct*(2); *visit*(2, 1); *construct*(3); *visit*(3, 1); **if** E_1.act_type = *real* **then** E_0.act_type = *real* **else** E_0.act_type = T.act_type **endif** *leave*(1);
E → T	T.exp_type = E.exp_type; *construct*(1); *visit*(1, 1); E.act_type = T.act_type; *leave*(1);
T_0 → P op T_1	*construct*(1); *construct*(2); *visit*(2, 1); **if** (T_0.exp_type = *int*) **or** (op.oper = *) **then** P.exp_type = *int*; T_1.exp_type = *int* **else** P.exp_type = *unspecified*; T_1.exp_type = *unspecified* **endif** *visit*(1, 1); *construct*(3); *visit*(3, 1); **case** op.oper **of** * : **if** P.act_type = *real* **then** T_0.act_type = real **else** T_0.act_type = T_1.act_type **endif** / : **if** T_0.exp_type = *int* **then** *ERROR()*; T_0.act_type = *int* **else** T_0.act_type = *real* **endif** *leave*(1);
T → P	P.exp_type = T.exp_type; *construct*(1); *visit*(1, 1); T.act_type = P.act_type; *leave*(1);
P → (E)	E.exp_type = P.exp_type; *construct*(1); *visit*(1, 1); *construct*(2); *visit*(2, 1); *construct*(3); *visit*(3, 1); P.act_type = E.act_type; *leave*(1);
P → id	*construct*(1); *visit*(1, 1); **if** (P.exp_type = *int*) **and** (id.type = *real*) **then** *ERROR()*; P.act_type = *int* **else** P.act_type = id.type **endif** *leave*(1);
op →*	op.oper = *; *leave*(1);
op → /	op.oper = /; *leave*(1);

Figure 6. The resulting parse-time visit sequences of Algorithm 1.

Figure 6 presents the resulting parse-time visit sequences of Algorithm 1. Figure 7 shows the free positions in the production rules. The free positions are marked by { }. Figure 8 shows the adjusted parse-time visit sequences by Algorithm 2.1. Figure 9 classifies the production rules according to whether they are PTE or not.

S → { } id { } = { } E { }
E → E { } + { } T { }
E → { } T { }
T → P { } op { } T { }
T → P { }
P → { } ({ } E { }) { }
P → { } id { }
op → { } * { }
op → { } / { }

Figure 7. The free positions of the expression grammar.

5 Conclusions and Future Work

In this paper, we have presented algorithms to generate parse-time visit sequences both for top-down and bottom-up parsing. With this technique, the generated compiler from attribute grammars can be more efficient in time and space. We plan to implement this technique in the system *ag++*, which is an object-oriented compiler generator based on attribute grammars and reusable components.

References

[Alb91a] Alblas, H., Introduction to attribute grammars. *LNCS 545*, Springer-Verlag, pp. 1-15 (1991)

[Alb91b] Alblas, H., Attribute evaluation methods. *LNCS 545*, Springer-Verlag, pp. 48-111 (1991)

[Kas80] Kastens, U.: Ordered Attributed Grammars, *Acta Informatica* 13, pp. 229-256 (1980)

[Kas91] Kastens, U.: Implementation of Visit-Oriented Attribute Evaluators. *LNCS 545*, Springer-Verlag, pp. 114-137 (1991)

[LRS74] Lewis, P.M., Rosenkrantz, D.J. and Stearns, R.E.: Attributed translations. *Journal of Computer and System Science* 9, pp. 279-307 (1974).

[PuB80] Purdom, P. and Brown, C.A.: Semantic Routines and LR(k) Parsers. *Acta Informatica* 14, pp. 299-315 (1980).

[Wat77] Watt, D.A.: The parsing problem for affix grammars. *Acta Informatica* 8, pp. 1-20 (1977).

attributes is delayed to the next free position by Algorithm 2.1, and this delay may make the computation unable to be invoked during parsing.

A production p: $X_0 \rightarrow X_1 \ldots X_n$ is *parse-time evaluable* if (1) the attribute occurrences in the parse-time visit sequence of p do not depend on X_0's inherited attributes; or (2) in any production q where X_0 is the j-th righthand side symbol of q, q is parse-time evaluable, and the used inherited attributes of X_0 are available before *construct(j)*. Algorithm 2.2 tests whether a production is parse-time evaluable.

Algorithm 2.2. *Output parse-time evaluable productions.*
Input. A grammar G and parse-time visit sequence for each production.
Output. A set of productions that are parse-time evaluable.
Begin
P := the set of productions of G;
$Q := \varnothing$; { initialization for parse-time evaluable productions }
for each p: $X_0 \rightarrow X_1 \ldots X_n$ in P **do**
if the inherited attributes of X_0 are not used in the parse-time visit sequence
then move p from P to Q;
repeat
for each p: $X_0 \rightarrow X_1 \ldots X_n$ in P **do**
avail := TRUE;
for every occurrence of X_0 in q: $Y \rightarrow \ldots X_0 \ldots$ **do**
if the used inherited attributes of X_0 in the parse-time visit sequence
of p are not available before *construct*(X_0) in vs_q
then *avail* := FALSE;
if *avail* **then** move p from P to Q;
until no production can move from P to Q;
for each q: $X_0 \rightarrow X_1 \ldots X_n$ in Q **do**
if there is a p: $X_0 \rightarrow Y_1 \ldots Y_n$ in P **then** move q from Q to P;
End of Algorithm

S → id = E	*visit*(1, 1); **if** id.type = *int* **then** E.exp_type = *int* **else** E.exp_type = *unspecified* **endif** *visit*(2, 1); *visit*(3, 1); *leave*(1);
E_0 → E_1 + T	E_1.exp_type = E_0.exp_type; T.exp_type = E_0.exp_type; *visit*(1, 1); *visit*(2, 1); *visit*(3, 1); **if** E_1.act_type = *real* **then** E_0.act_type = *real* **else** E_0.act_type = T.act_type **endif** *leave*(1);
E → T	T.exp_type = E.exp_type; *visit*(1, 1); E.act_type = T.act_type; *leave*(1);
T_0 → P op T_1	*visit*(2, 1); **if** (T_0.exp_type = *int*) **or** (op.oper = *) **then** P.exp_type = *int*; T_1.exp_type = *int* **else** P.exp_type = *unspecified*; T_1.exp_type = *unspecified* **endif** *visit*(1, 1); *visit*(3, 1); **case** op.oper **of** * : **if** P.act_type = *real* **then** T_0.act_type = real **else** T_0.act_type = T_1.act_type **endif** / : **if** T_0.exp_type = *int* **then** *ERROR()*; T_0.act_type = *int* **else** T_0.act_type = *real* **endif** *leave*(1);
T → P	P.exp_type = T.exp_type; *visit*(1, 1); T.act_type = P.act_type; *leave*(1);
P → (E)	E.exp_type = P.exp_type; *visit*(1, 1); *visit*(2, 1); *visit*(3, 1); P.act_type = E.act_type; *leave*(1);
P → id	*visit*(1, 1); **if** (P.exp_type = *int*) **and** (id.type = *real*) **then** *ERROR()*; P.act_type = *int*; **else** P.act_type = id.type **endif** *leave*(1);
op → *	op.oper = *; *leave*(1);
op → /	op.oper = /; *leave*(1);

Figure 5. The visit sequences of the AG in Figure 1.

4. An Example

This section presents an example to show how the algorithms in Section 3 work. Figure 5 shows the visit sequences of the AG in Figure 1.

S → id = E	*construct*(1); *visit*(1, 1); **if** id.type = *int* **then** E.exp_type = *int* **else** E.exp_type = *unspecified* **endif** *construct*(2); *visit*(2, 1); *construct*(3); *visit*(3, 1); *leave*(1);
E_0 → E_1 + T	E_1.exp_type = E_0.exp_type; T.exp_type = E_0.exp_type; *construct*(1); *visit*(1, 1); *construct*(2); *visit*(2, 1); *construct*(3); *visit*(3, 1); **if** E_1.act_type = *real* **then** E_0.act_type = *real* **else** E_0.act_type = T.act_type **endif** *leave*(1);
E → T	T.exp_type = E.exp_type; *construct*(1); *visit*(1, 1); E.act_type = T.act_type; *leave*(1);
T_0 → P op T_1	*construct*(1); *construct*(2); *visit*(2, 1); **if** (T_0.exp_type = *int*) **or** (op.oper = *) **then** P.exp_type = *int*; T_1.exp_type = *int* **else** P.exp_type = *unspecified*; T_1.exp_type = *unspecified* **endif** *visit*(1, 1); *construct*(3); *visit*(3, 1); **case** op.oper **of** * : **if** P.act_type = *real* **then** T_0.act_type = real **else** T_0.act_type = T_1.act_type **endif** / : **if** T_0.exp_type = *int* **then** *ERROR()*; T_0.act_type = *int* **else** T_0.act_type = *real* **endif** *leave*(1);
T → P	P.exp_type = T.exp_type; *construct*(1); *visit*(1, 1); T.act_type = P.act_type; *leave*(1);
P → (E)	E.exp_type = P.exp_type; *construct*(1); *visit*(1, 1); *construct*(2); *visit*(2, 1); *construct*(3); *visit*(3, 1); P.act_type = E.act_type; *leave*(1);
P → id	*construct*(1); *visit*(1, 1); **if** (P.exp_type = *int*) **and** (id.type = *real*) **then** *ERROR()*; P.act_type = *int* **else** P.act_type = id.type **endif** *leave*(1);
op →*	op.oper = *; *leave*(1);
op → /	op.oper = /; *leave*(1);

Figure 6. The resulting parse-time visit sequences of Algorithm 1.

Figure 6 presents the resulting parse-time visit sequences of Algorithm 1. Figure 7 shows the free positions in the production rules. The free positions are marked by { }. Figure 8 shows the adjusted parse-time visit sequences by Algorithm 2.1. Figure 9 classifies the production rules according to whether they are PTE or not.

S → { } id { } = { } E { }
E → E { } + { } T { }
E → { } T { }
T → P { } op { } T { }
T → P { }
P → { } ({ } E { }) { }
P → { } id { }
op → { } * { }
op → { } / { }

Figure 7. The free positions of the expression grammar.

5 Conclusions and Future Work

In this paper, we have presented algorithms to generate parse-time visit sequences both for top-down and bottom-up parsing. With this technique, the generated compiler from attribute grammars can be more efficient in time and space. We plan to implement this technique in the system *ag++*, which is an object-oriented compiler generator based on attribute grammars and reusable components.

References

[Alb91a] Alblas, H., Introduction to attribute grammars. *LNCS 545*, Springer-Verlag, pp. 1-15 (1991)

[Alb91b] Alblas, H., Attribute evaluation methods. *LNCS 545*, Springer-Verlag, pp. 48-111 (1991)

[Kas80] Kastens, U.: Ordered Attributed Grammars, *Acta Informatica* 13, pp. 229-256 (1980)

[Kas91] Kastens, U.: Implementation of Visit-Oriented Attribute Evaluators. *LNCS 545*, Springer-Verlag, pp. 114-137 (1991)

[LRS74] Lewis, P.M., Rosenkrantz, D.J. and Stearns, R.E.: Attributed translations. *Journal of Computer and System Science* 9, pp. 279-307 (1974).

[PuB80] Purdom, P. and Brown, C.A.: Semantic Routines and LR(k) Parsers. *Acta Informatica* 14, pp. 299-315 (1980).

[Wat77] Watt, D.A.: The parsing problem for affix grammars. *Acta Informatica* 8, pp. 1-20 (1977).

S → id = E	*construct*(1); *visit*(1, 1); **if** id.type = *int* **then** E.exp_type = *int* **else** E.exp_type = *unspecified* **endif** *construct*(2); *visit*(2, 1); *construct*(3); *visit*(3, 1); *leave*(1);
$E_0 \rightarrow E_1 + T$	*construct*(1); E_1.exp_type = E_0.exp_type; T.exp_type = E_0.exp_type; *visit*(1, 1); *construct*(2); *visit*(2, 1); *construct*(3); *visit*(3, 1); **if** E_1.act_type = *real* **then** E_0.act_type = *real* **else** E_0.act_type = T.act_type **endif** *leave*(1);
E → T	T.exp_type = E.exp_type; *construct*(1); *visit*(1, 1); E.act_type = T.act_type; *leave*(1);
$T_0 \rightarrow P$ op T_1	*construct*(1); *construct*(2); *visit*(2, 1); **if** (T_0.exp_type = *int*) **or** (op.oper = *) **then** P.exp_type = *int*; T_1.exp_type = *int* **else** P.exp_type = *unspecified*; T_1.exp_type = *unspecified* **endif** *visit*(1, 1); *construct*(3); *visit*(3, 1); **case** op.oper **of** * : **if** P.act_type = *real* **then** T_0.act_type = real **else** T_0.act_type = T_1.act_type **endif** / : **if** T_0.exp_type = *int* **then** *ERROR()*; T_0.act_type = *int* **else** T_0.act_type = *real* **endif** *leave*(1);
T → P	*construct*(1); P.exp_type = T.exp_type; *visit*(1, 1); T.act_type = P.act_type; *leave*(1);
P → (E)	E.exp_type = P.exp_type; *construct*(1); *visit*(1, 1); *construct*(2); *visit*(2, 1); *construct*(3); *visit*(3, 1); P.act_type = E.act_type; *leave*(1);
P → id	*construct*(1); *visit*(1, 1); **if** (P.exp_type = *int*) **and** (id.type = *real*) **then** *ERROR()*; P.act_type = *int* **else** P.act_type = id.type **endif** *leave*(1);
op → *	op.oper = *; *leave*(1);
op → /	op.oper = /; *leave*(1);

Figure 8. The adjusted parse-time visit sequences of Algorithm 2.1.

PTE:

S → id = E
T → P op T
T → P
op → *
op → /

non-PTE:

E → E + T
E → T
P → (E)
P → id

Figure 9. Production rules classified by PTE and non-PTE.