BIT 23 (1983), 308–320

ON INCREMENTAL EVALUATION OF ORDERED ATTRIBUTED GRAMMARS

DASHING YEH

The Division of Computer Science, The Norwegian Institute of Technology, N-7034 Trondheim, Norway

Abstract.

A method is presented to augment a conventional evaluator for an ordered attributed grammar into an incremental one, though it is applicable to tree-walker evaluators for any non-circular attributed grammars. Also three improvements are suggested. The resulting incremental evaluator is statically deterministic and evaluates a modified semantic tree in time proportional to the amount of attribute instances affected by the modification.

Keywords and Phrases: attribute evaluators, attributed grammars, incremental attribute evaluation, ordered attributed grammars.

1. Introduction.

Recently, the problem of incremental attribute evaluation was raised and studied (see [3, 4, and 6]). There have been a few incremental attribute evaluators running or coming, which have in common that the evaluation order is determined at run time.

In this paper, we first present a method of augmenting a conventional evaluator for an ordered attributed grammar (OAG, for short) into an incremental one, then suggest three ways to improve it. The resulting incremental attribute evaluator is statically deterministic, viz. the evaluation order is determined at construction time, and evaluates a modified semantic tree in time proportional to the amount of attribute instances affected by the modification.

Although the method is described here only for OAG evaluators, it can be readily extended to tree-walker evaluators constructed by the algorithm given in [2] for any non-circular attributed grammars (AGs, for short).

The rest of the paper is organized as follows: Section 2 contains an introduction to OAGs and their evaluators; section 3 formulates the problem of incremental attribute evaluation in its general form; section 4 presents our incremental evaluator generated by augmenting a conventional OAG evaluator; section 5 proves the validity of the incremental evaluator; in section 6, three

Received November 4, 1982. Revised February 24, 1983.

ways are suggested to improve the evaluator; and finally, some remarks are given in section 7. At the end of the paper, there is an appendix.

2. Preliminaries.

The terms and notations not defined here can be found in [1, 5]. Below is a brief introduction to OAGs and their evaluators; see [5] and others for a complete description.

An AG is a context-free grammar extended by attaching to each symbol of the grammar a finite set of attributes. Associated with each production of the grammar is a finite set of semantic functions defining values of the attributes occurring in the production. There are two kinds of attributes, viz. inherited and synthesized. The start and terminal symbols are assumed to have only synthesized attributes.

Within a production p, for each occurrence of a symbol and each attribute of the symbol, there is correspondingly an attribute occurrence. For instance, let X be a symbol occurring in p, and a an attribute of X, then we use $X.a$ to denote the attribute occurrence. If X occurs more than once in p, then we use $X_1.a$ to refer to the attribute occurrence corresponding to the first occurrence of X, and $X_2.a$ to that to the second, and so on. For each synthesized attribute occurrence of the left-hand side symbol, there is exactly one semantic function defining its value, and arguments, if any, are either inherited from the left-hand side symbol, or synthesized from right-hand side symbols; likewise for each inherited occurrence of a right-hand side symbol. If $X.a$ is used in defining $Y.b$, then $Y.b$ is said to depend on $X.a$, denoted by $X.a \rightarrow Y.b$. $Y.b$ is also called a dependant of $X.a$, while $X.a$ is called a donator of $Y.b$. The set of all dependants of $X.a$ in p is denoted by DEPENDANTS$(X.a, p)$.

Let G be an AG, and w a sentence in $L(G)$. Suppose w has been parsed yielding a parse tree T. A semantic tree of w is generated by attaching all attributes to their associated symbols that label the nodes of T. Each attached attribute is said to have an attribute instance on the semantic tree. For simplicity, we will use the same notations to denote an attribute occurrence and its corresponding attribute instance as well as a parse tree and its corresponding semantic tree; there would be no confusion if we read notations in the contexts in which they occur.

From T we can construct such a directed graph that the vertices are all attribute instances on T and the edges are all pairs $(X.a, Y.b)$ where $X.a \rightarrow Y.b$. The graph is called the dependency graph of T. If there is a circuit in the graph, then T is said to be circular. If for every sentence in $L(G)$ there is a non-circular semantic tree, then G is said to be non-circular or well-defined.

The term "attribute evaluation" refers to a process of assigning values to all attribute instances on a semantic tree in accordance with their defining functions. Algorithms accomplishing attribute evaluation are called attribute evaluators, or simply evaluators. An attribute instance is ready for evaluation when and only when its defining function is a constant or its donators all have been evaluated. Therefore, for an evaluator to evaluate a semantic tree it is essential to find a proper evaluation order.

Informally, an OAG is such an AG that for each symbol X of the grammar a partial order among the attributes of X can be set up in such a way that the attribute instances that are attached to a node labelled with X on a semantic tree are always evaluated according to such an order, no matter where on the tree, nor on which semantic tree the node is situated.

For each OAG G, a tree-walker evaluator can be mechanically constructed by the algorithm given in [5]. Let us call it the OAG evaluator of G. With each evaluation process the OAG evaluator starts at the root, then walks around on the tree; finally, it returns to the root and terminates there. When it comes to a node, which is also called visiting the node, it first evaluates all attribute instances that occur in the production applied at the node and are ready for evaluation, then leaves for the father or a son of the node. A node may be visited several times and the number of visits may vary from node to node. When the OAG evaluator eventually terminates at the root, the semantic tree is completely evaluated and the evaluation process is finished.

Structurally an OAG evaluator is a table-driven algorithm, i.e., a driver runs under control of a table. The driver is the same for all OAG evaluators, while one table is constructed for each OAG in particular. In detail, let G be an OAG, then for each production p of G, a so-called visit sequence VS(p) is constructed. VS(p) consists of two kinds of items: $X.a$ and $v(k, i)$, where $X.a$ is an attribute occurrence appearing in p. Suppose p is applied at a node N on a semantic tree, in which case p is also called the production indicator of N. Then, $X.a$ in VS(p) means evaluating the attribute instance $X.a$, while $v(k, i)$ paying the kth visit to the father or to the ith son of N, depending on whether $i = 0$ or $i > 0$.

If there is a total of m items of the form $v(k, 0)$ in VS(p), where $k = 1, 2, \ldots, m$, then VS(p) is further divided into m segments, each ended with a $v(k, 0)$. We will use VS(p, k) to denote the kth segment of VS(p). For all productions of G, all segments are collected into one table. A function called MAPDOWN is devised which maps the visit number k of a son visit $v(k, i)$ and the production indicator p of the son to be visited into the index d of the first table entry of VS(p, k), i.e.,

$$d = \text{MAPDOWN}(k, p).$$

The action of the OAG evaluator of G is directed by the table above. At any moment in an evaluation process there is exactly one table entry which is called the current entry, directing the OAG driver. The visit sequence to which the current entry belongs is called the current one and associated with the production applied at the node being visited. If the current entry is an item of the form $X.a$, then the driver calls the semantic function defining $X.a$, and the next item in the current visit sequence will be current next; if the current entry is of the form $v(k, i)$, then the driver leaves for the father or the ith son, depending on whether $i = 0$ or $i > 0$, and the current visit sequence will be changed accordingly. When a node is revisited, its associated visit sequence resumes control from the item next to where it was left during the last visit. In order to keep such a control flow, two parameters must be saved. One is the reference to the node being visited, and the other is the index of the entry that will be current next. A stack is used to maintain such parameter pairs throughout an evaluation process. Figure 1 is the main loop of the OAG driver, written in a PASCAL-like language.

```
begin
  push(root, MAPDOWN(1, root.rod_indicator));
  repeat
      case  stack_top.table_entry of
       $X.a$ : call semantic function defining $X.a$;
             increment(stack_top.table_entry);
    $v(k, i)$ : /* $i > 0$ */
             increment(stack_top.table_entry);
             push(stack_top.node_ref, MAPDOWN($k$,
                   stack_top.node_ref.prod_indicator));
    $v(k, 0)$ : pop;
      esac
  until stack_is_empty;
end
```

Figure 1. The OAG driver.

Finally, if an evaluation process is coordinated on the time axis, then for each node A there is a unique moment, viz. that of first visiting A, denoted by MOMENT$(1, A)$. The stack configuration at MOMENT$(1, A)$ is computable and the computation procedure will be described in the appendix.

3. Formulation of the problem.

Let G be an OAG. We assume that the underlying context-free grammar of G has an incremental parser such as, e.g., one described in [7].

Now let $w = xzy$ and $w' = xz'y$ be two sentences in $L(G)$. Suppose w and w' have been parsed yielding the parse trees T and T', respectively. As said in [7], T and T' must have the structures shown in Figure 2, where $x = x_0x_1$, $y = y_1y_0$, and the shaded parts are the same in both T and T', viz. T' can be

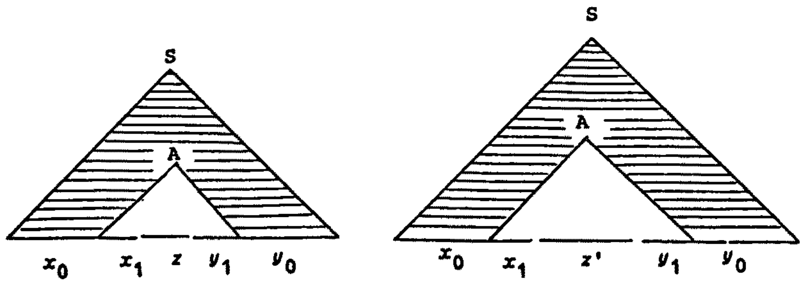

Figure 2. Incremental parsing.

obtained from T by replacement of the subtree rooted at the node labelled with a symbol A. Note that x_0, x_1, y_0, and y_1 all may be the empty string, and particularly, if both x_0 and y_0 are empty, then the node A becomes the root S in both T and T'.

Attaching all attributes to their associated symbols on T and T' we obtain the semantic trees of w and w'. For simplicity we still use T and T' to denote these two semantic trees and refer to nodes by their labeling symbols. Besides, the set of all attribute instances on T' is denoted by ATTR.

Suppose that (1) C is a node which appears on T as well as on T', and (2) the production applied at C on T is the same as on T', if C is not a leaf. Then, each attribute instance of C is said to be retained after the subtree replacement above was made. By this definition we divide ATTR into two subsets, one consisting of all those retained and the other of the rest. We denote them by RETAIN and NEWBORN, respectively. Clearly, the attribute instances in NEWBORN all belong to the subtree rooted at A and are brought into existence exclusively due to this subtree replacement. Furthermore, in evaluating T' only attribute instances in RETAIN may have the same values as in evaluating T. Let us denote the subset of all such attribute instances in RETAIN by EQUVAL, and the rest by NOTEQU. Needless to say, the subtree replacement above is also the sole source that causes the attribute instances in NOTEQU to have the different values in evaluating T and T'. Just for this reason we call the union of NEWBORN and NOTEQU the set of attribute instances affected by the subtree replacement, denoted by AFFECT. Summing up, we have the following relations:

$$\begin{aligned} \text{ATTR} &= \text{RETAIN} \cup \text{NEWBORN}, \\ \text{RETAIN} &= \text{EQUVAL} \cup \text{NOTEQU}, \\ \text{AFFECT} &= \text{NEWBORN} \cup \text{NOTEQU}, \text{ and} \\ \text{ATTR} &= \text{EQUVAL} \cup \text{AFFECT}. \end{aligned}$$

Now suppose that T has been evaluated and consider evaluation of T'. Of course, T' can be evaluated by running the OAG evaluator of G over T' from beginning to end; however, to the same goal, an incremental approach appears to be more time-efficient because there are two potential advantages available: (1) T has been evaluated, and (2) the attribute instances in RETAIN can retain their values obtained in evaluating T and therefore need not be re-evaluated. In essence, for the sake of evaluation of T', it suffices to evaluate the attribute instances in AFFECT only, and the size of AFFECT is usually by far smaller than that of ATTR. However, owing to our ignorance of the membership in NOTEQU, evaluating more than the attribute instances in AFFECT is unavoidable. In fact, there is no way of knowing a priori the membership in NOTEQU, though we can learn that in RETAIN from parsing. Thus we have to resort to heuristic methods. Precisely, we will search for the members of NOTEQU within RETAIN in a trial-and-error manner, which inevitably results in evaluating someones in EQUVAL.

Considering the discussion above we formulate the problem of incremental attribute evaluation as follows: Given two semantic trees T and T' such that (1) T has been evaluated and (2) T' can be obtained from T by replacement of the subtree rooted at a node A, without evaluating the whole set ATTR, how should we accomplish evaluation of AFFECT while re-evaluating members in EQUVAL as few times as possible.

4. Description of the algorithm.

The incremental evaluator generated by the algorithm below is a conventional OAG evaluator augmented by a marking procedure.

ALGORITHM. The incremental evaluator.

INPUT. Two semantic trees T and T' such as described in (1) and (2) at the end of § 3.

OUTPUT. The evaluated T'.

METHOD. The conventional OAG driver is augmented as follows:

(i) Assign to each attribute instance in RETAIN its value obtained from evaluation of T, and to each in NEWBORN a special value Ø, meaning undefined;

(ii) by using the procedure described in the appendix, compute the stack configuration at MOMENT$(1, A)$ in a conventional evaluation process of T', then set the stack to the resulting configuration;

(iii) mark every attribute instance in NEWBORN;

(iv) re-code the first case of the case statement in Figure 1 as shown in Figure 3.

```
begin
    .../* initialisation as described in (i), */
  ... /* (ii), and (iii) above.              */
  repeat
      case  stack_top.table_entry of
       X.a : increment(stack_top.table_entry);
             if X.a_is_marked then
             begin
                call the semantic function defining X.a;
                if not result_equal_old_value then
                   mark all members of DEPENDANTS(X.a,
                   stack_top.node_ref.prod_indicator);
             end;
    v(k, i) : /* i > 0. */
             increment(stack_top.table_entry);
             push(stack_top.node_ref, MAPDOWN(k,
                  stack_top.node_ref.pro_indicator));
    v(k,0) : pop;
      esac
  until stack_is_empty;
end
```

Figure 3. The incremental OAG driver.

In implementing the algorithm above, the common parts of T and T' can be combined into one structure, while keeping their different subtrees separated. This reduces storage use as well as assignment operations a great deal. Moreover, for an attribute instance, the information on its marking can be stored within its corresponding table entry, and also the first case of the case statement in Figure 3 is accordingly split into the two cases, one dealing with marked attribute instances and the other with unmarked ones. Thus, mark-checking can be saved.

5. Correctness of the algorithm.

LEMMA 1. *Every attribute instance in NOTEQU must directly or indirectly depend on some members of NEWBORN.*

PROOF. Let $X.a$ be an attribute instance in NOTEQU. The function defining $X.a$ must not be a constant. If there is a member of NOTEQU among the donators of $X.a$, then the claim is proved, otherwise there must be a donator of $X.a$, say, $Y.b$ which is a member of NOTEQU. Apply the same reasoning with $Y.b$, and so on. As G is non-circular, such a deductive process must end with an attribute instance, say, $Z.c$ such that a donator of $Z.c$ is a member of NEWBORN, because otherwise $Z.c$ would be in EQUVAL, which contradicts our selection of $Z.c$. ■

LEMMA 2. *Let BEFORE(1, A) denote the set of all attribute instances evaluated before MOMENTA(1, A) in a conventional evaluation process of Tβ. Then, BEFORE(1, A) ⊂ EQUVAL.*

PROOF. By definition, BEFORE(1, A) must be contained in RETAIN, because the attribute instances in NEWBORN all belong to the subtree rooted at A. Furthermore, recall that RETAIN = EQUVAL ∪ NOTEQU, and by lemma 1 it must hold that BEFORE(1, A) ⊂ EQUVAL, because the attribute instances in BEFORE(1, A) by no means depend on anyone in NEWBORN. ■

Lemma 2 justifies the skip of evaluating the members of BEFORE(1, A) which is carried out by step (ii) of the algorithm.

LEMMA 3. *Let MARK denote the set of the marked attribute instances throughout an incremental evaluation process of T′. Then, AFFECT ⊂ MARK.*

PROOF. First, by step (iii), we have NEWBORN ⊂ MARK.

Second, as seen from the first case of the case statement in Figure 3, an attribute instance will be marked if and only if it depends on a marked one which is not a member of EQUVAL. By lemma 1, we can conclude that a member of NOTEQU will be marked sooner or later, which completes the proof because AFFECT = NEWBORN ∪ NOTEQU. ■

Lemma 3 implies that the attribute instances in AFFECT will be all evaluated when the incremental evaluation process terminates.

LEMMA 4. *For each attribute instance in ATTR, the value assigned by the incremental evaluator accords with its defining function.*

PROOF. By induction on the evaluation order number of an attribute instance.

Basis. Suppose $X.a$ is the first evaluated attribute instance. As an attribute instance is ready for evaluation when and only when its defining function is a constant or its donators all have been evaluated, $X.a$ must depend on no others in ATTR. Therefore, no matter if $X.a$ is in RETAIN or in NEWBORN, the value assigned by the incremental evaluator to $X.a$ always accords with its defining function.

Induction. Assume that the claim holds for all attribute instances with the evaluation order numbers $< n$. Let $X.a$ be the nth evaluated attribute instance. There are two cases: $X.a$ is either marked or unmarked. If $X.a$ is unmarked, then, by lemma 3, $X.a$ must be in EQUVAL. Noticing that the incremental evaluator never re-evaluates an unmarked attribute instance, we get the claim proved for the unmarked $X.a$. Now, if $X.a$ is marked, then $X.a$ must have been evaluated by the incremental evaluator, as indicated in the first case of the case statement in Figure 3. There are also two subcases: viz. the defining function of $X.a$ is either a constant or not. In the former subcase, the claim to be shown holds spontaneously; in the latter, by the induction hypothesis, the values assigned to all donators of $X.a$ accord with their defining functions, and consequently the value assigned to $X.a$ accords with its defining function too. Thus, we complete the induction as well as the proof. ■

Combining lemmas 2–4, we obtain the following:

THEOREM 1. *The incremental evaluator generated by the algorithm in §4 evaluates T' correctly.*

The following theorem 2 is essential for later computing the time complexity of the incremental evaluator.

THEOREM 2. *The size of MARK is proportional to that of AFFECT.*

PROOF. By step (iii), MARK is NEWBORN at the start of incremental evaluation of T'. Afterwards, as seen from the first case of the case statement in Figure 3, MARK is enlarged only when a member of NOTEQU is found and each enlargement is always bound by the maximum member of dependants of a symbol in G. Noticing that AFFECT = NEWBORN $\cup$ NOTEQU, the theorem follows immediately. ■

6. Improvements.

In this section, three important improvements are in turn suggested to terminate an incremental evaluation process of T' sooner than by running the incremental evaluator described above.

6.1 *Termination conditions.*

Generally, an incremental evaluation process of T' eventually terminates when the incremental evaluator finally returns to the root of T'. However, a close study indicates that it may terminate sooner than such a moment. In fact, by the definition of AGs, each semantic function is associated with one production and only applied to the attribute instances occurring in the associated production. Consequently, the only way in which a replacement of the subtree rooted at A may propagate its influence is to expand successively upwards one subtree by another. Now suppose A_i is a node such that (a) A_i is an ancestor of A, (b) the incremental evaluator is going to execute the last item of the visit sequence associated with A_i, and (c) all attribute instances of A_i have been found to be in EQUVAL. Then, by the definition of A_i, we can see that the attribute instances that are evaluated thereafter must all be in EQUVAL because (1) they must all fall outside the subtree rooted at A_i, and (2) the replacement of the subtree rooted at A has no more effect outside the subtree rooted at A_i. Therefore, the incremental evaluator need not run any longer. Moreover, as far as semantics is concerned, only through its synthesized attribute instances does A_i interface with other parts of T' outside the subtree rooted at A_i, and therefore condition (c) above can be relaxed as (c′) all synthesized attribute instances of A_i have been found to be in EQUVAL. Thus, the incremental evaluator can terminate when we are going to execute the last item of the visit sequence associated with an A_i satisfying the following two conditions:

(1) A_i is A or an ancestor of A;

(2) all synthesized attribute instances of A_i have been found to be in EQUVAL.

We call (1) and (2) above the termination conditions. They should be tested with A and each ancestor of A. In order to detect such testing moments, a special symbol \$ is inserted as the second last item in each visit sequence, and a new case dealing with \$, as shown below, is accordingly added to the case statement in Figure 3:

```
$ : increment(stack_top.table_entry);
    if termination_conditions_hold then goto fin;
```

where fin is a label placed just before the **end** of the outmost block.

6.2 *Skip visiting the father.*

Suppose A_i is the first node satisfying the termination conditions and the visit sequence associated with A_i is VS(p). The incremental evaluator will terminate after executing \$ in VS(p). However, among the attribute instances that are evaluated before executing that \$, there may be some members of EQUVAL. VS(p) is generally of the following form:

$$\mathrm{VS}(p) = \ldots, A_i.a, \ldots, \; \$, v(m, 0)$$

where $A_i.a$ is the last synthesized attribute occurrence. Among the items between $A_i.a$ and \$ there may be some visits to the father of A_i which may further lead to visits to brothers or more ancient ancestors of A_i. Anyway, the attribute instances attached to such nodes definitely fall outside the subtree rooted at A_i and therefore must belong to EQUVAL. In other words, they need not be re-evaluated. This means that the incremental evaluator can skip all visits to the father of A_i between $A_i.a$ and \$. In order to achieve such a skip another special symbol £ is inserted immediately after the last synthesized attribute occurrence in each visit sequence. When the incremental evaluator executes £ the termination conditions are tested. If they hold true, then a flag skip_father is set up, meaning that from now on all visits to the father can be skipped. Naturally, when the incremental evaluator comes to $, the termination conditions need not be tested once again, because the result of testing can be learnt by checking whether the flag skip_father has been set up or not.

Digression. This improvement also brings in a new insight into construction of visit sequences. Indeed, in order to skip visiting the father as described above, it should always be favorable to have synthesized attribute occurrences evaluated as early as possible. In other words, if in a visit sequence there is a father visit $v(k, 0)$ and a synthesized attribute occurrence $X.a$ such that $v(k, 0)$ precedes $X.a$ but does not contribute to evaluation of $X.a$, then $v(k, 0)$ should be swapped with $X.a$ if possible.

6.3 *Skip visiting brothers.*

The termination condition (2) says that all synthesized attribute instances of A_i must be in EQUVAL. As a matter of fact it can be a bit more relaxed as described below.

Suppose s is such a synthesized attribute that if s is associated with a right-hand side symbol, then s is only used in defining synthesized attributes of the left-hand side symbol. Let us call s of type UP. Now suppose N is an ancestor of A such that the incremental evaluator has executed £ in the visit sequence associated with N and found that the synthesized attributes violating condition (2) are all of type UP. Then, by definition, it is easy to see that the attribute instances belonging to the subtrees rooted at brothers of N must all be in EQUVAL. Therefore, it is superfluous to visit such brothers of N hereafter. In order to skip visiting them, a flag skip_brothers is introduced. If the node being visited is an N as described above, then the flag skip_brothers is set up. After resuming execution of the visit sequence associated with the father of N, all visits to brothers of N are skipped if the flag skip_brothers has been set up.

This improvement could be better appreciated if one notices that a good many attributes used in practice are of type UP.

7. Conclusion.

In this paper, we have presented a method of augmenting a conventional OAG evaluator into an incremental one and suggested three ways to improve the time requirement of the resulting incremental evaluator. Noticing the two remarks made at the end of § 4 as well as the improvements, it is easy to see that the time complexity of our incremental evaluator can be represented in terms of the size of the set MARK. Hence, by theorem 2, we conclude that our incremental evaluator accomplishes evaluation of a modified semantic tree in time proportional to the amount of attribute instances affected by the modification.

Furthermore, as nothing more than the static determinacy of OAG evaluators has been utilized in the augmentation, the method can be readily extended to tree-walker evaluators constructed by the algorithm given in [2] for any non-circular AGs.

It is fair to note that we have been primarily dedicated to introducing the method, rather than a specific implementation. However, there should be no difficulties in implementing all ideas given in this paper.

The inherent drawback of all incremental evaluators, including ours, is the requirement of a large amount of storage for saving the evaluation result of an original semantic tree. Indeed, in an evaluation process a good many attribute instances are ephemeral and allocated only temporary storage by a conventional evaluator. Considering such a fact, incremental evaluators appear to be rather space-expensive. Nevertheless, the cost in space is paid off by the gain in time when using an incremental evaluator in an interactive environment. Very promisingly, along with the increasing interactivity in programming and the decreasing cost of memory, incremental evaluators will more and more go into use in fields such as code generation, data flow analysis, separate compilation, etc.

Appendix.

Below we describe for a given node A how to compute the stack configuration at MOMENT$(1, A)$ in an evaluation process of T'.

Let p_1 be the production applied at A, VS(p_1) the visit sequence associated with p_1, and A the ith son of A_1. From the mechanism of the OAG driver it is easy to see that the topmost stack element must be (A_1, d_1+1), where d_1 is the index of the table entry $v(1,i)$ in VS(p_1). Further suppose that $v(1,i)$ occurs in the kth segment of VS(p_1), p_2 is the production applied at A_2, VS(p_2) the visit sequence associated with p_2, and A_1 the jth son of A_2. Again from the mechanism of the OAG driver it is easy to see that the second topmost element must be (A_2, d_2+1), where d_2 is the index of the entry $v(j,k)$ in VS(p_2). Continuing this reconstruction until the root of T' is reached, we can recover all elements in reverse order compared with how they were pushed down into the stack, so long as we have a means of computing $d_1, d_2, \ldots$.

Two auxiliary functions are devised for computing $d_1, d_2, \ldots$. One is called TRACEUP and maps the visit number k, the son number i, and the production indicator p into the index d of the entry $v(k,i)$ in the visit sequence VS(p) associated with p. Thus, for instance, we have

$$d_1 = \text{TRACEUP}(1, i, p_1), \qquad d_2 = \text{TRACEUP}(k, j, p_2), \qquad \ldots$$

The other is called SEGMENTNO and maps the index d of an entry v and the production indicator p into the segment number s such that v occurs in the sth segment of the visit sequence associated with production p. Thus, for instance, we have

$$k = \text{SEGMENTNO}(d_1, p_1).$$

For any node N, we can learn from parsing the son number of N as well as the number of the production applied at N. Therefore, knowing TRACEUP and SEGMENTNO is enough for computing $d_1, d_2, \ldots$.

Acknowledgement.

The author wishes to thank G. A. Green and D. H. Wanvik at RUNIT, the Computing Centre of the University in Trondheim, for their helpful comments on the paper. The scholarship provided by NORAD, the Norwegian Agency of International Development, is gratefully appreciated.

REFERENCES

1. A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*. Vols. 1 & 2, Prentice-Hall, (1972 & 1973).
2. R. Cohen and E. Harry, *Automatic generation of near-optimal linear-time evaluators for non-circular attribute grammars*. Conference record of the 6th ACM symposium on principles of programming languages, (January 1979), 121–134.
3. A Demers, T. Reps and T. Teitelbaum, *Incremental evaluation for attribute grammars with application to syntax-directed editors*. Conference record of the 8th ACM symposium on principles of programming languages, (January 1981), 105–116.
4. H. Ganzinger, R. Giegerich, U. Moencker and R. Wilhelm, *A truly generative semantics-directed compiler-generator*. Proceedings of the SIGPLAN'82 symposium on compiler construction, (June 1982), 172–184.
5. U. Kastens, *Ordered attributed grammars*, Acta Informatica, Vol. 13 (1980), 229–256.
6. T. Reps, *Optimal-time incremental semantic analysis for syntax-directed editors*. Conference record of the 9th annual ACM SIGACT_SIGPLAN symposium on principles of programming languages, (January 1982), 169–176.
7. D. S. Yeh, *On incremental shift-reduce parsing*, BIT 23 (1983), 36–48.