Implementation of Visit-Oriented Attribute Evaluators

Uwe Kastens

University of Paderborn

Abstract

A large class of attribute evaluators (AE) are controlled by visit-sequences describing the computations during a tree walk. In this paper it is shown how such a control structure is implemented systematically. An AE has to provide storage for the attribute values associated to the tree nodes. Naive storage allocation techniques are not tolerable for practical applications. An introduction to well elaborated methods of attribute storage optimization is given. They are applied automatically by practical AG systems.

1 Introduction

An Attribute Grammar (AG) specifies computations to be executed in trees. Their structure is described by the underlying CFG. From those specifications attribute evaluators (AEs) are derived which execute the specified computations in any particular tree. The construction of an AE for a given AG has to solve two problems: The evaluation order of the computations has to be chosen such that it obeys the specified dependencies between computations. (This paper considers sequential evaluation only.) Storage has to be allocated for attribute values which are defined in one computation and used in others.

There are several methods for systematic construction of AEs with respect to the evaluation order. We here concentrate on **visit-oriented AE construction**, which determines the evaluation order at AE generation time. The control structure of the AE is

chosen such that it executes the specified computations during a walk through the given tree. A systematic construction starts from the **computation patterns** specified for each production context of the AG. The **dependency patterns** derived from them are analyzed and transformed into **evaluation patterns** for the AE, cf. [Kas91b]. They are sequences of tree walk operations and computations in the case of the method considered here. Those **visit-sequences** were introduced in [Kas80]. They are presented in Sect. 2. For a discussion of the technique of computing visit-sequences from dependency patterns, and of the restrictions required for the AG we refer the reader to [Kas80]. In Sect. 3 it is described how visit-sequences can be constructed such that attribute evaluation is interleaved with tree construction. These techniques, known as parse-time attribution, shorten the tree walk and may save tree space. In Sect. 4 different implementation techniques for visit-sequences are discussed.

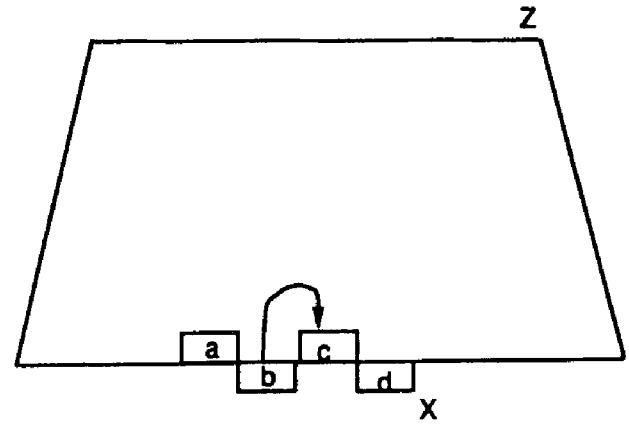
A naive storage allocation would allocate space for the attribute values in the tree nodes. Since many of the attributes are used only during a rather short time, they can be allocated in global variables or stacks. In Sect. 5 a technique for attribute storage optimization is presented, which is a generalization of the method given in [EnD90]. We here again concentrate on those techniques which make the allocation decision at AE generation time, in order to avoid additional runtime and space requirements at AE execution time.

2 Visit-Sequences

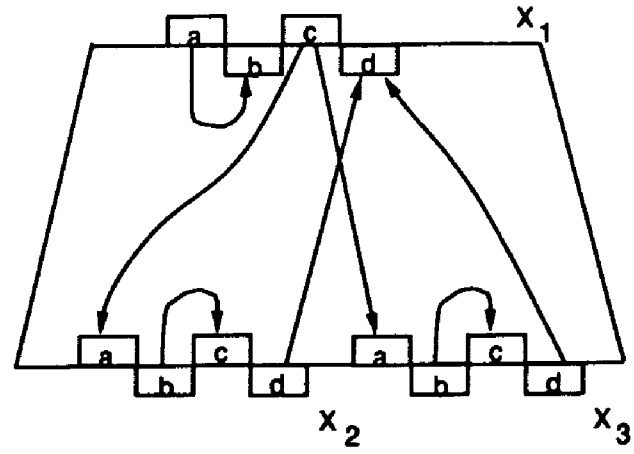
An AG specifies a **computation pattern** for each context p given by a production of the underlying CFG for the abstract syntax. It comprises computations to be executed in each instance of that context within a particular structure tree, i.e. a tree node with its immediate descendants which represent an application of the production p. Dependencies between these computations and those of adjacent contexts are specified by defining and applied occurrences of attributes of symbols in p. Hence the attribution of p also specifies a **dependency pattern**. A visit-oriented attribute evaluator executes the specified computations during a tree walk in an order which is compatible with the specified dependency patterns. Its control structure has an **evaluation pattern** for each context p. In

```
NONTERM X:      a, b, c, d:     int;
RULE p: Z ::= X
STATIC
        X.a = f ();
        X.c = g (X.b);
        h (X.d);
END;
RULE q: X ::= X X
STATIC
        X[1].b = g (X[1].a);
        X[2].a = g (X[1].c);
        X[3].a = g (X[1].c);
        X[2].c = g (X[2].b);
        X[3].c = g (X[3].b);
        X[1].d = s (X[2].d, X[3].d);
END;
RULE r: X ::=
STATIC
        X.b = g (X.a);
        X.d = g (X.c);
END;
```

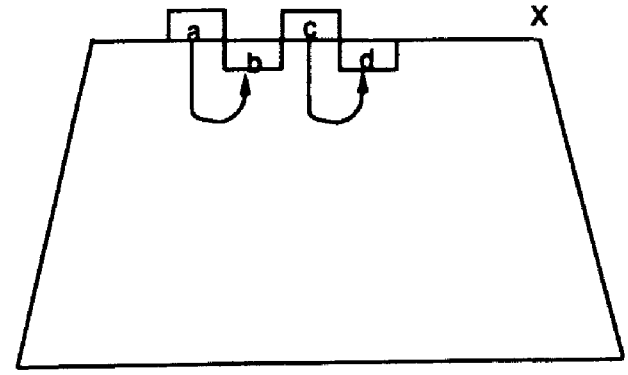
Fig. 2.1 Example for computation patterns



p: Z ::= X



q: X ::= X X



r: X ::=

Fig. 2.2 Dependency patterns for Fig. 2.1

Fig. 2.1 a small, artificial AG is given as an example for the specification of computation patterns; Fig. 2.2 shows a graphical representation of the dependency patterns; the evaluation patterns are given in Fig. 2.3. In this paper we consider only those evaluators where the evaluation patterns are sequences of computations and basic tree walk operations for visits of adjacent contexts, introduced as **visit-sequences** in [Kas80]. (Any pass-oriented evaluator is a special case of this evaluator class; evaluators for absolutely non-circular AGs and for general well-defined AGs are not covered by this class.)

Each visit-sequence vs_p is a sequence $v_1, \ldots, v_m$ where each v_k belongs to one of three operation classes:

v_k	$= comp_k$	a computation
v_k	$= visit(i,j)$	visit the i-th descendant ($i \geq 1$) for the j-th time ($j \geq 1$), also denoted as $\downarrow_j X_i$
v_k	$= leave(j)$	visit the ancestor context for the j-th time ($j \geq 1$), also denoted as $\uparrow_j$

The set of visit-sequences controlling an attribute evaluator must obey tree walk requirements and dependency requirements.

Execution of the attribute evaluator for any particular structure tree has to perform a complete tree walk. For the moment we assume that it starts and ends at the root of the tree. This is achieved by applying the visit-sequences at each node, and moving between adjacent contexts by *visit*- and *leave*-operations. Fig. 2.4 shows the interaction between vs_p and vs_q of our example graphically.

The tree walk requirements can be specified as follows. For each nonterminal X of the CFG there is a number s, we say X is a **s-visit** symbol. The root symbol is defined to be 1-visit. Terminal symbols are not visited. Then the following conditions hold for each vs_p:

1. If X is the lefthand side symbol of production p, vs_p contains operations *leave* (1), ..., *leave* (s) in this order, where *leave* (s) is the last element of vs_p. We say vs_p has s sections each ending with a *leave* (j).

2. If X occurs as the i-th symbol on the righthand side of p, then vs_p contains operations ***visit (i, 1), ..., visit (i, s)*** in this order.

```
Visit-sequence for p: Z ::= X

        comp (X.a = f ()),
        visit (X, 1),
        comp (X.c = g (X.b)),
        visit (X, 2),
        comp (h (X.d)),
        leave (1)

Visit-sequence for q: X ::= X X

        comp (X[1].b = g (X[1].a)),
        leave (1),

        comp (X[2].a = g (X[1].c)),
        visit (X[2], 1),
        comp (X[2].c = g (X[2].b)),
        visit (X[2], 2),
        comp (X[3].a = g (X[1].c))
        visit (X[3], 1),
        comp (X[3].c = g (X[3].b)),
        visit (X[3], 2),
        comp (X[1].d = s (X[2].d, X[3].d)),
        leave (2)

Visit-sequence for r: X ::=

        comp (X.b = g (X.a)),
        leave (1),

        comp (X.d = g (X.c)),
        leave (2)
```

Fig. 2.3 Visit-sequences for Fig. 2.1

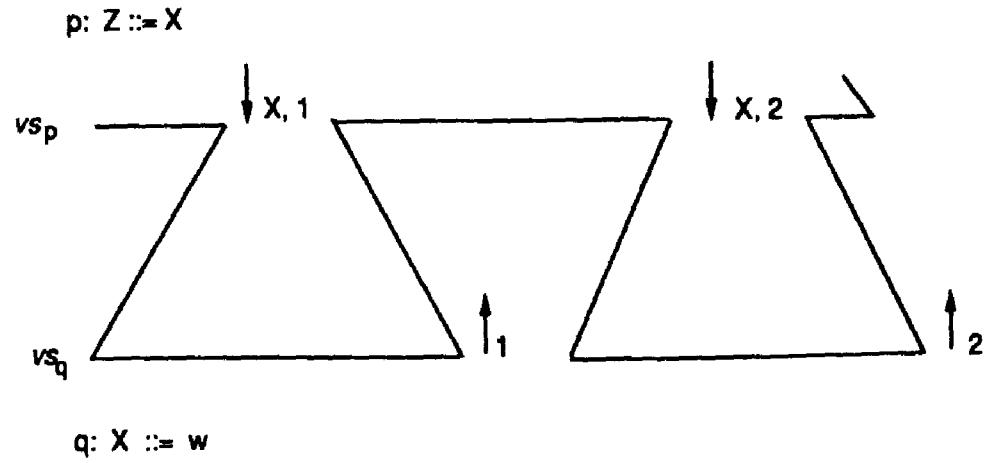


Fig. 2.4 Interaction between visit-sequences

In our example X is a 2-visit symbol. These conditions guarantee that the visit-sequences fit together for any structure tree, and that a complete tree walk is performed.

The dependency requirements can be specified as follows. A computation $comp_a$ in a visit-sequences vs_p may define an attribute of the context p, and/or may use some (or none) attributes. Then the following conditions hold for each vs_p:

1. If $v_k = comp_a$ defines an attribute a and $v_l = comp_b$ uses a, then $k < l$.
2. If $v_k = comp_a$ defines an attribute a of a symbol occurrence X_i in p, and $v_l =$ *visit (i, j)* or $v_l =$ *leave (j)* (and X_i is the lefthand symbol of p) leads to a use of a, then $k < l$.

Any system generating visit-sequence controlled attribute evaluators form AGs, like GAG [KHZ82], LIGA [Kas89], or the Synthesizer Generator [ReT89], produces visit-sequences, which obey the above conditions. They apply algorithms for dependency analysis as described in [Kas80].

3 Parse-Time Visit-Sequences

The structural requirements for visit-sequences described in Sect. 2 assume that a tree walk is performed on a tree which is completely build prior to attribute evaluation. The

tree walk can be shortened, and the maximum storage for the tree can be reduced, if evaluation already starts while the tree is being build. This technique is called **parse-time attribution**, since the problem was first attacked for parser driven tree construction. Parser generators have mechanisms to attach actions to productions which could be used to drive tree construction or attribution or both. This is obvious and simple for recursive decent parsers, and for bottom-up parsers if restricted to synthesized attribution only. Techniques like those of Tarhio [Tar89] allow for some inherited computations at parse-time as well. However, such techniques can be generalized to attribution merged with tree construction, not necessary driven by a parser. In the following we first express the problem in terms of visit-sequences and then distinguish between the top-down case and the bottom-up case.

Let us consider a production of the abstract syntax taken from an AG

$$p : X_0 ::= X_1 \ldots X_n$$

It can be rewritten as specification for tree construction augmented by actions:

$$p : X_0 ::= \&u_1\ X_1\ \&u_2\ X_2\ \ldots\ \&u_n\ X_n\ \&u_0$$

It specifies that the tree constructor builds a node for context p by executing the righthand side from left to right, where the X_i stand for building the i-th subtree, and each $\&u_i$ for executing some attribution operations. The p-node itself is either build immediately before that sequence in the top-down or after it in the bottom-up case. (The reader should not worry about storing attributes which may be computed in u_i before the node for X_i is made. They can always be kept on a stack, and may be copied into the node at the end of the sequence if their life-times exceed the tree construction phase.) Such a specification for tree construction can automatically be extracted from an AG specification. Furthermore it can be mapped to a concrete syntax augmented with the same actions, then specifying a parser that drives tree construction, as described in [Kas91a].

An AG system has to determine which computations can be executed during tree construction. It performs dependency analysis on the AG and expresses the results in

terms of visit-sequences vs_p. The first section (up to the *leave (1)* operation) of each vs_p describes the operations at tree construction time. Hence the vs_p have the form

$$vs_p: \ u_1, \ visit(1,1), \ u_2, \ visit(2,1), \ldots, u_n, \ visit(n,1), u_0, \ leave(1), \ w$$

The u_j may contain computations and other visits, such that the general restrictions for visit-sequences of Sect. 2 hold. As in the constructor specification above the *visit (i, 1)* operations stand for subtree construction and are executed under control of the tree constructor. This **construction-section** is followed by the rest w of the vs_p which is constructed as usual.

In the case of **top-down** tree construction the AG system is free to allocate any operations in the u_i of the construction-section, provided the dependency requirements hold. That result can be achieved by a dependency analysis in two phases: first LAG (1) analysis for one top-down left-to-right pass, then usual visit-sequence analysis for the remaining computations. Obviously the u_i can be easily inserted as actions into a recursive descent scheme.

In the **bottom-up** case the decision for constructing a node for context p is made at the end of the construction section, if no further information about the CFG is used. Hence all the u_i have to be empty except u_0. this approach yields a first bottom-up attribution pass with synthesized computations only.

More computations can be performed at construction time if the decision for context p can be made earlier. This problem can be solved using the concept of **free positions** in LR-grammars introduced by Purdom and Brown [PuB80]. Consider the tree construction specification for production p above. A position in a production is called free if an action like $\&u_i$ can be inserted without violation of the grammar class (LR (1) or LALR (1)). The information on free positions may be transformed from the concrete into the abstract syntax. Knowing which of the u_i are at free positions we can allow the visit-sequence construction to allocate operations in those u_i. The AG analysis can achieve that in a way similar to the two phase approach for the top-down case: The first phase performs LAG (1) analysis, and additionally removes computations from the first pass which would be allocated on non-free positions (and all computation which depend on them). The LALR (1), parser generator Cola [Pro89] computes the free positions according to the

Purdom and Brown algorithm. We are going to combine it with the LIGA system in the way described above.

Certain attribute storage optimizations (see Sect. 5) eliminate transfer computations, and hence reduce the number of non-free positions being hit. In combination these techniques yield effects like those achieved by the approach of [Tar89].

We finally describe a simple improvement for tree storage management. A rather large amount of storage is needed for the pure structure information of the tree, not regarding any attribute stored in tree nodes. During the process of attribute evaluation attribution for some subtrees gets completed. Since they are not accessed again their storage can be deallocated. If the whole tree is constructed prior to attribute evaluation, that deallocation would not reduce the maximum storage requirement for the tree. If attribution starts during tree construction (at parse-time), it may be completed for some subtrees before others are build. Then deallocation may significantly reduce the space needed. At the end of a visit-sequence vs_p none of the subtrees of that p instance will be visited again, and none of their attributes will be used any more. Hence at that point the descendant nodes may be deallocated.

The combination of both techniques can automatically avoid tree construction at all for certain subtrees: If all visits to a context p are contained in the construction section of visit-sequences for the upper contexts q then the nodes for p can be stored on a stack instead of allocating and deallocating tree storage for them. (This is a basic pattern for attribute storage optimization too, as described in Sect. 5.) That decision can be made on base of the visit-sequences at generation time. As a special case these optimizations automatically yield a one pass attribution without a tree being stored at all, if the dependencies of the AG, the CFG and the tree construction driver obey the stated conditions.

4 Implementing Control

The set of visit-sequences for an AG specifies the control structure of an attribute evaluator. The implementation techniques mainly differ in the implementation of the tree walk operations (coroutine calls, procedure calls, operations on an explicit stack), and in the encoding of the visit-sequence elements (directly executable or table encoded). There are

different consequences for runtime, code size, and storing of globalized attributes, which are discussed in Sect. 5.

The essential concepts of visit-sequence implementation are:

- a node stack for the tree nodes of the active node up to the root,
- the control state for each node, i.e. the position of the visit-sequence element which is executed next.

Each execution of a visit-sequence element advances the state of the actual node. A *visit* pushes a node on the stack, which is popped by a *leave* operation. The techniques discussed here use different implementations of these concepts.

We first present an **object oriented** implementation which is closest to the concept of visit-sequences. Each tree node can be considered as an active object of a class given by the production. It executes the operations of its visit-sequence within its context and in cooperation with the objects adjacent in the tree. Such an object behaves as a coroutine interacting with its neighbours. Fig. 4.1 gives a rough idea of such an implementation by SIMULA classes [DMN70] for the example of Sect. 2.

A descendant visit is implemented by a coroutine switch (*call*-operation in SIMULA). An ancestor visit switches control back to the calling object (*detach*-operation). The control state is the local execution pointer of each coroutine instance. The code is directly executable, but there are penalties in space and execution time caused by the runtime system for general coroutine management. It should be pointed out that the classes for tree nodes are defined in a very natural way: A class for a production is a subclass of its lefthand side symbol (denoted by a prefix class in SIMULA). The class of the symbol with its tree attributes is common to the production class, which additionally contains the descendant references for the particular righthand side. A corresponding data structure is used in all of these implementation techniques.

The second technique to be presented here is the transformation of visit-sequences into **recursive procedures**. Each visit-sequence can be decomposed into a sequence of sections such that the j-th section ends with the operation *leave (j)*. Fig. 4.2 shows the procedures for the visit-sequences of our example. A descendant visit *visit (i, j)* can be considered as a procedure call which leads to the *j-th* section of the visit-sequence

```
CLASS cZ; BEGIN END;
cZ CLASS p;
BEGIN REF (cX) X;
        X :- NEW cX;    Construction of subtrees
        DETACH;
                        Visit-sequence for p: Z ::= X
        X.a := f;
        CALL (X);
        X.c := g (X.b);
        CALL (X);
        h (X.d);
END;
CLASS cX; BEGIN INTEGER a, b, c, d; END;
cX CLASS q;
BEGIN REF (cX) X1, X2;
        X1 :- NEW cX;   Construction of subtrees
        X2 :- NEW cX;
        DETACH;
                        Visit-sequence for q: X ::= X X
        X1.b := g (X1.a);
        DETACH;
        X2.a := g (X1.c);
        CALL (X2);
        X2.c := g (X2.b);
        CALL (X2);
        X3.a := g (X1.c);
        CALL (X3);
X3.c := g (X3.b);
        CALL (X3);
        X1.d := s (X2.d, X3.d);
END;
cX CLASS r;
BEGIN
        DETACH;
                        Visit-sequence for r: X ::=
        X.b := g (X.a);
        DETACH;
        X.d := g (X.c);
END;
```

Fig. 4.1 SIMULA classes implementing visit-sequences

```
void p_1 (n) Node n;
{                    /* Visit-sequence for p: Z ::= X */
        n->X->a = f ();
        (* (n->X->proc)) (n->X);        /* visit (X, 1) */
        n->X->c = g (n->X->b);
        (* (n->X->proc)) (n->X);        /* visit (X, 2) */
        h (n->X->d);
}

void q_1 (n) Node n;
{        /* Visit-sequence section 1 for q: X ::= X X */
        n->b = g (n->a);
        n->proc = q_2;                  /* next section */
}

void q_2 (n) Node n;
{        /* Visit-sequence section 2 for q: X ::= X X */
        n->X2->a = g (n->c);
        (* (n->X2->proc)) (n->X2);  /* visit (X[2], 1) */
        n->X2->c = g (n->X2->b);
        (* (n->X2->proc)) (n->X2);  /* visit (X[2], 2) */
        n->X3->a = g (n->c))
        (* (n->X3->proc)) (n->X3);  /* visit (X[3], 1) */
        n->X3->c = g (n->X3->b);
        (* (n->X3->proc)) (n->X3);  /* visit (X[3], 2) */
        n->d = s (n->X2->d, n->X3->d);
}

void r_1 (n) Node n;
{            /* Visit-sequence section 1 for r: X ::= */
        n->b = g (n->a);
        n->proc = r_2;                  /* next section */
}

void r_2 (n) Node n;
{             /* Visit-sequence section 2 for r: X ::= */
        n->d = g (n->c);
}
```

Fig. 4.2 Recursive procedures implementing visit-sequences

```
n = root;
while (n != NULL) {
switch (n->state) {
                        /* Visit-sequence for p: Z ::= X */
case p_1:   n->X->a = f;
            n->state = p_2;                /* visit (X, 1) */
            nstack[ns++] = n; n = n->X; break;
case p_2:   n->X->c = g (n->X->b);
            n->state = p_3;                /* visit (X, 2) */
            nstack[ns++] = n; n = n->X; break;
case p_3:   h (n->X->d);
            n = nstack[--ns]; break;                /* leave */
            /* Visit-sequence section 1 for q: X ::= X X */
case q_1:   n->b = g (n->a);
            n->state = q_2;                         /* leave */
            n = nstack[--ns]; break;
            /* Visit-sequence section 2 for q: X ::= X X */
case q_2:   n->X2->a = g (n->c);
            n->state = q_3;             /* visit (X[2], 1) */
            nstack[ns++] = n; n = n->X2; break;
case q_3:   n->X2->c = g (n->X2->b);
            n->state = q_4;             /* visit (X[2], 2) */
            nstack[ns++] = n; n = n->X2; break;
case q_4:   n->X3->a = g (n->c));
            n->state = q_5;             /* visit (X[3], 1) */
            nstack[ns++] = n; n = n->X3; break;
case q_5:   n->X3->c = g (n->X3->b);
            n->state = q_6;             /* visit (X[3], 2) */
            nstack[ns++] = n; n = n->X3; break;
case q_6:   n->d = s (n->X2->d, n->X3->d);
            n = nstack[--ns]; break;                /* leave */
               /* Visit-sequence section 1 for r: X ::= */
case r_1:   n->b = g (n->a);
            n->state = r_2;                         /* leave */
            n = nstack[--ns]; break;
               /* Visit-sequence section 2 for r: X ::= */
case r_2:   n->d = g (n->c);
            n = nstack[--ns]; break;                /* leave */
}}
```

Fig. 4.3 Tree walk with directly controlled stack

associated to the *i-th* subtree. The descendant node is an argument of the call. Hence the node stack is implemented by elements of the runtime stack. The state of each active node is represented by the actual program pointer and the return addresses for visit-calls on the runtime stack. The state of inactive nodes is stored in a node component that contains the address of the procedure to be called when the node is visited. (The size of this node component may be reduced to that of an integer that encodes the production. Then a table maps production encodings and visit numbers to section procedures.) Obviously such an implementation can be smoothly combined with recursive decent parsers. (cf. Sect. 4). If attribute storage optimization is applied, as discussed in Sect. 5 stacked attributes can be allocated on the runtime stack in the activation record of those procedures.

In a third implementation technique the stack is organized explicitly in the attribute evaluator. A descendant visit is implemented by a push operation on the stack, a *leave* operation by a pop operation. The control points after each descendant visit are encoded as case labels and stored as the state of the node. The control structure of the attribute evaluator is a loop with a switch over the state of the actual node, as shown in Fig. 4.3. Compared with the procedure implementation there are the following disadvantages: A fixed stack size imposes a severe restriction upon the maximum depth of the trees. Furthermore, the checks for stack overflow (omitted in Fig. 4.3) cost additional runtime. A dynamic stack allocation is even more costly.

Finally it should be mentioned that the visit-sequences could be implemented by table-driven technique-instead of the three directly executable implementations above. The visit-sequence elements (*visit-* and *leave*-operation, code of a computation sequence) can be encoded in a table, which is interpreted by a control loop of the evaluator. Its body decodes the operations and performs the same operations as described for the technique of an explicit stack described above. Accessing and decoding of table entries costs additional runtime. The code size can be reduced if a compact table representation is chosen. In practice that gain in storage is not very large with respect to the size of the code for computations of the attribution.

In general the procedure implementation of visit-sequences is the smallest and fastest with respect to code size and runtime for control and storing attributes.

5 Attribute Storage Optimization

An attribute evaluator is responsible for storing values of attribute instances which are defined by one computation and used in others. We call these dependencies **value dependencies**, cf. [Kas91b]. Computations may be specified to depend on others without obtaining a value form them, e.g. in order to obey to the calling sequence of functions which alter an internal state of a module. Attributes used only for such sequencing dependencies do not need storage. They can completely vanish from the attribute evaluator as soon as the evaluation patterns (the visit-sequences) are determined.

A naive approach defines the record types for tree nodes such that a node for a symbol X has a component for each attribute of X. Hence each attribute instance occupies storage as long as its tree node exists. However that space is used only during the lifetime of the attribute instances, from its definition to its last use. Techniques for attribute storage optimization allocate storage for attributes in variables or stack elements and reuse them for several instances such that their lifetimes are not in conflict. Implementations of optimization-techniques have proven to yield significant improvements of storage requirements in practical applications, cf. [KHZ82,Sch89]. The need for such storage improvements is often emphasized by the argument that otherwise huge data structures like definition tables are copied all over the tree. This is a misleading argument: References to structured values can always be implemented by pointers. That is either done by the type mapping in an AG system like GAG [KHZ82], or by modules used in the AG for a system like LIGA [Kas89]. But even for attributes of pointer or integer size the large number of attribute instances justifies the effort for optimization.

Completely different directions for storage optimization have been followed: In [Rāi79] a technique is suggested which makes allocation decisions at evaluator runtime. It requires that space consuming dependency information is available, its analysis costs additional execution time. Sonnenschein [Son85] suggests, that the user specifies certain attributes to be global, then the AG system tries to find suitable evaluation patterns for that requirement. This approach puts the burden of implementation decisions back to the user. In this paper we concentrate on those techniques which make all allocation decisions at generation time on the base of a given set of evaluation patterns (visit-sequences). The

different techniques in [EnD90,FaY86,Kas87] coincide in this principle. They are successfully applied in systems like GAG, LIGA, and LINGUIST [Far82]. In [Hal87] additional techniques are presented which use the freedom in visit-sequence construction for better storage optimization. In the following we first explain the basic principle, and then present a decision algorithm, based on [EnD90], which removes as many attributes from the tree as possible.

The optimization problem can be stated as follows: Given a set of visit-sequences and an attribute $X.a$. Is it true for any structure tree, that the values of all instances of $X.a$ can be stored in a single variable (a single stack) such that they are never in conflict during evaluation according to the visit-sequences? If the answer is no, $X.a$ is allocated as a node component. The problem is solved by analyzing lifetimes on the base of the defining occurrences of $X.a$ and its last applied occurrences in the visit-sequences. That analysis may be pessimistic, i.e. decide to allocate a node component if a stack or variable were possible, or a stack if a variable would be sufficient. So we have to show that in case of a variable the lifetimes of all instances are disjoint, and that in case of a stack they are either disjoint or properly nested. If they partially overlap a node component has to be chosen. Fig. 5.1 shows the situations graphically over the time axis of the evaluator.

In order to check whether the above conditions hold for any instance of $X.a$ in an arbitrary structure tree we reformulate them in terms of visit-sequences. Since an instance of $X.a$ is accessible in two adjacent contexts any combination of such contexts have to be analyzed whether they contain conflicting lifetimes or visits may lead to those. Fig. 5.2 shows parts of visit-sequences for contexts p and q which are adjacent at X. For the moment we assume that X does not occur elsewhere in p and q. The visit-sequences are connected by lines where visit or leave operations lead from p to q and back; visits to other parts of the tree are indicated by arrows only. We now can describe the lifetime of $X.a$ in this context by points in the visit-sequences, e.g. the definition at a and the last use at b if $X.a$ is inherited, or the definition at e and the last use at f if $X.a$ is synthesized. Since the AG is not restricted to Bochmann Normal Form the last use of an inherited (synthesized) $X.a$ may also be in vs_p (vs_q).

We first describe the lifetime analysis informally; the precise definitions of the conditions are given at the end of this section. The lifetime of each occurrence $X_i.a$ of $X.a$ in

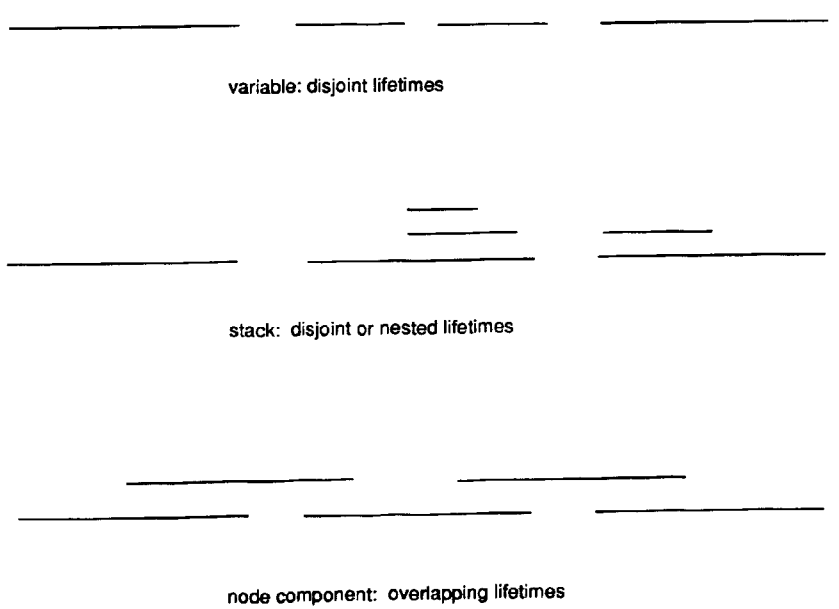


Fig. 5.1 Lifetimes of $X.a$ during evaluation

a context $p: \ X_0 ::= X_1 \ldots X_n$ is described by a subsequence of the visit-sequence vs_p. We say the lifetime of $X_i.a$ in p is the tuple *(i, r, s)* such that the lifetime begins at the element e_r in vs_p and ends at e_s. In the example of Fig. 5.2 we may get a lifetime *(i, a, c)* in vs_p if $X_i.a$ is defined at point a; or a lifetime *(i, d, f)* in vs_p if $X_i.a$ is defined at point e in vs_q. In fact the lifetimes in vs_p are determined by all combinations of contexts p and q which are adjacent in $X_i = X$ Hence in our example larger lifetimes than the two shown above can result from consideration of all adjacent pairs. We defer the details of the computation of lifetime tuples for visit-sequences to the end of the section.

On the base of the lifetimes expressed in terms of visit-sequence operations we have to check two kinds of conditions for both the variable allocation and the stack allocation:

Reachability conditions: From within such a lifetime no *visit* or *leave* operation

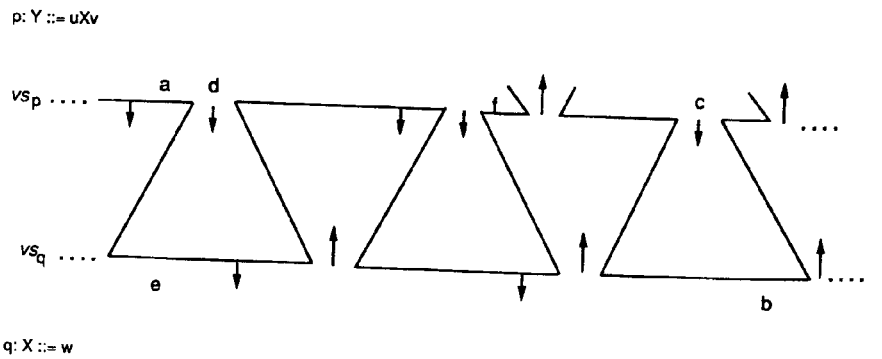


Fig. 5.2 Adjacent visit-sequences for lifetime analysis

may lead to another alive instance of $X.a$ such that the lifetimes are in conflict with the allocation.

Multiple occurrence condition: The lifetimes of several occurrences of $X.a$ in a vs_p must not be in conflict with the allocation condition.

In fact the multiple occurrence condition can be dropped if a more sophisticated allocation strategy is applied. Assume that $X.a$ can be implemented by variable according to the reachability condition. If there are vs_p where at most n occurrences of $X.a$ are alive at the same time, then n variables are allocated for $X.a$. The computations in vs_p use them correspondingly. Before a visit to the adjacent context the value is swapped to the first of the n variables, such that it can be accessed at a well-defined position. In the case of stack allocation the same technique is applied to the topmost n stack elements. This technique, described in [FaY86,Hal87], increases the number of attributes being allocated to variables or stacks. The price is the additional runtime and code size for the swap operations. If the technique is applied, the allocation decision is made on the base of the reachability condition only.

For variable allocation the reachability condition should be checked as stated above, whereas for stacks specific implementation considerations should be taken into account. For each stackable attribute an individual stack may be allocated. If it is a fixed size array it imposes limits on the nesting depth of the program, and the push operations cost additional runtime for bound checking. Alternatively additional runtime is required for dynamically allocated stacks. However, if the visit-sequences are implemented by recursive procedures, as described in Sect. 4, elements of all stacks can be allocated on the runtime stack for procedure calls. Hence no additional limit for individual stack sizes is imposed.

The runtime stack implementation is achieved by the following technique: In the procedures for the sections of a visit-sequence vs_p the stack attributes of the descendants of p are declared as local variables. Their addresses are passed as parameters of the calls for the descendant visits. Hence the push operations corresponds to the procedure entry and the pop operations to the procedure exit.

For this technique we have to reconsider the lifetimes of the attribute occurrences in vs_p: We first consider attributes having a lifetime which does not include a *leave* operation in any visit-sequence for the upper context, we say the attribute is 1-visit. Their lifetime is now extended to the whole section. Obviously the reachability condition holds for the extended lifetime if it holds for the exact lifetime. Even multi-visit attributes can be implemented by this technique: At the end of a procedure for a visit-sequence section those stack attributes, which are still alive in the next section, are pushed on an individual stack for that attribute, and popped on entry of the next section procedure into its local variable. The reachability condition guarantees that the stack discipline is obeyed.

Finally a further improvement can be applied to those attributes which can be allocated to variables: If A is a set of attributes which can be allocated to variables, it can be checked whether one single variable instead of individual ones can be used. For a given set A the check is easily described: The occurrences of all attributes in A are considered as different occurrences of the same attributes for the reachability condition and the multiple occurrence condition. The resulting storage improvement is in general not significant. But all the transfer rules (assignments of the form $X.a := Y.b$) between attributes in A can be eliminated. In [Gan79] it is proven that this optimization with

respect to a maximum of eliminated transfer rules is NP-complete. However, good results can be obtained if those attributes are collected in one set which are generated from the expansion of the constructs for remote attribute access (INCLUDING and CHAIN, described in [Kas91b]).

Description of the allocation decision algorithm

The allocation algorithm described here is based on the idea of [EnD90]. It is generalized in the following respects:

- The AG is not restricted to Bochmann Normal Form, i.e. attributes can be used in the same context where they are defined.
- There are no further assumptions made for visit-sequence construction than those specified in Sect. 2, i.e. the definition of attributes need not immediately precede the corresponding visit.

Furthermore we assume that the improvements for multiple occurrences are applied as presented above. Hence we omit the check of those conditions.

In the following the lifetime analysis for a single attribute $X.a$ is described. It is repeated for all attributes of the AG. In a second phase sets of attributes which each fulfill the variable condition may be analyzed in order to be implemented by a single variable. The algorithm has three steps: computation of lifetime tuples, computation of reachability sets, and check of lifetime conditions.

Computation of lifetime tuples. We express lifetimes of attribute occurrences in terms of a single visit-sequence, rather than of pairs for adjacent contexts. The lifetimes are mapped to the "upper context", i.e. the visit-sequence for the context p where $X = X_i$, $i > 0$ (X_i is on the righthand side of p). We define ED and LU to be visit numbers with respect to earliest definitions and latest uses of $X.a$ in all visit-sequences:

If $X.a$ is inherited ED is the smallest number j such that a definition of $X_i.a$ precedes *visit* $(i,\ j)$ in a visit-sequence. If $X.a$ is synthesized ED is the smallest number j such that a definition of $X_0.a$ precedes a *leave* (j) in a visit-sequence.

LU is the largest number j such that either a use of $X_0.a$ is contained in a visit-sequence section ending with *leave* (j), or a *visit* $(X_i,\ j)$ precedes a use of $X_i.a$. (If there

is no use of $X.a$ no storage is needed for it; only the computations for its definition have to be executed.)

These definitions of ED and LU take into account that the AG need not be in Bochmann Normal Form, i.e. attributes may be used in the same context where they are defined. As a consequence even the definition points may vary such that defining the minimal number for ED is necessary.

Using ED and LU the lifetime of an occurrence of $X.a$ in a production p is described as a subsequence of vs_p: Let $X_i = X$ in p. Then the tuple (i, r, s) is the lifetime of $X_i.a$ in p such that the visit-sequence element e_r is its begin and e_s is its end. Three cases have to be distinguished:

1. $X.a$ is inherited and $i > 0$ (righthand side occurrence):
   e_r is the definition of $X_i.a$, and e_s is the rightmost of $visit\ (X_i, LU)$ and the last use of $X_i.a$ in vs_p.

2. $X.a$ is synthesized and $i > 0$ (righthand side occurrence):
   $e_r = visit\ (X_i, ED)$ and e_s is the rightmost of $visit\ (X_i, LU)$ and the last use of $X_i.a$ in vs_p.

3. $i = 0$ (lefthand side occurrence):
   e_r is the first element of the section ending with $leave\ (ED)$ and $e_s = leave\ (LU)$

Informally spoken the upper of one of adjacent contexts is made responsible for the storage allocation for attributes in the lower context.

Reachability Sets. Two sets SL and CL are defined in order to describe whether an alive instance of $X.a$ can be reached by tree walk operations from within the lifetime of another instance:

The **subtree lifetime set** SL contains a tuple (Y, i, j) if there is a subtree y with a root labeled Y such that there is an instance of $X.a$ in y (but not at y), and the i-th visit to y leads to its definition and the j-th to its last use.

The **context lifetime set** CL contains a tuple (Y, i, j) if there is a subtree y of the whole tree t with a root labeled Y that has the following properties: In t outside of y there is an instance of $X.a$; its lifetime begins before the i-th visit to y (but not before the $(i-1)$-th) and ends after the j visit to y (but not after the $(j+1)$-th).

SL and CL are computed in the following steps:

1. If there is a lifetime tuple (i, k, l) for a visit-sequence vs_p, then add (Y, r, s) to SL such that v_k is in the r-th section of vs_p, and v_l is in its s-th section, and Y is the lefthand side symbol of p.

   Let X_j, $j > 0$ be a symbol occurrence in p such that *visit* (j, m), ..., *visit* (j, n), $m \leq n$ occur between v_k and v_l in vs_p. Then add (X_j, m, n) to CL.

2. Take a tuple (Y, r, s) of SL and a vs_p such that $X_i = Y$ in p. Add (Z, t, u) to SL if *visit* $(X_i, r) = v_k$ is in the t-th section, and *visit* $(X_i, s) = v_l$ is in its u-th section, and Z is the lefthand side symbol of p.

   For each X_j, $j > 0$, $j \neq i$ in p add (X_j, m, n) to CL if *visit* (j, m), ..., *visit* (j, m) occur between v_k and v_l in vs_p.

   This step is repeated until no more tuples can be added to SL or CL.

3. Take a tuple (Y, m, n) of CL and a vs_p such that Y is the lefthand side of p. For each X_j, $j > 0$ in p add (X_j, r, s) to CL if *visit* (j, r), ..., *visit* (j, s) occur in the m-th to n-th section of vs_p.

   This step is repeated until no more tuples can be added to CL.

Lifetime Conditions. The reachability conditions can now be stated as follows:

$X.a$ can be implemented by a **variable** if there are no tuples (Y, i, k), (Y, j, l) in CL such that $i < j < k \leq l$.

$X.a$ can be implemented by a **stack** if there are not tuples (Y, i, k), (Y, j, l) in CL such that $i < j < k < l$.

It is assumed that the multiple occurrence optimizations are applied as described above. (If not, conditions for overlaps of lifetimes have to be checked for each single visit-sequence,

without regarding its context.) In [EnD90] it is proven (for AGs and visit-sequences with the above mentioned restrictions) that such computations of SL and CL yield sets with the described properties, and that the conditions allow the allocation condition.

6 References

[DMN70] Dahl, O., Myrhaug, B. and Nygaard, K., *SIMULA 67 Common Base Language - Publication S-22*, Norwegian Computing Center, Oslo, 1970.

[EnD90] Engelfriet, J. and DeJong, W., *Attribute Storage Optimization by Stacks*, Acta Informatica 27 (1990), 567–581.

[Far82] Farrow, R., *LINGUIST-86 Yet Another Translator Writting Sysytem based on Attribute Grammars*, SIGPLAN Notices 17 (1982), 160–171.

[FaY86] Farrow, R. and Yellin, D., *A Comparison of Storage Optimizations in Automatically Generated Attribute Evaluators*, Acta Informatica 23 (1986), 393–427.

[Gan79] Ganzinger, H., *On Storage Optimization for Automatically Generated Compilers*, in 4th GI Conf. on Theoretical Computer Science, K. Weihrauch, ed., Lecture Notes in Computer Science, vol. 67, Springer-Verlag, New York–Heidelberg–Berlin, March 1979, 132–141.

[Hal87] Hall, M. L., *The Optimization of Automatically Generated Compilers*, Department of Computer Science, University of Colorado, Ph.D. Thesis, Boulder, CO, 1987.

[Kas89] Kastens, U., *LIGA: A Language Independent Generator for Attribute Evaluators*, Universität-GH Paderborn, Bericht der Reihe Informatik Nr. 63, 1989.

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

[Kas91a] Kastens, U., *An Attribute Grammar System in a Compiler Construction Environment*, Proceedings of the International Summer School on Attribute Grammars, Application and Systems, Prague (1991).

[Kas91b] Kastens, U., *Attribute Grammars as a Specification Method*, Proceedings of the International Summer School on Attribute Grammars, Application and Systems, Prague (1991).

[Kas87] Kastens, U., *Lifetime Analysis for Attributes*, Acta Informatica 24 (November 1987), 633–652.

[KHZ82] Kastens, U., Hutt, B. and Zimmermann, E., *GAG: A Practical Compiler Generator*, Lecture Notes in Computer Science, vol. 141, Springer Verlag, Heidelberg, 1982.

[Pro89] Prott, K-J., *Effiziente LALR(1)-Analyse mit Bestimmung sicherer Anknüpfungspositionen in einem Parsergenerator*, Universität-GH Paderborn, Diplomarbeit, 1989.

[PuB80] Purdom, P. and Brown, C. A., *Semantic Routines and LR(k) Parsers*, Acta Informatica (1980).

[Räi79] Räihä, K., *Dynamic Allocation of Space for Attribute Instances in Multi-pass Evaluators of Attribute Grammars*, SIGPLAN Notices 14 (August 1979), 26–38.

[ReT89] Reps, T. and Teitelbaum, T., *The Synthesizer Generator*, Springer Verlag, New York, 1989.

[Sch89] Schmidt, M., *Generierung effizienter Übersetzer*, Universität-GH Paderborn, FB 17, Dissertation, 1989.

[Son85] Sonnenschein, M., *Global Storage Cells for Attributes in an Attribute Grammar*, Acta Informatica 22 (1985), 397–420.

[Tar89] Tarhio, J., *A Compiler Generator for Attribute Evaluation during LR Parsing*, in Workshop on Compiler Compiler and High Speed Compilation, D. Hammer, ed., Lecture Notes in Computer Science, vol. 371, Springer-Verlag, New York–Heidelberg–Berlin, 1989.