Attribute Evaluation using Neighbour Functions

Khalid A. Mughal Lars Søraas

Department of Informatics, University of Bergen, N-5020 Bergen, Norway.
Tel.: +47-55 54 41 52, Fax: +47-55 54 41 99
E-mail: {khalid,larss}@ii.uib.no

February 1996

Abstract.

Design and implementation of attribute evaluators has received considerable attention ever since Knuth formulated the concept of attribute grammars. In particular, the class of Ordered Attribute Grammars (OAGs) has been of particular interest because practical and efficient attribute evaluators can been implemented based on the *statically* determined *fixed plans* for such grammars. Two main categories of attribute evaluators for OAGs can be distinguished in the literature: those that directly execute these plans and those that are implemented as functional programs, called *visit-functions*, derived from these plans. Incremental versions of these evaluators rely on extra machinery to achieve incremental behaviour. We report on a new functional approach, based on *neighbour functions*, also derived from fixed plans, which allows attribute re-evaluation to start in the context of the node of subtree replacement, and which can readily be extended to achieve efficient incremental behaviour.

Keywords. Attribute grammars, change propagation, incremental attribute evaluation, program generators, functional programming, programming environments.

1 Introduction

An attribute grammar [Knu68, Knu71] defines the *production rules* in the underlying context-free grammar (CFG) for the abstract syntax trees which can be constructed using this grammar. In addition, it defines the *attribution rules* for attributes attached to the grammar symbols for each production in the grammar. Abstract syntax trees thus have attribute instances attached to the tree nodes, and the process of *attribute evaluation* involves updating the attributes values with respect to the attribution rules for the productions in the grammar.

Attribute grammars have proven to be a viable declarative formalism for specifying the semantics of formal languages. In particular, they have been successfully used in automatic generation of compiler front-ends for programming languages [KHZ82]. Attribute evaluation technology has been effectively used to implement generators for various types of systems: interactive language-based editors, interactive theorem provers, program transformation systems, documentation preparation systems and verification tools [RT88, DJL88].

The class of Ordered Attribute Grammars (OAGs) [Kas80] has been of particular interest because practical and efficient attribute evaluators can be implemented based on the *statically* determined *fixed plans*[1] for such grammars. Two main categories of attribute evaluators for OAGs can be distinguished in the literature: those that directly execute these plans and those that are implemented as functional programs, called *visit functions*, derived from these plans [Kas91]. Various strategies have been proposed for *incremental* attribute evaluation [Yeh83, RT88, Alb91b,

1. In the literature, the term *visit-sequence* is also used. See [Pen94].

Pen94]. In this paper we report on a new functional approach, based on *neighbour functions*, also derived from fixed plans, which allows attribute re-evaluation to start in the context of the node of subtree replacement, and which can readily be extended to achieve optimal incremental behaviour. The approach has been successfully implemented in the AML system [EMR93, Sør94] which is a program generator for implementing attribute evaluation engines in the functional programming language ML [MTH90].

The rest of the paper is organised as follows: section 2 gives the necessary background on attribute grammars; section 3 outlines the various approaches for attribute evaluation but is mainly restricted to those concerning OAGs; section 4 describes the strategy for attribute evaluation using visit-functions; section 5 presents our approach using neighbour functions; section 6 deals with optimization of neighbour functions, discusses the results, and also compares and contrasts our work to others; finally in section 7 we present our conclusions together with future work.

2 Attribute Grammars

An attribute grammar [Knu68, Knu71] is a context-free grammar (CFG) augmented by a set of attributes for each grammar symbol and a set of attribution rules for each production. Table 1 shows a simplified attribute grammar for illustrating the relationship between *definitions* and *uses* of variable occurrences [Pen94]. The grammar consists of 3 nonterminals: `S`, `L`, `N`, there `S` is the *root* (or *start symbol*) of the grammar. The symbol `ID` is a terminal representing identifier names. The *attribute declarations* show the attributes attached to the nonterminals. Each attribute declaration specifies the name of the nonterminal (`S`, `L` or `N`) the attributes are attached to, the name of the attributes (`code`, `defs`, `env`, `id`), the type of the attributes (shown as monotypes in ML: `int list`, `(string*int) list`, `string`) and their classification (↑ for *synthesized* and ↓ for *inherited*). Synthesized attributes pass information up towards the root of the *attributed abstract*

Table 1. The DEF-USE Grammar

Attribute declarations:

```
S <    ↑code:int list >
L <    ↑defs:(string*int) list, ↓env:(string*int) list, ↑code:int list >
N <    ↑id:string >
```

Production	**Attribution Rules**
`S ::= Root(L)`	`L$env = L$defs`
	`S$code = L$code`
L_0 `::= Def(N` L_1`)`	L_0`$defs = (N$id,length(`L_1`$defs)+1)::`L_1`$defs`
	L_1`$env = `L_0`$env`
	L_0`$code = lookUp(N$id,`L_1`$env) :: `L_1`$code`
L_0 `::= Use(N` L_1`)`	L_0`$defs = `L_1`$defs`
	L_1`$env = `L_0`$env`
	L_0`$code = (-lookUp(N$id,`L_1`$env)) :: `L_1`$code`
`L ::= Empty()`	`L$defs = []`
	`L$code = []`
`N ::= Name(ID)`	`N$id = ID`

```
fun lookUp (_, []) =   ⊥
   | lookUp ("", _) =   ⊥
   | lookUp (i, (n,d)::t) = if i = n then d else lookUp(i,t)
```

syntax tree, and inherited attributes pass information down towards the leaves of the tree. In our notation attribute `a` of nonterminal `X` is referred to as `X$a`.

The grammar in Table 1 has five productions (`Root`, `Def`, `Use`, `Empty`, `Name`). The nonterminal `S` has one production (`Root`) while nonterminals `L` and `N` have three productions (`Def`, `Use`, `Empty`) and one production (`Name`) respectively. For each production, *attribution rules* are specified for defining the values of all synthesized attributes of the nonterminal on the left-hand side of the production and for all the inherited attributes of the grammar symbols on the right-hand side of the production. These are collectively known as *out* attributes of the production. There is at the most one attribution rule for each *out* attribute of a production.

3 Attribute Evaluation Strategies

Each node in an abstract syntax tree is an instance of a nonterminal and is decorated with instances of the attributes of the nonterminal. An abstract syntax tree is *attributed* (i.e. attributes are assigned values) according to the attribution rules. An abstract syntax tree generated according to the DEF-USE grammar consists of a list of definitions and uses. For example, the "program"

```
(use x; def x; use y)
```

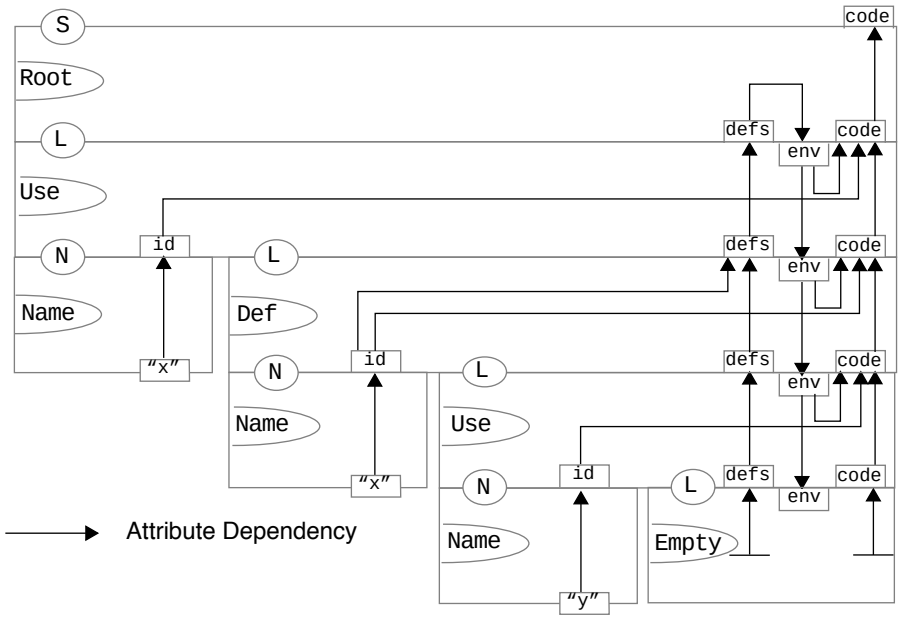
would be represented by the following abstract syntax tree in the above grammar:

```
Root(Use(Name("x"),Def(Name("x"),Use(Name("y"),Empty()))))
```

and attribute evaluation of this abstract syntax tree results in the following value for the attribute `code` attached to the root nonterminal `S`:

```
[-1, 1, ⊥]
```

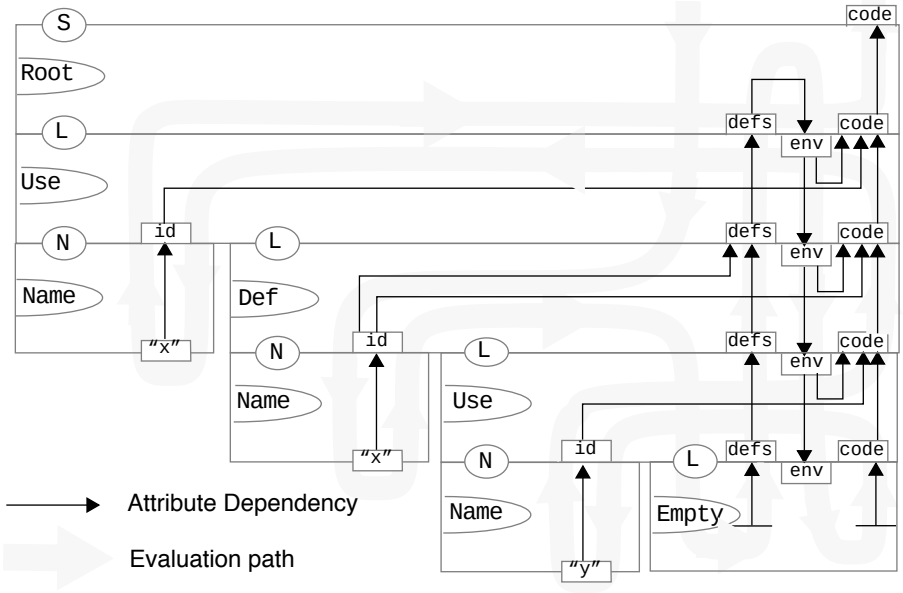
i.e. each element in `S$code` is either the "address" of the variable definition or negation of this address for variable uses. Since `y` is not defined, its address is "⊥". Figure 1[2] shows this abstract



`Root(Use(Name("x"),Def(Name("x"),Use(Name("y"),Empty()))))`

Fig. 1. Abstract Syntax Tree with Dependency Graph for the DEF-USE Example

2. We use the excellent graphical notation introduced by Pennings in [Pen94].



`Root(Use(Name("x"),Def(Name("x"),Use(Name("y"),Empty()))))`

Fig. 2. Abstract Syntax Tree with Dependency Graph and Evaluation Path for the DEF-USE Example

syntax tree together with its *dependency graph*. The dependencies are derived from the attribution rules. The definitions are synthesized and passed up to the root of the abstract syntax tree by the synthesized attribute `defs` and then passed down to the leaves of the abstract syntax tree by the inherited attribute `env`. Any attribute evaluation strategy must comply with the dependencies among the attributes given by the dependency graph for the abstract syntax tree. One such *evaluation path* respecting these dependencies is shown in Figure 2.

Various strategies have been devised for attribute evaluation [RTD83, Yeh83, RT88, Alb91a, Alb91b, Pen94]. In particular, *static* evaluators make use of a *static ordering* of attribute dependencies for the productions in the grammar. (This is in contrast to *dynamic* evaluators which determine the evaluation order of attributes at evaluation time.) Practical and efficient static evaluators have been implemented for a particular class of *partitionable* grammars called *Ordered Attribute Grammars* (OAGs) [Kas80]. OAGs have the important property that, for each production in the grammar, it is possible to statically determine the order in which its attributes should be evaluated – independent of the context of the production in any abstract syntax tree. This essential property allows the construction of *fixed plans* for each production using a *plan generating algorithm* [Kas80, RT88, Pen94]. The fixed plan for a production `p` consists of a sequence of following instructions:

`EVAL`(X$a)	Evaluate attribute a of nonterminal `X` of the production `p`.
`VISIT`(X,i)	Visit (i.e. transfer control to) child `X` of production `p` for the `i`-th time.
`SUSPEND`(i)	Suspend evaluation of the current fixed plan and transfer control to the parent of production `p` for `i`-th time.

Table 2 shows the fixed plans for the productions of the DEF-USE grammar. Given the fixed plans for a grammar, a *fixed-plan evaluator* can be used to attribute any abstract syntax tree of the

Table 2. Fixed Plans for the Productions in the DEF-USE Grammar

S ::= Root(L)	L_0 ::= Def(N L_1)	L_0 ::= Use(N L_1)	L ::= Empty()	N ::= Name(ID)
1.VISIT(L,1) 2.EVAL(L$env) 3.VISIT(L,2) 4.EVAL(S$code) 5.SUSPEND(1)	1.VISIT(N,1) 2.VISIT(L_1,1) 3.EVAL($L_0$$defs) 4.SUSPEND(1) 5.EVAL($L_1$$env) 6.VISIT(L_1,2) 7.EVAL($L_0$$code) 8.SUSPEND(2)	1.VISIT(L_1,1) 2.EVAL($L_0$$defs) 3.SUSPEND(1) 4.EVAL($L_1$$env) 5.VISIT(L_1,2) 6.VISIT(N,1) 7.EVAL($L_0$$code) 8.SUSPEND(2)	1.EVAL(L$defs) 2.SUSPEND(1) 3.EVAL(L$code) 4.SUSPEND(2)	1.EVAL(N$id) 2.SUSPEND(1)

grammar starting at the root of the tree [RT88, Pen94]. EVAL-instructions evaluate *out* attributes of a production whereas VISIT- and SUSPEND-instructions provide the necessary control flow for maintaining the total order for attribute evaluation. Due to the nature of its construction, the fixed-plan evaluator may make several visits to a node. The control flow for attributing our example abstract syntax tree using such an evaluator would be the same as the evaluation path shown in Figure 2.

3.1 Optimal Incremental Attribute Evaluation

Modifications to an abstract syntax tree are usually defined in terms of *subtree replacement*. In the face of modifications, the fixed-plan evaluator always reattributes an abstract syntax tree starting at the root regardless of the site of modification in the tree. Let AFFECTED be the set of attributes that change value as a result of reattribution of an abstract syntax tree after modification. This set is not known *apriori* to reattribution. An algorithm that reattributes an abstract syntax tree in time O(|AFFECTED|) is said to be *optimally incremental* (under the assumption that evaluating an attribute is constant-time). For certain applications (for example language-based editors) where response time is critical, incremental evaluation strategies are more appropriate [RT88, Alb91b, Pen94]. Incremental algorithms, based on the concept of *change propagation*, are devised to optimize the execution of VISIT- and SUSPEND-instructions – these instructions are skipped where necessary to avoid unnecessary attribute computations, and old attributes values are used where possible [RT88, Yeh83]. The essential idea behind the incremental algorithm is to only visit those nodes whose attribute instances are directly dependent on attribute instances whose values have changed. One way of keeping track of such nodes is by keeping around a set of *active* productions, called the REACTIVATED set, whose members are affected as explained above. In the incremental version of the fixed-plan evaluator, EVAL-instructions update the REACTIVATED set when an attribute instance changes value, and VISIT- and SUSPEND-instructions transfer control only to members of this set thereby limiting change propagation. Re-evaluation starts with the fixed plan for the parent of the node of subtree replacement in the abstract syntax tree. Such an incremental algorithm also requires an equality test for attribute values to limit change propagation since it is necessary to determine if the value of an attribute has changed.

4 Attribute Evaluation using Visit-functions

The relationship between attribute grammars and functional programs has been explored in [Kat84, Joh87]. Transformation of fixed plans into *visit-functions* to implement a *functional* attribute evaluator is presented in [Kas91]. A visit-function recursively computes a subset of synthesized attribute instances for a production instance (applied at a node in an abstract syntax tree) by taking

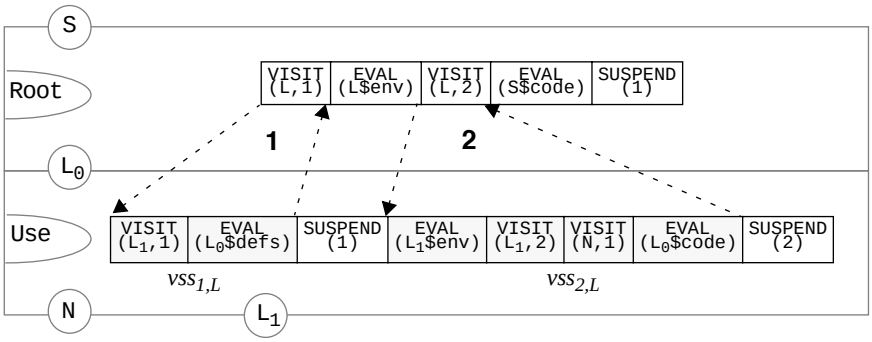


Fig. 3. Calculating visit-subsequences (*vss*)

the subtree at this node and a subset of inherited attributes of that production as parameters.

Each SUSPEND-instruction in a given fixed plan for a production transfers control to the parent production. In other words such an instruction marks the return from a visit to this production. This is illustrated in Figure 3 where the *subsequences* of instructions executed in the production `Use` on *first* and *second* visit (called *visit-numbers*) from its parent production `Root` are shown. The fixed plan for a production can be split into *visit-subsequences* where instructions between 2 consecutive SUSPEND-instructions constitute a visit-subsequence for the production ($vss_{2,L}$ in Fig. 3).The visit-subsequence for the first visit-function is the sequence of instructions from the start of the plan to the first SUPEND-instruction ($vss_{1,L}$ in Fig. 3).

Table 3 shows the visit-subsequences for the productions of our DEF-USE grammar. Visit-subsequences for the production `Use` are also shown as greyed subsequences of instructions ($vss_{1,L}$ and $vss_{2,L}$) in Figure 3. Since a nonterminal can have several productions, visit-subsequences for a given visit-number for this nonterminal can be combined into a single *visit-function*. Table 4 shows the visit-functions[3] for our DEF-USE grammar derived from Table 3. The visit-function to execute is determined by the production for the nonterminal at a particular node in the abstract syntax tree and the visit-number. For example, visit-function $\texttt{visit}_{2,\texttt{L}}$ is called during attribute evaluation of a node labelled with nonterminal `L` on visit-number 2 to this node. Pattern matching on the production of nonterminal `L` applied at the node differentiates which visit-function $\texttt{visit}_{2,\texttt{L}}$ to execute. Each visit-function takes as parameter, in addition to the subtree, (a subset of) inherited attributes and returns (a subset of) synthesized attributes. Note that in Table 4 the EVAL-instructions

Table 3. Visit-subsequences(*vss*) for the Productions in the DEF-USE grammar

	Vss for Production of S	Vss for Productions of L			Vss for Production of N
Visit no.	S ::= Root(L)	L_0 ::= Def(N L_1)	L_0 ::= Use(N L_1)	L ::= Empty()	N ::= Name(ID)
1	1.VISIT(L,1) 2.EVAL(L$env) 3.VISIT(L,2) 4.EVAL(S$code)	1.VISIT(N,1) 2.VISIT(L_1,1) 3.EVAL($L_0$$defs)	1.VISIT(L_1,1) 2.EVAL($L_0$$defs)	1.EVAL(L$defs)	1.EVAL(N$id)
2		5.EVAL($L_1$$env) 6.VISIT(L_1,2) 7.EVAL($L_0$$code)	4.EVAL($L_1$$env) 5.VISIT(L_1,2) 6.VISIT(N,1) 7.EVAL($L_0$$code)	3.EVAL(L$code)	

3. We have used an ML-like style for exposition purposes and omitted superfluous details.

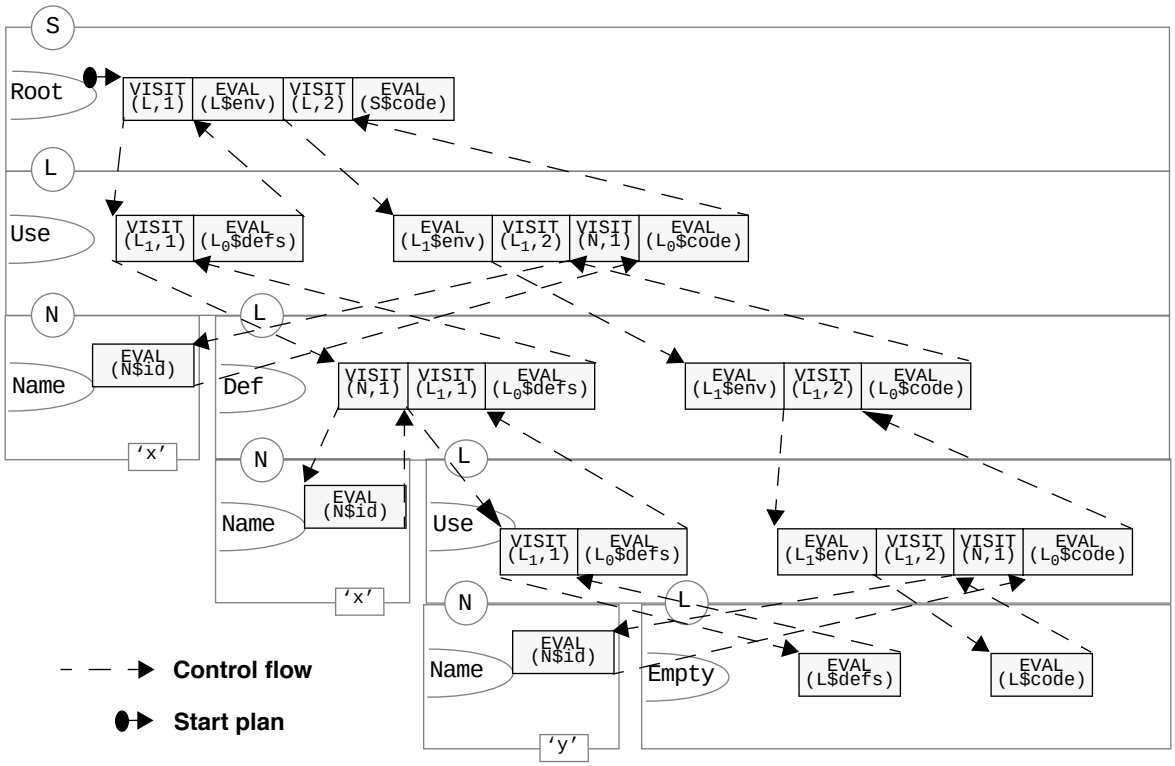


Fig. 4. Plan Tree for DEF-USE Example

have been replaced with the corresponding attribution rules. The notation `X_a` indicates a local binding and implements local optimization of attribute lookups. The control flow for attribute evaluation of our running example abstract syntax tree using visit-functions is shown in Figure 4 and is the same as depicted in Figure 2 but now the execution of a `SUSPEND`-instruction is implicitly embedded in the return from the corresponding visit-function call. Notice that reattribution always starts at the root of the abstract syntax tree.

4.1 Incremental Attribute Evaluation using Visit-functions

To implement an *incremental* attribute evaluator based on visit-functions is not so straightforward as for the incremental fixed-plan evaluator. A simple scheme of calling the appropriate visit-function at the node of subtree replacement can never result in the change propagation proceeding upwards towards the root as visit-functions can only call other visit-functions of the *children* of the production. In his Ph.D. thesis, Pennings discusses implementation of efficient *incremental* attribute evaluation based on visit-functions [Pen94]. Incrementality is primarily achieved by *memoisation* of visit-functions: all the arguments, including the subtree, and the result are cached. A visit-function is skipped if a *hit* is made in the cache. However this scheme also requires the problem of *intra-visit-subsequence dependencies* to be resolved. Such a dependency occurs when an attribute that changed value in a previous visit is needed in a subsequent visit to the same node but it does not occur as an argument in later visits because it is not an inherited attribute. In Table 4 we have an example of such a dependency: the second visit to `Def(N,`L_1`)`requires the value of attribute `N$id`. The visit-functions are thus not *pure functions*. In Pennings' scheme, any modification of the abstract syntax tree, for example, changing our running example abstract syntax tree

Table 4. Visit-functions for the DEF-USE Grammar

```
visit1,S (S as Root(L)) = S$code
    where  L_defs   = visit1,L L
           L$env    = L_defs
           L_code   = visit2,L L L$env
           S$code   = L_code
```

```
visit1,L (L0 as Def(N,L1))= L0$defs
    where  N_id     = visit1,N N
           L1_defs  = visit1,L L1
           L0$defs  = (N_id,length(L1_defs)+1):: L1_defs

visit1,L (L0 as Use(N,L1))= L0$defs
    where  L1_defs  = visit1,L L1
           L0$defs  = L1_defs

visit1,L (L0 as Empty()) = L0$defs
    where  L0$defs  = []
```

```
visit2,L (L0 as Def(N,L1)) L0_env = L0$code)
    where  L1$env   = L0_env
           L1_code  = visit2,L L1 L1$env
           L0$code  = (-lookUp(L0_env, N$id)) :: L1_code

visit2,L (L0 as Use(N,L1)) L0_env = L0$code
    where  L1$env   = L0_env
           L1_code  = visit2,L L1 L1$env
           N_id     = visit1,N N
           L0$code  = lookUp(L0_env, N_id) :: L1_code

visit2,L (L0 as Empty()) L0_env = L0$code
    where  L0$code  = []
```

```
visit1,N (N as Name(ID)) = N$id
    where  N$id     = ID
```

`Root(Use(Name(“x”),Def(Name(“x”),Use(Name(“y”),Empty()))))`

to

`Root(Use(Name(“x”),Def(Name(“x”),Use(Name(“x”),Empty()))))`

(where use of `y` is replaced by use of `x`), always requires the call to the visit-function $visit_{1,S}$ for the root nonterminal `S`. It also requires memoising of visit-functions in order to “skip” to the node of subtree replacement in the abstract syntax tree. In other words, the memoisation of visit-functions is essential to achieve incremental behaviour in this scheme without which *all* the attributes would be re-evaluated. There is no way to start an incremental update at an arbitrary node of subtree replacement. Optimizations of Pennings' scheme will be discussed in section 6.

5 Attribute Evaluation using Neighbour Functions

`VISIT`-instructions in the fixed plan for a production (applied at a node) transfer control down to the *children* of the production with the aim of computing *synthesized* attributes. The previous section showed how this control is achieved using visit-functions. `SUSPEND`-instructions in the fixed plan for a production transfer control to the *parent* of the left-hand nonterminal in the production. If the *parent production* of this node knew who was transferring control to it from below and which *suspend-number* this was, the parent production would be able to continue the execution of its plan

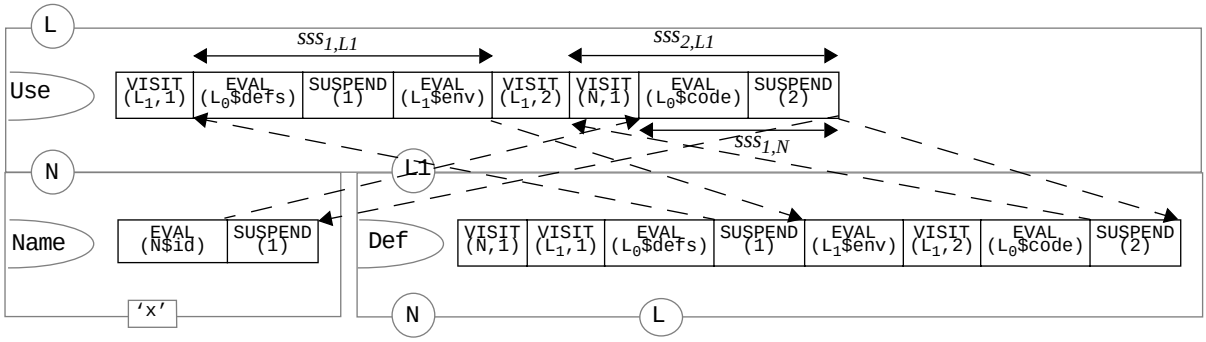


Fig. 5. Calculating Suspend-subsequences (*sss*)

as if it had first made a visit to this child and then resumed execution. The fixed plan for a production yields this information, which can be used to construct *suspend-functions.* Suspend-functions propagate change upwards in the abstract syntax tree analogous to visit-functions which propagate change downwards. They can be used to implement an attribute evaluation strategy which allows update to start at the node of subtree replacement and extends to various schemes for achieving incremental behaviour as discussed below. Visit- and suspend-functions in our scheme are collectively called *neighbour functions.*

Figure 5 illustrates the situation where the fixed plan of a *child production* (`Def`) controls the execution of the fixed plan of its *parent production* (`Use`). Each execution of a `SUSPEND`-instruction (in the plan of the child production) transfers control to the instruction after the corresponding `VISIT`-instruction (in the parent production). The plan of the parent production transfers control back to the child production right before the next `VISIT`-instruction to this *same* child. The fixed plan for a production can thus be split into *suspend-subsequences.* Instructions between 2 consecutive `VISIT`-instructions to the same child constitute a suspend-subsequence of the production for this particular child ($sss_{1,L1}$ in Fig. 5). The suspend-subsequence for the last suspend-function for a given child is the sequence of instructions from the last `VISIT`-instruction for this child to the end of the plan ($sss_{2,L1}$ in Fig. 5). Figure 5 shows the suspend-subsequences ($sss_{2,L1}$ and $sss_{1,L1}$) of production `Use` for second child (`L`) when it transfers control to `Use` from below. Notice that the second child (`L`) transfers control twice to the parent production `Use` in contrast to the first child (`N`) of `Use`

Table 5. Suspend-subsequences(*sss*) for the Nonterminal `L` in the DEF-USE grammar

	suspend-number = 1		suspend-number = 2	
	Nonterminal as child no. 1	Nonterminal as child no. 2	Nonterminal as child no. 1	Nonterminal as child no. 2

L	`S ::= Root(L)`
	`2.EVAL(L$env)`
		`L0 ::= Def(N,L1)`
		`3.EVAL(L0$defs)`
		`4.SUSPEND(1)`
		`5.EVAL(L1$env)`
			`S ::= Root(L)`
			`4.EVAL(S$code)`
				`L0 ::= Def(N,L1)`
				`7.EVAL(L0$code)`
				`8.SUSPEND(2)`

		`L0 ::= Use(N,L1)`
		`2.EVAL(L0$defs)`
		`3.SUSPEND(1)`
		`4.EVAL(L1$env)`
				`L0 ::= Use(N,L1)`
				`6.VISIT(N,1)`
				`7.EVAL(L0$code)`
				`8.SUSPEND(2)`

which transfers control only once ($sss_{1,N}$). A *suspend-number* `i` indicates that it is the `i`-th time control is transferred to the parent from the child. Suspend-subsequences for our DEF-USE grammar are tabulated in Tables 5 and 6. Each entry corresponds to a right-hand-side occurrence of a nonterminal which is able to transfer control to its parent

Transfer of control to the parent of a node (nonterminal instance) is determined by the production, the position of the nonterminal occurrence in the production (which child), and the suspend number. Notice that for *each* child of a production we derive a set of suspend-subsequences. We get three suspend-subsequences for the production `Def(N,L1)`: two suspend-subsequences for the second child (for nonterminal `L;` tabulated respectively under suspend-number = 1 and 2 for child no. 2 in Table 5), and one for the first child (for nonterminal `N`; tabulated under suspend-number = 1 and child no. 1 in Table 6).

Table 6. Suspend-subsequences (*sss*) for the Nonterminal `N` in the DEF-USE grammar

	suspend-number = 1
	Nonterminal as child no. 1
N	L_0 ::= Def(N,L_1) 2.VISIT(L_1,1) 3.EVAL($L_0$$defs) 4.SUSPEND(1) 5.EVAL($L_1$$env) 6.VISIT(L_1,2) 7.EVAL($L_0$$code) 8.SUSPEND(2)
	L_0 ::= Use(N,L_1) 7.EVAL($L_0$$code) 8.SUSPEND(2)

A nonterminal can occur on the right hand side in many productions. We are interested in all possible parents of the nonterminal, so we combine its suspend-subsequences for each suspend-number into a single suspend-function for that number. Table 7 shows the suspend-functions for our DEF-USE grammar. The suspend-function $\texttt{suspend}_{1,\texttt{L}}$ is called to execute the *first* suspend-subsequence of a production with `L` as a child. The suspend-subsequence selected is dependent on this production and the child-number of nonterminal `L` in this production. Any `SUSPEND`-instruction in a specific suspend-subsequence of a production is translated into an appropriate call to the relevant suspend-function for the left-hand nonterminal of this production in order to transfer control to the parent node (computed by the function `parentNode`) depending on the suspend-number (fixed and given in the `SUSPEND`-instruction) and child-number of the current node in relation to its parent (computed by the function `ChildNoInParentProd`). Each suspend-function (in addition to the subtree and suspend-number) also takes as parameter (a subset of) synthesized attributes and returns (a subset of) inherited attributes of the production. `EVAL`-instructions are replaced with the corresponding attribution rules. As noted previously for visit-functions, the notation `X_a` indicates a local binding and implements local optimization of attribute lookups.

In order to have the node of subtree replacement as the starting point of attribute evaluation it necessary that *all* the instructions in the fixed plan of its production are executed. This is achieved by constructing a *start-function* for each production which calls the appropriate neighbour functions to execute the whole plan. Return from a start-function thus signals termination of attribute evaluation. Since the instructions in a fixed plan can be divided into visit-subsequences interspersed with `SUSPEND`-instructions, the execution of the whole plan is equivalent to calling the relevant visit-functions corresponding to the visit-subsequences and the relevant suspend-function for each `SUSPEND`-instruction. Table 8 shows the start-function for our DEF-USE grammar. The plan generating algorithm fixes the order of visits and suspends for evaluating the attributes of a nonterminal in an OAG. Thus for any node labelled with nonterminal `L`, the order of visits and suspends for evaluating its attributes is the same regardless of which production of `L` is applied at the node (`Def`, `Use` or `Empty`).

Table 7. Suspend-functions for the DEF-USE grammar

```
(* Suspend-function for L when suspend-number is 1 *)

suspend1,L (S as Root(L)) 1 L_defs = L$env
    where  L$env    = L_defs

suspend1,L (L0 as Def(N,L1)) 2 L1_defs = L1$env
    where  L0$defs  = (N$id,length(L1_defs)+1) :: L1_defs
           L0_env   = suspend1,L (parentNode L0) (ChildNoInParentProd L0) L0$defs
           L1$env   = L0_env

suspend1,L (L0 as Use(N,L1)) 2 L1_defs = L1$env
    where  L0$defs  = L1_defs
           L0_env   = suspend1,L (parentNode L0) (ChildNoInParentProd L0) L0$defs
           L1$env   = L0_env
```

```
(* Suspend-function for L when suspend-number is 2 *)

suspend2,L (S as Root(L)) 1 L_code = ()
    where  S$code   = L_code

suspend2,L (L0 as Def(N,L1)) 2 L1_code = ()
    where  L0$code  = lookUp(L0$env, N$id) :: L1_code
           _        = suspend2,L (parentNode L0) (ChildNoInParentProd L0) L0$code

suspend2,L (L0 as Use(N,L1)) 2 L1_code = ()
    where  N_id     = visit1,N N
           L0$code  = (-lookUp(L0$env, N_id)) :: L1_code
           _        = suspend2,L (parentNode L0) (ChildNoInParentProd L0) L0$code
```

```
(* Suspend-function for N when suspend-number is 1 *)

suspend1,N (L0 as Def(N,L1)) 1 N_id = ()
    where  L1_defs  = visit1,L L1
           L0$defs  = (N_id,length(L1_defs)+1) :: L1_defs
           L0_env   = suspend1,L (parentNode L0) (ChildNoInParentProd L0) L0$defs
           L1$env   = L0_env
           L1_code  = visit2,L L1
           L0$code  = lookUp(L0_env, N_id) :: L1_code
           _        = suspend2,L (parentNode L0) (ChildNoInParentProd L0) L0$code

suspend1,N (L0 as Use(N,L1)) 1 N_id = ()
    where  L0$code  = (-lookUp(L0$env, N_id)) :: L1$code
           _        = suspend2,L (parentNode L0) (ChildNoInParentProd L0) L0$code
```

Table 8. Start-function for the DEF-USE grammar

```
start (S as Root(L)) = S_code
    where  S_code   = visit1,S S

start (L0 as (Def(N,L1) | Use(N,L1) | Empty())) = (L0_defs, L0_code)
    where  L0_defs  = visit1,L L0
           L0_env   = suspend1,L (parentNode L0) (ChildNoInParentProd L0) L0_defs
           L0_code  = visit2,L L0 L0_env
           _        = suspend2,L (parentNode L0) (ChildNoInParentProd L0) L0_code

start (N as Name()) = N_id
    where  N_id     = visit1,N N
           _        = suspend1,N (parentNode N) (ChildNoInParentProd N) N_id
```

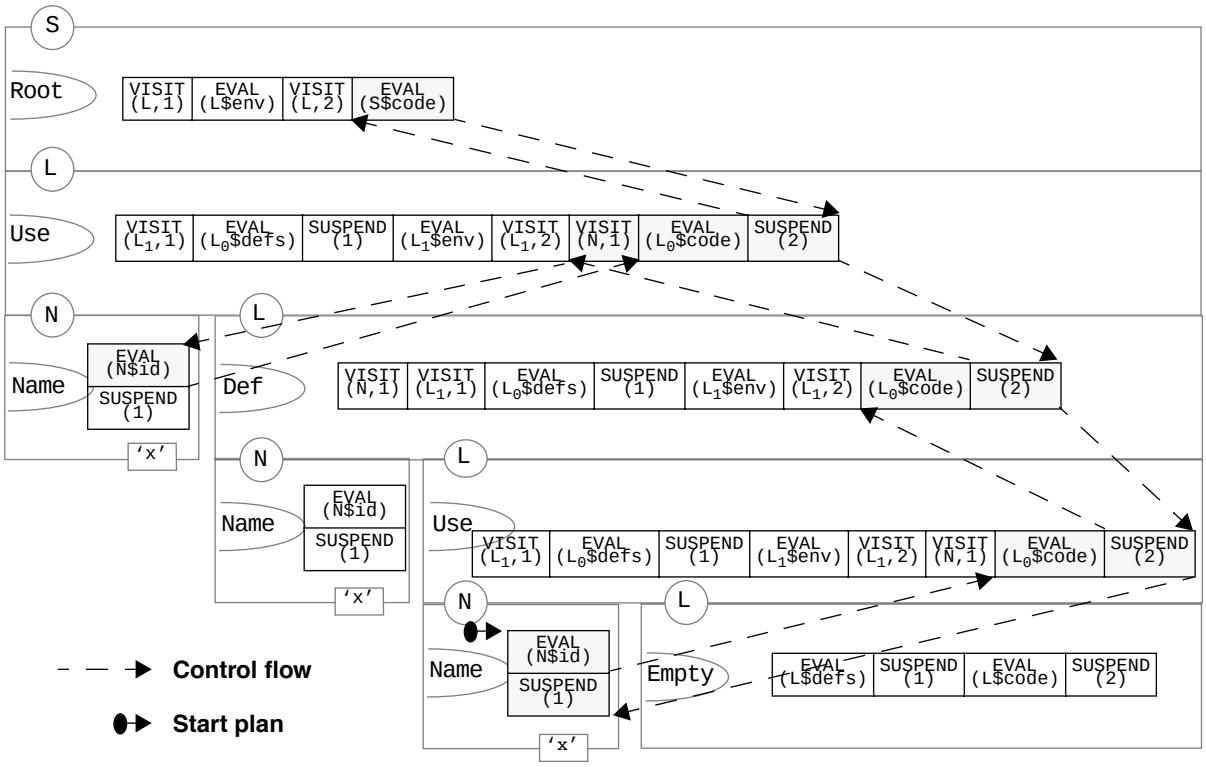


Fig. 6. Plan Tree for DEF-USE Example using Neighbour functions

Figure 6 shows the plan tree when the use of `y` in our running DEF-USE example is replaced by a use of `x`. The re-evaluation using neighbour functions still follows the evaluation path of Figure 2 but now only a segment of the path is computed to complete the attribution: the segment of the evaluation path up to the first visit to the node of subtree replacement (N) is skipped as attributes along this segment are unaffected by the modification. In this sense, this evaluation strategy is optimal w.r.t. the evaluation path.

Tables 4, 7 and 8 constitute a functional attribute evaluator where attribute evaluation can commence at the node of subtree replacement. Starting the attribute evaluation at the root is the same as just using visit-functions since a visit-function never calls a suspend-function.

5.1 Incremental Attribute Evaluation using Neighbour Functions

The neighbour functions are not sufficient to provide incremental behaviour. They are, however, sufficient in providing a *data-flow driver* for attribute evaluation which ensures that the total ordering of attributes is maintained in any context. With neighbour functions, we can use either of the incremental schemes outlined in the previous sections.

The scheme of computing the REACTIVATED set of productions for incremental fixed-plan evaluators can be readily adapted by making the set an extra parameter to the neighbour functions. The REACTIVATED set is updated as before when an attribute instance changes value. A neighbour function is only called if its production is in the REACTIVATED set. This scheme is optimal in the sense explained in section 3.1 that only those attributes that need evaluation are re-evaluated –

change propagation from the node of subtree replacement becomes quiescent as soon as no more attributes need re-evaluation.

The scheme of memoising visit-functions from section 4.1, if extended to include suspend-functions, readily provides incremental behaviour. The suspend-functions, together with their arguments and their results, must also be cached. As in the case of intra-visit-subsequence dependencies, the *intra-suspend-subsequence dependencies* must also be resolved. Such a dependency occurs when an attribute that changed value in a previous suspend is needed in a subsequent suspend to the same node but it does not occur as an argument in later suspends because it is not a synthesized attribute. In Table 7 we have examples of such dependencies: second suspend to `Def(N,L1)` requires the values of attributes `L0$env` and `N$id`. The analysis of these dependencies is analogous to the analysis for intra-visit-subsequence dependencies in [Pen94] but now the roles of the attributes are reversed and it needs to be carried out for *each* child of the production.

6 Discussion

The incremental attribute evaluation scheme based on neighbour functions extended with the REACTIVATED set has been implemented in the AML system [Sør94]. Since the AML system is implemented in the framework of ML, our attribute evaluation scheme fits well in the functional paradigm. The start- and neighbour functions are generated from the attribute specification as discussed in the previous sections. The evaluator has been tested on several small grammars, including a simple Pascal-like language.

In [Pen94], Pennings presents certain transformations to optimize the role of the tree as a data-flow driver. In particular, *tree splitting* is introduced to optimize visit-function cache hits. Since attribute evaluation always starts at the root, the claim is that attribute evaluation of the nodes on the path leading from the root to the node of subtree replacement can be optimized. Any node on this path, which contains the node of subtree replacement, will be visited for *every* visit-number to this node as the subtree has been modified by virtue of the fact that it contains the node of subtree replacement. Tree splitting optimizes the number of visits to such nodes. However such machinery is not relevant for our scheme as attribute evaluation always starts at the node of subtree replacement.

In implementing our scheme in the AML system, we have taken the approach of memoising tree constructors [TC90] and maintaining an auxiliary repository for the attributes. This repository is a hash table where the hash keys are a function of the *path* from the root to the node. From the hash key of a node, the hash key of one of its children can be computed in constant time. Since attribute values in the hash table are ML *eqtypes*, we rely on their implementation for efficiency of equality testing. Separate attribute repository allows simpler mapping between attributed and unattributed syntax trees.

7 Conclusion

The generation of *functional* incremental attribute evaluators from attribute grammar specifications is part of an ongoing research effort. Here we mention a few issues which we intend to explore in the future:

- Implementation of *binding analysis* is necessary to resolve intra-*visit*-subsequence dependencies [Pen94]. We need to extend it to resolve also intra-*suspend*-subsequence dependencies. This, together with memoising of neighbour functions, would implement our incremental attribute evaluator as a *pure function*.
- We would like to extend our approach to *Higher Order Attribute Grammars* (HAGs) [VS91]. In [Pen94], Pennings has already demonstrated the use of visit-function based evaluators for HAGs, so we do not envisage any fundamental problems in extending our approach to HAGs.
- In order to further demonstrate the viability of our approach we intend to apply it to larger grammars for real applications.

We have shown how a functional attribute evaluator can be generated from an attribute specification. We have also shown how it can be made incremental using at least two different strategies. We have also implemented our approach and will continue to enhance it in the directions outlined above.

Acknowledgements

We would like to thank Carla Marceau, John Reppy, Maarten Pennings and Daxing Yeh for their invaluable comments. We would also like to thank Thomas Yan, Roy Oma, Johnny Tvedt and Tord Kolsrud for valuable discussions pertaining to this work.

References

[Alb91a] Alblas, H.: Attribute evaluation methods. *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems.* Alblas, H., and Melichar, B. (eds.), *Lecture Notes In Computer Science*, 545:48-113, Springer-Verlag, 1991

[Alb91b] Alblas, H.: Incremental attribute evaluation. *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems.* Alblas, H., and Melichar, B. (eds.), *Lecture Notes In Computer Science*, 545:215-233, Springer-Verlag, 1991

[DJL88] Deransart, P., Jordan, M. and Lorho, B.: Attribute Grammars: Definitions, Systems and Bibliography. *Lecture Notes in Computer Science*, 323, Springer-Verlag, 1988

[EMR93] Efremidis, S., Mughal, K. A. and Reppy, J. H.: AML: Attribute Grammars in ML. Tech. Report 89, Department of Informatics, University of Bergen, Norway, 1993

[Joh87] Johnsson, T.: Attribute grammars as a functional programming paradigm. *Proceedings of the 1987 Conference on Functional Programming and Computer Architecture*, *Lecture Notes in Computer Science*, 274:154-173, Springer-Verlag, 1987

[Kas80] Kastens, U.: Ordered attribute grammars. *ACTA Informatica*, 13:229-256, 1980

[Kas91] Kastens, U.: Implementation of visit-oriented attribute evaluators. *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems.* Alblas, H., and Melichar, B. (eds.), *Lecture Notes In Computer Science*, 545:114-139, Springer-Verlag, 1991

[Kat84] Katayama, T.: Translation of attribute grammars into procedures. *Transactions on Programming Languages and Systems*, 6(3):345-369, 1984

[KHZ82] Kastens, U., Hutt, B. and Zimmermann, E.: *GAG: A practical compiler generator. Lecture Notes in Computer Science*, 141, Springer-Verlag, 1982

[Knu68] Knuth, D. E.: Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127-145, 1968

[Knu71] Knuth, D. E.: Semantics of context-free languages (correction). *Mathematical Systems Theory,* 5(1):195-96, 1968

[MTH90] Milner, R., Tofte, M. and Harper, R.: *The Definition of Standard ML*. The MIT Press, Cambridge, Mass, 1990

[Pen94] Pennings, M.: *Generating Incremental Attribute Evaluators*. Ph.D. Thesis, Department of Computer Science, Utrecht University, The Nederlands, 1994

[RT88] Reps, T. W., Teitelbaum, T.: *The Synthesizer Generator: A system for constructing language-based editors*. Springer-Verlag, 1988

[RTD83] Reps, T. W., Teitelbaum, T. and Demers, A.: Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449-477, 1983

[Sør94] Søraas, L.: *Generering av attribueringssystemer for AML spesifikasjoner.* Cand. Scient. Thesis, Department of Informatics, University of Bergen, Norway, 1994

[TC90] Teitelbaum, T., Chapman, R.: Higher-order attribute grammars and editing environments. *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, 25(6):197-208, 1990

[VS91] Vogt, H., Swierstra, D.: Higher order attribute grammars. *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems.* Alblas, H., and Melichar, B. (eds.), *Lecture Notes In Computer Science*, 545:256-296, Springer-Verlag, 1991

[Yeh83] Yeh, D.: On incremental evaluation of ordered attribute grammars. *BIT*, 23:308-320, 1983