TLucid and Intensional Attribute Grammars

Senhua Tao
Department of Computer Science
University of Victoria
Victoria, B.C. V8W 3P6
e-mail: stao@sanjuan.UVic.CA

April 2, 1993

Abstract

TLucid if a variation of Lucid. The semantics of TLucid is defined on a tree based context space - a multidimensional space consisting of an arbitrary tree dimension and multitime dimensions. TLucid is designed for defining attribute definition rules in attribute grammars. An attribute is a TLucid variable which is defined by a TLucid expression. We call such defined attribute grammars intensional attribute grammars. By defining attributes as TLucid variables, the semantics of attributes are also extended to the intensional context space of TLucid. The value of an attribute in a given parse tree is an intension from nodes on the tree and time points in the time dimensions to basic values. The attribute values on different dimensions are manipulated by node switching and time switching operators provided in TLucid. By extending an attribute value at a node from a scalar data value to a data stream, intensional attribute grammars extending support circular attribute definitions, using temporal attribute definitions.

1 Introduction

TLucid is a variation of Lucid [10] whose semantics is defined on an arbitrary tree based context space. TLucid provides a set of node switching operators which can switch context from one node to another node and a set of Lucid time switching operators. TLucid supports the algorithms that are based on tree structures.

Attribute grammars (AGs) [4][5] are a formal specification mothed to define syntax and semantics of programming languages. Using AGs, for a given parse tree, the semantics of a node in the tree is the values of attribute instances associated with the node. Attribute values are defined in the attribute definition rules, which are associated with the corresponding production rules of the

context-free grammar component of AG. An important advantage of attribute grammars is that the attribute definitions are declarative: the semantic evaluation done at a particular node of a parse tree can be understood by inspecting only their definitions at the corresponding production rule.

However, the advantage also brings some drawback, that is, an attribute associated with a grammar symbol in a production rule can only be defined by the attributes associated with the grammar symbols in the same production rule. Therefore, in order to pass the desired values to appropriate attribute instance in a parse tree, we need define communication attributes in some production rules, which actually do not contribute any meaning other than passing values. Consequently, these communication attribute instances in a parse tree require a large amount of storage space to store the duplicate values at evaluation time. Another problem of conventional AGs is that recursive algorithms cannot be used directly. Any circular attribute dependences will be treated as error, since it will create undefined values at evaluation time.

When we define attributes as TLucid variables, we extend the semantics of attributes to the context space of TLucid. We call such defined attribute grammars *intensional attribute grammars* (IAGs). For a given parse tree, which is considered as a subtree of the arbitrary tree, attributes as TLucid variables are defined at all the nodes of the tree. The attribute dependencies are explicitly specified by node switching operators in terms of attribute definition rules. Since the values of attributes at different nodes are directly manipulated by node switching operators, it is not necessary to use communication attributes to help pass values in the parse tree.

By extending attributes as TLucid variables, we also extend the value of an attribute at a node from a single data value to a data stream. The value of an attribute at a node at a time point can be defined by its previous values by using time switching operators. This kind of "self-dependency" does not create undefined values, because they refer to existing values from previous time points. Using time dimensions, circular attributes in conventional attribute grammars can be defined non-circularly as temporal attributes. The termination of the evaluation of a temporal attribute value is explicitly specified by the termination condition through time switching operators.

2 TLucid

TLucid is an extension of Lucid. In addition to the time dimension in Lucid, TLucid has an *arbitrary tree* dimension $\mathcal{T}$. The tree dimension is the set of all lists of natural numbers, i.e.

$$\mathcal{T} = \bigcup_{i \in \omega} \omega^i$$

where ω is the set of natural numbers and ω^i is the Cartesian product of i copies of ω.

Given a tree π, that is, a conventional ordered tree with one node indicated as the root, in which the child nodes of a non-leave node are ordered by natural numbers, we correspond nodes of π to elements of $\mathcal{T}$ as follows.

1. The root node of π corresponds to $[\,] \in \mathcal{T}$.

2. Let n be a non-root node of π, m be the parent node of n in π, and n be the i^{th} child node of m, n corresponds to $cons(i, s)$ where m corresponds to $s \in \mathcal{T}$.

We call a such defined tree a $\mathcal{T}$-index tree. For example, Figure 1 shows the $\mathcal{T}$-index tree corresponds to a parse tree for binary string "10.01".

Figure 1: An indexed tree

To manipulate values on the tree dimension, we define six primitive node switching operators in TLucid. They are:

```
index
root x
parent x
nextsib x
child(x, k)
globally x
```

For any node n in the tree, the operator **index** returns the index of n. The operator **root** switches context from any node n to the root node of the tree. The operator **parent** switches context from a node to its parent node. The operator **nextsib** switches context from a node to its next sibling. The binary operator **child** switches context from a node to one of its child nodes. Suppose that a node m is the k^{th} child of it parent node n, and a variable x at n is defined by

$$x = child(y, k)$$

and y has the value 370 at m, then x at n will have the value 370. The operator **globally** has one operand with boolean type. At a node n, the operator **globally** will test the value of its

operand whether equal to *true* on all the nodes in the subtree rooted by n. Notice the since a TLucid expression is defined on the infinite arbitrary tree $\mathcal{T}$, it is impossible to test the values at all the nodes in the infinite tree. To make the operator **globally** executable, we have to apply a restriction: for a given expression **globally** x at a node n, if the values of x at each node in the *treeshape* of x rooted by n are *true*, then the expression has the value *true*, otherwise it has the value *false*. Here the *treeshape* of an expression at a node n is a subtree rooted by n on which the expression has non-*eod* values, and the subtree is a $\mathcal{T}$-tree by removing the parent index of n from all the nodes in the subtree. For example, in Figure 2 the variable x has a tree shape in (a), but does not have a tree shape in (b) and (c) since the nodes at which x has the non-*eod* value do not constitute a $\mathcal{T}$-index tree.

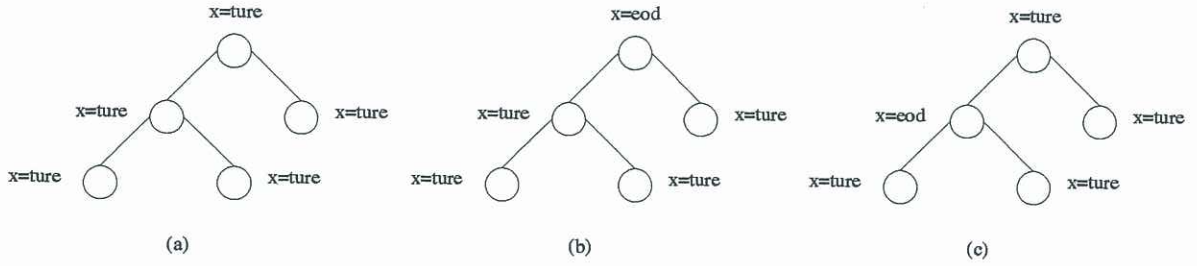

Figure 2: Treeshape of expressions

Using intensional operators, algorithms with tree structures can be expressed in TLucid. The following program defines variable *bstree* that represents a binary search tree formed by its tree shape according to a given list, *input-list*, of integers.

```
bstree
where
    bstree = if partial-list eq nil then eod
             else head(partial-list) fi;

    partial-list = if isroot then input-list
                   elseif parent partial-list eq nil then nil
                   elseif islchild then less(parent bstree,
                                             tail(parent partial-list))
                   elseif isrchild then greater(parent bstree,
                                             tail(parent partial-list))
                   else nil fi;

    less(x, list) = if list eq nil then nil
                    elseif x >= head(list)
                        then cons(head(list), less(x, tail(list)))
                        else less(x, tail(list)) fi;

    greater(x, list) = if list eq nil then nil
                       elseif x < head(list)
                           then cons(head(list), greater(x, tail(list)))
                           else greater(x, tail(list)) fi;

    islchild = ischild(0);
```

```
    isrchild = ischild(1);
    isroot = index eq [ ]
    ischild(k) = head(index) eq k
    sibling(x, k) = parent child(x, k)
end.
```

In the program, we define two variables *bstree* and *partial-list*. The value of *partial-list* at the root node is the input list. The value of *partial-list* at a left or right child node is a sublist of its parent's partial list that contains those elements smaller or greater than the head of the partial list at its parent node. The value of *bstree* at each node is the head of the partial list at that node. Figure 3 shows the binary search tree with the value of *partial-list* at each node, when the *input-list* = [5 2 7 3 6 9 8 1 4].

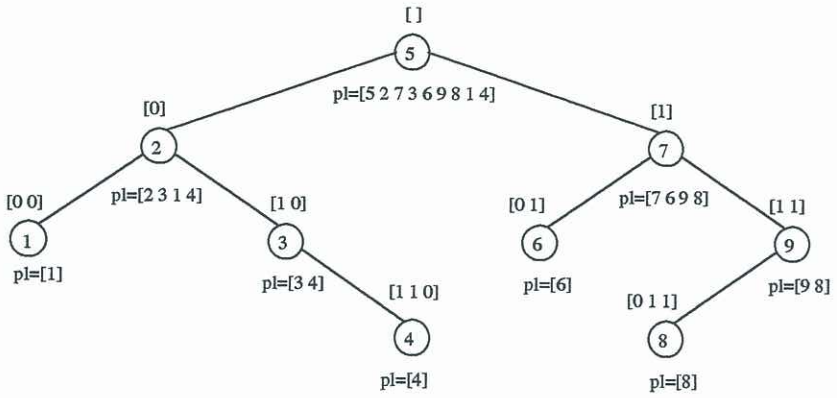

Figure 3: The binary search tree for input list [5 2 7 3 6 9 8 1 4]

3 Intensional Attribute Grammars

3.1 Attribute Grammars

Attribute grammars are a formalism for specifying the syntax and the context-sensitive semantics of programming languages, as well as for implementing editors and compiler-writing systems [1][7]. The basic idea of attribute grammars is that for each node in a given parse tree which represents the syntax of a language string, we associate some attribute values with the node. Those attribute values specify the semantics of the node. The attribute values in a parse tree are evaluated according to the corresponding definitions associated with the context-free grammar of the language.

For example, the following is an attribute grammar that defines a language whose sentences are binary numbers [4].

$N \rightarrow L \,.\, L$
$\quad N.value = L_1.value + L_2.value$
$\quad L_1.scale = 0$

L_2.scale = − L_2.length
N → L
N.value = L.value
L.length = 0
L → L B
L_0.value = L_1.value + B.value
L_0.length = L_1.length + 1
L_1.scale = L_0.scale + 1
B.scale = L_0.scale + 1
L → B
L.value = B.value
L.length = 1
B.scale = L.scale
B → 1
B.value = $2^{B.scale}$
B → 0
B.value = 0

In the above example, the attribute *value* is associated with grammar symbols N, L, and B. The attribute *length* is associated with grammar symbol L. The attribute *scale* is associated with grammar symbols L and B. At a node of a particular parse tree of the above attribute grammar, an attribute instance of *value* is a rational decimal value computed from attribute instances in the subtree rooted by the node. The value of *length* at a node is an integer that is the length of the substring represented by the subtree rooted by the node. The value of *scale* at a node is an integer that is the scale of the node. The attribute *value* associated with the start symbol N at the root node denotes the semantics of the sentence. The semantics of a sentence is obtained by evaluating attribute instances on the parse tree. The communication attributes in the example are $N.value$ in the second production, the $L.value$ and $B.scale$ in the 4th production.

3.2 Defining Attributes as TLucid Variables

As TLucid variables, using node switching operators, attributes in a parse tree can directly be defined as combinations of attribute values at other parts of the tree. Therefore most of communication attributes can be removed from IAGs. Let's consider a type checking example. To check the type consistency in a program, an attribute grammar needs to define an attribute *type-table* whose value is an aggregate value containing the type declaration information collected from the declaration part of a program. In the statment part of the program, when the type checking is performed, the value of *type-table* needs to be passed to each node which represents an identifier. In conventional attribute grammars, a communication attribute for passing the value of *type-table* is required. However, in IAGs, we can define the value of *type-table* at the root node, and all other attributes at any node in a parse tree can directly refer the value using the node switching operator *root*. The following example is an intensional attribute grammar for type checking. In the example, we define an attribute *type-table* which is treated as a "global". For simplicity, here we omit some

of the production rules and terminal symbols. The purpose of the attribute grammar is to report whether there is a type error but does not care about the location or nature of the error.

As shown in the example, an IAG consists two parts: (1) is an expression called *defining expression* whose value denotes the semantics of a given parse tree, and (2) another part is the context-free grammar with attribute definition rules.

```
% The defining expression E
root (child(no-error, 1))

% The context-free grammar and the local attribute definitions
Prog     -> DeclList StatList
            type-table = child(type-list, 0);

DeclList -> Decl DeclList
            type-list = cons(child(type, 0), child(type-list, 1));
            | .
            type-list = nil;

Decl     -> Id ":" Id ";"
            type  = maketype(child(lex, 0), child(lex, 1));

StatList -> Stat StatList
            no-error = child(no-error, 0) and child(no-error, 1);
            | Stat
            no-error = child(no-error, 0);

Stat     -> Id "=" Expr  ";"
            no-error = if match(gettype(child(lex, 0), type-table),
                                child(type, 1))
                       then true
                       else false fi;

Expr     -> Expr + Expr
            type   = if check(child(type, 0), child(type, 1))
                     then child(type, 0)
                     else Err fi;

            | Expr - Expr
            type   = if check(child(type, 0), child(type, 1))
                     then child(type, 0)
                     else Err fi;

            | Id
            type = gettype(child(lex, 0), root type-table);
```

There are five attribute symbols in the above example. The value of *type-table* is a list of pairs which can be viewed as a symbol table. The attribute *type-list* contains partial type information at the nodes labeled by grammar symbols `DeclList` and *Decl*. The attribute *no-error* has boolean value at nodes associated with grammar symbols `Expr`, `StatList` and `Stat`, which denotes whether there is a type error in the subtree rooted by the node. Attribute *type* at a node denotes the type of the node. *Err* is a special data value which denotes a type error.

Given a program fragment defined by the above attribute grammar

```
  ......
a : int
b : char
  ......
a = b ;
  ......
```

the attribute dependencies for the above program fragment on the corresponding attribute value (derivation) tree is illustrated in Fig 4.

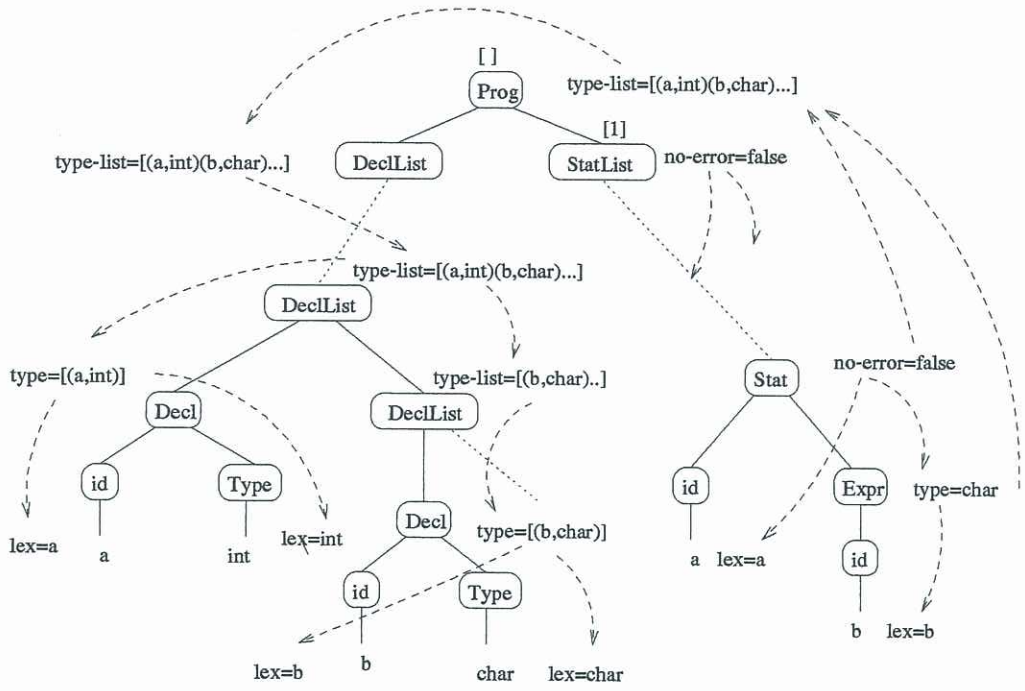

Figure 4: An attribute value tree

The evaluation is started by demanding the value of the defining expression, which in tern demands the value of *no-error* at node [1]. The demand then produces new demands according to the data dependencies and sends the new demands out. When a demand reaches a node labeled by `Expr` and the only child of the node is labeled by `id`, the attribute *type* obtains the type of the identifier from *type-table* at the root node [ ] directly by using the node switching operator *root*.

Thus, at evaluation time, the value of *type-table* does not have to pass to the nodes which require the value *type-table*. The attributes in the tree which depend on the value of *type-table* can directly refer to the value of *type-table* through the node switching operator *root*. The space and time saving is significant for an attribute evaluator using this approach. Since the size of a

symbol table is usually very large, it makes it almost impossible to implement attribute evaluation by copying the duplicated values to each node on a parse tree. Although we can just pass the address of the *type-table* to each node, but the process of passing addresses in a large parse tree is still inefficient on time and space.

3.3 Defining Circular Attribute Grammars

Conventional attribute grammars have no expressive power to specify circular attribute definitions except those that can be solved by the successive approximation fixed-point finding approach [2], since the circularly defined attributes will yield undefined values at evaluation time. As TLucid variable, IAGs define attribute values as data streams. Using time switching operators, circular attributes of conventional attribute grammars can be defined non-circularly in terms of data streams, as long as these *"circularly"* defined attribute values have computable solutions.

One of the application of circular attribute grammar is type checking. As we know, some programming languages allow type use before type definition. The type checking for these languages usually require multiple passes which can not be defined by conventional attribute grammars. The following example is an intensional attribute grammar fragment for Lucid without nested *where* clauses. An expression of Lucid may have one of the three basic types - integer, char, and list. To perform type checking, we associate an attribute *type* with each node labeled by any grammar symbol. The purpose to associate attribute *type* with nodes labeled by grammar symbols `DefList` and `Def` is to make *type* have the same tree shape as the attribute value tree, since the semantics of operator *globally* depends on the tree shape. The type checking terminates when the value of *type* at any node in a given parse tree does not change along the time dimension.

```
root (type asa globally (type eq next type))

%Productions and definitions
Prog      -> Expr
             type-list = [ ];
             type = child(type,0);
             | Expr "where" DefList "end"
             type-list = [ ] fby child(type-list,1);
             type = child(type,0);

DefList   -> Def DefList
             type-list = insert-type-list(child(newtype,0),
                                          child(type-list,1));
             type = "Def";
             | .
             type-list = [ ];
             type = "Def";

Def       -> ID "=" Expr ";"
             type = "Def";
             newtype = makepair(child(lex, 0),child(type,1));
```

```
Expr     -> expr "+" expr
            type = if child(type,0) eq "int" and
                      child(type,1) eq "int"
                      then "int"
                      else Err
                   if;
            | expr "-" expr
            type = if child(type,0) eq "int" and
                      child(type,1) eq "int"
                      then "int"
                      else Err
                   if;
            | expr "fby" expr
            type = if child(type,1) eq eod
                      then child(type,0)
                   elseif child(type,0) eq child(type,1)
                       then child(type,0)
                       else Err
                   fi;
            | ID
            type = get(child(lex,0),root type-list);
            | NUM
            type = "int";
            | CHAR
            type = "char";
```

For example, given an expression with the form:

```
a
  where
    a = b fby c;
    b = 5;
    c = 'A' fby c;
  end;
```

The expression will yield a wrong type since b has the type 'int' and c has the type 'char'. Figure 5 – 7 show the attribute value trees at time 0, 1, and 2.

The iteration is terminated at time 2 since the value of *type* at time 2 is equal to the value of *type* at time 1.

4 Summary and Future Work

TLucid is designed for defining attribute definitions, though it can be programmed independently. As TLucid variables, the values of attributes are intensions from the intensional context space to a domain of values. Using node switching operators, the value of an attribute at a node can directly depend on the values of attributes at other nodes which do not have directly links to the node. In other words, some attributes can be defined as non-local variables to a node. Using time switching operators, the value of an attribute at a node can depend on some previous values of other attributes or itself. The attributes based on iterative algorithms are defined as temporal objects whose values at different time points are viewed as the values at different iterations.

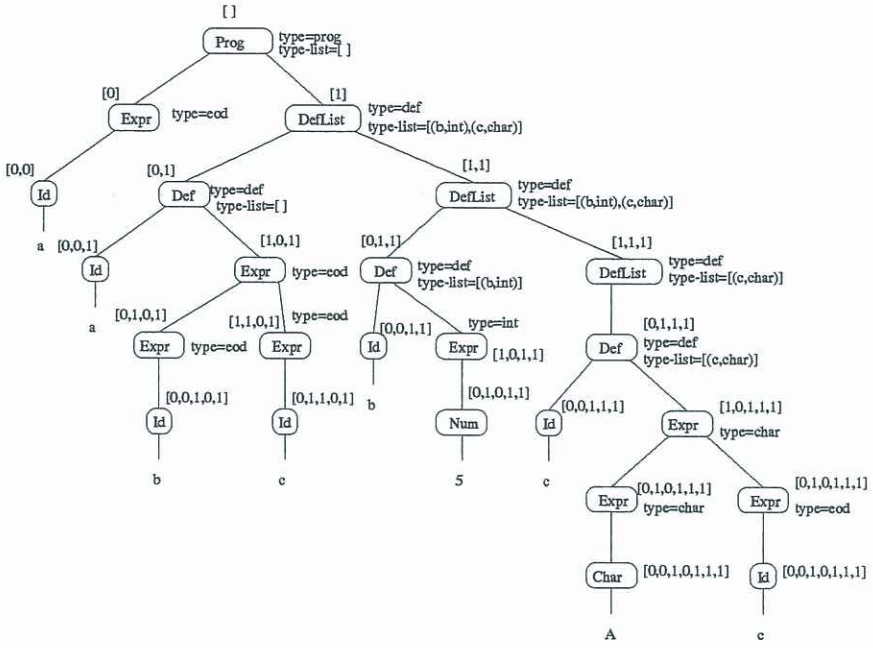

Figure 5: The attribute value tree at time 0

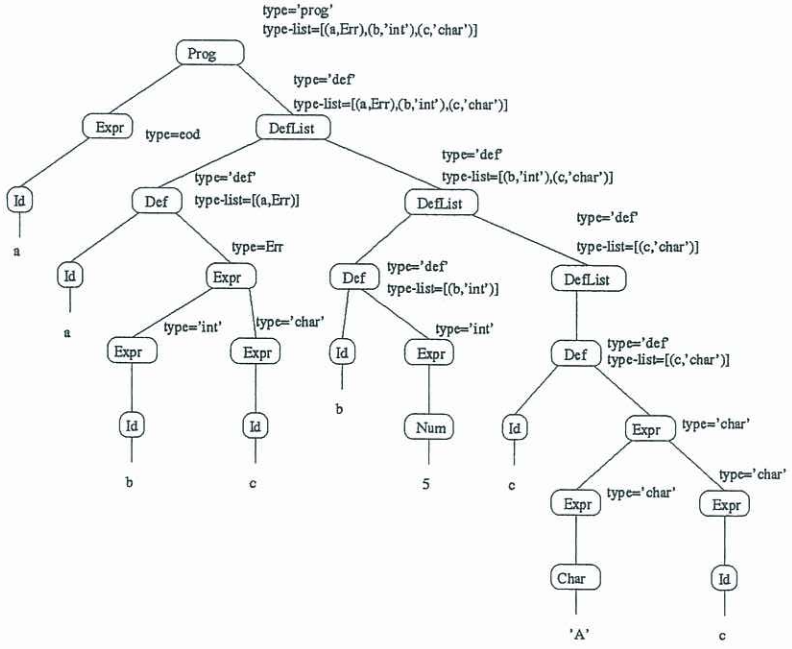

Figure 6: The attribute value tree at time 1

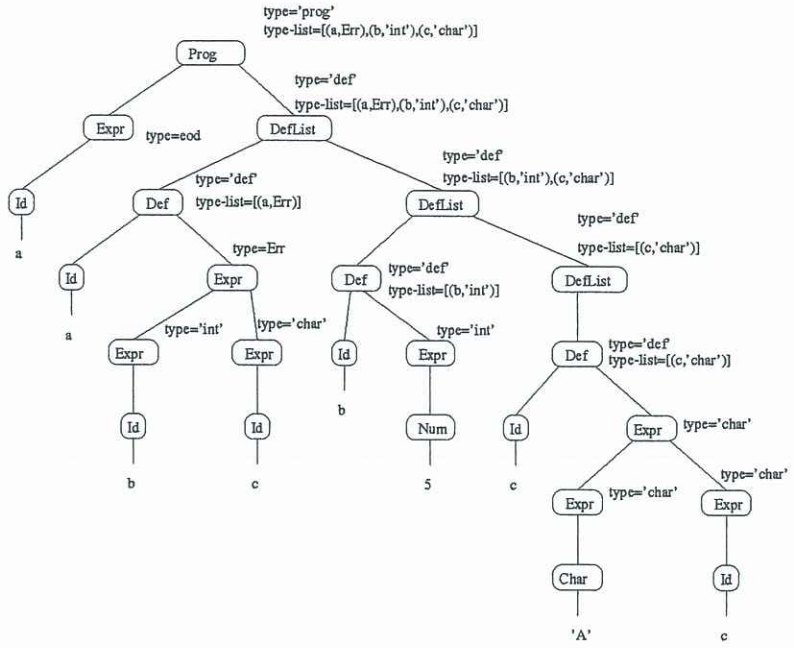

Figure 7: The attribute value tree at time 2

Incremental evaluation is an important issue in the research on programming environments, especially for large programs [6][3]. For a program of a language defined by an attribute grammar, we can view that the values of attributes on the parse tree corresponding the program is the environment of the program. Therefore, when a modification is made to the program, the environment or the attribute values have to be reevaluated. By adding a *version* dimension, we can consider the value of an attribute as a function of *version*. That is, after each modification indexed by a version point, the attributes whose values have been changed at certain nodes by the modification will have the updated values at these nodes at the version point corresponding to the modification. Such defined intensional attribute grammars will not only allow incremental evaluations, but also have the power to express different versions of a program. In other words, the *history* of the modifications to a program can be represented by the sets of attribute values, each of which corresponds to a version of the program, at different versions.

Conventional attribute grammars are first order in the sense that attribute instances are defined by first-order semantic functions, which take zero-order arguments and produce zero-order values. On a parse tree generated by a higher order attribute grammar [8][9], the value of a non-terminal attribute instance at a node can be a tree structure, instead of a single data value. In this sense, the grammars have higher order. A motivation of high order attribute grammars is to build a parse tree dynamically during attribute evaluation, according to some attribute values that have

been evaluated so far. For example, consider a programming language that lets users to declare the precedence of operators, instead of pre-defined. By adding an extra tree dimension to the context space, we can allow attribute values at a node of a parse tree that corresponds to an index tree in the outer tree dimension to vary in the inner time dimension, that is, the values as intensions have tree shape in the inner tree dimension. The dynamic expansion of the original parse tree can be done by transforming values in the inner tree dimension to ones in the outer tree dimension.

References

[1] R. Farrow. Generating a production compiler from an attribute grammar. *IEEE Software*, 1(4):77–93, 1984.

[2] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. *SIGPLAN Notices*, 21(7):85–98, 1986.

[3] S. E. Hudson. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems*, 13(3):315–341, 1991.

[4] D. E. Knuth. Semantics of context-free language. *Math. System Theory*, 2(2):127–145, 1968.

[5] D. E. Knuth. Semantics of context-free language: Correction. *Math. System Theory*, 5(1):95–96, 1971.

[6] T. Reps. Optimal–time incremental semantics analysis for syntax–directed editors. In *the ninth ACM Symposium on Principles of Programming Languages*, pages 169–176, 1982.

[7] T. Reps. *Generating Language–Based Environments.* The MIT Press, 1984.

[8] T. Teitelbaum and R. Chapman. Higher–order attribute grammars and editing environments. In *ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 197–208, 1990.

[9] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 131–145, 1989.

[10] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, London, England, 1985.