AN EDUCTIVE COMPILER GENERATOR FOR ATTRIBUTE GRAMMARS

Senhua Tao
Department of Computer Science
University of Victoria
B.C. Canada

Abstract

In this paper, an Attribute Grammar Compiler Generator AGCG is introduced. The compilers created by AGCG adopt a demand-driven computation model. In the demand-driven model, the evaluation order of variables of a program depends only on the data dependencies in the program. Since the evaluation order is determined automatically at run time, therefore no algorithm is needed to determine the evaluation order when a compiler is created.

1. Introduction

Attribute grammars have been studied widely as a method for specifying programming languages. In a language defined by an attribute grammar, the semantics of the language is represented by the attribute values. The evaluation of attributes is based on a parse tree. The value of an attribute of a node in a parse tree may depend on the attribute values of other nodes. If an attribute depends on the attribute values of its parent node, the required values will flow from the parent node to the child node; we call this kind attributes \textit{inherited attributes}. If an attribute depends on the attribute values of its child nodes, the required values will flow from the child node to the parent node; we call this kind attributes \textit{synthesized attributes}. Since there exist two directions of attribute value flow in a parse tree, the algorithms to determine the evaluation order become complicated.

This paper describes an attribute grammar compiler generator AGCG and its implementation. The compilers generated by AGCG are based on a demand-driven computation model. In the demand-driven model, the evaluation order of variables of a program depends only on the
data dependencies in the program. The evaluation order is determined automatically at run time. When this model is used for evaluating attributes, no pre-determined evaluation order is required.

In the demand-driven model, an eduction technique [AS85][FA85] is adopted, therefore, only those required attributes are evaluated. That means that there are no unnecessary computations performed during the evaluation of attributes. Another feature of the demand-driven method is that the parallelism in algorithms can be exploited automatically. With this feature, we expect that high performance can be obtained when the attributes are evaluated on a demand-driven multiprocessor architecture.

2. Attribute Grammar

An attribute grammar [AH86][KN68][KN71][WA84] is an extended context-free grammar which specifies the syntax and semantics of a language. An attribute grammar $AG = \langle G, A, D \rangle$ consists of a context-free grammar $G$, a set $A$ of attribute symbols, and an indexed family of attribute definition rules (or definition rules) $D = \langle D_p \rangle_{p \in \text{productions of } G}$. If $p$ is the production

$$X_0 \rightarrow X_1, X_2, ..., X_n$$

the definition rules in $D_p$ have the form

$$s_{i_0} : a_0 = E(s_{i_1} : a_1, s_{i_2} : a_2, ..., s_{i_k} : a_k)$$

where $s_{i_j}$ corresponds to the symbol $X_{i_j}$ in the production $p$ ($0 \leq i_j \leq n$), $a_j$ is an attribute symbol and $a_j \in A$ ($0 \leq j \leq k$), and $E$ is some expression in a side-effect free definition language.

Expressions of the form $s_i : a$ ($0 \leq i \leq n, a \in A$) are called attribute expressions, and the attribute expressions on the left-hand side are called defining attribute expressions.

Now let us see an attribute grammar given by Knuth [KN68] which transforms a binary string to the value denoted by the binary string.
Figure 1 An attribute grammar for transforming binary strings to the values

According to the attribute grammar, a parse tree can be constructed for an input binary string. In a parse tree, some attribute values are associated with each node. There are dependencies among those attribute values, which are determined by the definition rules. To chart the dependencies in a parse tree, we attach the attribute symbols to those nodes where they have values. If a value of attribute \( a \) at node \( N_i \) depends on the value of attribute \( b \) associated with node \( N_j \) (here \( i \) can be equal to \( j \)), an arc is drawn from the attribute symbol \( a \) of the node \( N_i \) to the attribute symbol \( b \) of the node \( N_j \). The arc shows the dependency relation between two attribute values implied by the definition rules. Such an arc is called an \emph{attribute dependence}. The resulting graph is an \emph{attribute dependency graph} which is called a \emph{derivation tree}. For example, a derivation tree of a binary string '10.01' is shown in Figure 2.
Figure 2 A derivation tree of '10.01'

The dependency graph of a derivation tree describes the attribute evaluation order in the tree, that is, if there is an arc from node $N_i.a$ to node $N_j.b$, then $N_j.b$ must be evaluated before $N_i.a$. Form the derivation tree, we can find that synthesized attributes are evaluated bottom-up and inherited attributes are evaluated top-down. If a derivation tree contains only synthesized attributes, a simple bottom up scheme can be used for evaluating attributes. In a mixed model, determining the evaluation order becomes a complicated problem.

3. Demand-Driven Computation

Demand-driven computation model is based on a data flow network. In a data flow network, each node represents an operation (such as "+" or "/"). Each arc between two nodes represents a data dependency relation. Data flow along the arc from one node to another node. In the demand-driven approach, the operation of a node can be performed only if there are data items on all the incoming arcs and, in addition, there is a demand for the result of the operation. If there is a demand for a node but there is an incoming arc of the node with no data item on it, then a demand for the data item is sent out along the reverse direction of the arc. The result of the
operation in a node is sent out along the node's output arc.

The demand-driven computation has two obvious characteristics. Firstly, no static evaluation order is required. The evaluation order is determined dynamically at run time. Secondly, an operation is performed only if it is required; no unnecessary operations are performed in this model. Therefore, it offers another benefit: some errors may be avoided if they only occur in those unnecessary operations. These two properties make the demand-driven approach suitable for attribute evaluation.

We can easily transform a derivation tree to a dataflow network. The definitions of the attributes in a derivation tree are like nodes in a dataflow network; the attribute dependencies among the attributes are like the arcs in a dataflow network; and the attribute values are like the data items flowing along the arcs. For example, for the input string '10.01', the corresponding dataflow network is shown in Figure 3.

![Diagram](image)

**Figure 3** A dataflow network for '10.01'

It is clear that in the demand driven computation model, those unnecessary operations for evaluating the value of attribute v of root node N are avoided.
4. Structure of AGCG

AGCG consists of three parts: Attribute Grammar Filter – AGF, Parser Generator – PG, and Demand Driven Attribute Evaluator Generator – DDAEG.

AGF is the filter of attribute grammars. It reads an attribute grammar and checks the format of the attribute grammar. If the attribute grammar has no syntax errors, the AGF divides the attribute grammar into two parts: a context-free grammar and a collection of definition rules. The context-free grammar is used to create a parse tree generator – Parser, and the definition rules are used to create a demand-driven attribute evaluator – DDAE. A lexical analyzer LEX will be produced automatically from the input attribute grammar. These three components constitute a complete compiler. The structure of AGCG and its output - a complete compiler is shown in Figure 4.

![Figure 4 The structure of AGCG](image-url)

An attribute grammar accepted by AGCG consists of three sections: the declaration section, the production and attribute definition section, and the user-defined function section. An attribute
grammar in this format would look like:

declarations
%%% productions and attribute definitions
%%% user-defined functions

The format of attribute grammars is similar to that of grammars accepted by YACC[JO75]. A PIFL-like functional language[BE83] is used to describe the definition rules and user defined functions. User-defined functions can be used in the attribute definitions. For example, the definition rule:

$0:\text{value} = (\$1:\text{value} + \$2:\text{value}) / 2 ** \$3:\text{value} ;$

may be defined as:

$0:\text{value} = f(\$1:\text{value} , \$2:\text{value} , \$3:\text{value}) ;$

The function $f$ can be defined in the user-defined function section as follows:

$f(a, b, c) = (a + b) / 2 ** c ;$

5. Implementation

DDAE is the most important part of a compiler produced by AGCG. DDAE is based on a Demand–Driven Evaluator — DDE, and a group of warehouses (associated memories) which store parse trees, attribute values, and definition rules. The structure of DDAE is shown in Figure 5. In DDAE, there are three warehouses, the definition rule warehouse — DRW, the parse tree warehouse — PTW, and the attribute value warehouse — AVW. Since the definition rules are never changed when the compiler is created, DRW is included in DDE.

DDAE accepts a parse tree and a group of attribute values (the lexical values of the terminal nodes) from the Parser, stores the parse tree in PTW, and initializes AVW with the lexical attribute values.
DDE includes a basic popmachine[WA85], a node stack – *nstack*, an attribute value address stack – *astack*, and a definition rule warehouse – *DRW*. The popmachine and the stacks support the demand-driven evaluation of attributes.

![parse tree and initial attribute values](image)

Figure 5 The structure of the DDAE

5.1. Basic Popmachine and Popcode

The basic popmachine is built around the value stack, which is a stack of POP-2[BU71] data items. The data types of POP-2 are numbers, strings, words, and lists. The basic operations work on the top elements of the stack, read from and write onto the stack, and can pop the stack, swap the top elements, rotate the top n elements, and so on.

The popmachine is driven by popcode programs. The format of popcode is very similar to that of PostScript[AD85]. A popcode program is a list of POP2 data items, each of which is either an operand to be placed on the stack, or an operation to be executed. If a data item is a list, a string, a number or a word which follows a quote, the popmachine recognizes it as an operand. If a data item is a word, the popmachine recognizes it as an operation. The popmachine scans a
popcode program from left to right. If an item is an operand, the popmachine pushes it on the top of the \textit{vstack}; if an item is an operation, the popmachine executes it.

Here is a simple popcode program:

\begin{verbatim}
[2 3 + [a] [b] <>]
\end{verbatim}

This program would be executed as follows: push the integer number 2 on the \textit{vstack}, push the integer number 3 on the \textit{vstack}, pop two operands from the \textit{vstack}, add the two operands, and push the result 5 on the \textit{vstack}, push the list [a] on the \textit{vstack}, push the list [b] on the \textit{vstack}, pop the two operands from the \textit{vstack}, append the two lists, push the resulting list [a b] on the \textit{vstack}. After the popcode program is executed, the two top elements of the \textit{vstack} will be:

\begin{verbatim}
[a b]
5
\end{verbatim}

The basic popmachine has no storage other than its \textit{vstack}. Therefore there are no fetch or store operations in the basic instruction set. In order to evaluate attribute values, the popmachine is extended by adding some warehouses. In DDAB, AVW constitutes the memory of the popmachine and therefore, the popcode must also be extended so that the popmachine can fetch/store the attribute values from/to AVW. Two new operators, "$" and "&" are introduced into the popcode. The "$" is "fetch an attribute value" operator. The function of "$" not only fetches an attribute value, but also evaluates the value when it is not in AVW. The "&" is "store an attribute value" operator. It stores an attribute value into AVW.

Besides the \textit{vstack}, there is an environment stack, \textit{estack}, in the popmachine. This stack is used to evaluate user-defined functions.

5.2. Translating a Definition Rule to Popcode

An attribute definition rule needs to be translated into a popcode program before it can be executed on the popmachine. Consider the third production and the set of definition rules in
Figure 1:

L:  L B  
    [  
        $0:v = $1:v + $2:v ;  
        $0;l = $1:l + 1 ;  
        $1:s = $0:s + 1 ;  
        $2:s = $0:s ;  ]  

The definition rules can be translated as follows:

[[c 1 v] $ [c 2 v] $ + &]  
[[c 1 l] $ 1 + &]  
[[p 0 s] $ 1 + &]  
[[p 0 s] $ &]

In a definition rule, only the right-hand side of the definition rule is translated to the popcode. The left-hand side of the definition constitutes the index of the definition in DRW. In the popcode, the operator "$" follows an operand list. The list represents the relation between an attribute expression (on the right-hand side) and the defining attribute expression in the definition rule. The first item of the list is a flag which can have one of the three values: "c", "p", and "s". The "c", "p", and "s" mean child, parent, and sibling respectively. The second item is an integer number to indicate which child or sibling it is. The last item is the attribute symbol. For example, a string reduced by the above production can be described as following:

```
  L1
 /  
L2   B
```

According to the definition rules of the production, the $0:v is associated with the node $L_1$; the first attribute expression, $1:v, is associated with the node $L_2$; the second attribute expression, $2:v, is associated with the node $B$; and so on. For the first definition rule, the generated popcode 
"[c 1 v] $" means that the value of the attribute expression $1:v is associated with the first child $L_2 of the node $L_1$. 

121
5.3. The Demand-Driven Evaluator – DDE

DDE is based on the popmachine. To evaluate attribute values, two new operators, "$" and 
"&", are introduced into the popcode. To implement the two operators, two new stacks, \textit{astack} 
and \textit{nstack} are introduced into the popmachine.

The \textit{astack} contains the entries for attribute values in AVW which are required but have not 
been evaluated yet. Since each attribute value is associated with a node in the parse tree, DDE 
moves the evaluation from one node to another when it evaluates the attributes on the parse tree. 
Therefore, DDE needs the \textit{nstack} to record the path followed through the tree. We call the node 
on top of the \textit{nstack} the \textit{current node}.

The evaluation of attributes begins by evaluating the value of an attribute \textit{a} at the root of 
the tree. Initially, the \textit{vstack} is empty, the \textit{nstack} contains the root node of the parse tree, and the 
\textit{estack} contains the user defined function definitions. DDE executes the popcode \([c\ 0\ a] \ $ \ &\] to 
request the value of \textit{a} from AVW. The popmachine pushes the list \([c\ 0\ a]\) onto the \textit{vstack}, and 
exeutes the operator "$". Next, the popmachine pops the operand from the \textit{vstack}. Since "child 
0" means the current node, DDE tries to find the value of \textit{a} at the current node in AVW. Since 
the value of \textit{a} is not available, DDE pushes the entry of the value in AVW onto the \textit{astack}. 
Then, DDE finds the definition rule of \textit{a} and pushes it onto \textit{vstack}. The evaluation process is 
recursive and the recursion is stopped when the definition of an attribute is a constant.

The "&" stores the attribute value which is on the top of \textit{vstack} into the AVW labeled 
according to the top element of \textit{astack}, and pops the top element of \textit{astack}. At the same time, 
DDE pops the top element from \textit{nstack} as well. Therefore, the current node is changed back to 
the node which requires the attribute value just computed. Then, DDE executes the rest of the 
popcode. The evaluation is stopped when the value of attribute \textit{a} of the root node is computed.

When an attribute value is required, DDE may transfer the evaluation to the parent, sibling, 
or child node of the current node to evaluate the required attributes. After the attribute is
evaluated, DDE moves the evaluation back to the original node so that it can execute the rest of the popcode. The movement from one node to another reflects the data dependency relationship between two attribute values, and thus DDAE builds a derivation tree dynamically at run time.

In the warehouses, the items in DRW and PTW are always available, but values in AVW may not be available. When DDE requests an attribute value from AVW, and the value is available, DDE simply returns the value; if the value is not available, DDE temporarily stores a special value in its place. The reason for storing a special value for a required attribute is that this value can be used to stop an infinite loop. Since there exist synthesized and inherited attributes in a derivation tree, the definition of a particular attribute value can be circular. If the circular definition cannot be detected before the DDAE is created, an infinite loop will be encountered when the attribute value is evaluated. In order to stop the infinite loop, when DDE fetches an attribute value from AVW, it checks the value fetched. If it is the special value, the attribute value is required by itself. In this case, DDAE reports a circular definition error and abandons the evaluation.

In addition, when there is a function in the definition rule of a demanded attribute, DDE will find the definition of the function from the estack, instead of DRW warehouse. Then DDE executes the function.

6. Conclusions

AGCG is a useful software tool for creating language compilers automatically. AGCG requires that compiler writers provide only attribute grammars. Thus, when designing compilers, they can concentrate on the characteristics of the language, instead of being bothered by the implementation details.

AGCG uses a functional attribute definition language. Thus, the definition rules can be given in a natural form. Since AGCG uses a demand-driven model, the evaluation order is deter-
mined at runtime; the evaluation of attribute values is simpler and more efficient; and unnecessary computations can be avoided.

In the further work, we will use the intensional dataflow language Lucid to specify the definition rules of attribute grammars and build an attribute evaluator which is based on the Lucid interpreter. When the definition rules specified by a functional language which has the intensional property, the attribute grammars will have more power to define the semantics of languages. We hope such a new attribute grammar evaluator can be suitable to deal with natural language recognition.

References


124
Appendix

An Example

Consider a language which includes only data type declarations and assignment statements. We give an attribute grammar which specifies the translation of a source program in the language to a target code. The translation uses two target machine instructions: the fetch instruction and the addition instruction, and four registers: r0, r1, r2 and r3. A source program consists of two statements: an identifier declaration statement and an assignment statement. In the source program, a variable must be declared as int and an expression includes only addition operations. After each addition operation, the source register will be released. The compiler checks the data type for each variable. A complete attribute grammar is shown in Figure 5.

```%
left "+

%%

prog: decl stem
    [ $0:code = $2:code;
      $2:dist = $1:vars; ]
    ;

decl: "int" vars ";"
    [ $0:vars = $2:vars; ]
    ;

vars: vars "," IDENT
    [ $0:vars = $3:lex :: $1:vars; ]
    ;

vars: IDENT
    [ $0:vars = [$1]; ]
    ;

stem: IDENT "=" expr ";"
    [ $0:code = [% "move", $3:reg, $1:lex %] <> $3:code;
      $3:dist = $0:dist;
      $3:dist = [r0 r1 r2 r3]; ]
    ;

expr: expr "+" expr
    [ $0:code = [% "add", $3:reg, $1:reg %] <> $3:code;
      $0:reg = $3:reg;
      $1:dist = $0:dist;
      $3:dist = $1:plist;
      $1:dist = $0:dist;
      $3:dist = $0:dist;
      $0:plist = $1:reg :: $3:plist; ]
    ;

expr: IDENT
    [ $0:code = if check($1:lex, $0:dist)
      then [% "move", $1:lex, ",", $1:reg %]
      ;
```

125
else "error"
fi;
$0:reg = hd($0:rlist);
$0:plist = tl($0:rlist); ]
;
check(id, typelist) = if typelist ne []
then
  if id eq hd(typelist)
  then "true"
  else check(id, tl(typelist))
  fi
else "false"
fi;

Figure 5 a sample attribute grammar

This attribute grammar has six attributes. Attribute vars describes a list of identifiers which are declared as integer; attribute reg describes a register containing the values of a variable; attribute rlist describes a free register list which can be allocated for a variable; attribute plist also describes a free registers list which links a register to the free register list after an operation releases the register; and attribute code describes the target machine code.

For a program:

int a, b, c;
a = b + c;

The compiled target machine code is:

[move b, r0
  move c, r1
  add r0, r1
  move r1, a]