Compiling circular attribute grammars into Prolog

by Bijan Arbab

This paper describes an algorithm for compiling attribute grammars into Prolog. The attribute grammars may include inherited and synthesized attributes and contain recursive (circular) definitions. The semantics of the recursive definitions is defined in terms of a fixed-point finding function. The generated Prolog code stands in direct relation to its attribute grammar, where logical variables play the role of synthesized or inherited attributes. Thus an effective method for the execution of recursive attribute grammars has been defined and applied.

1. Introduction

Attribute grammars, originally described by Knuth [1, 2], are a mechanism for assigning meaning to strings of a context-free language. To this end, and because of their intuitive nature, attribute grammars have been used for semantic definitions of programming languages, such as SIMULA 67 [3] and PL360 [4]; programming language design [5]; program correctness proofs [6]; program optimization [7]; program translation [8]; question-answering systems [9]; hierarchical and functional programming [10]; and compiler-generating system [11–13]. Attribute grammars have been studied and compared with other formal methods for the specification of semantics and have been given a precise formulation (including recursive definitions) within the framework of initial algebra semantics [14]. Prolog is also well suited for compiler writing and has been used for the development of the DEC 10 Prolog compiler [15].

Previous efforts to develop attribute evaluation mechanisms have included table-driven tree-walk algorithms [16] as well as compilation into recursive procedures with parameters corresponding to attributes [17]. These algorithms, however, deal with absolutely noncircular definitions. Knuth also shows a method whereby an attribute grammar can be converted into an algorithm for recognizing a language, but only if the equations exhibit no circularities [1]. Knuth's algorithm first forms the parse tree and then converts the equations into programs which are executed in an order guaranteed to define the attributes at all nodes. The attribute grammars dealt with in this paper, however, may contain recursive (circular) definitions, and their meaning is defined in terms of a fixed-point finding function.

In this paper, an algorithm for compiling synthesized-only attribute grammars with recursion (circularity) is presented. The language TINY [18] has been selected as the working example. Note that every inherited-and-synthesized attribute grammar can be transformed into a purely synthesized-only attribute grammar with the same underlying context-free grammar as the original attribute grammar [14]. Therefore, in principle the algorithm for compiling synthesized-only attribute grammars can be used to compile inherited-and-synthesized attribute grammars as well. To accomplish this, one can use the technique of Chirica and Martin [14] to transform the inherited-and-synthesized attribute grammar

into an equivalent synthesized-only attribute grammar and then apply the algorithm presented in this paper to compile the resulting synthesized-only attribute grammar.

Inherited-and-synthesized attribute grammars, however, are more intuitive than their equivalent synthesized-only attribute grammars, since the inherited attributes serve to reduce the functionality of the semantic equations. We present a new algorithm for compiling inherited-and-synthesized attribute grammars into Prolog. The language PAM [19] has been selected as the working example.

It is common practice to make use of lambda notation in writing the semantic equations of a synthesized-only attribute grammar. Following this practice, the semantics of TINY is presented in terms of lambda notation. In Section 2 we discuss the representation of lambda notation in Prolog. An outline of an algorithm for compiling synthesized-only attribute grammars appears in Section 3. At this point, it is possible to do semantic computation for subject programs; i.e., a lambda expression is constructed as a result of executing, via a Prolog interpreter/compiler, the Prolog clauses obtained from compilation of the synthesized-only attribute grammar together with an abstract syntax of a subject program. Reduction of the generated lambda expression, with respect to some input, corresponds to the execution phase of the subject programs. To accomplish this, a detour is taken in Section 4, where construction of a lambda machine in Prolog is discussed for the purpose of reducing (solving) lambda expressions. This lambda machine will have proper facilities to deal with run-time semantics of the fixed-point finding operator which was used during semantic evaluation for recursive (circular) attribute grammars. Sample compilation and execution of TINY programs are presented at the end of this section. In Section 5 an algorithm for compiling inherited-and-synthesized attribute grammars into Prolog is presented and applied.

After this work was completed, the author became aware of the work of Deransart and Maluszynski [20] on the relation between attribute grammars and logic programming. The main thrust of their work, however, is different from the theme of this paper. Deransart and Maluszynski show a formal correspondence between attribute grammars and logic programming and proceed to make use of the established result in one of them, namely attribute grammars, to establish some new results in the other, namely logic programming. In their conclusion they state that new evaluation methods for attribute grammars based on the procedural semantics of logic programs are possible. This type of application shows that the expertise in logic programming can also be applied to the field of attribute grammars. However, this issue was not discussed in their paper and was said to require further investigation. This paper, then, demonstrates one such evaluation mechanism for attribute grammars based on Prolog's procedural semantics of logic programs.

```
<exper> ::= <exper> <exper>        % Application
          | L <ids>. <exper>       % Abstraction
          | MU <id>. <exper>       % Least fixed-point finding operator
          | <conditional> | <id> | <op> | <number> | <pred>

<conditional> ::= <exper> → <exper>, <conditional> | true → <exper>

<ids> ::= <id> <ids> | <id>

<pred> ::= eq(<exper>, <exper>)    % Equality test
         | null(<exper>)           % Null test
         etc.

<op> ::= add(<exper>, <exper>) | sub(<exper>, <exper>) |
         mul(<exper>, <exper>) | div(<exper>, <exper>) | etc.
```

Figure 1

Syntax of lambda notation.

```
<exper> ::= apply(<exper>, <exper>)  % Application
          | abstract(<ids>, <exper>) % Abstraction
          | mu(<id>, <exper>)        % Least fixed-point finding operator
          | <conditional> | <id> | <op> | <number> | <pred>

<conditional> ::= cond(<clist>)

<clist> ::= <exper>-<exper>.<clist> | true-<exper>.nil

<ids> ::= <id> <ids> | <id>

<id> ::= Logical Variables (as in Prolog)

<pred> ::= eq(<exper>, <exper>)    % Equality test
         | null(<exper>)           % Null test
         etc.

<op> ::= add(<exper>, <exper>) | sub(<exper>, <exper>) |
         mul(<exper>, <exper>) | div(<exper>, <exper>) | etc.
```

Figure 2

Syntax of lambda notation in clause form.

2. Representing lambda notation in Prolog

It is common practice to make use of lambda notation in writing the semantic equations of synthesized-only attribute grammars. In order to manipulate these equations, we must be able to represent them in Prolog. Here then is one representation of lambda notation in Prolog. The syntax of the lambda notation we use is shown in **Figure 1**.

For example, the factorial function can be written as

MU f.(L x.(eq(x, 0) → 1, true → x * f(x−1))).

The syntactic representation of the above lambda notation in Prolog form is shown in **Figure 2**.

For example, the previous factorial function translated into Prolog form is

mu(F, abstract(X, cond(eq(X, 0)-1.true-mul(X, apply(F, sub(X, 1))).nil))).

The use of logical variables as variables of abstraction simplifies the substitution process; i.e., once X is bound to some structure, all its occurrences throughout the lambda expression will be bound to that same structure through Prolog's unification process. This, however, does imply that the normal scoping rules of lambda notation will not be obeyed! For example, in the lambda expression abstract(X, apply(abstract(X, X), X)), if the outer X gets bound to, say, a, all the X's will be bound to a's, leading to

```
1.   <command> ::= <id> := <expression>
2.               | output <expression>
3.               | if <expression> then <command1> else <command2>
4.               | while <expression> do <command1>
5.               | <command1> ; <command2>

6.   <expression> ::= 0
7.                  | 1
8.                  | true
9.                  | false
10.                 | read
11.                 | <id>
12.                 | not <expression1>
13.                 | <expression1> = <expression2>
14.                 | <expression1> + <expression2>
```

Figure 3

Syntax of the programming language TINY.

Nonterminal	Synthesized attribute
<command>	r: State → State
<expression>	e: State → Values → State

```
1.  r.command = L s. ((L v<z,i,o>.<z(v/Id), i, o>) (e.expression s))
2.  r.command = L s. ((L v<z, i, o>.<z, i, v.o>) (e.expression s))
3.  r.command = L s. ((L v1s1.(v1   → (r.command1 s1)
                               true → (r.command2 s1))
                      ) (e.expression s))
4.  r.command = L s. ((L v1s1.(v1   → (r.command (r.command1 s1)),
                               true → s1)
                      ) (e.expression s))
5.  r.command = L s. (r.command2 (r.command1 s))

6.   e.expression = L s. 0,s
7.   e.expression = L s. 1,s
8.   e.expression = L s. t,s
9.   e.expression = L s. f,s
10.  e.expression = L <z, i, o>. hd(i),<z, tl(i), o>
11.  e.expression = L <z, i, o>. z(Id),<z, i, o>
12.  e.expression = L s. ((L v1s1. not(v1),s1) (e.expression1 s))
13.  e.expression = L s. ((L v1s1. ((L v2s2. v1=v2,s2)
                                    (e.expression2 s1))
                          ) (e.expression1 s))
14.  e.expression = L s. ((L v1s1. ((L v2s2. v1+v2,s2) (e.expression2 s1))
                          ) (e.expression1 s))
```

Figure 4

Semantics of the programming language TINY.

the expression abstract(a, apply(abstract(a, a), a)). This is simply wrong, since the inner X is really a different variable from the outer X; i.e., it will be necessary to carry out substitution properly.

To overcome this problem, a utility function called capture is used to rewrite lambda expressions such that no variable capturing, as above, can occur. For example, capture applied to the above lambda expression will produce abstract(X, apply(abstract(Y, Y), X)), which avoids the problem while allowing us to use logical variables as variables of lambda expressions. The capture predicate is used by the lambda machine every time a substitution is made in order to make sure that the resulting expression will have no captured variables. The Prolog code for the capture predicate is not complicated and can be found in Appendix A.

3. Compiling synthesized-only attribute grammars

The steps which are needed to compile synthesized-only attribute grammars are discussed, justified, and applied with respect to a little programming language called TINY [18]. The purpose of TINY is to provide a vehicle for illustrating various formal concepts in use. The syntax of TINY, in terms of synthesized-only attribute grammars, is presented in **Figure 3**.

The semantics of TINY can be described by a synthesized-only attribute grammar. The state of execution for a TINY program can be taken to be a triple composed of a memory map z, which is a function from identifiers to their values, and an input i and an output o from some domain of values. The semantics of commands can then be described in terms of a function that transforms states into states and the semantics of expressions as a function with signature State → Value × State, or in Curried form, State → Value → State; i.e., expressions produce values as well as change the state. The semantics of the while command (4) is perhaps the most interesting to consider because it contains a cycle. First, the expression is computed with respect to the input state s, which results in the value v1 and the new state s1. If v1 is true (note that no type checking is being performed), then we proceed by executing command1 with respect to s1 and continue by passing the resulting state to r.command; i.e., re-enter the while loop with the state which has resulted from the computation of expression and command1. The complete semantics of TINY appears in **Figure 4**; note that <, >, and , are used only to aid readability and are not part of the syntax of the lambda expression presented earlier.

To compile the definitions in Figure 4, we must first build from the syntax an abstract syntax for the language which is also representable in Prolog form. For example, the abstract syntax for TINY can be represented as in **Figure 5**.

Now, for each production in the abstract syntax of the language, write a Prolog clause as follows:

1. The name of the head clause is the nonterminal appearing on the left of the production rule. For example, for production rule one, this would be command.
2. The first argument of the head clause is the production on the right-hand side, e.g., assign(<id>, <expression>), but all nonterminals are replaced by logical variables; thus the first argument is assign(Id, Expr). Note that logical variables are represented by uppercase letters, possibly numbered, or just an uppercase letter for the first character.
3. The remaining arguments of the head clause are the synthesized attributes corresponding to the nonterminal on the left-hand side of the production rules. Their corresponding values are constructed from attribute grammar definitions as follows:
   - Associate with each occurrence of an attribute and the variables of the lambda expression (of the semantic definition) a new logical variable, and substitute them into the lambda expression. For example, the lambda expression L s. (L v<z, i, o>.<z(v/Id), i, o>) (e.expression s) will become L S. (L V<Z, I,

O>.<Z(V/Id), I, O>) (Expr1 S). After translation into Prolog, according to the rules presented earlier, the lambda expression will become

```
abstract(S, apply(abstract(V.(Z.I.O),
                           ((V\Id).Z).I.O),
                  apply(Expr1, S)))
```

These expressions form the remaining arguments of the head clause. Thus, the head clause formed so far for production rule one is

```
command(assign(Id, Expr), abstract(S, apply(abstract(V.(Z.I.O),
                                                     ((V\Id).Z).I.O),
                                                apply(Expr1, S)))
        ) ← . . .
```

4. Now we must build a body for this head clause. For every synthesized attribute used in the semantic definition (e.g., e.expression), form a clause whose name is the nonterminal corresponding to that attribute (e.g., expression) and whose first argument is a logical variable corresponding to the nonterminal named by the attribute (e.g., Expr); the remaining arguments are those unique logical variables created in the above step (e.g., Expr1). Thus, for e.expression construct the clause expression(Expr, Expr1). The body of the clause, then, is the conjunction of all the above clauses; order is not important. Thus, the clause formed for production rule one is

```
command(assign(Id, Expr), abstract(S, apply(abstract(V.(Z.I.O),
                                                     ((V\Id).Z).I.O),
                                                apply(Expr1, S)))
        ) ←
    expression(Expr, Expr1).
```

Let us now consider the while construct; application of this algorithm would produce the following Horn clause:

```
command(while(E, C1),
        abstract(S,
                 apply(abstract(V1.S1,
                                cond((V1-apply(Rc,
                                               apply(Rc1,
                                                     S1))
                                     ).(true-S1).nil)),
                       apply(E1, S))))
    ) ←
  command(C1, Rc1) &
  expression(E,E1) &
  command(while(E, C1), Rc).
```

Execution of this definition will never terminate, due to the last call, i.e., command(while(E, C1), Rc). Note that the order in which the clauses of the body are formed is not important.

This definition does represent the correct semantics of the while construct. The only problem is that it leads our Prolog program into an infinite recursion. To get around this problem, we use a function called MU, the fixed-point finding function, which is in effect a shorthand for the above infinite definition. The job of the MU operator is to unfold this definition at run time as many times as necessary, possibly infinitely many. So the algorithm is modified to detect this type of infinite recursion and to use the MU operator instead.

```
1.   <command> ::= assign(<id>, <expression>)
2.               | output(<expression>)
3.               | if(<expression>, <command1>, <command2>)
4.               | while(<expression>, <command1>)
5.               | <command1>.<command2>

     <expression> ::= expr(0) | expr(1) | expr(true) | expr(false)
10.              | expr(read)
11.              | expr(id)
12.              | expr(not(<expression1>))
13.              | expr(<expression1>, eq, <expression2>)
14.              | expr(<expression1>, plus, <expression2>)
```

Figure 5

Abstract syntax for the programming language TINY.

Following is the revised algorithm:

1. For each production rule in the abstract syntax, build a Horn clause such that
   a. The name of the head clause is the nonterminal appearing on the left-hand side of the production rule.
   b. The first argument of the head clause is the production on the right-hand side of the production rule.
   c. The remaining arguments of the head clause are constructed from the attribute grammar definitions as follows:
      i. Associate with each occurrence of an attribute and the variables of the lambda expression (of the semantic definition) a new logical variable, and substitute these in the lambda expression.
      ii. If the attribute appearing on the left-hand side of the semantic definition also occurs on the right-hand side, then form a new lambda expression by simply placing a MU operator around the lambda expression. The variable of the MU operator is the logical variable corresponding to the nonterminal on the left-hand side of the semantic definition.
      iii. Translate the resulting lambda expression into Prolog using the syntactic rules presented earlier.
   d. For every synthesized attribute used in the semantic definition, form a clause whose name is the nonterminal corresponding to that attribute, whose first argument is a logical variable corresponding to the nonterminal named by the attribute, and whose remaining arguments are those unique logical variables created in the above steps. The body of the clause is the conjunction of the clauses just formed; order does not make a difference.

Application of this algorithm to the while command will now produce

```
command(while(E, C),
        mu(X, abstract(S, apply(abstract(V1.S1, cond((V1-apply(X,
                                                              apply(Rc1,
                                                                    S1))
                                                           ).(true-S1).nil)),
                                apply(E1, S))))
    ) ←
  command(C, Rc1) &
  expression(E,E1).
```

Application of the algorithm to the remaining semantic equations of TINY produces the Prolog clauses listed in Appendix B, which together can be used to do semantic computation for TINY programs. For example, application of the above Prolog clauses to the TINY program x:=read; output x produces the following lambda expression as the semantic definition:

```
abstract(*0,
        apply(abstract(*1,
                    apply(abstract(*2.*3.*4.*5,
                                   *3.*4.*2.*5),
                          apply(abstract(*6.*7.*8,
                                         apply(abstract(*9,
                                                        *9.*6.*7.*8),
                                               lookup(*6,
                                                      x))),
                                *1))),
              apply(abstract(*10,
                          apply(abstract(*11.*12.*13.*14,
                                         (*11\x.*12).
                                         *13.*14),
                                apply(abstract(*15.*16.*17.*18,
                                               *16.*15.*17.*18),
                                      *10))),
                    *0)))
```

where *N denotes distinct logical variables.

The relationship among the attribute grammar, Prolog code, lambda expression, and lambda machine is presented in **Figure 6**.

It is interesting to note that the inverse relation for the produced Prolog code holds as well; i.e., we can issue a Prolog goal with a particular lambda expression and, if there is a well-formed TINY program that corresponds to that lambda expression, it will be produced. This corresponds, approximately, to execution of the inverse of the attribute grammar. An algorithm for computing the inverse of an attribute grammar is discussed by Yellin and Mueckstein [21]. A closer investigation into this feature is required before any correspondence can be shown.

4. Constructing a lambda machine in Prolog

After the semantic computation of the subject programs, we are left with the semantic definition of the subject programs which are represented in lambda notation. These semantic definitions can be executed via a lambda machine. Thus, a lambda machine interpreter (named solve) is constructed in Prolog to allow execution of semantic definitions. The interpreter solve has two arguments: The first is the lambda expression, and the second is the value of that lambda expression as computed by solve.

Solve first calls capture to ensure that the scoping rules of lambda expressions are not violated; then it calls solve1, which is the actual interpreter. So solve is defined as

```
solve(M, R) ← capture(M, M1) & solve1(M1, R).
```

Solve1, then, has a clause for each of the primitive operations which are supposed to be known by this lambda machine, e.g., negate, eq, add, sub, etc.; i.e.,

```
solve1(negate(E), X) ← / & solve1(E, V1) & negate(V1, X).
solve1(eq(E1, E2), X) ← / & solve1(E1, V1) & solve1(E2, V2) & eq(V1, V2, X).
solve1(add(E1, E2), X) ←/ & solve1(E1, V1) & solve1(E2, V2) & add(V1, V2, X).
```

Note that inside each clause for a primitive operation, e.g., negate, a call is made to a procedure that implements the primitive operation, in this case negate(V1, X). The procedure negate(X, Y) can be defined in Prolog by asserting negate(t, f) and negate(f, t).

The heart of the lambda machine is its capability of dealing with cases where the lambda expression currently under reduction is not a primitive one. If the lambda expression under reduction is an abstraction, then no more simplification is possible and the result is the abstraction itself. Therefore, the last clause of the interpreter is solve1(X, X). However, if the lambda expression is not an abstraction, then it is an application, and the following possibilities exist:

1. Application of two lambda expressions, the first of which is another application: In this case, reduce the inner application and recursively call the interpreter with its result, i.e.,

```
solve1(apply(apply(A, B), C), R) ← / &  solve1(apply(A, B), A1) &
                                        solve1(apply(A1, C), R).
```

2. Application of an abstraction to another lambda expression: Assuming applicative evaluation order (also called inside-out or call-by-value) for the lambda expression, first find a value for the second lambda expression, and then bind this value to the variable of abstraction and solve the substituted body of the abstraction, i.e.,

```
solve1(apply(abstract(V, B), C), R) ← / &  solve1(C, V) & capture(B, B1) &
                                           solve1(B1, R).
```

However, if normal evaluation order (also called outside-in or call-by-name) is assumed, then the variable of abstraction is bound to the second lambda expression (unevaluated), and the process is continued by solving the

body of the substituted abstraction, i.e.,

```
solve1(apply(abstract(C, B), C), R) ← / & capture(B, B1) & solve1(B1, R).
```

Note that in both cases after the substitution a call to capture is made to ensure that the resulting lambda expression has no captured variables.

3. Application of a fixed point function mu to some lambda expression: In this case, bind the variable of mu to a fresh copy of the mu structure (itself) and simply continue by solving the application of the substituted body of mu to the second lambda expression, i.e.,

```
solve1(apply(mu(X, B), C), R) ← / & fresh(mu(X, B), mu(Y, B1)) &
                                 X = mu(Y, B1) & solve1(apply(B, C), R).
```

Note that fresh will return a structure identical to the one it was called with except that all the variables used in that structure are new variables. This is necessary since, when substituting X in the body of mu, no variable capturing should occur.

Using the above interpreter we can execute the definition of factorial, given earlier, to compute the factorial of numbers, e.g.,

```
solve(
   apply(mu(F,
            abstract(X,
              cond((eq(X, 0)-1).(true-mul(X, apply(F, sub(X, 1)))).nil))),
         4),
   R).
```

will succeed with R bound to 24. Semantic definitions produced by the attribute grammars together with the lambda machine define a complete execution model for a language. For example, with the TINY program

```
x := read; sum := 0;
while not x = true do sum := sum + x; x := read end;
output sum
```

as input to the compiled attribute grammar, a lambda expression will result as its semantic definition, which together with an input stream (1.2.3.true.nil) to the lambda machine will produce the final state as follows:

```
((true\x).(6\sum).(3\x).(3\sum).
 (2\x).(1\sum).(0\sum).(1\x).nil). Environment component of the State
nil.                              Input component of the State
6.nil                             Output component of the State
```

For a complete listing of the above lambda machine defined in Prolog, see Appendix A.

5. Compiling inherited-and-synthesized attribute grammars

Consider the attribute grammar mapping PAM [19] into a simple machine language. This attribute grammar uses both synthesized and inherited attributes; the complete definition of PAM's syntax and semantics can be found in [19].

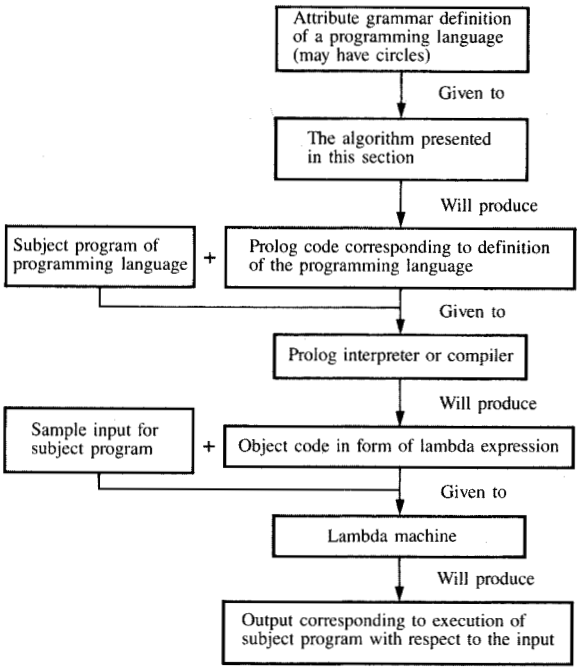

Figure 6

Relationship among AG, Prolog, and lambda expression.

The first step in the compilation, as before, is to build an abstract syntax for PAM that is representable in clause form, and then carry out the following steps:

1. For each production rule in the abstract syntax, build a Horn clause such that
   a. The name of the head clause is the nonterminal appearing on the left-hand side of the production rule.
   b. The first argument of the head clause is the production on the right-hand side of the production rule.
   c. The remaining arguments of the head clause are constructed from the attribute grammar definitions as follows:
      i. Associate with each inherited and synthesized attribute of this nonterminal a new logical variable. These variables form the rest of the arguments of the head clause.
   d. The body of the head clause is a conjunction of clauses formed as follows: For every call made to evaluate an attribute, form a clause whose name is the named nonterminal (in the call) and the arguments of which are the code plus all the logical variables associated with the synthesized and inherited attributes of the called nonterminal. The values of these arguments are further determined from the assignment statements of the attribute grammar.

As an example, consider the first production rule and its semantics:

```
<program> ::= <series>
              Code(<program>) ← append(Code(<series>), 'HALT')
              Temp(<series>) ← 0
              Labin(<series>) ← 0
```

According to the algorithm just presented, the head clause will be program(Series, Code) ←. The attribute grammar for this production contains only one call for attribute evaluation, i.e., Code(<series>). Also note that series has two inherited attributes (Temp, Labin) and two synthesized attributes (Code, Labout). Thus, the body will be series(Series, Temp, Labin, Labout, Code); however, since the attribute equations specify that Temp(<series>) and Labin(<series>) are set to zero, we must modify the body; hence, series(Series, 0, 0, Labout, Code). Since Code(<program>) ← append(Code(<series>), 'HALT'), we must modify the head clause as follows, noting that the . here plays the role of the append function: program(Series, Code.s(halt, nil)) ←. So the entire clause put together is program(Series, Code.s(halt,nil) ← series(Series, 0, 0, Labout, Code).. The structure s(halt, nil) is a representation for the machine language constructs, the first argument of s is the op-code, and the second is the argument for that op-code, if any.

As a second example, let us look at the series production rule and its associated attribute grammar definition.

```
<series> ::= <statement> ; <series>2
             Code(<series>) ← concat(Code(<statement>), Code(<series>2)))
             Labout(<series>) ← Labout(<series>2)
             Temp(<series>2) ← Temp(<series>)
             Labin(<series>2) ← Labout(<statement>)
             Temp(<statement>) ← Temp(<series>)
             Labin(<statement>) ← Labin(<series>)
```

The head clause for this production rule is series(S.Ss, Temp, Labin, Labout, Code) ←, since series has two inherited attributes (Temp, Labin) and synthesized attributes (Labout, Code). There are two clauses in the body of this head clause since there are only two calls made to evaluate attributes, i.e., Code(<series>) and Code(<statement>). The nonterminal <statement> has four attributes as well, two inherited (Temp, Labin) and two synthesized (Labout, Code). Thus, the clauses for <statement> and <series> are statement(S, Temp1, Labin1, Labout1, Code1) and series(Ss, Temp2, Labin2, Labout2, Code2). Now we must take care of the assignments. Since we know that Labout(<series>) ← Labout(<statement>), we can conclude that Labout2 = Labout1, and from the other assignment statements we can conclude that Temp1 = Temp, Labin1 = Labin, Temp1 = Temp, and Labin1 = Labin2. The . is again used to represent the concat operator, and the head clause is modified according to Code(<series>) ← concat(Code(<statement>), Code(<series>2))). Therefore, the final clause for <series> is

```
series(S.Ss, Temp, Labin, Labout, Code1.Code2) ←
    statement(S, Temp, Labin, Labout1, Code1) &
    series(Ss, Temp, Labout1, Labout, Code2).
```

Complete compilation of PAM's definition will produce the Prolog program listed in Appendix C; sample semantic computations for some PAM programs are illustrated in Appendix D.

Conclusion

It was demonstrated that synthesized-only and inherited-and-synthesized attribute grammars with recursive (circular) definitions can be compiled into a set of Prolog clauses, thus providing an effective method of execution for attribute grammars. The compilation algorithm was applied to two toy programming languages, TINY and PAM, which were vehicles for the demonstration and justification of the concepts involved. In the case of synthesized-only attribute grammars, the lambda machine plays the role of a target machine for evaluation of semantic definitions. A lambda machine was built in Prolog for reducing (solving) lambda expressions. Also, it has been shown (elsewhere) that denotational semantic definitions can be mapped into synthesized-only attribute grammars [14]. Thus, this model provides a means through which denotational semantic definitions can be executed.

It was noted that the inverse relation between the subject program and its semantic definition holds as well. This provides a mechanism for execution of the inverse of the attribute grammar. This point, however, needs further research.

Acknowledgments

The author is grateful to David Martin and Stott Parker for their discussions and comments on this work.

Appendix A: Lambda machine in Prolog

```
/* solve(M, R)  reduces M to R where: M is  a lambda expression
                                      R will be the result of solving M
*/

op('\', rl, 250).
```

```
solve(M, R) ←
   capture(M, M1) &
   solve1(M1, R).

/* Some primitive instructions for this lambda machine */

solve1(negate(E), X) ←
   / &
   solve1(E, V1) &
   negate(V1, X).
solve1(eq(E1, E2), X) ←
   / &
   solve1(E1, V1) &
   solve1(E2, V2) &
   eq(V1, V2, X).
solve1(add(E1, E2), X) ←
   / &
   solve1(E1, V1) &
   solve1(E2, V2) &
   add(V1, V2, X).
solve1(sub(E1, E2),  X) ←
   / &
   solve1(E1, V1) &
   solve1(E2, V2) &
   diff(V1, V2, X).
solve1(mul(E1, E2),  X) ←
   / &
   solve1(E1, V1) &
   solve1(E2, V2) &
   prod(V1, V2, X).
solve1(div(E1, E2),  X) ←
   / &
   solve1(E1, V1) &
   solve1(E2, V2) &
   quot(V1, V2, X).
solve1(lookup(Z, I), X) ←
   / &
   lookup(Z, I,  X).
solve1(cond(L), X) ←
   / &
   cond(L, V) &
   solve1(V, X).

/* Solving lambda expression in applicative evaluation order */

solve1(apply(apply(A, B), C), R) ←
   / &
   solve1(apply(A, B), A1) &
   solve1(apply(A1, C), R).
solve1(apply(abstract(V, B), C), R) ←
   / &
   solve1(C, V) &
   capture(B, B1) &
   solve1(B1, R).
```

```
/* Fix point finding function for solving recursive definitions */

solve1(apply(mu(X, B), C), R) ←
   / &
   fresh(mu(X, B), mu(Y, B1)) &
   X = mu(Y, B1) &
   solve1(apply(B, C), R).

/* Can not simplify it any more */

solve1(X, X).

/* fresh(X, Y) generates Y such that it is identical to X except that
              all variables in X have been renamed.  For example:
              fresh(f(*1, g(*2, *1)), f(*3, g(*4, *3))).
*/

fresh(X, Y) ← fresh1(X, Y, nil, Z).

fresh1(X, Y, Asoc, Asoc) ← var(X) & pres(X, Asoc, Y) & /.
fresh1(X, Y, Asoc, (X-Y).Asoc) ← var(X) & /.
fresh1(A, A, Asoc, Asoc) ← atom(A) & /.
fresh1(I, I, Asoc, Asoc) ← int(I) & /.
fresh1(X.Y, Xr.Yr, Asoc, Asoc2) ←
   fresh1(X, Xr, Asoc, Asoc1) &
   fresh1(Y, Yr, Asoc1, Asoc2).
fresh1(X, Xr, Asoc, Asoc1)←
   cons(Name.Arglist, X) &
   fresh1(Arglist, Arglistr, Asoc, Asoc1) &
   cons(Name.Arglistr, Xr).

/* pres(X, Asoc, Y) Y is bound to X in Asoc */

pres(X, (X1-Y1).Asoc, Y1)←
   not(vneq(X, X1)) &
   /.
pres(X, (X1-Y1).Asoc, Y2)←
   pres(X, Asoc, Y2).

/* vneq(X, Y) true if both X and Y are distinct variables */

vneq(X, Y) ← not(not(vneq1(X, Y))).

vneq1(1, 2).

/* capture(E, E1) rewrites E to E1 such that no possibility of name
                  capturing can occur.  For example:
                  capture(abstract(*1, apply(abstract(*1, *1), *1)),
                         (abstract(*1, apply(abstract(*2, *2), *1))).
*/

capture(X, X) ← var(X) & /.
capture(apply(E1, E2), apply(E3, E4)) ←
   / &
   capture(E1, E3) &
   capture(E2, E4).
```

```
capture(abstract(S, E), E2) ←
   / &
   capture(E, E1) &
   rename(abstract(S, E1), S, E2).
capture(mu(X, E), E2) ←
   / &
   capture(E, E1) &
   rename(mu(X, E1), X, E2).
capture(X, X).

/* rename(E, X, X1, E1) rename variable X in E to X1, the result is E1 */

rename(E, X, E1) ←
   var(X) &
   / &
   rename1(E, X, X1, E1).
rename(E, H.T, E1) ←
   / &
   rename(E, H, E2) &
   rename(E2, T, E1).

rename1(Y, X, X1, X1) ← var(Y) & not(vneq(X, Y)) & /.
rename1(Y, X, X1, Y) ← var(Y) & /.
rename1(A, X, X1, A) ← atom(A) & /.
rename1(I, X, X1, I) ← int(I) & /.
rename1(H.T, X, X1, H1.T1) ←
   / &
   rename1(H, X, X1, H1) &
   rename1(T, X, X1, T1).
rename1(Y, X, X1, Y1) ←
   cons(Name.Arglist, Y) &
   rename1(Arglist, X, X1, Arglist1) &
   cons(Name.Arglist1, Y1).

/* Definition of primitive instructions for this lambda machine */

negate(true, false) ← /.
negate(false, true).

eq(V1, V2, true) ← eq(V1, V2) & /.
eq(V1, V2, false).

add(V1, V2, V3) ← sum(V1, V2, V3).

lookup(nil, Id, unbound) ← /.
lookup((V\Id).Zs, Id, V) ← /.
lookup((V1\Id1).Zs, Id, V) ← lookup(Zs, Id, V).

cond((true-C).Ls, C) ← /.
cond((B-C).Ls, C) ←
   solve1(B, true) &
   /.
cond((B-C1).Ls, C) ←
   cond(Ls, C).
```

Appendix B: Compiled semantics of TINY

```
command(assign(Id, E), abstract(S, apply(abstract(V.(Z.I.O),
                                                  ((V\Id).Z).I.O
                                                 ),
                                          apply(E1, S)
                                         )
                                 )
        ) ←
   expression(E, E1).

command(output(E), abstract(S, apply(abstract(V.(Z.I.O),
                                              Z.I.(V.O)
                                             ),
                                      apply(E1, S)
                                     )
                             )
        ) ←
   expression(E, E1).

command(
   if(E, C1, C2),
   abstract(S,
            apply(abstract(V1.S1,
                      cond((V1-apply(Rc1, S1)).(true-apply(Rc2, S1)).nil)
                           ),
                  apply(E1, S)
                 )
           )
        ) ←
   expression(E, E1) &
   command(C1, Rc1) &
   command(C2, Rc2).

command(while(E, C),
        mu(X,
           abstract(S,
                    apply(abstract(V1.S1,
                                   cond((V1-apply(X,
                                                  apply(Rc1,
                                                        S1)
                                                 )
                                        ).(true-S1).nil
                                       )
                                  ),
                          apply(E1, S)
                         )
                   )
        )
     ) ←
command(C, Rc1) &
expression(E,E1).
```

```
command(C1.C2, abstract(S,
                        apply(Rc2, apply(Rc1, S))
                       )
       ) ←
   command(C1, Rc1) &
   command(C2, Rc2).
expression(expr(0), abstract(S, 0.S)).
expression(expr(1), abstract(S, 1.S)).
expression(expr(true), abstract(S, true.S)).
expression(expr(false), abstract(S, false.S)).
expression(expr(read), abstract(Z.(Hi.Ti).O, Hi.(Z.Ti.O))).
expression(expr(Id),
           abstract(Z.I.O,
                    apply(abstract(Val,Val.(Z.I.O)),
                          lookup(Z, Id)
                         )
                   )
          ) ←
   atom(Id).
expression(expr(not(E)),
           abstract(S,
                    apply(abstract(V1.S1,
                                   apply(abstract(V2,
                                                  V2.S1
                                                 ),
                                         negate(V1)
                                        )
                                  ),
                          apply(E1, S)
                         )
                   )
          ) ←
   expression(E, E1).
expression(
   expr(Ea, eq, Eb),
   abstract(S,
            apply(abstract(V1.S1,
                           apply(abstract(V2.S2,
                                          apply(abstract(V3,
                                                         V3.S2
                                                        ),
                                                eq(V1, V2)
                                               )
                                         ),
                                 apply(E2, S1)
                                )
                          ),
                  apply(E1, S)
                 )
           )
          ) ←
   expression(Ea, E1) &
   expression(Eb, E2).
```

```
expression(
   expr(Ea, plus, Eb),
        abstract(S,
                 apply(abstract(V1.S1,
                                apply(abstract(V2.S2,
                                               apply(abstract(V3,
                                                              V3.S2
                                                             ),
                                                     add(V1, V2)
                                                    )
                                              ),
                                       apply(E2, S1)
                                      )
                               ),
                        apply(E1, S)
                       )
                )
         ) ←
   expression(Ea, E1) &
   expression(Eb, E2).
```

Appendix C: Compiled semantics of PAM

```
program(Series, Scode.s(halt, j).nil) ←
   series(Series, 0, 0, L, Scode).

series(S.Ss, Temp, Li, Lo, Code1.Codes) ←
   statement(S, Temp, Li, Lo1, Code1) &
   series(Ss, Temp, Lo1, Lo, Codes).
series(S, Temp, Li, Lo, Code) ← statement(S, Temp, Li, Lo, Code).

statement(read(Vs), Temp, Li, Li, Code) ← variables(Vs, get, Code).

variables(V.nil, Opcode, s(Opcode, V)).
variables(V.Vs, Opcode, s(Opcode, V).Code) ← variables(Vs, Opcode, Code).

statement(write(Vs), Temp, Li, Li, Code) ← variables(Vs, put, Code).
statement(assign(V, Expr), Temp, Li, Li, Ecode.s(sto, V)) ←
   expression(Expr, Temp, Ecode).
statement(if(Comp, Then, Else),
          Temp,
          Li,
          Lo,
          Ccode.Tcode.s(j, L2).s(L1, lab).Ecode.s(L2, lab)) ←
   sum(Li, 1, L1) &
   sum(Li, 2, L2) &
   comparison(Comp, Temp, L1, Ccode) &
   series(Then, Temp, L2, Lo1, Tcode) &
   series(Else, Temp, Lo1, Lo, Ecode).

statement(dloop(Expr, Series),
          Temp,
          Li,
          Lo,
```

```
Ec.s(sto, T1).s(L1, lab).s(sub, 1).s(jn, L2).s(sto, T1).Sc.s(j, L1).s(L2, lab
)
        ) ←
   sum(Temp, 1, T1) &
   sum(Li, 1, L1) &
   sum(Li, 2, L2) &
   expression(Expr, T1, Ec) &
   series(Series, T1, L2, Lo, Sc).

statement(iloop(Comp, Series),
          Temp,
          Li,
          Lo,
          s(L1, lab).Ccode.Scode.s(j, L1).s(L2, lab)) ←
   sum(Li, 1, L1) &
   sum(Li, 2, L2) &
   comparison(Comp, Temp,L2, Ccode) &
   series(Series, Temp, L2, Lo, Scode).

comparison(comp(Expr1, Rel, Expr2),
           Temp,
           Li,
           Ecode1.s(sto, T1).Ecode2.s(sub, T1).s(Opcode, Li)) ←
   sum(Temp, 1, T1) &
   expression(Expr1, T1, Ecode1) &
   expression(Expr2, T1, Ecode2) &
   relation(Rel, Opcode).

expression(expr(Term), Temp, Code) ← term(Term, Temp, Code).
expression(expr(Term1, Wop, Term2),
           Temp,
           Tc1.s(sto, T1).Tc2.s(sto, T2).s(load, T1).s(Opc, T2)) ←
   sum(Temp, 1, T1) &
   sum(Temp, 2, T2) &
   term(Term1, Temp, Tc1) &
   term(Term2, T2, Tc2) &
   weakoperator(Wop, Opc).

term(term(Elem), Temp, Code) ← element(Elem, Temp, Code).
term(term(Elem1, Sop, Elem2),
     Temp,
     Ec1.s(sto, T1).Ec2.s(sto, T2).s(load, T1).s(Opc, T2)) ←
   sum(Temp, 1, T1) &
   sum(Temp, 2, T2) &
   element(Elem1, Temp, Ec1) &
   element(Elem2, T2, Ec2) &
   strongoperator(Sop, Opc).

element(elem(constant(C)), Temp, s(load, num(C))).
element(elem(variable(V)), Temp, s(load, V)).
element(Expr, Temp, Code) ← expression(Expr, Temp, Code).
```

```
relation('eq', jnp).
relation('le', jn).
relation('gt', jnz).
relation('ge', jp).
relation('ne', jz).

weakoperator(plus, add).
weakoperator(minus, sub).

strongoperator(times, mul).
strongoperator(divide, div).
```

Appendix D: Sample semantic computation for PAM

The Pam program x := (x + y * z) + b * c will produce the following code:

```
        load   x.
        sto   1.
        load   y.
        sto   3.
        load   z.
        sto   4.
        load   3.
        mul   4.
        sto   2.
        load   1.
        add   2.
        sto   1.
        load   b.
        sto   3.
        load   c.
        sto   4.
        load   3.
        mul   4.
        sto   2.
        load   1.
        add   2.
        sto   x.
        halt   j.
```

The Pam program

```
  read k;
  while k > 0
     do
        k := k - 1
     end
```

will produce the following code:

```
        get   k.
    1    lab.
        load   k.
        sto   1.
        load   num(0).
        sub   1.
        jnz   2.
        load   k.
        sto   1.
        load   num(1).
        sto   2.
        load   1.
        sub   2.
        sto   k.
        j   1.
    2    lab.
        halt   j.
```

The Pam program

```
  read x,y;
  while x <> 99
    do
      ans := (x + 1) - (y /2);
      write ans;
      read x,y
  end
```

will produce the following code:

```
        get   x.
        get   y.
    1    lab.
        load   x.
        sto   1.
        load   num(99).
        sub   1.
        jz   2.
        load   x.
        sto   1.
        load   num(1).
        sto   2.
        load   1.
        add   2.
        sto   1.
        load   y.
        sto   3.
        load   num(2).
        sto   4.
        load   3.
        div   4.
        sto   2.
        load   1.
        sub   2.
        sto   ans.
        put   ans.
        get   x.
        get   y.
        j   1.
    2    lab.
        halt   j.
```

References

1. D. E. Knuth, "Semantics of Context-Free Languages," *Math. Syst. Theory* **2,** 127–145 (1968).
2. D. E. Knuth, "Semantics of Context-Free Languages: Correction," *Math Syst. Theory* **5,** 95–96 (1971).
3. W. T. Wilner, "Declarative Semantic Definition," *Report No. STAN-CS-233-71*, Department of Computer Science, Stanford University, CA, 1971.
4. T. A. Dreisbach, "A Declarative Semantics Definition of PL360," *Report No. UCLA-ENG-7289*, Computer Science Department, University of California, Los Angeles, 1972.
5. D. M. Berry, "On the Design and Specification of the Programming Language OREGANO," *Report No. UCLA-ENG-7388*, Computer Science Department, University of California, Los Angeles, 1974.
6. S. Gerhart, "Correctness Preserving Program Transformation," *Proceedings of the Second ACM Symposium on Principles of Programming Languages*, Palo Alto, CA, 1975, pp. 54–66.
7. D. Neel and M. Amirchahy, "Semantic Attributes and Improvement of Generated Code," *Proceedings of the ACM National Conference*, San Diego, CA, 1974, pp. 1–10.
8. P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns, "Attribute Translations," *J. Computer & Syst. Sci.* **9,** 279–307 (1974).
9. S. R. Petrick, "Semantic Interpretation in the REQUEST System," *Research Report RC-4457*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1973.
10. T. Katayama, "HFP: A Hierarchical and Functional Programming Based on Attribute Grammar," *Conference Record of the Fifth International Conference on Software Engineering*, San Diego, CA, 1981, pp. 343–352.
11. H. Ganzinger, K. Ripken, and R. Wilhelm, "MUG1—An Incremental Compiler-Compiler," *Proceedings of the ACM Annual Conference*, Houston, TX, 1976, pp. 415–418.
12. U. Kastens and E. Zimmerman, "GAC—A Generator Based on Attribute Grammars," Institut für Informatik II, Universität Karlsruhe, Karlsruhe, West Germany.
13. K. J. Raiha, M. Saarinen, E. Soisalon-Soininen, and M. Tienari, "The Compiler Writing System HLP," *Report No. A-1978-2*, Department of Computer Science, University of Helsinki, Finland, 1978.
14. L. M. Chirica and D. F. Martin, "An Order-Algebraic Definition of Knuthian Semantics," *Math. Syst. Theory* **13,** 1–27 (1979).
15. D. Warren, "Logic Programming for Compiler Writing," *Software Pract. & Exper.* **10,** 97–125 (1979).
16. K. Kennedy and S. K. Warren, "Automatic Generation of Efficient Evaluators for Attribute Grammars," *Proceedings of the Third ACM Symposium on Principles of Programming Languages*, Atlanta, GA, 1976, pp. 32–49.
17. T. Katayama, "Translation of Attribute Grammars into Procedures," *ACM Trans. Prog. Lang. & Syst.* **6,** 345–369 (1984).
18. M. Gordon, *Denotational Description of Programming Languages: An Introduction*, Springer-Verlag New York, 1979.
19. F. Pagan, *Formal Specification of Programming Languages: A Panoramic Primer*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.
20. P. Deransart and J. Maluszynski, "Relating Logic Programs and Attribute Grammars," *Logic Program.* **2,** No. 2, 119–155 (1985).
21. D. Yellin and E. M. Mueckstein, "The Automatic Inversion of Attribute Grammars," *Research Report RC-10957-49159*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1985.

Received October 28, 1985; accepted for publication January 6, 1986

Bijan Arbab *IBM Scientific Center, 11601 Wilshire Boulevard, Los Angeles, California 90025.* Mr. Arbab is a member of the Scientific Center staff working on the robotics project on problems regarding machine learning, planning, and logic programming. He worked for Teledyne Microelectronics on the automation of data acquisition and reduction from 1980 to 1983. Joining IBM in 1983, he worked part time on the robotics project; he began working full time on the project in July 1985. Mr. Arbab received his B.S. in mathematics in 1982 from Loyola Marymount University, Los Angeles, and his M.S. in computer science in 1984 from UCLA; he is currently a Ph.D. candidate in computer science at UCLA. Mr. Arbab is a co-author of papers in the area of expert systems and semantics of logic programs.