Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars

Rodney Farrow

Declarative Systems
277 W. Hedding
San Jose, CA 95110

ABSTRACT

In the traditional formulation of attribute grammars (AGs) circularities are not allowed, that is, no attribute-instance in any derivation tree may be defined in terms of itself. Elsewhere in mathematics and computing, though, circular (or recursive) definitions are commonplace, and even essential. Given appropriate constraints, recursive definitions are well-founded, and the least fixed-points they denote are computable. This is also the case for circular AGs.

This paper presents constraints on individual attributes and semantic functions of an AG that are sufficient to guarantee that a circular AG specifies a well-defined translation and that circularly-defined attribute-instances can be computed via successive approximation. AGs that satisfy these constraints are called finitely recursive.

An attribute evaluation paradigm is presented that incorporates successive approximation to evaluate circular attribute-instances, along with an algorithm to automatically construct such an evaluator. The attribute evaluators so produced are static in the sense that the order of evaluation at each production-instance in the derivation-tree is determined at the time that each translator is generated.

A final algorithm is presented that tells which individual attributes and functions must satisfy the constraints.

---

[1]This work was partially supported by the National Science Foundation under grant number DCR-8310930.

1 Introduction

In the traditional formulation of attribute grammars (AGs) circularities are not allowed, that is, no attribute-instance in any derivation tree may be defined in terms of itself. Elsewhere in mathematics and computing, though, circular (or recursive) definitions are commonplace, and even essential. Given appropriate constraints, recursive definitions are well-founded, and the least fixed-points they denote are computable. This is also the case for circular AGs: if they satisfy appropriate constraints then circular AGs describe well-defined translations, and translators for them can be automatically constructed.

This paper presents constraints on individual attributes and semantic functions of an AG that are sufficient to guarantee that a circular AG specifies a well-defined translation and that circularly-defined attribute-instances can be computed via successive approximation. Broadly speaking, we require that the affected attributes take on values from a complete partial order (c.p.o.) and that the affected semantic functions be monotonic and satisfy an ascending chain condition. Given these restrictions, circularly-defined attribute-instances can be interpreted as denoting the least fixed-point of their defining expressions and their values can be computed by successive approximation.

An attribute evaluation paradigm is presented that incorporates successive approximation to evaluate circular attribute-instances. The attribute evaluator is static in the sense that the order of evaluation at each production-instance in the derivation-tree is determined at the time that the translator is generated. Static evaluators incur less run-time overhead and are amenable to more optimizations than are dynamic evaluators. As well, an algorithm is presented that generates such an evaluator for a possibly-circular AG.

Even in a circular AG, not every attribute will give rise to circularly-defined instances, and it would be overly restrictive to require that every

attribute and semantic function satisfy the conditions proposed in order that circular definition be well-founded and finitely computable. Therefore, we also give an algorithm that determines which attributes and functions must satisfy these conditions.

Interest in circular AGs is usually engendered by one of two other research activities: consideration of the relationship between denotational semantics and AGs for specifying programming language semantics, or work on using AGs as the basis of compiler-compilers. Of course, these two activities need not be disjoint.

Many compiler-compilers based on AGs are under investigation [21, 12, 9, 7]. They use AGs to describe the syntax of a programming language, especially its context-sensitive aspects, and to describe how the source program should be translated by the compiler. Although much success has been achieved, not all aspects of compilation are easy to express in an AG. One impediment is that all recursion that is NOT based on the recursive context-free phrase structure of the source program must be encapsulated within a single, out-of-line semantic function. Examples of compilation tasks that are most easily expressed with broadly distributed recursion are data-flow analysis, and resolution of symbolic types and constants in a language that allows identifiers to be used before they are defined. We present solutions to these two problems formulated as circular, but well-defined, AGs.

The relationship between AGs and the mathematical foundations of programming language semantics was early noted by Chirica and Martin [4]. More recently, Mayoh [19] has discussed translating AGs to denotational semantics, Ganzinger has considered translating denotational semantics to AGs, and Paulson [20] has proposed a combination of AGs and denotational semantics that he calls semantic grammars. In both Ganzinger's and Paulson's work circularities are possible and useful. Although many of the circularities that so arise can be dealt with by the techniques we present, care should be exercised. In general, recursively-defined functions over an infinite domain, such as those used in the denotational semantics to represent the effect of programs, will not satisfy our restrictions and can not be computed by our successive approximation techniques.

2 The utility of circular AGs

Beginning with Knuth's seminal paper introducing AGs [18], circularity in an AG has usually been viewed as an error. Any input whose context-free phrase structure actually exhibited a circularly-defined attribute-instance would yield an undefined value as the translation and would cause problems for an attribute evaluator. However, recursive (i.e. circular) definitions are common elswhere in mathematics and computing, and many problems are naturally and traditionally described by recursive definitions and solved by recursive algorithms. Constraining AGs to be always non-circular has meant that:

1. the recursive aspects of a problem must be encapsulated completely within out-of-line functions, or

2. an AG-based specification of a problem must be more complicated than other kinds of descriptions of the problem which can use recursion, or

3. the problem is never formulated as an AG, or

4. all of the above.

To illustrate how useful circularity can be, and to show how the lack of it has caused the above-enumerated effects, this section discusses AG-based formulations of two problems from compiler design: LIVE variable analysis, and the definition of symbolic constants in a Pascal-like language that allows a symbol to be used before it is defined. These two problems are each representative of a large class of problems: data-flow analysis for program optimization, and symbol-table manipulation, respectively.

Figure 1 shows an AG-fragment for computing LIVE variable information. For the purpose of this example, variables are just identifiers, and the solution is, for each statement, the set of identifiers that are live on entry to that statement.[1] The values of instances of the stmt.LIVE attribute constitute this solution.

This AG is circular; the circularity is caused by the semantic functions associated with the production for WHILE-statements, where stmts.LIVE depends indirectly on stmts.OUT, which depends directly on stmts.LIVE. Although the formulas specified by this AG to compute the values of attribute-instances are potentially circular, the values are nonetheless well-defined - they are the least fixed-points of these recursive formulas. These least-fixed points are known to exist (and be effectively computable) because of properties of the "type" of the attributes (subsets of identifiers) and properties of the semantic functions of the AG (monotonicity).

---

[1] A variable X is live on entry to statement S if there is a control flow path from S to another point P in the program such that P uses the value of X and X is not redefined anywhere along the path from S to P.

```
attributes for <stmts> and <stmt>
  LIVE (synthesized) -
    set of variables (identifiers) that
    are "live" on entry to this stmt(s)
  OUT (inherited) -
    set of variables (identifiers) that
    are "live" on exit from this stmt(s)

attributes for <expr>
  IN (synthesized) -
    set of variables (identifiers) that
    are referenced within this expr


stmt  ::= ID "=" expr.
  stmt.LIVE =
    (stmt.OUT - (ID)) U expr.INSIDE

stmt  ::= "IF" expr "THEN" stmts1
                    "ELSE" stmts2 "END".
  stmt.LIVE =
    expr.IN U stmts1.LIVE U stmt2.LIVE,
  stmts1.OUT = stmt.OUT,
  stmts2.OUT = stmt.OUT

stmt  ::= "WHILE" expr "DO" stmts "END".
  stmt.LIVE =
    stmt.OUT U (expr.IN U stmts.LIVE),
  stmts.OUT =
    stmt.OUT U (expr.IN U stmts.LIVE)

stmts ::= stmt ";" stmts1.
  stmts.LIVE = stmt.LIVE,
  stmt.OUT = stmts1.LIVE,
  stmts1.OUT = stmts.OUT

stmts :: = .
  stmts.LIVE = stmts.OUT
```

Figure 1
A circular AG for LIVE analysis

For some time, the computation of many different kinds of data-flow information has been formulated as the least fixed-point of a set of recursive equations defined on the nodes of a directed graph (the control-flow graph of the program) [17, 11]. Indeed, this AG-fragment is merely the straight-forward translation of the standard graph-based formulas for LIVE-analysis given on page 489 of [1]. The AG is concise and easy to understand, but it is also circular.

A non-circular AG to compute the same information is presented in figure 2. This AG uses more attributes, semantic functions, and an additional operator on sets, intersection. The difference between this AG and the former one is quite illustrative of how a circular AG can be made non-circular. The circularity of the first AG occurred in the third production, where stmt.LIVE depends directly on stmts.LIVE, which depends indirectly on stmts.OUT, which depends directly on stmt.LIVE. This cycle is broken in the second AG by having stmt.LIVE depend on the new attribute stmts.INSIDE, rather than stmts.LIVE. In some sense, the stmts.INSIDE value is "as much as we need" of the stmts.LIVE value in order to compute stmt.LIVE. Furthermore, stmts.OUT isn't needed to compute stmts.INSIDE. Unfortunately, computing the INSIDE attributes duplicates much of the work of computing the LIVE attributes (all of which are still

```
attributes for <stmts> and <stmt>
  LIVE (synthesized) -
    set of variables (identifiers) that
    are "live" on entry to this stmt(s)
  OUT (inherited) -
    set of variables (identifiers) that
    are "live" on exit from this stmt(s)
  IN (synthesized) -
    set of variables (identifiers) that
    are referenced within this stmt(s)
    before they are redefined
  THRU (synthesized) -
    set of variables (identifiers) that
    are not redefined on some path
    through this stmt(s)

attributes for <expr>
  IN (synthesized) -
    set of variables (identifiers) that
    are referenced within this expr

I denotes the set of all identifiers.

stmt  ::= ID "=" expr.
  stmt.LIVE = (stmt.OUT-(ID)) U expr.IN,
  stmt.IN = expr.IN,
  stmt.THRU = I - (ID)

stmt ::= "IF" expr "THEN" stmts1
                    "ELSE" stmts2 "END".
  stmt.LIVE =
    expr.IN U stmts1.LIVE U stmts2.LIVE,
  stmts1.OUT = stmt.OUT,
  stmts2.OUT = stmt.OUT,
  stmt.IN =
    expr.IN U stmts1.IN U stmts2.IN,
  stmt.THRU = stmts1.THRU U stmts2.THRU

stmt ::= "WHILE" expr "DO" stmts "END".
  stmt.LIVE =
    expr.IN U stmt.OUT U stmts.LIVE,
  stmts.OUT =
    expr.IN U stmt.OUT U stmts.IN,
  stmt.IN = expr.IN U stmts.IN,
  stmt.THRU = I

stmts ::= stmt ";" stmts1.
  stmts.LIVE = stmt.LIVE,
  stmt.OUT = stmts1.LIVE,
  stmts1.OUT = stmts.OUT,
  stmts.THRU = stmt.THRU   stmts1.THRU,
  stmts.IN =
    stmt.IN U (stmts1.IN   stmt.THRU)

stmts :: = .
  stmts.LIVE = stmts.OUT,
  stmts.IN = emptySet,
  stmts.THRU = I
```

Figure 2
A non-circular AG for LIVE analysis

needed), and INSIDE depends on the THRU attributes, whose computation is still more redundant.

The second example of this section involves symbolic constants in a Pascal-like language. Pascal itself does not exhibit this problem, but let us consider a language that allows a reference to a symbolic constant to occur before its definition in the source text. Figure 3 shows a group of such definitions, and figure 4 shows an AG-fragment to compute a list of pairs, [ID,value], that gives the value for each defined symbolic constant. This is realized as the value of attribute scope.DEFS. Typically, this list of pairs would be used elsewhere in the compilation. Also computed is a list of error messages, flagging duplicate definitions of symbols and references to undefined names.

```
X = Y + 14;
CHI = X *3 + 2;
Y = Z + 3*CHI;
Z = 1;
```

Figure 3
An example of symbolic constants with use before definition

This AG is circular because the ENV attributes are needed to compute the expr.VAL attributes, which are needed to build the dcls.DEFS values, from which the ENV attributes originally get their value. If the language required that these symbols be defined before they are used (as does Pascal) then the expr.ENV value of any dcl could be just the dcls.DEFS value of all declarations to its left.

Like the LIVE variable example, this application can also be formulated as a non-circular AG, albeit at much higher cost in terms of increased complexity. Indeed, Knuth originally argued [18] that any computable function defined on the underlying context-free phrase structure of an AG could be formulated as a non-circular AG by encoding the parse-tree as an attribute of the goal symbol and then applying the computable function as a semantic function whose only argument was the encoded parse-tree. Our non-circular AG for the symbolic constants example has much of this flavor.

The AG is shown in figure 5. There are two essential differences between this AG and the previous, circular one. First, the ENV attributes are now truly functions (i.e. functional values) instead of being lists of pairs. Second, in order to define the particular functional value that will be assigned to these ENV attributes, new H, G, and F attributes have been introduced. These take on higher-order functional values, i.e. ones that map functions to functions. An H attribute has type tp_ENV ---> tp_ENV, a G attribute has type tp_ENV ---> int, and an F attribute has type int x int ---> int. Here tp_ENV is type ID ---> int. The semantic functions that define H attributes are actually written as an uncurried function returning an integer, i.e. of type (tp_ENV, ID) ---> integer. The value of the ENV attribute is the least fixed-point of an H value. This is denoted by the first production's semantic function, dcls.ENV = Y(dcls.H). Of course, this is just the notation necessary to describe the usual implementation of a directly recursive procedure.

The functional values taken on by instances of H, G, and F attributes are nearly as complicated as the APT itself, and evaluation of an instance of ENV is comparable to traversing a part of the APT. Here we have indeed made the AG much more complicated in order to forstall circularity. Furthermore, we have had to introduce a very powerful semantic function, the paradoxical combinator, Y. The following sections describe what we think is an attractive alternative to altogether banning circlarity.

3 Recursive AGs and finitely recursive AGs

In the examples of the previous section, the meaning we wanted to assign to circularly-defined attribute-instances was the least fixed-point of their defining formulas. In order to precisely formulate this we adopt Chirica and Martin's [4] definition of the semantics of an AG. Although the algebraic details of their definition are beyond the scope of this paper, the basic idea is straight-forward.

Loosely speaking, this idea is that, for each input string S, the semantic functions of an AG comprise a mapping from one APT for S to another APT for S with the same context-free phrase-structure. The only differences between the two APTs are the values of their attribute-instances. This induced mapping is depicted in figure 6. The value of the AG's translation on S is then that APT which is the least fixed point of the mapping between APTs for S, or more precisely, the values of the attribute-instances of the root of that APT.

This definition of an AG's semantics agrees precisely with the traditional semantics if no attribute-instance is circularly defined. If some attribute-instances are circularly-defined then these get defined with the least fixed-point of their defining formulas; non-circular attribute-instances are still defined as usual. Under this definition an AG, even if it is circular, is well-defined so long as, for any input S, the mapping between APTs for S (described above) has a least fixed-point. If this is true then call this AG a recursive attribute grammar.

```
attributes for <dcls>
  DEFS (synthesized) list of [ID,int] pairs
  ENV  (inherited)   list of [ID,int] pairs
  ERRS (synthesized) list of error messages

attributes for <dcl>
  NAME (synthesized) an identifier
  PAIR (synthesized) an [ID,int] pair
  ENV  (inherited)   list of [ID,int] pairs
  ERRS (synthesized) list of error messages

attributes for <expr> and <term>
  VAL  (synthesized) an integer that is the value of this construct
  ENV  (inherited)   list of [ID,int] pairs
  ERRS (synthesized) list of error messages


scope ::= ... dcls ...
  scope.DEFS = dcls.DEFS,
  scope.ERRS = dcls.ERRS,
  dcls.ENV = dcls.DEFS


dcls ::= dcls1 dcl.
  dcls.DEFS, dcls.ERRS =
    if lookup(dcl.NAME, dcls1.DEFS)
            <> bottom            then
      dcls1.DEFS,
      mergeMsgs( dcls1.ERRS,
      mergeMsgs(   dcl.ERRS,
          MsgOf("%n is multiply defined",
            dcl.NAME
              )       ))
    else
      cons(dcl.PAIR, dcls1.DEFS),
      mergeMsgs( dcls1.ERRS, dcl.ERRS),
    fi,
  dcls1.ENV, dcl.ENV = dcls.ENV


expr ::= expr1 op term.
  expr.VAL = op.VAL,
  op.L_VAL = expr1.VAL,
  op.R_VAL = term.VAL,
  expr1.ENV, term.ENV = expr.ENV,
  expr.ERRS = merge(expr1.ERRS, term.ERRS)


op ::= "+".
  op.VAL = if (op.L_VAL = bottom)
                    or
             (op.R_VAL = bottom)
           then      bottom
           else  L_VAL + R_VAL
          fi


op ::= "*".
  op.VAL = if (op.L_VAL = bottom)
                    or
             (op.R_VAL = bottom)
           then      bottom
           else  L_VAL * R_VAL
          fi


term ::= ID.
  term.DEFS = lookup(ID, term.ENV),
  term.ERRS =
    if term.DEFS = bottom then
      MsgOf("unknown identifier %n", ID)
    else
      nullList
    fi


dcls ::= .
  dcls.DEFS = nullList,
  dcls.ERRS = nullList


dcl ::=  ID "=" expr ";".
  dcl.PAIR = pairOf(ID, expr.VAL),
  dcl.NAME = ID,
  expr.ENV = dcl.ENV


expr ::= term.
  expr.VAL  = term.VAL,
  expr.ERRS = term.ERRS,
  term.ENV  = expr.ENV


op ::= "-".
  op.VAL = if (op.L_VAL = bottom)
                    or
             (op.R_VAL = bottom)
           then      bottom
           else  L_VAL - R_VAL
          fi


op ::= "/".
  op.VAL = if (op.L_VAL = bottom)
                    or
             (op.R_VAL = bottom)
           then      bottom
           else  L_VAL / R_VAL
          fi


term ::= DIGITS.
  term.VAL = DIGITS.VAL,
  term.ERRS = nullList
```

Figure 4: A circular AG for symbolic constants with "use before definition"

```
scope ::= ... dcls ...
  scope.DEFS = dcls.DEFS,
  scope.ERRS = dcls.ERRS,
  dcls.ENV   = Y ( dcls.H )
```

```
dcls ::= .
  dcls.DEFS = nullList,
  dcls.ERRS = nullList,
  dcls.H =
    (E:tp_ENV,ID:tp_ID) int: bottom
```

```
dcls ::= dcls1 dcl.
  dcls.DEFS, dcls.ERRS =
    if lookup(dcl.NAME, dcls1.DEFS)
            <> bottom            then
      dcls1.DEFS,
      mergeMsgs( dcls1.ERRS,
      mergeMsgs(   dcl.ERRS,
          MsgOf("%n is multiply defined",
            dcl.NAME
              )       ))
    else
      cons(dcl.PAIR, dcls1.DEFS),
      mergeMsgs( dcls1.ERRS, dcl.ERRS),
    fi,
  dcls1.ENV, dcl.ENV = dcls.ENV,
  dcls.H =
    (E:tp_ENV,ID:tp_ID) integer:
      if ID = dcl.NAME
        then dcl.G(E)
        else dcls1.H(E,ID)
      fi
```

```
dcl ::=  ID "=" expr ";".
  dcl.PAIR = pairOf(ID, dcl.ENV(ID)),
  dcl.NAME = ID,
  expr.ENV = dcl.ENV
  dcl.G    = expr.G
```

```
expr ::= expr1 op term.
  expr1.ENV, term.ENV = expr.ENV,
  expr.ERRS =
    merge(expr1.ERRS, term.ERRS),
  expr.G =
    (E:tp_ENV) int: op.F(expr1.G(E),term.G(E))
```

```
expr ::= term.
  term.ENV  = expr.ENV,
  expr.ERRS = term.ERRS,
  expr.G    = term.G
```

```
op ::= "+".
  op.F =
    (L_VAL:int, R_VAL:int) int:
      if (L_VAL = bottom)
                or
         (R_VAL = bottom)
        then      bottom
        else  L_VAL + R_VAL
      fi
```

```
op ::= "-".
   op.F =
     (L_VAL:int, R_VAL:int) int:
       if (L_VAL = bottom)
                 or
          (R_VAL = bottom)
         then      bottom
         else  L_VAL - R_VAL
       fi
```

```
op ::= "*".
  op.F =
    (L_VAL:int, R_VAL:int) int:
      if (L_VAL = bottom)
                or
         (R_VAL = bottom)
        then      bottom
        else  L_VAL * R_VAL
      fi
```

```
op ::= "/".
  op.F =
    (L_VAL:int, R_VAL:int) int:
      if (L_VAL = bottom)
                or
         (R_VAL = bottom)
        then      bottom
        else  L_VAL / R_VAL
      fi
```

```
term ::= ID.
  term.ERRS =
    if term.ENV(ID) = bottom then
      MsgOf("unknown identifier %n", ID)
    else
      nullList
    fi,
  term.G = (E:tp_ENV) int: E(ID)
```

```
term ::= DIGITS.
  term.ERRS = nullList,
  term.G =
    (E:tp_ENV) int: DIGITS.VAL
```

Figure 5: A non-circular AG for symbolic constants with "use before definition"

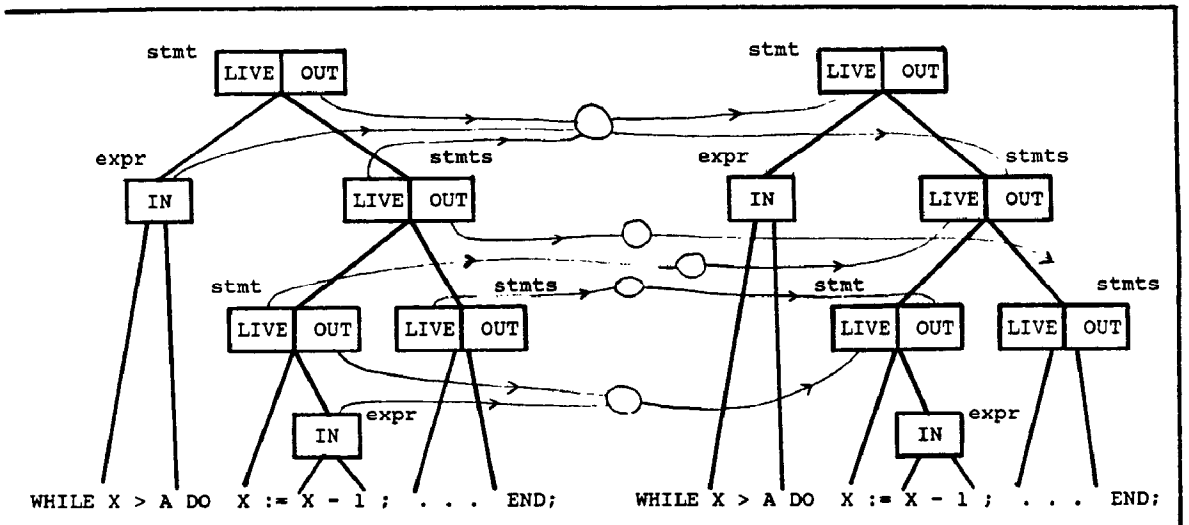

Figure 6: the mapping among APTs induced by the semantic functions of the AG of figure 1.

This approach is sufficient for specifying the semantics of circular AGs, but it does not tell us when the specified translation is effectively computable, or how to go about computing it. In this paper we propose that fixed-points be computed by succesive approximation and that constraints be placed on certain attributes and semantic functions sufficient to guarantee that the fixed-point is effectively computable by succesive approximation. In the remainder of this section we present a paradigm for attribute evaluation that deals with circularaly-defined attribute-instances by using successive approximation to compute their fixed-points. Also presented are the constraints we propose in order to guarantee that these attribute evaluators are correct and will terminate.

Our treatment of circularly-defined attribute-instances is a special case of solving the equation $X = f(X)$ for the value of X. If X takes its value in a CPO and if f is a continuous function from CPO to CPO, then the solution is the least fixed-point of f and can be expressed as: $X = \bot \cup f(\bot) \cup f(f(\bot)) \cup \ldots \cup f^i(\bot) \cup \ldots$ Here $\bot$ denotes the least, or "bottom", element of CPO, and $\cup$ denotes its least-upper-bound, or "join" operator.

The ith partial sum of this "infinite" expression is the first i terms of it and is an approximation to the value. The ith partial sum can be computed by the program in figure 7a. If it happens that $f^i(\bot) = f^{i+1}(\bot)$ then the sequence of partial sums converges after i terms to the complete solution of $X = f(X)$. This is computed by the program in figure 7b. Note that this will not terminate unless the "infinite" expression for X converges after finitely many terms.

Consider now a semantic tree for a particular AG and some source program. The nodes of this tree are labeled with symbols from the grammar and these nodes consist of fields that are the attribute-instances. Kennedy and Ramanathan [15] describe a dynamic attribute evaluator that builds a directed graph whose nodes are the attribute-instances and whose edges are the dependencies among the attribute-instances induced by the semantic functions of the AG. For a non-circular AG this graph must be acyclic and the attribute-instances can be evaluated in any order that is consistent with the partial order of the directed graph. A circularly-defined attribute-instance would be part of a cycle in the directed graph, and each attribute-instance on the cycle is circularly defined. We propose computing these attribute-instances as follows:[2]

```
initialize all attribute-instances
  in the cycle to ⊥,

REPEAT
  FOR EACH   X, in the cycle
    recompute the value of X, using
      the current value of any
      attribute-instances upon which
      X depends
    assign this new value to X
  ENDFOR
UNTIL no recomputation changed the
      value of any X
```

In order that this process terminate and accurately compute the least fixed-point of a set of recursively defined

[2] A similar paradigm for computing circularly-defined attribute-instances in an incremental system for VLSI design was recently presented in [10]. The constraints they imposed on attributes and semantic functions were stricter than those proposed here.

```
X := ⊥;
j := 0;
while j < i do
  X := f(Y);
  j := j+1;
end;
```

Figure 7a

```
X := ⊥;
repeat
  Y := X;
  X := f(Y);
until X = Y;
```

Figure 7b

attribute-instances we propose that the attributes and semantic functions satisfy certain conditions:

1. every node in a dependency cycle must represent an instance of an attribute whose domain is a c.p.o. in which it is possible to test pairs of elements for equality, and

2. every edge in a dependency cycle must represent a monotonic function such that for any sequence of domain elements, s[0] < s[1] < s[2] < ... the corresponding sequence of co-domain elements, f(s[0]) < f(s[1]) < ... ... < f(s[k]) = f(s[k+1]) becomes stationary. (ascending chain condition)

The first condition says that the test for loop termination can be done; the second condition says that the loop must terminate after finitely many iterations. Any AG that satisfies these conditions shall hereinafter be refered to as a finitely recursive AG.

Theorem 1: If AG is a finitely recursive AG, and S is a string in the domain of the translation described by AG, then the attribute evaluator described above terminates and correctly computes the translation of S given by AG.

Proof - is omitted.

These constraints we have imposed will exclude many recursive AGs. There are three reasons that a recursive AG could fail to be finitely recursive:

1. some affected attribute domain is not a c.p.o., or

2. in some affected attribute domain, equality is not computable, or

3. some affected semantic function is not monotone or does not satisfy the ascending chain condition.

Note that the second of these conditions is not as pathological as one might think; the computable functions whose domains are not finite can not be tested for equality. Hence if the Y combinator is replaced by the identity function in the example of figure 5 then the result is recursive but not finitely recursive. Successive approximation is not a sufficiently strong technique to implement recursive functions over an infinite domain.

The circular AGs of figures 1 and 4 are finitely recursive, if care is taken in defining the types of the attributes. These attribute values can not be "a subset of the set of all identifiers", or "a function from the set of all identifiers to the integers" because that may be an infinite set or function (depending on whether the length of an identifier is bounded). However, if the types of these attributes are chosen as "finite subsets of the set of all identifiers", and "functions with finite non-trivial support from the set of all identifiers to the integers" then the constraints are satisfied even if there is no bound on the number of identifiers.

Let us consider briefly how this evaluation paradigm for circular AGs would work for the AG of figure 4. The cycles in the dependency graph for some input will include the ENV, DEFS and VAL attributes for all APT nodes that contain an instance of production [term ::= ID] in their sub-tree. However, the VAL and PAIR attributes of any dcl whose defining expression does not contain a symbolic reference will not be in a cycle. These values will be computed first, and then the circularly-defined attribute-instances will be computed by successive approximation. All ENV attribute-instances, and affected DEFS attribute-instances will be initialized to the nullList, and all affected VAL attribute-instances will be initialized to bottom. This completes the initialization of the cyclic components of the APT.

Next, DEFS attribute-instances will be recomputed to construct a list of pairs that contains an entry for every declared identifier; if there is no symbolic reference in the defining expression for ID then its value is included in the DEFS list, otherwise its value is bottom. The next iteration through the cyclic component will cause to become defined (with a value other than bottom) the DEFS entries for those identifiers whose only symbolic references were already defined (i.e. not bottom) in the value for DEFS from the previous iteration. This process continues until no more identifiers become defined during some iteration.

Finally, the ERRS attribute-instances, which don't belong to any cycle, are evaluated. Their evaluation uses the least fixed-points just computed for appropriate instances of ENV and VAL attributes.

Note that there are two reasons why a declared identifier, ID, might still have bottom as its value after attribute evaluation is finished:

1. the expression defining ID might contain a reference to an identifier, X, that is never declared in the list, or

2. there could be a pair of declarations of the form: X = 3 + ID; ID = X - 4.

4 A static evaluator for recursive AGs

There are two drawbacks to the evaluation paradigm presented in the previous section:

1. it is a dynamic evaluation paradigm rather than a static one, and

2. it does not provide a way to verify that the AG itself is well-formed, i.e. that it can't generate an APT for which the evaluator will not terminate.

Both of these objections are resolved by the following evaluation strategy, which is a modification of an evaluation paradigm proposed independently by Katayama [14] and Courcelle and Franchi-Zannettacci [5]. This evaluator will herein be called the synth-function evaluator because it is organized as separate recursive functions for each synthesized attribute of the AG.

The synth-function evaluator evaluator consists of a collection of mutually recursive functions, R-(X.S); one for each synthesized attribute X.S in the AG. Each R-(X.S) computes the value of an instance of synthesized attribute X.S and takes as its input arguments:

- a sub-parse-tree whose root is an X-type node, and

- the values of all those instances of inherited attributes of X upon which X.S may depend, as determined by the IO-graph for X.

Each R-(X.S) procedure consists of a CASE statement with one clause for each production whose left-part is X. This clause is determined by taking the augmented dependency graph for the production, considering only those nodes and edges that are needed to compute X.S, topologically sorting this subgraph, and replacing each node in the resulting linear order with the code necessary to compute its value. If the node represents either a synthesized left-part attribute-occurrence, or an inherited right-part attribute-occurrence then this code is just the invocation of its defining semantic function. If the node represents a synthesized, right-part attribute-occurrence, Y.A, then this code is an invocation of R-(Y.A).

A synth-function evaluator can be built only if the augmented dependency graph for every production is acyclic. This condition is known as absolute non-circularity [16]. There are AGs that are well-defined and yet not absolutely non-circular. This is because some spurious edges can be introduced by the construction of the IO-graphs; edges that could never acutally appear in any sub-tree whose root is labelled with the corresponding non-terminal symbol. In [14] an algorithm is presented that shows how to build an absolutely non-circular AG that is equivalent to any non-circular AG. In [6, 8] it is shown how to build static attribute evaluators for non-circular AGs even if they are NOT absolutely non-circular, without going through the intermediate step of constructing a new AG. The central point of both these constructions is to arrange that the IO-graph of every non-terminal, when used to build an evaluator, has no spurious edges. We assume that this property holds in the following discussion.

The synth-function evaluator is modified so that it also constructs least fixed-points for circularly defined attribute-instances via successive approximation. The modified evaluator is called the recursive synth-function evaluator. For this we must know which attribute-occurrences are circularly-defined. This is determine by using the augmented dependency graph: the production's dependency graph, Dp, augmented with the IO graphs for the right-part non-terminals, Dp[IO(p[1]),...,IO(p[np])]. Circularities are manifested as cycles in the augmented dependency graph for some production. For us these cycles indicate recursively defined attributes. As such, the semantic functions that are represented by the edges of these cycles must be monotonic functions satisfying the ascending chain condition, and the attributes whose occurrences are represented by the nodes on these cycles must take values from a c.p.o. in which we can test for equality.

The nodes of each augmented dependency graph can be partitioned into circular dependency classes (CDCs). Nodes X and Y belong to the same CDC iff X depends on Y and Y depends on X. This corresponds to partitioning the nodes of the augmented dependency graph into strongly connected regions. An attribute-occurrence that is NOT recursively defined will be in a CDC by itself. The CDC that contains attribute-occurrence X will be denoted by [X]. Note that X and Y are in the same CDC iff [X] = [Y]. Furthermore, we say that an edge (X,Y) belongs to CDC [Z] iff [X] = [Y] = [Z].

An acyclic graph for a production is created in which each node represents a CDC, and there is an edge from [X] to [Y] iff there are attribute-occurrences X in [X] and Y in [Y] such that the original augmented dependency graph contains an

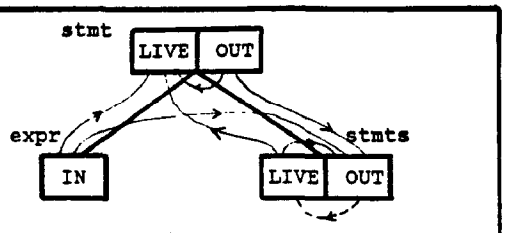

Figure 8a
The augmented dependency graph for the WHILE-production of figure 1.

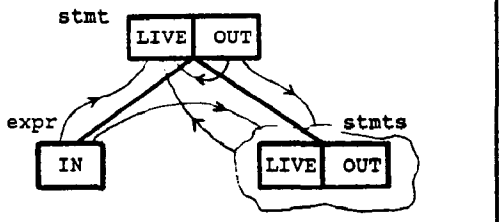

Figure 8b
Dp[...]/CDC for the above augemented dependecy graph

edge from X to Y. Denote this graph by Dp[...]/CDC. Figure 8 shows the [circular] augmented dependency graph for the WHILE-stmt production from the AG of figure 1, the circular dependency classes in this production, and the quotient graph Dp[...]/CDC.

The attribute-occurrences of these CDCs will be evaluated in an order compatible with the partial order given by Dp[...]/CDC, in the sense that no attribute-occurrence in [X] will be computed before any attribute-occurrence in [Y] if there is a path from [X] to [Y] in Dp[...]/CDC. If CDC [X] is a singleton class then the code to evaluate X is the same in the recursive synth-function evaluator as it is in the original: if X is defined in this production then execute its defining semantic function, otherwise call the appropriate synth-function procedure for X. However, if [X] contains two or more attribute-occurrences then these are recursively defined and the code to evaluate them constructs their least fixed-point by successive approximation. The model for this code is:

```
FOR EACH Y in [X] DO Y := bottom; END;

REPEAT
  STOP := true;
  FOR EACH Y in [X] DO
     TMP_Y := EVAL(Y);
     if TMP_Y = Y then
       STOP = false
     end;
     Y := TMP_Y;
  END;
UNTIL STOP;
```

```
R-(stmt.LIVE)(TREE, OUT) =
 case rootOf(TREE) of
  [stmt ::= ID "=" expr]:
   return(
    (OUT - setOf(firstSonOf(TREE))) U
     R-(expr.INSIDE)(secondSonOf(TREE))
         );

  [stmt ::= "IF" expr "THEN" stmts1
                     "ELSE" stmts2 "END"]
   return(
     R-(expr.INSIDE)(firstSonOf(TREE))
   U R-(stmts.LIVE)(secondSonOf(TREE),OUT)
   U R-(stmts.LIVE)( thirdSonOf(TREE),OUT)
         );

  [stmt ::= "WHILE" expr "DO" stmts "END"]
   expr_IN = R-(expr.IN)(firstSonOf(TREE))

   stmts_OUT  = emptySet;
   stmts_LIVE = emptySet;
   repeat
     STOP = true;

     TMP_OUT = OUT U expr_IN U stmts_LIVE;
     if stmts_OUT <> TMP_OUT then
       STOP = false;
     end;
     stmts_OUT = TMP_OUT;

     TMP_LIVE =
       R-(stmts.LIVE)(
               secondSonOf(TREE),
               stmts_OUT
                    );
     if stmts_LIVE <> TMP_LIVE then
       STOP = false
     end;
     stmts_LIVE = TMP_LIVE;
   until STOP;

   return(
      OUT U expr_IN U stmts_LIVE
        );
 esac


R-(stmts.LIVE)(TREE, OUT) =
 case rootOf(TREE) of
  [stmts ::= ] :
   return( OUT );

  [stmts ::= stmt ";" stmts1]:
   stmts1_LIVE =
     R-(stmts.LIVE)(TREE, OUT);
   return(
     R-(stmt.LIVE)(TREE, stmts1_LIVE)
         );
 esac
```

Figure 9
The recursive synth-function evaluator for the AG of figure 1.

Here, EVAL(Y) is the code needed to evaluate Y and produce its value by executing a semantic function or visiting a right-part sub-tree. The recursive synth-function evaluator for the AG of figure 1 is shown in figure 9.

Theorem 2: For any AG, the evaluator-construction algorithm described above terminates. Furthermore, if the input AG is also finitely recursive then a recursive synth-function evaluator is generated that, when applied to any string S in the domain of the translation described by AG, will terminate and correctly compute the translation of S as specified by AG.

Proof - is omitted.

The final result of this section is an algorithm that determines whether or not an AG is finitely recursive. This algorithm consists of two parts. One part examines the dependencies of the grammar and produces:

1. the set of attributes, some instance of which can participate in a circularity in some APT, and

2. the set of semantic functions that can participate in a circularity in some APT.

The second part of the algorithm is to ascertain that each attribute in the first set takes values in a c.p.o. for which equality is a computable function, and that each function in the second set is monotonic and satisfies the ascending chain condition.

The algorithm that finds the sets of attributes and semantic functions that participate in circularities is called the Marking algorithm. It uses successive approximation to compute the least fixed-point of a function, MARK, whose domain (and co-domain) is a triple: set of attributes, set of semantic functions, and set of IO-graph edges. The attribute, and semantic function component of the least fixed-point of MARK are the "marked" attributes and semantic functions.

The MARK function is defined in terms of two other functions, DIRECT_MARK and INDIRECT_MARK. DIRECT_MARK takes a production as argument and produces a (attributes, functions, IO-graph edges) triple. The attributes component of DIRECT_MARK(p) is all attributes that lie in a CDC of the augmented dependency graph for p; the semantic functions component is all semantic functions that are represented by an edge in a CDC; the IO-graph edges component is all augmenting IO-graph edges that are in a CDC.

INDIRECT_MARK takes as arguments both a production, p, and an edge A ---> B of the IO-graph for the left-part symbol of p. It produces a (attributes, functions, IO-graph edges) triple that consist of all attributes, semantic functions, and right-part, augmenting IO-graph edges, respectively, that lie on any path from the left-part occurrence of A to the left-part occurrence of B in the augmented dependency graph for p.

The MARK function is shown below in a high-level pseudo-code. The syntax, "op forEvery X in FOO: <expr> endEvery" denotes the value expressed by "<expr>/X1 op <expr>/X2 op...op <expr>/Xn" where FOO = {X1, X2, ..., Xn), "op" is any associative, dyadic operator (such as set union), and "<expr>/Xi" denotes the value produced by evaluating <expr> with Xi substituted for free-variable X. Below, the operator "union3" denotes the component-wise set union operation applied to pairs of (attributes, functions, IO-graph edges) triples, and "PRODS" is the set of all productions in the grammar.

```
MARK(attribs, functs, edges) =
  union3
  forEvery p in PRODS:
    union3(
        DIRECT_MARK(p),

        union3
        forEvery E in edges:
          INDIRECT_MARK(p,E)
        endEvery
          )
  endEvery
```

The details of implementing successive approximation in general have been discussed at length in previous sections and will not be repeated here, except to note that a more efficient implementation than that of the general approach is possible due to special knowledge of the function MARK; the complexity of this algorithm need be no more than O(m), where m is the sum of the number of marked attributes, semantic functions and IO-graph edges. Similarly, the proof of termination and correctness of the algorithm rests on establishing certain algebraic properties of MARK and its domain; these are left as an exercise for the reader.

Theorem 3: Suppose that ATTRIBS and FUNCTS are the set of attributes and semantic functions, respectively, marked by the Marking algorithm applied to some AG. Then AG is finitely recursive iff:

1. every attribute in ATTRIBS takes values in a c.p.o. that allows effective testing of equality, and

2. every F in FUNCTS is monotonic and satisfies the ascending chain condition.

Proof - that the two conditions imply that AG is finitely recursive is omitted. That any finitely recursive AG must

satisfy these conditions follows from the assumption, stated earlier in this section, that no IO-graph has any spurious edges.

We do not propose an automatic algorithm to decide whether an attribute takes values in a c.p.o. that allows testing for equality, or whether a given semantic function satisfies the ascending chain condition. Rather, the AG author should decide this, facilitated, perhaps, by some extra input syntax that lets the author tell the evaluator-generator (and any later readers of the AG) which attributes and semantic functions are appropriately constrained. The evaluator-generator could signal an error if some marked attribute or semantic function was not in the respective list. Alternatively, the evaluator-generator could produce a list of attributes and semantic functions that should be constrained and leave it to the AG author to ascertain that the constraints are satisfied. The former strategy seems less error-prone and more maintainable.

Those cycles in the augmented dependency graph that do NOT represent recursively-defined attributes, but instead indicate an ill-formed grammar, can be identified as such because, either some of the semantic functions involved will not satisfy the ascending chain condition, or some of the affected attribute domains will not be c.p.o.s in which we can test for equality.

5 Relation to other research

Other researchers have been interested in using AGs to solve problems that are usually expressed as fixed-points. The most common such problems are data-flow analysis computation for optimizing compilers. The usual solutions involve some mechanism for specifying the re-evaluation of certain attributes. Attribute re-evaluation has been discussed by Schulz [22], Babich and Jazayeri [3], and Skedzelski [23]. The MUG2 system [9] has a special notation for describing the re-evaluation of all attributes defined during one "sweep" of the evaluator.

Our approach allows an AG to describe solutions to problems that were previously solved by ad hoc techniques for attribute re-evaluation. There are many reasons to eschew such ad hoc techniques, among them that the correctness and termination of algorithms for iterative attribute re-evaluation must be established for each instance of circularly-defined attributes.

Another reason to avoid ad hoc attribute re-evaluation is to preserve the non-procedural nature of AGs. The ad hoc techniques proposed all involve some form of user specification of how attribute-instances are to be re-evaluated. This tends to restrict the attribute evaluation strategies that can be used, as well as require that the AG author completely understand the global consequences of her local attribute dependencies. It is much harder to design a correct evaluator for an AG than it is to write an AG, just as it is harder to build an LR(1) parser for a language than it is to write a context-free grammar for it.

By formulating the definitions of finitely recursive AGs and giving algorithms that check whether an AG is finitely recursive and that generate attribute evaluators for them, we have removed these burdens from the user and placed them upon the type-system of the attributes and semantic functions. It is in this realm that questions of well-definedness and convergence of successive approximations must ultimately be settled.

In a different vein, there has been some recent interest in using AGs to describe the translation from a program to the recursive function over a continuous c.p.o. that represents its effect (i.e. its denotational semantics). The work of Paulson [20] is most representative and advanced in this area. Although recursive definitions are clearly necessary, Paulson's translator did not allow circular AGs. Instead, least fixed-points were explicitly denoted where needed. As mentioned earlier, successive approximation would not work in this case; what's needed is some form of the paradoxical combinater, Y, to build the usual implementation of a recursive function that "unravels" at execution. Recently Arbab [2] has investigated writing a denotational semantics as an AG, detecting circularities and inserting a form of Y operator to compute the least fixed-point of the circularly defined functions. However, he discusses doing this only in those contexts where the circularity is manifested in the semantic functions of a single production, and he does not consider whether the least fixed-point computation is well-founded.

6 Some directions for further research

The fixed-point finding evaluators introduced here were adaptations of the synth-function evaluator, but most AG-based translator-writing systems use other, less flexible, more optimizable evaluation strategies (e.g. ordered evaluators [13] or protocol evaluators [6]). The important point in using the synth-function evaluator was to isolate the circularly defined attribute-occurrences so that they could be re-evaluated during a VISIT without re-evaluating other, [non-circular] attribute-occurrences. We suspect that "protocol propagation", introduced in [6], could be used to build protocols in which all circularly-defined attributes are grouped together in their own elements of the protocols. This would give rise to protocol, sub-protocol, or even ordered

evaluators that could be used for recursive attribute evaluation.

Another issue concerns the efficiency of successive approximation, as herein outlined, for computing least fixed-points. When recursively building non-atomic data structures, a naive implementation of successive approximation can be quite expensive. Consider the list of [ID,value] pairs in the AG of figure 4. Here the list of pairs must be completely rebuilt for each iteration, rather than just being updated during each iteration. Ideally, we would like to have the evaluator-generator discover appropriate optimizations, but that is a hard problem, variations of which lie at the heart of several aspects of automatic programming. A less ambitious approach is to devise alternate formulations of the recursive AG that do not exhibit such performance penalties, or to devise ways of telling the evaluator-generator about possible optimizations.[3]

Finally, the Marking algorithm for finding attributes and semantic functions that participate in circularities suggests an approach to testing an AG for circularity when it is NOT absolutely non-circular. What we hope for is an algorithm that starts with the usual test for absolute non-circularity and then gracefully degrades when it discovers a cycle in some augmented dependency graph. We want to avoid the scenario wherein, if the absolute non-circularity test fails then we start over running the expensive, complete circularity test on the whole grammar without being able to reuse any of the information already gathered.

7 Conclusions

We have argued that the recursive specification of attributes is a valuable, general-purpose technique that can be profitably used in many AGs. Often, recursive specification is both simpler for the AG author to formulate and easier to understand for the [non-expert] reader of the AG. Recursion often exists in AGs, but it is hidden within the semantic functions. This unduly complicates both the semantic functions, and the attribute domains upon which they are defined, as is evidenced by a comparison between the AGs of figure 4 and figure 5.

The classes of recursive and finitely recursive AGs were defined. A general, dynamic attribute evaluator was presented that computes the least fixed-point for circular attribute-instances via successive approximation. An algorithm was presented that constructs a static attribute evaluator for each finitely recursive AG. Another algorithm was presented that reduces the question of whether an AG is finitely recursive to the question(s) of whether certain attributes and semantic functions individually obey a simple constraint.

One of the major strengths of AGs is that they are non-procedural specifications, rather than very high-level programming languages. Recursive definition of attributes is even more of a specification: it describes conditions that the attributes must satisfy. The author of a recursive AG should not have to supply algorithms for constructing the least fixed-point of each recursive specification. Rather, (s)he should only have to supply enough information to determine that such fixed-points do exist and can be computed. The details of constructing these least fixed-points and of integrating this construction into the rest of the attribute evaluator should be done automatically by the evaluator constuctor. This reflects the principle that the specification of the translation should be distinct from the programming of the translator; i.e. that the correctness of the AG should be independent of any implementation details of the attribute evaluator.

[3] this application was proposed in a discussion with Ken Kennedy

References

[1] Alfred V. Aho and Jeffery D. Ullman. Principles of Compiler Design. Addison-Wesley, 1977.

[2] Bijan Arbab. Compiling Attribute Grammars into Prolog. IBM Los Angeles Scientific Center, July 1985.

[3] W. Babich and M. Jazayeri. Data-flow Analysis: the Method of Attributes. Acta Informatica , March, 1978.

[4] L. M. Chirica and D. F. Martin. An algebraic formulation of Knuthian semantics. Mathematical Systems Theory 13:1-27, 1979.

[5] B. Courcelle and P. Franchi-Zannettacci. Attribute Grammars and Recursive Program Schemes. In Theoretical Computer Science 17, pages 163-191. , 1982.

[6] Rodney Farrow. Sub-Protocol Evaluators for Attribute Grammars. In Proceedings of the SIGPLAN 84 Symposium on Compiler Construction. ACM, June, 1984.

[7] Rodney Farrow.
LINGUIST-86 Yet another translator writing system based on attribute grammars.
In Proceedings of the SIGPLAN 82 Symposium on Compiler Construction. ACM, June, 1982.

[8] Rodney Farrow.
Covers of Attribute Grammars and Sub-Protocol Attribute Evaluators.
Technical Report, Department of Computer Science, Columbia University, New York, NY 10027, September, 1983.

[9] H. Ganzinger, R. Giegerich, U. Moncke and R. Wilhelm.
A Truly Generative Semantics-Directed Compiler Generator.
In Proceedings of the SIGPLAN Symposium on Compiler Construction. ACM, June, 1982.

[10] Larry G. Jones and Janos Simon.
Hierarchical VLSI Design Systems Based on Attribute Grammars.
In Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages. ACM, January, 1986.

[11] J. B. Kam and J. D. Ullman.
Global data flow analysis and iterative algorithms.
J. ACM 23(1), 1976.

[12] Uwe Kastens, Brigitte Hutt, and Erich Zimmermann.
GAG:A Practical Compiler Generator.
Spring-Verlag, Berlin-Heidelberg-New York, 1982.

[13] U. Kastens.
Ordered attribute grammars.
Acta Informatica 13, 1980.

[14] Takuya Katayama.
Translation of Attribute Grammars into Procedures.
ACM Transactions on Programming Languages and Systems 6(3), July, 1984.

[15] K. Kennedy and J. Ramanathan.
A deterministic attribute grammar evaluator based on dynamic sequencing.
ACM TOPLAS 1, 1979.

[16] K. Kennedy and S. K. Warren.
Automatic generation of efficient evaluators for attribute grammars.
In Conference Record of the Third ACM symposium on Principles of Programming Languages. ACM, 1976.

[17] G. A. Kildall.
A unified approach to global program optimization,
In Conference Record of the ACM Symposium on Principles of Programming Languages. ACM, October, 1973.

[18] D. E. Knuth.
Semantics of context-free languages.
Mathematical Systems Theory 2, 1968.
correction in volume 5, number 1.

[19] Brian H. Mayoh.
Attribute Grammars and Mathematical Semantics.
SIAM Journal of Computing 10(3), August, 1981.

[20] Lawrence Paulson.
A Semantics-Directed Compiler Generator.
In Conference Record of the Ninth ACM Symposium on Principles of Programming Languages. ACM, January, 1982.

[21] Kari-Jouko Raiha, M. Saarinen, M. Sarjakoski, S. Sippu, E. Soisalon-Soininen and M. Tienari.
Revised Report on the Compiler Writing System HLP78.
Technical Report A-1983-1, Dept. of Computer Science, Univ. of Helsinki, 1983.

[22] W.A. Schulz.
Semantic analysis and target language synthesis in a translator.
PhD thesis, University of Colorado, Boulder, Colorado, July, 1976.

[23] Stephen K. Skedzeleski.
Definition and Use of Attribute Reevaluation in Attributed Grammars.
PhD thesis, University of Wisconsin, Madison, Wisconsin, December, 1978.