CUCS-102-84

Experience with a Production Compiler Automatically Generated from an Attribute Grammar

by

Rodney Farrow

Computer Science Department
Columbia University
New York, New York 10027

2 March 1984

The bulk of this research was performed while the author was on the staff of Intel Corp., Santa Clara, Ca.
Earlier versions of some of the material in this document were previously presented in [7] and [8]

Experience with a Production Compiler Automatically Generated from an Attribute Grammar

by

Rodney Farrow

Computer Science Department
Columbia University
New York, New York 10027

Abstract

This paper relates our experience implementing a production compiler from an attribute grammar. The compiler is Intel Corporation's Pascal-86 compiler. It runs on a microcomputer-based development system *without* virtual memory. An attribute grammar was written describing semantic analysis, storage allocation, and translation to intermediate code. Attribute evaluation is done in two alternating passes and the program tree is kept in intermediate files on disk.

The first version of the compiler was manually implemented from the attribute grammar. Using what was learned from this experience, an automatic attribute evaluator-generator was then written and a new version of the compiler was mechanically created from the attribute grammar.

Various techniques for optimizing the evaluator were tried. Their degree of success is reported and they are compared with other ideas from the literature. Complex attribute-values, such as sets, sequences and finite functions, were carefully implemented using *applicative data structures* in order to conserve memory.

The attribute grammar was designed using the principles of *data abstraction* and *information-hiding*. The internal organization of many types of attributes is completely hidden from the attribute grammar author and the attribute evaluator. These values are manipulated only by specific out-of-line semantic functions that can be viewed as the operators of an *abstract data type* for this attribute. This also contributed to an efficient use of memory.

Table of Contents

List of Figures

1. Introduction

In the past several years attribute grammars [19] have been discussed in the literature as a basis for compiler-writing-systems [26, 6, 10, 5, 23, 16]. Most of the attention has focused on attribute evaluation strategies and on testing the evaluability of an attribute grammar according to some particular strategy. Less attention has been paid to actually writing an attribute grammar for a real language and looking at what sort of compiler would be generated from it. This paper relates our experience of writing an attribute grammar for Pascal and then implementing a compiler based on this attribute grammar by using a particular attribute evaluation scheme.

The compiler is Intel Corporation's Pascal-86 compiler. It generates code for the iAPX-8086 microprocessor family and is hosted on an 8086-based microcomputer system. The compiler can run in as little as 96K bytes of memory and can use either floppy diskettes or a faster rigid disk for temporary files and for the output. The language supported is a slight superset of the ISO standard for Pascal. The principal extensions are:

- a separate compilation facility,
- allowing the use of names before their definition,
- extra built-in functions that are useful in a microprocessor environment.

The Pascal-86 compiler is neither a "toy" nor an experiment; rather it is one of Intel's three production compilers that support the iAPX-8086.

The attribute grammar specifies those phases of the compiler that do semantic analysis, storage allocation, and translation to intermediate code. These three phases of the compiler are collectively referred to as the SEMANTICIST. The outputs of the SEMANTICIST are:

- a list of intermediate code that is the code generator's input,
- a list of semantic errors and warnings,
- a list of cross-reference transactions,
- a set of literal constants that will be memory-resident at run-time and the storage locations that need to be initialized to these values,
- a set of user-defined and predefined objects (e.g. variables, types, named constants, procedures and functions),
- a set of procedure-objects that have been designated to field particular interrupts,
- a set of objects of the compilation unit whose name's are exported beyond the compilation unit.

We decided to use attribute grammars because we thought this was a good formalism for specifying and designing these protions of a compiler. Also, we anticipated that later on we would be able to automatically generate much of the compiler from its attribute grammar. However, the first released version of the compiler was written by hand, rather than being automatically generated by a *translator-writing-system*. The attribute evaluator was written by having the programmers "play evaluator-generator" on the Pascal-86 attribute grammar.

Even so, we definitely wanted the hand-implementation to be faithful to the attribute grammar that was its design. To this end the implementors strove to write the actual high-level language source code of the SEMANTICIST as a mechanical translation of the attribute grammar according to our attribute evaluation paradigm. Optimizations of this code were allowed only so long as they could be exactly specified and uniformly applied.

The resulting hand-coded version was 58,000 bytes of machine code and cost 18 man-months of design and implementation effort. 34K of the 58K bytes of machine code corresponded to the attribute evaluator that would eventually be automatically generated; the rest was either out-of-line semantic functions or support routines that would be hand-coded even in an automatically generated compiler.

After the compiler was delivered we started working on automatically generating the attribute evaluator portion of the SEMANTICIST from its attribute grammar description. This involved two distinct tasks. The first of these was to build a program to generate attribute evaluators from attribute grammars. We already had a program that would check that an attribute grammar was well-defined and that it was evaluable according to the alternating-pass evaluation strategy that we used. This program had to be enhanced to actually generate the evaluator. Furthermore, our experience with the manually-coded version had suggested several minor changes to the evaluation strategy and several not-so-minor optimizations. These also had to be included in the attribute evaluator-generator. This generator program, LINGUIST-86, is described in [8].

The second task was to bring our attribute grammar up-to-date with the SEMANTICIST. Since the first version of the SEMANTICIST was not automatically generated it was possible to correct compiler errors that were discovered without having to make appropriate modifications to the attribute grammar; in fact, it usually took less time to do so. Since the primary goal of this project was to produce a compiler rather than to explore the technology, we used this short-cut more and more frequently as the delivery-time approached. After the first version of the compiler was finished all of these modifications had to be incorporated into the attribute grammar. Doing this took about three man-months. Several heretofore undiscovered errors in the released compiler were found by this process.

While we were working to automatically generate the SEMANTICIST the compiler was a major software product that required on-going support. This support included both repairing any errors that were discovered, and adding new functionality to the compiler (including the SEMANTICIST). This continuing modification of the SEMANTICIST complicated our efforts to automatically generate a version of the current SEMANTICIST and test it against the production version. However, this was eventually done. At this time the manually-coded production-SEMANTICIST was 70K bytes of machine code, of which 41K bytes were attribute evaluator; the automatically-generated SEMANTICIST was 71K bytes of machine code, of which 38K bytes were attribute evaluator.

Other implementations of attribute evaluators have needed a lot of memory and they are usually designed to run on large machines with some sort of virtual memory system. The SEMANTICIST runs in a small amount of memory (as little as 96K) on a system <u>without</u> virtual memory. We think the principal reasons for this success are that:

- the semantic tree is kept on sequentially-accessed intermediate files,
- complex attribute-values, such as sets, sequences, and finite functions, are implemented using applicative data structuring techniques,
- data abstraction techniques are used in the design and implementation of semantic functions in order to successfully <u>exploit</u> the value-oriented nature of attribute grammars.

This paper is intended to explain how we used attribute grammars in designing and implementing a compiler; it is not intended to be a description of the Pascal-86 compiler. To this end, the paper is organized into 3 major sections:

- section 2 is a brief tutorial on attribute grammars; readers who are familiar with attribute grammars and their terminology can skip this section, or skim it quickly, and return to it only if some unfamiliar terminology or notation is later encountered,
- section 3 discusses the Pascal-86 attribute grammar and the philosophies behind its design,
- section 4 describes the SEMANTICIST's attribute evaluator.

Several issues concerning the design and implementation of the semantic functions are explored in sub-section 3.2. The symbol table is a major component of most compilers; sub-section 3.3 discusses those aspects of the SEMANTICIST that correspond to the symbol table. Sub-section 4.1 presents the attribute evaluation strategy we use, sub-section 4.2 describes our implementation of the attribute evaluator, sub-section 4.3 discusses several optimizations of the basic attribute evaluator, and sub-section 4.4 describes how the attribute evaluator fits in with the rest of the compiler.

Our conclusions are presented in the final section. Various statistics about the SEMANTICIST are distributed around the paper; an appendix collects all of these in one place. These statistics are from three different versions of the SEMANTICIST: the original hand-coded version that was released as version 1.0 of the compiler, an updated hand-coded version that was released as version 3.0 of the compiler, and the latest automatically-generated version that has not yet been released.

2. A Brief Introduction to Attribute Grammars

A *context-free grammar* is a 4-tuple (N,Σ,S,P), where N is the set of *non-terminal* symbols, Σ is the set of *terminal* symbols, $S \in N$ is the *start symbol*, and P is the set of *productions*. A production is of the form $[p\ :\ X_0\ ::=\ X_1\ \cdots\ X_{np}]$. $X_0 \in N$ is the *left-part* of p; $X_1\ X_2 \ldots X_{np}$ is the *right-part* of p and for $i > 0$, either $X_i \in N$ or $X_i \in \Sigma$. Sometimes the expression "p[i]" is used to denote X_i.

Attribute grammars were first proposed by Knuth [19] as a way to specify the semantics of context-free languages. The basis of an attribute grammar is a context-free grammar. This describes the context-free language that is the domain of the translation, that is, those strings on which the translation is defined. This context-free grammar is augmented with *attributes* and *semantic functions*. Attributes are associated with the symbols of the grammar, both terminal and non-terminal. We write "X.A" to denote attribute A of symbol X, and $\mathcal{A}(X)$ to denote the set of attributes associated with X. Semantic functions are associated with productions; they describe how the values of some attributes of the production are defined in terms of the values of other attributes of the production.

Below is an attribute grammar that describes based integers in Ada and the values they denote. Examples of Ada based integers are:

16#9f = 9f base 16 = 159 base 10
2#10110 = 10110 base 2 = 22 base 10
3#10110 = 10110 base 3 = 93 base 10.

```
 1  number ::= digits1 '#' digits2.
 2    number.VAL    = digits2.VAL
 3    digits2.RADIX = digits1.VAL
 4    digits1.RADIX = 10
 5    digits1.POWER = 1
 6    digits2.POWER = 1

 7  digits ::= digit.
 8    digits.VAL  = digit.VAL
 9    digit.POWER = digits.POWER

10  digits0 ::= digits1 digit.
11    digits0.VAL   = digits1.VAL + digit.VAL
12    digits1.RADIX = digits0.RADIX
13    digits1.POWER = digits0.POWER * digits0.RADIX
14    digit.POWER   = digits0.POWER

15  digit ::= '0'.
16    digit.VAL = 0

17  digit ::= '1'.
18    digit.VAL = digit.POWER

19  digit ::= '2'.
20    digit.VAL = 2 * digit.POWER

          .
          .
          .

45  digit ::= 'F' | 'f'.
46    digit.VAL = 15 * digit.POWER
```

Figure 2-1: An attribute grammar for based numbers in Ada

Lines 1, 7, 10, 15, 17, ..., 45 of figure 2-1 are context-free productions; the other lines denote semantic functions. The notation of this example will be used throughout this paper: in the production of line 1, `digits1` and `digits2` denote separate occurrences of the same symbol, `digits`; the numeric suffixes distinguish these different occurrences. Different symbol-occurrences in a production, such as `digits` in the first production, give rise to different *attribute-occurrences*. Symbol-occurrences and attribute-occurrences are associated only with individual productions.

A semantic function specifies the value of an attribute-occurrence of the production, e.g. `digits1.VAL`. Semantic functions are pure functions, they have no side-effects. Their only arguments are either constants or other attribute-occurrences of the production.

Let us consider just how an attribute grammar specifies a translation. The underlying context-free-grammar of an attribute grammar describes a language. Any string in this language has a parse tree associated with it by the grammar. The nodes of this parse tree can be labelled with symbols of the grammar. Let N be an interior node of the tree. Each N has two productions associated with it. The left-part production (LP) is the production that applies at N. The children of N are labelled with the symbols in the right-part of N's LP-production. The right-part production (RP) is the production that applies at the parent of N. The parent of N is labelled with the left-part symbol of N's RP-production; N and its siblings are labelled with the symbols in the right-part of N's RP-production. Leaves of the tree don't have LP productions; the root doesn't have an RP production. Figure 2-2 shows a parse tree for the string 7#53. Each node in this tree is labelled with its associated grammar symbol, which is the left-part symbol of its LP production.

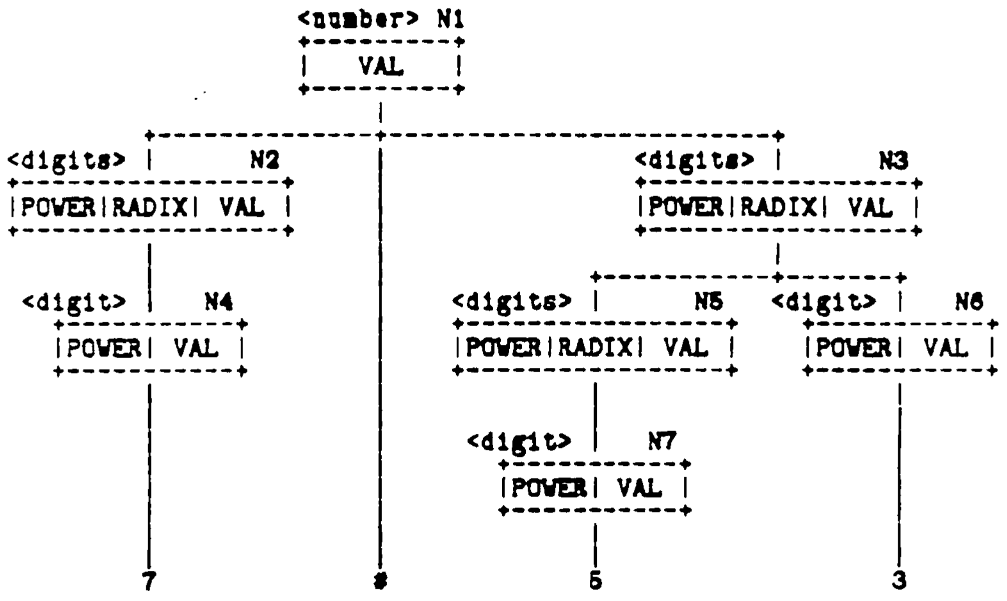

Figure 2-2: A semantic tree fragment

A *semantic tree* is a parse tree in which each node contains fields that correspond to the attributes of its labelling grammar symbol. Each of these fields is an *attribute-instance*. Associated with each attribute is a set of possible values that instances of this attribute can be assigned. This is analogous to the "type" of a variable in a programming language. However, each attribute-instance takes on precisely one such value; attribute-instances <u>are not variables</u>. The values of attribute-instances are specified by the semantic functions.

The semantic functions of a production represent a template for specifying the values of attribute-instances in the semantic tree. Consider figure 2-2 again. N3 is a semantic tree node that is associated with `digits2` in the production [`number ::= digits1 '#' digits2`] (its RP production) and N3 is associated with `digits0` in production [`digits0 ::= digits1 digit`] (its LP production). The semantic function `digits2.RADIX = digits1.VAL` indicates that the value of attribute-instance N3.RADIX will be copied from the value of attribute-instance N2.VAL. Similarly, the semantic function `digits0.VAL = digits1.VAL + digit.VAL` indicates that the value of attribute-instance N3.VAL should be calculated by adding together the values of N5.VAL and N6.VAL.

The productions, and their associated semantic functions, can be viewed as "tiling" the semantic tree. Each interior node lies on the boundary between two "tiles", those corresponding to the node's LP-production and RP-production, respectively. Thus, interior nodes lie partly in one "tile" and partly in another. Figure 2-3 is a pictorial representation of this. The "tiles" can be viewed as "templates" that describe how to define the attribute-instances that lie [partially] within them. Two different "tiles" cooperate or communicate through the values of the attribute-instances of their common node.

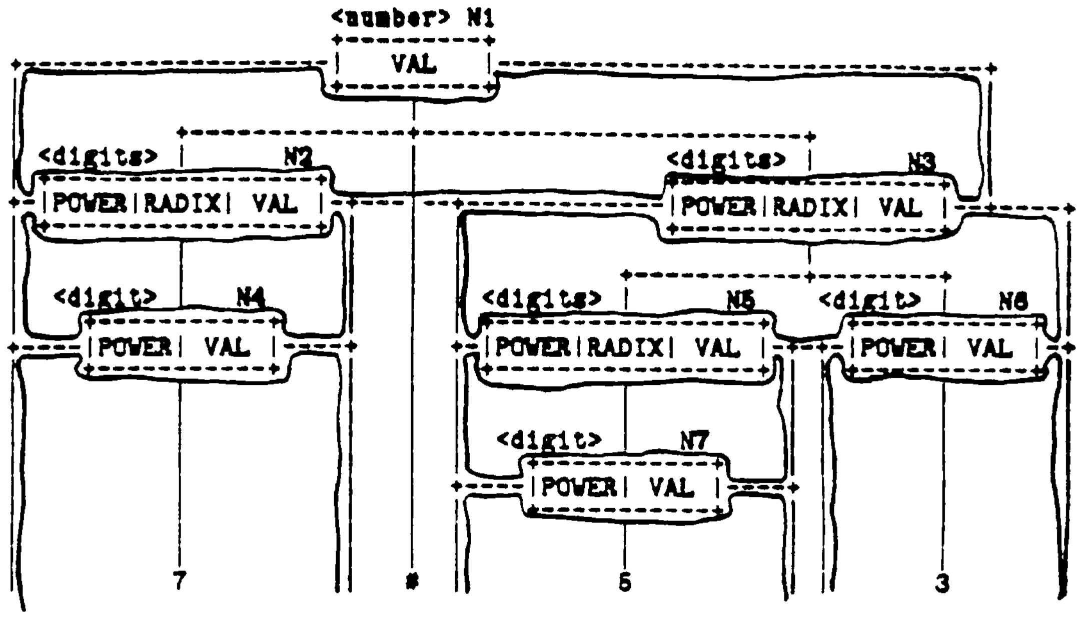

Figure 2-3: "Tiling" the semantic tree

Since two different productions are associated with each attribute-instance, there could be two semantic functions that independently specify its value, one from the LP "tile" and one from the RP "tile". If we assume that each attribute-instance is defined by only one semantic function, either from the LP production xor from the RP production, then we must guard against an attribute-instance not being defined at all because the LP production assumed that the RP production would define it and vice versa. These difficulties are avoided in attribute grammars by adopting the convention that for every attribute, X.A, either: (1) every instance of X.A is defined by a semantic function associated with its LP production, or (2) every instance of X.A is defined by a semantic function associated with its RP production. Attributes whose instances are all defined in their LP production are called *synthesized* attributes; attributes whose instances are all defined in their RP production are called *inherited* attributes. Every attribute is either inherited or synthesized. The start symbol has no inherited attributes; terminal symbols have no synthesized attributes. From the point of view of an individual production these conditions require that the semantic functions of a production MUST define EXACTLY the right-part occurrences of inherited attributes and all synthesized attributes of the left-part symbol. Inherited attributes propagate information down the tree, towards the leaves. Synthesized attributes propagate information up the tree, toward the root. The inherited attributes of a non-terminal X are denoted by $I(X)$, the synthesized attriubtes by $S(X)$: $A(X) = I(X) \uplus S(X)$.

Thus the semantic functions of an attribute grammar specify a unique value for each attribute-instance. However, in order to actually compute the value of attribute-instance Z we must first have available the values of those other attribute-instances that are arguments of the semantic function that defines Z. In the example of figure 2-2, before N3.RADIX can be computed the value of N2.VAL must have already been computed. Such ***dependency relations*** restrict the order in which attribute-instances can be evaluated. The semantic tree of figure 2-2 is reproduced in figure 2-4 with arcs drawn in to show the various dependency relations among the attribute-instances of the tree.

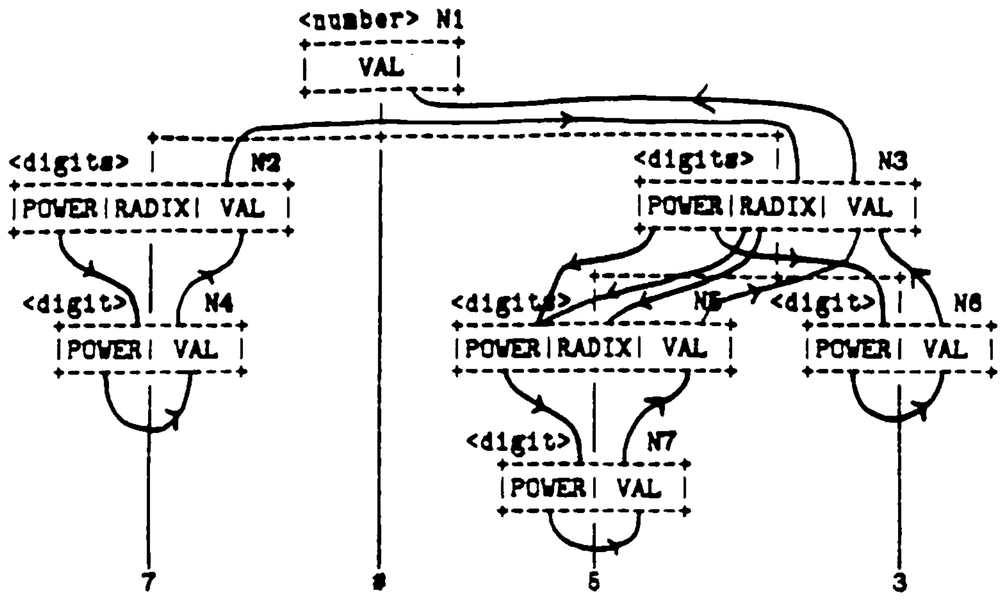

Figure 2-4: A semantic tree fragment with dependency arcs.

In extreme cases an attribute-instance can depend on itself; such a situation is called a circularity and by definition such situations are forbidden from occuring in well-defined attribute grammars. In general, it is an exponentially hard problem [13] to determine that an attribute grammar is *non-circular*; i.e. that no semantic tree that can be generated by the attribute grammar contains a circularly defined attribute-instance. Fortunately there are several interesting and widely applicable sufficient conditions that can be checked in polynomial time [4, 14, 18, 17].

The result of the translation specified by an attribute grammar is realized as the values of one or more attribute-instances of the root of the semantic tree. In order to compute these values the other attribute-instances must be computed. An ATTRIBUTE EVALUATION PARADIGM is a meta-algorithm for building an algorithm that will compute attribute-instances in such an order that: (1) no attribute-instance is computed before all dependent attribute-instances are available, and (2) all attribute-instances of the root are computed. It may be that an attribute evaluation paradigm works correctly only on a subset of all well-defined attribute grammars, but it must work correctly on every semantic tree of an acceptable attribute grammar.

One of the simplest and most commonly used family of evaluation paradigms is that of *evaluation in passes*. For evaluation in a left-to-right pass the attribute-instances are defined during a left-to-right, depth first walk over the semantic tree [4]. A general left-to-right pass evaluator consists of several such passes run in succession. Each pass is able to evaluate more attributes because their dependencies were made available by the preceding passes. The model for a left-to-right pass evaluator is shown in figure 2-5.

In order for a pass-structured attribute evaluator to work one must know in advance when attributes can be evaluated. That is, for each attribute X.A, there must be a pass number, $i_{X.A}$, such that every instance of X.A in any semantic tree can be evaluated on pass $i_{X.A}$. Because of this requirement, not all attribute grammars can be evaluated in left-to-right passes.

An *alternating-pass attribute evaluator* [14] consists of one or more passes in which the semantic tree nodes are visited in either left-to-right order or right-to-left order. If pass i visits nodes in left-to-right order, then pass i+1 visits them in right-to-left order, and vice versa. Alternating-pass evaluators can be used on a wider class of attribute grammars than left-to-right pass evaluators, but not every non-circular

```
                    X0 ::= X1 X2 ... Xn.

        evaluate inherited attribute-instances of X1 for this pass
        visit node X1

        evaluate inherited attribute-instances of X2 for this pass
        visit node X2
                              .
                              .
                              .

        evaluate inherited attribute-instances of Xn for this pass
        visit node Xn

        evaluate synthesized attribute-instances of X0 for this pass
        return from visiting X0
```

Figure 2-5: The paradigm for a left-to-right pass evaluator.

attribute grammar can be evaluated in alternating passes. The SEMANTICIST uses an alternating-pass evaluator.

Attribute grammars are attractive specification tools. Two principal reasons for this are their *locality of reference* and their *non-procedural nature*. We say that an attribute grammar has locality of reference in that the values it defines (i.e. the attribute-instances) are specified exclusively in terms of other attribute-instances local to a production. An attribute grammar does not contain any global variables or implicit state information that can affect the translation. Each local piece of an attribute grammar, i.e. each production, communicates with the rest of the attribute grammar only through strictly defined interfaces: the attributes of the symbols occurring in this production.

Although there are many similarities between writing an attribute grammar and writing a program in a high-level language, these two tasks are fundamentally different. An attribute grammar is a process-independent *specification of what is to be computed*. The relationship between an attribute grammar and an attribute evaluator for that grammar is very similar to the relationship between a context-free grammar and a parser for that grammar.

A context-free grammar is a specification of the set of strings in some language. Although we may be able to generate a parser for this language from the grammar, creating the grammar is different from and easier than writing the parser. The widespread adoption of automatic parser-generating-systems based on context-free grammars is testimony to this. The context-free grammar is a declarative specification of the language; the parser can also serve as a specification of the language, but it is a procedural specification. In general, a context-free grammar can be used as the basis of several different parsers: an LL parser, an LR parser, a recursive descent parser, a precedence parser, etc. The details of operation of these parsers will be different (e.g. a production will be recognized at different times by a recursive descent parser than it will by an LR parser) but the parses that are produced will be the same.

In just the same way an attribute grammar is a declarative specification of a translation rather than a procedure for computing the translation. Different translators can be produced from an attribute grammar based on different attribute evaluation strategies, e.g. a translator based on a left-to-right pass evaluator versus an alternating pass evaluator.

The declarative nature of an attribute grammar is manifested in several ways:

1. attribute-instances are not variables, they are names for values,
2. semantic functions are pure functions, they do not use global variables and they have no side-effects,
3. the order in which the semantic functions of a production will be evaluated is not determined by the textual order in which they occur in the attribute grammar; they can be evaluated in any order that is consistent with the *attribute dependencies* of the grammar.

When writing an attribute grammar one does not refer to the *current* value of an attribute-instance; it has only one value and at any time during attribute evaluation it will either be undefined or it will be defined with this value. In particular, one is not concerned about whether an attribute-instance has been "correctly updated" at the time one wants to use it; one simply uses it. The attribute evaluator-generator is then responsible for ordering the semantic functions so that no instance of this attribute is used before it is defined.

The lack of side-effects in semantic functions means that all the results of a semantic function must be returned as explicit results; that is, all the effects of a semantic function must appear as a component of the value(s) that are returned by that function. The lack of global variables means that all arguments to a semantic function must be explicit arguments.

The effect of this is to put a heavy emphasis on values; the values that are assigned to attribute-instances and that are used and produced by semantic functions. Besides the usual data types of integer, character, string, etc., the attribute grammar uses more complex values such as: [variant] records, sets, sequences, functions, sets of sets, sequences of sets, functions from sets to sequences, etc. In this respect attribute grammars are similar to the *applicative* programming languages, such as: pure LISP; FP [3], ML [22][1] ; VAL [1, 2] and SISAL [21][2,3].

Although attribute-instances are not variables, attribute grammars can be designed so that sets of attributes are connected by semantic functions in such a pattern that together they simulate the effect of a global variable. For example, consider the following attribute grammar fragment that assigns consecutive "addresses" to a list of elements whose individual lengths can vary, and that computes the total amount of storage needed by the entire list.

```
GOAL    ::= Xlist
    Xlist.IN    = 0
    GOAL.RESULT = Xlist.OUT

Xlist0 ::= Xlist1 ';' Xlist2
    Xlist1.IN  = Xlist0.IN
    Xlist2.IN  = Xlist1.OUT
    Xlist0.OUT = Xlist2.OUT

Xlist  ::= X
    X.ADDRESS  = Xlist.IN
    Xlist.OUT = Xlist.IN + Xlist.LEN
```

The pair of attributes, `Xlist.IN` and `Xlist.OUT`, denote the value that would be in the global variable when the attribute evaluator enters the sub-tree and leaves the sub-tree, respectively. Thus the various instances of these attributes in the semantic tree reflect the values that the global variable would take on during the course of attribute evaluation.

3. The Pascal-86 Attribute Grammar

Although there are some similarities between writing an attribute grammar and writing a program in a high-level language, these two tasks are fundamentally different. An attribute grammar is a process-independent *specification of what is to be computed*, rather than an algorithm for doing this computation. In this section we describe portions of the Pascal-86 attribute grammar and use these to illustrate our techniques for designing the attribute grammar and implementing the semantic functions; our techniques for implementing the attribute evaluator are described in section 4.2. The four sub-sections describe:

- the notation we use for describing attribute grammars, especially semantic functions,
- the use of applicative data structures to represent the *values* of attribute-instances,

[1] these are members of the sub-class of applicative languages known as functional languages

[2] these are members of the sub-class of applicative languages known as data-flow languages

[3] a more detailed discussion of the relationship between attribute grammars and applicative languages, especially data-flow languages, can be found in [9]

- grouping together selected attributes and semantic functions into *abstract data types*,
- adjusting the underlying context-free phrase structure and attribute-dependencies to make the attribute grammar evaluable in two alternating passes.

Several productions and their semantic functions are shown in figures 3-1 and 3-2. Figure 3-1 shows the two productions used to expand a VarList non-terminal in the Pascal-86 attribute grammar; figure 3-2 shows the two productions used to expand a StmtList non-terminal.

```
VarList           =  COLON TypeSpec.
  &semantics
   TypeSpec.NAME = nullName,
   TypeSpec.PACKED, TypeSpec.NEED_ORD = false,
   VarList.TYPE = TypeSpec.OBJ,
   VarList.TOO_BIG = (widthof(TypeSpec.OBJ) = OVLwidth)
  ;
*** Implicit copy-rule: VarList.DEFS = TypeSpec.DEFS
*** Implicit copy-rule: VarList.ERRS_0 = TypeSpec.ERRS_0
*** Implicit copy-rule: VarList.XREF_0 = TypeSpec.XREF_0
*** Implicit copy-rule: TypeSpec.SYMS = VarList.SYMS
*** Implicit copy-rule: TypeSpec.ERRS_I = VarList.ERRS_I
*** Implicit copy-rule: TypeSpec.XREF_I = VarList.XREF_I
*** Implicit copy-rule: TypeSpec.CIRCULAR_LIST = VarList.CIRCULAR_LIST
*** Implicit copy-rule: TypeSpec.PUB_EXT_LOC_FLAG = VarList.PUB_EXT_LOC_FLAG
*** Implicit copy-rule: TypeSpec.SCOPE_NAME = VarList.SCOPE_NAME
*** Implicit copy-rule: TypeSpec.LEVEL = VarList.LEVEL
*** Implicit copy-rule: TypeSpec.PUBLIC_SUBSYSTEM = VarList.PUBLIC_SUBSYSTEM
*** Implicit copy-rule: TypeSpec.IS_DOMESTIC = VarList.IS_DOMESTIC


VarList0          =  ID  VarList1.
  &semantics
   OBJ =
     if IsAccessible(VarList0.LEVEL,
                     ID.INDEX,
                     VarList0.PUBLIC_SUBSYSTEM,
                     VarList0.IS_DOMESTIC,
                     VarList0.PUB_IND)
                                   then genObjVar(ID.INDEX,
                                                  VarList1.TYPE,
                                                  VarList0.LEVEL,
                                                  VarList0.PUBLIC_SUBSYSTEM,
                                                  VarList0.IS_DOMESTIC)
                                   else   nullObj
     endif,
   VarList0.DEFS =
     if OBJ = nullObj then           VarList1.DEFS
                      else cons(OBJ, VarList1.DEFS)
     endif,
   ERR_NUM =
     if     lookup(VarList0.SYMS, ID.INDEX) <> OBJ then   multDefError
     elsif         VarList1.TOO_BIG                then varTooBigError
     else                                                     noError
     endif,
   VarList0.ERRS_0 =
     consErr(ID.LOC,ERR_NUM,ID.INDEX,VarList1.ERRS_0),
   VarList0.XREF_0 =
     putXREFdef(OBJ, ID.LOC, VarList0.SCOPE_NAME, VarList0.PUB_EXT_LOC_FLAG,
                  VarList1.XREF_0)
  ;
*** Implicit copy-rule: VarList0.DEFS = VarList1.DEFS
*** Implicit copy-rule: VarList0.TYPE = VarList1.TYPE
*** Implicit copy-rule: VarList1.SYMS = VarList0.SYMS
*** Implicit copy-rule: VarList1.ERRS_I = VarList0.ERRS_I
*** Implicit copy-rule: VarList1.XREF_I = VarList0.XREF_I
*** Implicit copy-rule: VarList1.CIRCULAR_LIST = VarList0.CIRCULAR_LIST
*** Implicit copy-rule: VarList1.PUB_EXT_LOC_FLAG = VarList0.PUB_EXT_LOC_FLAG
*** Implicit copy-rule: VarList1.SCOPE_NAME = VarList0.SCOPE_NAME
*** Implicit copy-rule: VarList1.LEVEL = VarList0.LEVEL
*** Implicit copy-rule: VarList1.PUBLIC_SUBSYSTEM = VarList0.PUBLIC_SUBSYSTEM
*** Implicit copy-rule: VarList1.IS_DOMESTIC = VarList0.IS_DOMESTIC
```

Figure 3-1: Attribute grammar fragment for VarList

```
StatList        = .
  &semantics
    StatList.TMP_DEFS, StatList.LBL_DEFS, StatList.RANGE_LBLS = emptySet,
    StatList.GET87, StatList.MASK87 = false,
    StatList.ERRS_O = StatList.ERRS_I,
    StatList.XREF_O = StatList.XREF_I,
    StatList.ILR_O = StatList.ILR_I
  ;

StatList0       = stat  StatList1.
  &semantics
    stat.IS_LABELLED = false,
    StatList0.LBL_DEFS = union(stat.LBL_DEFS, StatList1.LBL_DEFS),
    StatList0.TMP_DEFS = union(stat.TMP_DEFS, StatList1.TMP_DEFS),
    StatList0.RANGE_LBLS = union(stat.LBL_SET, StatList1.RANGE_LBLS),
    StatList0.GET87  = stat.GET87 or StatList1.GET87,
    StatList0.MASK87 = stat.MASK87 or StatList1.MASK87,
    StatList0.ERRS_O = stat.ERRS_O,
    stat.ERRS_I      = StatList1.ERRS_I,
    StatList0.XREF_O = stat.XREF_O,
    stat.XREF_I      = StatList1.XREF_I,
    StatList0.ILR_O  = stat.ILR_O,
    stat.ILR_I       = StatList1.ILR_O
  ;
*** Implicit copy-rule: StatList1.ERRS_I = StatList0.ERRS_I
*** Implicit copy-rule: StatList1.XREF_I = StatList0.XREF_I
*** Implicit copy-rule: StatList1.ILR_I = StatList0.ILR_I
*** Implicit copy-rule: StatList1.LEVEL = StatList0.LEVEL
*** Implicit copy-rule: StatList1.SCOPE_NAME = StatList0.SCOPE_NAME
*** Implicit copy-rule: StatList1.INPUT = StatList0.INPUT
*** Implicit copy-rule: StatList1.OUTPUT = StatList0.OUTPUT
*** Implicit copy-rule: StatList1.PUB_DCLS = StatList0.PUB_DCLS
*** Implicit copy-rule: StatList1.DCLS = StatList0.DCLS
*** Implicit copy-rule: StatList1.LBLS = StatList0.LBLS
*** Implicit copy-rule: stat.SYMS = StatList0.SYMS
*** Implicit copy-rule: stat.LEVEL = StatList0.LEVEL
*** Implicit copy-rule: stat.SCOPE_NAME = StatList0.SCOPE_NAME
*** Implicit copy-rule: stat.INPUT = StatList0.INPUT
*** Implicit copy-rule: stat.OUTPUT = StatList0.OUTPUT
*** Implicit copy-rule: stat.PUB_DCLS = StatList0.PUB_DCLS
*** Implicit copy-rule: stat.DCLS = StatList0.DCLS
*** Implicit copy-rule: stat.LBLS = StatList0.LBLS
```

Figure 3-2: Attribute grammar fragment for StmtList

3.1. Some Notation for Attribute Grammars

In order to make sense of the attribute grammar fragments of figures 3-1 and 3-2 one must be aware of several conventions and extensions of the notation that we used: intrinsic attributes, the form of semantic functions, temporary attributes, and implicit copy-rules. These do not affect the power of attribute grammars to describe translations; they only make it easier to write and read an attribute grammar.

Intrinsic attributes[4] are attributes that are already defined before attribute evaluation starts, much as the semantic tree is defined before evaluation starts. Intrinsic attributes are set by the parser, just as the semantic tree is built by the parser. In the Pascal-86 attribute grammar they are used to denote either the name-table-index of a terminal symbol or the location (line number) in the source text of an occurrence of a terminal symbol. An intrinsic attribute is like any other attribute except that it is evaluated before any pass; no semantic function can define an intrinsic attribute.

Form of semantic functions. For the purpose of formally defining attribute grammars a semantic function is just that, a function. It has a name, takes arguments, and returns results. However, for the purposes of actually engineering a translation we found this to be restrictive. One does not want to write an out-of-line function call in order to add 1 to the value of an argument. The early versions of the Pascal-86 attribute grammar had many semantic functions that were each used in just one or two places. These functions tended to have many arguments and be very simple. A common situation was for a function to compute as its result one of several different simple expressions, with the choice of which expression to use being based on the [e.g. boolean] value of one argument. Frequently several different attribute-

[4]Our use of intrinsic attributes, including the term itself, was proposed by Schulz [20].

occurrences would be defined this way using exactly the same conditions, so that the most natural and succinct expression of these functions would be as a single function that returned multiple values.

The management of so many small, out-of-line functions soon became unwieldy. In order to read and understand the attribute grammar one had to constantly flip back and forth between the source of the attribute grammar and a desription of the out-of-line functions. Additions to, or modifications of, the attribute grammar involved either searching through the definitions of existing out-of-line functions, or creating a new function and writing its definition somewhere other than in the source of the attribute grammar. Furthermore, we feared that the performance penalties of using so many out-of-line functions could be serious, although we did not experimentally test this.

To avoid these problems we allowed some simple semantic functions to be specified in-line, and these were translated into in-line code in the attribute evaluator. The syntax of semantic functions was expanded to include multi-valued expressions that could include both arithmetic and conditional operators. An example is:

```
X, Y, Z =
  if A+1 > 5
        then A+1, 0, NullObj
        else 1,   A, LookUp(SYMS,NAME)
  endif
```

Because we only adopted an expanded expression structure, this did not change the applicative nature of the specification. The readability of the attribute grammar improved remarkably.

Temporary attributes. Normally attributes are used to transmit information around the semantic tree. But sometimes it is useful to compute a value with a semantic function, but only use this value as an argument to other semantic functions associated with the same production. For example, to be able to name a value that is private to this production in order to avoid repeating a calculation, or to hold a value that is computable in one pass but not used until a later pass. To accomodate this, simple identifiers (i.e. FOO rather than X.Y) can occur in the left-hand-side of semantic functions, and can be referenced as arguments to other semantic functions. Like attributes, these can have but one value specified and can be written to/read from the intermediate files. Unlike attributes, they are only accessible to the semantic functions of a single production. Examples are ERRNUM and OBJ in the second production of figure 3-1.

Implicit copy-rules. Normally, the semantic functions of a production must define all synthesized attributes of the left-part symbol and all inherited attributes of any right-part symbols; if not then this is an error. However, in many cases where such definitions are missing it makes sense to automatically supply *implicit* copy-rules.

Our formula for inserting these implicit copy-rules has two flavors: one for defining synthesized attribute-occurrences of a left-part non-terminal, and one for defining inherited attributes of right-part symbols. If R.A is an inherited attribute of right-part symbol R which is not defined by any semantic function of this production, and if there is an attribute L.A of the left-part symbol L with the same attribute name, A, then an implicit copy-rule of the form `R.A = L.A` will be inserted as a semantic function of this production. If L.B is a synthesized attribute of the left-part symbol L which is not defined by any semantic function, and if there is exactly one right-part symbol, R, such that R has a synthesized attribute named B, and if there is only one occurrence of R in the right-part of the production, then an implicit copy-rule of the form `L.B = R.B` will be inserted.[5]

The Pascal-86 attribute grammar has 2030 semantic functions. 1147 of these are copy-rules and 910 of

[5]These implicit copy-rules are analogous in effect to the TRANSFER construct of the GAG system [16]. The major difference is that the TRANSFER construct must be explicitly supplied by the attribute grammar author, whereas our insertion of implicit copy-rules is automatic.

the copy-rules are implicit copy-rules.

3.2. Representing attribute values with applicative data structures

Many researchers have recognized the necessity of not creating and copying around many instances of non-atomic values such as sets, sequences, functions, relations, etc. This was addressed early in the development of the SEMANTICIST.

The SEMANTICIST builds these complex values in a separate data space: the attribute-instance fields in semantic tree nodes hold pointers to these values. Copy-rules are implemented by copying pointers instead of by copying the list or array that represents a set. This solution has been proposed in the literature by so many researchers that it has become a sort of folklore.

Copying pointers instead of complete data structures requires that one be careful when updating these data structures. For example, when taking the union of two sets, the result must be represented in a manner that preserves the two arguments; i.e. existing pointers to the argument sets must still be valid after the UNION operation is completed. Otherwise a UNION operation could inadvertantly change the value of an unrelated attribute-instance that happened to point to the same data structure. Figure 3-3 illustrates the problem that could arise; If `UNION( A, B)` is built from A and B by modifying A then any existing references to A will now be wrong.

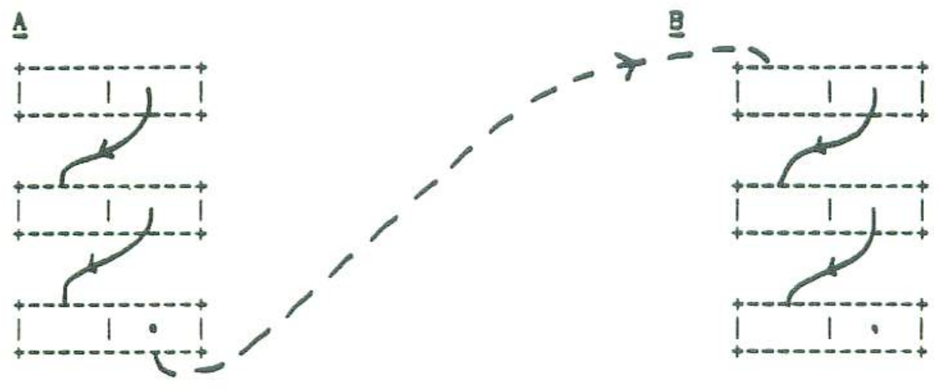

Figure 3-3: Problems with updating a shared data structure.

One way to observe these restrictions is to copy the arguments before modifying them. All the existing pointers are preserved because the original data structure isn't changed. This strategy can be expensive, not only in terms of the space required to represent so many values, but also in terms of the time needed to copy large data structures.

The SEMANTICIST uses an alternative strategy: implementing large data structures in such a way that they share identical sub-structures. As an example, consider, again, the problem of forming the UNION of two sets. To do this the SEMANTICIST allocates another cell (sets are implemented as linked lists) that points to both sets and sets a special flag in this cell to indicate that both components are sets. This is illustrated in figure 3-4.

With this strategy, it requires only marginally more memory to create a new complex value (e.g. set of definitions in a declaration list, symbol table for a new scope) if it differs just slightly from other complex values (e.g. set of definitions in a sublist, symbol table for an outer scope). As an example, consider the semantic function `VarList0.DEFS = UnionSetOf(VarList0.OBJ,VarList1.DEFS)` in the attribute grammar fragment of figure 3-1. `VarList.DEFS` is a synthesized attribute that denotes the set of all variables that are being declared in the sub-tree. As figure 3-5 shows, because the internal representation of `VarList1.DEFS` is reused to represent `VarList0.DEFS`, the internal cell-space needed to represent all of these intermediate set values is no more than the cell-space needed to represent the final, resulting set.

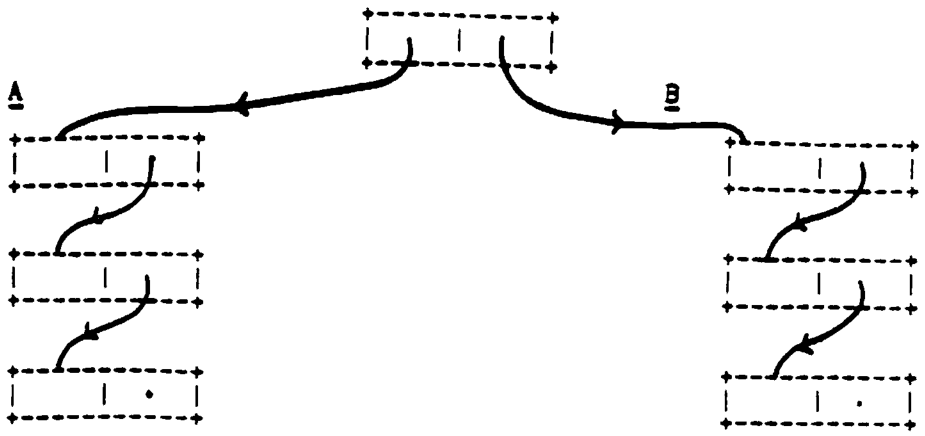

Figure 3-4: An implementation of UNION that preserves shared data structures.

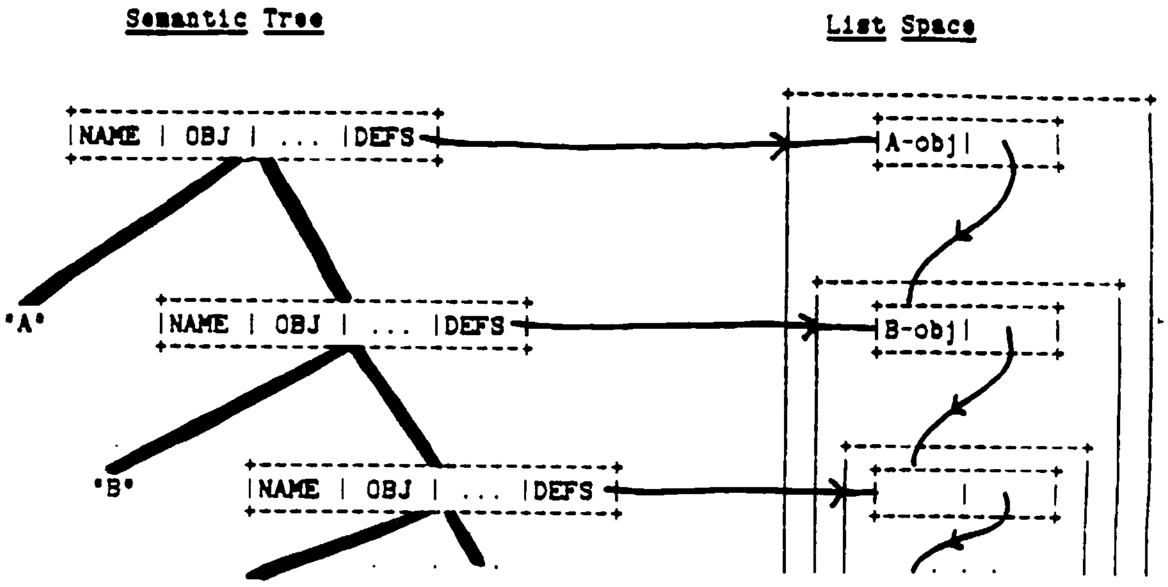

Figure 3-5: Re-using internal representations in the values of several attribute-instances.

Another example of reusing large, complex values is the table used to resolve symbolic references to variables, types, procedures, etc. This is discussed in sub-section 3.3.

The drawback to viewing complex data structures as immutable values is that it's hard to determine when they are longer needed so that their space can be reused. These values can be copied from one attribute-instance to another and written to the intermediate file. It is difficult to tell that a complex value will never be referenced again, and thus that its space can be reclaimed. Garbage collection in the SEMANTICIST is a serious problem.

Raiha [24] also describes allocating large attribute values in a separate data space and copying around pointers to these values. He suggests doing garbage collection by having the attribute evaluator determine equivalence classes of attribute-instances that are copies of the same value and finding the last reference to this value; i.e. the last reference to an attribute-instance in the equivalence class. The data structure that represents this set or list or function, etc., can be "released" and its memory reused after this last reference.

The SEMANTICIST does not use this strategy. In our scheme a complex data value can be used as a component of several other large data structures. Determining which is the last reference to such a value is much more complicated than just examining the attribute grammar and looking for copy-rules. Each semantic function would have to be examined to see whether any of its arguments, or components thereof, could be incorporated into another data structure that is returned as the result of the semantic function.

The solution used in the SEMANTICIST is to manually identify those attribute-occurrences that always hold "dead" values and to garbage-collect those values as a side-effect of evaluating a dummy semantic function. This is a very unsatisfying solution for several reasons:

1. this is a side-effect that may only work for a particular evaluation order and hence may be completely invalidated if the evaluation order changes,
2. it is very prone to error; incorrectly garbage-collecting a list is the most frequent error in the automatically-generated evaluator, and
3. this strategy doesn't reclaim all dead space.

One possible enhancement to the SEMANTICIST would be to implement a garbage collection algorithm based on dynamic reference counts. However, such an algorithm should be implemented as part of a more general data abstraction mechanism, as well as part of the attribute evaluator.

Despite the problems with garbage collection, we feel that the ability to re-use large data structures as a component of the values of many different attribute-instances is a key reason why the SEMANTICIST can be used in a production compiler on a small machine.

3.3. Data abstraction in semantic functions

The techniques of the previous sub-section can be viewed as a simple bit of *data abstraction*. In the SEMANTICIST, attributes and collections of semantic functions are grouped together to form simple *abstract data types*. There are functions to create new values of a type, to interrogate a value, to combine values, etc. and these functions take care of all the bookkeeping behind the internal representations. The external representations of such values are usually one word wide. The exported operations of such an abstract data type are the semantic functions that manipulate these values.

Collecting sets of semantic functions into abstract data types imposes (or perhaps,just makes apparent) a lot of structure in the resulting compiler. The out-of-line semantic functions are not an amorphous collection of functions; they are a small set of familiar modules: a symbol table module, a module for generating intermediate code, a module for keeping track of literal constants, a module for generating cross-reference information, etc. When this structure on the semantic functions is recognized then the central role of the attribute grammar can be appreciated: the attribute grammar describes how the various pieces of the translator communicate and coordinate with one another.

In the minds of many, the techniques of data abstraction are linked to those of *object-oriented programming*. In the context of an attribute grammar the distinction must be drawn between these two. Attribute-instances are not objects; they are values. There can be no global variables associated with the type; they must be associated with some instance of the type, i.e. with some value. The exported operations must be "pure" functions that do not modify their arguments. In the SEMANTICIST this does not mean that the implementation of semantic functions does not use hidden state variables and side-effects; they are used widely, though carefully. Rather, it means that such uses should not be detectable through the externally visible semantic functions of the abstract data type. The values returned by these semantic functions should depend only on the arguments supplied to them. On the other hand, the particular representation of such results may depend quite heavily on various hidden pieces of state information.

These techniques will be illustrated below by three different abstract data types used in the SEMANTICIST.

Literal constants. The SEMANTICIST contains a module that keeps track of addresses of memory-resident constants. These addresses are assigned "on demand". When asked for the address of a literal a search is performed of a list of literals that have already been assigned addresses. If the literal has already been assigned an address then this address is returned; otherwise the next available address is assigned to this literal, the literal and its address are put on the list, and the newly-assigned address value is returned.

This module contains two private variables, the list of previously assigned values and the next available memory address. In order for the OffsetOfLiteral function to be a true semantic function these two values would have to be passed as arguments, and updated values would have to be returned. These values would have to be passed around the semantic tree as values of attribute-instances. The result would be similar to the pattern of attributes and semantic functions used to simulate a variable that was described in section 2.

The SEMANTICIST uses the function OffsetOfLiteral even though it is not a pure function. Two distinct applications of this function with the same argument will always yield the same value, and although different evaluation orders will give slightly different results, the difference is not important. Unfortunately the attribute grammar formalism does not make allowances for "a difference that makes no difference." This topic deserves further consideration by researchers in the field.

Intermediate code generation. Lists of intermediate language instructions, lists of semantic errors, and lists of cross-reference transactions are the principal outputs of the SEMANTICIST. These lists, especially the list of intermediate code, can be quite large; their size is of the same order as the size of the semantic tree. To keep them in memory would require exorbitant amounts of memory and seriously restrict the capacity of the compiler. Moreover the contents of these lists are never referenced by the SEMANTICIST; only the single list that represents the intermediate code for the entire program is ever examined, and that happens after attribute evaluation, during the code generation phase of the compiler.

The SEMANTICIST implements the lists of intermediate code as an abstract data type whose internal representations reside in a file. In the attribute grammar we are careful to never combine two lists of intermediate code. Lists are built by adding one or two intermediate language tokens to the front of an existing list. The implementations of these semantic functions just write out these tokens to the intermediate file whenever they are added to the front a list. Since lists are never combined and since the process generates just one list there must be only one list ever built (or at least all but one list is thrown away). This means that the list to which a token is added must be the list of all tokens that will appear "after" it in the final list of intermediate code.

To see how this is done, consider two consecutive statements, stmt-i ; stmt-j, in a list of statements. A semantic tree fragment for this is shown below in figure 3-6; the attribute grammar fragment that describes it was in figure 3-2.

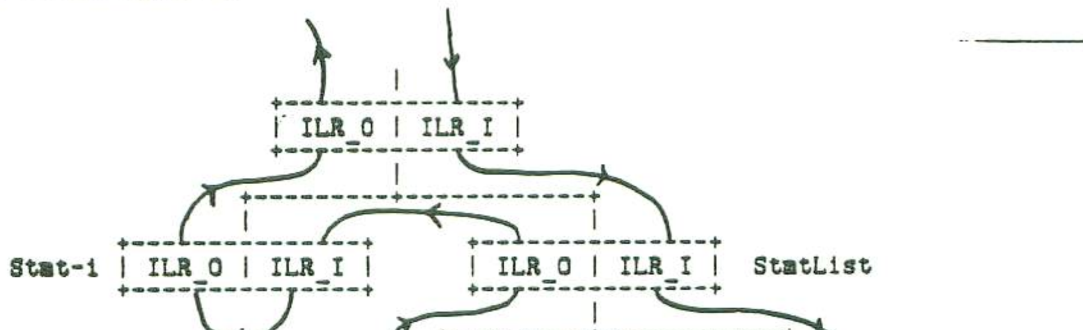

Figure 3-6: Generating intermediate code for a list of statements

The intermediate code for Stmt-j should be at the head of a list, to which the intermediate language tokens for Stmt-i are added. To do this each StmtList node must have a synthesized attribute-instance, ILR_O, that propagates this list-value from the Stmt-j node, through its parent StmtList node, and on to the Stat-1 node. Similarly, each Stat node must have an inherited attribute-instance, ILR_I, that accepts this list-value from its sibling StatList node and propagates it down to the sub-tree that it roots. Each Stat node must also have a synthesized attribute-instance, ILR_O, that transmits an updated list-value up the tree through its parent StatList node, and then to the Stat node to its left. This list-value will have been updated by adding the intermediate code for Stat-1 to the front of the list. This is exactly like the pattern of attributes and semantic functions needed to simulate a variable during attribute evaluation; the file of intermediate code can be viewed as sort of a variable.

This construction constrains the attribute evaluator to evaluate the semantic functions that generate intermediate code in the proper order, so that tokens are added to the list (i.e. written to the file) in the proper sequence. Notice that we have paid a heavy price for this efficient implementation of the lists of intermediate code. The semantic functions so constrain the evaluation order that the ILR_I and ILR_O attributes must be evaluated during a right-to-left pass. If we were to decide to generate intermediate-language during a left-to-right pass then portions of the attribute grammar would have to be revised.

The symbol table. Unlike most compilers, the SEMANTICIST does not have a central, monolithic symbol table that contains all information about program-objects, such as variables, procedures, types, symbolic constants, etc. Instead, this information is distributed among several attributes and the semantic tree itself. In more traditionally structured compilers, the symbol table serves to transfer information about program-objects between the places in the program where they are declared and the places where they are used. However, as usually designed, the symbol table is a large global variable, and putting information into it is a side-effect that is awkward to describe in the value-oriented formalism of an attribute grammar.

Other researchers have addressed this problem by creating pairs of attributes, X.SYMTBL_IN and X.SYMTBL_OUT, that together simulate the variable nature of the symbol table [20]. This sort of construction, whereby several pairs of attributes are used during attribute evaluation to simulate the effect of a variable, was described in section 2. One virtue of this approach is that the usual, efficient algorithms for manipulating symbol tables can be used. However, unless implemented very carefully, it will also create many different versions of a very large value: a symbol table. Creating so many large values will strain the capacity of even the largest virtual memory system, and those who have used this approach have also integrated special garbage-collection procedures into their attribute evaluation paradigms [24, 20].

The SEMANTICIST takes a different approach. Each program-object gives rise to a *dictionary-object* that will be the value, or part of the value, of various attribute-instances. Each dictionary-object contains part of the information that is accumulated about program-objects. There is only one dictionary-object for a given program-object.

Each dictionary-object contains the name of its program-object and what kind it is (e.g. variable vs. symbolic constant), as well as the type of a variable or procedure object, or the value of a symbolic constant. But the dictionary-object does not contain other information, such as where its program-object is referenced, or in what scopes its program-object is visible. Such information is represented in other attributes: XREF_I and XREF_O attributes for cross-reference information, SYMS attributes for scope information.

Each kind of user-defined program-object is associated with a particular non-terminal symbol that describes its declaration: ProcList symbols for procedure-objects, VarList symbols for variable-objects, TypeSpec symbols for type-objects, and Const symbols for symbolic constant-objects. Conceptually, the fields of a dictionary-object are like the attributes of a symbol. For instance, the dictionary-object for a variable contains the following fields:

NAME	an index into the compiler's table of source-text-identifiers
TYPE	a reference to the dictionary-object for the type of this variable
OFFSET	an integer value giving the offset, in the appropriate stack-frame, of the memory allocated to this variable
LEVEL	the nesting level of this variable.

Each of these values is an attribute of the `VarList` symbol (see figure 3-1). These values are computed as part of attribute evaluation and then copied into the dictionary-object.

These values are grouped together into a dictionary-object in order to make it easy to propogate this information around the semantic tree - from the node associated with the object's declaration to a node that describes a reference to the object. This propogation takes place through a set of attributes (of many symbols) whose name is SYMS: `ProcBody.SYMS`, `stat.SYMS`, `expr.SYMS`, etc. The SYMS attributes also incorporate all of the scope information.

A SYMS value is a function that maps an identifier to a dictionary-object, and thus to the values of certain attributes of the semantic tree node associated with a program-object's declaration. The SYMS values are the explicit data-paths used in the attribute grammar to transfer information from the point of declaration to the point of use. Recall that such explicit data-paths are necessary because an attribute grammar does not use global variables or other state information.

The SEMANTICIST implements a function as a list with an even number of elements. For each pair of consecutive elements, the second element is the value of the function at the first element. For the SYMS-functions the further assumption is made that if the list has more than one pair with the same first element then the value of the function is specified by the pair that is closest to the head of the list. This assumption makes it easy to update a SYMS-function; just insert another pair on the front of the list. Furthermore, as discussed in section 3.2, the SEMANTICIST implements lists so that inserting elements at the front of the list does not disturb the original list. Consequently any existing pointers to that list, and hence any existing SYMS-functions, remain valid for later or concurrent use. Figure 3-7 illustrates this. The function that represents a new scope, NEWSYMS, is formed as follows:

```
Let OLDSYMS be the function that represents the parent scope.
Let DEFS    be the set of dictionary-objects for the new, local variables,
                procedures, types, etc.

For each OBJ in DEFS
     Let NAME be the name of OBJ.
     Insert a pair (NAME,OBJ) onto the head of OLDSYMS.

The result is NEWSYMS.
```

Thus, the SEMANTICIST divides the symbol table into two pieces: dictionary-objects and SYMS-functions. Dictionary-objects are created during the first pass, and are relatively long-lived. But the SYMS-functions exist only during the second pass, while the SEMANTICIST is "visiting" the sub-tree that corresponds to the inclusive extent of the associated scope.

3.4. Making the attribute grammar evaluable in 2 alternating passes

Heretofore our discussion of the Pascal-86 attribute grammar has focused on how it describes the translation; questions of efficiency and feasibility have been pretty much hidden behind the "curtains" of data abstraction. However, at least one aspect of the efficiency and feasibility of the attribute evaluator had to be addressed directly in the design of the attribute grammar: it had to be evaluable in alternating passes.

The evaluability of an attribute grammar is determined by both the attribute dependencies of the

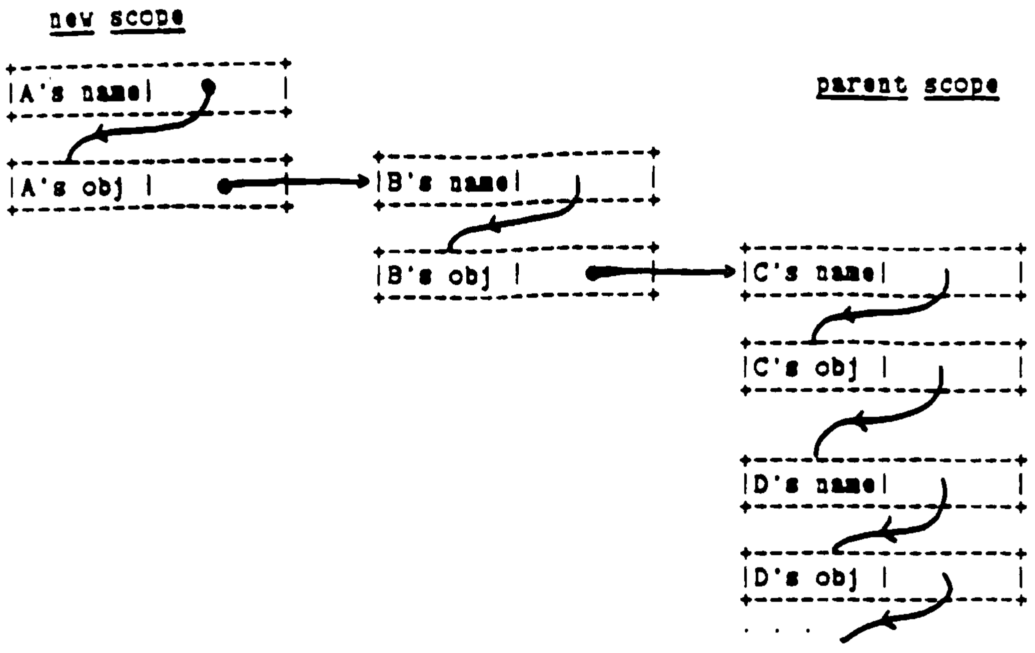

Figure 3-7: Updating a SYMS-function to represent a nested scope having local definitions of A and B

semantic functions and by the underlying context-free phrase structure. Modifying the latter turned out to be a very effective technique for coercing the attribute grammar into 2-pass evaluability.

The method should be quite familiar to readers who have designed context-free, phrase structure grammars that had to be LL(1) or LALR. There are many equally valid attribute grammars for a programming language, just as (and often because) there are many valid context-free grammars for the phrase structure of the language. In order to get the attribute grammar to be evaluable in just two alternating passes we sometimes had to use awkward or artificial constructions.

The best way to see this is with an example. Consider once again the productions of figure 3-1. They describe a variable declaration list in Pascal. An alternative way to describe the phrase-structure of variable definitions would be:

```
VarList    ::=  VarIdList COLON TypeSpec.
VarIdList  ::=  ID.
VarIdList0 ::=  ID COMMA VarIdList1.
```

These productions seem more straight-forward and they build a shallower semantic tree. However, dictionary-objects for both types and variables need to be created during the first pass. A type-object is a component of a variable-object and must be available before the variable-object can be created. The type-object for a variable declaration is computed during pass 1 as the synthesized attribute **TypeSpec.TYPE**. If the above phrase-structure were used then during the first pass (a left-to-right pass) the **TypeSpec.TYPE** attribute would not be computed before visiting the sub-tree corresponding to **VarIdList**. Therefore, the variable-objects could not be defined until after the first pass.

The productions used in the Pascal-86 attribute grammar (shown in figure 3-1) avoid this problem by making the **TypeSpec** sub-tree a descendant of the **VarList** nodes. These nodes correspond to variables and variable-objects in the dictionary are created as the value of their attributes. The type-object is propagated back to these nodes as the value of the synthesized attribute **VarList.TYPE**. This attribute is evaluated during pass 1, and hence the semantic functions that create dictionary-objects can also be evaluated during pass 1.

The Pascal-86 attribute grammar contains many places like this, where careful attention to the underlying phrase-structure of the attribute grammar was necessary to achieve two pass evaluability.

4. The Attribute Evaluator of the SEMANTICIST

4.1. The Attribute Evaluation Strategy

The SEMANTICIST uses the strategy of "evaluation in alternating passes" proposed by Jazayeri [14]. The Pascal-86 attribute grammar is evaluable in two alternating passes: a left-to-right pass followed by a right-to-left pass. For alternating pass evaluation, Schulz [26] describes a simple strategy that keeps most of the semantic tree on linearly-accessed, secondary storage (e.g. disk or tape). The SEMANTICIST uses alternating pass evaluation in order to take advantage of this and not have to keep the semantic tree in main memory.

Schulz's evaluation strategy stores a linearized version of the semantic tree in an intermediate file. When a semantic tree node, N, is encountered during the course of attribute evaluation it is read from the intermediate file into a stack in main memory. N is kept on the stack while the sub-tree descended from N is visited (and those nodes get put on the stack "below" N) and attribute-instances in that subtree are assigned values. The evaluation of the sub-tree may use the values of some attribute-instances of N and may define other attribute-instances. When the evaluation pass over N's subtree has finished, node N is written to the intermediate file. Because of the evaluation order, the nodes of N's subtree will have already been written to the file.

Figure 4-1 describes the SEMANTICIST's paradigm for semantic tree traversal and attribute evaluation in a left-to-right pass. Depicted is the process of "visiting" a node X0 that has children X1, X2, ..., Xn. For a right-to-left pass the paradigm would be the same except that the right-part nodes would be visited in the order Xn,...,X2,X1.

```
                    X0 ::= X1 X2 ... Xn.

read attribute-instances of X1 from input intermediate file
evaluate inherited attribute-instances of X1 for this pass
visit node X1
write attribute-instances of X1 to output intermediate file

read attribute-instances of X2 from input intermediate file
evaluate inherited attribute-instances of X2 for this pass
visit node X2
write attribute-instances of X2 to output intermediate file

                          .
                          .
                          .

read attribute-instances of Xn from input intermediate file
evaluate inherited attribute-instances of Xn for this pass
visit node Xn
write attribute-instances of Xn to output intermediate file

evaluate synthesized attribute-instances of X0 for this pass
return from visiting X0
```

Figure 4-1: Alternating-pass evaluation paradigm with semantic tree on intermediate files.

Schulz discussed an interpretive approach that used a single intermediate file. The SEMANTICIST contains compiled, in-line code to read and write semantic tree nodes and to evaluate semantic functions. Two intermediate files are used for each pass: nodes are read from one intermediate file and written to the other intermediate file.

This model of attribute evaluation calls for reading nodes from the input intermediate file in prefix order and writing them to the output intermediate file in postfix order. Thus, for a left-to-right pass, the input intermediate file contains the nodes in prefix, left-to-right order and the output intermediate file contains the nodes in postfix, left-to-right order. For a right-to-left pass, the input intermediate file contains the nodes in prefix, right-to-left order and the output intermediate file contains the nodes in postfix, right-to-left order.

As it happens, the reverse of a postfix, left-to-right order is the same as a prefix, right-to-left order; and the reverse of a postfix, right-to-left order is the same as a prefix, left-to-right order. This trait is illustrated in **Figure 4-2**.

```
                          M
                          |
            +-------------+------------+
            |             |            |
            F             G            L
            |                          |
      +-----+-----+              +-----+-----+
      |           |              |           |
      B           E              H           K
      |           |                          |
  +---+---+       |                      +---+---+
  |       |       |                      |       |
  A       C       D                      I       J

prefix,  left-to-right --->  <--- right-to-left. postfix
               M F B A C E D G L H K I J

postfix, left-to-right --->  <--- right-to-left, prefix
               A C B D E F G H I J K L M
```

Figure 4-2: Ordering of nodes in intermediate files

Thus, if the output intermediate file of a left-to-right pass is read backwards then its nodes are in the order expected for the input intermediate file of a right-to-left pass. Similarly, if the output intermediate file of a right-to-left pass is read backwards then it can be used as the input intermediate file for a left-to-right pass.

In the SEMANTICIST the output intermediate file of one pass becomes the input intermediate file for the next pass. Each intermediate file is first written during one pass, then read backwards during the next pass, then discarded. Because the direction of the passes alternates, an output intermediate file can serve as an input intermediate file for the next pass if it is read backwards. It is crucial to this strategy that attribute evaluation be done in alternating passes.

This strategy does not tell how to build the first input intermediate file. There are two approaches that fit naturally with the strategy. The first approach is for the parser to emit tree nodes in bottom-up order. This creates an intermediate file that is identical to what would have been created by a left-to-right attribute evaluator. No attribute evaluation is done during the first pass (the parsing pass) because there is no prefix encounter of the semantic tree nodes. The first semantic evaluation pass is a right-to-left pass.

The other approach is for the parser to emit nodes in prefix order, like a recursive descent parser. Accepting the next node from the parser takes the place of reading the next node from the input intermediate file. In this case, semantic functions can be evaluated during the same pass as parsing, so the first semantic pass is a left-to-right pass. The Pascal-86 compiler uses this second strategy.

4.2. Implementing Attribute Evaluators

The code that reads and writes semantic tree nodes, evaluates semantic functions, and "visits" other sub-trees is organized as two sets of mutually recursive procedures called *production-procedures*. There is a distinct sets of pps for each pass. There is a one-to-one correspondence between productions and the production-procedures of a pass.

Each grammar symbol has its own type definition. This type is a record that contains one field for every attribute of the symbol. A semantic tree node is implemented as a variable of this type. Each production-procedure has one value/result parameter and several local variables that hold various kinds of semantic tree nodes. The parameter corresponds to the left-part non-terminal of the associated production; the local variables correspond to the right-part symbols of the production. The body of eachproduction-procedure:

- **reads right-part** semantic tree nodes from the input intermediate file,
- computes values and defines attribute-instances by assigning to the appropriate record-fields,
- "visits" the right-part nodes by calling other production-procedures, passing the right-part node as the argument, and
- writes right-part nodes to the output intermediate file.

The prototypical production-procedure for a left-to-right pass is given in figure 4-3; it is quite similar to the evaluation paradigm described in figure 4-1. Figure 4-4 shows the pass 1, left-to-right production-procedure for the production `[digits0 ::= digits1 digit]` of figure 2-1. By convention, the name of the production-procedure in pass i that corresponds to production FOO is FOO_PPi.

This organization is similar to that of a recursive descent compiler. Notice that the stack of semantic tree nodes is intermixed with the system run-time stack that supports procedure call/return, parameter passing, and recursion.

```
  /*  production FOO is   X0 ::= X1 X2 ... Xn.  */

FOO_PP1 : procedure (LHSptr);
  declare
    LHSptr pointer,
    X0 based LHSptr X0_nodeType,

    X1 X1_nodeType,
    X2 X2_nodeType,
    .
    .
    .

    Xn Xn_nodeType;

  call GetNode( X1);
    /* evaluate inherited attribute-instances of X1 for pass 1 */
  call PP1( X1);
  call PutNode( X1);

  call GetNode( X2);
    /* evaluate inherited attribute-instances of X2 for pass 1 */
  call PP1( X2);
  call PutNode( X2);
              .
              .
              .

  call GetNode( Xn);
    /* evaluate inherited attribute-instances of Xn for pass 1 */
  call PP1( Xn);
  call PutNode( Xn);

    /* evaluate synthesized attribute-instances of X0 for pass 1 */
  return;
end; /* of procedure FOO_PP1 */
```

Figure 4-3: The prototypical production-procedure for a left-to-right pass

Listed below are some figures describing performance characteristics of the three versions of the SEMANTICIST. The object-code size figures were computed by adding together the size of each module in the SEMANTICIST, ignoring whether or not the module occured in both passes. The lines-per-minute figures were detemined by averaging the lines-per-minute observed on a set of 6 test programs of varying sizes; the figures for individual test programs can be found in the appendix.

```
DigitsList_PP1 : procedure (LHSptr);
  declare
    LHSptr pointer,
    digits0 based LHSptr digits_nodeType,

    digits1 digits_nodeType,
    digit   digit_NodeType;

  call GetNode( digits1);
    /* evaluate inherited attribute-instances of digits1 for pass 1 */
    digits1.RADIX = digits0.RADIX;
    digits1.POWER = digits0.RADIX * digits0.POWER;
  call PP1( digits1);
  call PutNode( digits1);

  call GetNode( digit);
    /* evaluate inherited attribute-instances of digit  for pass 1 */
    digit.POWER = digits0.POWER;
  call PP1( digit);
  call PutNode( digit);

    /* evaluate synthesized attribute-instances of digits0 for pass 1 */
    digits0.VAL = digits1.VAL + digit.VAL;
  return;
end; /* of procedure DigitsList_PP1 */
```

Figure 4-4: Pass 1, L-to-R production-procedure for [digits0 ::= digits1 digit] of figure 2-1

object-code (bytes)	original	updated	automatic
total	58387	69760	70722
production-procedures	34046	41367	38417
% in production-procedures	58%	59%	54%
avg. speed (lines-per-minute)	568	460	339

The notion of a production-procedure *shell* is very useful for analyzing the performance of the SEMANTICIST. The shell of a production-procedure is that production-procedure without any of the code to evaluate semantic functions: just the procedure prologue and epilogue, calls to GetNode and PutNode, and the recursive calls to production-procedures for the right-part non-terminals of the production. The shell of a production-procedure is slightly different for a left-to-right pass than it is for a right-to-left pass (if the production has more than one right-part non-terminal), but the size of these is exactly the same. However, the pass 2 production-procedures of the SEMANTICIST do not contain any PutNode calls because the last attribute evaluation pass need not write an intermediate file of semantic tree nodes; consequently, the production-procedure shells for the second pass also do not have PutNode calls and so are smaller than the the first pass shells.

The total size of all the production-procedure shells gives the code-size "overhead" of a pass. The running-time of a shell tells the dynamic (i.e. run-time) overhead of a pass: that is, the amount of time the pass will take, exclusive of the time needed for any semantic function evaluation. Information about shells is available only for the automatically-generated production-procedures: the total object-code size of all the shells for pass 1 is 5595 bytes, for pass 2 it is 3860 bytes. The average running time of both the pass 1 and pass 2 shells is 838 lines-per-minute. The appendix gives the individual figures for each member of the performance test-suite. Note that the figures on shell performance can be usefully compared only to the automatically-generated version of the SEMANTICIST.

4.3. Attribute Evaluator Optimizations

The previous sub-section described the basic attribute evaluation paradigm, according to which the SEMANTICIST was produced from its attribute grammar. However, there were several optimizations that were applied to this paradigm. Some of these were simple, local re-arrangements of the production-procedures, or minor modifications to the alternating pass evaluability criterion (see page 164 of [8]).

Others involved more complicated global analysis of the attribute grammar. This sub-section will describe three optimizations. Two of the three are the most effective of the optimizations that were applied. The other was a complex optimization that we expected to be effective, but that turned out to be a disappointment.

Shortening intermediate files. One important and obvious optimization is to reduce the amount of data transferred between the intermediate files and memory by not writing attribute-instances that will never be referenced after the current pass. In a two pass evaluator this means that we must write to the intermediate file only those attributes that are defined in pass 1 and referenced in pass 2. The majority of attributes in our attribute grammar are used only during the same pass in which they are defined; the attribute grammar has 898 total attributes, only 166 of them are actually transferred between the intermediate file and memory.

The effect of not writing such *dead* attribute-instances to the intermediate file turns out to be very similar to an evaluator optimization suggested by Saarinen [25]. He proposes that attributes be divided into "significant" attributes and "temporary" attributes. An attribute is "significant" if it is referenced in a later visit than the one in which it was defined; otherwise it is "temporary". "Significant" attributes are kept in the data structure for a node (roughly corresponding to the SEMANTICIST's intermediate files); "temporary" attributes are kept on a stack (analogous to the SEMANTICIST's stack-resident local variables of production-procedures). Jazayeri and Pozefsky [15] analyze this approach in detail for pass-structured attribute evaluators, especially alternating pass evaluators. Their experience is also that most attributes are "temporary" attributes.

Nesting production-procedures. Another optimization we considered was to nest the production-procedures within one another when possible. By doing this we sought to avoid explicitly passing a pointer to the left-part node as an argument to the nested production-procedure; references to left-part attribute-instances would just be up-level variable references. Furthermore, we hoped to eliminate, in the parent production-procedure, many copy-rules whose target or source was an attribute-instance of node X, where X was the node that was not explicitly passed as an argument to the nested procedure. This could be done because the attribute-instances of X would be known to to be the source or target of such a copy-rule and furthermore, they would be addressable from within the nested production-procedure.

Figure 4-5 shows an example of how this optimization works. The statements that are commented out are the usual statements; they are replaced by the non-commented statements on the same lines, which may be a null statement.

This turned out not to be an effective optimization for two reasons:

1. it was not applicable very often, and
2. even when it could be applied, most of its improvements could be realized anyway through the *static subsumption* optimization described later.

Production-procedure GLORP_PPi can be nested within FOO_PPi only if production FOO contains the only right-part occurrence of the left-part symbol of production GLORP. Our attribute grammar has few such occurrences, and no instances of this optimization survived in the manually-coded production-procedures. LINGUIST-86 does not implement this optimization.

Static subsumption is the third optimization. Its effect is to eliminate copy-rules and to decrease both the stack space needed to evaluate an attribute grammar and the size of the intermediate semantic tree files. Static subsumption can be applied to the entire class of "tree-walk" evaluators [18], but we will discuss it only for alternating pass evaluators.

In order to understand static subsumption it is useful to reconsider how theevaluation paradigm calls for information to be communicated around the attribute evaluator. Attribute-instances are allocated on the stack by making them local variables of recursive procedures, so that right-part attribute-instances are

```
/*
    X_list0  ::=  X  X_list1.

        X.A        = X_list0.A,
        X_list0.B = X.B,
        X_list1.C = X.C


    X  ::=  Y  Z.

        X.B = f(Y.B),
        X.C = g(Z.C),
        Y.A = h(X.A)
*/

X_list_PPi : procedure (LHSptr);
  declare
    LHSptr     pointer,
    X_list0    based LHSptr X_list_nodeType,
    X                       X_nodeType,
    X_list1                 X_list_nodeType;

  X_PPi : procedure (LHSptr);
    declare
      LHSptr     pointer,
      X    based LHSptr  X_nodeType,
      Y                  Y_nodeType,
      Z                  Z_nodeType;

    call GetNode( Y);
      Y.A = h(X_list0.A);            /* Y.A = h(X.A) */
    call PPi( Y);
    call PutNode( Y);

    call GetNode( Z);
    call PPi( Z);
    call PutNode( Z);

      X_list0.B = f(Y.B);            /* X.B = f(Y.B) */
      X_list1.C = g(Z.B);            /* X.C = g(Z.B) */
    return;
  end;  /* of procedure X_PPi */

  call GetNode( X);
                                     /* X.A = X_list0.A */
  call PPi( X);
  call PutNode( X);

  call GetNode( X_list1);
                                     /* X_list1.C = X.C */
  call PPi( X_list1);
  call PutNode( X_list1);

                                     /* X_list0.B = X.B */
  return;
end; /* of X_list_PPi */
```

Figure 4-5: An example of nesting one production-procedure within another and of the optimizations that could then be done.

directly addressable as local variables of the corresponding production-procedures. Left-part attribute-instances are referenced through a pointer that is passed as an argument to the production-procedure. This pointer represents the only communication between the various production-procedures.

An alternative to passing a pointer is to copy the attribute-instances of a node into a global variable just prior to calling the production-procedure and then, after returning from that call, copying these attribute-instances back to the local variables. Suppose that FOO is a production whose left-part non-terminal is symbol S. If every instance of attribute S.A is always copied to a specific global variable, say S_A, before entering FOO_PPi, then code in the body of FOO_PPi can obtain the value of the left-part attribute-instance S.A from the global variable S_A. Similarly, if FOO_PPi defines a [synthesized] attribute-instance S.B and by assigning the value to global variable S_B, then any production-procedure with an S-node in its right-part can access global variable S_B to obtain the value of attribute-instance S.B. If this is done then we say that the attribute is *statically allocated*.

In most cases, copying several attribute-instances back and forth is more expensive than passing a single pointer and making indirect references through it. However, if the semantic function that defines an attribute-instance is a copy-rule whose source and target are merely different instances of the same attribute, and if this attribute is statically allocated, then no explicit code is required to implement the copy-rule; the proper value is already in the global variable. We say that such a copy-rule is *subsumed* by the static allocation of the attribute. Figure 4-6 shows a simple example of how copy-rules can be subsumed; the subsumed copy-rules are commented out.

```
/*
 Production LISTprod is
    S  ::=  X S1.
        S1.A = S.A,
         X.A = S.A,
         S.B = S1.B,
        S1.C = f(S.C, X.E)
         S.D = g(X.A, S1.C, S1.D)
*/
/*
   global variables for static attributes
*/
 declare
   S_A  A_attribType,
   S_B  B_attribType;

 LISTprod_PP1 : procedure (LHSptr);
   declare
     LHSptr  pointer,
     S based LHSptr   S_nodeType,
     X                X_nodeType,
     S1               S_nodeType;

   call GetNode( X);
     X_A := S_A;
   call PP1( X);
   call PutNode( X);

   call GetNode( S);
                                  /* S1.A := S.A */
     S1.C := f( X.E, S.C);
   call PP1( S);
   call PutNode( S);

                                  /* S.B := S1.B */
     S.D = g(X.A, S1.C, S1.D);
   return;
 end;  /* of LISTprod_PP1 */
```

Figure 4-6: A simple example of static subsumption.

The penalty for eliminating this explicit copying is paid at those points where the affected attributes are not defined by subsumable copy-rules. In these cases a new value will be assigned to the global variable for propagation to the sub-tree. However, the previous value of the global variable is not "dead"; it may be referenced later in when evaluating attributes at the parent node. Hence the old value must be saved in a temporary variable in the production-procedure's stack-frame. After visiting a right-part node (i.e. upon return from the call to the production-procedure) the saved value must be "restored" to the global variable. A further complication is that any newly-computed, right-part values (i.e. synthesized, right-part attribute-instances) may be used elsewhere in this production-procedure intermingled with references to the old value of the global variable (i.e. the value of the left-part attribute-instance), and after this old value has been restored to the global variable. Figure 4-7 shows the production-procedure of the earlier example modified as would be required if attributes S.C and S.D were statically allocated.

Static subsumption can be even more widely applied by allocating several different attributes to the same global variable. The major restriction is that two different attributes of the same symbol can not be allocated to the same global variable. Many more copy-rules are subsumable by such a strategy and hence can be eliminated. In the example above, S.A and X.A could be allocated to the same variable, thereby enabling us to eliminate the copy-rule X.A := S.A. On the other hand, this strategy may require that global variables be saved and restored more frequently.

```
/*
 Production LISTprod is
    S  ::=  X S1.
         S1.A = S.A,
          X.A = S.A,
          S.B = S1.B,
         S1.C = f(S.C, X.E)
          S.D = g(X.A, S1.C, S1.D)
*/
/*
    global variables for static attributes
*/
 declare
   S_A  A_attribType,
   S_B  B_attribType,
   S_C  C_attribType,
   S_D  D_attribType;


 LISTprod_PP1 : procedure (LHSptr);
   declare
     LHSptr  pointer,
     S based LHSptr  S_nodeType,
     X               X_nodeType,
     S1              S_nodeType;

   /*
     local variables for saving
     the values of global variables
   */
   declare
     S_C_PZQ  C_attribType,
     S_D_PZQ  D_attribType;

   declare
     S1_C     C_attribType,
     S1_D     D_attribType;

   call GetNode( X);
     X.A := S_A;
   call PP1( X);
   call PutNode( X);

   call GetNode( S1);
                                   /* S1.A := S.A */
     S1_C     := f( X.E, S_C);
     S_C_SAV  := S_C;
     S_C      := ST_C;
   call PP1( S1);
     S1_D := S_D;       /* save a new right-part global value */
   call PutNode( S1);

                                   /* S.B := S1.B */
     S_D = g(X.A, S1_C, S1_D);
   return;
 end;  /* of LISTprod_PP1 */
```

Figure 4-7: How global variables must be saved and then restored under static subsumption.

Static subsumption also reduces the amount of stack space needed to store attribute-instances. If a collection of attribute-instances is being used to transmit information around the semantic tree via copy-rules then explicit fields in the record that represents a node need not be allocated. This can result in significant decrease both in the stack space needed for nodes and in the size of the intermediate file.

In general, the extra code neccessary to save and later restore a global variable is as much as the code saved by subsuming several copy-rules. For static subsumption to be effective we must be careful to statically allocate a set of attributes that allows many copy-rules to be subsumed, but that doesn't cause global variables to be saved and restored too often. When the SEMANTICIST was manually coded only those attributes with obviously high pay-off were statically allocated. LINGUIST-86 uses a more systematic, polynomial-time algorithm to select the attributes to statically allocate. It statically allocates more attributes than in the manually-coded SEMANTICIST, but even this algorithm will not always find an optimal set of attributes to statically allocate.

Our experience indicates that it is very effective to allocate to the same global variable all inherited attributes that have the same name. An enormous amount of context information is copied down the tree via inherited attributes. Static subsumption can eliminate most of these copy-rules at very little cost because this context information is not often updated.

The Pascal-86 attribute grammar has 1147 copy-rules; static subsumption eliminates 746 of these, or 65%. Static subsumption eliminates 13% of the code that implements semantic functions in the SEMANTICIST's attribute evaluator (i.e. not counting the "shells" of the attribute evaluator). At first blush this is disappointing in an optimization that can potentially eliminate, on average, half of the semantic functions. However, notice that each copy-rule generates very little code, whereas semantic functions that aren't copy-rules can be quite large.

We also timed versions of the SEMANTICIST that were automatically generated without having static subsumption applied. The results are tabulated in the appendix. Because the evaluator is I/O bound there was essentially no change in running times.

Static subsumption has some elements in common with a method investigated by Ganzinger [11], who also suggests trying to allocate attribute-instances to global variables. His main purpose is to conserve storage by allocating many attribute-instances to the same variable. However, his algorithm tries to do such allocation independent of an attribute evaluation strategy and his results are pessimistic. On the other hand, static subsumption is tailored to a particular evaluation strategy, and it permits saving and later restoring the values in global variables.

Static subsumption is also similar to some optimizations done by the GAG translator-writing-system [16]. GAG uses attribute-stacks and static variables to hold some attribute-instances during attribute evaluation. Static subsumption is a sort of hybrid of these two techniques in that the effect of saving and then restoring a global variable is similar to that of an attribute-stack. The possibility, under static subsumption, of allocating several attributes to the same global variable would correspond to using a single stack for several attributes.

4.4. The SEMANTICIST and the Pascal-86 compiler

This sub-section describes how the SEMANTICIST fits into the rest of the Pascal-86 compiler. As was mentioned in the introduction, the intent of this paper is to describe how attribute grammars were used in the compiler rather than to describe the compiler itself. Therefore, this description will be brief and at a very high level.

Figure 4-8 is an overview of the Pascal-86 compiler. The compiler is organized as seven logical *phases*, which are partitioned into five physical *passes*. Each phase takes an intermediate representation of the program and "massages" it to produce another intermediate representation of the program; thus, the original source program is incrementally "massaged" into the resulting object-code. The phases communicate with one another both through these intermediate files, and through various data-structures in memory (esp. the dictionary and the table of source-text identifiers). This is a pretty traditional organization for a compiler hosted on a mini-computer or micro-computer. The SEMANTICIST comprises two of the phases: one in the first pass and one in the second pass. The first phase of the SEMANTICIST is the first attribute evaluation pass; the second phase is the second attribute evaluation pass. The SEMANTICIST shares the first pass with the lexical analyzer and the parser; the second pass is entirely the SEMANTICIST.

The dictionary is created by the SEMANTICIST during the first pass, is used by the SEMANTICIST during the second pass, and is left in memory for use by later passes. The dictionary is one of the data-values that is the result of attribute evaluation.

Other results of attribute evaluation are the lists of semantic errors, cross-reference transactions, and intermediate-code instructions. These are generated in intermediate files that are used by the third and

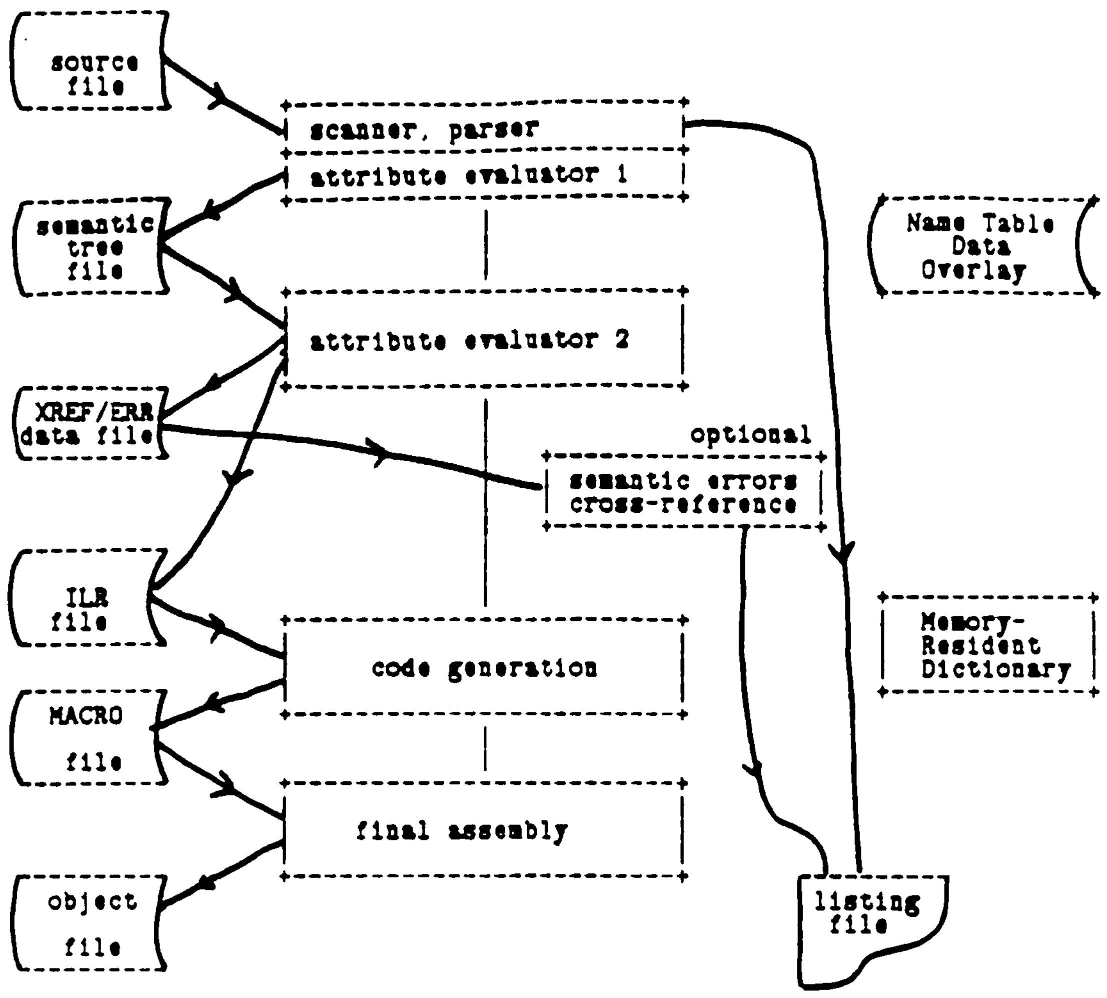

Figure 4-8: An overview of the Pascal-86 compiler.

fourth overlays. Section 3.2 described how we arrange to have a data-value that is the result of attribute evaluation be written to a file without violating the applicative nature of an attribute grammar.

The intermediate file of semantic tree nodes produced by the first pass of the SEMANTICIST is the compiler's intermediate file between overlay 1 and overlay 2. Because there is no attribute evaluation after the second pass, the SEMANTICIST does not produce an output intermediate file of semantic tree nodes during the second pass. The compiler's intermediate file between overlay 2 and overlay 4 is the list of intermediate-code produced by the SEMANTICIST as a result of attribute evaluation.

5. Conclusions

The most interesting thing we learned from this project is that the SEMANTICIST is efficient enough to compete in the market place with hand-coded compilers. Surprisingly, the SEMANTICIST spends half its time in reading and writing intermediate files: besides writing and then reading the intermediate file of semantic tree nodes, this includes writing the intermediate files of semantic errors, cross-reference transactions, and intermediate code for the code-generator. Any inefficiency due to copying values around the semantic tree is not great compared to the rest of the evaluation process.

We were somewhat surprised by another conclusion: attribute evaluation in alternating passes is quite restrictive. At the beginning of the project we recognized that other semantic evaluation strategies could theoretically handle a wider class of grammars, but we thought that alternating pass evaluation was robust enough to specify the translation of most programming languages. We still think this is true but we have come to appreciate that it can take many passes to evaluate some grammars. It is irksome to discover that one pass of the evaluator may do little but turn the tree around. Of course, if we had it to do over again we would still use alternating pass evaluation in order to be able to put a linearized semantic tree into intermediate files.

Attribute grammars were originally proposed for specifying the semantics of languages. We have also found them to be a good way to specify the design of a compiler. By the nature of the attribute grammar formalism, all data paths are explicit; that's the function of all those copy-rules. Although this may be an obstacle to deriving an efficient implementation, it is exactly the information a good design should specify and document. Furthermore, the attribute grammar has a very cohesive influence on all aspects of the compiler that it touches. We mentioned earlier (section 3.2) that the attribute grammar controls how the various modules of the SEMANTICIST interact with one another. The automatically-generated attribute evaluator serves as a sort of "glue" to connect the other components. This "glue" is flexible and easily modified while still being sufficiently regular and tractable that it can be mechanically checked for consistency. Much of the maintenance to the SEMANTICIST is done as maintenance on the attribute grammar; after each modification the attribute evaluator is mechanically regenerated and in the process LINGUIST-86 checks that all attributes are defined where they should be, that no attribute is multiply defined, and that all of the various optimizations reflect the new structure of the grammar.

Furthermore, the intermediate files that connect one pass with another are linearized semantic trees. This structure need not be explicitly designed or maintained, and it is conceptually the same from one pass to another (although its actual physical structure may vary for efficiency considerations). At first we saw this as a disadvantage. For example, an identifier reference in an expression can not be changed to a niladic procedure reference when the compiler identifies it as such. Now, after writing and using the attribute grammar and the production-procedures, we see this stability as a positive feature. If the structure of the semantic tree remains the same throughout the compiler then different aspects of the translation can be specified separately as collections of semantic functions associated with an unchanging phrase-structure that is specified by the context-free productions. These separate collections of semantic functions can communicate with one another through the attributes, and this communication can be checked for consistency.

Using a common intermediate representation to unify the compiler is not unique to attribute grammars. For instance, in [12] the authors discuss this as a valuable technique to use with the S/SL compiler-writing system. However, unlike systems such as S/SL, an attribute grammar does not directly specify an order of evaluation; i.e. the evaluator-pass during which a semantic function should be computed. Thus, partitioning an attribute grammar into functional pieces need not lead to a similar, physical organization of the compiler.

The SEMANTICIST is the result of combining and trying a lot of ideas about how to build a compiler from an attribute grammar. Many of these ideas had appeared in the literature, some were developed along the way. Some of the ideas worked well, some were not so effective. Sometimes our preconceptions about what was important turned out to be right; often they did not. What is most interesting about this effort is that there were enough good ideas available that a major part of a production compiler could be automatically generated from an attribute grammar.

Acknowledgements

In a project so large, and with results accumulated over so long a period of time, there are simply too many people whose contributions were important for them all to be listed here. Even so, the Pascal-86 compiler (of which the SEMANTICIST is only a part) was a team effort and it came to fruition only because of the hard work, dedication, and occasional brilliance, of the original development team: John Crawford, Dain Samples, Sue Ojeda and especially the Pascal-86 project leader, Tom Wilcox.

Since the original release of the compiler several engineers have worked on the SEMANTICIST, enhancing its capabilities and working to automatically generate it from the attribute grammar. Without the efforts of Ron Farrer, Marek Jeziorek, Mike Ouye, and Nancy Simmons many of the results of this paper would not exist to be reported. Ron Farrer was especially instrumental in finally generating the SEMANTICIST from its attribute grammar, and in determining most of the performance figures that are cited herein.

And finally, I am indebted to the engineering management of Intel's Development Systems Operation for allowing me to bother their engineers and to have access to a lot of proprietary information, even after I was no longer associated with the company.

I. Appendix - collected facts and figures

object-code (bytes)	original	updated	automatic
total	58387	69760	70722
production-procedures	34046	41367	38417
% in production-procedures	58%	59%	54%

production-procedure shells, pass1 (including GetNode and PutNode)	5595
production-procedure shells, pass2 (including only GetNode)	3860

production-procedures without static subsumption	43148

14% of non-shell production-procedure code eliminated by static subsumption

SEMANTICIST's speed (lines/min)

test	lines	original	updated	automatic	shell
Null Prog	--	10	14	10	4
1	36	196	141	196	240
2	63	289	192	212	363
3	108	498	362	381	689
4	1093	293	713	305	874
5	2493	1081	662	478	1511
6	2614	1063	638	463	1464
average	--	568	450	339	838

These figures reflect only the SEMANTICIST, not the entire compiler. The figures for the production-procedure shells are available only for the automatically-generated version. The figures for the Null Program are included to show the run-time overhead for just loading the program into memory. A comparison of the figures for the updated version of the SEMANTICIST (v3.0) with those for the automatically-generated version shows that the latter is smaller and hence loads faster, but that the former is faster when this overhead is discounted. This reflects the continuing effort to put more functionality into the SEMANTICIST, and hence the necessity to trade time for space in order to do this.

The Pascal-86 attribute grammar

source lines	6299	
symbols	149	
attributes	898	
two-pass		166 (18% of all attributes)
static		367 (41% of all attributes)
productions	107	
semantic functions	2030	
copy-rules		1147 (57% of all semantic functions)
implicit		910 (79% of copy-rules, 45% of all functions)
subsumed		746 (65% of copy-rules, 37% of all functions)

References

[1] W.B. Ackerman and J.B. Dennis.
VAL - a value oriented algorithmic language: Preliminary reference manual.
Technical Report 278, Laboratory for Computer Science, MIT, June, 1979.

[2] W.B. Ackerman.
Data flow languages.
IEEE Computer 15(2), February, 1982.

[3] J. Backus.
Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs.
CACM 21(8), August, 1978.

[4] G.V. Bochmann.
Semantic evaluation from left to right.
Communications of the ACM 19, 1976.
pp. 55-62.

[5] B. Lorho.
Semantic attribute processing in the system DELTA.
In A. Ershov and C.H.A. Koster (editor), *Methods of Algorithmic Language Implementation.* Springer-Verlag, Berlin-Heidelberg-New York, 1977.

[6] I. Fang.
FOLDS, a declarative formal language definition system.
Technical Report STAN-CS-72-329, Stanford University, 1972.

[7] Rodney Farrow.
Experience with an attribute grammar based compiler.
In *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages.* ACM, January, 1982.

[8] 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.

[9] Rodney Farrow.
Attribute Grammars and Data-Flow Languages.
In *Proceedings of the SIGPLAN'83 Symposium on Progamming Language Issues in Software Systems.* ACM, June, 1983.

[10] H. Ganzinger, K. Ripken, and R. Wilhelm.
Automatic generation of optimizing, multipass compilers.
In *Proceddings of IFIP 1977.* 1977.

[11] H. Ganzinger.
On storage optimization for automatically generated compilers.
In K. Weirauch (editor), *Theoretical Computer Science - Fourth GI Conference*, . Springer-Verlag, Berlin-Heidelberg-New York, 1979.

[12] R.C. Holt, J.R. Cordy, and D.B. Wortman.
An Introduction to S/SL: Syntax/Semantic Language.
ACM Transactions on Programming Languages and Systems 4(2), April, 1982.

[13] M. Jazayeri, W.F. Ogden, and W.C. Rounds.
The intrinsically exponential complexity of the circularity problem for attribute grammars.
Communications of the ACM 18, 1975.

[14] M. Jazayeri and K.G. Walter.
Alternating semantic evaluator.
In *Proceedings of ACM 1975 Annual Conference*. ACM, 1975.

[15] Diane Posefsky and Mehdi Jazayeri.
Attribute evaluation without a parse tree.
Technical Report, University of North Carolina, Chapel Hill, 1979.

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

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

[18] 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.

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

[20] K. Koskimies, K-J. Raiha, and M. Sarjakoski.
Compiler Construction Using Attribute Grammars.
In *Proceedings of the SIGPLAN Symposium on compiler construction.* ACM, June, 1982.

[21] McGraw, James, Stephen Skedzielewski, Stephen Allen, Dale Grit, Rod Oldehoeft, John Glauert, Ivan Dobs, and Paul Hohensee.
SISAL: Streams and Iteration in a Single-Assignment Language, Language Reference Manual, Version 1.1.
Technical Report, Lawrence Livermore Laboratory National Laboratory, July, 1983.

[22] Luca Cardelli.
ML under Unix.
Technical Report, Bell Laboratories, Murray Hill, N.J., 1983.

[23] Kari-Jouko Raiha, M. Saarinen, E. Soisalon-Soininen and M. Tienari.
The Compiler Writing System HLP (Helsinki Language Processor).
Technical Report A-1978-2, Dept. of Computer Science, Univ. of Helsinki, 1978.

[24] Raiha, Kari-Jouko.
Dynamic allocation of space for attribute-instances in multi-pass evaluators of attribute grammars.
In *Proceedings of the SIGPLAN 79 Symposium on Compiler Construction.* ACM, 1979.

[25] M. Saarinen.
On constructing efficient evaluators for attribute grammars.
In C. Ausiello and C. Bohm (editor), *Automata, Languages, and Programming: 5th Colloquium.* Springer-Verlag, Springer-Verlag, New York, 1978.

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