Interfacing Syntax and Semantics*

by

Cynthia A. Brown and Paul W. Purdom, Jr.

Computer Science Department

Indiana University

Bloomington, Indiana 47405

TECHNICAL REPORT NO. 113

INTERFACING SYNTAX AND SEMANTICS

Cynthia A. Brown and Paul W. Purdom, Jr.

Revised July 1982

*Research reported herein was supported in part by the National Science Foundation under grant number MCS 7906110.

Interfacing Syntax and Semantics

by

Cynthia A. Brown and Paul W. Purdom, Jr.

Abstract. A method for specifying the first pass of bottom-up compilers using a specialized form of attribute grammars is given. The notation specifies both the syntax of the language being accepted and the detailed flow of semantic information in the compiler. It is a powerful aid in developing a structured, well-documented detailed design.

1. Introduction

The design of a compiler is usually broken into two major facets: a parsing, or syntactic, component, which analyzes the structure of the input, and a semantic component, which generates code. Control, at least for the first pass of compilation, is in the parser: it supervises the collection of information, and calls the appropriate semantic routines as each phrase of the input is recognized.

The design of the parsing part of a compiler using context-free grammars and syntax-directed translation is well understood (see [1], for example). Recently, formal notations for the specification of semantics have been developed. These include denotational semantics [11] and attribute grammars (see [9] for a bibliography). Attribute grammar notation provides a convenient way to specify the overall flow of information in a compiler. Information originates in the scanner and in semantic routines, and is used by semantic routines under the direction of the parser. Since no one routine is responsible for the overall information flow, obtaining a correct, smoothly functioning design is made much easier by the use of an appropriate design tool, such as attribute grammars.

Attribute grammars typically show where information originates and where it is used, but the details of how it gets from one point to another are omitted from the specification. It is left to the implementor to determine what changes each routine should make to the stack on which information is passed, and which quantities, if any, to store in global variables. Algorithms have been proposed for automatically checking certain classes of attribute grammar specifications and generating detailed implementations from them [2,3,6,10]. Such algorithms, while they do much of the compiler writer's work, do not necessarily lead to programs of the same efficiency as those designed by hand.

In this paper we present a notation based on attribute grammars for specifying the detailed flow of information during the first pass of compilation: the phase during which an intermediate representation is built in an optimizing compiler or code is generated in a simple one. This notation, which we call <u>attribute flow</u> notation, is meant to facilitate hand design of compilers. Because our attention is focussed on one left-to-right pass, the notation is closely related to classes of attribute grammars that can be evaluated in one left-to-right pass, such as the one described by Watt [12]. Our approach differs from previous authors' in its goal of producing a detailed specification of an implementation, rather than a high-level framework for one; accordingly, many details that are suppressed in other notations are made explicit in ours. Attribute flow notation is primarily a tool for the implementor, who is interested in these details, rather than the language designer, for whom they represent an unnecessary distraction.

Unlike systems that develop compilers automatically from high-level specifications, attribute flow notation requires only modest software support. It encourages good design in hand coded compilers, and ensures that the information flow portion of a design is correct. It makes a rather troublesome aspect of compiler design straightforward; it is a useful aid in designing and documenting efficient implementations of the interface between the parser and the semantic routines.

2. Notation

In parsers that use LR(k) methods, information flow may be handled through an explicit stack separate from the stack the parser uses to keep track of its internal state. We call this stack the *attribute stack*. In syntax directed translation, names of semantic routines are inserted in the grammar at the points where those routines should be called by the parser. The attribute flow notation elaborates on this by showing the (relevant part of) the current top entries on the attribute stack throughout the grammar.

For example, Fig. 1 shows a grammar specifying one and two branch *if* statements. Knowing what object code should be produced enables the designer to develop routines, to insert calls to them into the proper places (as shown in Fig. 2), and to determine what information is needed by each routine. Even so, the stack manipulations needed for this construct are not obvious: we have seen many student compilers that dodge the issue by using two stacks, one for the labels associated with the *then* parts and another for the *else* parts.

Fig. 3 shows the solution, using the attribute flow notation. Items preceded by up arrows (*derived attributes*) are pushed on the value stack, items preceded by down arrows (*inherited attributes*) are popped.

1. <Statement> ::= <Balanced Statement>

2. <Statement> ::= <Unbalanced Statement>

3. <Balanced Statement> ::= <Simple Statement>

4. <Balanced Statement> ::= if <Relation> then <Balanced Statement>
   else <Balanced Statement>

5. <Unbalanced Statement> ::= if <Relation> then <Statement>

6. <Unbalanced Statement> ::= if <Relation> then <Balanced Statement>
   else <Unbalanced Statement>

7. <Relation> ::= <Expression> relation.operation <Expression>

Figure 1. A portion of a grammar for balanced andunbalanced statements.

1. <Statement> ::= <Balanced Statement>

2. <Statement> ::= <Unbalanced Statement>

3. <Balanced Statement> ::= <Simple Statement>

4. <Balanced Statement> ::= if <Relation> CONDITIONALBRANCH then
   <Balanced Statement> JUMPCODE else PUTLABEL
   <Balanced Statement> PUTLABEL

5. <Unbalanced Statement> ::= if <Relation> CONDITIONALBRANCH then
   <Statement> PUTLABEL

6. <Unbalanced Statement> ::= if <Relation> CONDITIONALBRANCH then
   <Balanced Statement> JUMPCODE else PUTLABEL
   <Unbalanced Statement> PUTLABEL

7. <Relation> ::= <Expression> relation.operation <Expression> RELCODE

Figure 2. A portion of a grammar for balanced and unbalanced statements, showing calls to semantic routines.

```
1. <Statement> ::= <Balanced Statement>

2. <Statement> ::= <Unbalanced Statement>

3. <Balanced Statement> ::= <Simple Statement>

4. <Balanced Statement> ::= if <Relation>↑reg
        CONDITIONALBRANCH↓reg↑label.1 then
        <Balanced Statement> JUMPCODE↑label.2 else
        REARRANGE↓label.2↓label.1↑label.2↑label.1 PUTLABEL↓label.1
        <Balanced Statement> PUTLABEL↓label.2

5. <Unbalanced Statement> ::= if <Relation>↑reg
        CONDITIONALBRANCH↓reg↑label then <Statement> PUTLABEL↓label

6. <Unbalanced Statement> ::= if <Relation>↑reg
        CONDITIONALBRANCH↓reg↑label.1 then <Balanced Statement>
        JUMPCODE↑label.2 else REARRANGE↓label.2↓label.1↑label.2↑label.1
        PUTLABEL↓label.1 <Unbalanced Statement> PUTLABEL↓label.2

7. <Relation>↑reg.3 ::= <Expression>↑reg.1 relation.operation↑relop
        <Expression>↑reg.2 RELCODE↓reg.2↓relop↓reg.1↑reg.3
```

Figure 3. A portion of an attribute flow grammar for balanced and unbalanced statements.

(Derived attributes are called synthesized attributes by some authors). The pushing and popping occur in the order given by reading the right side of the production from left to right. Thus A↑a B↑b↑c produces a stack with c on top, b next to top, and a below b. The left side of a production summarizes the overall effect of the right side. The derived attributes for terminal symbols are produced and pushed by the scanner. Routines pop their inherited attributes and use them as input parameters. They compute their derived attributes and push them onto the attribute stack. For nonterminal symbols the inherited attributes show the items on top of the attribute stack before the nonterminal is processed by the parser, and the derived attributes show the items on top of the attribute stack after the nonterminal is processed. The inherited attributes of a nonterminal are popped and used by the right sides of its productions; the derived attributes are produced and pushed by the right sides. In all cases inherited attributes indicate information that is used while processing a symbol or routine, while derived attributes indicate information that is produced as a result of the processing.

Table 1 describes the semantic routines used in Fig. 3 and their parameters. In production 4 of Fig. 3, procedure CONDITIONALBRANCH is to produce code for testing the value of the relation and jumping to a label (generated by CONDITIONALBRANCH) if the value is false. To do this it needs to know the run-time location of the value of the relation. This is provided by the inherited attribute reg, which was placed on the stack as a result of processing the nonterminal <Relation>. The label that is generated by CONDITIONALBRANCH will need to be placed on the appropriate line of code by a future procedure (PUTLABEL), so it is a derived attribute of CONDITIONALBRANCH.

The procedure JUMPCODE generates code to jump over the else part of the code. This routine also generates a label and places it on the stack. At this point it is clear from the attribute notation that the top of the value stack is label.1 label.2 (where the stack is growing to the right), while the first call to PUTLABEL should use label.1 as its input. An operation is needed to interchange the two labels on the top of the stack so that the two calls to PUTLABEL get the correct input. Procedure REARRANGE accomplishes this by popping the top two

Routine	Purpose
CONDITIONALBRANCH$\downarrow$reg$\uparrow$label	CONDITIONALBRANCH produces code for conditionally skipping over the following block of code. The Boolean value at reg is tested and a jump to label is performed when the value is false.
JUMPCODE$\uparrow$label	JUMPCODE produces code for unconditionally jumping over the following block of code using a jump to label. Label is generated by the routine.
PUTLABEL$\downarrow$label	PUTLABEL puts a label at the start of the following block of code.
RELCODE$\downarrow$reg.2$\downarrow$relop$\downarrow$reg.1$\uparrow$reg.3	RELCODE produces code for computing the value of the logical expression that results from applying the operation relop to the values reg.1 and reg.2. The result is stored at reg.3.
REARRANGE$\downarrow$label.2$\downarrow$label.1$\uparrow$label.2$\uparrow$label.1	REARRANGE interchanges the top two elements of the stack.

Table 1. Explanation of the semantic routines for the grammar fragment of Figure 3.

items and pushing them in reverse order.

Production 7 illustrates a derived attribute for a terminal symbol. The terminal *relation operation* could be any one of several possible operators ($\leq, <, \geq, >, =, \neq$). Rather than distinguishing them grammatically, the scanner identifies them all as the terminal *relation operation*, and pushes the actual operator on the attribute stack. Procedure RELCODE pops the operator and generates the appropriate code for it.

An attribute flow grammar satisfies the following conditions. An *attribute description* consists of a direction (up arrow for derived attributes, down arrow for inherited attributes) followed by a type, a dot, and a numerical tag. The *pattern* of a list of attribute descriptions is the sequence of (direction, type) pairs. Every occurrence of a symbol in the grammar is followed by a list of attribute descriptions having a fixed pattern that is associated with the symbol. (Only the tag portion may vary.) The inherited attributes (if any) must precede the derived attributes.

The type and tag portion of an attribute description denote an attribute variable. The type component indicates the type of values the variable can assume. The tag is used to distinguish between variables of a given type within the same production. (If only one variable of a type appears, the tag may be omitted.) An attribute variable can take on only one value within a production.

An inherited attribute position on the left side of a production, or a derived position on the right, is a *defining* position. An inherited position on the left side represents information that is already on the attribute stack when the production is entered; a derived position on the right represents information placed on the attribute stack while parsing the symbol it follows. In both cases these are positions where, within the range of the current production, an attribute variable first receives its value.

The remaining attribute positions are applied positions. An inherited attribute on the right side represents information used while processing its associated symbol; a derived attribute on the left represents information left on the attribute stack for use in other productions. A variable must have a defining position to the left of any applied positions that it has on the right side of a rule. It may have a second defining position, but only following an applied position on the same symbol where the second defining position occurs. In this case our convention is that the symbol uses the value and then pushes it, unchanged, back on the value stack. A variable in an applied position on the left side of a rule must have a defining position in the rule.

The attributes give a detailed picture of the manipulations that are done to the top of the attribute stack. The inherited attributes for a symbol show the relevant part of the top of the attribute stack (in reverse order) before the symbol is processed. The derived attributes show the corresponding part of the attribute stack (in normal order) after the symbol is processed. Each symbol pops its inherited attributes from the value stack in the order in which they are listed, uses them, and pushes its derived attributes, again in the order in which they are listed.

The inherited attributes for the symbol on the left side of a production show the top of the stack before the production is processed. The inherited attributes of the first symbol on the right side of the production show the attribute variables that are popped from the value stack and used by the first symbol. These attribute variables must have the _same_ type as the variables that are on the top of the value stack, as indicated by the inherited attributes of the symbol on the left. No reference is permitted to the part of the stack below that described by the inherited attributes on the left side of the current production. The derived attributes of the first symbol on the right side show what is added to the stack after the first symbol is processed. Similarly, the remaining symbols on the right side show the changes in the attribute stack as they are processed.

The derived attributes of the symbol on the left side summarize the results of the production. They show the top of the attribute stack (down to the same level descibed by the inherited attributes) after the

Place	Stack afterwards
Initial Stack	...
<Relation>	... reg
<u>CONDITIONALBRANCH</u>	... label.1
<u>JUMPCODE</u>	... label.1 label.2
<u>REARRANGE</u>	... label.2 label.1
<u>PUTLABEL</u>	... label.2
<u>PUTLABEL</u>	...
Final Stack	...

Figure 4. The top of the attribute stack during the processing of production 4 from the grammar in Fig. 3. Three dots are used to represent the unspecified part of the stack. Since the symbol on the left side of production 4 has no inherited attributes, initially none of the attribute stack is specified. Each change in the stack is shown. The top of the stack is the rightmost symbol on the line.

process is complete. Fig. 4 shows the top of the stack of attribute variables during the processing of production 4 of the grammar in Fig. 3.

Occasionally a symbol needs to use an attribute value and then pass it on, unchanged, to other parts of the production. This is shown in the notation as a pop followed by a push. This action may be implemented by simply copying the value and leaving it on the attribute stack instead of actually popping and pushing, as long as this leaves the rest of the stack in the correct configuration.

The following algorithm checks the attribute portion of the grammar for syntactic correctness. For each production in turn, let X_0 be the symbol on the left side of the production and let $X_1,\ldots,X_n$ be the symbols on the right side. Let I_i be the string of inherited attribute variables for X_i and let D_i be the string of derived attribute variables. Form the strings $S_0,S_1,\ldots,S_n$ as follows. Let S_0 be the string of inherited attribute variables for the symbol on the left side of the production. Then S_i is formed by deleting I_i from the right end of S_{i-1} and concatenating D_i to the right end. The production is syntactically correct if (1) for each i, I_i is equal to the right end of S_{i-1} and (2) S_n is equal to D_0.

To summarize, the attributes in a grammar are syntactically correct if they satisfy the following three rules:

1. Each symbol in the grammar is always followed by the same pattern of attributes. The pattern consists of inherited attributes followed by derived attributes.
2. In each production the consecutive attribute specifications represent legal manipulations of the attribute stack as described

by the above algorithm.

3. A variable may have a second defining position in a rule , but only following an applied position on the same symbol; the convention is that the symbol is popped, used, and then pushed, unchanged, back on the stack.

There is a close correspondence between designing the mutually recursive procedures of a recursive descent parser and designing an attribute flow grammar. The inherited attributes of a symbol correspond to the input parameters of a procedure, and the derived attributes to values returned. The structure of the rules, with their pattern of embedded symbols and calls to semantic routines, corresponds to the pattern of calls to subprocedures for grammatical constructs and calls to code generating routines in a recursive descent procedure. The key to this correspondence is the rule that each symbol must always be followed by the same pattern of attributes. This rule enforces a clear, "top down" structure in a bottom up compiler.

In the first pass of compilers the information flow is predominantly from the leaves to the root of the parse tree. As a result, inherited attributes are used primarily by semantic action routines that occur in the same production as the symbol that produces the information. This is the case for all productions in Fig. 3. Occasionally, however, it is convenient to use information from more distant parts of the parse tree. In such cases the grammar needs nonterminals with inherited attributes. Fig. 5 shows a grammar designed to illustrate this. It is a grammar for a language with <u>while</u> loops and <u>exit</u> statements. If <u>exit</u> is encountered inside a while loop it causes an exit from the innermost loop; if it is encountered at top level, it causes a branch to the end of the program and a return. The inherited attribute is the label to which to jump when an exit statement is encountered. There is one such label for the main program, generated by routine GENLAB at the start of the compilation (as shown in rule 1). That label is put at the end of the generated code by PUTLAB. There is no further use for the label here, but PUTLAB leaves it on the stack. In rule 4, where PUTLAB is called near the beginning of the rule, later routines do need to know the value of the label. Routine POP is used to remove the label where it is not needed. An alternative would be to have two versions of PUTLAB.

```
1. <Program> ::= GENLAB↑label <Statement-List>↓label↑label
      PUTLAB↓label↑label POP↓label RETURNCODE

2. <Statement-List>↓label↑label ::= <Statement-List>↓label↑label;
      <Statement>↓label↑label
3. <Statement-List>↓label↑label ::= <Statement>↓label↑label
4. <Statement>↓label.1↑label.1 ::= while
      GENLAB ↑label.2 PUTLAB↓label.2↑label.2 GENLAB↑label.3
      <Boolean-Relation> ↑reg CONDITIONALBRANCH↓reg↓label.3↑label.3
      do  <Statement-List>↓label.3↑label.3
      endwhile REARRANGE↓label.3↓label.2↑label.3↑label.2
      JUMPCODE↓label.2↑label.2 POP↓label.2
      PUTLAB↓label.3↑label.3 POP↓label.3
5. <Statement>↓label↑label ::= simple statement
6. <Statement>↓label↑label ::= exit JUMPCODE↓label↑label
7. <Boolean-Relation>↑reg.3 ::= expression↑reg.1 relation operator↑relop
      expression↑reg.2 RELCODE↓reg.2↓relop↓reg.1↑reg.3
```

Figure 5. An attribute flow grammar for programs with while loops and exits.

Rule 5 shows the attribute label only on the left side of the rule. This indicates that the exit label remains undisturbed on the stack while a simple statement (a statement other than an exit or a while loop) is processed. In rule 2, where <Statement-list>↓label↑label apears on the left, the label may or may not actually be popped and pushed. The notation shows that the label is at the top of the attribute stack when the rule is started, and that it is again at the top of the stack, to which no other items have been permanently added, when the rule is completed.

Rule 4 is the most complicated, but a close look shows that it simply gives a straightforward detailed description of the attribute stack manipulations that occur as the rule is parsed. By following the stack configuration at each step it becomes immediately clear that the call to REARRANGE before JUMPCODE is needed. Without the notation, this would be a likely place for a compiler bug to creep in.

3. Global data

Information in the compiler flows from the input, through the semantic routines, to the output. The notation of Section 2 is suitable for specifying the information that flows through the attribute stack. This stack is a natural mechanism for passing information between routines that process recursively defined constructs.

It is desirable for the attribute flow grammar for a compiler to provide a complete description of the information flow, so that it can serve as a self-contained design and documentation tool. It can be cumbersome, however, to handle global data through the only mechanism provided by the notation developed so far, the attribute stack. It is possible to use the attribute stack to transfer global information, but this can lead to a large number of attributes for each symbol and to frequent need to rearrange the top entries on the value stack.

To adhere to the goal of an easy to use, natural notation which promotes good design, another mechanism is needed for specifying the transfer of global information. We use the notation $\rightarrow$variable (read "store variable") on symbols that store into global variables and $\leftarrow$variable (read "get variable") on symbols that use global variables. In this notation, any terminal symbol or semantic routine that changes the value of variable x has $\rightarrow x$ as an attribute. A routine that uses a value of x has $\leftarrow x$ as an attribute. A nonterminal which has any production that changes the value of x has $\rightarrow x$ as an attribute. A nonterminal which has any production that uses a previously defined value of x has $\leftarrow x$ as an attribute.

Store attributes are quite similar to derived attributes, and get attributes are similar to inherited attributes. Get attributes on the left side of productions, and stores on the right sides, are defining positions; the remaining positions are applied positions.

A design that uses get and store attributes must obey the following additional rules.

1. Each symbol in the grammar is always followed by the same pattern of attributes; the pattern has inherited attributes, followed by get attributes, followed by derived attributes, followed by store attributes.
2. A nonterminal with a production that changes the value of x has $\rightarrow x$ as an attribute. A nonterminal with a production that uses a previously defined value of x has $x\leftarrow$ as an attribute.
3. Any global variable that occurs in an applied position on the

right side of a rule has a defining position to the left of the applied position. A global variable with an applied position on the left side of a rule has a defining position in the rule.

4. Two store attributes for the same variable on the right side of a production must be separated by at least one get attribute. If a nonterminal has a get attribute for a global variable, the first attribute for that variable on the right side of each production for the nonterminal must also be a get attribute.

5. Any global variable that occurs in a grammar has at least one applied position in the grammar.

These rules ensure that global variables are defined before they are used, and that each global variable included in the design is actually used somewhere. A scan of the productions makes it clear where the variables are redefined (a common source of errors).

Rule 4 deserves a further explanation. It is designed to outlaw situations where a variable is set to a value and is subsequently reset before the value is used. Such situations usually represent bugs in the design. However, there may be cases where, for instance, some productions for a nonterminal use a preset value of a variable while others do not. To provide for this situation the notation includes a dummy routine, CONSUME. This routine may have any number of get attributes, and is to be inserted into productions that would otherwise violate Rule 4. No routine call actually takes place. This convention forces the designer to consciously notice and mark in the design places where values are discarded.

In nearly every compiler, the symbol table is a global data structure. Fig. 6 shows a grammar adapted from Watt [12] which illustrates the use of get and store attributes to show accesses to a symbol table. Production 1 shows that the symbol table obtains its value while ⟨List⟩ is being parsed and that the value is used while processing ⟨Sequence⟩. The other productions give more detailed information on the manipulations of the symbol table.

Specifying the use of the symbol table with stack-oriented attributes would have been much more cumbersome in our notation and much less efficient as an implementation. For example, production 5 of Fig. 6 would have been:

5. ⟨Sequence⟩↓symboltable↑symboltable ::=

  ⟨Sequence⟩↓symboltable ↑symboltable <u>use</u> <u>tag</u>↑tag

  <u>IDENTIFY</u>↓tag↓symboltable↑symboltable

Watt uses only stack-oriented attributes in his specification. His conventions allow him to avoid showing all the explicit stack manipulations. While his choice makes the ordinary attribute notation more convenient for specifying the compiler's use of global quantities,

1. ⟨Program⟩→symboltable ::= ⟨List⟩→symboltable
   ⟨Sequence⟩←symboltable <u>end</u>
2. ⟨List⟩→symboltable ::= <u>dcl</u> <u>tag</u>↑tag <u>EMPTY</u>→symboltable
   <u>DECLARE</u>↓tag←symboltable→symboltable
3. ⟨List⟩→symboltable ::= ⟨List⟩→symboltable <u>dcl</u> <u>tag</u>↑tag
   <u>DECLARE</u> ↓tag←symboltable→symboltable
4. ⟨Sequence⟩←symboltable ::=
5. ⟨Sequence⟩←symboltable::= ⟨Sequence⟩←symboltable <u>use</u> <u>tag</u>↑tag
   <u>IDENTIFY</u>↓tag←symboltable

Figure 6. A grammar showing the use of get and store attributes to represent a symbol table. The attribute for the symbol table was called set in Watt's paper [12].

it reduces its effectiveness as a tool for designing and specifying explicit stack manipulations.

A global item used by the first pass of one-pass compilers is the output file. An attribute such as $\rightarrow$output on routines can be used to indicate where code is actually produced. If two different output files are used, the places where each is written to can be shown. Explicit inclusion of the output file as a global variable is desirable because it leads to a uniform structure where all data flow is specified. This simplifies the construction of programs to enforce design standards.

The introduction of get and store attributes gives us a natural notation for handling both of the main types of data flow in a compiler. Get and store attributes are very convenient for global data, and the use of global variables can save a great deal of rearranging of the attribute stack. Inherited and derived attributes are needed to show how information used by recursive constructs is stored in and retrieved from the attribute stack. Often data is produced and consumed in the same production without any intervening recursive structures. For such local data either global or stack variables can be used.

Some compiler writers prefer to use global, or static, storage for as many data items as possible. To determine whether static storage can be used for an item, use the following method.

<u>Global Attribute Marking</u>

1. Identify the places where a new value for the item is created and mark them with store attributes for the item.
2. Identify the places where the item is used and mark them with get attributes. Any calls to CONSUME must be included here.
3. Mark with a store attribute every nonterminal that has a pro-

duction whose right side contains a store attribute. Repeat this step until no more nonterminals can be marked.

4. Mark with a get attribute every nonterminal that has a production whose right side contains a get attribute not preceded by a store attribute. Repeat this step until no more nonterminals can be marked.

5. Check that the rules for global attributes are satisfied. If they are not, the item may not be stored in a global variable under the current design. Two causes for failure are forgetting to initialize the item (so that some productions have a store on the left and none on the right), and the presence of two store attributes without an intervening get. The first problem can be remedied by inserting a routinue to initialize the variable where needed. The second can be remedied by inserting calls to CONSUME, but extreme caution must be used to avoid discarding needed values. This is more likely to be the case if the rule specifies a recursive construct. If adding calls to CONSUME would result in discarding needed values, the variable should be stored on the stack.

6. Check that the compiler works as intended under the present specification. In particular, any rules involving calls to CONSUME should be doublechecked.

Global variables tend to cause fewer problems for the inexperienced compiler designer than stack variables do, but the additional clarity gained from the store and get attribute notation can be very beneficial. Its use, by requiring each instance of a symbol to be treated uniformly, also encourages the development of a well-structured design.

4. Extensions

As the grammar of Fig. 3 illustrates, in the course of processing a production it is occasionally necessary to rearrange the items on the top of the attribute stack so that the inputs for a symbol are at the top when the symbol is reached by the parser. We use the generic term REARRANGE for routines that accomplish this task. The derived attributes of a call to REARRANGE must all be the same as the inherited attributes of the call. The routine may remove variables without replacing them, or place duplicate copies of variables on the stack. REARRANGE is exempted from the requirement that a symbol be followed by the same pattern of attributes wherever it appears, and in general a different routine will be needed for each call to REARRANGE (though of course the same routine may be used wherever the same rearrangement pattern is needed). In an automatic system the code for the necessary routines can be generated by the system.

Another way to simplify manipulating the stack is to allow data structures containing more than one value to be pushed on the attribute stack. A routine call could pop and push elements of the top structure, using only the relevant parts. This would eliminate the need for some rearranging, at the cost of making the notation more complicated.

The rules we have given require that all references to a particular item on the stack have the same type. In some cases it may be desirable to use one type name to refer to items from one set of values and a second type name to refer to items from a larger set that includes the first. Then the need may arise to change the type of an item. There are two aspects to solving this problem. One is to have a dummy semantic routine COERCE↓a↓b↓c...↑c'↑b'↑a', which declares that the item at the top of the attribute stack, which was of type a, is now considered to be of type a′, the second item, which was of type b, is now of type b′, etc. We assume that the item of type a was also in the set for type a′, etc., so that the coercion is valid. No routine COERCE is actually called; rather the dummy routine call is a device to emphasize the fact that a type change is taking place. The advantage of this aspect of the solution is that information about changing types is presented locally in the attribute flow grammar, so it is available where it is needed by someone trying to understand or check the specification. However, it does not do much to enforce global consistency. An improper renaming would be hard to detect using this method alone. The second aspect of the solution is a separate declaration section that indicates which types are subtypes of others. This enforces global consistency.

When an attribute grammar is used to specify a programming language, it is desirable to also present the underlying grammar with the portions pertaining to semantics omitted. A comparison of Fig. 1 and Fig. 3 shows that the syntax of the language is much easier to follow in Fig. 1, where attributes and routine calls are omitted. It may also be useful to present the version that includes routine calls but not attributes (as in Fig. 2), since it is this version that determines the parsability of the grammar. (For a discussion of where routine calls may be placed in an LR(k) grammar, see [8].) An automatic parser-building system for attribute grammars should include facilities for printing all three versions from the full attributed version. It should also be possible to print a version containing only selected attributes. This is particularly important when a large grammar is being developed or when a more elaborate notation (such as the one involving the use of get and store attributes) is used.

5. Conclusions

The main motivation for the development of the attribute flow notation was the observation that, while bottom-up parsing methods are more powerful than top-down methods and more compatible with sophisticated error recovery techniques, many people prefer top-down methods. The reason seems to be that people find top-down compilers easier to write and understand because the flow of data can be handled by passing parameters among mutually recursive routines, without the need for explicit attribute stack manipulations. The routines provide a clear connection between the language syntax and the data flow. We feel that the attribute flow notation, which explicitly shows the connection between syntax and data flow, makes bottom-up compilers as pleasant to write and work with as their top-down counterparts.

Our attribute flow notation is similar to the affix notation used by Watt. The principal difference is one of detail. Our productions show the top of the stack at each step in the grammar. Watt's notation shows the top of the stack only at the beginning and end of the productions; it also shows what is used and produced as the right side of the stack is processed. The system is left with the job of inserting the necessary calls to the REARRANGE routine. In some cases it is difficult or impossible to insert such calls without making the grammar impossible to parse in left to right order [8]. When such problems arise it is easier to develop a workable version if routine calls are being inserted manually.

The use of either notation is much better than trying to design a compiler without using attributes. Our notation produces a much more detailed design, which many people find easier to understand and which requires less sophisticated software support. Our notation could be used in a system where input was given in Watt's to describe how the final parser with the automatically inserted REARRANGE routines operates. This would be quite useful to anyone debugging the compiler, and it could be used to modify the final parser so that it made more efficient use of stack space. If, for example, the REARRANGE routines are inserted in the way suggested in Watt's original paper, the stack will often contain a lot of information that will never be used again. It is easy to modify explicit REARRANGE routines to correct this problem.

Designing data flow with the help of attributes encourages a top-down, modular approach to compilers, and makes poor designs less likely. For example, we have seen student compilers where the grammar allowed an else followed by an arbitrary statement as a legal statement of the language. The association of the else with the preceding if-then was enforced by the semantic routines. The use of the attribute flow grammar discourages this sort of poor design. It makes the inclusion in the same grammatical rule of constructs that pass data to one another the most natural and easiest way to do the specification.

The attribute flow notation makes the entire process of developing a bottom-up compiler, from design to debugging to documentation, easier and clearer. It gives the user the advantages of top-down, modular design usually associated with top-down parsing, along with the more powerful parsing and error recovery techniques that bottom-up parsing provides.

References

1. Alfred V. Aho and Jeffrey D. Ullman, "The Theory of Parsing, Translatien, and Compiling", Prentice-Hall, Inc., New Jersey, 1973.
2. Rodney Farrow, "Linquist-86: Yet Another Translator Writing System Based on Atrtribute Grammars", SIGPLAN Notices 17, No. 6, June 1982.
3. Harald Ganzinger, Robert Giegerich, Ulrich Monke, and Reinhard Wilhelm, "A Truly Generative Semantics-Directed Compiler Generator", SIGPLAN Notices 17, No. 6, June 1982.
4. Mehdi Jazayeri, William F. Ogden, and William C. Rounds, "The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars", CACM 18, No. 12 (1975), pp. 697-706.
5. Uwe Kastens, "Ordered Attribute Grammars", Acta Informatica 13 (1980), pp. 229-256.
6. Uwe Kastens and E. Zimmerman, "GAG-A Generator Based on Attribute Grammars", Institute fur Informatik II, Universitat Karlsruhe, Bericht Nr. 14/80.
7. Donald E. Knuth, "Semantics of Context-Free Languages", Math. Syst. Th. 2 (1968), pp. 127-145.
8. Paul Purdom and Cynthia A. Brown, "Semantic Routines and LR(k) Parsers", Acta Informatica 14, (1980), pp. 299-315.
9. Kari-Jouko Raiha, "Bibliography on Attribute Grammars", SIGPLAN Notices 15, No. 3, March 1980, pp. 35-44.
10. Kari-Jouko Raiha, M. Saarinen, E. Soisalon-Soininen and M. Tienari, "The Compiler Writing System HLP (Helsinki Language Processor)". Report A-1978-2, Dept. of Computer Science, University of Helsinki, 1978.

11. Joseph E. Stoy, "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory", The MIT Press, Cambridge, MA., 1977.

12. David Anthony Watt, "The Parsing Problem for Affix Grammars", Acta Informatica 8 (1977), pp. 1-20.