TECHNIQUES AND ADVANTAGES OF USING
THE FORMAL COMPILER WRITING SYSTEM FSL
TO IMPLEMENT A FORMULA ALGOL COMPILER*

Renato Iturriaga, Thomas A. Standish, Rudolph A. Krutar
and Jackson C. Earley
Carnegie Institute of Technology
Pittsburgh, Pennsylvania

INTRODUCTION

Implementing a compiler, as everybody knows, is not an easy task. There have appeared in the past few years a number of compiler writing systems.1-3 One of these is Feldman's "Formal Semantic Language" (FSL).4 In Feldman's thesis the assertion is made that FSL is potentially a powerful compiler writing system. The Formula Algol compiler6 is a large, nontrivial compiler incorporating several new language features, and the use of FSL to implement it constitutes the first significant test of the power of FSL. We find Feldman's assertion is justified, and the ideas he set forth in theory have been found to be successful in practice.

Some of the more important advantages of FSL that we have found are as follows.

First, the amount of time and programming effort required to implement a compiler such as the Formula Algol compiler is reasonably small (on the order of a man-year).

Second, because FSL is a high-level language incorporating certain power of expression, the task of describing compiling processes is sufficiently manageable and easy that experimental flexibility is achieved. By this we mean that we were able to experiment with a variety of organizations of parts of the compiler in order to select those with desired properties. In particular, we were able to change the syntax of Formula Algol without appreciably changing its semantics. We could also change its semantics without appreciably changing the syntax. Thus we were able to use FSL as a tool to improve the source language and at the same time to improve its implementation by finding the best compilation techniques. In contrast to the case of a hand coded compiler, we were not forced to make any organizational commitments which were of prohibitive expense to change. This is the essential reason underlying the property of flexibility, and flexibility makes FSL a good tool for a programming language designer.

A third feature of using FSL to write a compiler is that it is a rare counterexample to the familiar tradeoff between efficiency and generality. The use of the Floyd-Evans production language7,8 permits one to write an efficient syntax analyzer with sophisticated error recovery, and a compiler written in FSL not only is reasonably efficient but if written

---

*The research reported here was supported by the Advanced Research Projects Agency of the Department of Defense under Contract SD-146 to the Carnegie Institute of Technology. R. Iturriaga is partially supported by the National University of Mexico and the Instituto Nacional de la Investigacion Cientifica. T. A. Standish is a National Science Foundation graduate fellow.
properly can produce efficient object code. For example, the Formula Algol compiler produces for some classes of expressions more efficient code than the current handwritten Algol compiler in use at Carnegie Tech.

A fourth feature of FSL is that it is a language sufficiently general to allow several of the better-known useful compiling techniques to be expressed and utilized. For example, THUNKS were used to implement parameter calls in procedures, dope vectors were used to implement array storage and accessing, and the use of symbol table techniques was made easy by the fact that tables are primitive in the language. The reason for this generality is that FSL contains a powerful set of primitives that permit a user to express a large variety of compiling mechanisms directly by combination of these primitives. This feature also permitted us to invent several new variations on known compiling techniques which were well adapted to the problem at hand.

A fifth property for which, at present, we can produce no real evidence attesting to its usefulness, is that a compiler written in FSL is given a formal description. This means that in contrast to handwritten compilers we are provided with a framework in which we can begin to approach the problems of proving that compilers recognize given source languages correctly or that they compile correct code. In the case of handwritten compilers these questions are unthinkable. As an example of the kind of approach that can be made once a formal description is given we cite a doctoral thesis by Evans, in which we can begin to approach the problems of proving that compilers recognize given source languages correctly or that they compile correct code. In the case of handwritten compilers these questions are unthinkable. As an example of the kind of approach that can be made once a formal description is given we cite a doctoral thesis by Evans, in which certain properties of the production language are proven.

Finally, the activity of implementing Formula Algol had a feedback effect on the design and implementation of FSL itself. Modifications were made easy by the fact that FSL is in effect compiled in itself and thus possesses the same organization as the compilers it produces. For example, an accumulator symbol was introduced as a variable to allow the user to deal formally with the use of the accumulator. This represents a small change in the original philosophy of FSL, which was designed with machine independence in mind. It is, however, a small change with far-reaching consequences.

With the exception of the property of the usefulness of formal descriptions of compilers, we will present later in this paper concrete evidence supporting each of the claims we have made in this introduction. Our first task, however, is to explain briefly the operation of the compiler writing system. A BRIEF EXPLANATION OF THE COMPILER WRITING SYSTEM

The compiler writing system uses two formal languages to describe a compiler. First, a syntax analyzer for the source language is written as a program in the production language. This program is processed by a translator called the production loader producing as output a set of driving tables which are stored for later use. Second, a collection of semantic routines is defined by writing a program in the formal semantic language. Another translator called the semantic loader then translates this collection of routines into a set of tables and a block of code, which code is compiled for use as a part of the compiler itself. This output is also intermediate and is stored for later use.

The compiler itself (Fig. 1) is another program which reads in both the syntax tables and the semantic tables and code, and by using these translates a source language program into an object program. For the sake of efficiency a preliminary lexical transformation is performed on source language text as it is read in by a routine called the subscan. This routine recognizes the primary units of the language which are operators, reserved words, identifiers, and constants. These primary units of the source language are not fixed by the system but are declared in the production language. The subscan is a closed routine called by statements in the production language. Each time it is called it returns with the next primary unit in the source language string. As each identifier is recognized by the subscan its print name is stored in a table unless it has been entered previously, and an integer representing its relative address in the table of print names is transmitted. This integer functions from that time on as the internal name of the identifier. Abbreviations of reserved words and of operators are transmitted directly, and constants which are too long to transmit directly are saved in a table and their locations are transmitted instead.

The fundamental mechanism in the compiler is a push-down stack of ordered pairs \((\alpha, \beta)\) where \(\alpha\) is a primitive syntax unit and where \(\beta\) holds semantic information and is called the "description of \(\alpha\)." Syntactic analysis of the source language proceeds by a sequence of manipulations of this stack. The production language is used to define these manipulations and it consists of a sequence of productions of the following form:
LABEL \ L5 \ L4 \ L3 \ L2 \ L1 \ \rightarrow \ R3 \ R2 \ R1 \\
| ACTION *LINK

where the appearance of everything except L1 and the two vertical bars is optional. Each production tests for the presence of a particular configuration among the topmost syntactic units in the stack by attempting to match the pattern given by L5 L4 L3 L2 L1 against them. If a match is found and if \ \rightarrow \ R3 \ R2 \ R1 occurs in the production then the configuration matching L5 L4 L3 L2 L1 is transformed into the configuration represented by R3 R2 R1. Then the ACTION is executed. If \ \rightarrow \ R3 \ R2 \ R1 is absent and a match is found then the ACTION is executed and no transformation is performed. The only three actions we will discuss in this paper are the actions EXEC, SUBR, and RETURN. The actions SUBR and RETURN will be explained later. The action EXEC \ N, with parameter \ N, is a call to the semantic routine numbered \ N. This semantic routine may, among other things, alter the descriptions of the syntactic units involved in the stack transformation described by the production. For example, the description of an identifier may consist of the address of the run time location assigned to the variable that identifier represents. The description may also bear type bits telling the type of the variable (e.g., real, integer, formula). After the action is executed the link is examined to see if it is prefixed by a *. If it is then the subscan is called and the next primary syntactic unit in the source language string is recognized and

Figure 1. Flow of systems.
placed on top of the stack. Then control passes to the production whose label is given by the link. If a * does not precede the link then subscan is not called and control changes as in the previous case. In the event that the pattern L5 L4 L3 L2 L1 did not match the configuration of syntactic units at the top of the stack control passes to the next production in sequence. Also if the link is blank the action on the next line is executed and this process repeats until a nonblank link is found.

A sequence of productions may be organized into a closed subroutine by use of the actions SUBR and RETURN. The first production in such a sequence must be labeled and its label, say L, is the name of the closed subroutine. To call the closed subroutine starting at label L we execute the action SUBR L. When we wish to return from the subroutine we execute the action RETURN and control returns to the link of the production that called the subroutine originally. The control mechanism contains a pushdown stack permitting recursive calls on the closed subroutines.

The general structure of the syntax analyzer for Formula Algol is as follows. The major units of the source language, such as statements and expressions, correspond directly to subroutines in the production language, which subroutines analyze the given major units. For example, there is a production subroutine called the "statement scanner," which is called every time a statement is expected in the source language. There is also a production subroutine called the "expression scanner," which processes expressions of all types and which is called every time an expression is expected in the source language. The statement scanner may call the expression scanner and, in fact, corresponding to the occurrence in the source language of statements which contain other statements as parts, the statement scanner may call itself. The flow of control through the syntax analyzer is governed by the structure of the source language program being analyzed. It is roughly true that the structure of such large production subroutines as the statement scanner and the expression scanner is the following. Upon entrance to the subroutine the first few characters of an expected source language construction are subjected to a sequence of tests which separate the various possible classes of constructions that may be encountered into cases. Corresponding to each case a transfer is made to a part of the routine which treats that case specially. This basic scheme of organization was first introduced by Evans in the writing of the Carnegie Tech Algol compiler and the structure of the syntax analyzer for the Formula Algol compiler is basically an extension of it.

Because the flow of control in the syntax analyzer is directed by the constructions encountered in the source language it will be possible to use the following technique to explain various mechanisms found in the compiler. We will focus our attention on a critical subset of productions responsible for processing a given type of construction. This critical subsystem will always be embedded in a larger context but since the flow of control will never involve that context we may isolate the critical subsystem for study. This subsystem will involve calls on a set of semantic routines, and these semantic routines will be solely responsible for the compilation corresponding to the constructions which the critical subsystem processes.

An explanation of the primitives in the formal semantic language is given in Refs. 4 and 5. A complete summary of those primitives is not given in this paper, but the subset of primitives used below are accompanied by explanations in order to make the treatment self-contained.

A DETAILED EXAMPLE

Let us consider an example of the compilation of the assignment statement \( X \leftarrow A + B \times C \). As we begin to process this statement control in the productions will be transferred to the statement scanner at label S1 where at entrance to the scanner the first character X has been recognized by the subscan and has been stacked as a syntactic unit I on top of the push-down stack. Furthermore, subscan sets the description of I to be the integer which is the relative address of its print name. The critical productions in the statement scanner which treat this case are as follows:

\[
\begin{align*}
S1 & \quad I & & | & & | & & *S2 \\
S2 & \quad I & & \leftarrow & & E & & \leftarrow & & \rightarrow & & EXEC 9 & & *E1 \\
& & I & & \leftarrow & & EXEC 91 & & *S1
\end{align*}
\]

The first production at S1 matches all statements which start with an identifier, and control is transferred to S2 after scanning the next character. At S2 a discrimination is performed on the second character and in the case of assignment statements the initial identifier I is changed to an E and control is transferred to EXEC 9 where a look-up in the symbol table is performed using the integer in the
description of I. This causes the retrieval of the location of the variable X which was assigned previously and stored in the symbol table while processing the declaration of X. It is conceivable that X was not declared and thus not stored in the symbol table. The table look-up procedure sets a signal in the event that it fails to locate an object during a table look-up and a test on this signal enables us to write a semantic error exit corresponding to the case where a variable is used but not declared in a table look-up.

Upon finding the entry corresponding to X in the symbol table the run time location of X, and its type (real, formula, etc.) are retrieved and the description of E in the stack is set to contain the run-time address, the type bits, and a bit to denote that the location rather than the value of X is desired. This description will be carried along as the associate of E until code is compiled to perform the assignment. Thus the FSL code for EXEC 9 looks as follows:

```
9 ↓ MARKJUMP [FIND];
   SIGNAL →
   RIGHT2 ← KEY + MODE0
   + TYPE + RELOC;
   FAULT 9 $ ↓
```

```
“FIND”
   KEY ← SYMBOL [LEFT2 , $ , ] ;
   TYPE ← SYMBOL [0 , , $ ] ;
   RELOC ← SYMBOL [0 , , $ ] ;
   JUMP [ < FIND > ]
```

Upon entrance to EXEC 9 we execute a mark transfer to a closed subroutine called FIND which performs the symbol table look-up using the integer given in the description in the LEFT 2 entry (same as L2 position defined above) in the push-down stack. LEFT2 is a variable whose value is this description. It is used in the statement KEY ← SYMBOL [LEFT2 , $ , ] to locate the entry in the symbol table named SYMBOL which begins with the integer given by the value of LEFT2. Each line in the symbol table is of the form [integer, location, type, relocation base]. The relative position of the dollar sign $ among the commas indicates which of the entries in the located line we wish to extract.

Hence the statement KEY ← SYMBOL [LEFT2 , $ , ] extracts the location assigned to the variable whose internal integer is the value of LEFT2, and assigns this location to be the value of the variable KEY. If a zero occurs in place of LEFT2, as in the statement TYPE ← SYMBOL [0 , , $ , ], the extraction defined uses the line previously selected saving the cost of an additional identical look-up. Thus for our example the routine FIND simply sets the KEY to the location of X, TYPE to the type of X, and RELOC to the relocation base of X (the relocation base is used to implement recursion and is too complicated to explain here). The statement JUMP [ < FIND > ] is a return to a mark transfer call. Returning now to the consideration of EXEC 9 we assume we have executed a mark transfer to FIND and have returned with either the signal set false to denote that the table look-up was a failure, or the signal set true and the variables KEY, TYPE, and RELOC set with the extracted values. The statement SIGNAL → RIGHT2 ← KEY + MODE0 + TYPE + RELOC : FAULT 9 $ is equivalent to the Algol statement if SIGNAL = TRUE then RIGHT2 ← KEY ∨ MODE0 ∨ TYPE ∨ RELOC else PRINT ("SEMANTIC FAULT 9") where MODE0 is a variable containing a bit denoting that a location rather than a value will be used. Executing this statement causes the logical union of the values of the variables to be stored as the description of the element in the R2 position of the stack in the event that the signal was true, and it causes a semantic fault to be printed otherwise.

At this point in the consideration of our example, control is returned from EXEC 9 back to the production that called it. This in turn causes another character to be scanned and control to be transferred to E1 which is the beginning of the expression scanner. The expression scanner contains two main parts, one starting at E1 which expects an operand, as would be the case, for instance, at the beginning of an expression, and the other starting at E2 which expects an operator or separator. Thus upon transferring control to E1 we will find the following set of productions:

```
E1 I | → E | EXEC 7 *E2
E2 <OP> | | SUBR COM *E1
```

At E1 the first production matches and control is transferred to EXEC 7 with the syntax units in the stack looking like E ← E . EXEC 7 is roughly the same as EXEC 9 the main difference being that the description of the E in the RIGHT 1 position is set to contain a bit denoting that the value rather than the location of the variable is desired. So far X ← A has been scanned and converted to E ← E. We now scan the operator + and transfer control to E2 in the productions. The expression <OP> stands for a class of binary operators including the
The semantic routines used to accomplish this test the types of the operands, which types have been stored in the descriptions assigned by EXEC's 7 and 9, and they compile the appropriate code. At the completion of this compilation the syntax stack is:

\[ E \leftarrow E + E \times E; \]

Here the expression \(<SG>\) matches the semicolon on top of the stack at production COM+15 and control passes to production H16. The first production to match the stack is production H30. This leads to the first instance of object code compilation in the processing of the statement. All previous actions up until this point have consisted of postponements. The compilation is accomplished by transfers to EXEC 100 and to EXEC 125 which compile code to multiply B and C. In the case of arithmetic operands CLA B is constructed. In the MPY C case of formula operands, code is produced to construct when executed the formula tree

\[ B \times C \]
altered to look like $E \leftarrow E + E; |$ because the
terminal $E \times E$ has been replaced by a single $E$ as
is seen by inspecting production H30. The semantic
routines also set the description of the topmost
(rightmost) $E$ to contain the type of the expression
and the fact that the code compiled leaves the value
of the expression in the run-time accumulator. Con-
control now passes back to the beginning of subroutine
COM for another iteration of the process. Sub-
routine COM will be seen to reenter itself iteratively
until the entire expression is consumed, until code
for it has been compiled, and until its external rep-
resentation in the syntax stack has been replaced by
$E$ in the case of pure expressions and nothing in the
case of assignment statements. We are now at the
point where the syntax stack looks like $E \leftarrow E + E; |$ and where we have reentered COM. On this
pass production COM+15 matches and passes control
to H16 where successive productions fail to
match the syntax stack until production H28, at
which point $E + E$ is compiled by EXEC 100 and
EXEC 123. The compile-time routines responsible
for producing code detect the fact that code has
been compiled leaving the value of the second
operand in the run-time accumulator. Thus the
code compiled is ADD A. Again the semantic
routines analyze the types of LEFT2 and LEFT4
to determine whether code should be compiled to
add numerical expressions or to add formula ex-
pressions. After compiling ADD A the stack con-
figuration is changed to $E \leftarrow E; |$ and control
passes back to the beginning of subroutine COM.
On this final trip through subroutine COM pro-
duction H16 constructs code to perform the assign-
ment of LEFT2 to LEFT4 and subroutine COM is
exited with only the semicolon remaining in the
syntax stack, the assignment statement having been
consumed entirely. In the case of expressions,
rather than assignment statements, an $E$ is left upon
exit in the RIGHT2 position with its semantic
description set to contain its type and the fact that
it resides in the run-time accumulator.

The strategy of subroutine COM comes from a
well-known compiling technique for which no claim
to originality is made. Both Floyd and Evans have
used similar techniques in their Algol compilers.

FLEXIBILITY OF THE SYNTAX ANALYZER

We will now present examples showing how we
can experiment with, extend, redesign or improve
the syntax of the source language.

Suppose we want to add a new type of binary
logical operator NOR (equivalent to the Pierce
operator familiar to logicians and electrical en-
gineers) and suppose we choose to denote it by the
pair of characters $\sim \lor$ in the source language. Let's
agree that in the expression $\sim A \land B \lor C$
and $\lor D$ the NOR operator binds less tightly than
$\sim \land$, and $\lor$ so that fully parenthesized the ex-
pression looks like $( (\sim A) \land B ) \lor$ $(C \lor D)$. We need to do four things to add this
operator to the source language: 1) we declare the
character pair $\sim \lor$ to be a primary syntactic unit
for subscan; 2) we expand the definition of the class
of operators $< \langle \rangle >$ to include $\sim \lor$ so that the pro-
duction labeled E2 (above) will detect the presence
of this new operator and will pass control to sub-
routine COM. The last two steps are additions to
subroutine COM itself: 3) we insert the production
$\sim \lor |$ | HNOR

after production COM+12 (cf. Fig. 2); and finally
4) we insert the production

| HNOR E \sim \lor E <SG> | \rightarrow E <SG>
| \rightarrow EXEC K COM

after production HA1+1 (cf. Fig. 2). Here EXEC
K must compile code for Boolean expressions using
the NOR operator and it looks as follows:

K ↓ TEST{LEFT4, BOOLEAN} \land TEST{LEFT2,
BOOLEAN} \rightarrow CODE{RIGHT2 \leftarrow (LEFT4 \lor LEFT2);} \downarrow

This is our first example of the use of the code
brackets CODE (. . . ) . An expression con-
tained in code brackets describes code to be gen-
erated and inserted into the object program. The
test commands test the descriptions in the LEFT4
and LEFT2 positions to see if they contain bits de-
noting the type BOOLEAN.

From this example we see that we can 1) add op-
erators and choose their hierarchies at will, 2)
change the hierarchy of an operator without chang-
ing the operator itself, 3) by redefining an EXEC
routine, change the meaning of an operator without
changing its syntax, and 4) delete operators at will.
This exemplifies the kind of experimental flexiblility
available to users of the FSL system.

The organization of the compiler is such that
other kinds of additions, deletions and alterations
may be performed in the syntax analyzer with ease.
For example we may add a new type of variable by
adding its declarator to various lists, by inserting a
production in the production subroutine that processes declarations, and by inserting tests and consequent compiling actions in the semantic routines which are called corresponding to expressions and statements using the new type of variable. It is also easy to add new types of statements to the statement scanner and to add new types of expressions to the expression scanner. The implementation of the list processing part of the Formula Algol language demonstrated to us the ease with which it was possible to extend the compiler. Unfortunately, space precludes an adequate description of this extension.

As a last example of a notational change we consider the evolution of the notation for formula patterns \( F \text{ INST } P \) as defined in the first Formula Algol paper.\(^{13}\) \( F \text{ INST } P \) is a Boolean primary which tests whether the formula \( F \) is an instance of the formula pattern \( P \). Later this notation was changed to \( F = = P \) and an additional type of test was added of the form \( F >> P \) to test whether \( F \) contains a subexpression which is an instance of \( P \). In the case of changing \( \text{INST} \) to \( = = \) one merely had to substitute \( = = \) for \( \text{INST} \) in the productions. When \( >> \) was added it was possible with a minor correction to share the productions for \( = = \) to process \( >> \). This correction consisted essentially of substituting a class symbol representing the set \{ \( = =, >> \} \) for occurrences of \( = = \) in certain productions. The semantic routines associated with these productions were also altered to compile different code for the two different cases.

**FLEXIBILITY OF THE SEMANTIC ROUTINES**

Having treated some examples of the flexibility of the syntax analyzer we now turn our attention to corresponding properties of the semantic routines. The following example is intended to demonstrate the kind of experimentation that can be done with the compiling processes in order to improve both the compile-time efficiency and the parsimony of the formal description of the compiler. The example deals with a type discrimination problem encountered in the compilation of expressions involving unary and binary operators. For instance, when code is generated for \( A + B \), the types of \( A \) and \( B \) must be checked. \( \text{REAL} + \text{REAL} \) is well defined, the operands are already complete, and the result is of type \( \text{REAL} \).

\[ \text{FORMULA} + \text{REAL} \] is an illegal construction.

We shall describe two methods of implementing this type discrimination (which is the function of EXEC 100). The first is the most recent obsolete method and the second is its successor.

In the former method the set of operators was partitioned into a small number of equivalence classes. Two of these are the arithmetic operators \(+, -, \times, =, >, \ldots\) and the Boolean operators \(\lor, \land, \sim, \ldots\). For arithmetic operators, EXEC 100 checks the types of the operands for compatibility and sets a switch (MACHINE) to 2 if either operand is of type \(\text{FORMULA}\), and to 1 if both operands are arithmetic. In the following version of EXEC 100, \(X7\) is an address extractor.

\[
\text{100} \quad \text{TEST[LEFT4, BOOLEAN]} \lor \text{TEST[LEFT2, BOOLEAN]} \rightarrow \text{FAULT 100}:
\]

\[
\text{RIGHT2} \leftarrow \text{RIGHT2} \land X7;
\]

\[
\text{TEST[LEFT4, FORMULA]} \lor \text{TEST[LEFT2, FORMULA]} \rightarrow \text{MACHINE} \
\]

\[
\leftarrow 2;
\]

\[
\rightarrow \text{ CODE( CONSTRUCT FORMULA [LEFT4])$};
\]

\[
\rightarrow \text{ CODE ( CONSTRUCT FORMULA[LEFT2])$}:
\]

\[
\text{MACHINE} \leftarrow 1;
\]

\[
\text{TEST[LEFT2, REAL]} \lor \text{TEST[LEFT4, REAL]} \rightarrow \text{SET[RIGHT2, REAL]}:
\]

\[
\text{SET[RIGHT2, INTEGER]$}$
\]

As more operators and data types are added to a language this method becomes complex and inefficient both with regard to the space required to express the sequence of tests and the average time required to execute such a sequence. Therefore we have invented a successor to the above method.

The successor is described as follows. It is based on a single four-column table (DISCR) which may be preloaded. The first entry in a row of this table is a coded word which has three fields:

\[ \text{[TYPE1, OPERATOR, TYPE2]} \]
Any combination of two types and an operator may be described in a single word. Unary operators have one of the two types fixed. The second entry indicates a compile time routine to be executed which makes the operands compatible. The third entry points to a compile time routine which actually compiles code for the expression. The final entry is the type of the result. DISCR may be initialized as follows:

```
REAL + REAL    none  ADD REAL
REAL + INTEGER none  ADD REAL
FORMULA + REAL CONS2 FADD FORMULA
FORMULA ∨ BOOLEAN CONS2 OR FORMULA
NONE ~ BOOLEAN none  NOT BOOLEAN
```

When the syntax stack matches any production of the form

```
E <OP> E <SG> 1 → E <SG> |
```

then the following code is executed:

```
189 ↓

COMB ← (LEFT4 × L12 + LEFT2);
R ← DISCR[ COMB, $, ];
~ SIGNAL ← FAULT 189 : FINAL ← DISCR[ 0, $, ];
TYPE ← DISCR[ 0, $, ];
R ≠ 0 ← MARKJUMP[R]$;
JUMP[FINAL]$ ↓
```

Here L12 and L6 are shift constants, and LEFT3 contains a small but unique integer representing the operator.

In some cases we may experiment with the organization of compile-time processes to improve the quality of the object code produced, by which is meant we can reorganize some processes so that they produce less code which is more efficient. A small example of this is as follows. Certain patches of object code may be defined as nonexecutable because the flow of control may not enter them directly. For example, control must bypass a procedure declaration which may be entered only through a procedure call. If there are several adjacent procedure declarations then one may jump around each of them in turn or, preferably, one may jump around all of them simultaneously. The latter scheme is preferable because the object code requires less machine space, it runs slightly faster, and it looks less complex to the programmer trying to debug the system. The actual scheme for compiling these jumps has changed several times because we were able to try one method, tear it out, and try another both in the same day.

**ORGANIZATIONAL EFFICIENCY IN THE COMPILER**

When planning the structure of a compiler written in FSL we can take advantage of an organizational principle commonplace in programming, which states that when performing a class of operations which have certain common processing requirements we should, if possible, make a division of labor allowing the common processing requirements to be treated by a single shared routine. It is easy to apply this principle when writing EXEC routines since we can write a single EXEC routine performing labor common to several different compilation processes and we can share it in conjunction with other EXEC's to perform each separate process required. An example of this is EXEC 100 which is shared in the compilation of arithmetic expressions as is seen by looking at subroutine COM (Fig. 2). Another example is EXEC 160 which does everything common to procedures and blocks.

Production subroutines may also be shared. It may occur that certain syntactic constructs are used in different places in the source language with different semantics. For example, a list of identifiers can be used as a variable list in a declaration, as an array name list, as a formal parameter list, as a value list and as a specifier list. The productions in the syntax analyzer are written so that all identifier lists, no matter the context in which they occur, are processed by a common subroutine of the form:

```
ID | → | EXEC 190 * AID
<SG> | → | ERROR 190 AID
AID | → | * ID
<SG> | → | RETURN
```

As is seen, this production subroutine transfers control to EXEC 190 with the integer corresponding to the identifier on top of the stack. It does this for every identifier in the identifier list. In each of the different contexts of an identifier list it is necessary to process the identifier list in a different manner. To accomplish this, EXEC 190 is made into a variable capable of containing transfers to other EXEC's. For example, when, in FSL, the statement XEQ 190 ← XEQ 2 is encountered, it means the next time EXEC 190 is called, EXEC 2 will be executed. This will cause an identifier list to be processed as a variable list by the semantics. Similarly the statement XEQ 190 ← XEQ 3 will cause EXEC 190 to call EXEC 3 thus allowing an identifier list to be processed as a list of array names. By this mechanism one can treat the same syntactic
construct differentially in the semantics on the basis of context.

The addition of the XEQ construct to FSL is an example of the effect of feedback from the process of implementing Formula Algol on the design of FSL.

FORMULA MANIPULATION

It was decided to represent formulas inside the computer as trees or list structures built from cells taken from an available space list in a standard linked list memory. To add formula manipulation to the source language formula variables were introduced. In most cases the syntax already existing for numerical Algol was shared for formula manipulation. While no changes in the productions were necessary for this shared syntax, tests had to be added to the semantic routines to discriminate between numerical and formula compiling operations. For the new constructions added to the source language such as EVAL, = =, > > , and the extraction operator in patterns, additions were made to the productions and semantic routines were defined for them. Because most actions involving formulas are either interpretive at run time or involve manipulations which cannot be compiled into the object code as macros due to the size of the code involved, a set of run-time routines were constructed in machine code. These run-time actions constitute, more or less, a basic order code for formula manipulation. In effect, the compiler produces code for two machines, one an interpreter accomplishing formula manipulation and the other the hardware accomplishing numerical manipulation. For example, we saw (Fig. 2) that in subroutine COM EXEC 100 and EXEC 123 are called in sequence when we compile code to add two operands together, E + E. As we have seen previously EXEC 100 checks the types of the operands and sets a switch (MACHINE) specifying the machine for which we are to compile code. The structure of EXEC 123 is as follows:

123 ↓ MACHINE = 1 → CODE (RIGHT2 ← LEFT2 + LEFT4);
CODE (X1 ← LEFT2; RO ← '+ '+;
ACC ← LEFT4;
MARKJUMP (CONSTRUCT FORMULA); RIGHT2 ← ACC)
$↓$

Here the routine CONSTRUCT FORMULA is a basic operation of the formula manipulation machine which expects a right operand in X1, a left operand in the accumulator, and an operator in RO. Using cells from the list of available space it constructs a tree structure representing the sum of the operands and leaves the address of the head of this tree structure in the run-time accumulator.

The reader can now see how we could implement complex arithmetic by defining yet a third machine, which performs complex operations, and by extending the compiler by the same process used to accomplish the formula manipulation extension.

LIST PROCESSING

We will consider one example of list processing to try to convey some of the flavor of the mechanisms involved. Consider the statement INSERT [A, B, C] (AFTER LAST, BEFORE FIRST T) OF S. Here we assume that S contains a list (represented by a chain inside the computer). For the sake of specificity let S contain the list [ V, T, V, V] where V and T have been declared of type SYMBOL. After the insertion statement is performed the list is to look like [ V, A, B, C, T, V, V, A, B, C]. In a manner similar to that for formula manipulation a list processing machine is defined with an order code represented by a set of run-time routines. The compiler compiles a sequence of list processing instructions chosen from this order code corresponding to each list processing statement. Basic to the operation of the list processing machine is a push-down stack extant at run time called the chain accumulator. Most of the run-time list processing operations consist of manipulations on chains stored in the chain accumulator. The code produced by the compiler corresponding to the statement INSERT [A, B, C] (AFTER LAST, BEFORE FIRST T) OF S is a sequence of list processing operations whose mnemonics are as follows:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>STACK A</td>
<td>(on top of the chain accumulator)</td>
</tr>
<tr>
<td>STACK B</td>
<td>(on top of the chain accumulator)</td>
</tr>
<tr>
<td>CONCATENATE</td>
<td>(the top two chains in the chain accumulator)</td>
</tr>
<tr>
<td>STACK C</td>
<td>(on top of the chain accumulator)</td>
</tr>
<tr>
<td>CONCATENATE</td>
<td>(the top two chains)</td>
</tr>
<tr>
<td>GO TO \p:</td>
<td>CLA symbol to denote last</td>
</tr>
<tr>
<td>FIND POSITION</td>
<td>(this routine locates the last element of the chain)</td>
</tr>
</tbody>
</table>
Let us now trace the effect of the execution of this code on the chain accumulator. We will adopt the symbolism that \( \phi \) represents the state of the chain accumulator before we start to execute the code. As we enter the code we build up the list \([A, B, C]\) and stack it on top of the chain accumulator. This proceeds in the following steps. First we stack \( A \) on top of \( \phi \) producing \( A \phi \). We then stack \( B \) on top of this producing \( AB \phi \). Then we concatenate the top two chains on the chain accumulator producing \( AB \phi \), where \( \phi \) has been used as a symbol for the concatenation of chains. Next we stack \( C \) producing \( AC \phi \), where \( \phi \) has been used as a symbol for the concatenation of chains. Then we stack \( B \) on top of \( AC \phi \) producing \( AB \phi \), where \( \phi \) has been used as a symbol for the concatenation of chains. Finally we stack \( S \) producing \( S \phi \) and take its contents producing \( V \cap T \cap V \cap V \mid A \cap B \cap C \mid \phi \). Control now returns to \( \rho \) where we compute and stack a pointer to the last element of the contents of \( S \) giving \( V \cap T \cap V \cap V \mid A \cap B \cap C \mid \rho \mid \phi \). This pointer is moved to the third position in the chain accumulator producing \( V \cap T \cap V \cap V \mid A \cap B \cap C \mid \rho \mid \phi \). A second pointer is now computed and stacked. It points to the position before the first \( T \) in the chain on top of the chain accumulator giving \( V \cap T \cap V \cap V \mid A \cap B \cap C \mid \rho \mid \phi \). We could continue in this fashion computing and stacking as many pointers as we wish, each pointer corresponding to a place where an insertion is to be performed. We now transfer control to a routine which actually performs the insertions. This routine pops the chain \( V \cap T \cap V \cap V \) from the top of the chain accumulator and inserts a copy of the chain \( A \cap B \cap C \) after the position given by each pointer in the chain accumulator looping until all pointers in the chain accumulator are exhausted. The state of the chain accumulator after the execution of this statement is \( \phi \). Control in the code now passes to \( X \) where the execution of the program continues. The reason for the existence of transfers in the code sample given is because the order of recognition of syntactic constructions in the insertion statement in the source language is the reverse of the order in which we utilize these constructions in the computation expressed by the code. Specifically we must stack \( S \) and compute its contents before we compute any pointers locating positions in the contents where insertions are to be performed. However, the constructions telling us where to make insertions are encountered in the source language before we encounter the expression telling us the object on which the insertions are to be performed. Floating addresses are used in the compiler to implement such reversals.

Since the semantics of the source language demands that all insertions be performed simultaneously we are forced to compute all locations where insertions are to be made before performing any insertions.

**CONCLUSIONS**

In this paper we have outlined the broad organization of the Formula Algol compiler. We have also presented examples exhibiting various proper-
ties of the FSL compiler writing system. We have not described completely or, in some cases at all, the implementations of declarations, switches, arrays, for statements, recursive procedures, block administration, formula manipulation or list processing. For a complete and detailed treatment of these the reader is referred to another paper by the authors, "The Implementation of Formula Algol in FSL." The subject matter was chosen to reveal what we feel to be interesting techniques involving the use of a formal compiler writing system.

ACKNOWLEDGMENTS

The authors are deeply indebted to Professor Alan J. Perlis who guided and inspired our effort. Many of the creative and original ideas presented are his. However, we alone remain responsible for errors of style or content.

REFERENCES

12. R. Krutar, FSL II, Carnegie Institute of Technology, Computation Center, internal publication.