COMMEN: A new approach to programming languages

by LEO J. COHEN

Leo J. Cohen Associates
Trenton, New Jersey

INTRODUCTION
The Compiler Oriented for Multiprogramming and Multiprocessing Environments is an outgrowth of the FADAC Automatic Test Analysis Language (FATAL) developed for the University of Pennsylvania under their contract with the Frankford Arsenal to investigate techniques for the automatic check-out of electronic circuits. The purpose of the FATAL Compiler was to prepare programs for execution in the FADAC Computer. This computer has a two-wire teletype input-output system that can be used to establish connections between a unit under test and a set of test equipments through an interface device. The configuration for the check-out system is shown in Figure 1.

Figure 1 - FATAL checkout configuration

Numerical and logical techniques for the automatic check-out of electronic circuits were extensively developed. Their application to automatic check-out by the FADAC Computer required the development of a programming language that would allow the expression of both numerical and logical techniques necessary. In addition, the programming language necessarily had to provide statements for accomplishing the various connections and disconnections between the test equipment and the unit under test. The difficulty in the development of the FATAL compiler derived from the fact that the interface device was not in existence and the commands for effecting the various check-out functions were unknown. As a consequence it was decided to develop the FATAL compiler along lines that would permit new statements to be specified as desired. A further objective was to allow the specification of the meaning of these new statements via the already existing FATAL language.

The COMMEN compiler is an extension of the techniques developed in the FATAL project. The name for this compiler derives from certain invaluable properties that it possesses relative to multi-access systems (multi-programming, time sharing) and from its unique capability for automatically developing the control of a parallel process. The arrangement of the name into the acronym “COMMEN” was designed to indicate its most important application as a language unification technique, providing a single language for system command, computer programming, program documentation, and user-program dialogue.

The syntactical categories
The philosophical approach to the COMMEN compiler is perhaps unique in that it implements the translation of four mutually exclusive syntactical statement types. This is in opposition to compilers which determine a set of statements for which an implementation is provided.

The four syntactical categories for the statement types are declarators, assignments, procedure calls and verbs. Statements falling into the second and third syntactical categories are familiar from ALGOL, and generally adhere to the structural rules and techniques of application defined there. The syntactical form for declarators in COMMEN is an alphanumeric string terminated by “.”. The alphanumeric string-plus-comma is interpreted as the declarator operator, and the alphanumeric string as the name of the declarator. Thus, the names of the declarators of the
system are not reserved words. In fact, there are no reserved words at all in the system so that the user has complete freedom in the construction of new statements.

\[ A_1(P_1)A_2(P_2) \ldots A_n(P_n) \ldots \]

\[ A_i = i^{\text{th}} \text{ alphanumeric string} \]

\[ P_j = j^{\text{th}} \text{ parameter set} \]

Examples

\[ \text{IF} (A) \text{ GTR (B)} \]
\[ \text{FOR} (\text{CNT}) \text{ EQL} (\{2,5,5\}(A+B)) \]
\[ \text{COMPARE} (\text{VALUE}) \text{ TO} (AX-B, AX+B) \]
\[ \text{COMPARE} (\text{VALUE}) \text{ TO} (AX) + \text{ OR } -(B) \]

Figure 2—General verb syntax

The most important syntactical category is for the set of statements that we have defined as verbs. In particular, verbs differ from declarators in the following sense. Declarators may be considered as commands from the user to the compiler for compiler time action, whereas verbs may be considered as commands from the user to the computer itself for object time action.

The syntactical form of a verb is described as an indefinite sequence of character strings and parameter sets enclosed in parentheses. A parameter set is defined as a string of parameters separated by commas. Figure 2 gives a pictorial representation of the general syntax for verbs and several examples. Note that though these examples of verbs may be suggestive of the functions that they represent, their exact meaning has yet to be specified. Since the functions that a compiler language statement offers to the user are defined in terms of the object machine code that implements it at execution time, the technique for the specification of a new statement in COMMEN is to allow the user to provide the necessary programming for his new statements. Further, it allows him to provide this programming in the form of COMMEN statements. In the subsequent sections of this paper this idea is made more specific, and implications for programming are developed in an example.

Phrase structure

The notion of phrasing and phrase structure is of basic importance in a compiler language if it is to allow the user a high degree of similarity between his flow chart and his compiler language program. In general, a phrase is defined as a single statement, or as a sequence of statements that are designated collectively as a phrase.

\[ \text{IF} (A) \text{ GTR (B)} \]
\[ \text{BEGIN}, \]
\[ Z = A - B \]
\[ Z_1 = Z^* Z/5.0 \]
\[ \text{GO} (\text{NEXT}) \]
\[ \text{END}, \]
\[ Z = B - A \]
\[ Z_1 = Z^* Z/1.5 \]
\[ \text{GO NEXT} : \]

Figure 3—Phrase structure, ALGOL & FORTRAN

The crudest form of phrase structure is found in FORTRAN IV where only the statement following an IF is treated as a phrase. A second example from FORTRAN is where the collection of statements within a DO loop is treated as a phrase.

An example of phrase structure in FORTRAN IV is shown in Figure 3 along with a comparative example of phrase structure as it is employed in ALGOL. In Figure 4 the phrase structure technique used in COMMEN for the same example is shown. As can be seen, the technique for phrase structure in COMMEN is virtually identical to that used in ALGOL with the two examples differing essentially in the syntactical structure of the verbs (IF and GO). This is to show that in general the two languages
agree on the application of phrasing relative to the control statements that use them. Thus, in these two examples for ALGOL and COMMON, the IF implements an implied reference to the first, as well as to the second, phrase following this statement.

The notion of first, second, etc., phrase following a statement is of basic importance in the development of the COMMON language technique. This importance derives from the fact that in order to specify the relationship between a new statement that the user is constructing and the phrases to which he might wish it to relate, we must have some concept of relative phrase location, or as it will be used in what follows, implicit phrase address.

Figure 5 is an example of a collection of statements that have been organized into phrases and sub-phrases and which consequently determine relative phrase relationships. In this example statement $S \phi$ is followed by four phrases, so that the third phrase following statement $S \phi$ is actually the statement $S12$. Within the first phrase following $S \phi$ there are three phrases consisting of the statements $S1$ and $S2$, and the 10 statements concatenated into the second phrase following $S1$. Thus, by virtue of position, each phrase at a certain level of the nesting may be said to be in a position relative to any particular statement at that same level. This position relative to a statement will be referred to as the implicit phrase address of this phrase for that statement.

The other type of addressing possible in a compiler language program is the explicit phrase reference wherein a label (an alphanumeric string terminated by ‘.’ ) is used to designate a phrase. References to such a phrase may be made via an explicit reference to the alphanumeric string associated with the label, as in the GO verb of the present example.

Some examples of COMMON verbs which are familiar from ALGOL and which contain references to the first and second phrases following their ap-

\[ \text{FOR (integer \_identifier) \_EQL (initial \_step \_final) value} \]
\[ \text{UNTIL (expression \_relation \_expression)} \]
\[ \text{IF (expression \_relation \_expression)} \]

Figure 6-Some COMMON verbs

specification are shown in Figure 6. In the first of these examples, the FOR verb refers to the first phrase following its use as the one over which the control of iteration is exercised. Once the iteration has been completed the implementation of the verb causes transfer to the second phrase following the statement. The first and second phrases following the UNTIL and IF verbs are utilized in a similar way, and as in the case of the FOR, their exercise is designated implicitly as part of the meaning of the verb itself. It should be pointed out here, of course, that within the first phrase following any one of these three verbs, an explicit transfer may be to some other phrase of the program.

Specification of new verbs

The language of ALGOL is open ended to the extent that the user is allowed to declare and define new procedures in his program. A declarator is used to notify the compiler of the name of the procedure and of the formal parameters that are used when it is called. The declaration of the name for the new procedure is then followed by compiler language programming which supplies a definition of the procedure, and that in turn is followed by a termination declarator which informs the compiler that all definition programming for the procedure has been received. Once a procedure has been completely declared and defined to the compiler, the user is free to call this procedure in procedure call statements. Thus, this technique now allows the user to construct new statements of a specified syntactical form known as the procedure call.

The specification of new verbs in COMMON follows much along the lines of procedure specification. In this case the language of the verb, along with the formal parameters for the verb, are declared to the compiler via the VERB, declarator. This declaration is the notification to the compiler that the programming that follows is the definition of the verb, and that this definition will be completed by a termination declarator. The statements used for defining the meaning of a new verb may be any of the statements
An example of the declaration of a new verb is shown in Figure 7. The language of the verb which we wish to declare is \( \texttt{COMPARE (X) TO (Y,Z)} \) where \( X, Y \) and \( Z \) are formal parameters that may be replaced on use of the verb by actual parameters, as in the case of the actual parameters for procedure call statements. The statement form is declared to the compiler via the declarator \( \texttt{VERB} \), which also signals that all ensuing statements up to the occurrence of the declarator \( \texttt{END CODE} \), serve to define the meaning of this new verb. For this example, the meaning of the verb is given by the logic of the flow chart shown in the figure. The connectors in this flow chart refer to the first, second and third phrases following the use of the new verb. As such they represent symbolic references to these three phrases and will become implicit phrase addresses in the definition of the verb. Thus, the logic for the new verb requires that the first, second or third phrase following the occurrence of the new verb is executed if \( X \) lies respectively to the left of, within, or to the right of, the closed interval \( (Y,Z) \).

In this example, the first statement following the declaration is the \( \texttt{DONT CARE} \), declarator. It declares the formal parameters \( X, Y, Z \) to be either of floating or integer type. The next five statements encode the logic of the flow chart and refer to the symbolic phrase references as indicated. When the compiler recognizes the \( \texttt{VERB} \) declarator it stores away the format and language of the new verb in a verb library location. It then continues the compilation of the verb definition statements that follow, but diverts the result of the compilation of the statements to an area associated with the new verb. Thus the compiled version of the verb (plus information relative to its formal parameters and the symbolic phrase references) appears in the verb library and is available for application where required in the program.

With the new verb declared, the programmer is free to use it as he requires for the implementation of his flow chart. Figure 8 shows a pictorial example of the use of this new verb with actual parameters, and the three following phrases to which it implicitly refers. When this verb call is translated, the compiled programming for the verb that was placed in the verb library is now inserted in-line in the output code. The actual parameters are substituted for the formal parameters and the symbolic phrase addresses are translated into the equivalent of assembly language references to symbolic tags that will later be associated with the phrases to which they refer.

**Compounding verbs**

Thus far the verb appears as the analog of the macro instruction in an assembly language with the major difference that the verb may implicitly refer to other phrases in the program that have a position relative to the verb. The real power of this technique at the compiler language level however, lies in our ability to compound the language of verbs into a very high level, problem oriented language that may serve directly as the formal documentation for the program for the non-programming technician.

In general the technique is as follows: Starting from a basic set of verbs, they are used to define new verbs which expand the set. These in turn are used for still newer verbs, until the language of the resulting verbs is in suitable agreement with the language of the problem. In order to provide a sufficient degree of generality, \( \texttt{COMMEN} \) accepts an \( \texttt{OWN CODE} \), declarator which allows the insertion of assembly language programming. This declarator may appear within verb definition code.

The technique described above is illustrated by the following example. Suppose we wish to make available a statement by means of which we may determine whether or not a given variable is within a specified range of a second variable. Let us choose...
the following language for our verb, using the dummy parameters X1, X2, and X3; COMPARE (X1) TO (X2) + OR - (X3). Explicitly, we want this verb to transfer control to the first phrase following its occurrence if X1 is less than (X2 - X3), to the second phrase following if X1 is within the closed interval (X2 - X3, X2 + X3) and to the third phrase following if X1 is greater than (X2 + X3). Figure 9 shows the necessary programming for the declaration and definition of this new verb.

```
VERB, COMPARE (X1) TO (X2) + OR-(X3)
DONT CARE, X1, X2, X3
COMPARE (X1) TO (X2 - X3, X2 + X3)
go ($+1)
go ($+2)
go ($+3)
END CODE,
```

Figure 9 – Compounding verbs

```
VERB, SKIP NEXT PHRASE IF (X) WITHIN 10% RANGE OF (Y)
REAL, X, Y
COMPARE (X) TO (Y) + OR -(Y/10.0)
go ($+1)
go ($+2)
END CODE,
```

Figure 10 – The SKIP verb

Notice that the new verb used the previously declared COMPARE with properly selected parameters. The example assumes that this verb already exists in the verb library. Since the definition for a verb is in essence a self contained program whose only relationship to the environment in which it appears is through its formal parameters and its symbolic phrase references, the three transfer statements will be necessary to complete the definition of the new verb. This new form of the COMPARE verb is used to construct the compound shown in Figure 10. In this example the elision mark “...” is used to signify the continuation of a statement onto the next line.

```
VERB, EXECUTE (LABEL) IF (X) OUT OF 1000 RANGE
REAL, X
LABEL, LABEL
COMPARE (X) TO (900.0, 1100.0)
go [LABEL]
go ($+1)
go [LABEL]
END CODE,
```

Figure 11 – The EXECUTE verb

As an example of the high degree of problem orientation that can be achieved by this process consider the verb defined in Figure 11. For this verb the first formal parameter, LABEL, will be the name, i.e., explicit address, of some phrase in the program. The second formal parameter for the verb is the variable X, and for this case is defined to be a real (floating decimal) number. Our new COMPARE verb is then applied to the variable X along with the constants shown. If the value of X is out of the 1000 range, i.e., lies either to the left or to the right of the range, then a transfer of control is made to the selected phrase. Otherwise, if the value of X lies within the 1000 range a transfer is made to the next phrase following the occurrence of this verb.

```
TEST 1.. EXECUTE (ERROR 1) IF...
   (341.5X+K1) OUT OF 1000 RANGE
TEST 2.. EXECUTE (ERROR 2) IF...
   (395.8X-K1) OUT OF 1000 RANGE
   etc.
```

Figure 12 – Example of language orientation

The application of this verb in a general testing environment might be as shown in the example Figure 12. Here the just defined verb is used with various labels for the first actual parameter, each representing a specified routine to be executed in the event that the value of the second actual parameter is out of the 1000 range.

**CONCLUSION**

It was originally felt that the real value of compilers derived from the amount of labor that they saved the programmer by cutting back the number of instructions that he had to write in the compiler language relative to the number of instructions necessary to do the same job written in assembly language. As compilers came more and more into popular acceptance it has been readily recognized that their true value lies in the verbal facility they provide for the expression of a problem in a form that may then be translated into an operating computer program. More particularly, it is often the method used to express our understanding of a problem that shapes and informs our method of approach to its solution. Therefore, the higher the degree of problem orientation in the language with which we represent a problem for translation into computer executable terms, the closer we come to intellectual symbiosis with computing machines. The various simulation languages available are excellent examples of this.

The development of COMMEN is another step in this direction. As a programming technique its objective is to reduce to a minimum the difficulties inherent in expanding the problem expression language. At its ultimate application, it allows us to unify the several languages required to program, document, command, and converse with our large scale com-
puter systems. As an example of the application of COMMen to the language unification problem, see reference 3.

REFERENCES
1 Study of piece part fault isolation by computer logic V.3 Circuit analysis solvability and systems studies ICR Univ of Pa Philadelphia Pennsylvania 1964
2 Study of piece part fault isolation by computer logic V.1 User's Manual-FATA compiler assembler system ICR Univ of Pa Philadelphia Pennsylvania 1964
3 Additions to COMMen for Time Sharing Time Sharing Workbook Inst for Advanced Tech C-E-I-R 1200 Jefferson Davis Hwy Alexandria Virginia