COMPILER WRITING IN FORTRAN ON A SMALL COMPUTER*

Susan T. Dean, Programmer/Analyst
Steven E. Wixson, Senior Systems Analyst

Clinical Cardiology Computer Center
University of Alabama in Birmingham

ABSTRACT

This paper shares the experience gained in writing a compiler using FORTRAN on a relatively small computer. Technical details of the programs which make up the compiler are given, rather than a general discussion of practices of compiler writing. Practical suggestions to someone who might undertake a similar project are emphasized.

The language for which this compiler was developed is oriented toward interactive data entry from specialized terminals. An overview of this language is provided to demonstrate the types of statements the compiler must handle.

The compiler is a set of overlay programs ("links") which were originally required to fit into an 8K word (16 bit) batch partition. The organization of these processing segments, and the communication of information between them, are explained in detail. Key support subroutines are described. The compiler's output is a set of dynamically relocatable coded instructions and operand references which are stored on disk for later execution by an interpreter.

*Supported in part by Specialized Center of Research for Ischemic Heart Disease, Contract Number 1P17HL1766

BACKGROUND

The environment of the Clinical Cardiology Computer Center (CCCC) at the University of Alabama in Birmingham (UAB) requires programming of on-line displays, such as figure 1, and logic for interactive communication between the medical personnel and the computer. The CCCC computer is an IBM 1800 with 64K 16-bit words of memory. The languages available for use on this machine are basic FORTRAN and an assembler language. In both of these languages, a significant amount of programming effort is required to write terminal displays and the associated logic. Therefore, a special purpose language was implemented to fill this need.

The interactive display oriented language (IDOL) supports pre-defined interactions between a computer and a human via an on-line terminal. The basic unit of the language is the display frame, which includes the information to be displayed to the user and the logic for processing the user's response. The sample of IDOL source code (figure 2) and the display produced by it (figure 1) provide an example of the use of the language.

IDOL was designed in two versions. Based on a language, "Driver" (1), developed at Massachusetts General Hospital, the preliminary versions of the compiler and execution monitor (interpreter) were written entirely in FORTRAN. This version was implemented quickly and, although not optimal either in execution time or expandability, provided an opportunity to "live with" the language before proceeding with the final design.

The second version of IDOL (2) includes these features as improvements and special-purpose aids:

- Interpreter is written entirely in assembler language for rapid execution.
- Compiler and interpreter are both organized modularly to provide ease in adding or modifying language functions.
- IDOL is considered to be the main program who occasionally "calls" a portion of a companion FORTRAN program which provides graphic display or calculations not supported by IDOL. In the first version the display driver was a subroutine called for each new frame by a main FORTRAN program. The new version requires less user awareness of the transfer mechanism between frames.
- Source code for the frames may be on cards or disk.
- Variables defined to exist in standard system files or control blocks are coded as simple variables by the user.

Constraints on the design of the language and its compiler include:

- Originally the compiler was required to fit into an 8K (now 16K) batch partition.
- The interpreter was to fit into a 4K partition (now 8K).
- The display frame code is stored in disk files with fixed 320 word sectors.

THE COMPILER

The compiler is, of necessity due to computer memory partition size as well as through the desirability of this programming technique, a set of modules which can be overlaid, communicating through COMMON.

Figure 3 shows the relationship between the major overlays (main programs) which make up the compiler. As needed, each overlay is brought in as the program currently executing in the partition ("linked") leaving only the COMMON area undisturbed. Descriptions of these major programs ("links") are given in the following paragraphs. A list of the major variables in COMMON which provide communication between modules is given in figure 4.

The initial program (MLBLD) is the only compiler module the user is aware of, since all the "links" are transparent to him. MLBLD performs the setup functions described in figure 5.

The symbol table build module (MLSTB) shown in figure 6 became a separate "link" because of the original partition size. Conceptually it is part of the initialization process and could be an activity of MLBLD. However, since it is an easily separated function, it is an excellent break point between programs. MLSTB reads the data definition cards which are required to come before all display frame source code and creates a symbol table entry for each variable.

After all variables have been recorded in the symbol table, the display frame source code is processed. Each display frame is a self-contained unit, therefore the compiler concerns itself with only one frame at a time in a series of "links" which is repeated as many times as needed.

The first "link" for frame code processing is MLFIN (figure 7). To avoid attempting to fit all the FORTRAN I/O routines in memory at the same time with all the processing logic, MLFIN's primary purpose is to read the source code from disk or cards, list the statements at the user's option, and store the source statements into a work file on disk. All code processing modules store error indicators on disk for later printing at the end of the frame, considerably reducing the memory needed for the processing routines since only disk I/O is performed.

MLCMP (figure 8) is the main "link" in that it controls the actual frame code processing. It reads each source statement from disk, maintains control information within the frame (count of statements, etc.), and generates any automatic function pertinent to the type of frame (automatic transfer indexing, for example). The code for frame-to-frame transfer statements is generated by MLCMP itself. MLQRP (figure 9) is called to process text-defining statements, and MLHCS (figure 10) generates code for the logic-processing statements. Both MLHCS and MLQRP terminate by returning control to MLCMP. Figure 11 describes in detail the code generation process for a sample frame.

When all statements for the frame have been processed MLCMP calls MLERP to print messages for any error records which have been detected

during generation of code for the frame. MLERP then calls MLFIN to repeat the cycle for the next frame.

After code for all source frames has been generated and stored into a disk work file, MLPCK determines optimum packing of the frames' code and copies the code into the permanent disk file specified by the user.

The following subroutines support multiple compiler activities:

- The symbol table lookup routines.
- Expression parser (EXPR) - given an expression consisting of any combination of variables, constants and operators, uses the operator precedence method (3) of parsing the expression and then generates the appropriate code.
- Keyword recognizer - given a list of possible keywords and a candidate keyword, checks to see if the candidate is a keyword and, if so, returns the keyword number (its position on the list) and the position within the given statement of the first character after the keyword. This is used by the data definition processor to recognize the specification of the various global locations. Another use is by MLHCS to recognize keywords in statements (IF, GO TO, PUT, etc.) and names of builtin functions (e.g. HEADING for a standard set of information at the top of a display or RTEXT to extract the display text associated with a particular use response and assign it to a character variable.) The keyword number is then used in a computed GO TO to indicate appropriate processing logic.
- Source statement input routine - accesses the next source statement.
- MLERR - stores error number and statement number on disk.
- Variable reference generator - EXPR has subroutines to handle code generation for arithmetic operations, string operations, relational operations, etc., and they all need to check for proper variable type and generate the accessing code.

A variety of functions and subroutines permit one to think of a FORTRAN array as a character string. These are used by many CCCC programs besides the compiler.

MOVE - copy a specified number of words from one array to another.
INSTR - determine if a given word or array is matched in another array, and if so beginning at what position.
LSTNB - determine the subscript of the last non-blank word in a given array.
NONBL - determine the first non-blank word's position.
EBSC - convert EBCDIC representation of numbers to actual integer representation. (Needed for handling numeric constants.)

The error printing routine MLERP includes the only assembler language subroutine in the compiler. Defining the text of error messages in DATA statements is cumbersome. Therefore, the text is defined in an assembler subroutine which is given an error number and an array into which to store the corresponding text to then be printed by MLERP.

THE INTERPRETER

The interpreter, or execution monitor, is activated at execution time to execute the code stored in a frame file. This code consists of a stream of operators and operands, and is executed by use of a software stack. Any operands encountered in the input are "pushed" onto the stack by the execution monitor and are "popped" as needed by the operators.

There are seventy-five operators. Some are simple operations, such as addition, where two operands are popped, their values accessed and added, and the result is then pushed onto the operand stack. A more complex operation would be accessing a particular value from a system patient information block and storing it into a specified variable.

Each operation is coded as a separate assembler language subroutine. Advantages of this are:

- Modular organization made debugging easier.
- Modification of one operator does not affect others.
- Addition of new operations is easy, requiring only a new subroutine and an addition to the execution monitor's table of operators.
- Seldom-used operations can be "localed" against each other. (Only one of a set is in memory at any time, as a subroutine is brought in only when it is to be used.)

SUMMARY

Implementation of IDOL has added a much needed tool at the Clinical Cardiology Computer Center. Particularly important in developing this language were: 1) The opportunity to live with the first "quick and dirty" version and then from it design the version to be optimally implemented, and 2) The modular approach to the implementation, thus providing for future expansion of features and easier debugging.

The guidelines presented here are intended to make compiler writing seem almost simple. Given a model, it is easy to implement a special-purpose language designed to meet the needs of a particular installation. On a small machine where FORTRAN is the only high level language, a special purpose language can become so useful that the programming time it saves will easily recover the cost of developing the compiler.

REFERENCES

1. Swedlow, D. B., Barnett, G. O., Grossman, J. H., and Souder, D. E.: A Simple Programming System ("Driver") for the Creation and Execution of an Automated Medical History. Computer and Biomedical Research 5, 90-98, 1972.

2. Dean, S. T., Wixson, S. E., Garrett, H. M., and Harrell, F. E., Jr.: A User-Oriented Language for Interactive Medical Programming. Proceedings of the Southeast Regional ACM Conference, 122-139, 1975.

3. Gries, D.: *Compiler Construction for Digital Computers* (Wiley, New York, 1971), pp. 121-131.

4. Wixson, S. E., Strand, E. M., and Perlis, H. W.: A Computer System for Bedside Medical Research. 1970 Spring Joint Computer Conference Proceedings 36:475, 1970.

```
COMP ROOM T555555 TIME 1411. 2/10/77

 CORONARY SINUS BLOOD FLOW

SET APPROXIMATE DIAL VALUES TO
SIMPLIFY LATER CALIBRATION

SET INDICATOR DIAL TO     439
SET DILUTION DIAL TO      599
```

Figure 1. Display on Terminal Screen resulting from execution of frame whose source code is in figure 2.

```
N:  3,IN

H:     CALL PROG SETRATE;
H:     HEADING
H:     TITLE 5
H:     PUT(10,23) INDIC
H:     PUT(11,22) DILUTION

Q:     7,SET APPROXIMATE DIAL VALUES TO
Q:     SIMPLIFY LATER CALIBRATION
Q:     10, SET INDICATOR DIAL TO
Q:     SET DILUTION DIAL TO

T:  4

END OF FRAME 3
```

Figure 2. Sample Display Frame Source Code.

This generates terminal display shown in figure 1.
The CALL PROG causes a transfer to a companion FORTRAN program who uses the value of variable SETRATE in a computed GO TO to determine what section of itself to execute, then transfers back to the IDOL interpreter who resumes execution at the statement following the CALL PROG.
INDIC and DILUTION are global variables which have been defined by user as being in a portion of main memory which is accessible to both IDOL and the companion program because of the transfer mechanism (4).

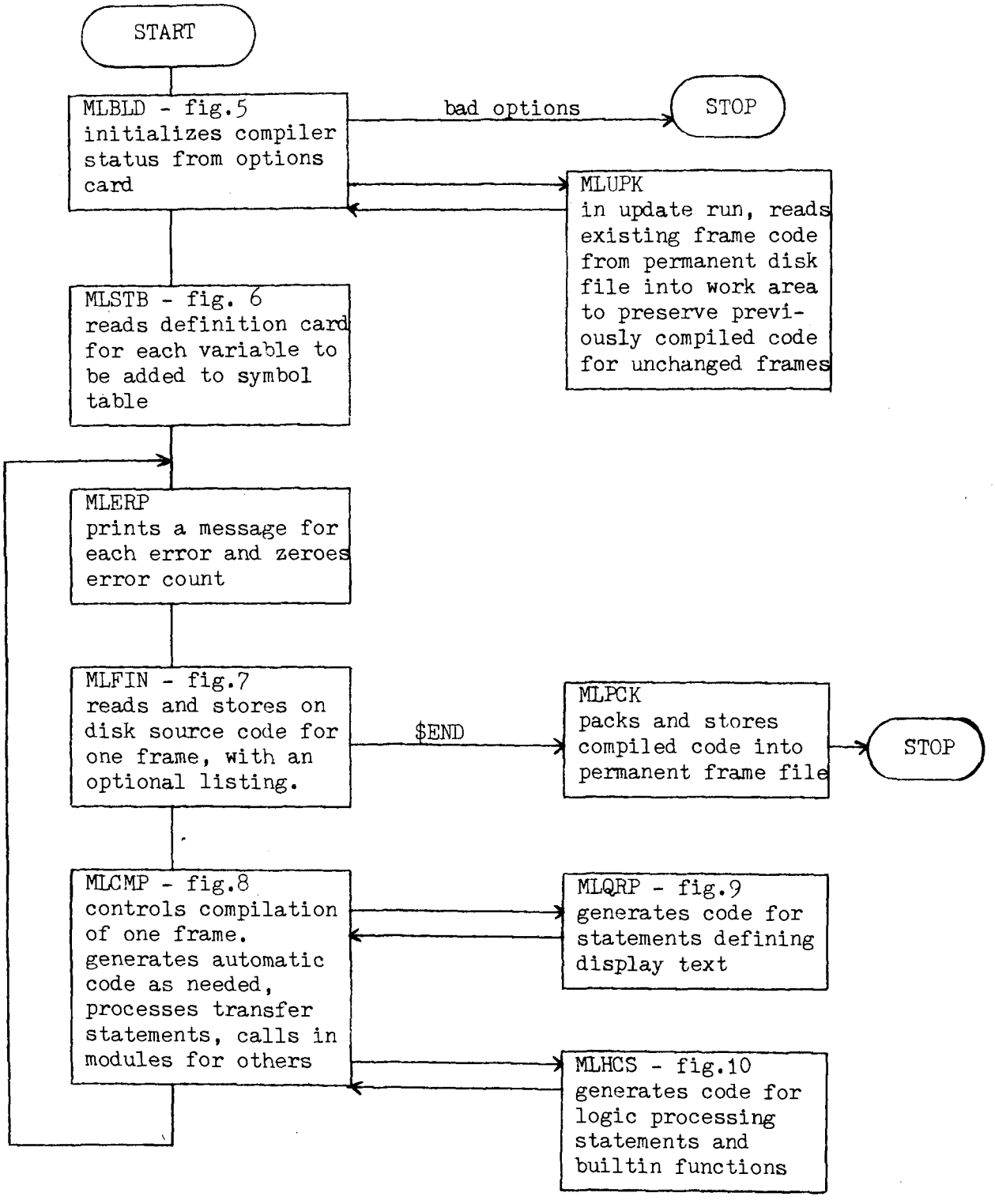

Figure 3. Major Compiler Modules.

SYMBOL TABLE - One entry for each variable (name, type, dimension, location). Also contains the frame file index, which gives the location of each frame's code within the file.

FRAME CODE - All code generated so far for the frame currently being compiled and the subscript of the next word to be used in this array.

SOURCE STATEMENT - The source code for the statement currently being compiled.

STATEMENT COUNT, TOTAL NUMBER OF STATEMENTS (current frame)

ERROR COUNT (current frame)

DISK ADDRESS OF PERMANENT FRAME CODE FILE

TABLE OF LABELS USED IN THIS FRAME and their ADDRESSES

TABLE OF UNRESOLVED LABEL REFERENCES THIS FRAME and FRAME CODE WORD where the address is to be stored.

"GET" indicators - Source input from cards/disk, Update/create run, Address of File for input from disk, and Other information needed by the "GET" routine.

Figure 4. Communication via COMMON between Compiler Modules.

---

1. Read the options card and process as follows:

2. If the specified frame file (where compiled code is to be stored) has not been allocated on disk, terminate with error message.

3. If "update" run,
read symbol table and existing frames code from frame file to disk work area;

Otherwise
If "create" run,
set up an initial-state symbol table containing only the automatic special-purpose variables and an empty frame index;

Otherwise
terminate with message of invalid option.

4. Set source-of-input (cards or disk) indicator in COMMON for the "get" routine.

5. Activate the symbol table builder (MLSTB).

Figure 5. Basic logic of MLBLD, the compiler initialization routine.

1. Repeat for each variable's definition card:

If this variable name is already in the symbol table,
record an error and ignore this definition.

Otherwise
Build symbol table entry (recording errors on any invalid specification):

Variable type - integer or character string of specified length,
Dimension for array or zero,
Location information for "global" variables. (This is the code to be incorporated into the frame code at any point of reference to this variable to activate the appropriate operator to retrieve or store a value in a standard system file or control block.)

2. Activate the error message routine (MLERP) to print a message for each error that has been recorded on disk.

Figure 6. Basic logic of MLSTB, the symbol table builder.

---

1. Set up frame number and type from frame's "N" card (1st card).

2. Repeat for each statement until END (of frame) statement is encountered:

If the $END card is found, exit this routine and activate the frame packer (MLPCK) to permanently store all frame code on disk.

Read ("GET") the next source statement.

Increment statement counter.

Print the statement unless printing is suppressed.

Write the statement to the disk working file.

3. Activate the frame compiler (MLCMP).

Figure 7. Basic logic of MLFIN, the routine to set up compilation of one frame.

1. *Repeat varying N from 1 to number of statements in this frame:*

   Read Nth source statement from disk work file.

   Generate statement's address into pointers section of frame code and into label table, if statement is labeled.

   Determine statement type and take appropriate action:

   'H' statement (heading logic) -
   Activate logic statement's code generator (MLHCS).

   'Q' or 'R' statement (question or response) -
   Activate display text code generator (MLQRP).

   'C' statement (control) -
   Generate automatic operation code depending on frame type (operators providing temporary exit for user response, numeric input operator, etc.)
   Activate logic statement's code generator (MLHCS).

   'T' statement (frame transfer) -
   If multiple choice frame type and first T statement, generate automatic code to select proper transfer statement depending on response by user.
   Generates code defining number of next frame to execute, and operation code for transfer.

2. *MLHCS will have generated an entry in the Unresolved Label Table for any forward-reference GO TO's. From the unresolved and statement label tables, generate the proper addresses. Record errors for any which are still unresolved.*

3. *Activate error messages printer (MLERP).*

Figure 8. Basic logic of MLCMP, the frame compiler.

---

1. *Generate code for display operation.*

2. *Build, edit for errors, and generate control word indicating line number, character position in line, character size (normal, small, large), response number to display (R statement only), and number of characters to display.*

3. *Copy display text into frame code.*

4. *Re-activate frame compiler (MLCMP) to process next statement.*

Figure 9. Basic logic of MLQRP, the text code generator.

1. *Call keyword recognizer to determine type of logic statement.*
   If no keyword, assume assignment (SET) statement.

2. *Using a computed GO TO depending on keyword number, process one case as follows (record errors as detected):*

   2.1 *IF statement*
   Call EXPR to generate code for the relational expression.
   Generate branch-around-next-statement-if-false operator.
   Set up the "then" clause as a source statement and return to level 1.

   2.2 *SET statement*
   Change = to = = representing assignment.
   Call EXPR to generate expression code, target variable operand, and assignment operator.

   2.3 *HEADING statement*
   Generate operation code for display of standard heading information (patient I.D., date, time) at top of screen.

   2.4 *TITLE statement*
   Generate operand reference and opcode for generation of user supplied title at top of screen.

   2.5 *STOP statement*
   Generate code for operator which erases the screen and terminates execution.

   2.6 *CALL PROG Statement*
   Call EXPR to generate code for expression supplying a number to be used as switch value in computed GO TO by a companion FORTRAN program (TASK).
   Generates code for operation which stores that value to be accessed by TASK and activates the TASK transfer saving information necessary to return to this frame.

   2.7 *(Other types of statements are processed similarly, and new functions may be added here at a later time.)*

3. *Re-activate frame compiler (MLCMP) for processing of next statement.*

Figure 10. Basic logic of MLHCS, the logic statement code generator.

N: 11,MC	*Compiler module MLFIN stores the frame number 11 and frame type (multiple choice) into the frame code.*
H: HEADING	*MLHCS generates a one word operation code which will at execution time cause execution of standard routine HEADR to display room number, patient's hospital I.D. and name, current time and date.*
Q: 6,SELECT DESIRED OPTION	*MLQRP generates an operation code for text display onto the terminal. A portion of this operation code indicates that the display will be on line 6, with default character position and size. The number of characters to display and the line of text are stored into the frame code.*
R: SETUP *R: SAMPLE TEMPERATURE* *R: DISPLAY DATA*	*For R statements, which define the user's possible responses, MLQRP generates, in additional to the same information generated in Q statements) a response number to be displayed with the text. For these R statements the terminal display would be* *1 - SETUP* *2 - SAMPLE TEMPERATURE* *3 - DISPLAY DATA* *At this point, several operations are automatically generated by MLCMP. For this multiple choice frame:* *1. Exit to await user's entry of data.* *2. Conversion of this entry to an integer value assigned to keyword variable REL1.* *3. Range check to make sure that the value of REL1 corresponds to an R statement* *4. Automatic transfer indexing where the value of REL1 is used to determine which of the three frame transfer statements will be executed (similar to the computed GO TO concept).*
T: 3 *T: 10* *T: 20*	*For each of these statements MLCMP generates a constant for the frame number and a transfer operation.*
END	*MLCMP stores the frame size into the index (to be used later by the packing routine) and writes the frame code into the 11th record of the disk work file.*

Figure 11. Details of the Code Generation Process for a Sample Frame.