Design and Implementation of an Interactive Diagnostic PL/I System

Kenichi Harada* and Marvin V. Zelkowitz**

Abstract

Typically programs in a university environment are written, debugged, executed, and then discarded. Most of their time is spent in the debugging phase of program development. Therefore, in order to optimize system resources, a compiler is needed which is designed to execute programs quickly, and to produce detailed diagnostic messages.

PLUM is an interactive diagnostic PL/I system which has been developed at the University of Maryland for the Univac 1100 series computer. It compiles source programs directly into machine language and immediately executes them.

1. Introuction

In a university environment, especially in programming practice at computer science education, it can be observed that the programs are relatively small and the execution time is short. Furthermore, they often contain various trivial errors, and the number of programs submitted to run is very large. Then, a system is needed which is intended to run such programs as faster as possible rather than to produce highly optimized code. Powerful diagnostic facilities are also required. This type of compiler is useful as a diagnostic tool for checking out production programs. PLUM is an interactive diagnostic PL/I system which has been designed to run in time-sharing mode under the EXEC 8 operating system for Univac 1100 series computer.

2. Design Criteria

Before starting of the design and implementation of PLUM, the following assumptions were made on the environment of the system and the purspose of the use.

(1) The most of users are beginners in computer programming.

(2) The average program consists of 50 to 200 statements.

(3) Most computer time is spent in diagnostic phase of program development, i.e. correction of a source program, and recompilation and execution of the program. The execution time is too short.

(4) When a program is completed, it would be descarded.

(5) This system is mainly used in time-sharing mode through demand terminals.

From these assumptions, the characteristics of PLUM were decided as follows.

This paper first appeared in Japanese in Joho－Shori (Journal of the Information Processing Society of Japan), Vol. 16, No. 2 (1975), pp. 85～92.

* Keio Institute of Information Science, Keio University

** Department of Computer Science, University of Maryland

(1) The compiler generates absolute code directly into memory and then immediately executes the program(PLUM compiles about 10,000 average statements per minute on an 1108). No link edit step is necessary.

(2) The compiler generates detailed diagnostic messages.

(3) The compiler is designed around an error correcting philosophy that has been successfully employed on other compiler[1]. This means that given an error condition, the compiler will modify the source program in order to do more debugging than a student getting no output.

(4) The system has extensive data collection facifities for monitoring program behavior.

3. Language

PLUM accepts a significant subset of the PL/I language[2,3]. All arithmetic data types are accepted. That is, arithmetic data can be real or complex and can have the attributes: fixed binary, fixed decimal, float binary and float decimal. Up to 18 digits of precision are allowed for decimal arithmetic. String data can be either character or bit. Strings can have fixed lengths or can be varying length.

Using the primitive data types, aggregate data types can be declared. Arrays may be declared with arbitrary bounds and arbitrary dimensionality. Static and automatic storage classes can be defined on such data.

PLUM implements most PL/I statements. This includes the following statements:

Assignement	BEGIN	CALL	CLOSE	DECLARE
DO	END	ENTRY	FORMAT	GET
GOTO	IF	OPEN	PROCEDURE	PUT
RETURN	STOP			

and the special debugging statements which are described later:

SIGNAL	FLOW	NOFLOW

The statements which have not as yet been implemented include record oriented I/O, ON conditions and statements concerning pointer variables.

4. Diagnostic Facilities

Most of PLUM's usefulness stems from its diagnostic capability. Diagnostic information can be divided into two classes: automatic diagnostic aids and programmer controlled diagnostic aids.

4.1 Automatic Diagnostic Aids

This class of diagnostic information is always active during execution of a PLUM program. Some of these a user can disable in order to increase his execution speed slightly.

(1) Error correcting compiler: Whenever the compiler finds an error it will correct it before continuing. Fig. 1 is an example of such an output.

(2) Use of reserved words: Certain words like GET, PUT and DO are classified as reserved words and may be used by the user as a variable names. This enables to error correcting facility to determine a program's state should a syntax error occur.

(3) Subscript checking: All array references are checked for validity.

(4) Parameter matching:

All arguments to a procedure are compared with their corresponding formal parameters for validity.

(5) Execution profiles:

This option produces a histogram giving the frequence of execution of each statement in the program[4]. This facilitates debugging since it shows which statements may never have been executed, and which procedures are used frequently, and should be optimized.

```
     1    1 P  PR3C 3PTI3NS(MAIN) ;
*** IN    1  ERR3R SY 64 MISSING C3L3N
C3RRECTED: P : PR3C 3PTI3NS ( MAIN ) ;
     2    2     DCL (N,A(10))) FIXED BINARY (35,0) ;
*** IN    2  ERR3R SY 84 EXTRA )
C3RRECTED: DECLARE ( N , A ( 10 ) ) FIXED BINARY ( 35 ,
           0 ) ;
     3    3     FINDMAX : PR3C (A,N
*** IN    3  ERR3R SY 27 MISSING C3MMA
*** IN    3  ERR3R SY 30 MISSING RIGHT PARENTHESIS
*** IN    3  ERR3R SY 47 MISSING SEMIC3L3N
C3RRECTED: FINDMAX : PR3C ( A , N ) ;
     4    4               DCL (A(*),N,MAX) FIXED BINARY (35,0) ;
     5    5               MAX=0 ;
*** IN    5  ERR3R SY 25 INC3MPLETE EXPRESSI3N
*** IN    5  ERR3R SY 47 MISSING SEMIC3L3N
C3RRECTED: MAX = 0 ;
```

Fig. 1 An Example of Diagnostic Messages

(6) Conditional code generation: The sequence /*D *text* */ is normally interpreted as a PL/I comment. However, with a certain option specified, the /*D will be ignored, and *text* will be interpreted as part of the program. Thus the user could place debugging code as a permanent part of his program. Should an error occur, he could simply specify this option, rerun the program, and produce any desired diagnostic information.

4.2 Programmer Controlled Diagnostic Aids

Should an error occur during execution of a PLUM program, if the user is executing from a demand terminal an interactive debugging routine will be activated. The user has the following options(Fig. 2):

(1) Any variable name can be typed at the terminal, and its value will be printed.

(2) A PL/I procedure can be called from the terminal. For example, a user could have different error routines. Should an error occur, the user could call the appropriate routine to take corrective action. It is this feature, along with the ability of displaying values symbolically, that obviates the need for the ON condition.

(3) Breakpoints can be set. Thus the user can specify when the program is to return control to the terminal. Execution of the program may then be resumed at either the point of error, or to any other label in the program. In order to invoke the interactive routines from a program the special statement SIGNAL ERROR; can be used.

command	function
ALTER *v* = *c* ;	The variable *v* is assigned the constant value *c*.
BREAKPOINT *s*[D]	If statement number *s* is specified, then it becomes a breakpoint. Once executed, then control will return to the diagnostic routine. If D is also specified, then the breakpoint at statement *s* will be deleted.
CALL *p*	This causes the parameterless procedure *p* to be called. Upon completion, control will return to the diagnostic routine.
DISPLAY *v*	The variable *v*'s current value will be printed.
EXECUTE	The execution phase of the program will be redone without recompiling the program.
FLOW	The flow statement trace will be turned on.
GOTO *l*	Execution resumes at the label *l*.
HELP	A listing of all commands is printed.
INFORMATION *s*	Information about the command *s* will be printed.
LABEL	A list of the last 18 GOTO statements executed is printed.
NOFLOW	The flow trace is turned off.
RETURN	Execution resumes at the point of error.
STOP	Execution is terminated.
TIME	Elapsed memory time since the start of execution or since the last TIME command is printed.
UPDATE	The University of Maryland Text Editor is called and the source program can be edited. Upon return from the editor the program is recompiled without a source listing.
WALKBACK	The names of all currently active procedures will be printed. The location where each of these procedures was called will also be printed.

* Only the first character of each command is necessary.

Fig. 2 List of Interactive Debugging Command

(4) Trace routines can be turned on and off interactively. A statement trace can also be stated via the statement FLOW; from a PLUM program.

5. Compiler Organization

PLUM executes as a multisegmented program in the instruction bank of the 1108. A root segment is always present while each compilation and execution phase overlays the previous phase in memory. Passes 1 and 2 constitute syntax analysis and semantic analysis are contained in one phase; the cross reference listing is a second phase; code generation is a third and execution is a fourth phase[5].

5.1 General Data Flow through PLUM

In order to minimize I/O calls, all data is core resident. This means that the symbol table is always resident as well as the parsed program after syntax analysis and the object code after code generation. The organization of the workspace called PLUM common area during each phase is as shown in Fig. 3.

During pass 1, the source program is read and the symbol table is constructed for each identifier. The symbol table starts immediately after the hash table in memory. Pass 1 also constructs an internal form of the program. This internal form starts halfway down in the workspace. By the end of pass 1 both the symbol table and internal source program size are fixed. Pass 2 does not chang the size or location of either the symbol table or the source program table. Just prior to code generation the internal source program called Gamma code is moved to the end of the workspace. This frees up the largest block of memory possible between the end of the symbol table and the start of the Gamma code. As the Gamma code is scanned, the object code is placed in this freed block. Once code is generated for a statement, the space it took in the Gamma code is released. Finally, after code generation, the remainder of the workspace served as the run-time stack for the executing PL/I program.

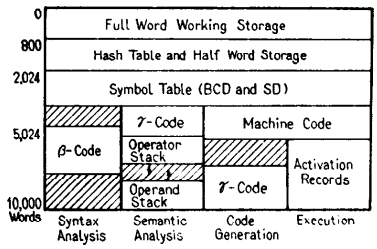

Fig. 3 PLUM Common Area

5.2 Syntax Analysis

Syntax analysis consits of lexical scanner, syntax analyzer, and expression analyzer. A lexical scanner reads in the source program and converts the program into a stream of tokens called Alpha code. Each Alpha code is contained within a lexical class, also associated with the token. The parsing algorithm uses one token lookahead where the parser is parsing token (n) while the scanner has already found token (n+1). This one token lookahead helps resolve some ambiguous decisions for the parser.

Syntax analyzer calls appropriate driver that sets up a tree for each statement type. Since the leaves of the tree are ususally expressions, an expression analyzer is called to parse expressions.

The expression analyzer is passed a code giving the context of the expression. The entire expression is converted to Beta code and attached to the tree. The expression analyzer then returns to the syntax analyzer for processing of the next phrase.

5.3 Semantic Analysis

Semantic analysis resolves name pointers in the Beta code into symbol table addresses. Semantic analysis looks for start expression indicators in the Beta code. When an expression is scanned, semantic analysis knows which blocks are currently active, and can therefore resolve name pointers to the appropriate symbol table entry according to the proper scope rules of PL/I.

Semantic analysis also checks expressions for validity. Associated with each expression in the expression type giving the context in which it is used. Using this token the context of the expression is checked, and if invalid, a default expression is used for it. The validity checking also includes checking arguments to function with the defined attributes of the parameters of the function.

Finally, since the code generator is stack oriented, semantic analysis will convert the expression to polish postfix for ease in generating code.

5.4 Code Generation

The Code generation techniques used in PLUM are similar to the techniques used in the PL/C compiler[6]

6. Summary

This brief description cannot describe all of the features available with PLUM and the structure of the system. It is designed as a total interactive system to be used in designing and implementing PL/I programs. Various options at compile and execution time enable the user to get valuable information about the behavior of his program, and the interactive facilities enable the user to easily, effectively and quickly switch among compiling, executing, debugging and editing phases of his program. These facilities should enable the user to implement programs in minimum amount of time with resulting lower overhead costs to the EXEC 8 operating system.

Reference

1. Conway, R. W., and Wilcox, T. R., Design and Implementation of a Diagnostic compiler for PL/I, CACM 16-3, 169-179(March 1973).
2. ECMA and ANSI, PL/I BASIS 1, BASIS/1-10, ECMA(June 1973).
3. Zelkowits, M. V., PLUM Reference Manual, University of Maryland, Computer Science Center, Computer Note CN-8(July 1974).
4. Ingalls, D., The Execution Time Profile as a Programming Tool, Design and Optimization of Compilers, Courant Computer Science Symposium 5, Prentice-Hall, 107-128 (1972)
5. Zelkowits, M. V., PLUM: The University of Maryland PL/I System, University of Maryland, Computer Science Center, TR-318(July 1974).
6. Wilcox, T. R., Generating Machine Code for High-Level Programming Language, p. 198, Cornell University(1971).