GENERAL ELECTRIC
COMPANY

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS . . . . . TELEPHONE 491-6300

CAMBRIDGE
INFORMATION
SYSTEMS
LABORATORY

March 25, 1969

TO: R. C. Daley
F. J. Corbato
J. M. Saltzer

FROM: R. A. Freiburghouse

The attached document is a draft of a paper which I plan to present at the Fall Joint Computer Conference. It is somewhat incomplete in that the accompanying diagrams are not included. Your comments and opinions are welcomed.

Robert A. Freiburghouse

Robert A. Freiburghouse

/mc
attachment

Very good technical description
- Needs more comprehensive overview and introduction
- Assumes reader has already thought a lot about complexities PL/I.

THE MULTICS PL/I COMPILER

1. INTRODUCTION

The Multics PL/I compiler is in many respects a "second generation" PL/I compiler. It was built at a time when the language was considerably more stable and well defined than it had been when the first compilers were built. It has benefited from the experience of the first compilers and avoids some of the difficulties which they encountered. The Multics compiler is the only full PL/I compiler written in PL/I and is the first PL/I compiler to produce highly efficient object code.

1.1 The Language

The Multics PL/I language is the language defined by IBM publication Y33-6003-0 dated March, 1968. At the time this paper was written most language features were implemented by the compiler but the run time library did not include support for input and output, as well as several lesser features. Since the Multics operating system provides a form of tasking which does not permit the data sharing required by PL/I tasking, PL/I tasking was not implemented. Interprocess communication (Multics tasking) may be performed through calls to operating system facilities.

1.2 The System Environment

The compiler and its object programs operate within the Multics operating system. The environment provided by this system includes a virtual two dimensional address space consisting of a large number

of segments. Each segment is a linear address space whose addresses range from 0 to 64K. The entire virtual store is supported by a paging mechanism which is invisible to the program. Each program operating in this environment consists of two segments: a text segment containing a pure re-entrant procedure, and a linkage segment containing out-references (links), definitions (entry names), and static storage local to the program. The text segment of each program is sharable by all other users on the system. Linking to a called program is normally done dynamically during program execution.

1.3 Implementation Techniques

The entire compiler ~~was~~ were written in a subset of PL/I called EPL. ~~This same subset was used to implement~~ And the Multics operating system. EPL is a large subset containing most of the complex features of PL/I. The Multics PL/I compiler can compile itself and the operating system.

Mention history of EPL/ System

The compiler was built and de-bugged by four experienced system programmers in 18 months. All program preparation was done on-line using the CTSS time-sharing system at MIT. Most de-bugging was done in a batch mode on the GE645, but final de-bugging was done on-line using Multics.

The extremely short development time of 18 months was made possible by these powerful tools. The same design programmed in a macro-assembly language using card input and batched runs would have required twice as much time, and the result would have been extremely unmanageable.

1.4 Design Objectives

The project's design decisions and choice of techniques were influenced by the following objectives:

1. A correct implementation of a reasonably complete PL/I language.
2. A compiler whose object code was as efficient as that produced by most Fortan compilers.
3. Object program compatibility with EPL object programs and other Multics languages.
4. An extensive compile time diagnostic facility.
5. A machine independent compiler capable of bootstrapping itself onto other hardware.

2. AN OVERVIEW OF THE COMPILER

A phase is a set of procedures which perform a major logical function of compilation such as syntactic analysis. A phase is not necessarily a memory load or a pass over some data base although it may, in some cases, be either or both of these things.

The notion of a phase is particularly useful when discussing the organization of the Multics PL/I compiler. The dynamic linking and paging facilities of the Multics environment have the effect of making available in virtual storage only those specific pages of those particular procedures which are referenced during an execution of the compiler. A phase of the Multics PL/I compiler is therefore

only a logical grouping of procedures which may call each other. The PL/I compiler is organized into five phases: Syntactic Translation, Declaration Processing, Semantic Translation, Optimization, and Code Generation.

2.1 The Internal Representation

The internal representation of the program being compiled serves as the interface between phases of the compiler. The internal representation is organized into a modified tree structure (the program tree) consisting of nodes which represent the component parts of the program such as blocks, groups, statements, operators, operands, and declarations. Each node may be logically connected to any number of other nodes by the use of pointers.

Each source program block is represented in the program tree by a block node which has two lists connected to it: a statement list and a declaration list. The elements of the declaration list are symbol table nodes representing declarations of identifiers within that block. The elements of the statement list are nodes representing the source statements of that block. Each statement node contains the root of a computation tree which represents the operations to be performed by that statement. This computation tree consists of operator nodes and operand nodes.

The operators of the internal representation are n-operand operators whose meaning closely parallels that of the PL/I source operators. The form of an operand is changed by certain phases, but operands

generally refer to a declaration of some variable or constant. Each operand also serves as the root of a computation tree which describes the computations necessary to locate the item at run time.

This internal representation is machine independent in that it does not reflect the instruction set, the addressing properties, or the register arrangement of the 645. The first four phases of the compiler are also machine independent since they deal only with this machine independent internal representation. Figure II-1 shows the internal representation of a simple program.

3. SYNTACTIC TRANSLATION

Syntactic analysis of PL/I programs is basically no more difficult than syntactic analysis of other languages such as Fortran. PL/I is a larger language containing more syntactic constructs, but it does not present any significantly new problems. The syntactic translator consists of two modules called the lexical analyzer and the parse.

3.1 Lexical Analysis

The lexical analyzer organizes the input text into groups of tokens which represent a statement. It also creates the source listing file and builds a token table which contains the source representation of all tokens in the source program. A token is an identifier, a constant, an operator or a delimiter. The lexical analyzer is called by the parse each time the parse wants a new statement.

The lexical analyzer is an approximation to a finite state machine. Since the lexical analyzer must produce output as well as recognize

tokens, action codes are attached to the state transitions of the finite state machine. These action codes result in the concatenation of individual characters from the output until a recognized token is formed.

The token table produced by the lexical analyzer contains a single entry for each unique token in the source program. Searching of the token table is done utilizing a hash coded scheme which provides quick access to the table. Each token table entry contains a pointer which may eventually point to a declaration of the token. For each statement, the lexical analyzer builds a vector of pointers to the tokens which were found in the statement. This vector serves as the input to the parse. Figure III-1 shows a simple example of lexical analysis.

3.2 The Parse

The parse consists of a set of possibly recursive procedures, each of which corresponds to a syntactic unit of the language. These procedures are organized to perform a top down analysis of the source program. As each component of the program is recognized, it is transformed into an appropriate internal representation. The completed internal representation is a program tree which reflects the relationships between all of the components of the original source program. Figure III-2 shows the parse of a simple program.

Syntactic contexts which yield declarative information are recognized by the parse, and this information is passed to a module called the

context recorder which constructs a data base containing this information. Declare statements are parsed into partial symbol table nodes which represent declarations.

3.3 The Problem of Backup

The top down method of syntactic analysis is used because of its simplicity and flexibility. The use of a simple statement recognition algorithm made it possible to eliminate all backup. The statement recognizer identifies the type of each statement before the parse of that statement is attempted. The algorithm used by this procedure first attempts to recognize assignment statements using a left to right scan which looks for token patterns which are roughly analogous to X = or X ( ) =. If a statement is not recognized as an assignment, its leading token is matched against a keyword list to determine the statement type. This algorithm is very efficient and is able to positively identify all legal statements without requiring keywords to be reserved.

4. DECLARATION PROCESSING

PL/I declaration processing is complicated by the great variety of data attributes and by the context sensitive manner in which they are derived. Two modules, the context processor and the declaration processor, process declarative information gathered by the parse.

4.1 The Context Processor

The context processor scans the data base containing contextually derived attributes produced during the parse by the context recorder. It either augments the partial symbol table created from declare

statements or creates new declarations having the same format as those derived from declare statements. This activity creates contextual and implicit declarations.

4.2 The Declaration Processor

The declaration processor develops sufficient information about the variables of the program so that they may be allocated storage, initialized and accessed by the program's operators. It is organized to perform three major functions: the preparation of accessing code, the computation of each variable's storage requirements, and the creation of initialization code.

All machine independent characteristics, such as the number of bits per word and the alignment requirements of data types, are contained in a table. The declaration processor is relatively machine independent. All computations or statements produced by the declaration processor have the same internal representation as source language expressions or statements. Later phases of the compiler will not distinguish between them.

4.2.1 The Use of Based References by the Declaration Processor

The concept of a based reference is useful to the understanding of PL/I data accessing and to the implementation of a number of language features. A based declaration of the form DCL A BASED is referenced by a based reference of the form P -> A, where P is a pointer to the storage occupied by a value whose description is given by the declaration of A. Multiple instances of data having the characteristics of

A can be referenced through the use of unique pointers, i.e., Q -> A, R -> A, etc.

The declaration processor implements a number of language features by transforming them into suitable based declarations. Automatic data whose size is variable is transformed into a based declaration. For example the declaration:

```
DCL  A(N)  AUTO;
```

becomes

```
DCL  A(N)  BASED(P);
```

where: P is a compiler produced pointer which is set upon entry to the declaring block.

Based declarations are also used to implement defined data and parameters. For example:

```
DCL  A(N)  DEFINED B;
```

becomes

```
DCL  A(N)  BASED (ADDR(B));
```

and

```
X:  PROC  (C);  DCL C;
```

becomes

```
X:  PROC  (C);  DCL C BASED(P);
```

where: P is a pointer which points to the argument corresponding to the parameter C.

4.2.2 Data Accessing

The address of an item of PL/I data consists of three basic parts: a pointer to some storage location, a word offset from that location

and a bit offset from the word offset. Either or both the offsets may be zero. The term "word" is understood to refer to the addressable unit of a computer's storage.

Example 1

```
DCL  A  AUTO;
```

The address of A consists of a pointer to the declaring block's automatic storage, a word offset within that automatic storage and a zero bit offset.

Example 2

```
DCL  1  S  BASED(P),
         2  A  BIT(5),
         2  B  BIT(N);
```

When referenced by P -> B, the address of B is a pointer P, a zero word offset, and a bit offset of 5. The word offset may include the distance from the origin of the item's storage class, as was the case with the first example, or it may be only the distance from the level-one containing structure, as it was in the last example. Subscripted array element references, A(I,5), or substring references, SUBSTR(X,I,J), may also be expressed as offsets. The term "level-one" refers to all variables which are not contained within structures.

4.2.2.1 Offset Expressions

The declaration processor constructs offset expressions which represent the distance between an element of a structure and the data origin of its level one containing structure.

Given a declaration of the form:

```
DCL 1 S,
    2 A BIT(N),
    2 B BIT(5),
    2 C FLOAT;
```

The offset of A is zero, the offset of B is N bits, and the offset of C is N+5 bits rounded upward to the nearest word boundary.

In general, the offset of the nth item in a structure is:

$$b_n(c_{n-1}(s_{n-1})+b_{n-1}(c_{n-2}(s_{n-2})+b_{n-2}(\ldots$$
$$(c_2(s_2)+b_2(c_1(s_1)))\ldots)))$$

where: b_k is a rounding function which expresses the boundary requirement of the kth item.

s_k is the size of the kth item.

c_k is the conversion factor necessary to convert the s_k to some common units such as bits.

The declaration processor suppresses the creation of unnecessary conversion functions (c_k) and boundary functions (b_k) by keeping track of the current units and boundary as it builds the expression. As a result, the offset expressions of the previous example do not contain conversion functions and boundary functions for A and B.

During the construction of the offset expression, the declaration processor separates the constant and variable terms so that the addition of constant terms is done by the compiler rather than by

accessing code in the object program. The following example demonstrates the improvement gained by this technique.

```
DCL 1 S,
    2 A BIT(5),
    2 B BIT(K),
    2 C BIT(6),
    2 D BIT(10);
```

The offset of D is K+11 instead of 5+K+6.

The word offset and the bit offset are developed separately. Within each offset, the constant and variable parts are separated. These separations result in the minimization of additions and unit conversions. If the declaration contains only constant sizes, the resulting offsets are constant. If the declaration contains expressions, then the offsets are expressions containing the minimum number of terms and conversion factors.

The development of size and offset expressions at compile time eliminates the need for run time data descriptors or dope vectors. In general, the offset expressions constructed by the declaration processor remain unchanged until code generation. Two cases are exceptions to this rule: subscripted array references, A(I,J), and sub-string references, SUBSTR(X,I,J). In both cases, the offsets are made unique to each reference by the actions of the semantic translator.

4.2.3 Allocation

The declaration processor does not allocate storage for most classes of data, but it does determine the amount of storage needed by each ~~level one~~ variable. Level ONE variables are allocated within some segment of storage by the code generator. Storage allocation is delayed because, during semantic translation and optimization, additional declarations of constants and compiler created variables are made.

4.2.4 Initialization

The declaration processor creates statements in the prologue of the declaring block which will initialize automatic data. It generates assignment statements, or patterns of do loops, if statements and assignment statements to accomplish the required initialization.

The expansion of the initial attribute for based and controlled data is identical to that for automatic data except that the required statements are inserted into the program at the point of allocation rather than in the prologue.

Since array bounds and string sizes of static data are required by the language to be constant, and since all values of the initial attribute of static data must be constant, the compiler is able to initialize the static data at compile time. The initialization is done by the code generator at the time it allocates the static data.

5. SEMANTIC TRANSLATION

The semantic translator transforms the internal representation so that it reflects the attributes (semantics) of the declared variables without reflecting the properties of the object machine. It makes a single scan over the internal representation of the program. A compiler, ~~which~~ if it had no equivalent of the optimizer phase and which did not separate the machine dependencies into a separate phase, could conceivably produce object code during this scan.

5.1 Organization of the Semantic Translator

The semantic translator consists of a set of recursive procedures which walk through the program tree. The actions taken by these procedures are described by the general terms: operand processing and operator transformation. Operand processing determines the attributes, size and offsets of each operator's operands. Operator transformation includes the creation of an explicit representation of each operator's result and the generation of conversion operators for those operands which require conversion.

order is inverted in next two sections

5.2 Operator Transformation

The meaning of an operator is defined by the attributes of its operands. This meaning determines which conversions must be performed on its operands, and it determines the attributes of the operator's result.

An operator's result is represented in the program tree by a temporary node. Temporary nodes are a further qualification of the original operator. For example, an add operator whose result is fixed-point

is a distinct operation from an add operator whose output is floating-point. There is no storage associated with temporaries - they will be allocated either core or register storage by the code generator. A temporary's size is a function of the operator's meaning and the sizes of the operator's operands. A temporary, representing the intermediate result of a string operation, requires an expression to represent its length if any of the string operator's operands have variable lengths.

5.2 Operand Processing

Operands consist of sub-expressions, references to variables, constants, and references to procedure names or built-in functions. Sub-expression operands are processed by recursive use of operator transformation and operand processing. Operand processing converts constants to a binary format which depends on the context in which the constant was used. References to variables or procedure names are associated with their appropriate declaration by the search function. After the search function has found the appropriate declaration, the reference may be further processed by the subscriptor or function processor.

5.2.1 The Search Function

During the parse, it is not possible for references to source program variables to know the declared attributes of the variable because the PL/I language allows declarations to follow their use. Therefore, references to source program variables are parsed into a form which contains a pointer to a token table entry rather than to a declaration of the variable. Figure III-2 shows the output of the parse. The search function finds the proper declaration for each reference to a

source program variable. The algorithm employed by the search depends heavily on the structure of the token table and the symbol table. ~~After declaration processing, the token table and the symbol table.~~ After declaration processing, the token table entry representing an identifier contains a list of all the declarations of that identifier. See Figure V-1.

The search function first tries to find a declaration belonging to the block in which the reference occurred. If it fails to find one, it looks for a declaration in the next containing block. This process is repeated until a declaration is found. Since the number of declarations on the list is usually one, the search is quite fast. In its attempt to find the appropriate declaration, the search function obeys the language rules regarding structure qualification. It also collects any subscripts used in the reference and places them into a subscript list. Depending on the attributes of the referenced item, the subscript list serves as input to the function processor or subscriptor.

The declaration processor creates offset expressions and size expressions for all variables. These expressions, known as accessing expressions, are rooted in a reference node which is attached to a symbol table node. The reference node contains all information necessary to access the data at run time. The search function translates a source reference into a pointer to this reference. See Figure V-2.

5.2.2 Subscripting

Since each subscripted reference is unique, its offset expression is unique. To reflect this in the internal representation, the subscriptor creates a unique reference node for each subscripted reference. The following discussion shows the relationship between the declared array bounds, the element size, the array offset and subscripts.

Let us consider the case of an array declared:

$a(l_1:u_1, l_2:u_2, \ldots, l_n:u_n)$

Its element size is s and its offset is b.

The multipliers for the array are defined as:

$$m_n = s$$
$$m_{n-1} = (u_n - l_n + 1)s$$
$$m_{n-2} = (u_{n-1} - l_{n-1} + 1)m_{n-1}$$
$$\vdots$$
$$m_1 = (u_2 - l_2 + 1)m_2$$

The offset of a reference $a(i_1, i_2, \ldots, i_n)$ is computed as:

$$v + \sum_{j=1}^{n} i_j m_j$$

where: v is the virtual origin. The virtual origin is the offset obtained by setting the subscripts equal to 0. It serves as a convenient base from which to compute the offset of any array element.

During the construction of all expressions, the constant terms are separated from the variable terms and all constant operations are performed by the compiler. Since the virtual origin and the multipliers

are common to all references, they are constructed by the declaration processor and are repeatedly used by the subscriptor.

Arrays of PL/I structures which contain arrays may result in a set of multipliers whose units differ. The declaration:

```
DCL 1 S(10),
    2 A PTR,
    2 B(10) BIT(2);
```

yields two multipliers of different units. The first multiplier is the size of an element of S in words, while the second multiplier is the size of an element of B in bits. Array parameters which may correspond to an array cross section argument must receive their multipliers from the argument, since the arrangement of the cross section elements in storage is not known to the called program.

5.2.3 The Function Processor

An operand which is a reference to a procedure is expanded by the function processor into a ~~cal~~ CALL operator and possible conversion operators. Built-in function references result in new operators or are translated into expressions consisting of operators and operands.

5.2.3.1 Generic Procedure References

A generic entry name represents a family of procedures whose members require different types of arguments.

```
DCL ALPHA GENERIC (BETA  ENTRY(FIXED),
                   GAMMA ENTRY(FLOAT));
```

A reference to ALPHA (X) will result in a call to BETA or GAMMA depending on the attributes of X.

the declaration processor chains together all members of a generic family, and the function processor selects the appropriate member of the family by matching the arguments used in the reference with the declared argument requirements of each member. When the appropriate member is found, the original reference is rewritten as a reference to the selected member.

"substituted" is more graphic

5.2.3.2 Argument Processing

The function processor matches arguments to user-declared procedures against the argument types required for the procedure. It inserts conversion operators into the program tree where appropriate, and it issues ~~source diagnostics~~ diagnostics when it detects illegal cases.

The return value of a function is processed as if it were the n+1th argument to the procedure, eliminating the distinction between subroutines and functions.

The function processor determines which arguments may possibly correspond to a parameter whose size or array bounds are not specified in the called procedure. In this case, the argument list is augmented to include the missing size information. A more detailed description of this issue is given later in the discussion of object code strategies.

5.2.3.3 The Built-in Function Processor

The built-in function processor is basically a table driven device. The driving table describes the number and kind of arguments required for each function and is used to force the necessary conversions and

diagnostics for each argument. Most functions require processing which is unique to that function, but the table driven device minimizes the amount of this processing.

The SUBSTR built-in function is of particular importance since it is a basic PL/I string operator. It is a three argument function which allows a reference to be made to a portion of a string variable, i.e., SUBSTR (X,I,J) is a reference to the ith through i+j-1th character (or bit) in the string X.

This function is similar to an array element reference in the sense that they both determine the offsets of the reference. The processing of the SUBSTR function involves adjusting the offset and length expressions contained in the reference node of X. As is the case in all compiler operations on the offset expressions, the constant and variable terms are separated to minimize the object code necessary to acces the data.

6. THE OPTIMIZER

The compiler is designed to produce efficient object code without the aid of an optimizing phase. Normal execution of the compiler will by-pass the optimizer, but if extensively optimized object code is desired, the user may set a compiler command option which will execute the optimizer. The optimizer consists of a set of procedures which perform two major optimizations: common sub-expression removal and removal of computations from loops.

The data bases necessary for these optimizations are constructed by the parse and the semantic translator. These data bases consist of a cross-reference structure of statement labels and a tree structure representing the DO groups of each block. Both optimizations are done on a block basis using these two data bases.

The optimizer is the only phase of the compiler which is not implemented at the time this paper was written. The compiler does construct all of the data bases required by the optimizer and does determine the abnormality of all variables. Excessive compile times may require that general sub-expression optimization be restricted to accessing computations. These computations are generally not under the control of the source programmer and are often redundant, particularly when they result from PL/I based data references.

Experience may show that specialized optimizations are more profitable than the general optimizations described here. If this proves to be the case, the design of the optimizer will be modified. The optimization of PL/I programs is not significantly more difficult than the optimization of Fortran programs. The concept of abnormality applies to both languages and is discussed in more detail in the next section. The condition prefix of PL/I constrains the circumstances in which computations may be removed from loops. This prefix specifies a state in which a computation must be performed (i.e., divide by zero inhibited etc.). A computation cannot be moved to a region of the program whose state differs from the state of the original region.

6.1 Abnormality

Abnormal variables and values which depend on abnormal variables cannot become candidates for optimization. A variable is abnormal to some block if its value can be altered during the execution of the block without an explicit indication of that fact present in that block. The number of situations which cause abnormality depend on two things: the properties of the language and the range over which the optimizer expects the values to hold. If values are not expected to hold across a call or function reference, the number of situations which constitute abnormality in the PL/I language is greatly reduced. Table 1 shows the conditions which cause abnormality in PL/I without tasking. The compiler does not depend on the programmer to correctly use the source language attributes normal and abnormal. It determines this attribute by examining the usage of the variables.

7. THE CODE GENERATOR

The code generator is the machine dependent portion of the compiler. It performs two major functions: it allocates data into Multics segments and it generates 645 machine instructions from the internal representation.

7.1 Storage Allocation

A module of the code generator called the storage allocator scans the symbol table allocating stack storage for constant size automatic data, and linkage segment storage for internal static data. For each external name the storage allocator creates a link (an out-reference) or a definition (an entry point) in the linkage segment. All internal static data is initialized as its storage is allocated.

Due to the dynamic linking and loading characteristics of the Multics environment, the allocation and initialization of external static storage is rather unusual. The compiler creates a special type of link which causes the linker module of the operating system to create and initialize the external data upon first reference. Therefore, if two programs contain references to the same item of external data, the first one to reference that data will allocate and initialize it.

7.2 Code Generation

The code generator scans the internal representation transforming it into 645 machine instructions which it outputs into the text segment. During this scan, the code generator allocates storage for temporaries, and maintains a history of the contents of index registers to prevent excessive loading and storing of index values.

Code generation consists of three distinct activities: address computation, operator selection and macro expansion. Address computation is the process of transforming the offset expressions of a reference node into a machine address or an instruction sequence which leads to a machine address. Operator selection is the translation of operators into n-operand macros which reflect the properties of the 645 machine.

A one-to-one relationship often exists between the macros and 645 instructions but many operations (load long string, etc.) have no machine counterpart. All macros are expanded in actual 645 code by the macro expander which uses a code pattern table (macro skeletons) to select the specific instruction sequences for each macro.

8. OBJECT CODE STRATEGIES

8.1 The Object Code Design

The design of the object code is a compromise between the speed obtainable by straight in-line code and the necessity to minimize the number of page faults caused by large object programs.

The length of the object program is minimized by the extensive use of out-of-line code sequences. These out-of-line code sequences represent invariant code which is common to all PL/I object programs, they are not in any respect interpretive. The object code produced for each operator is very highly taylored to the specific attributes of that operator.

All out-of-line sequences are contained in a single "operator" segment which is shared by all users. The in-line code reaches on out-of-line sequence through a transfer instruction and return is effected by another transfer. The time overhead associated with the transfers is more than redeemed by the reduction in the number of page faults caused by shorter object programs. System performance is improved by insuring that the pages of the operator segment are always retained in storage. Figure VII-1 shows the relationship of the operator segment to several object programs.

8.2 The Stack

PL/I object programs utilize a stack segment for the allocation of all automatic data, temporaries, and data associated with on-conditions. The stack is extended (pushed) upon entry to block and is freed (popped)

upon return from a block. Prior to the execution of each statement it is extended to create sufficient space for any variable length string temporaries used in that statement. Constant size temporaries are allocated at compile time and do not cause the stack to be extended for each statement.

8.3 Prologue and Epilogue

The term prologue describes the computations which are performed after block entry and prior to the execution of the first source statement. These actions include the establishment of the condition prefix, the computation of the size of variable size automatic data, extension of the stack to allocate automatic data, and the initialization of automatic data. Epilogues are not needed because all actions which must be undone upon exit from the block are accomplished by popping the stack. The stack is popped for each return or non-local goto statement.

8.4 Accessing of Data

Multics PL/I object code addresses all data, including members of variable sized structures and arrays, directly through the use of in-line code. If the address of the data is constant, it is computed at compile time. If it is a mixture of constant and variable terms, the constant terms are combined at compile time. Descriptors are never used to address or allocate data.

8.5 String Operations

All string operations are done by in-line code or by "transfer" type subroutinized code. No descriptors or calls are produced for

string operations. The SUBSTR built-in function is implemented as a part of the normal addressing code and is therefore as efficient as a subscripted array reference.

8.6 String Temporaries

A string temporary or dummy is designed in such a way that it appears to be both a varying and non-varying string. This means that the programmer does not need to be concerned with whether a string expression is varying or non-varying when he uses such an expression as an argument.

8.7 Varying Strings

The PL/I implementation of varying strings uses a data format which consists of an integer followed by a non-varying string whose length is the declared maximum of the varying string. The integer is used to hold the current size of the string in bits or characters. Using this data format, operations on varying strings are just as efficient as operations on non-varying strings.

8.8 On-Conditions

The design of the condition machinery minimizes the overhead associated with enabling and reverting on-units and transfers most of the cost to the signal statement. All data associated with on-conditions, including the condition prefix, is allocated in the stack. The normal popping of the stack reverts all enabled on-units and restores the proper condition prefix. Stack storage associated with each block is threaded backward to the previous block. The signal statement uses this thread to search back through the stack looking for the first enabled unit for

the condition being signalled. Figure VII-2 shows the organization of enabled on-units in the stack.

8.9 Argument Passing

The PL/I language permits parameters to be declared with unknown array bounds or string lengths. In these cases, the missing size information is assumbed to be supplied by the argument which corresponds to the parameter. This missing size information is not explicitly supplied by the programmer as is the case in Fortran, rather it must be supplied by the compiler as shown in the following example.

```
SUB:  PROC(A);              MAIN:  PROC;
        .                           .
        .                           .
DCL   A(*)  FLOAT;          DCL   SUB ENTRY
        .                   DCL   B(10)  FLOAT;
        .                     CALL  SUB(B);
                                    .
```

The declaration of a called procedure may or may not include a description of the arguments required by that procedure. If such a description is not supplied, then the calling program must assume that argument descriptors are needed, and must include them in all calls to the procedure. If the complete argument description is supplied to the calling program, the compiler can determine if the descriptors are needed.

Since descriptors are often created by the calling procedure but not used by the called procedure, it is desirable to separate them from the argument information which is always used by the called procedure.

Communication between procedures written in PL/I and other languages is facilitated if the other languages do not need to concern themselves with PL/I argument descriptors. The Multics PL/I implementation of the argument list is shown in Figure VII-3. Note that the argument pointers point directly to the data (facilitating communication between languages) and that the descriptors are optional. Since descriptors contain no addressing information, they are quite often constant and can be prepared at compile time.

9. CONCLUSIONS

Several insights and opinions were formed by the Multics PL/I project which are of general interest.

1. It is feasible, but difficult, to produce efficient object code for the PL/I language as it is presently defined.

2. The difficulty of building a compiler for the current language has been seriously underestimated by most implementors. Unless the language is markedly improved and simplified, this problem will continue to restrict the availability and acceptance of the language and will lead to the implementation of incompatible dialects.

3. Simplification of the existing language would make it more suitable to users and implementors. The language can be simplified and still retain its "universal" character and capabilities.

There need support in the form of examples and usages.

4. The language contains a number of concepts which make it an extremely powerful system programming language. An efficiently implemented language containing these concepts is markedly superior to macro-assembly languages for system programming.

10. ACKNOWLEDGEMENTS

The author wishes to express recognition to members of the Multics PL/I Project for their contributions to the design and implementation of the compiler. J. D. Mills was responsible for the design and implementation of the syntactic analyzer and the PL/I command, B. L. Wolman designed and built the code generator and operator segment, and G. D. Chang implemented the semantic translator. Valuable advice and ideas were provided by A. H. Kvilekval of General Electric. The earlier work of D. M. McIlroy and R. Morris of Bell Telephone Laboratories provided a useful guide and foundation for our efforts.