Design And Implementation Of A Diagnostic Compiler For PL/I

Richard W. Conway
Thomas R. Wilcox

Research Report 71-107

September 1971

Abstract:

PL/C is a compiler for a dialect for PL/I. The design objective was to provide a maximum degree of diagnostic assistance in a batch processing environment. For the most part this assistance is implicit and is provided automatically by the compiler. The most remarkable characteristic of PL/C is its perseverance-- it completes translation of every program submitted and continues execution until a user-established error limit is reached. This requires that the compiler repair errors encountered during both translation and execution, and the design of PL/C is dominated by this consideration.

PL/C also introduces several explicit user-controlled facilities for program testing. Some are conventional, providing a convenient high-level trace and dump capability. An experimental version of PL/C also permits the user to controllably reverse the direction of program execution and to write routines that are asynchronously invoked when an arbitrary condition becomes true. To accomodate these extensions to PL/I without abandoning compatibility with the IBM compiler, PL/C permits 'pseudo-comments' -- constructions whose contents can optionally be considered either source text or comment.

In spite of the diagnostic effort PL/C is a fast and efficient processor. It effectively demonstrates that compilers can provide better diagnostic assistance than is customarily offered, even when a sophisticated source language is employed, and that this assistance need not be prohibitively costly.

Key Words And Phrases: Compilers, Debugging, PL/I

CR Categories: 1.5, 4.12, 4.42

Design And Implementation Of A Diagnostic Compiler For PL/I

Richard W. Conway and Thomas R. Wilcox*
Department of Computer Science
Cornell University

PL/C is a development effort intended to explore the limits of diagnostic assistance by a compiler in a conventional batch-processing computing environment. At present this is the mode employed for most introductory instruction and it seems unlikely that it will soon be entirely replaced by terminal-oriented systems. It is generally believed that superior diagnostic control can be exercised by an interpretive system--although at the cost of greatly increased execution time. It is also assumed that the diagnostic dialog possible with an interactive system is more effective than the traditional debugging cycle--although probably at greater cost (Reference 11). It would seem appropriate to evaluate the virtues of both the interpretive and interactive diagnostic systems by comparison with a specialized diagnostic compiler, operating under a contemporary fast-turnaround supervisor (6), rather than by comparison with a general purpose production compiler in an environment with long and unpredictable turnaround.

In most respects PL/C differs only in degree from the diagnostic services of other compilers, but it is quite unique in its perseverance. Unless the user explicitly inhibits it, PL/C will complete the translation phase and begin execution for <u>any</u> source program submitted-- no matter how erroneous it may be. The objective of this strategy is to maximize the amount of useful information that can be obtained from each program submission, and hence reduce the number of submissions required to achieve successful execution. The PL/C hypothesis is that by prolonging, within reason, the life of an incorrect program some potentially useful diagnostic information can be obtained from the execution output. Some of this output will be automatically supplied by the compiler; some might be generated by statements inserted by the user to capitalize upon the compiler's perseverance. The duration and effectiveness of this prolongation of the life of a faulty program is very largely dependent upon the action taken by the compiler when source errors are discovered. One could, of course, simply compile a 'stop' into the object program at the point of each source error and continue execution only until the first of these is encountered. It hardly seems that this is the most imaginative assistance that a compiler could provide. PL/C attempts to improve upon this by effecting a plausible repair of each source error where possible, or by replacing the erroneous statement with a 'message-producing null' statement where repair is beyond the ingenuity of the compiler. With the exception of a few immediately fatal conditions execution is terminated only by reaching a user-specified cumulative error count, rather than by encounter with the first error.

* Current address: University of Illinois

In addition to the implicit diagnostic facilities that are invoked whenever the user violates a syntactic or semantic restriction of the language, PL/C provides rather unusual explicit facilities with which the user can enhance the diagnostic information obtained from early submissions. In every case these facilities really just provide a more convenient means of eliciting information that could be, but rarely is, obtained with standard source language facilities.

None of these facilities (except possibly for the RETRACE statement) is entirely original with PL/C. The same approach was used in a compiler for an ALGOL-dialect called CORC in 1961 (1) and further developed in a compiler for a PL/I-dialect called CUPL in 1966 (2). However, both of these predecessors were severely restricted subsets intended strictly for introductory instruction and one might wonder whether the utility of this diagnostic approach was limited to such restricted languages. PL/C represents the first time that this extreme diagnostic approach has been applied to a rich and sophisticated source language.

PL/C is similar to several other specialized diagnostic compilers in that it achieves relatively high compilation speeds by avoiding the use of auxiliary storage, by compiling absolute executable machine language (to avoid an assembly stage, and linking and loading) and by permitting independent programs to be considered a single job (to avoid operating system inter-job overhead). It produces object code that is reasonably efficient but not optimized, and that contains calls to various run-time monitoring routines (in particular, for subscriptrange testing.) Again, all of these properties were present in CORC ten years ago and have appeared in many compilers since, such as PUFFT, WATFOR, CUPL, ALGOL-W and WATFIV. PL/C is notable only in that PL/I is the most sophisticated source language to which these compilation strategies have been applied. Actually the compilation speed realized by PL/C (10-20,000 source statements per minute on an IBM 360/65) was a pleasant surprise and is apparently just a consequence of employing a relatively small group of very experienced programmers rather than any innovation in compilation strategy. Speed was an important criterion in design, but was clearly subordinate to diagnostic performance--although we rather suspect that in fact high compilation speed is the most useful single diagnostic facility that PL/C provides.

The PL/C compiler was designed to be open-ended in two different senses. It was designed so as not to preclude the implementation of the entire PL/I language, although manpower limitations dictated a subset for initial implementation. (The current version (3) lacks only multi-tasking, compile-time facilities, list processing and direct access auxiliary files in comparison to IBM PL/I-F.) The subset was deliberately richer than would be required for most introductory instruction simply to demonstrate that the diagnostic techniques are generally applicable. Secondly, the implementation was structured so that the diagnostic algorithms were accessible and replaceable. It is hoped that eventually more systematic and powerful repair algorithms will be developed (5).

The Organization Of The Compiler

The modular structure of the PL/C compiler was based upon that of the earlier CUPL compiler which had proven remarkably successful with respect to permitting parallel development of major sections, ease of maintenance, and extensive repair of source errors. This structure is shown in Figure 1. PL/C is basically a three-pass compiler although only the first of these passes has access to actual source program symbols. The communication interface between the first and second, and second and third passes is provided by core-resident compacted and encoded representations of the source program, called 'beta-code' and 'gamma-code' respectively.

The key to the error-repair and compilation efficiency of PL/C is the design of this internal program representation. This was based on a similar representation developed in the CUPL compiler. The following characteristics were sought in the design:

1. It had to facilitate simple and efficient interpretation in the following phase.
2. It had to be decompilable into source symbols to display to the user the effect of PL/C error repair.
3. It had to be relatively compact so that the compiler could process large programs without requiring the use of auxiliary storage.
4. It had to be self-relocating to permit efficient core management.

Beta and gamma code both consist of contiguous sequences of 16 bit symbols representing:

1. Source tokens (keywords, identifiers, operators, punctuation, etc.)
2. Program structure descriptors (statement numbers, pointers to positions in the internal code.)

Beta and gamma code are very easily scanned and interpreted since:

1. Each statement is syntactically well-formed.
2. The presence or absence of various options in the statement is described by bit patterns in a control symbol.
3. The arbitrary ordering of phrases in PL/I is negated by phrase-pointers that permit semantic analysis and code generation to approach each statement as if it were in a certain canonical form.
4. All expressions are specially identified and delimited and complete context information is encoded in a single control symbol at the beginning of each expression.

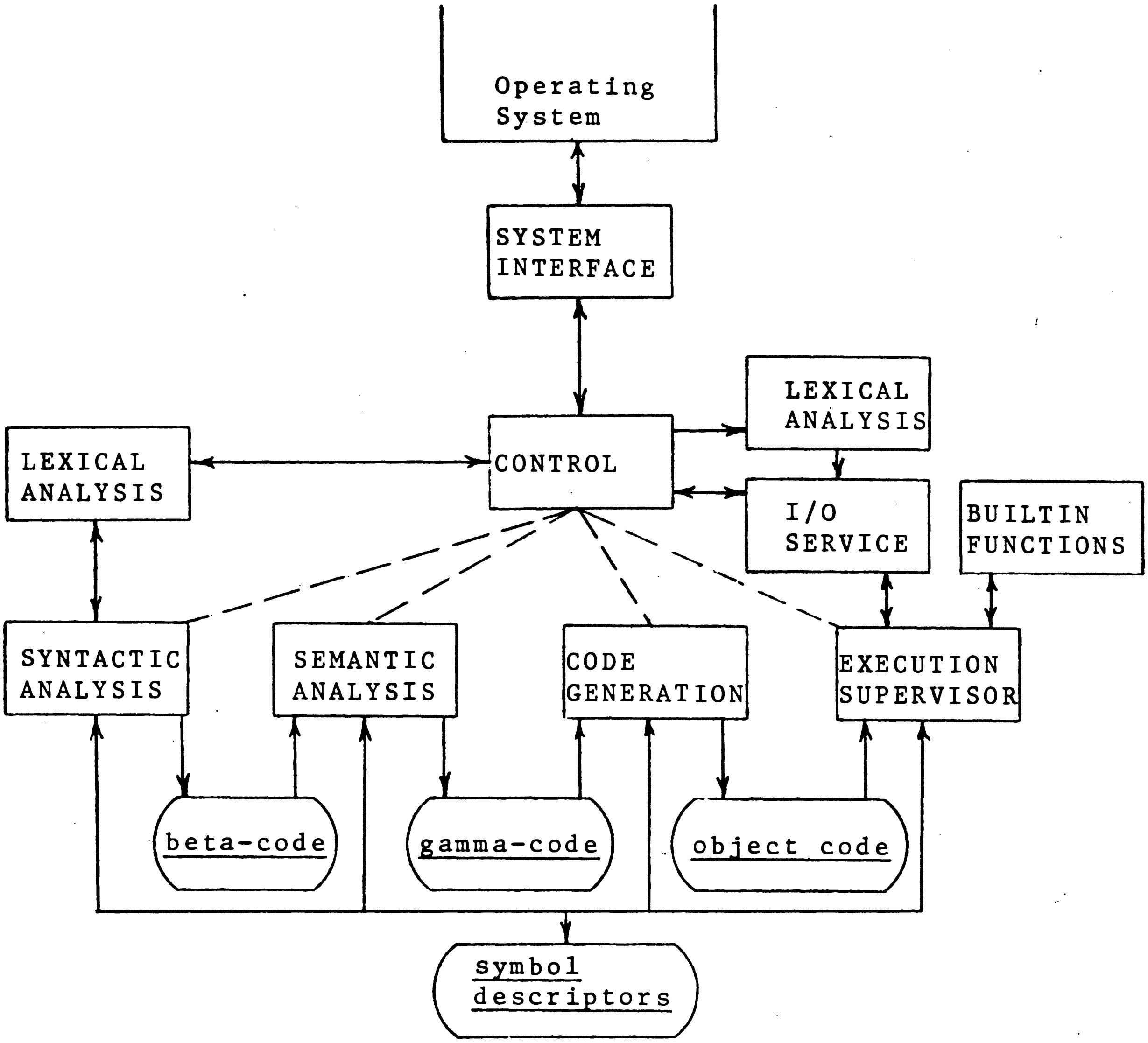

Figure 1. Organization Of The PL/C Compiler

For example, the beta-code representation of an IF statement is the following:

symbol	interpretation
1	start-of-statement flag
2	source statement number
3...	Prefixes, if any present
4	IF statement identifier
5	pointer to beginning of ELSE phrase in beta-code
6	pointer to beginning of 'continue' phrase
7	start-of-expression flag and context information (demanding a scalar, string-valued expression)
8...	Symbols in expression
9	end-of-expression flag

The beta-code for a GET statement is the following:

symbol	interpretation
1	start-of-statement flag
2	source statement number
3...	Prefixes, if any present
4	GET statement identifier and option bits
5	pointer to FILE|STRING phrase in beta-code
6	pointer to data specification phrase in beta-code
7	pointer to carraige control phrase in beta-code
8...	Phrases

There is a general 'principle of efficiency' in translators that directs one to perform tasks 'as early as possible', or equivalently -- 'only as late as necessary'. Within the limits prescribed by the PL/I language, and with the few exceptions noted in the following sections PL/C observes this principle and is a compiler (rather than an interpreter.) A similar principle applies to error detection--detect errors as early in the translation process as possible. Doing so permits each successive phase to make progressively stronger assumptions about the quality of its input and hence eliminate much redundant testing.

Given the structure of Figure 1 the general guidelines laid down for implementation were the following:

1. Whenever possible, on encountering an error effect a repair and continue.
2. All communication with the user is to be in source language terms.
3. All code is to be pure-reentrant procedure.
4. All system functions are to be routed through a control module to a single system-interface module.
5. The coding criteria, in order of importance, are:
    a. Compatibility with IBM PL/I-F for a <u>correct program</u>.
    b. Diagnostic assistance and control.
    c. Compilation speed.
    d. Execution speed.
    e. Memory space.
6. Frequency-sensitive coding is to be used. That is, whenever a choice has to be made assign the faster path to the more frequently used alternative. While this should be obvious it does not always seem to be observed. Its use is particularly important in a highly diagnostic compiler. While programs are very likely to contain some error, the probability of occurrence for each particular error is very low. Hence for reasonable efficiency it is crucial to favor the processing of a correct program at each point, even if this requires substantial effort to recover and repair when the assumption of correctness is found to be false. Several good examples of this strategy are described in the following sections.

The coding was done in assembly language. Very extensive use was made of the macro facility of the assembler, but this cannot really be considered an automatic 'translator writing system'. Hence PL/C was programmed in the old-fashioned way, but at least at the present state of the art, this is probably necessary to obtain very high compilation speeds.

As a consequence of observing the principle of early error detection the diagnostic character of the phases differs. Error detection and repair completely dominates the syntactic analysis phase. The semantic phase is perhaps half concerned with errors; it also performs the conventional tasks of parsing and resolution. The code generation and execution phases have certain special duties with respect to user-communication and error repair, but at least in comparison to the earlier phases their tasks are more conventional.

Syntactic Analysis

The first pass, performed by the module labeled 'syntactic analysis' in Figure 1, is responsible for:

1. Scanning and tokenizing the input stream, and printing a copy of the source program.
2. Constructing the beta-code representation of the source program-- syntactically repaired as may be required.
3. Constructing a symbol descriptor (SD) for each identifier name encountered and an SD for each different identifier. These descriptors are chained together in several different ways to reflect the block structure of the program and the required storage management of the variables.

The principal burden of error detection falls upon syntactic analysis. The general virtue of early detection is reinforced by the fact that only during this phase can PL/C position error messages where they are most convenient and intelligible to the user--immediately after the offending statement.

The general organization of syntactic analysis is shown in Figure 2. Each source statement involves passage through the 'statement start' routine, possibly one or more of the 'prefix drivers', exactly one of the 'statement drivers' and finally the 'statement end' routine. The drivers are logically parallel so that as the PL/C subset has expanded it has increased the size of the compiler but not significantly affected the speed of translation. All of the modules of Figure 2 draw upon the lexical analyzer (not shown) and various service routines that construct and chain the SDs. There is also a common expression analyzer that is used by all of the drivers. The efficiency of lexical analysis and the communication between syntactic and lexical analysis is crucial to the performance of a compiler. Lexical analysis in PL/C makes heavy use of the powerful 360 translate-and-test instruction and uses a double random algorithm for hash-searching of the symbol table. Communication between the syntactic and lexical analyzers is facilitated by a careful allocaton of base and working registers so that the main part of both routines is always covered and lexical analysis does not even have to save and restore registers when it is called. Lexical analysis passes a token that is carefully encoded to facilitate the 'transition table' organization of the syntactic analyzer.

The blocks in Figure 2 improperly suggest a symmetry of module size. Certain of the drivers are trivial--the STOP statement driver, for example. On the other hand, the driver for the DECLARE statement represents half of the code of the entire syntactic analyzer (and probably more than half of the programming effort.) It is just barely possible to analyze and process a PL/I DECLARE statement in a single pass. The combination of structures, factoring and arbitrary ordering of attributes results in a complicated maze of stacks, chains and recursive procedures.

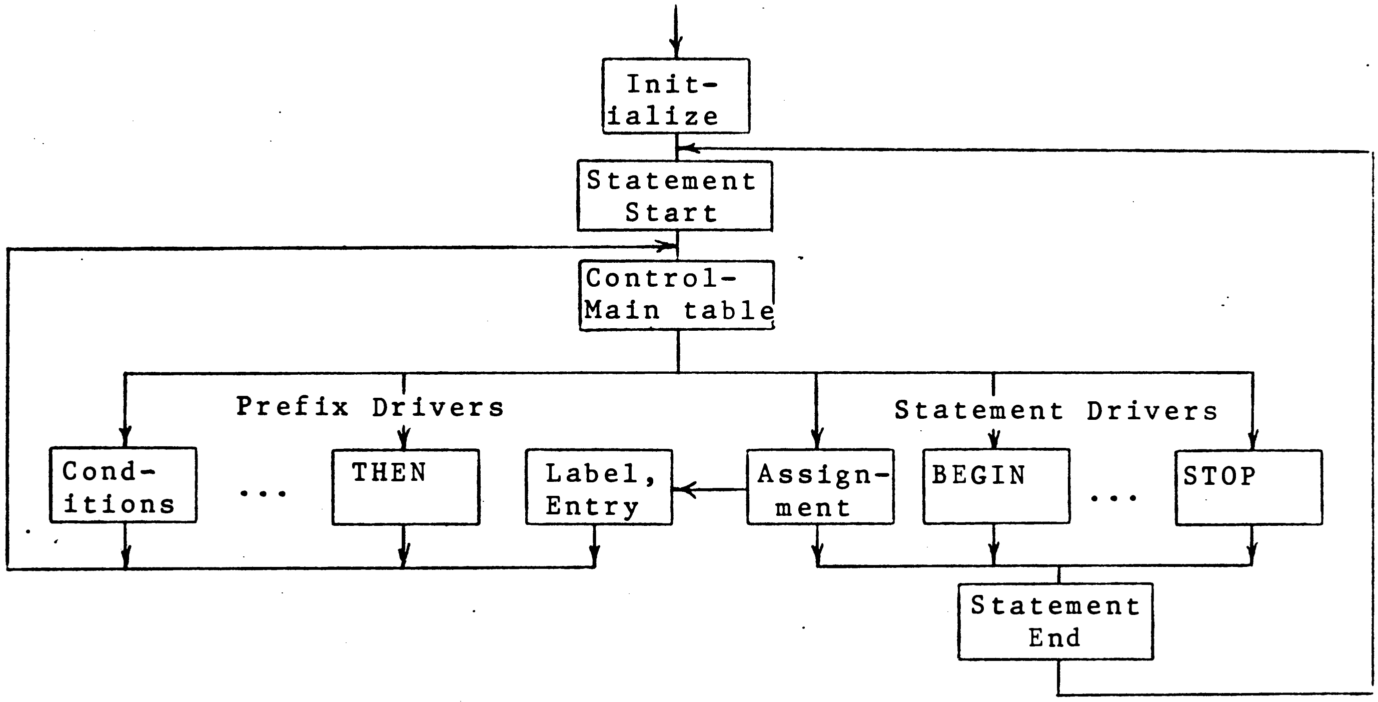

Figure 2. Organization Of The Syntactic Analyzer

The 'transition table' structure (4) pervades the syntactic analyzer. Most of the drivers are organized in this way, in some cases going as many as three tables deep in the analysis of a statement. Essentially this approach represents a two-dimensional 'branch table' where the row of the table can be considered to represent the class of the last symbol 'accepted' by the compiler, and the column represents the class of the next symbol presented to the syntactic analyzer by the lexical analyzer. More formally, the analyzer is a finite state machine where the state (row) represents the class of the last symbol accepted, and the column is the class of the next input symbol. This approach is especially well-adapted to error-correction. Where a more conventional transition-table compiler has a relatively sparse table with most of the entries pointing to a common error routine, PL/C has different processing routines for each entry of the table. Some, in fact most, of these routines involve error conditions, but each takes an action that preserves the syntactic correctness of the output string (beta-code) and returns to the table to process the next symbol. For example, consider the transition table for a highly simplified expression analyzer for arithmetic expressions that consist only of operands and binary operators as shown in Figure 3.

'State', Class of last symbol:	Class of next symbol: operand	operator	other
operand	GOTO E11	GOTO R12	GOTO R13
operator	GOTO R21	GOTO E22	GOTO E23
none	GOTO R31	GOTO E32	GOTO E33

Figure 3. Transition Table For A Simple Expression Analyzer

Only four of the nine conditions in the table of Figure 3 are proper: routines R12, R21 and R31 accept the next symbol, add it to the output string, set the state of the machine and re-enter the table; routine R13 indicates a successful exit. All of the other routines face an error condition--they must issue a message and effect a repair. In general variations upon four basic tactics are available for local repair of syntactic errors:

1. Delete the next symbol and reenter the table.
2. Insert a synthetic symbol in the output string, change the state of the analyzer appropriately and reenter the table.
3. Replace the next symbol with a synthetic symbol and reenter the table.
4. Delete the previous symbol from the output string, change the state of the analyzer appropriately and reenter the table.

For example, routine E11, faced with two consecutive operands probably should either delete the next symbol or insert a synthetic binary operator. Routine E22, faced with two consecutive binary operators, could delete the next symbol, insert a synthetic operand, or delete the previous symbol. In each case the 'best' choice of tactic, and the choice of particular symbol to supply if one is to be generated, depend importantly upon context. Moreover in many cases it is not clear whether 'best' implies maximum probability of reconstructing what the programmer intended, or maximum probability of prolonging the useful life of the program. PL/C does not have a systematic algorithm to specify which repair tactic is to be employed under particular circumstances. This was a matter of judgement on the part of the implementer of that particular module as to what would constitute the most plausible repair. Hence, while PL/C uses a consistent and systematic method for applying corrections, it has no general model or theory that specifies just which type of correction should be applied in a particular instance.

An example of the use of the insertion strategy occurs when the next symbol is the keyword PROCEDURE and the state of the analyzer is not 'entry-name'. (The colon is considered an integral part of the entry-name and not a separate state. A colon symbol is generated, if not already present, by the routine that accepts an entry-name.) The insertion tactic is employed--a synthetic entry-name is generated and inserted in the output string. The table is re-entered and the analyzer is now prepared to accept the keyword PROCEDURE. The synthetic entry-name is of course never referenced in the user's program. This particular error is perhaps most likely to occur on the main procedure of a PL/C program and in this case the repair is satisfactory and the program will run as intended. If the omission occurs on a procedure other than the main procedure, that procedure cannot be called but there is at least some chance that the execution of other portions of the program will yield some useful diagnostic information.

The last resort of the syntactic analyzer, when it despairs of effecting a plausible repair, is to delete the offending statement-- that is, to replace it by a null statement that will produce an identifying message when encountered in execution. While this preserves the logical structure of the program and still maintains a syntactically correct program, it is nevertheless an admission of defeat and this recourse is used as seldom as possible. When it is necessary, recovery is greatly facilitated by the fact that the statement keywords (DECLARE, CALL, BEGIN, GOTO, etc) are 'reserved' in PL/C and cannot be used as identifiers. This means that the analyzer can (almost) just scan for the next reserved word to find its recovery point. Only prefixes and assignment statements require special conditional handling. (The alternative to keyword reservation is to scan for the delimiting semi-colon, but since the omission of semi-colons is probably the most frequent single PL/I error this seemed like a dubious strategy.) The reservation of the statement keywords also permits much earlier resolution of ambiguity and commitment to a particular statement driver and this undoubtedly contributes significantly to the compilation speed of PL/C.

While the entire syntactic analyzer could be constructed as a single massive transition table it saves a great amount of space and probably simplifies coding to partition the table and use a hierarchy of transition tables. The overall control of the syntactic analyzer is one table, whose entries point to the tables for the individual prefix and statement drivers. Most of these drivers in turn are controlled by a transition table. In the DECLARE statement driver there is a heirarchy of tables since a number of the attributes (INITIAL, for example) have sufficiently complex syntax to warrant their own table. A separate subroutine analyzes expressions for all of the drivers. In comparison with the simple three-state analyzer of Figure 3 the expression analyzer in PL/C consists of four separate tables, each having eight states and eleven symbol classes. The expression analyzer is, in effect, a push-down-automaton which saves its state and an indication of which table is active on top of a push-down stack each time that a left parenthesis is encountered. It then enters a new table. When a right parenthesis is encountered the top state and table of the stack are restored. This expression analyzer is both compact and very fast. The tables, their control and all of their processing routines require just under one thousand bytes of core.

Reporting Syntax Errors

One of the most distinctive characteristics of PL/C is its technique for informing the user of error detection and subsequent repair. The system produces a maximum of six error messages per statement, giving text as well as a reference number to a more complete explanation in the User's Guide (3). While these messages are probably as cryptic and confusing as with most compilers at least they reference source language symbols and statement numbers. However, the most effective communication rests not in the specific messages but in a reconstruction of the source statement that syntactic analysis is actually passing on to semantic analysis. This is done by a decompiling routine that transforms beta-code back into source language symbols for any statement that incurred one or more error messages. For example:

```
STMT 4             P2 PROC ORDER FIXED REORDER
IN   4          ERROR  SY09 MISSING :
IN   4          ERROR  SY0F MISSING KEYWORD
IN   4          ERROR  SY02 MISSING (
IN   4          ERROR  SY04 MISSING )
IN   4          ERROR  SY20 IMPROPER OPTION
IN   4          ERROR  SY0C RETURN ATTRIBUTES OVERRIDE DEFAULT
PL/C USES          P2: PROCEDURE ORDER RETURNS (FIXED);


STMT 9             PUT LIST ( X(I) Y(I) DO I = 1 TO N);
IN   9          ERROR  SY06 MISSING COMMA
IN   9          ERROR  SY02 MISSING (
IN   9          ERROR  SY04 MISSING )
PL/C USES          PUT LIST ((X (I), Y (I) DO I=1 TO N));
```

```
STMT 34            DCL 1 PAYREC, 2DIV, 3 PAY_RATE, 3 PAY_PD,
                        2 DIV, 3 PAY_CODE, 3 CUM_PAY;
IN   34         ERROR  SYFD SPACE MISSING BETWEEN NUMBER & LETTER
IN   34         ERROR  SY27 MULTIPLE DECLARATION
PL/C USES          DECLARE 1 PAYREC, 2 DIV, 3 PAY_RATE,
                     3 PAY_PD, 2 $V005, 3 PAY_CODE, 3 CUM_PAY;

STMT 2             DELCARE G, ( A1 B1 CHARACTER VARYING;
IN   2          ERROR  SY00 MISSPELLED KEYWORD
IN   2          ERROR  SY06 MISSING COMMA
IN   2          ERROR  SY02 MISSING (
IN   2          ERROR  SY11 MISSING EXPRESSION
IN   2          ERROR  SY04 MISSING )
IN   2          ERROR  SY04 MISSING )
PL/C USES          DECLARE G, ( A1, B1 CHARACTER(1) VARYING );
```

The first two examples above illustrate repair that is probably successful in the sense of recreating what the programmer intended. If these were the only flaws in the program it will run satisfactorily and PL/C will have avoided at least one program submission. The third example will probably not give completely satisfactory results (if the DIV field is used in the program) but at least there is some prospect of obtaining useful information about the structure of the program and flow-of-control. The fourth example illustrates the inherent limitation in a single-pass repair strategy. While the repair performed in this case is syntactically correct it is very unlikely that it has reconstructed what the programmer intended. The right parenthesis probably should have been inserted after B1 so that both A1 and B1 are CHARACTER but PL/C did not detect the error until the end of the statement was reached. By that time, in a single pass analysis it is impractical to find a more plausible point for the insertion of the missing parenthesis.

An interesting example is provided by the submission of a one-statement program:

```
STMT 1             PTU FILE(OUTPUT A+1 'CORRECTION.
IN   1          ERROR  SY00 MISSPELLED KEYWORD
IN   1          ERROR  SY1D MISSING EXTERNAL PROC
IN   1          ERROR  SY3B MISSING LABEL OR ENTRY NAME
PL/C USES          $L001: PROCEDURE;

IN   2          ERROR  SY04 MISSING )
IN   2          ERROR  SY22 IMPROPER I/O PHRASE
IN   2          ERROR  SY02 MISSING (
IN   2          ERROR  SYEB STRING CONSTANT RUNS ACROSS CARD BOUN
IN   2          ERROR  SY06 MISSING COMMA
IN   2          ERROR  SY04 MISSING )
PL/C USES          PUT FILE(OUTPUT) LIST(A+1,'CORRECTION.');

IN   3          ERROR  SY0E MISSING END
PL/C USES          END;

                ERROR  SY1C MISSING MAIN PROC
```

A seventh error-- MISSING SEMI-COLON-- was detected and corrected in statement 2 but a message did not appear (because of a limit on the error buffer stack). (The same thing occured in the statement 4 example given earlier.)

Since beta-code is not the only communication vehicle between syntactic analysis and the later phases there are a few situations in which repairs of errors are not effected in beta-code. For example, in the one-statement program above the user did not designate a MAIN procedure. When the omission is detected at the end of the source program an error message is issued and the error is repaired by setting a pointer (used by code generation) to the first external procedure of the program. (This is a relatively frequent error, and the repair is very often correct.) However, it was decided that it was just too much trouble to repair the beta-code for the first procedure statement so that it could be decompiled onto the listing. This problem arises most often in the analysis of a DECLARE statement. Many syntactic errors in a declaration are not revealed until several symbols beyond the error have been accepted in beta-code. The syntactic analyzer manages to create correct SDs in spite of such problems but does not have the ability to go arbitrarily far back in beta-code to correct the symbol string. This is only of concern to decompilation since the later phases of the compiler depend upon the information in the SDs and not the beta-code for the attributes of identifiers. The only solution to date has been to suppress the decompilation in these cases and substitute a message asking the user to consult the cross-reference and attribute listing to determine what attributes were actually applied to what variables. The XREF and ATR options are automatically turned on in this case.

Semantic Analysis

The second pass of the compiler is performed by a routine labeled 'semantic analysis' in Figure 1. Since PL/C permits normal PL/I use-before-declaration it is not possible, in a single pass over the source text, to detect all errors of dimensionality and type. Since PL/C also permits multiple use of identifiers it is also not entirely possible, in a single pass, to determine just which instance of a particular identifier is referred to by a particular reference. For example, syntactic analysis can insist that the keyword GOTO be followed by an identifier. However, that particular identifier may be used in several different ways in the program and syntactic analysis cannot always know which instance is intended and cannot check that that instance is in fact a label constant or label variable. Hence, a second pass is made over the internal beta-code after all of the declarations have been scanned. The function of the semantic analysis pass is to transform beta-code to gamma-code and modify the SDs as necessary to:

1. Transform all identifier references so that they point to the SD for the proper identifier, rather than the common

descriptor for the identifier name.

2. Ensure that all expressions are consistent and conformable with respect to type and structure.

3. Parse all expressions into a combination prefix-postfix form. All infix operators--including assignment equal and multiple assignment comma--and the unary operators 'plus', 'minus', and 'not' are represented in postfix form. Array, user-defined procedure, and built-in function references are left in prefix form with the source syntax of an argument list used to delimit operands. For example, the expression:

```
A = B(1 + MAX(I * J, 100))
```

would have symbols in gamma-code in the following order:

```
A B ( 1 MAX ( I J * , 100 ) + ) =
```

These tasks do not necessarily have to be performed in a separate pass-- they could be accomplished as a preliminary task of the pass that generates object code. The decision was made to accept the overhead of an additional pass to perform these tasks in order to be able to perform more ambitious and sophisticated error correction. The semantic pass is relatively fast and efficient. Since semantic analysis is only interested in the <u>expressions</u> in beta-code it makes a simple linear search for the distinguishing symbol placed by syntactic analysis to mark the beginning of each expression. The low-order bits of this <start expression> symbol convey all necessary context information. As indicated in the earlier example the context symbol for the Boolean expression of an IF statement will specify that a scalar expression of string type is required. Semantic analysis will enforce this requirement without having to know what source construction contained the expression.

For each expression, in turn, semantic analysis constructs a parse-tree, in postfix order. Simultaneously, it produces a postfixed, resolved form of the expression, and determines the conversion class (type) and structuring (dimensionality) of each subexpression. Where a conflict between the conversion class (arithmetic, string, label, pointer) of a subexpression and its governing operator arises semantic analysis reports an error and repairs it by modifying the parse tree. If no errors are detected the postfixed string replaces the original infixed string in beta-code, creating gamma-code. Since the postfixed form is always less than or equal in length to the infixed form this can always be done in the same space. If errors are detected a recursive top-down pass is made over the corrected parse tree to get the new postfixed string. The correction algorithm over the tree is quite simple and straightforward at the moment, but with the procedure logically isolated and accessible there is an invitation to substitute more sophisticated techniques.

When errors are encountered by semantic analysis error messages are produced and the repaired gamma-code is decompiled and displayed on the source listing. Since this occurs in a second pass it is no longer possible to place the messages and reconstruction immediately after the offending statement, so that all semantic diagnostic communications appear together at the end of the source listing.

Code Generation

PL/C code generation is designed to produce reasonably efficient, error-tolerant object code for immediate execution. No global optimization is performed, but some effort is expended to generate locally optimized code sequences. Code generation is less dominated by diagnostic considerations than the other phases of the compiler. It is provided with input (gamma-code) that is very largely error-free. The only errors that could remain in gamma-code have to do with certain special PL/C restrictions on the source language-- e.g., all array parameters must be declared with asterisk bound fields-- and very special semantic constraints placed on expressions in some obscure contexts -- e.g., the second argument to the REPEAT built-in function must be an unsigned, fixed decimal constant. Since this type of error is easily detected in the normal course of code generation the early phases were deliberately permitted to overlook them. The diagnostic emphasis is also apparent in some of the code that is generated. For example, at each source statement boundary in object code, instructions are inserted so that during execution the program will 'know' the source number of the statement that is being executed at each moment and hence can report that number to the user if trouble develops. Similarly for each source label or entry-name, code is inserted to increment an associated counter each time that it is encountered in execution so that usage frequency can be reported.

Object code is generated into the same space that beta and gamma-code have occupied. To permit this the self-relocating gamma-code is shifted from the bottom to the top of the workspace at the beginning of code generation. Object code is then generated into the bottom of the work space. Eventually, depending upon the amount of workspace available and the length of the source program, the object code begins to overlay the beginning of gamma-code. Hopefully the portion overlayed has already been translated. Since gamma-code and object code are of roughly the same length (an average of about 50 bytes per source statement) space becomes available at approximately the same rate as it is consumed. If object code 'catches up' to gamma-code the compiler stops and requests a restart with a larger core assignment so that object code can have a greater 'headstart'.

Although code generation is not dominated by diagnostic considerations the strategy and structure of that phase of PL/C are quite unusual. The code generator is divided into three major functional units: the statement translator, the expression translator and the coder. The translators make one pass through the gamma-code to produce an equivalent sequence of instructions for an artifical PL/I machine. To produce this sequence, the translators must:

1. order the operations of the program so that only evaluated operands are used in any operation,
2. make explicit the sequencing of expression evaluation by inserting the appropriate (conditional) branch instructions between expressions, and
3. expand operations involving aggregates (arrays and structures) to repeated operations on scalars.

Storage allocation is also performed in this process.

As each PL/I-machine instruction is generated by the translators, a subroutine associated with the instruction is executed interpretively by the coder to produce the corresponding 360 machine code. Each subroutine, or <u>code template</u>, is written in a macro interpretive coding language (ICL) developed especially for this purpose. ICL is best viewed as the machine language for a special purpose automaton for generating machine instructions--that automaton being the coder. ICL is reasonably powerful, permitting, in addition to the generation of machine instructions, conditional branching, the setting and testing of flags, and calls to both machine language and ICL subroutines.

The coder automatically keeps track of the location of each PL/C program variable. The generation of accessing code is wholly a function of the coder, as is the management of registers and temporaries. For example, the code template for FLOOR(X), where X is REAL FIXED BINARY (p,q) is:

```
    GIFR   L1,REAL1          skip to L1 if X is in a register
    GLOADG REAL1             generate load of X into register
 L1 GRS    SRA,REAL1,R0,(R0,A1)  generate right shift
                                   to delete fraction bits
    GFIN                     finish code sequence
```

Here, the number q has been placed in cell A1 by previous instructions. The GRS ICL statement causes generation of an RS format instruction as follows:

```
GRS opcode,register1,register3,(base2,displacement2)
```

The opcode appears literally. The other three items are pointers to various compile-time cells describing the run-time environment. Thus, R0 points to a description of general register zero. REAL1 is a pointer to the real component for the entry on the top of the compile-time stack. Recorded in REAL1 is information which allows the coder to generate code to access the quantity involved no matter how complicated the path to that quantity may be. As code is generated (by the GLOADG ICL statement) to move X from storage to a register, the coder records X's new conceptual location in the compile-time stack entry REAL1. Any subsequent reference to X, as in the GRS ICL statement, will automatically use its register location.

The organization of the statement translator, shown in Figure 4, is strikingly similar to that of the syntactic analyzer (Figure 2). Each statement in gamma-code is viewed as a string of expressions held together by a network of pointer symbols. The statement drivers use these pointer symbols to select in the proper order for translation the expressions which make up the statement. The selected expressions are translated by a single general-purpose subroutine--the expression translator--which is called by the drivers as needed. The drivers themselves generate code to control the evaluation of expressions and to dispose of the values computed by expressions within the statement in the manner appropriate to the statement.

The expression translator is prepared to handle the most complex PL/I expression in any context, even though the language restricts the form of expressions in many contexts. This is a good example of a philosophy used throughout PL/C. Wherever possible PL/C compilation routines were designed to accomodate the most general case of input, and relied on previous actions to restrict input to be appropriate to the particular context. This 'principle of generality' has two advantages. First, the implementation is more compact because of reduced 'special casing'. Second, and more important to the objectives of PL/C, this provides an opportunity to 'repair' errors simply by issuing a message that a language constraint has been violated and then proceeding to to accept the construction exactly as given by the programmer. That is, in some cases this strategy permits PL/C to be slightly less restrictive than the language definition, although it will issue a warning message for the sake of compatibility. For example, PL/C will accept a non-constant argument to the built-in function LOW, although flagging it as a PL/I error. This particular type of 'error-repair' has very high probability of reconstructing exactly what the programmer intended.

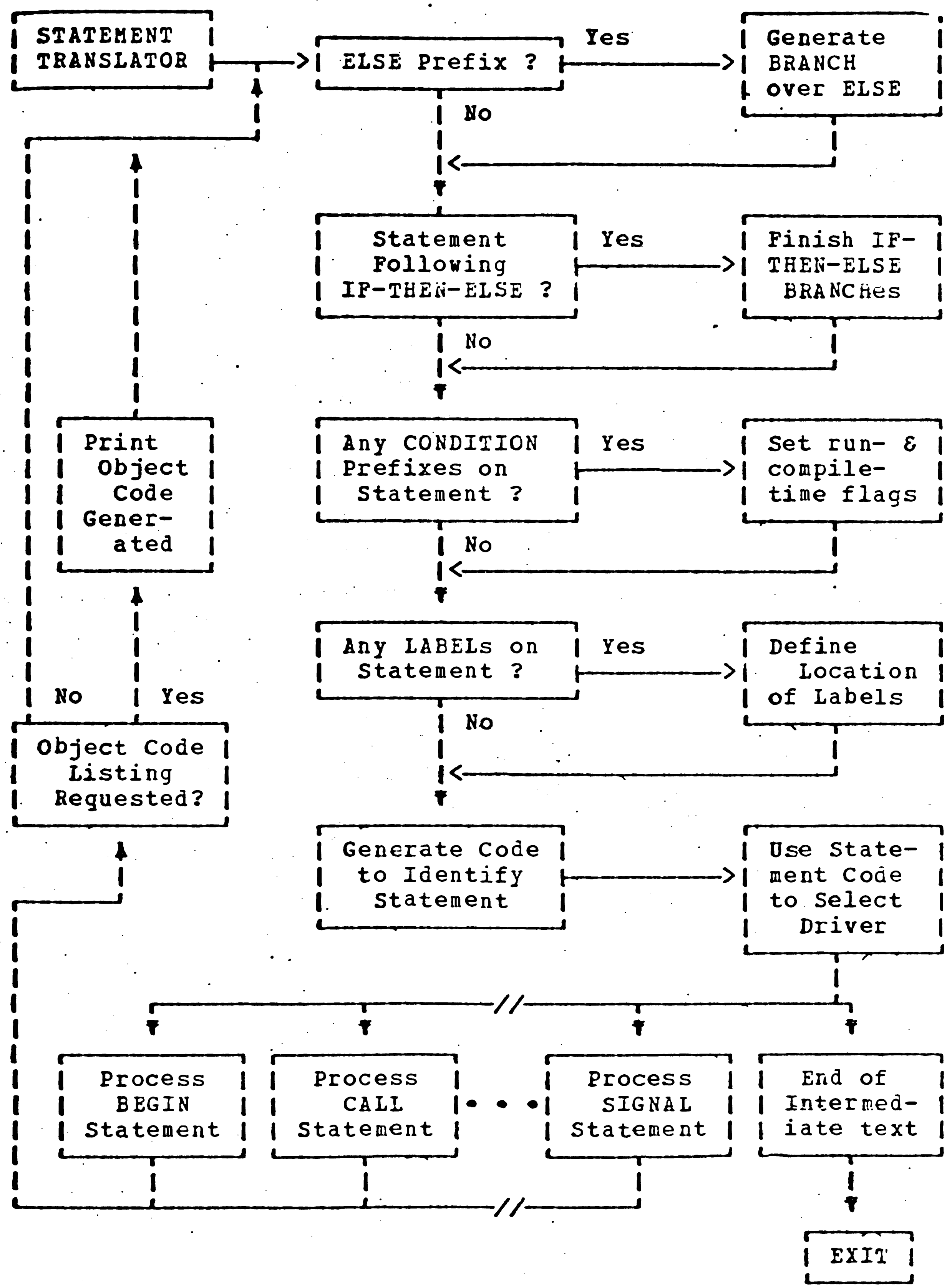

Figure 4. Basic Process Flow In Statement Translator

The combination postfix-prefix form of gamma-code was designed to facilitate the task of the expression translator. The organization of this routine is shown in Figure 5. As long as no prefix operators appear the translation proceeds in the conventional manner:

-A description of each operand is pushed onto the compile-time stack as its gamma-code symbol is scanned.

-When an operator symbol is scanned, code is generated using the information on the top of the stack.

-A description of the result replaces the descriptions of the operands.

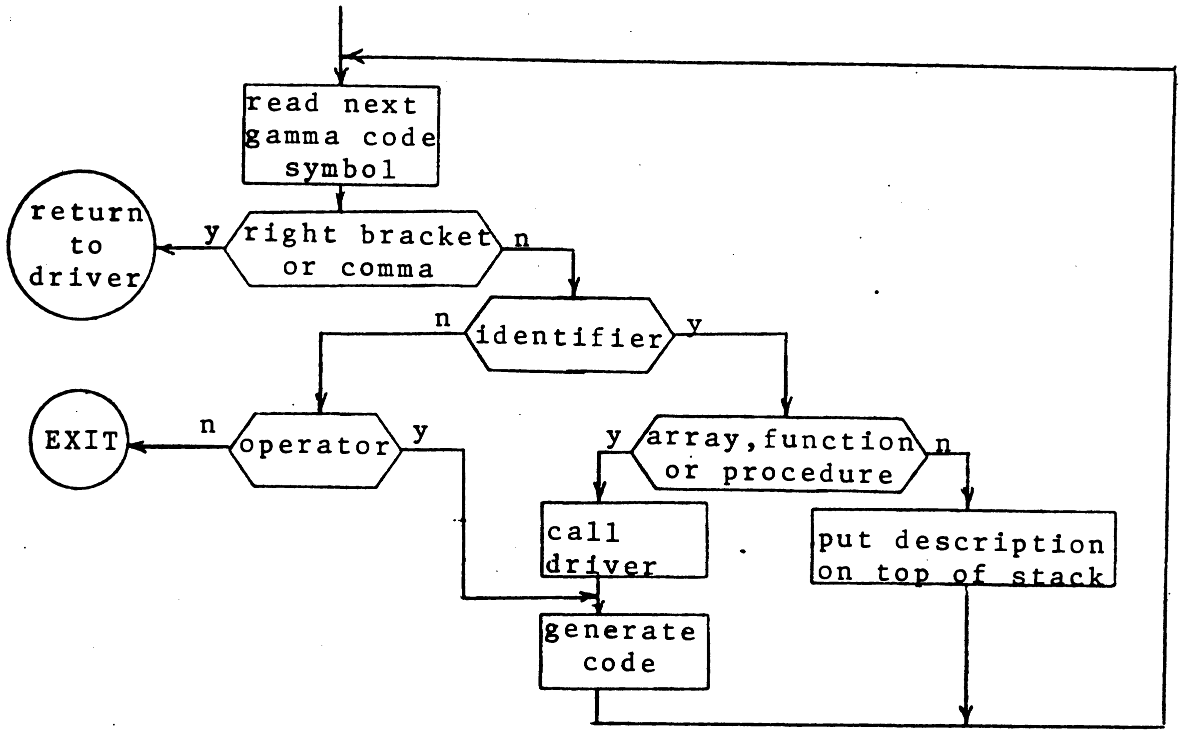

Figure 5. Translation Of Prefix/Postfix Expressions

The code generation step is table driven. Associated with each operator symbol is an <u>action table</u> having three or more entries. The first two entries designate machine language subroutines which are called to convert operands to the proper type, determining result attributes, or testing operands for compatibility and substituting corrections if necessary. The third and following entries in the table identify ICL routines (templates) which will generate code to perform the operation in the manner appropriate to the types of its operands. The template which is selected is determined from the output of the machine language routines.

Because array subscripts and arguments to built-in functions and user-defined procedures usually require special translation, the constructs in which they appear are left in prefix form. Their translation is handled by special drivers which are called before the operands--subscripts or arguments--

are translated. The operations of each driver are essentially the same. First, the driver sets up the data structures necessary to process the forthcoming operands. The expression translator state is saved on the compile-time stack and then a new state is determined. Some code may be generated by the driver at this time to prepare run-time structures needed in the evaluation of operands. After this basic setup has been completed, the operands are translated from left to right via successive (recursive) calls to the expression translator. Before each operand is translated flags and pointers are set to establish a translator state appropriate to the operator/operand combination. After an operand has been translated, code may be generated by the driver to dispose of the argument value, or its description may be left on the compile-time stack for later processing. After all operands have been translated code to compute the final result of the operation is generated. This last step uses action tables similar to those used to translate the postfix operators.

As shown in the appendix the code generator represents approximately one-third of the compiler. One-half of the code generation phase is the expression translator; one-eighth of it is the coder. About one-eighth of code generation is written in ICL. From one-fourth to one-third of total compilation time in PL/C is spent in code generation and half of this is spent in the coder.

Execution Supervisor

The execution supervisor of PL/C is unusual primarily for its perseverance. Almost all of the run-time tests are constructed to issue an appropriate message (completely in source language terms), effect some form of repair and continue execution from the point of the error. Execution is terminated when the count of such events reaches a user-specified limit (which, of course, can be 1). There are a few error conditions--a faulty GOTO, for example-- where no plausible repair is obvious and execution is immediately terminated.

The range testing of subscript values was first done in CORC (1) and has since become a standard technique in diagnostic systems. This particular test is of special significance since it not only warns the user of a computational problem; it effectively precludes the user's program from destroying itself or the supervisor under which it is being run. CORC demonstrated that this type of error was the only way in which a user program in an ALGOLic language could get 'out of bounds' and that by ruling out such errors one could guarantee the integrity of the compiler/supervisor and permit highly effective program batching. PL/I presents some interesting new problems in this regard. First, the language permits explicit enabling and disabling of subscript range testing and, after much discussion, it was decided to permit this disabling in PL/C. This significantly improves run-time performance for some programs but jeopardizes the integrity of the supervisor and hence the possibility of program batching. As an installation option the disabling of SUBSCRIPTRANGE may be prevented. Secondly, the list processing features of PL/I offer the user a new way of stumbling out of bounds. Whenever the list processing features are added to PL/C some form of execution monitoring of the values of pointer variables will have to be provided if the integrity of the system is to be guaranteed.

The symbol table (the SDs), containing complete information on all program identifiers, is present at execution time in PL/C so that source language communication can be provided. Therefore any operation which could be compiled into directly executable code could also be performed interpretively at execution time. A certain amount of interpretive execution is required by PL/I (GET DATA for example) but PL/C does as little interpretation as is practical in order to achieve reasonably efficient execution.

In PL/C the code generator produces directly executable code for the evaluation of all expressions in the program and for the transfers of control directed by DO and IF statements. The execution supervisor is responsible for I/O, string manipulation, the GOTO statement, dynamic storage allocation, and the management of the run-time stack during block entry and exit. These supervisor services are requested through calls to run-time support routines which have been compiled into the object code.

The dynamic storage allocation process provides a good illustration of the choice that must be made between compilation and interpretive execution. Although object code _could_ be compiled to perform this process it would be executed relatively infrequently and would require such an inordinate amount of machine code that interpretation of tables at run time was judged a preferable choice in PL/C. Space for AUTOMATIC and STATIC variables is allocated in block-related units called _activation records_ (AR) using a simple stack structure. During execution the stack is free to expand into all of the region assigned to PL/C by the operating system that is not already occupied by the supervisor routines, the symbol table or object code. The initial portion of each AR is formatted by the code generator for non-string scalar variables, dope vectors for strings, dope vectors for arrays and structures, save areas, intermediate result temporaries, and other miscellaneous cells used for recording the run-time environment. Information regarding this initial allocation is recorded in the symbol table where it will be accessible to both the code generator and the execution supervisor.

When a block is entered during execution the execution supervisor creates an AR. A pointer chain constructed by syntactic analysis runs through the SDs of all variables in the block which will need storage in addition to the initial allocation. Associated with each SD on the chain is a dope vector skeleton. The skeleton contains the values of constant array bounds and string lengths and the address of a compiled subroutine which will compute the value of any bounds or lengths which were given as an expression. After initializing the dope vector in the AR from the skeleton and invoking the subroutine to complete its bounds and length fields, the execution supervisor checks the bounds and length fields for legal values, computes the space needed for the variable and allocates it at the end of the AR. If the variable has the INITIAL attribute the execution supervisor invokes another compiled subroutine which performs the initialization. By generating directly-executable code for the evaluation of all expressions involved in the allocation of storage for a variable the block entry process is considerably accelerated. To have to interpret a general PL/I expression with type conversions and built-in functions would be both slow and more space-consuming than the equivalent compiled code. On the other hand it would be a waste of object code space and compilation effort to generate code to do the straight-forward and repetitive tasks common to the allocation of all variables.

The monitoring of variables to detect use before initialization was also introduced in CORC. The PL/C strategy for this monitoring is a good example of frequency sensitive coding--a test was designed to be efficient for correct usage at the cost of extra work when an error is encountered. Since no compile-time analysis for this condition is possible a test must be compiled into the object code at every reference to every variable (unless optionally the user suppresses the feature). Therefore the test must use as little time and space as possible. However, if an error is detected, the name and address of the variable involved and the statement number and code address of the location of the error must be available so that an informative error message can be printed, the variable initialized (to zero), and execution continued from the point of error. Because in recursion there may be more than one generation of a variable the indication whether a variable has been initialized must be retained with each generation and not in a central location such as the symbol table. To conserve space it is convenient if one of the possible values of a variable itself may be designated as the 'uninitialized value'. All variables are set to this value when they are allocated, and on each reference the variable is tested for this particular value. The first assignment to the variable --ie, its initialization--will change its value and thus automatically turn off the indicator. The value representing an uninitialized FIXED BINARY variable in PL/C is -2**31. (All other variables use the long floating point number -8*16**-71 in a similar manner.) This is the smallest fixed point number on the 360 and so will not be missed by the user. (It is technically too large for a FIXED BINARY(31) variable, but most implementations, including PL/C, allow it.) Of more importance to the use-before-initialization test is the fact that -2**31 is the only representable value whose absolute value is too large for the machine. The test for use-before-initialization is, therefore, to attempt to take the absolute value of the variable with the instructions:

```
        L     R,<variable>           load value into register
        LPR   R4,R                   load positive value into R4
```

If the variable has been initialized properly the sequence adds only six bytes and two instruction executions (less if the variable was already in a register). If the variable had not been initialized the LPR will generate a fixed-point overflow interrupt in the hardware. The fact that an LPR into R4 caused the interrupt indicates to PL/C that the error is an uninitialized variable and not a normal arithmetic overflow. The old PSW provides the address of the error. The statement number is always present in the AR on the top of the stack. The address of the variable which caused the interrupt is obtained by decoding the address field of the load instruction. Using this address, the information in the stack, the chains of SDs in the symbol table, and more than a little work, the SD for the variable which has been referenced improperly is located. From this the name of the variable is printed. Hence the test is very efficient if it fails, and adequate, with considerable recovery effort, when it succeeds.

There are many critical time/space choices that must be made in the design of the execution supervisor. For example, it was decided that all built-in functions would be made core-resident to simplify the I/O interface with the host operating system and to avoid the delay involved in dynamic loading. However, to limit the space required for these routines only the long floating point form of the math built-in functions are provided and conversion is required. Internally only two arithmetic data-types are recognized by PL/C-- long floating point and fullword integer. All coded-arithmetic data types are mapped into these representations by the code generator. Except for the extra storage taken up, the user is never aware of this mapping because the variables participate in expression evaluations just as their declarations dictate. The price of this simpliciation is felt in record I/O and the UNSPEC built-in function which must convert between internal and external representations. Use of these features was considered relatively infrequent in the environment in which PL/C would be used and the price was accepted. Another move to reduce the number of cases which must be handled was to represent BIT strings internally as CHARACTER strings--only one set of string manipulation routines are required.

The execution speed of a program under PL/C relative to the same program under PL/I-F is rather heavily dependent upon the content of the program. A program that involves heavy use of conversion of representation and diagnostic checking (for example, record I/O and use of subscripted variables) will run several times longer than under PL/I-F. Programs that are dominated by scalar computation and block entry and exit are quite comparable. Performance on student jobs ranges from one-half to twice as much execution time as PL/I-F. The comparison is slightly unfair since the PL/I-F default for SUBSCRIPTRANGE is disabled, but for PL/C it is enabled. The margin is considerably narrowed if subscript checking is also disabled in PL/C. On short jobs, of course, the execution advantage of a production compiler may be offset by the absence of a link-edit step in PL/C and by the considerable advantage of PL/C compilation speed.

Diagnostic Extensions To PL/I

The discussion so far has concerned the implementation of a compiler that provides enhanced diagnostic assistance for the user while still being strictly compatible with the IBM F-level implementation of PL/I. However, PL/C also provides a convenient vehicle with which to experiment with explicit diagnostic facilities that can be _user-controlled_ at the source level.

Tracing And Snapshot Dumping

The most powerful techniques for program debugging in the batch processing environment are still the classic _trace_ - dynamically following the progress of the program, and the _dump_ - periodically displaying the instantaneous status of relevant parts of storage. Unfortunately, most high-level programming languages do not particularly assist the user in either regard. PL/I provides some facilities for this purpose, and PL/C has extended these facilities considerably.

The principal tracing facility of PL/I is the CHECK prefix. Applied to a PROCEDURE or BEGIN block this lists the names of identifiers to be dynamically monitored during execution of that block. Unfortunately, it is somewhat inflexible (it can only be applied to an entire block) and it is static rather than dynamic in control. PL/C has strengthened the check facility by adding CHECK and NOCHECK statements in order to give flexible and dynamic control over the CHECK action. The NOCHECK statement simply suppresses the printing that results from the raising of the CHECK condition, and the CHECK statement resumes the printing.

The tracing facility is further extended by the addition of FLOW and NOFLOW prefixes and the FLOW and NOFLOW statements. They are quite analogous to CHECK but are concerned with the flow of control of the program. When the FLOW condition is not disabled, it is raised whenever a statement is encountered that would potentially alter the normal sequential flow-of-control. The PL/I statements that raise the PL/C FLOW condition are CALL, DO, GOTO, IF and RETURN. In-line procedure references and exceptional conditions which would cause an ON-unit to be entered will also raise the FLOW condition. The standard system action is to make an origin-destination entry in a queue whose contents may be displayed by the PUT FLOW statement and the post-mortem dump that is automatically provided by PL/C. Depending upon the appearance of FLOW and NOFLOW statements the standard system action may instead print an origin-destination line. For example:

```
0018 -> 0038        0042*( 0063 -> 0038 )
```

would indicate that flow-of-control went from statement 18 to 38 and then 42 times in succession went from 63 to 38. This standard action can of course be altered by a user-supplied ON FLOW unit. Optional parameters on FLOW control the extent and depth of its action.

A dumping facility is provided by the addition of non-standard phrases in the PL/I PUT statement:

PUT FLOW; displays the recent FLOW history.
PUT SNAP; displays the current 'calling' history.
PUT ALL; displays the current values of all automatic, scalar variables in the blocks active when the statement is encountered.
PUT ARRAY; displays the values of arrays as well as scalar variables.
DEPTH(exp) can be used with any of these (except PUT FLOW) to limit the nesting depth for which the display is to be given.
PUT OFF; initiates suppression of all printed output (permitting various diagnostic display statements to be left in a program and selectively activated.)
PUT ON; resumes printing of output.

When the execution of a PL/C program is terminated a 'post mortem dump' is automatically produced (unless the user has specified NODUMP on the initial program option card.) The dump includes the following information:

1. The final values of all automatic, scalar variables in the blocks active at the time of termination. (The user has an option to include arrays.)
2. The final values of all static or external scalar variables.
3. The dynamic nesting of procedure, block and ON unit entries at the point of termination.
4. The current values of the condition built-in functions for each active ON unit.
5. A count of the number of times encountered for all entries and labels in the program.
6. A history of the last 18 changes in flow-of-control.
7. Core utilization statistics.

Pseudo-Comments

In order to permit the introduction of these non-PL/I constructions in PL/C and still preserve strict compatibility with the IBM processor for PL/I, the concept of a 'pseudo-comment' has been introduced. PL/C permits sections of source to be treated <u>either as source text or as comment</u>, depending upon an option specified on the initial program option card. The appropriate text is written as a normal PL/I comment--except that a colon, or an integer is given as the first character of the comment. For example:

```
/*5 X(I) = P(I+K); */

/*: PUT DATA(X(I)); */
```

With normal default options PL/C will treat these as comments (as would the IBM processor.) However, if the option COMMENTS is given on the initial program option card PL/C will include as source text the content of all comments whose first character is a colon-- hence the PUT DATA statement of the example would be included in the source program. If the option COMMENTS=(5) is given then PL/C will scan as source text the content of all comments whose first character is either a 5 or a colon-- hence both of the statements in the example would be included in the source program. Since the integers 1 to 6 may be used this means that the programmer can establish seven different classes of 'pseudo-comments' in his program and selectively include and exclude their contents from compilation just by changing the specification on the initial program option card. While this feature was included in order to preserve compatibility with PL/I it seems likely that programmers will find many other uses for it.

Reversible Execution And A Generalized ON Statement

One of the inherent difficulties in program debugging is that the cause of a problem and its manifestation may be separated by a great amount of executed code. That is, an error may occur and create conditions that do not reveal the existence of the error until much later in the program. One is then faced with the task of working backwards over the tree-graph of the program in search of the error. The usual approach is to 'back up' discretely by running the program repeatedly, stopping or invoking diagnostic facilities at earlier and earlier points. It seemed that it might be useful to be able to literally back up the program. That is, at an arbitrary point in execution (presumably the point at which an error is revealed) be able to <u>reverse the path of execution</u> in order to re-establish conditions that existed earlier in execution. The possibility was intriguing and a special version of PL/C was implemented that provided this ability (13).

This 'palindromic PL/C' involves two new source constructions. The first is the RETRACE statement:

```
RETRACE <condition> AND <statement>;
```

This has the effect of retracing execution until the specified <condition> becomes true, at that point executing the specified

<statement>, and then resuming normal forward execution from that point. For example:

```
RETRACE X3 < 0 AND CALL DIAGNOSE;
```

Alternate forms of RETRACE permit one to retrace a specified number of statements, or to a specified label, or to the last point at which a specified variable was assigned a value.

The second new construction is a generalization of the PL/I ON statement:

```
ON <expression> <BEGIN block>
```

The ON unit is activated whenever the specified <expression> becomes true.

These two facilities would seem to make possible an interesting new approach to program diagnosis. Rather than include anticipatory diagnostics in the hope of capturing information in the neighborhood of an error, one can run to a point where the error is actually detected and then use the RETRACE statement to attempt to reconstruct the conditions that led to the error. The generalized ON statement permits increased flexibility in detecting error conditions. Although the implementation has been completed and run experimentally no general field-test has been conducted and it remains to be seen just how useful and powerful this capability really is.

The modifications to the PL/C compiler to permit programs to run backward as well as forward were neither extensive nor difficult. Essentially all that was required was a 'push-down stacking' mechanism for value assignment and transfer of control. Reversing transfer of control across a block boundary does present the complication of reestablishing the proper environment so this was avoided in the initial implementation by limiting reversal to a single block. Elimination of this restriction would not be prohibitively difficult. The implementation of the generalized ON statement requires the insertion of testing instructions immediately after the code for every source statement that changes the value of one of the terms in the <expression> of the ON statement. This imposes significant overhead in both space and time and must be used with discretion. Preliminary testing of both features over a modest number of examples seemed to indicate that program space was increased by 40 percent and that execution time was approximately doubled. This is a non-trivial cost, to be sure, but it is still significantly less than would be required for interpretive execution, and at least in some cases may offer some of the advantages of an interactive system without its increased cost. Since the usual approach to debugging involves repeated runs and voluminous output it is possible that it might be relatively economical to run at half-speed if the required number of runs was thereby significantly reduced.

Conclusions

We have encountered two principal criticisms of the diagnostic approach represented by PL/C. First, there is concern that this approach tolerates poor and sloppy practice and postpones the necessity of developing the discipline requisite to good programming. It permits students to become dependent upon a degree and type of assistance that they will not receive from other translators. There is some merit to the argument, but it might also be interpreted more as a criticism of the diagnostic assistance provided by most other translators. Secondly, some users believe that PL/C usefully repairs only the most trivial of errors and that in general the execution of a repaired program is a waste of computer time. In our experience PL/C, and its predecessors, have seemed to correctly repair a useful fraction of the punctuation errors that are endemic with neophytes and which even afflict programmers with more experience. While its repair of significant syntactic and semantic errors is less often successful in the sense of being able to recreate what the programmer intended, it is very often successful in prolonging the life of a program sufficiently to yield additional useful diagnostic information. Eight years of experience with this approach at Cornell have convinced the authors that this results in a significant reduction in the number of job submissions required to obtain successful execution of a program. Less-biased users of PL/C at institutions other than Cornell have reported reductions of 30-50 percent in the number of trys required to achieve successful execution. As far as we know, the arguments, both pro and con, are based on general impressions and observations. We have not conducted any carefully controlled experiments and know of none elsewhere that would supply quantitative information to this debate.

PL/C clearly demonstrates that compilers can provide more diagnostic assistance than has typically been exhibited. Moreover this is true even for rich and sophisticated source languages. The techniques of error-repair, which had previously been explored only on severely restricted languages appear to be equally applicable and effective on PL/I. PL/C undoubtedly represents an extreme position and it is unlikely that translators intended primarily for populations not dominated by neophyte programmers would go this far. Certainly in file processing applications one would never permit execution of a repaired program since execution times are large relative to the cost of recompilation, and the consequences of an improper run can be serious. However, even in such an environment it is not inconceivable that it would be productive to allow a repaired program to execute briefly, assuming that it protected the file from any alteration. In any event certain of the specific features of PL/C would be generally useful even if the concept of compiler perseverance is not. Specifically, the decompilation of a possible repair is a very effective way of communicating with the user. In many cases it describes the location and nature of the problem very effectively and suggests an appropriate response. It would seem that this technique would be especially effective in an interactive system where the user would have the immediate opportunity to accept the compiler's suggestion or to substitute his own repair. The explicit diagnostic facilities-- FLOW, CHECK, PUT and the pseudo-comments--also would seem to be generally useful. The generalized ON statement is a very powerful tool and could permit an entirely different organization of programming (8, 9), but it is prohibitively costly to implement this concept entirely in software and its realization must wait until some assistance can be provided in hardware (14).

PL/C would also appear to demonstrate that the cost of increased diagnostic assistance is not excessive--regardless of whether this cost is measured in terms of compiler implementation effort, compilation time or memory required. The diagnostic facilities are inextricably distributed throughout PL/C so it is not possible to state even roughly what they 'cost'. Nor does it seem particularly useful to contrast PL/C with PL/I-F, which had a different objective, or with WATFOR, which translates a different source language. Qualitatively speaking, the implementation effort could not be excessive since the first production release of PL/C was designed and implemented by a relatively small group working part-time for fifteen months. Space requirements were certainly increased--perhaps as much as doubled in the early phases--but this is generally not a critical limitation in a multi-phase, overlayed production compiler. Compilation speed is an interesting question. No doubt the diagnostic effort slows PL/C compilation. Nevertheless it is signficantly faster than any other compiler for PL/I, and apparently even faster than the interpreters of the language. In fact, the only translators that we know of that are faster than PL/C translate a significantly simpler source language than PL/I.

In very general terms we will consider the PL/C project to have been successful if it serves to raise the computing community's expectations with regard to the diagnostic services that a translator should provide.

Acknowledgements

In preparing this description the authors are spokesmen for the entire PL/C project group. H. L. Morgan and R. A. Wagner were, with the authors, the principal designers of the system. The implementation group also included M. Bodenstein, H. Cabassa, P. Dormont, R. Fisher, T. Kahne, R. Holt, S. Lisberger, N. Weiderman, K. Wong, W. Worley, S. Worona, T. Vari and M. Zelkowitz. We also wish to acknowledge the major contributions of W. L. Maxwell to the previous Cornell compilers that were influential in the PL/C design.

Appendix: PL/C Memory Organization

The logical structure of the compiler, as described in the body of the paper, is independent of the physical overlay structure. The choice of overlay structure for PL/C is an installation option. Where a large region is available the entire system can be made resident and no overlaying takes place during the processing of a user program and no reloading of the compiler is required between the programs of a batch. This is certainly the fastest form of the system. However, most installations reduce core requirements by having PL/C operate with three overlays per user program. In multi-programming systems the performance degradation associated with the references to auxiliary storage for this overlaying is very slight.

Figure A1 shows the core layout during each pass (bottom scale) and each overlay (top scale) for the typical 3-overlay version. The 'compiler module' section is of fixed size and a portion of this section is the only area affected by the overlay structure. The approximate size (in bytes) of each of these modules is shown. The function of all but the 'list option' and error message modules is discussed in the body of the paper. The list option is a 'disassembler' that optionally displays the object code produced by the code generator as it would appear as input to an assembler (although in fact there is no assembly phase in PL/C.) The error message modules contain the phrases from which the error messages are generated by a routine located in the control module.

The work space section is unaffected by the overlay structure and is automatically adjusted to fit whatever region is available. A portion of this work space is of fixed size and this imposes certain compilation limits on the number of names and static nesting depths that cannot be increased by running the compiler in a larger region. These limits are not unduly restrictive and are not often encountered. The principal areas determined by the nature and size of the source program--number of symbols, length of object code, array space, and run-time nesting depths--are all in the expandable work space which can be allocated to meet individual requirements. For pass 1 the available space is divided equally between area for new symbols and area for beta-code. SDs are created only during pass 1 so the area of the symbol table is fixed at the end of this pass. (If the SD space overflows compilation is terminated immediately with a request for more space. If beta-code overflows the allocated space it simply wraps-around and overwrites the initial beta-code. A flag is set so that compilation terminates at the end of pass 1 with a request for more space.) In the 2nd pass the beta-code is transformed to gamma-code _in place_. At the beginning of pass 3 the relocatable gamma-code is moved to the bottom of the work space to leave as much space for object code as possible. Object code is generated from the end of the symbol table upwards, overwriting the beginning of gamma-code if necessary. Since gamma-code and object code are of roughly the same length object code will not overwrite gamma-code before it can be translated unless the gamma-code occupies essentially the entire space. If this occurs translation is terminated with a request for more space. Figure A1 assumes a 100K region. The work space areas shown are those that would be generated by a program of approximately 150 source statements that introduced 110 new symbols.

When program batching is employed the control module must reinitialize the compiler between programs. This involves clearing the symbol table area (but not the beta-code area) and some portions of the transfer vector. In the overlayed version it also requires reloading the first overlay of the compiler.

The sizes shown in Figure A1 are based on Release 6 of PL/C. They do not include the routines for reversible execution and the generalized ON statement described in the paper.

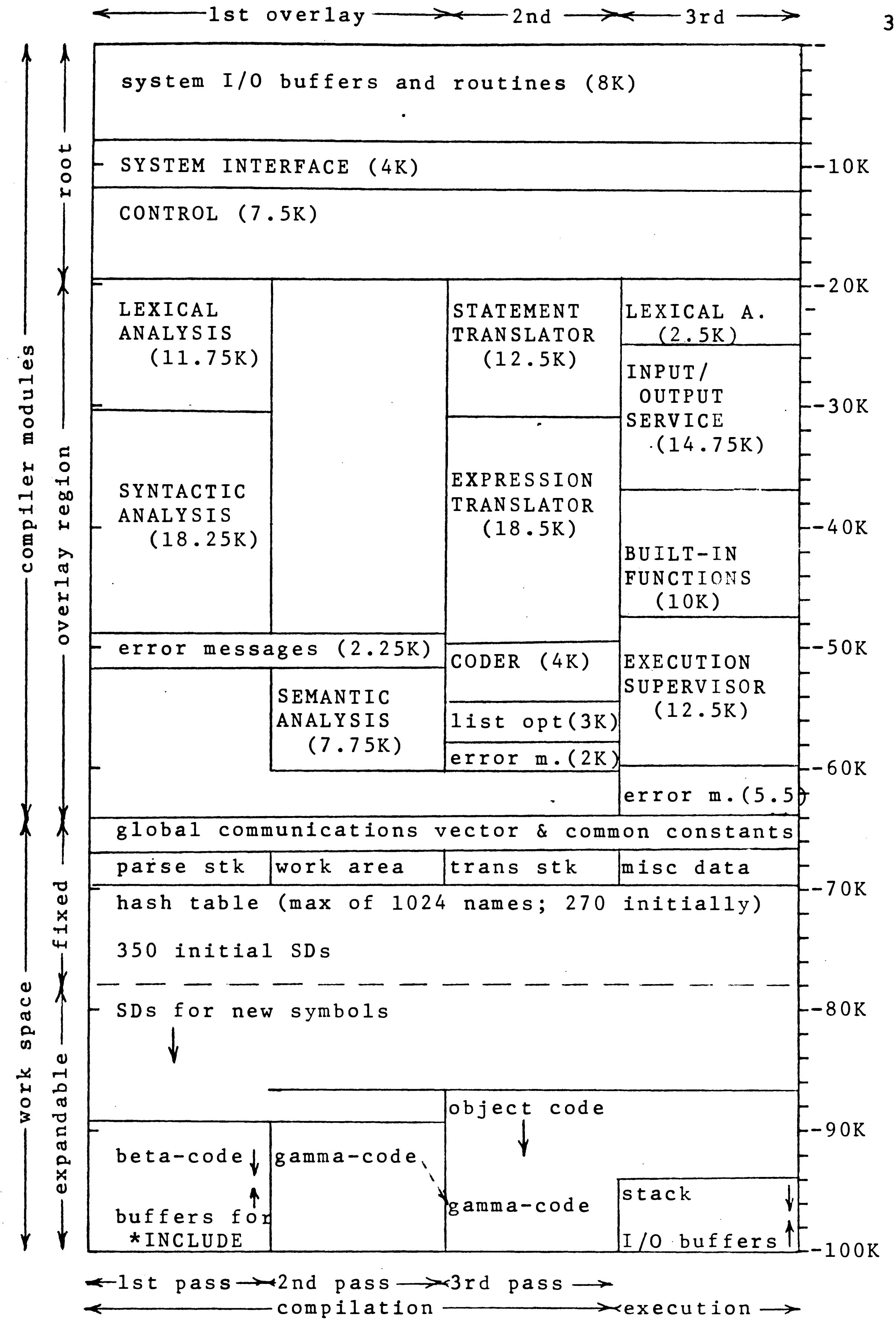

Figure A1. Core Utilization Of A 3-overlay Version Of PL/C Operating In A Region Of 100K Bytes (k=1024)

References

1. Conway, R. W. and W. L. Maxwell; 'CORC The Cornell Computing Language', Communications of ACM, June, 1963

2. Conway, R. W. and W. L. Maxwell; 'CUPL- An Approach To Introductory Computing Instruction', Research Report 68-2, Dept. of Computer Science, Cornell Univ., Also In Proc Of Conf On Computers In Eng'g, Commission On Eng'g Educ. 1966

3. Conway, R. W., H. L. Morgan, R. A. Wagner and T. R. Wilcox; PL/C, The Cornell Compiler For PL/I. User's Guide To Release 6, Dept. of Computer Science, Cornell University, August 1, 1971

4. Gries, D.; 'The Use Of Transition Matrices In Compiling', Communications Of The ACM, January 1968

5. Levy, J. P.; 'Automatic Correction Of Syntax Errors In Programming Languages', Research Report 71-116, Dept. of Computer Science, Cornell University, Dec. 1971

6. Lynch, W. C.; 'Description Of A High Capacity, Fast Turnaround University Computer Center', Proceedings Of ACM Nat'l Conference, 1967

7. Morgan, H. L.; 'Spelling Correction In Systems Programs', Communications Of The ACM Feb. 1970

8. Morgan, H. L.; 'An Interrupt Based Organization For Management Information Systems', Communications Of The ACM, December 1970

9. Morgan, H. L.; 'Event Sequenced Programming', Technical Report 119, Dept. of Operations Research, Cornell University 1970

10. Morgan, H. L. and R. A. Wagner; 'PL/C- The Design Of A High-performance Compiler For PL/I', Spring Joint Computer Conference 1971

11. Sackman, H.; 'Time-sharing Versus Batch Processing: The Experimental Evidence', Spring Joint Computer Conference 1968

12. Wilcox, T. R.; 'Generating Machine Code For High-level Programming Languages', Research Report 71-103, Dept. of Computer Science, Cornell University, August 1971

13. Zelkowitz, M.; 'Reversible Execution As A Diagnostic Tool', Research Report 71-92, Dept. of Computer Science, Cornell University, January 1971

14. Zelkowitz, M.; 'Interrupt Driven Programming', Communications Of The ACM June 1971