PL/C:—The design of a high-performance compiler for PL/I

by HOWARD L. MORGAN and ROBERT A. WAGNER
Cornell University
Ithaca, New York

INTRODUCTION

A general purpose production compiler faces many diverse and demanding tasks. It must obviously accept the full source language—including all rarely used and difficult to compile features; it must produce efficient object code; it must be prepared to accept very large and long programs—and accept them in pieces to be integrated later. It is obvious that by yielding on some of these requirements, and by sacrificing generality for efficiency for a particular class of program or user, improved compiler performance should be obtainable.

One large and important class of users are those just learning the language. Their programs are typically short, submitted all at once, and require only a modest subset of the language. They need to be compiled repeatedly to eliminate errors, and are often discarded as soon as they execute properly. In general, their execution time is very small relative to the total compilation time.

It should also be noted that such programs occur in tremendous numbers since classes of hundreds of students submitting programs daily are not at all uncommon. This class of program is further enlarged by the observation that even experienced programmers have rather similar requirements (except for requiring a larger subset of the language) during the development and checkout of a new system.

PL/C is a special purpose processor for PL/I. It has been designed and implemented by a group of faculty and students in the Department of Computer Science at Cornell University, both to serve the needs of instruction and program checkout, and to serve as a test vehicle for some novel concepts in compiler construction and diagnostic strategy. It is an open ended project, but the work to date has yielded a very high-performance compiler for a usefully large and strictly compatible subset of the PL/I language.

This paper describes the strategies and structures employed in the design of PL/C. Surprisingly many of the techniques used appear to be generally applicable in compiler design. The overall organization of the compiler project is discussed in terms of both the technical details and the personnel assignment strategy.

DESIGN OBJECTIVES

Before commencing the design of any large system, the overall objectives must be well thought out. Four major goals, along with several smaller objectives, shaped the design of PL/C. These were:

1. High-performance—PL/C had to compile programs of the size generally found in instructional situations an order of magnitude faster than the speeds obtainable with the “production” compilers. Efficiency of the generated object code was not nearly as important, although one would not want to completely overlook this question.

2. Diagnostic—Diagnostic assistance was at least as important as processing efficiency. The goal here was one of error correction rather than simply error detection. The PL/C reaction to the discovery of an error should be to supply an explanatory message, effect a repair, display the nature of the repair to the user, and continue processing. No source error should terminate the scanning of the program, and as few as possible should inhibit execution. In addition, as many execution errors as feasible should also be handled in this manner. This approach had been developed on a series of previous compilers at Cornell, beginning in 1962, and had been found to be remarkably effective.

3. Upward compatibility—in order to be useful as a debugging and checkout compiler, and to serve as a widespread aid, the subset had to be strictly upward compatible with the IBM F-level implementation of PL/I, both in language specification and in the semantics actually im-
implemented by IBM. Thus, any program which
will run under PL/C without incurring any
diagnostic messages will run under the IBM
implementation and provide the same results.

4. Space—There are many installations which now
dedicate partitions to the high performance com­
pilers such as WATFOR. In addition, there
are a large number of IBM 360's of 128K and
larger. Thus, PL/C had to run in a 128K (byte)
machine under either the OS or DOS systems.
This meant that PL/C, along with any associ­
ated symbol tables and generated object code
had to run in at most 100K bytes, with the rest
reserved for the operating system.

Clearly, all of these goals interact heavily, and deci­
sions made in attaining any one will affect some or all
of the others. For example, adding more diagnostic
checking affects both space and speed, while increasing
the subset causes the need to add more diagnostic
checking.

Several subgoals were also adopted by the design
group. These were:

5. Portability—Previous experience had shown
that isolating operating system interfaces per­
mits easy conversions of compilers to new
operating systems.

6. Open-endedness—As a Computer Science group,
it was hoped that the structure which would
emerge would be one which could be modified
and added to with relative ease. For example,
object code optimization, compile time facilities,
and language extensions have all been proposed,
and should be able to fit within the PL/C
structure.

7. Breadth of subset—Goal 3. above merely stated
the compatibility requirement. It was hoped
that PL/C would provide far more than the
FORTRAN subset of PL/I. Here the interaction
with the open-endedness goal was high. If that
one was met, then additions to the subset should
be easy.

OVERALL DESIGN AND INTERFACES

The overall design of the PL/C implementation
leaned heavily on the design of the CUPL compiler. Several members of the group had participated in
CUPL's design and implementation a few years earlier,
and CUPL actually meets all of the PL/C design goals
except for that of source language compatibility.

Experience with several error correcting compilers
and operating systems has proven the utility of an
explicit, syntactically correct representation of the
source program as a major interface between the “scan
and correct” phase and the object code generation phase
of the compilers. The presence of this intermediate
language code (I-code) has several advantages.

First, it permits the compiler to diagnose and correct
errors in near source language form. This in turn allows
error corrections to be displayed to the user in the form
of a reconstructed source statement. For example, the
following is a typical set of diagnostics for the state­
ment in error:

```
PUT LIST (A, B)
ERROR SY06 MISSING COMMA
ERROR SY07 MISSING SEMI-
COLON
```

PL/C USES PUT LIST (A, B);

The pedagogical advantages of displaying correct
statements to the user should not be underestimated.
It will often save him the trouble of looking in a manual
and trying to figure out what he should have done.

A second major advantage of I-code is that it con­
itutes a very clean interface between the major com­
piler sections. The scan phase guarantees syntactically
perfect I-code to the code generation phase. As a result,
the design and coding of these sections could proceed
in parallel. Communication between designers took
place only when changes were made in I-code or in the major interface, the symbol table. In addition, the code generation phase could be streamlined to take advantage of the fact that there would be no errors in its input.

The syntax of I-code was carefully worked out early in the design process, and was precisely documented using BNF-like notation. The independence of designers thus afforded the opportunity for each section's designer to choose and develop algorithms according to personal taste and programming style. The machine representation of I-code is a string of 16 bit items (halfwords), each representing one token of the source program. In addition, certain pointers are inserted to speed the code generation process, and to retain close links to the source language form for reconstruction purposes.

In order to further speed the code generation task, and to provide more diagnostics for the users, an intermediate pass over the I-code is made after scan, and before code generation. This pass, called "semantics" resolves problems created by the fact that PLII is a block structured language which permits explicit declarations of variable names to appear anywhere in a block. Thus, although a statement may be syntactically correct, it may be semantically incorrect, i.e., the declared attributes of a name may conflict with that name in usage. For example, if L has the attribute "label," it is semantically incorrect to write:

\[ L = L + 1; \]

One of the important design decisions was to accept and solve this semantics problem without resorting to what probably would have been a simpler solution—rule out declarations after use in the source language subset. Since the above restriction would still not remove all semantics errors, it was deemed better to accept the challenge.

In a language with as rich a variety of data types as PL/I, it is only natural to expect that the design of the symbol table will have an important effect on the overall compiler design. In PL/C, this is doubly true since the symbol table is required not only during compilation but also during execution of the object program. Its use during execution is in continuing the source level error messages. For example, a typical execution error message is:

IN STMT NNNN ERROR EXB7
ABC HAS NOT BEEN INITIALIZED.
IT IS SET TO 0.

The fact that this table had to be present in all phases also meant that space was at a premium.

Memory organization for PL/C was dictated by the space needs of the two major interfaces, I-code and the symbol table. The goals of the memory allocation were simple—make as much use of all available space as possible in all phases. Thus, it was desired to maximize the amount of core available for the runtime stack during execution. Since the symbol table and object code must also be present, this stack must clearly start at the end of these two and grow to the maximum core address allowed. A look at the code generation phase reveals that I-code, symbol table, and growing object code must all coexist. The organization presented in Figure 1 is now used.

During the syntactic phase, both the symbol table and the I-code must grow in the same direction. Thus, the fraction of available space allocated to the symbol table must be chosen carefully, so that the probability of the symbol table catching up to the start of I-code is small. In fact, in the first implementation, enough space had to remain between the completed symbol table and the beginning of I-code to hold most of the object code. This came about because our initial strategy was to generate code for an entire source block, thus skipping around in I-code if necessary. It was felt that this might save execution time, by reducing the number of base register loads necessary. A later pass at the design indicated that large programs could be more easily compiled if the code generation were done linearly, with branches inserted where necessary.

![Figure 1—Memory organization of PL/C](from the collection of the Computer History Museum (www.computerhistory.org))
due to the static structure of the program. In addition, the I-code is now moved so that it abuts the end of core just before code generation, thereby making the maximum possible room available. The cost of this move is negligible for small programs, but makes possible the compilation of some large programs which would otherwise overwrite their I-code with object code, before the I-code has been compiled.

Figure 2 shows the overall organization of PL/C into the three passes over the source code or its I-code equivalent. The first pass, syntactic analysis (SYNA), processes declare statements and performs syntactic analysis, error detection, and correction on the entire source program. SYNA produces the initial version of I-code (called betacode) and constructs the symbol table.

The second pass, semantic analysis (SEMA), performs the semantic error detection and correction on all expressions in betacode. In addition, it modifies these expressions into postfix form and rewrites them leaving a new I-code, called gamma code. This gamma code is the promised "perfect" intermediate language representation of the program which is then processed by the third pass, code generation (CGEN).

DETAILED ORGANIZATION

The PL/C design proceeded in two phases. First, the design of the major interfaces was performed. This portion of the design process alternated with the design of those algorithms in each pass which directly interacted with the interfaces. Design and redesign of interfaces and algorithms proceeded through roughly three iterations before coding began in earnest.

During this period of interface design communications among the design group were quite free, and all of the group were encouraged to participate in all of the designs under consideration. Once coding began, each designer became primarily responsible for some section of the compiler. Thus they then became rather concerned with the development of efficient local algorithms.

During the coding phase of the project, the value of a simple linear data flow (Figure 2) became apparent. Such a structure reduced the burden of communication among people dramatically. The person responsible for code generation knew (and needed to know) little about the algorithms and capabilities of the scan or semantics sections of the compiler. Had a co-routine organization been chosen, which would have involved calls between generate and semantics on an expression by expression basis, many more interfaces would have had to be designed and documented.

Figure 3 shows a more detailed breakdown of the compiler into actual modules, together with an indication of the personnel assigned to each task (letters are used instead of names). In addition to meeting the sub-goal of an isolated operating system interface and control module, the organization was consciously designed to minimize the number of person to person interfaces which had to be created. This was done by assigning the same person to routines on both sides of what would require a natural interface. Not only is documentation reduced, but the ability to change to a more efficient interface on short notice is vastly increased.

INNOVATIONS

The combined result of allowing a high degree of independence to the individual implementers, and the amount of experience present was predictable. Each of the major sections of the compiler contains some innovation or new method of doing things. In addition, those sections which were more cut and dried employ algorithms which appear to be as efficient as any now known. All parts of the compiler were implemented in IBM 360 Assembler Language, most with extensive use of the macro facilities available. This assembly language implementation allowed highly efficient code to be produced for each algorithm, and probably saved a factor of two or three in speed and space over a possible implementation in higher level language. (Another compiler being developed at the same time was forced to abandon its high level implementation language for using too much space.)

The following sections are intended to describe the most important or innovative algorithms used in the various modules of the compiler. It is our suspicion that the relatively high speed attained by PL/C is due more to the combination of good algorithms than to any specific one.

Lexical analysis

The lexical analyzer (LEXI) produces a stream of tokens from the stream of input characters forming the PL/I source program. Thus it must recognize constants, identifiers, operators, and reserved words and condense
these into single tokens. The recognition of these is straightforward. The AED\textsuperscript{9} approach of building a finite state machine is not followed closely, although the TRT instruction of the IBM 360, which in effect simulates the FSM, is heavily used. The algorithm worth mentioning is that used to hash identifiers into the symbol table. A double random algorithm, which constructs two hash functions, \( h \) and \( g \), is used. The \( i \)th look into the hash table for an item with key \( K \) is made at location \( L(i, K) \) where

\[
L(i, K) = h(K) + (i-1)g(K) \mod p,
\]

\( p \) being the size of the hash table, and \( g(K) \) is made relatively prime to \( p \). The hash functions are chosen to minimize the possibility that \( h(K_1) = h(K_2) \) and \( g(K_1) = g(K_2) \) for \( K_1 \neq K_2 \).

### Syntactic analysis

SYNA accepts the stream of tokens produced by LEXI, recognizes statement boundaries (this task was simplified through the introduction of 18 reserved words, mostly statement keywords), and produces I-code for each statement after checking for syntactic correctness. In addition, SYNA must process DECLARE statements, which accounts for about one half of SYNA's instructions. Errors are corrected, generally by either ignoring or supplying tokens to fit the required syntax. The arithmetic expression analyzer is interesting for its conciseness, and flexibility. It is basically a pushdown automaton, with a finite state machine as a component. The FSM is implemented as a branch table. LEXI supplies a class for each token, \( CL(i) \). SYNA simply issues a GOTO TAB \((S, CL(i))\), where \( S \) is the current state. The indicated routine performs some action, which may be an error correction, sets the new state of the machine, and goes back to the main loop. Parentheses and certain other delimiters simply push down the current state and start the FSM again. Thus, an 8 class by 9 state branch table summarizes the processing of expressions, by far the most exercised mechanism in SYNA.

### Semantic analysis

Semantic analysis (SEMA) scans the betacode produced by SYNA, searching for expressions. A special betacode token for expression start has already been inserted by SYNA to make the search simple. This token also includes a "semantic requirement code" which specifies the restrictions to be imposed on the expression. A typical restriction might be (SCALAR, STRING) imposed on the Boolean expression in an IF statement. SEMA proceeds to construct a parse-tree for the expression, 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, SEMA reports an error, and corrects it by modifying the parse tree. If no errors are detected, the postfixed string replaces the original infixed string in betacode, creating gammacode. Note that the postfixed form must be shorter than the infixed string, and thus only expressions change from betacode to gammacode. If errors were detected, a recursive, top-down pass is made over the corrected parse tree to get the new postfixed string. This strategy permits some highly sophisticated error correction to be used; for example, a scheme which will minimize the number of tokens changed has been proposed. Such a scheme requires that information about the entire expression be conveniently available, and the parse tree fills this need. In addition spelling correction for identifiers could be conveniently introduced at this stage.\textsuperscript{7}

---

From the collection of the Computer History Museum (www.computerhistory.org)
Code generation

The code generation pass consists of a series of statement drivers, each called into action by a particular statement body token placed in I-code. Each statement is viewed by CGEN as a string of expressions held together by a network of pointer tokens which appear in I-code. The statement drivers follow these pointer chains to generate machine language code to evaluate the imbedded expressions in an appropriate order.

Interpretive coder

An important feature of CGEN is the mechanism by which individual machine language instructions are generated. A macro language to facilitate this task has been developed. This macro language is best viewed as the machine language for a special purpose automaton for generating instructions. The language is reasonably powerful, permitting conditional branching, the setting of flags, and both machine language and interpretive language subroutine calls. In addition, the interpreter keeps near automatic track of the location of each PL/C program variable. The generation of accessing instructions is wholly a function of the interpreter.

To produce the code needed to compute FLOOR(X), where X is REAL FIXED BINARY (p, q), one writes:

\[
\begin{align*}
& \text{GLOAD G, REALI} \quad \text{generate load of X into register} \\
& \text{GRS SRA, REALI, RO, A1} \quad \text{generate SRA X, q to delete the fraction bits of X} \\
& \text{GFIN} \quad \text{finish code sequence}
\end{align*}
\]

Here, the number q has been placed in cell A1 by previous instructions. The GRS macro causes generation of any RS format instruction as follows: GRS opcode, register1, base2, displacement2. The opcode appears literally. The other three items are pointers to various locations during compile time. Thus, REALI is a pointer to REAL component of the 1st stack entry. The interpretive coder records in location REALI information allowing it to generate code to access the quantity involved. As X is moved (by the GLOAD macro) from storage to a register, the interpretive coder records its new conceptual location in the compile time stack entry REALI.

Execution

The PL/C runtime environment is mostly quite standard. Thus there is a runtime stack management routine, an I/O package which interfaces to the PL/C operating system interface package, and a collection of mathematical and string manipulation routines. Two execution time interrupts are handled in interesting manners. These are the timeout and uninitialized variable interrupts.

Traditionally, the expiration of a step time limit has been somewhat destructive of the executing program environment. The timeout can occur deep within the evaluation of an expression, and often results in un­ intelligible diagnostic output. PL/C takes the approach that timeouts should be permitted only between statements, and accomplishes it as follows:

Upon completion of each statement, control is passed by subroutine call to a small (one instruction) subroutine, which is located at the beginning of the common data area always covered by general register 12. The routine usually contains a BCR 15, R5 instruction (unconditional return), and is generally called by a BALR R5, R12. When the time out occurs, the return is replaced by a two instruction sequence which passes control to the appropriate diagnostic routine. The state of the environment is fairly clean, permitting the user to recover more information about the progress of his program than would have been possible if intra-state­ ment timeouts occurred.

The uninitialized variable interrupt is not a hardware interrupt at all. PL/C creates such an interrupt by the following device: Each variable is initialized to the constant '80000000', which is the smallest possible negative integer on the 360, and whose complement cannot be represented. PL/C generates an LPR instruction before the accessing instructions, which will cause a hardware fixed overflow interrupt to occur if the indicated value is encountered. This constant represents the integer $-2^{31}$, which requires 32 bits of precision for its representation. Since PL/I imposes a precision limit of 31, the constant $-2^{31}$ cannot arise through fixed-point binary computations without overflow. In fact, in the 360, implementation of PL/C, this constant can occur legitimately only as the representation of a character­ string. The possibility of this occurrence is felt to be so rare as not to require further checking.

Service routines

Every compiler includes a gamut of programs which are not central to the operation, but which may be called by many other parts of the compiler. The ensemble of these programs provides an internal environment for the compilation task. In PL/C, these programs include an overall control module, an error message writer (phrase expander) similar to that described for the
PL/C

PUFFT system, and the "reverse translator," which produces source images from I-code, and is used to indicate corrections which have been made by SYNA and SEMA.

Special assemblers

Several special programs have been developed to create the initial environment for compiling PL/C programs. There are over 250 keywords which must be hashed into the symbol table, for example, and several hundred error messages which must be broken down into phrases. For both of these tasks, special routines have been written which create these tables in the PL/C compile time environment, and then punch out object deck copies of core. These are then linkededit with the rest of the compiler to produce a great savings in initialization time.

DESIGN ACHIEVEMENTS

The actual performance of PL/C has been as good or better than hoped for with respect to most of the seven goals mentioned earlier. The compilation speed seems to be an order of magnitude faster than PL/I-F. On a 360/65 running under OS/MVT, compilation speeds of between 6,000 and 20,000 statements per minute have been observed, depending upon the programs used. A moderate sized sample of random student jobs, with a substantial number of errors yielded the formula compile time = .04 + .007s seconds, where s is the number of statements. This gives an impressive 8500 statements/minute compile speed for programs in which the error correction mechanisms are heavily exercised.

The compiler correctly repairs a useful fraction of the punctuation errors that are endemic with neophytes and which even afflict those 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 author intended, it is often very successful in prolonging the life of a program sufficiently to yield additional useful diagnostic information. Eight years of experience with this approach have clearly demonstrated a significant reduction in the number of job submissions required to obtain a successful execution of a program. In its current form PL/C probably offers more ambitious and sophisticated diagnostic assistance than any other compiler for any language in general use, and this aspect of the system is quite open ended, with further developments anticipated.

The breadth of subset which has been achieved has surprised even the authors. The only significant omissions from the current version are the compile time facilities, multitasking, I/O other than stream sequential, and list processing. The achievement of open-endedness makes the addition of these possible, and work is proceeding on the list processing area now.

The two factors which were most underestimated by the group were space and implementation time. The goal of running in 100K was met through the use of overlays. The memory organization proved robust enough to handle the overlays without requiring any redesign of modules. PL/C will compile and execute small programs in as little as 90K, and can handle about 250 statement programs in 100K. A completely core-resident version (no overlays) is obtained by changing one linkedit control card, and runs in 100K.

The original implementation schedule called for a rough version ready for testing in 6 months, and a production version at the end of 9 months. Outside sources who had implemented PL/I compilers estimated 18 months as a more realistic target for a production version. In fact, the major testing with live student audiences began after about 13 months, and production versions were shipped in 17 months. Thus, the old rule of thumb about multiplying any estimates of programming time by two is again validated.

CONCLUSIONS

A high-performance compiler has been designed and implemented for a subset of PL/I using the strategies outlined in this paper. In retrospect, one of the few major changes in this strategy would have been the early implementation of debugging aids for the compiler writers. While we still feel strongly that assembler language was a wise choice, formatted dumps of I-code, symbol table, and object code might have significantly reduced the debugging time, and helped to meet implementation deadlines.

In our judgment the key factors in the success of the PL/C project were:

1. Overall design which provided clean, clearly specified interfaces between the major compiler phases.
2. Elimination of many internal communication problems through the assignment of personnel. In particular, placing the same person on both sides of an interface is seen as a major factor.
3. Use of new techniques in analysis and code generation.
4. Refinement of "standard" techniques for high-performance compilers. These include limiting the size of programs so that auxiliary storage is
not used during the compilation process, and use of a single pass over the actual source to minimize I/O time.

Confirmation of the fact that high-performance compilers are here to stay has come recently in the form of the announcement of the IBM Checkout PL/I, which is actually an interpreter. It appears from the announcement that PL/C still enjoys a three or four to one performance advantage, but the days of having all users pay the high penalties inherent in using production compilers for debugging are numbered.

ACKNOWLEDGMENTS

The PL/C design is directed by Professor Richard Conway and includes the authors, T. Wilcox, and M. Zelkowitz. Additional design and programming assistance has been provided by M. Bodenstein, H. Cabassa, P. Dormont, R. Fisher, T. Kahne, R. Holt, S. Libberger, N. Weiderman, K. Wong and W. Worley.

REFERENCES

1 R W CONWAY W L MAXWELL
CORC—The Cornell computing language
Comm ACM 6 Sept 1966 pp 317-321

2 R W CONWAY W L MAXWELL
CUPL—An approach to introductory computing instruction
Technical Report 68-2 Dept of Computer Science
Cornell University

3 R W CONWAY DELFASSE MAXWELL WALKER
CLP—The Cornell list processor
Comm ACM 8 April 1965 pp 215-216

4 IBM 360 assembler language
IBM manual no C28-6514

5 IBM 360 PL/I (F) reference manual
IBM manual no GC28-8201-2

6 IBM scientific computing report
Fall 1970

7 H L MORGAN
Spelling correction in systems programs

8 S ROSEN R SPURGEON J DONNELLY
PUFFT—The Purdue University fast FORTRAN translator
Comm ACM 8 Nov 1965 pp 661-666

9 D T ROSS JOHNSON PORTER ACKLEY
Automatic generation of efficient lexical processors using finite-state techniques
Comm ACM 11 Dec 1968 pp 805-814

10 P SHANTZ GERMAN MITCHELL SHIRLEY ZARNKE
WATFOR—The University of Waterloo FORTRAN IV compiler
Comm ACM 10 Jan 1967 pp 41-44

11 R A WAGNER
Common phrases and minimum space text storage
Technical Report 70-74 Dept of Computer Science
Cornell University