# PL/CS -- A DISCIPLINED SUBSET OF PL/I

R. Conway and R. Constable

TR 76-293

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

# PL/CS -- A Disciplined Subset of PL/I

Richard Conway and Robert Constable
Department of Computer Science
Cornell University

PL/CS is an instructional dialect of PL/I. It is defined by selecting features of PL/I, and then restricting the manner in which those features can be used. The implementation is an error-repairing compiler based on PL/C, in which error-repair is carried to the point where a user is deliberately encouraged to use an abbreviated entry syntax and rely on the compiler to expand this to produce a complete, PL/I-compatible program. PL/CS also includes an "assertion" facility that can be used either as a conventional diagnostic tool, or as the basis of a formal proof of correctness in a logical system provided within the language.

## 1. Introduction

The PL/CS system was designed to meet two major objectives. We sought an instructional language that would support the contemporary view of programming, and we sought a language in which we could experiment with teaching new approaches to the development of correct program. Such a language should emphasize clarity and correctness, and should require many of the practices that the student programmer must currently be persuaded to elect. Among the experimental facilities to be supported by this language would be extensive automatic error-repair, several types of user-controlled diagnostic tools (including "assertions"), a formal correctness verifier, and format-controlling "statement comments".

Our choice of PL/I as a host language was quite obvious, considering our special circumstances. Cornell's PL/C project offered an existing implementation group, as well as access to more than 200 PL/C installations worldwide, which meant that the result could be more than just another parochial teaching language. However, quite apart from this practical advantage, PL/I can be plausibly defended as representing the best current compromise between wide-spread use and contemporary form. That is, the few languages more widely used than PL/I are considerably less contemporary in form, and more modern languages are not yet as widely used.

We posited strict upward-compatibility with PL/I as a basic design constraint, and steadfastly resisted frequent temptations to "improve" upon the host language in ways that had little to do with our principal objectives. For example, the temptation to substitute some unambiguous assignment operator for the PL/I "=" was almost irresistable, but we feared that incompatible changes such as this would erode the "adopability" of our dialect. We wanted an instructor proposing use of PL/CS to be able to honestly claim that he would still be teaching a standard, widely-used language -- albeit with some restrictions imposed upon the programmer, and a few optional facilities added.

Of course, most of the ideas that PL/CS represents are not peculiar to PL/I. PL/I (in the form of PL/C) represented a convenience in implementation and an opportunity for distribution but the ideas would be equally applicable to, say, ALGOL or PASCAL and in fact would be even more attractive with such a host language.

## 2. The PL/CS Language

PL/CS is a spare but non-trivial language that is adequate for at least the first two semesters of practical programming instruction, as well as for a course in the mathematical aspects of programming. It is derived from PL/I primarily by two kinds of restrictions -- many PL/I features are omitted, and certain limitations are imposed on the usage of the features that are included. Compatibility with PL/I is maintained since the usage required in PL/CS is allowed in PL/I.

The PL/CS subset is similar to several others with respect to the statement types included. For example, it is almost identical to the PL/I subset embedded in King's EFFIGY system (ref 14). The usage restrictions are more unusual, and probably more important. For example, rather than eliminate the GOTO, as for example in the SP/k system (ref 13), the PL/CS GOTO is rendered harmless by restricting it to use as an "exit". So restricted, it cannot be seriously abused, and its presence is very helpful in a language that otherwise lacks the exit, case and repeat-until constructs (ref 15).

The most important restrictions are those that render PL/CS functions completely free of side-effects. A PL/CS function returns a scalar value and has no other effect (except the consumption of time). This does not preclude any significant programming tactic, since the non-function procedure is still available, but it does mean that all PL/CS expressions are more understandable since any function invocations they contain can neither assign values nor alter the path of control. Since expressions appear in many places in PL/I syntax this restriction alone permits a substantial simplification in the semantic specification of various statement types.

Similarly, there are restrictions on the iterative-DO loop that prevent the variables of the control phrase from being altered in the body of the loop. This simplifies the understanding of this loop, rule out various "trick" constructions, and guarantees termination, but does not preclude any useful tactic since the general DO WHILE loop is still available.

Some restrictions simply improve syntactic consistency. For example, all PL/CS conditions are enclosed in parentheses. This has the dual benefits of clarifying the mysterious "A = B = C" statement and of eliminating the PL/I syntactic discrepancy between IF and WHILE conditions.

The process of arriving at these restrictions was itself interesting. We approached the task from two very different points-of-view. On one hand we sought to enforce a particular programming style, and to circumvent peculiarities in PL/I that consume much lecture time and contribute to many student errors. On the other hand, we sought to eliminate the features and usages that make it almost impossible to give a simple semantic description of PL/I. It was initially surprising, and ultimately very reassuring, to discover that the remedies were the same and that the two objectives were never in conflict. The final design of PL/CS is a consensus of these views and not a compromise. The success from the semantic point-of-view is apparent in the brevity of the complete formal description of PL/CS (ref 3).

A formal syntax specification of PL/CS has been given (ref 17) but for present purposes an informal comparison to PL/I is more appropriate.

## 2.1 Statement Types

A complete list of the statement types included in PL/CS is given below. Except as noted, the syntax and semantics of these statements correspond to those of PL/I. Two incompatible features are included in this list -- the ASSERT statement and the READONLY attribute. Both of these are essentially diagnostic features and can be deleted from a correct program without altering the results of execution. In both cases compatibility with PL/I can be preserved even when using these features by sheltering them within "convertible comments" (see reference 6, section V.1.1). Input sheltered in this way can be optionally considered source text or comment text, depending upon whether the PL/CS or PL/I processor is being used.

**ASSERT**

1. Initial limited form is: ASSERT condition [quantifiers]
2. The form of a quantifier is either:
   FOR ALL index-var = expr1 TO expr2 BY expr3
   or FOR SOME index-var = expr1 TO expr2 BY expr3
3. If satisfied, ASSERT has no side effects; if not satisfied, it generates appropriate printed output.
4. See section 4 for discussion of the semantics of this statement, and the expanded forms for formal verification.

**Assignment**

1. No multiple left-sides.
2. Array assignment is allowed, but not array operations.

**CALL**

1. Neither labels nor entry-names allowed as arguments.
2. Types of arguments and parameters must match exactly.

**DECLARE**

1. Explicit declaration of variables and parameters is required, and must be positioned at beginning of procedure.
2. Variables and parameters must be given in separate declarations.
3. Required form is: DECLARE (identifiers) attributes;
4. Array dimensions (including both lower and upper bounds) are given in the identifier list, as in PL/I.
5. Attributes for variables:
   a. one of the following is required: FLOAT, FIXED, CHARACTER(expr) VARYING, BIT(1)
   b. optional: INITIAL(list), STATIC, EXTERNAL

6. Attributes for parameters:
   a. one of the following is required: FLOAT, FIXED, CHARACTER(*) VARYING, BIT(*)
   b. optional: READONLY
7. FLOAT implies FLOAT DECIMAL(16), FIXED implies FIXED DECIMAL(15,0).

| | |
|---|---|
| DO | 1. Compound statement as in PL/I.<br>2. DO WHILE does not require parentheses (since condition is already parenthesized).<br>3. DO index-var = expr1 TO expr2 BY expr3<br>a. All three expressions are required, with TO-BY in order shown. No other forms of list allowed.<br>b. index-var and all variables in expr1,2,3 are readonly within body of the loop -- they cannot be object of assignment or argument of CALL.<br>c. Value of index-var after normal termination is "uninitialized".<br>d. expr3 must have non-zero value.<br>4. Each loop must have a loop-name label. |
| END | 1. All ENDs are explicit, and loop-names and procedure entry-names must be given after END. |
| GET | 1. Only the LIST, DATA and EDIT options are allowed. No DO iteration in lists is allowed. |
| GO TO | 1. Only forward reference is allowed.<br>2. Target must be a labelled null statement.<br>3. Target can be in a loop or compound statement only if all references to that target are also in that loop or compound statement.<br>4. Target can be in a THEN or ELSE-unit only if all references to that target are also in that unit. |
| IF | 1. A null THEN or ELSE unit is not allowed.<br>2. While the IF syntax does not require the condition to be parenthesized, all PL/CS conditions are already parenthesized. |
| Null | 1. Used only as a target for a GO TO.<br>2. Must have a single label. |
| ON | 1. Required form is: ON ENDFILE GO TO target-name;<br>2. At most one ON per procedure.<br>3. ON must be positioned immediately after declarations. |

PROCEDURE
1. Main procedure must be given first.
2. All procedures are external.
3. All parameters and variables must be explicitly declared at the beginning of the procedure.
4. Types of arguments and parameters must match exactly; lengths and dimensions are dynamic as in PL/I.
5. Parameters may be declared READONLY, in which case they cannot be the target of assignment (including GET and DO) or argument of a CALL.
6. Variables can be STATIC or EXTERNAL.
7. Function procedures can have no side-effects:
   a. All parameters must be READONLY.
   b. Variables can be neither STATIC nor EXTERNAL.
   c. Function body cannot contain CALL, GET or PUT statements
8. Explicit RETURN is required.

PUT
1. Only the SKIP, LIST, DATA and EDIT options are allowed. No DO iteration in lists is allowed.

RETURN
1. Explicit RETURN required before END of procedure.

## 2.2 Other Language Elements

Prefixes
1. No condition prefixes are allowed.
2. Single entry-name prefix is allowed on procedures.
3. Single loop-name prefix is required on iterative DOs.
4. Single target-name prefix is required on null statement.
5. No other statement type is allowed a label prefix.

Macros
1. The PL/C macro facility (see reference 6, section V.11) is included (rather than the PL/I compile-time facilities).

Comments
1. PL/C convertible comments are included (see reference f, section ).
2. "Statement comments" are provided (ref 8). These comments (distinguished with a "*" as their first character) act as a compound statement in the presentation of output, and permit higher-level statements to be used in program documentation.

Trace
1. This is a control-card option, requiring no insertions in the source program.
2. Option can be selectively applied to GET, PUT, assignment, DO, IF, GO TO, CALL, PROCEDURE and null statements, to automatically generate run-time status messages whenever a statement of the specified type is encountered in execution.
3. Option can also be selectively applied with respect to the nesting depth .
4. Trace can be invoked, revised or cancelled at any point in the source program by inserting a *OPTIONS control card.

### 3. Display vs Entry Forms

The PL/CS language has a high degree of redundancy, arising primarily in the requirement for PL/I compatibility. For example, the main procedure of a PL/CS program is always first, but compatibility requires that it be designated "OPTIONS(MAIN)" anyway. As another example, PL/CS has only varying strings, but compatibility requires that every string must be explicitly declared "VARYING". These extra keywords would certainly be an annoyance -- except that we don't expect the user to supply much of this redundant verbiage anyway. The language described above is the full, compatible form we call the "display language". This is the language the compiler <u>produces</u> -- not the language that it expects to receive. The user presents a program to the compiler in an abbreviated "entry" form. In fact, due to the error-tolerance of the compiler, there are many different entry forms, all equivalent in the sense that they will evoke the same display form response from the compiler. Consider the following four examples:

1. To generate the following full display-form declaration:

   ```
   DECLARE (A, B(1:10), C) CHARACTER(50) VARYING;
   ```

   a reasonable, readable entry form would be:

   ```
   (A, B(10), C) CHAR(50)
   ```

   Even more extreme abbreviation is possible (the commas and parentheses could be omitted) but would be less readable and is not recommended.

2. To generate the following declaration of a string parameter:

   ```
   DECLARE (STRG) CHARACTER(*) VARYING;
   ```

   an adequate entry form would be simply:

   ```
   STRG CHAR
   ```

3. The PL/CS syntax says that all ENDs are explicit, but in practice the user is not expected to supply them. As in PL/I an identified END may be given to close a number of blocks or loops, but unlike PL/I all of the generated ENDs will be printed, numbered and identified (eliminating a major source of student error).
4. PL/CS requires a label on every DO-loop, but does not expect the user to always supply it. When omitted by the user, the compiler generates a label from the statement number. This label is also used to identify the appropriate END, and is referenced in the automatic statement execution count in the post-mortem dump (reference 6, section I.7.2.4). Hence the user has full benefit of loop identification, without necessarily doing any of the requisite work himself.

The "corrections" required to expand from entry to display form elicit "warning" rather than "error" messages from the compiler, and the default message level will not print warning messages. Moreover the entry text and the full display form of the program are printed separately, rather than interleaved as, for example, in PL/C. The display form is automatically indented ("prettyprinted") to show the actual block and nesting structure of the program, regardless of how the entry lines were positioned. Collectively, these characteristics focus attention on the <u>output</u> of the compiler, rather than the input. The program which is executed has been produced by the compiler, based in a deterministic manner upon input received, but not identical to that input.

This deemphasis of the input text is most appropriate when PL/CS is used in interactive mode, with input being supplied from a terminal. In this mode the input line only exists momentarily. The expanded display form used in the program is also the line saved for subsequent editing and re-use. In batch mode, with card input, it may be difficult to update a card deck from a source listing that looks quite different. It may turn out to be simpler for neophytes to use an entry form on cards that is equivalent to the display form -- but this remains to be seen.

The expansion from entry to display form can be viewed as just an extension of an error-repairing philosophy -- but it is really more significant than that. Previously, the user was expected to provide a complete and correct program. If he occasionally and inadvertently failed, certain compilers would attempt to assist him but the responsibility was entirely the programmer's. Now, in PL/CS, the user is expected to deliberately depart from the required language and take advantage of the compiler's corrective ability. Our intention is to shift as much as possible of the mechanical aspects of programming to the compiler, and to minimize the clerical burden of entering input -- but the overall effect on the user's habits may not be salutary.

The PL/CS compiler can also be viewed as a tutoring device. Receiving input in a casually punctuated, intuitive programming language, it displays the proper, full, PL/I equivalent construction. The display form makes explicit the tacit default assumptions of the entry form -- lower bounds of arrays, "BY 1" increments, VARYING strings, etc.

## 4. Verification of Correctness

A significant experimental facility of PL/CS is a provision by which the programmer can insert statements that describe the intended state of the computation at the point of insertion. These statements are, in a sense, just comments as far as the underlying program is concerned, since they do not affect the state or course of computation. But they are comments that imply certain actions on the part of the system, and we have elected to use a syntax that treats them as source statements (although they can be enclosed in convertible comments to preserve absolute compatibility with PL/I).

There are two types of assertions in PL/CS. The first is a statement of presumed fact, to be monitored automatically during execution of the program. That is, the programmer asserts that some condition is supposed to be true whenever this point is reached in execution. The system automatically tests the condition at that point and reports any time it is not true. The second type of assertion is essentially an explanation of the program text -- a line in a proof that the program is correct. Such a proof can be verified automatically, during compilation, confirming the correctness of the program without requiring its execution.

The first type of assertion is relatively straightforward in implementation, and is experimental only in the sense that it is not clear just how useful such assertions really are, and how extensively programmers would really use them. The proof-assertions are tentative in form and it is not yet clear how powerful a verification system can be provided.

We are proceeding in two stages. The first involves the monitored assertions, and is an integral part of PL/CS. The second involves proof-assertions, and is being developed as a separate, parallel dialect called PL/CV. This dialect is identical to PL/CS except for expanded forms of assertions. Initially, a separate processor for PL/CV permits greater flexibility in experimenting with different forms of proof-assertions, and different verification algorithms. Eventually, the proof-assertions will be incorporated into PL/CS and the verification processor will be added to the PL/CS system.

The following sections give a brief explanation of both types of assertions. The proof-assertions are described fully in reference 4.

### 4.1 Monitored Assertions

PL/CS assertions are similar to those described in references 1 and 16. The simplest form is:

```
ASSERT condition;
```

This is operationally equivalent to:

```
IF ¬condition THEN DO;
    PUT SKIP(2) LIST('ASSERTION nnnn NOT SATISFIED');
    PUT SKIP DATA(list of variables in condition);
    END;
```

where "nnnn" is the statement number of the assertion in the source listing. Such assertions obviously add no new capability to the language since the programmer could write the equivalent statements, but the increased effort makes it unlikely that this would often be done. Moreover, the assertion has the convenient property of being able to be activated and deactivated simply by means of a control card option. (When deactivated, assertions are simply comments; when activated they act as the equivalent statements shown above.)

The assertion can also have one or more levels of universal or existential quantification. The universal form is:

```
ASSERT condition FOR ALL index-var = expr1 TO expr2 BY expr3;
```

This is equivalent to:

```
BEGIN;
   DECLARE (index-var) FIXED;
   DO index-var = expr1 TO expr2 BY expr3;
      IF ¬ condition THEN DO;
         PUT SKIP(2) LIST('ASSERTION nnnn NOT SATISFIED FOR');
         PUT DATA(index-var);
         END;
      END;
   END;
```

The existential form is:

```
ASSERT condition FOR SOME index-var = expr1 TO expr2 BY expr3;
```

This is equivalent to:

```
BEGIN;
   DECLARE (index-var) FIXED;
   DO index-var = expr1 TO expr2 BY expr3;
      IF condition THEN GO TO $nnnn;
      END;
   PUT SKIP(2) LIST('ASSERTION nnnn NOT SATISFIED');
   $nnnn:;
   END;
```

Note that assertions have no side-effects when satisfied (except for the consumption of time), and when not satisfied their only effect is to print a diagnostic message.

A simple example will illustrate both the power and the limitations of the monitored assertion. Consider the following program segment. The task is defined by a "statement comment", implemented by some sequence of statements, and then concluded with assertions that check to see whether the task has been accomplished:

```
          /** SET M TO MAX OF X(J:K) */
                 .
                 .
                 .
              ASSERT (M >= X(I)) FOR ALL I = J TO K BY 1;
              ASSERT (M = X(I)) FOR SOME I = J TO K BY 1;
              /** SET */
```

Presumably one would try to teach programmers to write such assertions at the time they write the program segment and make them an integral and permanent part of the program. The assertions can be selectively activated for testing, and are always available for renewed testing.

However, note that in general such assertions establish a necessary but not sufficient condition for correctness of the segment. For example, the following statements, obviously incorrect, would satisfy the assertions given above:

```
          /** SET M TO MAX OF X(J:K) */
              MAXLOOP: DO I = J TO K BY 1;
                  X(I) = X(J);
                  END MAXLOOP;
              M = X(J);
              ...
```

Note also that at best "partial correctness" is sought since there is no assurance that the segment will terminate and the assertion will ever be encountered.

The most important limitation is the characteristic that only specific executions are monitored. While we would like to know that a program is correct for all possible executions, the monitored assertions can only imply this from a sample of trial executions. This is the usual problem of seeking correctness by testing -- one can only fail to prove incorrectness. Even if we were never to deactivate the assertions we would never know that the program segment is correct -- only that certain types of incorrectness would be detected if they occurred.

### 4.2 Proof Assertions

The second form of assertion seeks to overcome the limitations of run-time correctness checks, and establish the correctness of a program for all possible executions. This is done by analyzing the program text. There are two alternative approaches to this task. The first seeks to prove properties of the program directly, reconciling this with a formal statement of requirements. The second requires that the programmer supply the correctness proof along with the program -- with the validity of the proof being automatically checked. We are employing the latter approach.

The following example of a PL/CV proof is a simple program to execute the Euclidean division algorithm for non-negative integers. The program without proof-assertions is short, but not trivial. It has become a canonical example in papers on programming methodology and program verification, so it provides a good basis on which to compare the PL/CV approach to that of others.

The PL/CV language differs from PL/CS only in the form of assertions, and in the fact that any PL/CV statement can have a label (so that proof-assertions can refer to particular statements). PL/CV assertions begin with a keyword (ASSERT, ASSUME or CONCLUDE) and generally include a "BY phrase" in which the justification for the assertion is given. The justification can specify a proof rule ("AND-introduction", for example), a property of a program statement, or reference to another statement or assertion. Reference can either be by name (label) or by relative position ("-1" means "the preceding statement").

```
/* EUCLIDEAN DIVISION ALGORITHM */
  DIVIDE: PROCEDURE(A, B, Q, R);
    DECLARE (A, B) FIXED READONLY;
    A1: ASSUME A >= 0, B > 0;
    CONCLUDE (A = B*Q+R & 0 <= R & R < B);
    DECLARE (Q, R) FIXED;
    DECLARE (W) FIXED;
    RL: R = A;
    WL: W = B;
    QL: Q = 0;
    /** FIND A MULTIPLE OF B GREATER THAN A */
      INC: DO WHILE (W <= R);
        W = W * 2;
        END INC;
      ASSERT (W > R) BY WHILE ASSUME;
      ASSERT (W > B) BY EQ, WL, -1;
      ASSERT (A = W*Q+R) BY ARITH, RL, QL;
      ASSERT (0 <= R & R < W) BY ANDIN, A1, -3;
      /** FIND */
    /** SUBTRACT MULTIPLES OF B FROM A */
      LOOP: DO WHILE (W ¬= B);
        A2: ASSUME A = W*Q+R, 0 <= R, R < W;
        ASSERT (A=(W/2)*(2*Q)+R & 0 <= R & R < 2*(W/2))
              BY ARITH, -1;
        Q = 2 * Q;
        W = W/2;
        INVAR1: ASSERT (A = W*Q+R & 0 <+ R & R < 2*W)
              BY ASSIGN, -1, -2, -3;
        IF (W <= R)
          THEN DO;
            ASSERT (W <= R & R < 2*W) BY ANDIN, -2, -3;
            ASSERT (A = W*Q+1 + R - W) BY ARITH, INVAR1;
            R = R - W;
            Q = Q + 1;
            INVAR2: ASSERT (A = W*Q+R & 0 <= R & R < W)
                  BY ASSIGN, -1, -2, -3, -4;
            END;
          ELSE;
            INVAR3: ASSERT (A = W*Q+R & 0 ¬= R & R < W)
                  BY ELSE, INVAR1;
        END LOOP;
      ASSERT (W = B) BY WHILE ASSUME;
      /** SUBTRACT */
    END DIVIDE;
  ASSERT  (A = B*Q+R & 0 <= R & R < B) BY INVAR2, A2;
```

Providing such proof-assertions is obviously a formidable task, at least in its present form, but when the verifier finally accepts the program "total correctness" has been proved. That is, the program correctly completes its task, and halts normally, for every possible execution.

Even though the proved program text is longer and more complex than the plain program, a potential reader's effort in understanding what it does can be confined to the ASSUME and CONCLUDE statements in the program heading. The PL/CV logic provides the full first-order predicate calculus (ref 11) -- in fact, even more than this -- as part of its assertion language. It also provides a precise notion of formal program correctness proof and a full set of rules for reasoning about arithmetic and string-processing programs (ref 4).

Whatever the eventual verdict on formal verification as a tool for programmers, it is likely that it will in any event be useful as a pedagogical tool for computer scientists, who should achieve a deep understanding of the meaning of correctness and proof. An informal form of verification (ref 10) is already becoming important in this role. The verification facilities in PL/CV also offer new approaches to teaching some basic ideas about programming and about computing theory. For example, the PL/CV axioms for the "DO I = 0 TO E" construct will firmly establish the relationship between program iteration and mathematical induction.

## 5. Implementation

As noted earlier, the prototype system consists of two separate processors -- a compiler for PL/CS and a verifier for PL/CV. Assuming all goes well, eventually the verifier will be incorporated into the compiler so that verification will simply be a compile-time option. At present it requires a separate run. Here we are primarily concerned with the compiler; the verifier is described elsewhere (ref 4 ).

The prototype PL/CS compiler is just a modification of the PL/C compiler. In essence, the "front-end" of the PL/C compiler is replaced by a special PL/CS front-end that processes PL/CS source text into the standard PL/C internal form. The PL/C code generation phase and the run-time support system are used intact, and are unaware that a modified source language is being employed. Actually, since PL/CS is an interactive system, it is based upon the interactive version of PL/C (ref 7, Appendix C) rather than the standard batch version.

The overall organization of PL/C is shown in figure 1 (ref 5). For PL/CS the modules to the left of the dotted line interface in figure 1 are replaced by the PL/CS syntactic analyzer outlined in figure 2. The replacement is transparent since the PL/CS analyzer, although more complex internally, still represents a single phase to PL/C control, and still produces the standard internal program form (called "beta-code") and the standard symbol table.

We could have modeled the PL/CS analyzer after the PL/C version, but we elected to explore a new strategy for error-repair, hence the internal structure is very different. PL/C error-repair is based on

20

Figure 1 PL/C System Organization

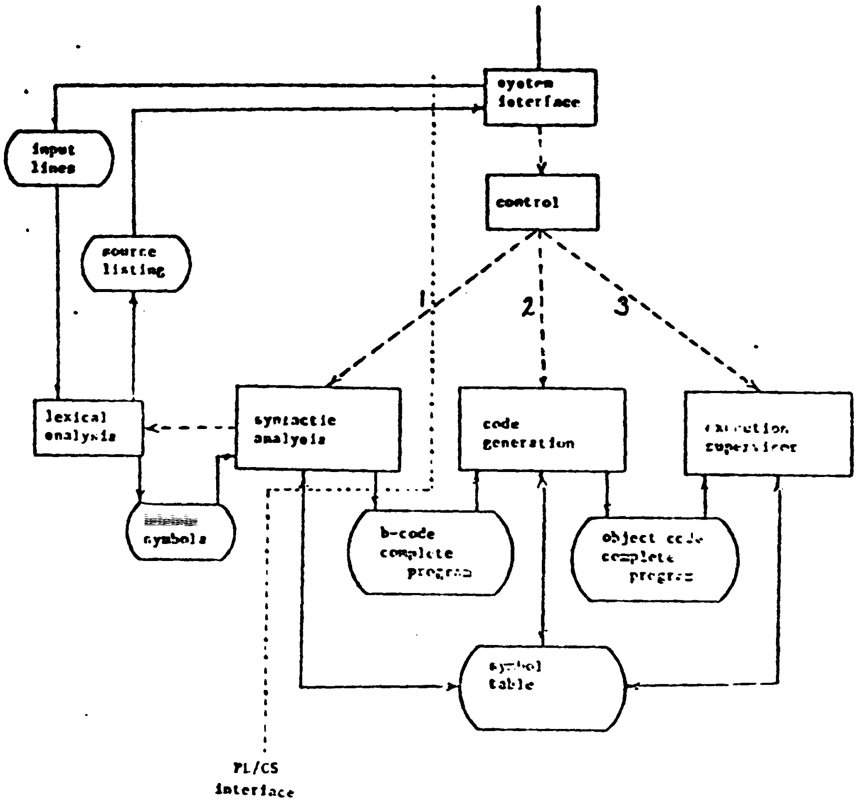
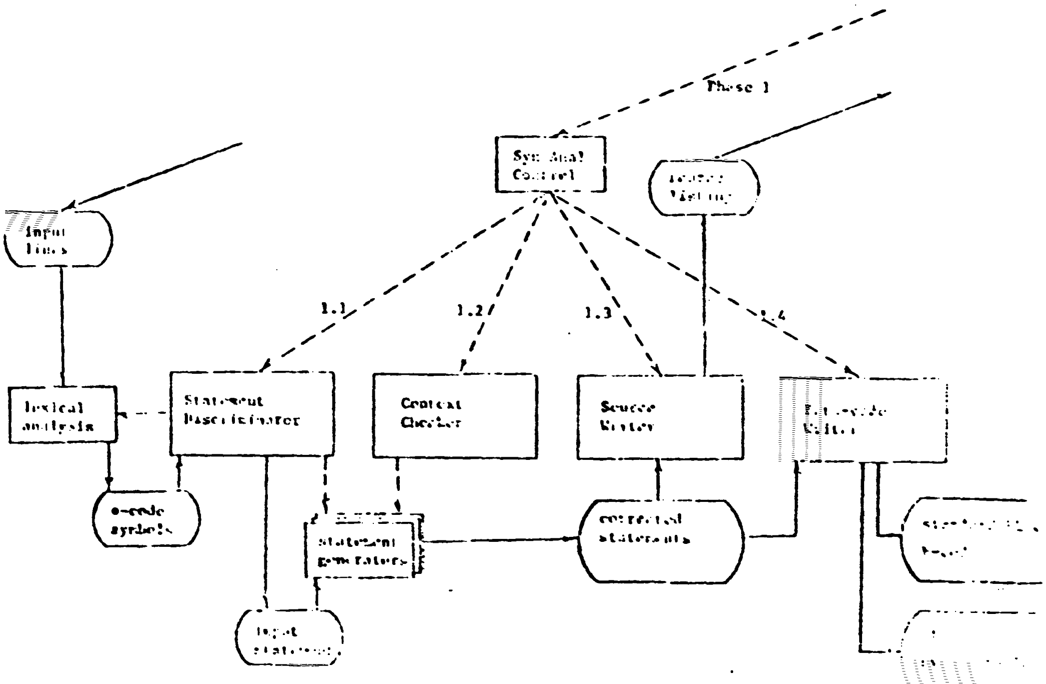



Figure 2 PL/CS Syntactic Analysis



a single left-to-right scan of input symbols, with very limited look-ahead and back-up (ref 5). In general, this appears to be useful for simple local errors, but its left-to-right bias is apparent and it is not difficult to exhibit examples where this near-sighted strategy produces unreasonable repairs (ref 12). Since PL/C is already a third-generation implementation of this single-pass strategy it seemed unlikely that further refinement would be highly productive.

Our strategy in PL/CS is essentially to separate the process of deciding <u>what kind of statement</u> is being presented, from the process of converting that statement into valid syntactic form (ref 9). A first pass over the input symbols is made by the Statement Discriminator (figure 2). This notes the presence and positions of all keywords (which are reserved) and notes certain characteristics of the symbol strings between keywords, but makes no attempt to judge the correctness of the input or to modify it. When "enough" input has been scanned in this way, the Discriminator "decides upon" a particular statement type and calls the appropriate Statement Generator. The Generator expands and corrects the input string into a statement of the required type. It has the advantage of knowing in advance what keywords are present, where they are positioned, as well as several other properties of the input string. With this strategy embedded keywords are just as influential as opening keywords. For example, "FIXED" is just as indicative of a declaration as "DECLARE", and in fact is far more informative.

The Context Checker then determines whether the statement received is a valid successor to the program statements that have been accepted up to that point. If not, it calls various Statement Generators recursively to supply statements until a suitable context has been produced to receive the new input statement. Hence a segment of one or more correct, display-form statements is produced from each entry-form input phrase.

At this point the display-form statements can be added to the source listing and delivered to the text-editor (not shown) to be saved for future use. It is also at this point that the proof verifier will eventually be incorporated into the compiler.

The final sub-phase of the analyzer is the Beta-code Writer which transforms the corrected, expanded program into PL/C internal form and completes the preparation of the symbol table. This is largely straightforward, except for PL/CS features that do not exist in PL/C. For example, monitored assertions in the display program are replaced by the equivalent PL/C statements (see section 4.1 in beta-code. These statements are numbered to correspond to the assertion in the source program and are entirely invisible to the programmer. Similarly, the PL/CS trace feature is implemented by the generation of invisible PL/C source statements.

It is not yet clear how much this multi-pass analyzer will cost in terms of compilation speed as compared to the single-pass PL/C version. Much of the prototype PL/CS analyzer is written in PL/I (optimizer) to facilitate modification of the error-repair algorithms, and the inefficiencies of this hybrid system confound the difference in strategy. It is little more than a guess at this point, but we would expect an eventual production version of PL/CS (excluding verification) to compile at not less than half the speed of PL/C. Run-time efficiency would, of course, be equivalent unless the program is extensively monitored with assertions.

There is also an interesting fall-back position. If it should turn out that the PL/CS dialect itself is attractive, but that the enhanced error-repair is of marginal utility and formal verification of real programs by real students is impractical, we could always produce a very efficient processor just by modifying PL/C. A few additions to the syntactic analyzer would be required to restrict usage, but the modification would consist primarily of pruning sections unneeded by the simpler language. We estimate that the result would be half the size of PL/C (perhaps only 60K bytes) and significantly faster in compilation. This is of course well within the capability of present mini-computer systems and we find the prospect very interesting.

2

## 6. Conclusions

Although prototype versions of both the Compiler and the Verifier exist, they are in relatively inefficient hybrid form, and do not include final algorithms for either error repair or proof verification. Moreover, the system has not as yet been tested in a realistic classroom environment.

The most interesting aspects of PL/CS are almost all behavioral - not what the system is or does, but what the effect will be on the behavior of people who learn to program using it. Only with extensive use, including use at institutions other than Cornell, will questions like the following be answered:

1. What is the pedagogical value of mandating style and structure, rather than making these issues conscious and deliberate effort by the programmer?
2. Can the time freed from explaining language peculiarities be effectively employed to better advantage?
3. Is the improved error-repair capability of a multi-pass analyzer relative to a single-pass analyzer of practical value?
4. Does the distinction between entry and display forms help to liberate the programmer from the tyranny of keyboard input -- or just confuse the whole process?
5. Are monitored assertions and the PL/CS trace feature realistic diagnostic facilities that will be effectively used by ordinary programmers?
6. What is the practicality of formal verification for non-trivial programs by ordinary student programmers?

PL/CS represents a significant extension of the PL/C view of programming instruction, and the answers to these questions will indicate which directions of the extension were useful and which are not.

We expect it will be several years before the system is fully implemented, tuned, and refined, and instructional techniques have evolved to exploit its characteristics. Only then can these questions be answered with confidence, if not objectivity. Nevertheless, we have become increasingly enthusiastic about the general directions that PL/CS represents.


## Acknowledgements

The authors are serving as spokesmen for a large design and implementation effort. A. Demers, J. Donahue, D. Gries, J. McGraw, M. O'Donnell and R. Teitelbaum participated in the language design. The Compiler implementation group consists of P. Bishop, M. Bodenstein, M. Hartstein, P. Tretor and S. Worona. The Verifier implementation group consists of C. Hauser and S. Johnson. The work has been in part supported by the National Science Foundation under Grant DCR-74-14701.


## References


1. ALGOL-W Reference Manual, Dept of Computer Science, Stanford University, Feb. 1972

2. Constable, R. L., "Language Features that Support Program Verification", Technical Report 76-276, Dept of Computer Science, Cornell University

3. Constable, R. L. and J. E. Donahue, "An Elementary Formal Semantics for the Programming Language PL/CS", Technical Report 76-271, Dept of Computer Science, Cornell University

4. Constable, R. L. and M. O'Donnell, "A Logic for Reasoning about Programs", Technical Report 76- , Dept of Computer Science, Cornell University

5. Conway, R. W. and T. R. Wilcox, "Design and Implementation of a Diagnostic Compiler for PL/I", Communications of ACM, 16,3 (March 1973) 169-179

6. Conway, R. W. and D. Gries, Introduction to Programming, 2nd ed. Winthrop Publishing Co., 1975

7. Conway, R. W. and D. Gries, Primer on Structured Programming, Winthrop Publishing Co., 1976

8. Conway, R. W. and D. Gries, "The Use of Comments in Programming", Technical Report 76- , Dept of Computer Science, Cornell University

9. Conway, R. W. and T. Bishop, "Two-Pass Syntactic Error-Repair", Technical Report, Dept of Computer Science, Cornell University

10. Dahl, O. J., E. W. Dijkstra and C. A. R. Hoare, Structured Programming, Academic Press, 1972

11. Enderton, H. B., A Mathematical Introduction to Logic, Academic Press, 1972

12. Graham, S. L. and S. P. Rhodes, "Practical Syntactic Error Recovery", Communications of ACM, 18,11 (Nov 1975)

13. Holt, R. C. and D. B. Wortman, "Structured Subsets of the PL/I Language", Research Report, Computer Systems Research Group, University of Toronto

14. King, J. C., "Symbolic Execution and Program Testing", Communications of ACM, 19,7 (July 1976) 385-394

15. Ledgard, H. F. and M. Marcotti, "A Geneology of Control Structures", Communications of ACM, 18,11 (Nov 1975)

16. Matuzek, D., "The Case for the Assert Statement", SIGPLAN Notices, August 1976

17. Teitelbaum, R. T., "Formal Syntax of the PL/CS Language", Technical Report 76-[illegible], Dept of Computer Science, Cornell University