COPE: A COOPERATIVE PROGRAMMING ENVIRONMENT

James Archer, Jr.
Richard Conway

TR 81-459
June 1981

Department of Computer Science
Upson Hall
Cornell University
Ithaca, New York 14853

COPE: A Cooperative Programming Environment

James Archer, Jr.* and Richard Conway
Cornell University

1. Introduction

COPE is unusual among interactive program development systems in its cooperative attitude with respect to user actions. It is cooperative both in the sense of being flexible and tolerant with respect to the form of user entries, and in being willing to perform chores that the user is generally asked to do for himself. COPE is a research vehicle, so it is deliberately extreme in this respect to permit exploration of the feasibility, psychology and cost of such an approach.

COPE is a self-contained environment, providing a syntax-cognizant editor, an incremental translator, and an interactive execution supervisor. The most obvious contrast is to the Cornell Program Synthesizer [Ref 7], and the easiest way to characterize COPE is as a Synthesizer that incorporates the PL/C approach [Ref 3] to automatic error-repair. While COPE and the Synthesizer were begun at the same time, they are completely separate systems. The Synthesizer was completed first, and COPE has had the benefit of experience with that system.

Typically, modern development systems require the user to know at least three different languages:

1. the host programming language
2. a command language to control the actions of the development system
3. a debugging language of 'immediate' statements or commands.

In some cases, notably the Cornell Synthesizer, a subset of the command language consists of 'structure commands' that generate (e.g. synthesize) constructs in the host language. COPE simplifies this situation by:

1. eliminating the debugging language altogether by allowing (almost) any construct in the host programming language to be (optionally) executed immediately, and
2. allowing any fragment of the host programming language to, in effect, be interpreted as a 'structure command'.

The result is a system with only two languages:

1. the host programming language, with a few extra statement types (TRACE, SLOW, PAUSE, and NOCHECK)

This work has been supported in part by the National Science Foundation under grant MCS 77-08198, and in part by ARPA under grant MDA 903-80-C-0102.

*Current address: Computer Systems Laboratory, Stanford University

2. a minimal command language (only 24 commands, including those for cursor motion, character editing, tabing, etc.)

Moreover, COPE is totally 'mode-free' -- the commands are not hierarchical -- so that every command is available at all times. The overall result is a system with an exceptional degree of conceptual integrity that is, at the same time, unusually easy to learn and use.

COPE also has an unusual recovery facility -- directed at user errors rather than system errors. UNDO and REDO commands make it easy to recover from editing errors during development of a program. The same facilities make it possible to 'back up' the execution of a program during testing.

COPE is written in 'C', runs under UNIX on a VAX, and is available to other institutions on request.

2. Actions and Commands

COPE draws a clear distinction between statements -- the underlying objects with which the system is concerned, and commands -- the instructions by which the system itself is controlled. Statements are constructs in the host programming language that can either be saved (in procedure files) for subsequent execution, or can be executed immediately (and not saved). Commands are performed immediately when given, and cannot be saved.

The use of COPE is a sequence of actions. Each action is initiated by the user entering some command; the system response completes the action. The commands are the following:

Primary commands:	
[entry] EXECUTE	execute (expanded) entry; resume if null entry
[entry] FILE	insert (expanded) entry at edit-ptr
[entry] REPLACE	replace edit-ptr unit with (expanded) entry
[n][file] MOVE	replace file with n units from edit-ptr
[file] COPY	insert copy of file at edit-ptr
Recovery commands:	
UNDO	undo effect of previous primary command
REDO	re-enter the last command undone
Display format commands:	
CONDENSE	condense edit-ptr unit to single line format
EXPAND	expand edit-ptr unit to multi-line format

Editing and cursor motion commands:

[entry] FETCH	copy unit from edit-ptr into entry-window
UP	move edit/exec-ptr up one line
DOWN	move edit/exec-ptr down one line
BACK PAGE	move edit/exec-ptr up one page
FORWARD PAGE	move edit/exec-ptr down one page
LEFT	move entry-cursor left one character
RIGHT	move entry-cursor right one character
LEFT END	move entry-cursor to left end of line
RIGHT END	move entry-cursor to right end of line
WORD TAB	move entry-cursor right to next word
STMT TAB	move entry-cursor right to next stmt
CLEAR	clear character at entry-cursor
ERASE	erase characters from entry-cursor to right end

Miscellaneous:

STATUS	display detailed description of system state
QUIT	save state and interrupt session

Each command is specified by a special-function key, in contrast to statements, which are textual. (For keyboards with an inadequate number of special-function keys, different commands are assigned to the same key and distinguished by some form of 'shift'.)

2.1 The Choice between FILE and EXECUTE

Statements are entered as text. For each textual entry, the user has the choice of saving the resulting statements, or executing them immediately. Consequently, the FILE and EXECUTE commands dominate COPE, both in concept and in use.

The FILE command directs the system to make an insertion (at the 'edit-pointer position' in the 'current file') of a construct based on the textual argument given with the command. The meaning of 'based on the argument' depends on the type of the target file. Each file type is served by a separate entry-editor. Most of these editors are simple; the editor for procedure-type files is very complex (see Section 3). The procedure-syntax-editor (PSE) generates an insertion that may represent an expansion, repair or other modification of the entry supplied by the user.

The EXECUTE command causes some object to be executed according to the semantics of the host programming language (which, in the prototype implementation of COPE is a highly disciplined subset of PL/I called PL/CS [Ref 4]). There are, in effect, three different forms of EXECUTE, depending on the nature of the text argument:

1. If the argument is a file-name, the contents of that file are executed as a program (ab initio, as a MAIN procedure).

2. If the argument is null, the system resumes execution of whatever procedure was last executing.

3. If the argument is neither null nor a file-name, it is executed as if it had been inserted into the current procedure at the point of interrupt (although that insertion is not, in fact, made).

This means that any construct of the host language can be executed 'immediately' (much like the facility in APL or PPL), providing a powerful debugging language as well as a 'desk calculator facility' (see Section 4.2).

Facilities to control the pace of execution and the nature of the screen display, that would ordinarily be viewed as commands, have been made statements in the PL/CS language. When executed immediately they act as the usual commands, but the ability to include them as statements means that a procedure can control its own display and rate of execution.

2.2 Undoing and Redoing User Actions

The UNDO and REDO commands give the user access to a command history automatically maintained by the system, and provide a convenient way either to recover from accidents during development or to reverse the course of execution.

The UNDO command simply undoes the previous primary command -- the system is restored to precisely the state that existed prior to entry of the last command. The result of undoing a FILE or REPLACE command is obvious. The file is restored to its state before the last insertion, and the argument of the last command is restored to the 'entry-window'. Less obvious is the enormous power of undoing the EXECUTE command. This effectively allows one to 'back up' the course of an execution. Note that commands and not statements are undone, so the execution is restored to the state that existed prior to the last EXECUTE command. But since execution during testing is typically a sequence of short intervals, this is adequate to effectively achieve 'reverse execution'. Moreover, it is easily understood, easily used, and not prohibitively costly (in comparison to true reverse execution [Ref 8]). To use this facility for testing, one runs normally until trouble is apparent. Then execution can be undone to some prior safe point, and advanced more slowly and informatively from there.

REDO simply re-enters the last command that was undone. REDO is undoubtedly less important than UNDO, since in effect, it only provides recovery from over-enthusiastic use of UNDO. Only extensive experience with COPE will indicate how useful the REDO capability really is.

UNDO and REDO are implemented by a substantial stack, so that for all practical purposes unlimited depth of recovery is provided. The system is able to move backward and forward in this command history far beyond most users' extreme requirements. (See Section 5.2.)

These recovery facilities are both convenient and powerful. We suspect that even those who might consider COPE unnecessarily charitable, will find the UNDO facility very attractive. Once it is apparent that interactive systems can easily provide such a facility, it should be obvious that interactive systems should all behave this way.

3. The Entry of Procedure Text

The most distinctive characteristic of COPE (relative to other integrated development systems) is the treatment of procedure text by the 'procedure-syntax editor'. The task of the PSE is the following:

> Given any arbitrary textual entry, and a position in the procedure specified by the edit-pointer, make an insertion that
> a. honors the information in the textual entry
> b. respects the context at the edit-pointer
> c. generates everything that is unambiguously implied by the entry
> d. leaves the procedure in a syntactically-correct state.

The PSE can be viewed in different ways:

1. Given complete, correct, continuous procedure text, the PSE is an unobtrusive editor and 'pretty-printer'.

2. Given flawed text, the PSE is an interactive, error-repairing parser, in the PL/C tradition.

3. Given isolated keywords, the PSE is a program generator, somewhat like Hansen's EMILY [Ref 5] or Teitelbaum's Synthesizer.

The requirement that the procedure always be left in a syntactically-correct state is aided by two special devices:

1. Required elements can be arbitrarily generated. For example, PL/CS requires that all loops be named. If the user gives an entry that implies a loop, but does not specify a name, a unique name is automatically generated. (Subsequently this assigned name can easily be changed by the user.)

2. When it is not practical to generate a required element (for example, for conditions or expressions), a 'syntactic variable' is supplied. For example, the syntactic variable 'cond' represents a condition, and is supplied when the syntax requires a condition but the user has not yet supplied one. Syntactic variables are distinctively displayed in lower-case letters. (The case in which user entries are given is immaterial, but in the programs generated by COPE cases are significant and informative.)

The entry process is associated with the display of the 'edit screen'. This consists of five windows, arranged as follows:

```
-------------------------------------------------------------
! entry window                                              !
!-----------------------------------------------------------!
! previous command window                 ! title window    !
!-----------------------------------------!                 !
! message window                          !                 !
!-----------------------------------------------------------!
! text window                                               !
!                                                           !
```

The COPE prototype is not a 'full-screen' system: user entries are echoed in the entry-window; the system-generated procedure text appears in the text-window. The previous-command-window shows the top level of the command stack, effectively indicating the command that would be undone by UNDO. The message-window displays various prompts and explanations. The title-window identifies the name and type of the file being edited.

For example, suppose a new procedure SAMPLE is to be developed. A file named .SAMPLE is used to store procedure SAMPLE. The user would give the FILE command with text argument '.sample'. Initially the edit-screen would look like the following:

```
Prev cmd: <FILE> .sample                    !Editing proc
beginning edit                              !  .SAMPLE
---------------------------------------------------------
=> SAMPLE: PROC;
       END SAMPLE;
```

The '=>' symbol at the left is the edit-pointer, which marks the point at which new constructs will be inserted (and also controls FETCH, REPLACE, MOVE, COPY, CONDENSE and EXPAND).

Now suppose you wanted to construct a loop in the body of SAMPLE. Any reasonable fragment of the source language syntax for a loop will suffice as an entry. For example, the command 'while FILE' would cause the edit screen to appear as follows:

```
Prev cmd: <FILE> while                      !Editing proc
                                            !  .SAMPLE
-----------------------------------------------------------
   SAMPLE: PROC;
=>     W1: DO WHILE cond;
           END W1;
       END SAMPLE;
```

At this point, any entry that generated a valid condition would be inserted in place of 'cond', but an entry that is not acceptable as a condition would simply be inserted in the body of the loop. For example, the command 'set x sum = sum + x FILE' would cause the screen to appear as follows:

```
Prev cmd: <FILE> set x sum = sum + x          |Editing proc
                                              |  .SAMPLE
-------------------------------------------------------------
   SAMPLE: PROC;
      DCL (X) FLOAT;
      DCL (SUM) FLOAT;
      W1: DO WHILE cond;
         GET LIST(X);
=>       SUM = SUM + X;
         END W1;
      END SAMPLE;
```

If the UNDO command were given at this point, the edit screen would appear as follows:

```
set x sum = sum + x
Prev cmd: <FILE> while                        |Editing proc
previous command undone                       |  .SAMPLE
-------------------------------------------------------------
   SAMPLE: PROC;
=>    W1: DO WHILE cond;
         END W1;
      END SAMPLE;
```

The entry line could now be arbitrarily edited, and then the FILE command could be given again.

Alternatively, instead of UNDO, suppose the edit-pointer had been positioned at the first line of the loop and the FETCH command had been given. The entire loop would be copied into the entry window:

```
W1: DO WHILE cond; GET LIST(X); SUM = SUM + X; END W1;
Prev cmd: <FETCH>                             |Editing proc
                                              |  .SAMPLE
-------------------------------------------------------------
   SAMPLE: PROC;
      DCL (X) FLOAT;
      DCL (SUM) FLOAT;
=>    W1: DO WHILE cond;
         GET LIST(X);
         SUM = SUM + X;
         END W1;
      END SAMPLE;
```

Many COPE commands, including FETCH, are 'unit-oriented'. A unit is defined as a line, plus all subsequent lines indented with respect to that initial line. For example, an assignment statement is a unit consisting of a single line. In the example above, the entire W1 loop is a single unit -- which FETCH copies in its entirety into the entry area. The entry window displays a segment of this area, automatically scrolling left or right with movement of the entry-cursor.

The system is indifferent to the origin of text in the entry area. It doesn't matter whether this has been entered directly from the keyboard, copied from text with FETCH, copied from another file with COPY, or some combination of these. Whatever its origin, the text can be processed by FILE, REPLACE, or EXECUTE. For example, the text FETCHed in the example above could be edited to the following:

```
W1: DO WHILE i < n; GET LIST(X); SUM = SUM + X; i = i + 1; END W1;
```

Then this revised version could be used to replace the original. The REPLACE comand replaces the entire unit denoted by the edit-pointer with the PSE-edited version of the entry. In this case, the entire W1 unit would be replaced with a new version that includes a condition and an additional statement in the body:

```
Prev cmd: <REPLACE> W1: DO WHILE i < n; GET L!Editing proc
                                           !  .SAMPLE
------------------------------------------------------------
   SAMPLE: PROC;
      DCL (X) FLOAT;
      DCL (SUM) FLOAT;
      DCL (I) FLOAT;
      DCL (N) FLOAT;
      W1: DO WHILE (I < N);
         GET LIST(X);
         SUM = SUM + X;
         I = I + 1;
=>       END W1;
      END SAMPLE;
```

The FETCH-edit-REPLACE sequence did not have to be used to make these changes in W1. With the edit-pointer positioned on the W1 line, the entry 'i<n FILE' would have replaced 'cond' by '(I<N)' since insertions are made at the first valid position. Similarly, after repositioning the edit-pointer to the SUM line, the entry 'i=i+1 FILE' would cause the assignment statement incrementing I to be inserted after the SUM statement.

Note that in this example, if FILE had accidently been given instead of REPLACE, a second loop (named W2) would have been inserted in the body of W1 instead of making the desired changed in W1. But any such error in the use of the commands is easily remedied by UNDO.

All variables must be explicitly declared in PL/CS but, as indicated in the examples, COPE is quite willing to generate the required declarations. The normal default data type is FLOAT, but certain types of usage imply other attributes. In any event the user can easily change the attributes provided, or add others (EXTERNAL, STATIC, INITIAL, READONLY).

3.1 Entry Forms and Methods

Even these short examples should suggest the wide variety of entry methods allowable in COPE. At one extreme is the 'structure command' strategy, as employed by the Synthesizer. Entry of individual keywords causes structure templates to be generated, with syntactic variables to prompt the user for missing elements. For example, 'while' is effectively equivalent to the Synthesizer '.dw' command. However, unlike the Synthesizer, COPE does not have a unique command associated with each construct -- any unambiguous fragment of the construct itself serves as a command. A second difference is that COPE treats all constructs in the language consistently -- no distinction is made between constructs that must be generated by commands and those that must be entered directly as text. Any construct can be generated by a 'keyword command' or can be entered directly as text.

At the other extreme, a COPE user can plan a procedure on paper first, and then enter it as complete, continuous text. The user need not use, or even be aware of, COPE's willingness to generate substantial portions of the procedure text.

Between these extremes, each user can devise a personal entry method. In actual practice, many programmers tend to plan and write programs using some informal, abbreviated dialect of the programming language, and then fill this out to full and proper form more or less automatically as they enter the statements. COPE accepts these informal, abbreviated forms directly. However, COPE does not specify a single, rigid shorthand form, but rather allows each user to devise his own. In practice, a user's entry form may evolve over time with increasing experience. It may also vary with different contexts and for different types of problem.

Essentially, COPE allows the user to choose the entry mode, rather than have the system impose one particular mode. (Teitelbaum characterizes COPE as 'anarchistic' -- in contrast to the Synthesizer's 'totalitarian' approach.)

A useful byproduct of COPE's entry flexibility is its ability to accept ordinary source language text from arbitrary files or other external sources. This means that program segments prepared under other editors can easily be imported into COPE. It also means that COPE programs can write other programs -- that can subsequently be executed under COPE. (Also see Section 4.3.)

3.2 The 'Real' Source Language for COPE

While the host programming language for the COPE prototype is PL/CS, it is apparent that this is the output language rather than the entry language. That is, COPE generates programs in PL/CS -- it does not require entry of programs in that language. While this tolerant approach has been used before, notably in the PL/C family of translators, COPE carries the approach substantially farther than any of its predecessors. In fact, it provides sufficient difference in degree that a significant change in kind is achieved.

In PL/C the system was expected to detect and attempt repair of accidental lapses by the user. What has now become apparent is that if automatic repair is sufficiently powerful, reliable, and unobtrusive, the user learns to capitalize on this facility to simplify his own task. That is, the user deliberately makes 'mistakes' and relies on the system to repair them. Most of these mistakes are deliberate errors of omission. The user in effect enters an abbreviated form of the source language and relies on COPE to supply the missing redundant elements to achieve full syntactic form. The overall reduction in necessary keystrokes depends both on the particular procedure and on the severity with which the user abbreviates, but in general reduction by a factor of least two seems practical.

This raises the interesting question of what is the 'real' source language that is recognized by COPE? It accepts PL/CS, and in any event generates PL/CS procedures, but it obviously also accepts other languages that are essentially abbreviated forms of PL/CS. However, it is not easy to characterize the languages accepted by COPE since the response to a particular entry is heavily dependent on context. One way to view the process is the following.

Consider minimal entries -- a single keyword or a single expression. Keywords can be partitioned into two lists:

Unambiguous keywords (uniquely imply one particular construction):
ASSERT CALL DCL DECLARE DELETE ELSE EXT EXTERNAL FOR GET GOTO IF INIT INITIAL LEAVE NEXT NOCHECK OTHERWISE PAUSE PROC PROCEDURE PUT READ READONLY RECORD RETURN RETURNS SELECT SKIP SLOW SOME STATIC THEN TRACE UNTIL WHILE

Ambiguous keywords (used in two or more constructions):
ALL BIT BY CHAR CHARACTER DO EDIT FILE FIXED FLOAT LIST TO VAR VARYING

The easiest case to consider is the entry of a single token entry from the list of unambiguous keywords. For each such entry the system has a production that generates the 'context-free response'. For example, as illustrated previously, the response to the entry 'while' is:

```
Wn: DO WHILE cond ; END Wn;
```

As another example, the context-free response to the entry 'when' is:

```
Sn: SELECT; WHEN cond ; OTHERWISE ; END Sn;
```

This much is straightforward -- both in implementation and understanding. What makes the process complicated, for both the user and the system, is the necessary sensitivity to the context in which the insertion is to be made.

For example, suppose the context for a 'when' entry is the following:

```
S9: SELECT; WHEN cond ; <insertion-point> OTHERWISE ; END S9;
```

Obviously, the context-free response would be inappropriate at this point, so the system must check first to see whether an implied element is already present before supplying that element. This works in essentially the same way whether the element already exists in the procedure as a result of some previous entry, or whether it is a prior token in the current entry.

The response to the entry of an ambiguous keyword is similar, except that the context-free production is a 'preferred' or 'most probable' construct, rather than a uniquely implied construct. The adaptation to context in this case must be prepared to shift to a less-probable production as well as suppress generation of elements already present.

The entry of an expression is essentially like the entry of an ambiguous keyword. The preferred response has been chosen to be a PUT statement with the given expression as argument. The user can of course alter this choice by providing some context. This can be done by giving a keyword before the expression. For example, 'set x' or 'skip x' produces a different response than just 'x'. Context can also be supplied by a suffix colon or a prefix left parenthesis, which cause an expression to be interpreted as a label or condition, respectively.

Two special cases reflect the fact that the source language was not optimally designed for this service. PL/I's ambiguous use of the '=' symbol is a great nuisance, and ambiguous entries in COPE are resolved in favor of the assignment statement. The system can be coerced into making 'a=b' into a condition by entering '(a=b'. The other problem is in distinguishing between the following constructions:

```
DO;                           DO I = 1 TO expr BY 1;
   I = 1;                        END;
   END;
```

The entry 'do i=1' is resolved to the construction on the right; to achieve the one on the left 'do; i=1' must be entered.

The entry of 'end' is an example of the importance of locating context. END's in the procedure are always automatically generated, so the user never is required to enter any 'end'. Consequently, the entry of 'end' is defined to mean 'move the edit-pointer forward to the next END'. For example, the entry 'while i<j i=i+1 get x sum = sum + x end x' results in the following construction:

```
W1: DO WHILE (I < J);
   I = I + 1;
   GET LIST(X);
   SUM = SUM + X;
   END W1;
PUT SKIP LIST(X);
```

Without the 'end' in the entry, the PUT statement would be included in the body of the loop. Note that this interpretation of the 'end' entry is what makes COPE capable of both generating program structure and accepting complete program input.

Another way to view this procedure entry process is the following. The procedure under development, in its final form, consists of a sequence of n tokens (words, numbers, operators, punctuation, etc.):

S1, S2, S3, ..., Sn

The user must specify many of these tokens, but in general not all of them since there is considerable redundancy in programming languages. COPE is designed to give the user maximum flexibility both in determining which tokens to supply and the order in which to supply them.

At each point in the entry process some skeleton of the token sequence exists, and the user specifies some subsequence of additional tokens, and a position in the existing sequence where the insertion is to be made. For each such insertion COPE generates the redundant tokens implied by those given, and also any tokens necessary to make the existing string hospitable to the new insertion.

For example, suppose the following subsequence is to be inserted:

```
W1: DO WHILE (I < J); END W1;
```

There are thirteen tokens in this subsequence, but only four of these:

```
WHILE, I, <, J
```

must be supplied by the user (assuming he will accept a generated loopname). COPE accepts any of the following choices by the user:

'WHILE I < J' as a single entry

'WHILE' and 'I < J' as two consecutive entries

'WHILE' as an initial entry, and 'I < J' arbitrarily later, after appropriate positioning of the edit pointer

Similarly, the body of the loop can be supplied as part of the initial entry, as an immediate subsequent entry, or arbitrarily later. In general, COPE strives to allow the user to arbitrarily partition any subsequence entry into reasonble sub-entries without changing the construction that results. Conversely, he should be able to combine any consecutive entries into a single entry without changing the results. That is, 'Si Si+1 ... Si+k' as a single entry should have the same effect as 'Si Si+1 ... Si+J' and 'Si+J+1 ... Si+k' as two consecutive entries. While this cannot be absolutely achieved, it works surprisingly well and the astonishment factor is quite low.

The implementation of the PSE is discussed in Section 5.1.

3.3 Condensation of Procedure Text

A general limitation of all screen editors is the relatively few lines of text that can be displayed at one time. Even as larger screens becoming available, this will still be a significant limitation. Consequently, it is useful to be able to 'condense' text that is peripheral to the current locus of interest. Both the Synthesizer and PLE1L [Ref 6] employ variations of this strategy; COPE demonstrates a third version of condensation.

COPE uses two alternative display formats for procedure text. The normal, or 'expanded' format, is the multi-line, indented form shown in the examples above. The alternative 'condensed' form concatenates statements onto a single line, replacing all but the first with an elipsis. For example, consider the following program segment, in normal expanded form:

```
            L3: DO I = 1 TO N BY 1;
               SUM = 0;
               L4: DO J = 1 TO M BY 1;
                  GET LIST(X);
         =>       SUM = SUM + X;
                  END L4;
               PUT SKIP LIST(SUM);
               END L3;
```

If the CONDENSE command were given with the edit-pointer positioned as shown, the 'L4 unit' would be condensed and the display would appear as follows:

```
            L3: DO I = 1 TO N BY 1;
               SUM = 0;
         =>    L4: DO J = 1 TO M BY 1; ...
               PUT SKIP LIST(SUM);
               END L3;
```

Condensation is hierarchical -- if CONDENSE were given again, the L3 unit would be condensed and the display would appear:

```
         => L3: DO I = 1 TO N BY 1; ...
```

Repetition of CONDENSE would eventually cause the entire procedure to be condensed to a single line:

```
         => SAMPLE: PROC; ...
```

The EXPAND command restores a condensed unit to normal, multi-line form. Each EXPAND command expands only the outermost condensed level (enclosing the edit-pointer line), so repeated EXPANDs are be necessary to fully expand deeply condensed text.

The display format is a relatively permanent property of a unit; that is, a condensed unit remains condensed until explicitly expanded, persisting from one session to another. It appears in condensed form on both the edit screen and the execution screen (described below).

4. Program Execution

At any stage of development, a program can be executed. If a file name is given as argument to the EXECUTE command, the previous execution environment is erased and the specified file is executed as a MAIN procedure. Execution will continue until the one of the following events occurs:

1. a syntactic variable (denoting a required but unspecified element) is

encountered

2. a PAUSE statement is encountered

3. a run-time error is encountered

4. the output window is filled

5. a GET statement has insufficient data

6. normal execution is completed.

7. any key of the keyboard is pressed, presumably indicating the user wants to enter a command.

When any of these events occurs, execution is paused and the system awaits the next user command. While execution is paused, there are six types of action the user can take:

1. Execution can be resumed.

2. Execution can be backed up (by UNDO).

3. Immediate statements can be executed.

4. The execution-pointer can be moved (effectively an immediate GOTO).

5. Some file can be modified.

6. The execution can be cancelled (by beginning execution of another program.

All commands are available during an execution pause; none are inaccessible because of the state or mode of the system. In particular, the editing commands that change a file cause the edit screen to reappear, but this happens automatically and the user need not explicitly 'shift to edit mode'. If display is changed to the edit screen, it automatically reverts to the execution screen whenever an EXECUTE command is given.

It is important to realize that the system is almost always in this paused-in-execution state. (The exceptions are brand new systems in which nothing has been executed, or a system in which the most recently executed procedure has been deleted.) Even when an execution has been normally completed, it is still considered to be paused and the environment is preserved (so that immediate statements can still be executed in that environment). The user can always determine what state the system is in by giving the STATUS command. This displays a comprehensive summary of procedure calling history, position of execution pointer, position in standard input and output files, etc.

4.1 The Execution Screen

The execution screen is displayed whenever statements are being executed. It consists of seven windows, as shown below:

```
 ----------------------------------------------------------------------
 ! entry window                                                       !
 !--------------------------------------------------------------------!
 ! previous command window                          ! stack window    !
 !--------------------------------------------------!                 !
 ! message window                                   !                 !
 !--------------------------------------------------------------------!
 ! trace window                                 ! check window        !
 !                                              !                     !
 !                                              !                     !
 !--------------------------------------------------------------------!
 ! execution output window                                            !
 !                                                                    !
```

This is much like the execution screen of the Synthesizer in that it simultaneously displays:

a. the trace window with the text of the procedure being executed, with an 'execution-pointer' indicating the statement currently being executed

b. the check window showing variables and their current values

c. the output window showing the results of executing (PUT) statements.

For example, the screen might appear as follows when execution is paused for a 'page turn' in the output window:

```
Prev cmd: <EXEC> .stars                                  !Executing:
Output window full; <EXEC> to clear and continue         ! .STARS
-------------------------------------------------------------
   STARS: PROC;                                 ! S = ********
      S = '*';                                  ! I = 7
      I = 1;                                    ! N = 20
      N = 20;                                   !
      W1: DO WHILE (I < N);                     !
         S = S !! '*';                          !
>>       PUT SKIP LIST(S);                      !
         I = I + 1;                             !
      PUT SKIP(2) LIST('last line has be!
                                                !
                                                !
-------------------------------------------------------------
**
***
****
*****
******
*******
```

The Synthesizer has dramatically demonstrated the effectiveness of this type of display -- it is hard to appreciate until one has seen it. It seems relatively difficult for a user not to understand what is taking place in execution when confronted with the dynamic and simultaneous display of these three types of information. COPE differs from the Synthesizer in this regard only in minor ways.

As on the edit screen, lines on the execution screen are a scarce resource. When the execution-pointer moves to a line not shown in the trace window the window contents are automatically scrolled. Similarly, when the number of variables exceeds the size of the check window, replacements are made by a least-recently-changed algorithm. But since both scrolling and replacement tend to be visually disturbing, the effectiveness of the display would be impaired if either event occurs too frequently. To minimize the frequency of scrolling:

1. Lines are divided between the trace window and the check window (rather than arrange the three major windows vertically). This means that statement lines must often be truncated in the trace window, and names and values must sometimes be truncated in the check window, but neither seems to interfere with the user's ability to understand what is happening during execution. (The user can easily pause during execution and cause the full text of a line or the full value of a variable to be displayed.)

2. Lines that are not essential to understanding the progress of execution are automatically omitted from the trace window. For example, neither END nor DECLARE lines are displayed on the execution screen. (Note that with automatically formatted display, END lines are completely redundant -- structural information is completely supplied by indentation. ENDs are displayed on the edit screen only for compatibility with the source language. Future versions might well omit END lines altogether.) SLOW, TRACE and PAUSE statements are also not shown on the execution screen. The overall reduction in the displayed length of a procedure can be quite significant.

3. The user has considerable control over the granularity of the trace. CONDENSEd units appear in single-line elided form, and the TRACE(expr) statement also provides explicit control over the nesting depth of statements to be displayed in the trace window. TRACE(0) suspends the trace altogether, and if given as the first statement of a called procedure, leaves the trace window unchanged.

The SLOW statement controls the speed of execution by limiting the frequency with which the execution screen is redrawn. Since the movement of the execution pointer is a redraw, this can limit speed to a point where it can be visually followed, and precisely interrupted. For example, SLOW(100) would limit execution speed to at most one 'step' every 100 tenths of a second. But note that a step (for this purpose) is essentially a line in the trace window. This means that a condensed unit is a single step, and consequently can run faster than when it is expanded. Similarly, the suppression of detail trace display by the TRACE statement also permits undisplayed units to run at full speed.

Since COPE is a development system, the defaults have been established to favor testing rather than efficient execution. This also means that the most powerful diagnostic environment is automatically established for a user, without his having to know of the available features. For example, SLOW(5) is the default, and the user must specify SLOW(0) to set full-speed execution. TRACE(2) is the default, and the user must specify TRACE(0) to suppress tracing. Similarly, the default is to 'check' all variables, and the user must specify NOCHECK(list) to selectively exempt variables from checking. NOCHECK(ALL) suppresses all checking (in the procedure in which this statement is executed), and TRACE(0) suppresses both tracing and checking.

Note also that TRACE and SLOW are statements, rather than commands. While they can be executed immediately, effectively as commands, they can also be inserted in a procedure so it can manage its own execution display. For example, the statement TRACE(0) placed at the beginning of a procedure after one is satisfied with its correctness, allows it to be called and executed unobtrusively regardless of the display characteristics of the procedures that call it. As another example, stored SLOW statements give a procedure control over the timing of output statements so that programs involving animation or real-time response are feasible. Both flexibility and conceptual simplicity are enhanced by extending the source language with such statements, rather than adding them only to the command language.

4.2 Immediate Execution

The entire source language (except for PROC and DCL) is capable of immediate execution. Any text entry that could be FILEd in a procedure, can alternatively be EXECUTEd directly and immediately. This can be useful both as a 'desk calculator system' independent of any stored procedure, and as a 'debugging language' during testing of a stored program.

Text to be executed is subjected to precisely the same treatment by the PSE as text destined for insertion in a procedure file. For example, a variable or expression alone is 'repaired' to a PUT statement:

```
'x'                  becomes    'PUT SKIP LIST(X);'

'sqrt(3.1416)'       becomes    'PUT SKIP LIST(SQRT(3.1416));'
```

This repair is convenient for both the desk calculator and debugging uses (which is why this seemingly arbitrary repair choice was made). More complicated entries can be given:

```
'do j = 1 to 10 sqrt(j)'    for desk calculator use

'do j = 1 to 10 x(j)'       for diagnostic display
```

Immediate execution in a block-structured language is, of course, a potentially confusing process. The rule is simply that execution takes place in the current environment -- literally as if the statements were inserted in the current procedure at the point at which it is paused, and executed there. The normal PL/I scope rules apply. No extraordinary provision is made to give immediate statements access to objects that normal stored statements could not see.

4.3 Execution of non-Procedure Files

A file specified as argument to the EXECUTE command will ordinarily be of type procedure -- but COPE does not require that this be the case. Any type of file can be executed. If execution of a non-procedure file is specified, that file is simply streamed through the PSE into .TEMP, and .TEMP is executed. The specified file itself is of course not changed.

The same privilege is accorded to the CALL statement. The argument of CALL is a procedure name, stored in a file with the same name. If no such file exists, one is created containing an empty procedure, so the CALL effectively becomes a null statement. On the other hand, if a file exists, but is not of type procedure, the contents of that file are streamed through the PSE into .TEMP, and the procedure is executed from there. This type-indifference coupled with the enthusiastic repair facility of the PSE gives COPE an interesting capability of executing almost anything.

5. Structure of the COPE Implementation

A series of Cornell CS Technical Reports describe the COPE implementation in detail (Ref 1), but several unusual aspects of the system might be mentioned here. All involve the exploitation of a novel file system.

The entire COPE implementation is based on its file system. Almost every module of the system draws its input from files and writes its output to files. Moreover, the file system used internally in the implementation is the same system the user sees for procedures, data and output.

In many respects COPE can be viewed as a database system. Each user command is a 'transaction', which is analyzed and processed by updating various files. The top level control is simply the following loop:

```
DO UNTIL (command = QUIT);
   Get next command (textual-entry and command-key);
   Process the command;
   END;
```

The 'process command' module is a set of parallel routines -- one for each different command. These routines draw upon one, or both, of the following processes:

```
Procedure syntax editor, with arguments specifying
     textual-entry
     target file
     position in target file (based on edit pointer)

Execution supervisor, with arguments specifying
     procedure file
     position in procedure file (based on execution pointer)
     environment (which is itself a file, and also contains
                pointers to other files)
```

As described in Section 3, the PSE constructs a program segment based on the textual entry, and inserts it at the specified point in a procedure file. Procedure files are maintained in an internal form that is a compromise between the requirements of display and execution. A procedure display routine translates this form into displayable lines (in slightly different form for the edit and execution screens), and the execution supervisor interprets this form to execute the program.

The FILE, REPLACE and COPY routines use only the PSE, in the obvious way. Similarly, the new-program and the resume forms of EXECUTE use only the execution supervisor. But some of the unusual flexibility in COPE comes from the ability to use first the PSE and then the execution supervisor in the same command. For example, immediate execution is provided simply by invoking the PSE with .TEMP as the target file, and then invoking the execution supervisor with .TEMP as its subject. Similarly, the execution of non-procedure files (either by the EXECUTE command or the CALL statement) simply requires exercise of the PSE before invocation of the execution supervisor.

5.1 Two-Level Parsing

The COPE PSE is implemented by a novel two-level parsing strategy. The top level is an LL(1) parser for the structural syntax of the language, in which each expression is essentially regarded as a single token. 'Insertion-only' correction is provided at this level by replacing the normal error entries in the parsing table by calls on insertion routines. Quite powerful structural repair is achieved from a small set of parsing tables. It does, of course, require that all structural keywords be reserved.

The second level parses only the expressions. In many contexts the type of expression required has already been determined by the top level structural analysis. Some error repair is effected by the expression parser, but it is relatively modest compared to the structural parser. There are several reasons for this restraint. In the first place, the system must be careful not to repair two consecutive expressions into a single expression. For example, the entry 'x y' should probably not be made into a single expression, but the entry '(x y)' should probably be repaired by the addition of a comma. But this requires arbitrarily extensive lookahead, since it is not clear what action should be taken for the prefix '(x y ...'. Note however, that the two-level strategy has effectively partitioned the input stream so that 'arbitrary lookahead' is actually not very far, and even cubic repair algorithms are quite practical.

The second restraint on expression repair is a consequence of the user's privilege of skipping required expressions and filling them in later. For example, the user may elect to specify the body of a loop in the initial entry and come back later to supply the condition. That is, given the entry 'while i=i+1', the expression parser should not be too enthusiastic to make a condition out of whatever follows the keyword 'while'. (PL/I's ambiguous use of the '=' symbol is particularly unfortunate for our purposes.)

Returning to the top level parser, the parsing stack of the LL(1) parser is increased whenever the input lookahead requires a new production. COPE uses a 'uniquely implied' criterion to push new productions when it is 'known' that the matching terminal does not already exist in the stack. Consequently, it is crucial that the parser intelligently determine, for each new keyword, whether it is confirmation of a keyword already implied by some prior keyword (and consequently already on the stack), or is a new construct. To facilitate this decision, COPE keeps track of the number of each terminal that it 'expects' to see before completing the current entry. New parsing actions are used whose function is conditional on the count associated with the input terminal. When the terminal has been predicted, parsing proceeds by simulating insertions into the input. When the terminal has not been predicted, a new production is pushed onto the stack.

The COPE parsers are table-driven and the PL/CS host language could readily be replaced with comparable subsets of, say, PASCAL or Ada. But it should be noted that to some extent the effectiveness of the COPE PSE depends on the number of different contexts in which each individual keyword can appear in the source language. Recall from Section 3.2 that in PL/CS the list of unambiguous keywords is much longer than the list of ambiguous keywords, so 'uniquely implied' responses are much more common than 'most probable' ones. This would be less true of a rich language like full PL/I or Ada.

5.2 The Recovery Mechanism

The implementation of the UNDO and REDO facilities is based on a special file named .LOG. This file is a chronological record of commands (with the corresponding textual entries), automatically maintained by the system. Schematically, it works in the following way:

```
                                   .LOG contents:
                                ----------------------
                                |commands previously |
                                |  executed          |
                                |                    |
   prev command pointer=>       | prev cmd executed  |
                                |                    |
                                |commands previously |
                                |  undone            |
                                ----------------------
```

The line indicated by the previous-command-pointer is the line displayed in the 'previous command window' of the display screen.

Each time a primary command (FILE, REPLACE, EXECUTE, MOVE, COPY) is given, the following action takes replace with respect to .LOG:

1. The command and its text entry (if any) are inserted as a new line immediately after the 'previous command line'.
2. The previous command pointer is moved to the line just inserted, and that line is also displayed in the previous command window.
3. The command is performed, and checkpoint information is added to .LOG (in a form not visible to the user).

When the UNDO command is given, the following takes place:

1. The checkpoint information in .LOG is used to restore the system to the

state that existed prior to the previous command.

2. The previous command pointer is moved up one line in .LOG.

3. The line denoted by the previous command pointer is displayed in the previous command window.

When the REDO command is given, the following takes place:

1. The line in .LOG immediately below the previous command line is 'submitted' to the system, just as if it had just been entered from the keyboard.

2. The previous command pointer is moved down to the line just submitted, and that line is displayed in the previous command window.

The .LOG file is displayable, like any other file, but the portion above and including the previous command line is not editable, since changes could well jeopardize the UNDOability of the commands. On the other hand, the portion below the last command line can be arbitrarily edited, which of course, changes the result of REDO commands.

The checkpoint information that makes UNDO possible is surprisingly simple to manage. This is a consequence of COPE's complete dependence on its file system. Everything -- the user's files, the environment stack, the symbol table, the screen images -- is maintained in a file. Every file is paged, and modified pages are not overwritten. Consequently, the checkpoint information consists simply of references to both the old and new version of each page that is modified during the execution of a command. This log of page references is maintained in .LOG, with each command line separating one sequence of page references from the next. (Command lines, but not page references, are displayable.) UNDO is accomplished simply by restoring the old versions of file pages, back as far as the last command line, and deleting the modified pages. The modified pages need not be preserved in anticipation of REDO commands since they can readily be recreated by the system when the command is resubmitted. (Furthermore, this allows the possibility of modifying the commands before resubmission.)

The checkpoint facility is not intended as protection against system failures, and the decision as to what pages and when pages should be copied into non-volatile storage is considered a separate issue from the UNDO- REDO facility.

6. The File System

Although the file system used by COPE was developed specifically for this system, it embodies ideas that are not peculiar to this type of development environment and could well be exploited in other systems. Similarly, the concepts are not peculiar to the PL/I language, although the source language facilities in the prototype COPE are, of course, expressed in PL/I-like terms.

The central idea of this file system is the use of a single file structure that is very simple to understand and use, yet provides considerable flexibility. Essentially this is accomplished by sacrificing some degree of execution efficiency. The second point is that the same file system is used throughout COPE. The user employs it for procedures, input data, and auxiliary files. The system itself employs it to manage its displays, provide working storage (symbol table, runtime stack, etc.), and to implement the recovery facilities.

Each file consists of a sequence of records, automatically numbered sequentially. Optionally, each record can have a key value, and keyed records are maintained in order of increasing key value. Unkeyed records can exist between any pair of keyed records. Each record consists of zero or more 'items'; each item is simply a (varying length) character string. For sequential access, the file can be regarded as simply a sequence of items (the record boundaries are insignificant). But the keyed records are directly accessible, providing the basis of a convenient direct-access facility. Essentially, each item in the file is an 'item' in the PL/I LIST format sense, except that strings are not quoted (so the system can easily read what it has written).

A file can have a particular 'type' of content. For each special type an encoder and decoder are provided to translate between display form and the internal file form. In particular, files whose type is 'procedure' are maintained in a coded form for which the procedure syntax editor (PSE) is the encoder, and a decoder restores procedures to textual form for display. This decoder is cognizant of the different requirements of the edit and execution screens, and is also sensitive to the CONDENSE/EXPAND choice the user has made for each unit of the procedure.

At a lower level (invisible to the user), files are paged into fixed size blocks and the interface with the host file system is entirely in terms of such blocks. More detailed descriptions are given in References 1 and 2.

6.1 The .TEMP File Stack

As noted in Sections 4.2 and 4.3, the special file .TEMP is used to host segments that need to be translated before execution. But although this method of introducing content to .TEMP is special, the user can subsequently edit the contents of .TEMP just like any other file.

However, .TEMP is different from other files in that it is, in fact, a stack of files rather than a single file. That is, each immediate execution does not destroy the previous contents of .TEMP; it just 'pushes' the previous file onto the stack and creates a new level of .TEMP. (The stack can be 'popped' by the DELETE statement.) Each level of .TEMP has its own file type.

In addition to its role in immediate execution, the .TEMP stack serves two other purposes. First, it is the recovery file for the MOVE command. That is, the semantics of MOVE specify that the target file is cleared (before receiving units from the text of the current file). In fact, the previous contents of the target file are moved onto .TEMP, and the user can recover them intact from that location.

.TEMP is also the default file for both the MOVE and COPY commands. That is, if no file is explicitly specified, .TEMP is the target for MOVE and the source for COPY. This provides a convenient way to manipulate segments of a procedure. For example, to move a unit from one position to another the following is all that is required:

```
position the edit-pointer to the unit to be moved
MOVE
reposition the edit-pointer to the new position
COPY
```

6.2 Eile Erocessing Exteosioos to EL/CS

The standard PL/CS language has only LIST and EDIT forms of GET and PUT, and only a single input and single output file. In COPE the language has been modestly extended to give the user access to the capability of the file system. While this could certainly have been done by including more of PL/I in the PL/CS subset, PL/I's formidable I/O facilities seemed somewhat at variance with the frugal structure of PL/CS. Consequently, we elected to sacrifice compatibility in this regard and experiment with a novel file facility. The following is intended only to suggest the nature of this facility, and not to serve as a detailed user's guide.

The extensions consist only of additional optional phrases in the GET and PUT statements, the addition of a DELETE statement, and the addition of six builtin functions and pseudo-variables. The extensions do not require structures, record I/O, file declarations, file variables or ON conditions. But these simple extensions give the user multiple input and output files, direct access as well as sequential access, mixed keyed and unkeyed records, and variable length records.

Each file has a 'current record pointer', and in the current record, a 'current item pointer'. These pointers dictate the semantics of the GET and PUT statement. For sequential processing, their action is automatic, natural and unobtrusive, and an unsophisticated user need not be aware of them. But the user can interrogate and manage these pointers by means of the added builtin functions and pseudo-variables. These are the following:

REC(filename)
- Function returns the current record number.
- Assignment to pseudo-variable repositions the current record pointer.

REMAIN(filename)
- Function returns the number of records before end-of-file.

KEY(filename)
- Function returns the key value of the current record.
- Assignment to pseudo-variable of a new key value creates a record.
- Assignment to pseudo-variable of an old key value makes that record current.

FIND(filename, keyvalue)
- Function sets the current record pointer to the first record whose key is greater than or equal to keyvalue, and returns the key value of that record.

COUNT(filename)
- Function returns the number of items in the current record.
- Assignment to the pseudo-variable changes the number of items.

ITEM(filename)

Function returns the current item number.
Assignment to the pseudo-variable moves the current item pointer.

The input and output statements can be described in terms of these functions and the pointers they control:

PUT [FILE(fn)] [PAGE[(ps)]] [SKIP[(n)]] [LIST or EDIT ...];

1. If FILE is omitted, output is to standard file .OUTPUT. (File .OUTPUT is automatically cleared each time a new program execution is begun.) If FILE is specified and fn is new, this constitutes an implicit declaration of a new file. If the fn is known, and the first reference executed is a PUT, the current record pointer is set to filesize+1, and the current item pointer to 0.

2. If SKIP is omitted, items are written starting at the end of the current record (that is, at REC(fn), COUNT(fn)+1). If SKIP is specified, output begins at the current record plus n. If n is positive n-1 null records are inserted. If n is zero, output begins at ITEM(fn), updating fields or extending the record as needed.

3. PAGE controls the output window of the execution screen, clearing the window and allocating ps lines to it (the remainder going to the trace/check windows). If ps is not specified, the output window is cleared but its size is unchanged.

GET [FILE(fn)] [NEXT] LIST or EDIT ... ;

1. If FILE is omitted, input is from standard file .DATA. (A new program execution resets the current record pointer in .DATA to 0, but .DATA is not automatically cleared.)

2. If NEXT is given, REC(fn) is incremented and ITEM(fn) is set to 1 (prior to transmission).

3. Items are read starting with REC(fn), ITEM(fn), ignoring record boundaries. RECN(fn) and ITEM(fn) are incremented to reflect number of items read.

DELETE FILE(fn) [RECORD[n]];

1. If RECORD is given, n records are deleted starting with REC(fn).

2. If RECORD is omitted, the entire file fn is deleted. Note that there is no COPE command to delete a file -- immediate execution of the DELETE statement is used.

7. Conclusions

As this is written, COPE is demonstrable but not yet fully serviceable. Since its objective is to permit evaluation of a novel user interface, we will not be able to conclude how well this goal has been achieved until substantial user experience has been accumulated. Consequently, the following are really conjectures or predictions rather than conclusions firmly supported by experience. However, we believe that COPE will support the following points:

1. The full power of an integrated, interactive program development environment can be offered without requiring the user to learn much more than the host programming language, and without forcing users to submit to a rigid entry protocol. In other words, many such systems make the process more complicated than necessary for the user.

2. A generalized 'immediate execution' facility is a natural concept (even in a block-structured language) that is both simple to understand and use, and very powerful.

3. A general recovery facility (with respect to system commands) can be simple to understand and use, and very convenient for the user.

4. Contrary to the conventional wisdom that automatic error-repair is inappropriate in interactive systems, since the user is available and can be forced to make his own repairs, automatic repair is especially useful and effective when each repair can be immediately submitted to the user for acceptance or rejection. When automatic repair is complemented by a convenient UNDO facility, both the system and the user can be relatively bold in this respect. The consequence is a cooperative effort in which the user can make deliberate errors of omission and rely on the system to generate some elements of the program.

Acknowledgements

COPE is the most recent of a long sequence of Cornell systems that have explored diagnostic facilities and fault-tolerant operation. Our debt to the CORC, CUPL, PL/C and PL/CS projects is significant. Many people have been involved in these projects, with key contributions having been made by William Maxwell, Howard Morgan and Thomas Wilcox.

Our colleagues in the prototype implementation of COPE have been Andrew Shore, Len Silver and Steven Worona. Tim Teitelbaum has made many valuable suggestions, Fred Schneider made key suggestions concerning the recovery facility, and Alan Demers and Dean Krafft have also been most helpful in this effort.

References

1. Cornell Department of Computer Science Technical Reports concerning COPE:

TR79-397	'A Program Development System Execution Supervisor' (Archer and Shore)
TR79-366	'A File System Extension to PL/CS' (Archer)
TR79-367	'Implementation of an Unrestricted File Organization for PL/CS' (Archer)
TR79-399	'System Architecture for the PL/CS Development System' (Archer, Conway, Shore, Silver)
TR80-437	'The COPE User Interface' (Archer, Conway, Shore, Silver)

2. Archer, J., 'The Design and Implementation of a Cooperative Program Development Environment', PhD Thesis, Dept. of Computer Science, Cornell 1981

3. Conway, R., and T. Wilcox, 'Design and Implementation of a Diagnostic Compiler for PL/I', Communications of ACM, March 1973

4. Conway, R., and R. Constable, 'PL/CS - A Disciplined Subset of PL/I', Technical Report TR76-293, Dept. of Computer Science, Cornell 1976

5. Hansen, W., 'Creation of Hierarchical Text with a Computer Display', PhD Thesis, Computer Science Dept., Stanford 1971

6. Mikelsons, M., and M. Wegman, 'PDE1L: The PL1L Program Development Environment', RC8513, Watson Research Lab, IBM 1980

7. Teitelbaum, T., 'The Cornell Program Synthesizer: A Microcomputer Implementation of PL/CS', Technical Report TR79-370, Dept. of Computer Science, Cornell 1979

8. Zelkowitz, M., 'Reversible Execution as a Diagnostic Tool', PhD Thesis, Dept. of Computer Science, Cornell 1971

Appendix: The PL/CS Language

PL/CS is a small and 'disciplined' subset of PL/I [Ref 4]. The following PL/I constructs are included:

```
        Statement types:
             Assignment; CALL; GET; PUT; GOTO; LEAVE; RETURN;
             indexed, WHILE and UNTIL loops; compound statements;
             IF; SELECT;
             PROCEDURE (external only, but including functions)

        Data types:
             FLOAT, FIXED, CHARACTER VARYING, BIT(1), arrays,
             INITIAL, STATIC, EXTERNAL
```

PL/CS has the following features incompatible with PL/I:

```
        assertions:
             ASSERT cond
             ASSERT cond FOR ALL index = expr1 TO expr2 BY expr3
             ASSERT cond FOR SOME index = expr1 TO expr2 BY expr3
        READONLY attribute
```

PL/CS is further extended in COPE with the following:

```
        TRACE, SLOW, PAUSE and NOCHECK statements (see Sections 4.1 and 4.2)
        options in GET and PUT, and DELETE statement (see Section 6)
```

Many PL/CS constructs are severely restricted relative to their PL/I counterparts. For example, functions have absolutely no side-effects (even GET, PUT and CALL are not allowed in functions), loops have simple control phrases (WHILE and UNTIL cannot be given in the same loop), GOTOs allow forward reference only, etc. The exclusions and restrictions eliminate many of the inconsistencies and unpleasant surprises found in PL/I, and yield an attractive and useful language. PL/CS is somewhat comparable to PASCAL, and has some relative advantage in string processing and dynamic array dimensioning, and relative disadvantage in the lack of user-defined types and complex data structures.