The CORE User Interface

James Archer, Jr.

Richard Conway

Andrew Shore

Leonard Silver

TR 80-437

Department of Computer Science
Cornell University
Ithaca, New York 14853

September 1980

The CORE User Interface

James Archer, Jr., Richard Conway, Andrew Shore and Leonard Silver

Department of Computer Science
Cornell

The distinctive characteristic of the CORE program development environment is its tolerant user interface. This paper describes that interface. In addition to its unusual response to incomplete and incorrect constructions entered by the user, the interface is "mode-free", unusually frugal in the size of the command set, and unusually consistent in its treatment of statements, immediate statements, and commands.

This paper is organized into the following sections:

1. Overview
2. The Display Screens
3. The PL/CS Language
4. Commands
5. The File System
6. Program Entry
7. Program Execution
8. Keyboard Layouts
9. References

1. Overview

A user session with CORE consists of a sequence of distinct "actions". Each action consists of a "command" given by the user, followed by an appropriate response by the system. The response performs the task specified by the user command, or explains why the command is unsuitable. When the response to a command is completed, the system returns control to the user for entry of the next command. In the case of the <EXECUTE> command, since the response may be of long duration, the user can terminate the response just by beginning the entry of the next command.

This process continues until the <QUIT> command is given, which terminates the session. The next session automatically begins in exactly the state that existed when the previous session was terminated.

There are only twenty-three different commands, including cursor motion (<LEFT>, <RIGHT>, (<LEFT END>, <WORD TAB>, etc.) and editing commands (<ERASE>, <CLEAR>, <BACK PAGE>, <EXPAND>, etc.)

Many commands allow a string of characters to be given as an argument; one command requires such an argument. The argument precedes the command key; in effect, the command key specifies what is to be done with the characters that have just been entered.

There is a clear distinction between commands and "statements". Statements are the underlying objects of the system. They can be executed or they can be saved -- presumably for execution later. Commands are the imperatives that control the actions of the system; that is, control the saving and execution of statements. Commands take effect immediately -- there is no provision by which a command can be saved in a file. All commands are entered by means of special keys, rather than by keywords or English-like sentences.

There are two different forms of display called the "edit screen" and the "execution screen". One or the other of these is always on the terminal screen, the choice depending on what command was last given. In general, the execution screen appears while statements are being executed, and the edit screen appears when some file is being explicitly changed by the user. The shift from one screen to the other is handled implicitly by the various commands.

Every command is always available to the user. There are no "modes" or "levels" in the system that partition the set of commands. There are no special commands whose sole function is to change mode.

CORE automatically formats (indents) the lines of a program to display its logical control structure. Systematic indentation makes a program easier to read and understand, but also indentation in CORE is important in the interpretation of various commands. Commands deal with program "units", rather than lines or statements. In the case of simple statements such as assignment, GET, PUT, etc., the unit corresponds to a single statement. But for complex statements such as loops and conditionals, the unit consists of the initiating statement and _all lines indented with respect to that statement_. For example, consider the following segment:

```
T = X(1);
L1: DO I = 1 TO N-1 BY 1;
    X(I) = X(I+1);
    END L1;
X(N) = T;
```

At the highest level shown in this segment there are three units. The three lines of the loop are all part of a single unit. If, for example, the <REPLACE> command were given when the edit-pointer is pointing to the T = X(1) line, that single line would be replaced (by whatever is in the entry area of the screen). But if the edit-pointer were pointing at the L1: DO line when the <REPLACE> command is given, then all three lines of that unit would be replaced. At the next lower level, there is a single unit inside the loop. If the edit-pointer were pointing at this X(I) = line, then that single line would be replaced.

The user has the option of compressing the display of any individual unit -- presumably to reduse screen space for sections of the program that are not of immediate interest. The <COMPRESS> command causes all the lines of a unit to be strung out on a single line. The normal indented, multi-line display is restored by <EXPAND>.

2. The Display Screens

There are two kinds of display screens:

the execution screen
: displayed while the system is executing statements, and while it is paused in execution

the edit screen
: displayed while the user is explicitly altering or examining the contents of some file.

One or the other of these screens is displayed on the terminal screen at all times.

The format of the edit screen is the following:

```
 ____________________________________________________________
| entry area                                                 |
|____________________________________________________________|
| previous command area                    | title area      |
|__________________________________________|                 |
| message area                             |                 |
|__________________________________________|_________________|
| text area                                                  |
|                                                            |
```

The title area identifies the file being changed, and indicates the file-type.

The entry area echoes the user's keyboard input.
The entry cursor moves horizontally in the entry area to indicate the position in which the next character will be inserted. It is controlled by commands such as <LEFT>, <RIGHT>, <LEFT END>, <RIGHT END>, <WORD TAB>, and <STMT TAB>.

The previous command area displays the last command entered, with its entry argument. (The command displayed here may be undone by the <UNDO> command.)

The message area displays messages from the system:
error messages
prompting messages
confirmation messages

The text area displays a segment of the file being modified.
At the left margin of the text area is the edit pointer (edit-ptr) identifying one line of the text area for certain editing commands. The text is automatically scrolled to keep the edit-ptr well-positioned in the text area. Scrolling is by half-pages: when the edit-ptr is moved off the bottom of the area the contents are scrolled up one-half page so the edit-ptr is in the middle of the area. The edit-ptr can be moved by the <UP>, <DOWN>, <BACK PAGE>, and <FORWARD PAGE> commands.

Immediately to the right of the edit-ptr is the <u>change indicator</u>, where a * appears on each line modified by the previous command.

The normal format for display in the text area is canonically indented. Alternatively, the user can <CONDENSE> a section so that all the lines that would normally be indented with respect to a given line are concatenated to that line. However, for display in the text-area all statements after the first on any line are replaced by an ellipsis ("...").

If the length of a line exceeds the width of the text area the display of that line is simply truncated. (It is not continued on the next line.) To see the right part of a long line, <FETCH> the line to the entry area where it can be scrolled left and right.

The execution screen is formatted as follows:

```
 ______________________________________________________________________
| entry area                                                           |
|______________________________________________________________________|
| previous command area                          | stack area          |
|________________________________________________|                     |
| message area                                   |                     |
|______________________________________________________________________|
| trace area                                 | check area              |
|                                            |                         |
|                                            |                         |
|______________________________________________________________________|
| output area                                                          |
|                                                                      |
```

The entry, previous command, and message areas are the same as on the edit screen.

The stack area displays the top of the execution environment stack -- the procedures currently being executed. The rightmost procedure on this list is called the "current procedure".

For example, the stack might show

P1 -> P2 -> P3

which would indicate that P1 is being executed as a main procedure; P2 was called from P1; P3 was called from P2 and is currently being executed.

If the procedure stack exceeds the capacity of the stack area, only the top of the stack is displayed.

The trace area shows a somewhat abbreviated version of the text of the current procedure.

Only executable lines are shown: PROC, DCL, END are omitted. Lines are truncated at the right margin of the trace area; no continuations are shown.

Statements are shown only to the indentation level established by the TRACE statement in the current procedure.

Individual units will be displayed in either normal (indented) or condensed (single line) form, depending on previous actions with <CONDENSE> and <EXPAND> commands.

At the left margin of the trace area is the "execution pointer" (exec-ptr), indicating what statement is currently being executed. When execution is paused, the exec-ptr indicates the first statement that will be executed when execution is resumed. The exec-ptr can be moved by means of the <UP> and <DOWN> commands, which are similar in effect to an "immediate GOTO".

If TRACE(0) is executed in a called procedure, tracing of that procedure ceases (or may never begin). The trace area then shows the calling procedure with the exec-ptr on the calling statement (unless, of course, the calling procedure is also TRACE(0)).

The check area records each assignment of value to a variable.

The form is:

```
X = 20.7
STR = 'new value'
```

Both names and values may be truncated to fit in the check area.

If a variable receiving a new value is already listed in the check area, the value is changed. If the variable is not already listed, it is added to the bottom of the list if there is room. If the check area is already full, the newly assigned variable replaces the variable in the area to which a value was <u>least recently assigned</u>.

The variables participating in this checking can be controlled with the NOCHECK statement.

Checking is subordinate to tracing in the sense that TRACE(0) suppresses checking as well as tracing. The procedure being checked is always the same as the procedure being traced.

The output area displays the output generated by execution of the PUT statements without a FILE phrase.

When the output area is filled and the program needs to write another line, execution is paused for "page turn". The <EXECUTE> command (with null entry) clears the output area and resumes execution.

The output area can be cleared under program control by the PAGE option in a PUT statement.

The size of the output area can be modified by using the PAGE option. The space on the display not claimed by the output area is automatically included in the trace/check area.

3. The PL/CS Language

This first version of CORE supports program development in a slightly modified dialect of the PL/CS langunge. PL/CS is a carefully disciplined subset of PL/I. "Disciplined" in this context means that the usage of the constructs included in PL/CS is also restricted in many cases, relative to the comparable construct in PL/I. For example, in PL/CS functions have no side-effects whatever, and GOTO statements only permit forward references. In effect, these restrictions enforce usage that is generally considered good programming practice. The usage required in PL/CS is allowable in PL/I, so the language remains upward compatible with PL/I. (PL/CS includes an ASSERT statement and a READONLY attribute that are not part of PL/I. A program that uses either feature will not be upward compatible.)

PL/CS is a very small subset -- easily learned and used. It is quite comparable to BASIC in this regard. It was intended for introductory instruction but, like BASIC, can be used for significant programming tasks.

There are two previous implementations of PL/CS. The first was a highly diagnostic batch processing system for 370-compatible machines. The second was the Cornell Program Synthesizer, an interactive development system quite comparable to CORE.

The modifications of batch PL/CS that have been made for CORE are the following:

1. enhancement of the file processing capability,

2. elimination of ON ENDFILE, which is unnecessary with the file processing enhancements, and

3. the addition of statements to facilitate interactive execution.

The statements of CORE-PL/CS are briefly described below. Complete descriptions are cited in Section 9. Except as noted, the syntax and semantics of statements are those of PL/I.

Procedures
1. All procedures are external; no nesting of definitions.
2. All procedures are recursive.
3. Any procedure is executable as the "main" procedure of a program.
4. Function procedures can have no side effects.
   They cannot contain GET, PUT or CALL statements, or assign value to any STATIC or EXTERNAL variable.

Declarations

DECLARE (name-list) attributes;

For array declaration: name(lowerbound:upperbound, ...)

Attributes are:

FIXED

FLOAT

The default type is FLOAT, regardless of identifier spelling.

CHARACTER (length) VARYING

```
        The default length is 80.
    BIT(1)
    STATIC
    EXTERNAL
    INITIAL (value-list)
    READONLY
        This is not a standard PL/I attribute.
        It prevents the variable from being the target of assignment.
```

Variables and parameters cannot appear in the same declaration.

All variables and parameters must be explicitly declared.

Declarations appear at the beginning of a procedure, before any executable statement.

Statements

```
Assignment
   variable = expression;
          Logical expressions must be parenthesised.
   array = constant or array;
   SUBSTR( ... ) = expression;

IF (condition)
    THEN s1;
    [ELSE s2;]
          where s1 and s2 are either simple statements or DO; ... END;
          compound statements

          The condition must be parenthesized.

select-name: SELECT;
    WHEN (condition) s1;
       ...
    OTHERWISE sn;
    END select-name;
          OTHERWISE and at least one WHEN are required

          All conditions must be parenthesized.

          The select-name is required.

loop-name: DO control;
       ...
    END loop-name;
          Control is one of following:
             WHILE (condition)
             UNTIL (condition)
             index = e1 TO e2 BY e3

          Control phrases cannot be combined.

          After an indexed loop, the index-variable is uninitialized.
```

All conditions must be parenthesized.

The loop-name is required.

LEAVE loop-name;
Explicit specification of the loop-name is required.

GOTO target-label;
...
target-label:;
Forward reference only.
Target-label is only on a null statement.

CALL procedure-name [(argument-list)];

RETURN [(expression)];

GET [FILE(filename)] [NEXT] LIST|EDIT(...);
The FILE and NEXT phrases are not included in the standard PL/CS language, and are not available in either of the other PL/CS implementations. They are included in CORE-PL/CS to permit multiple input files. Neither phrase is compatible with PL/I. See Section 5.

PUT [SKIP(expr)] [PAGE(expr)] [FILE(filename)] LIST|EDIT(...);
The FILE and PAGE phrases are not included in the standard PL/CS language, and not available in either of the other PL/CS implementations. The FILE phrase permits multiple output files. The FILE and PAGE phrases are not compatible with PL/I. See Section 5.

The PAGE phrase controls the output area of the execution screen:

PAGE(n) assigns n lines to the output area, and clears the area.

PAGE clears the output area without changing its size.

ASSERT (condition) [quantifier];
Quantifier is:
FOR ALL index = e1 TO e2 BY e3
FOR SOME index = e1 TO e2 BY e3

This is not a standard PL/I statement. It is a diagnostic statement used to monitor the truth of the assertion during program execution. The result of an unsatisfied assertion is a pause in execution with an appropriate message in the message area of the execution screen.

Comments
Ordinary comments can appear only between statements or between the names in a declaration.

"Statement comments", denoted by /** ... */ are used to describe the

Many of the commands allow an argument (the <REPLACE> command requires an argument). An argument is a string of characters typed on the normal alpha-numeric portion of the keyboard. The argument is always entered before the command. The argument of a command that allows an argument is the content of the entry area at the time the command is given. The system response to such commands includes clearing the entry area in preparation for receipt of the next command.

On the other hand, commands that do not allow an argument (for example, <UP>, <ERASE>, <EXPAND>, etc.) do not use or affect the entry area. The content of the entry area remains intact as an argument for the next command.

The following notation and terminology is used in the description:

entry	the character-string argument
[entry]	an optional argument
exec-ptr	the execution pointer on the execution screen
edit-ptr	the edit pointer on the edit screen
cursor	the character pointer in the entry area

The commands are the following:

[entry] <EXECUTE>
Switch to execution screen, if not already on screen.

If entry is a file-name:
Clear .OUTPUT file.
Reset pointer in .DATA file to beginning.
Begin execution of specified file (as a main procedure).
If entry is null:
Resume execution of current procedure at exec-ptr.
If entry is present, but not a file-name:
Execute entry as (an) immediate statement(s).

[entry] <FILE>
If entry is null:
Display edit screen for file of current procedure.
Set edit-ptr to position of exec-ptr.
If entry is a file-name:
Switch to edit screen for specified file.

If this file is the current procedure, set edit-ptr to position of exec-ptr. Otherwise, edit-ptr is preserved from last edit of this file.
If entry is present, but not a file-name:
If necessary, switch to edit screen (for current procedure, with edit-ptr set from exec-ptr).

Insert entry after edit-ptr.

[entry] <ENTER>
If exec-screen is on display, and execution is paused for data input:
Append entry to end of whatever input file caused the pause. Execution remains "paused for data input", so this command can be

repeated.
If exec-screen is on display, and execution is <u>not</u> paused for data input:
Exactly equivalent to <EXECUTE>.
If edit-screen is on display:
Exactly equivalent to <FILE>.

<QUIT>
Preserve system state, and terminate session.

<UP>
Move exec-ptr (for execution screen) or edit-ptr (for edit screen) up 1 line. Scroll display if necessary.

<DOWN>
Move exec-ptr (for execution screen) or edit-ptr (for edit screen) down 1 lines. Scroll display if necessary.

<BACK PAGE>
Switch to edit screen if necessary, setting edit-ptr from exec-ptr. Then move edit-ptr back (toward beginning of file) by n lines (where n is the number of lines in the text-area of the edit screen). This command does not alter the position of the exec-ptr.

<FORWARD PAGE>
Switch to edit screen if necessary, setting edit-ptr from exec-ptr. Then move edit-ptr forward (toward end of file) by n lines; (where n is the number of lines in the text-area of the edit screen). This command does not alter the position of the exec-ptr.

<LEFT>
Move cursor one position left in entry area. Scroll entry area right if necessary.

In moving the cursor a "prompt" (see Section 6) is considered a single character.

<RIGHT>
Move cursor one position right in entry area. Scroll entry area left if necessary.

In moving the cursor a "prompt" (see Section 6) is considered a single character.

<LEFT END>
Move cursor to left end of entry area. Scroll entry area right if necessary.

<RIGHT END>
Move cursor to position following rightmost character in entry area. Scroll entry area left if necessary.

<CLEAR>
Clear character at cursor position in entry area, drawing all right-side characters left one position.

5. The File System

The same file system is used in CORE to store procedures, data, and the results of execution. The same commands are used to create, display and change any of these files.

5.1 File Names

File names in CORE are normal identifiers (letters and digits, starting with a letter) except that a period is added as a prefix. File names are global and this distinctive prefix prevents ambiguity between file names and other identifiers used in procedures.

In some contexts, such as the FILE phrase of GET and PUT statements where an identifier is obviously a file name, CORE will forgive omission of the period prefix. But in general, the distinction is important and the user should supply the prefix period for all file names. For example, consider the entry

p <FILE>

If P is interpreted as a file name, this means "switch to edit screen for file .P, creating a new file .P if such a file does not already exist". On the other hand, if P is not a file name, this means "insert a new line in the current file, based on the entry 'p'". This entry will be repaired to

PUT LIST(P);

and if a declaration for P does not already exist, such a declaration is generated. In short, the two commands

.p <FILE>

p <FILE>

are both always meaningful and valid, but each has an entirely different meaning. Obviously, specification of the period prefix is crucial and the CORE editor cannot supply or discard it in this context.

Essentially the same situation exists with the <EXECUTE> command. Consider the command

.p <EXECUTE>

Assuming .P is a procedure file, this command means "start execution of a new program with procedure P as the main procedure. This implies clearing the previous environment, clearing the .OUTPUT file, resetting the pointer in the .DATA file, etc. On the other hand, the command

p <EXECUTE>

is an immediate execution command. If P is a variable, this is repaired to

PUT LIST(P);

and executed immediately. If P is a procedure, the entry is repaired to

CALL P;

and executed immediately. In neither case is the current environment changed. So, again it is critical whether or not the period prefix is given.

5.2 File Types

At least from the user's point of view there are only two types of file in CORE: procedure files and text files. Every procedure file contains one PL/CS procedure. Entries to procedure files are controlled by a "procedure syntax editor" (PSE) that ensures that a syntactically and structurally correct procedure is constructed.

Text files are used for everything except procedures. Entry into text files is essentially undisciplined -- in particular, it is not controlled by the PSE.

A file becomes a procedure file by having the keyword PROC (or PROCEDURE) appear in the first entry after it is created. If PROC appears, the file type is committed, and the PSE is invoked for all entries to the file. If not, the file becomes a text file, and the entries are unconstrained.

One can store PL/CS statements in a text file, but then they are simply raw text and are neither checked nor formatted by the PSE. They could subsequently be copied into a procedure file. For example, suppose there is such a text file named .T containing a segment of a procedure. One could create a new procedure file named .R, and copy .T into .R by the following commands:

```
.r <FILE>
proc <ENTER>
.t <COPY>
```

The <COPY> command streams the text from .T through the PSE, so it is checked, repaired, and formatted as it enters .R just as if it was arriving from the terminal. This can be useful for dealing with small segments of a procedure, but it is not recommended as the normal mode of program development.

This ability to store a segment of a procedure in a text file is used for immediate statements in CORE. Every entry made for immediate execution is also automatically saved in file .TEMP as a text file. As such it can be re-executed, changed, streamed into a procedure file, etc.

5.3 Standard Files

CORE uses five standard files:

.DATA -- normal input for GET statements

.OUTPUT -- normal output from PUT statements

<UP> or <DOWN> to reposition the edit-ptr
<COPY>

In between the <MOVE> and the <COPY> in this sequence, the segment could be altered by editing the .TEMP file.

5.5 Multiple Input and Output Files

The PL/CS language has been extended in CORE to allow multiple input files, multiple output files, and convenient access to particular items in these files. The additions to the language for this purpose are simple compared to their PL/I counterparts, and are not compatible with PL/I.

The additions consist of:

1. the FILE phrase in the PUT statement,
2. the FILE and NEXT phrases in the GET statement,
3. the DELETE statement, and
4. six functions and pseudo-variables:
   REC, REMAIN, KEY, FIND, COUNT and ITEM.

These elements provide a powerful, yet simple facility to manage files in either sequential or direct access mode.

A CORE file is a sequence of records. Each record is numbered by its sequential position within its file. The number of a record changes automatically whenever another record is added to or deleted from a lower-numbered position in the file. Optionally, each record can also have a "key value", which must be unique within that particular file. The key-value of a record cannot be changed (although the same record contents can be rewritten in a different record with a different key value).

A record is a sequence of items. Each item is a distinct object in PL/I "list format". (If an "edit format" output statement is used to write an item, all of the elements in the edit list form a single item in the record.) The number of items may vary from record to record.

Each file has a "current record pointer" (CRP), and within the record to which the CRP points there is a "current item pointer" (CIP).

This generalized file organization has the following properties:

1. Some records in a file can be keyed, others in the same file need not have key values. Records can be retrieved directly by key value.

2. Records are automatically maintained in order so that the key values of keyed records form an increasing sequence.

3. Record numbers are sequential, and automatically maintained as records are inserted or deleted. Any record can be directly accessed by its current number, but note that these numbers change from time to time with file activity.

4. Records can be inserted at, deleted from, or modified at any position in the file.

5. Records can be accessed in "sequential" or "serial" processing, without regard to numbers or key values.

The use of the file system depends on a special set of functions and pseudo-variables (which are not compatible with PL/I). These are the following:

REC(filename) (function and pseudo-variable)
Function returns the number of the CRP record (from 1 to the size of the file). 0 is returned if the file is unreferenced; -1 if the file is undefined.

Assignment of a number to the pseudo-variable caused the CRP to be set to the record with that number. Positive numbers can be assigned that arbitrarily extend the file, but any negative number is an error.

REMAIN(filename) (function only)
Function returns the number of records until the end-of-file is encountered. 0 indicates that the CRP record is the last. The total number of records is returned for an unread file. -1 is returned for an undefined file. If file .F exists, then REMAIN(.F) + REC(.F) is always equal to the number of records in .F.

Since REMAIN predicts the end-of-file, the ENDFILE condition of batch PL/CS is not necessary (and is not included) in CORE-PL/CS.

KEY(filename) (function and pseudo-variable)
Function returns the key value (as a character string) for the CRP record. If the CRP record is unkeyed, KEY returns the key value of the most closely preceding keyed record (or 'LOW' if no preceding keyed record exists).

Assignment to the pseudo-variable sets the CRP to the record with the assigned key value. If the assigned key value did not previously exist in the file, a new null record with that key value is created.

FIND(filename, keyvalue) (function only)
Function sets the CRP at the first record whose key value is greater than or equal to the keyvalue argument, and returns the value of that key. If no suitable key value exists, 'HIGH' is returned, and CRP is set to end-of-file.

COUNT(filename) (function and pseudo-variable)
Returns the number of items in the CRP record.

Assignment to the pseudo-variable <u>changes</u> the number of items in the CRP record -- either truncating the record, or extending it with null items.

ITEM(filename) (function and pseudo-variable)
Returns the item number of the CIP item in the CRP record. (Items are numbered from 1 in each record.)

A new file is created by giving the <FILE> command with the new file-name as argument. For example, to create a new file named .P

```
            .p <FILE>
```

The edit screen is displayed, with the title indicating that a new file .P is being edited. The text area is blank.

The file .P becomes a procedure file if the first insertion includes the keyword PROC (or PROCEDURE). The type thus established is a permanent characteristic of the .P file. For example, if the command is

```
            proc <ENTER>
```

the text area of the edit screen will show the following:

```
->* P: PROC;
  *    END P;
```

Note the following:

1. The procedure name is the same as the file name, but without the identifying prefix period.

2. The result is given in upper-case letters, although the input was given in lower-case. All input except string values and comments is automatically translated to upper-case for entry into a procedure file.

3. The edit-ptr is positioned on the PROC line, indicating that this is the current insertion point in the file. The next insertion entry will follow this line.

4. Both lines are marked with an asterisk, indicating that they were changed by the previous command.

Note also that the <FILE> command could have been used instead of <ENTER>. When the edit screen is being displayed <ENTER> is equivalent to <FILE>.

CORE allows the user a great deal of flexibility in the manner in which a procedure is developed. At one extreme, the user can first write his procedure out completely on paper and then copy the text, line by line, onto the terminal. Although this is the traditional way of entering a program, it does not exploit the capability of a structured development system like CORE.

At the other extreme, the user can construct his procedure interactively, in a strict "top-down" development, by entering only various distinctive keywords and allowing the system to supply whatever is implied. For example, if the next entry in P is

```
            while <ENTER>
```

the response shown in the text area is the following:

```
         P: PROC;
  ->*       W1: DO WHILE cond;
    *           END W1;
            END P;
```

The word "cond" in the response is called a prompt. Prompts are always displayed in lower-case letters. A prompt indicates that the construction is incomplete, and that an element of the specified kind must be supplied -- although not necessarily immediately. The user can supply the prompted element at once, or can procede to develop another part of the program first and then return to complete this aspect of the procedure. A partially complete procedure still containing prompts can be executed but each prompt is a barrier to execution. Every time a prompt is encountered execution pauses and cannot proceed until the prompt is replaced by a suitable element.

In the example above, the user has, in effect, prompted the system to construct a loop by entering the keyword "while". The system constructed all that was implied by this keyword, and in turn prompted the user for any elements that are necessary to complete the construction.

At this point the user has many choices. He can supply the required element immediately, or he can go on to some other aspect of the development and fill this in later. Assuming he chooses to fill in the condition at once, and if his next entry is acceptable as a condition, it will replace the condition prompt on the edit-ptr line. For example, the entry:

```
                a < b <ENTER>
```

results in the following display:

```
         P: PROC;
  ->*       W1: DO WHILE (A < B);
                END W1;
            END P;
```

However, if the entry was not acceptable as a condition, it would automatically be considered as the entry of the next statement and would be inserted in the body of the loop. For example, suppose instead of "a < b" the following entry had been given:

```
                call <ENTER>
```

The result would be the following:

```
         P: PROC;
            W1: DO WHILE cond;
  ->*           CALL proc;
```

initially. The entry

```
shift do i = 1 BY 2 <ENTER>
```

would produce the line

```
SHIFT: DO I = 1 TO expr BY 2;
```

2. A declaration without specified type attribute yields the default type. For example, the entry

```
dcl x <ENTER>
```

produces

```
DCL (X) FLOAT;
```

The FLOAT attribute can subsequently be changed, but if another type is intended it would have been easier to specify the correct type initially. For example, the entry

```
x fixed <ENTER>
```

would produce

```
DCL ( X ) FIXED;
```

3. The default string length of 80 is generated in the absence of initial length specification. For example, the entry

```
s 10 char <ENTER>
```

produces

```
DCL ( S(1:10) ) CHAR(80) VAR;
```

Each of these generated elements is intended to simplify program entry by eliminating a prompt-response cycle in cases where there is a plausible default. However, in each case the user can avoid having the system generate elements simply by specifying his choice with the initial entry.

Any line in a procedure can be changed by moving the line into the entry area with the <FETCH> command, making the changes and then returning the line to the procedure with the <REPLACE> command. The <WORD TAB> command is useful in positioning the cursor in the entry area to make such changes. If the line includes several statements (because of condensed format), the <STMT TAB> command moves the cursor from one statement to the next. If the changes involve replacing a prompt with a specific element, recall that the entire prompt is treated as a single character by the <LEFT>, <RIGHT>, and <CLEAR> commands. When the changed line is replaced in the procedure, it is completely rechecked by the PSE just as if it were a brand new entry.

The examples above illustrate that the ENDs in a procedure are all <u>generated automatically</u> by entries such as PROC, WHILE or SELECT. The user need never enter an END explicitly, and in fact, cannot do so. However, the entry

```
end <ENTER>
```

has special significance and is very useful. It causes no change in the procedure, but simply repositions the edit-ptr to the END line of the containing unit. This provides a convenient way of escaping from the body of a loop, so that the next insertion follows the loop, rather than be contained in it.

For example, consider the following sequence of entries:

```
to <ENTER>
when <ENTER>
end; end <ENTER>
get p; p <ENTER>
```

The resulting program segment would be the following:

```
     L1: DO index = expr TO expr BY 1;
        S1: SELECT;
           WHEN cond ;
           OTHERWISE;
           END S1;
        END L1;
     GET LIST( P );
->*  PUT LIST( P );
```

This treatment of end entries allows a complete PL/CS program to be entered directly, in spite of the fact that END lines are generated automatically by the PSE. This allows a program to be written out in advance, in the traditional manner, and then entered as continuous text without requiring any changes to accomodate CORE. More importantly, it allows text files to be given as input to the PSE. This permits communication with other development systems, and also allows programs to write files that are program text.

6.1 Entry of Immediate Statements

Very little need be said about the entry of immediate statements, since almost all of the section above applies directly. The distinction is simply that the <EXECUTE> command is given instead of <FILE>. The entry is still corrected and expanded in the same way, but the result is executed immediately rather than inserted in some file.

Note that the execution takes place in the environment of the procedure whose execution is paused, and not in the procedure being edited if those happen to be different procedures.

In fact, an entry for immediate execution is also saved in the special file named .TEMP. This means that the user can subsequently examine the entry (by editing .TEMP), change it, and re-execute it. It is important to understand that the storage of such an entry in .TEMP does not make a procedure file out of .TEMP.

pause.

Users with experience on other interactive systems should note the absence of a "break" or "interrupt" command. Such a command would be redundant since every CORE command is capable of interrupting execution.

During a pause in execution the user can:

1. Edit any file.
   When the file containing one of the procedures currently being executed is changed, the ability to resume execution may be impaired.

   Except when one of the program's input files is augmented, the display will automatically change to an edit screen for the file being edited.

   The standard .DATA and .OUTPUT files can be edited with the normal editing commands, but these do not allow repositioning of the read/write pointers in these files. (See Section 5.)

   When the file of the current procedure is edited, the edit-ptr is initially positioned to coincide with the exec-ptr. However, the converse is not done -- on resuming execution the exec-ptr has not been moved by the editing commands.

2. Execute immediate statements.
   Almost any PL/CS statement (see Section 3.2) can be executed "immediately" in the current execution environment. This can alter the values of variables, move the exec-ptr, produce output, read input, call other procedures.

   The most common immediate statements are PUT and assignment. Note that both are subject to normal scope rules: they can only access variables in the current environment.

   The CORE repair facilities are particularly convenient for immediate PUT statements since a variable name alone is automatically "repaired" to a PUT statement. For example, the entry

   ```
   x <EXECUTE>
   ```

   is equivalent to

   ```
   PUT LIST( X );
   ```

3. Perform other commands.
   <CONDENSE>|<EXPAND> to change the format of display
   <UP>|<DOWN> to move the exec-ptr
   This is, in effect, a generalized GOTO. It allows backward as well as forward jumps, and allows jumps to arbitrary statements rather than just labelled nulls.
   <BACKPAGE>|<FWDPAGE> to examine other sections of the procedure
   These commands automatically switch the display to the edit screen. This does not move the exec-ptr; when execution resumes the execution screen is restored with the trace area restored to

the locale of the exec-ptr.
<UNDO> to undo the effect of the previous command
<QUIT> to terminate the session.

After the completion of execution of a program these same actions can be taken, except of course, that execution cannot be resumed. (Consequently moving the exec-ptr is pointless.)

There are a few situations in which execution can be interrupted in the middle of a statement, rather than between statements:

1. pause for page turn
2. pause for data input
3. execution error

When paused for page turn, execution resumes exactly where it was interrupted, completing execution of the PUT statement that overflowed the output area of the execution screen. (To avoid completing the PUT statement move the exec-ptr ahead before resuming.) In every other case execution resumes by repeating from the beginning the execution of the statement that was partially executed.

7.2 Testing Facilities

Unfortunately most programs contain errors as initially written, in spite of careful development by the user, and enthusiastic error-repair by CORE. Consequently, thorough testing is still required before a program can be convincingly claimed to be correct.

CORE has many helpful facilities for testing. Moreover, since a development system is primarily concerned with new programs, the default state for these facilities is enabled. That is, the testing facilities are automatically active until explicitly disabled. Thus neither effort nor knowledge on the part of the user is required to exercise these diagnostic facilities.

Specifically, these default facilities are the following:

1. Procedures are traced to 2 levels of indentation.

2. All variables are checked.

3. The system is slowed (at least one-half second between screen draws) so the progress of execution can be visually tracked.

During testing, the user can alter these actions by means of immediate TRACE, NOCHECK and SLOW statements. When testing of a procedure is complete, and the user is satisfied as to its correctness, these facilities can be disabled by including TRACE, NOCHECK and SLOW statements in the body of the procedure.

Note that the TRACE, NOCHECK and SLOW statements are completely local -- they affect only the treatment of the procedure in which they appear (either stored or immediate). This facilitates testing on a procedure-by-procedure basis.

Note also that incomplete procedures can be executed for testing. Execution will proceed until a missing element is encountered.

8. Keyboard Layouts

8.1 The assignment of command keys on a Datamedia keyboard is shown below. Only the special keys are shown. The lower-case lettters denote the key label; the upper-case letters give the CORE assignment. When two assignments are shown, the top one is the key by itself; the bottom one is home/key.

esc	eol	eos	clear	printoff	up	print
QUIT	UNDO	FETCH	CLEAR	CONDENSE	UP	MOVE
		REPLACE	ERASE	EXPAND	BACKPAGE	COPY
tab			return	left	home	right
WORD-TAB			ENTER	LEFT		RIGHT
STMT-TAB				LEFT-END		RIGHT-END
			delete		down	
			EXECUTE		DOWN	
					FWD-PAGE	
			break			
			FILE			

8.2 The key assignments for the Terak keyboard are shown below:

esc	tab	backspace	up			
QUIT	WORD-TAB	EXECUTE	UP			
	STMT-TAB		BACKPAGE			
linefeed		return	down			
UNDO		ENTER	DOWN			
			FWDPAGE			
dc3			left	dc1	dc2	ua
FETCH			LEFT	CDENSE	MOVE	FILE
REPLACE			LEFT-END	EXPAND	COPY	
			right	home		delete
			RIGHT			CLEAR
			RT-END			ERASE

9. References

1. Conway, R. W. and R. L. Constable, "PL/CS -- A Disciplined Subset of PL/I", Technical Report 76-293, Department of Computer Science, Cornell.

2. Conway, R. W., A Primer on Disciplined Programming, Winthrop Publishing Co., 1978.

3. Conway, R. W. and D. Gries, Intro to Programming (3rd ed), Winthrop Publishing Co., 1979.

4. Teitelbaum, R. T., "The Cornell Program Synthesizer, a Micro-Processor Implementation of PL/CS", Technical Report 79-370, Department of Computer Science, Cornell.

5. Archer, James, Jr., "A File System Extension to PL/CS", Technical Report 79-366, Department of Computer Science, Cornell.