THE CORNELL PROGRAM SYNTHESIZER:
A MICROCOMPUTER IMPLEMENTATION OF PL/CS

by

Tim Teitelbaum*

TR 79-370

Revised June 1, 1979

Department of Computer Science
Cornell University
Ithaca, New York 14853

*The research described in this report was supported in part by the National Science Foundation under Grant MCS 77-08198.

The Cornell Program Synthesizer:
A Microcomputer Implementation of PL/CS

Tim Teitelbaum

Department of Computer Science
Cornell University

1. Introduction

The Cornell Program Synthesizer is a system for developing structured programs at a video display terminal. The system provides a self-contained programming environment with integrated facilities to edit, file, execute and debug programs.

The principal innovation of the system is its syntax-directed editor: entry and modification of program text is guided by a grammar for the host programming language. Because the parser is incorporated into the editor, it is normally impossible to create a syntactically incorrect program. There is no need for error repair because errors are prevented on entry.

All but the simplest statement types are predefined in the editor as _templates_. The template is a formatted syntactic skeleton that contains the keywords, matched parentheses and other punctuation marks of the given statement form. The template includes "placeholders" at each position where additional code is required to complete the statement. The placeholders serve as prompts indicating the syntactic class of each component required to complete the statement.

Programs are created top-down by inserting new statements and expressions within the skeleton of previously entered templates. Syntax error detection is immediate because placeholders can only be replaced by syntactically appropriate insertions.

The text of a template is never typed; rather, it is inserted into the program by command. Because a template is immutable, errors are not only prevented on entry, but can not be introduced after the fact. The entire statement can be moved or deleted as a unit and its constituent parts may be modified, but the concrete syntax of the statement form is unalterable.

Another significant benefit of the template insertion approach is the sheer reduction of key-strokes; short commands insert long templates. The effect of each template insertion is instantly visible in the program display. It is fast and convenient to move an entire template and all its constituent parts as a single entity.

Expressions and assignment statements are typed directly as text. Since the compiler is invoked by the editor on a phrase-by-phrase basis, errors in user-typed text are detected immediately. Most errors are typographical and can be corrected by local editing of the erroneous segment.

It is possible, however, to override the error prevention mechanism when non-local modification is required to correct the program. Erroneous code remaining in the program is then highlighted on the display until corrected. Such highlighted sections are also introduced when changing a declaration introduces non-local errors in the rest of the program. For example, when a declaration is deleted, all fragments containing uses of undeclared variables are marked as "erroneous" on the display.

The editor's "statement-comment" facility encourages programming by "stepwise refinement"[4]. Subordinate to a statement comment is its refinement: a list of statements that performs the action specified in the comment. By suppressing the display of a refinement, it is possible to conceal irrelevant implementation details while increasing the "field of view" on the screen. Thus, the system rewards the use of meaningful precise comments with a condensed yet readable presentation of the program outline.

Since programs are translated into interpretable form during program entry, compilation is transparent to the user; there is no delay between editing and execution. Execution is suspended when a missing program element is encountered and can be resumed after the required code has been inserted. Thus, program development and testing can be conveniently interleaved.

The video display allows the incorporation of some unique run time debugging features. The flow of execution through the program can be traced using the screen cursor to indicate the location of the instruction pointer at each moment. Selected variables can be monitored during execution by displaying their values in a separate partition of the screen. The utility of these features is enhanced by reducing the rate of execution and by suppressing the display of irrelevant program refinements. A suppressed refinement is always executed at top speed, giving it the appearance of being an indivisible operation.

The Synthesizer is currently implemented for the PDP-11 (under UNIX) and the LSI-11 (stand alone on a TERAK). The initial host programming language is PL/CS, a highly disciplined subset of PL/I [1,2,3]. Although the key power of the Synthesizer is its language-specific editor, the idea of a "language-specific editor" is essentially language independent. The implementation itself is syntax-directed and other languages will surely be incorporated in the near future.

This report provides a tutorial introduction to the use of the Cornell Program Synthesizer to write PL/CS programs.

2. Creating a program

We illustrate the editing process by writing a program to compute the square of a given number. The cursor is indicated by an underscore in the examples below.

Assume that we are already on the system and wish to create the program in the file named "squares". Then after typing the command ".ed squares" followed by <return>, the screen would look as follows:

```
                                   editing   squares
---------------------------------------------
object
```

The system is ready to create a program. Since there is no program yet, the word "object" appears on the screen as a placeholder. As the program is developed, it will be displayed below the dashed line where "object" now appears.

Every PL/CS main procedure has the same structure. It is unnecessary, time-consuming and error-prone to type this information explicitly each time a program is written. Rather, the notion of a PL/CS main procedure is built directly into the editor in the form of a "command".

The command ".main" directs the system to create a main procedure. The dot (.) at the beginning distinguishes editor commands from program text. The command ".main" is not part of the program. Rather, it is a directive to the system to create a main procedure. Because it is not part of the program, ".main" appears on the screen above the dashed line. The cursor remains below the line to indicate your place in the developing program.

After typing the five characters ".main", the screen appears as follows:

```
.main                              editing   squares
---------------------------------------------
object
```

The command has not yet been obeyed. However, upon typing <return>, the system immediately responds:

```
                                    editing   squares
---------------------------------------------
/* comment */
squares: PROCEDURE OPTIONS (MAIN);
    {declaration}
    {statement}
    END squares;
```

In one step, the placeholder "object" is replaced by the complete skeleton of a PL/CS main procedure. The name of the procedure is "squares", inherited from the file name.

Three new placeholders, "comment", "{declaration}" and "{statement}", identify locations where additional program elements can be inserted. The placeholders are descriptive names that serve as prompts. The cursor is positioned at "comment". The braces in the placeholder "{declaration}" indicate that a list of declarations is permitted there. Similarly, "{statement}" identifies a place where a list of statements may occur.

By first repositioning the cursor, any one of the three placeholders could next be expanded. It is simplest just to enter the comment next since the cursor is already positioned there. Only the comment text is required since the delimiters "/*" and "*/" have already been provided.

Any text typed at this point is inserted directly into the program. (The only restriction is that the text must not begin with a dot.) The first character typed replaces the placeholder "comment" whereupon the delimiter "*/" slides left to close the gap. For example, when an "s" is typed the screen responds:

```
                                    editing   squares
---------------------------------------------
/* s_ */
squares: PROCEDURE OPTIONS (MAIN);
    {declaration}
    {statement}
    END squares;
```

As each additional character is typed, the delimiter "*/" slides right to make room.

```
                                  editing   squares
-----------------------------------------------
/* square of an input value_ */
squares: PROCEDURE OPTIONS (MAIN);
    {declaration}
    {statement}
    END squares;
```

A mistake can be erased by typing <rubout>, whereby the "*/" shifts left again. If the entire text is rubbed-out, the placeholder "comment" reappears.

Using <left> and <right> cursor keys, it is possible to position the cursor at any point within a line of text. As characters are inserted or deleted, the entire context to the right of the cursor immediately shifts to accomodate the change. For example, we may position the cursor at the beginning of the comment and insert the phrase "print the":

```
                                  editing   squares
-----------------------------------------------
/* print the_square of an input value */
squares: PROCEDURE OPTIONS (MAIN);
    {declaration}
    {statement}
    END squares;
```

Having completed the comment, the cursor is moved to "{declaration}" by typing <return>. The cursor skips over "squares: PROCEDURE OPTIONS(MAIN)" since it is part of the generated program template and can not be altered.

```
                                  editing   squares
-----------------------------------------------
/* print the square of an input value */
squares: PROCEDURE OPTIONS (MAIN);
    {declaration}
    {statement}
    END squares;
```

Two modes of program entry have been illustrated: template insertion by command and direct insertion of typed text:

> In general, commands are used to insert declarations and statement forms containing lengthy keywords, parenthesized lists or DO ... END. Such commands give the user significant generative power while precluding the possibility of syntax error. The use of a single command to insert an entire construct emphasizes its unified, abstract purpose. The specific syntax used to

represent the construct can not be typed and can not be modified.

In general, direct insertion of typed text is used for comments, assignment statements, and lists of variables and expressions. These program elements consist mainly of English text, user defined symbols and short operators. The use of commands at this level would be counter-productive.

Continuing to develop the sample program, we generate a declaration template for FIXED variables by typing the command ".fx" followed by <return>:

```
                                  editing   squares
      ---------------------------------------------
      /* print the square of an input value */
      squares: PROCEDURE OPTIONS (MAIN);
          DECLARE ( list-of-variables ) FIXED;
          {statement}
          END squares;
```

and by typing "k" followed by <return> we obtain:

```
                                  editing   squares
      ---------------------------------------------
      /* print the square of an input value */
      squares: PROCEDURE OPTIONS (MAIN);
          DECLARE ( k ) FIXED;
          {declaration}
          {statement}
          END squares;
```

Once again the cursor is positioned at "{declaration}". The editor guarantees the syntactic correctness of the program and will only accept a declaration at this point. By simply typing <return>, however, "{declaration}" disappears and the cursor moves to "{statement}":

```
                                  editing   squares
      ---------------------------------------------
      /* print the square of an input value */
      squares: PROCEDURE OPTIONS (MAIN);
          DECLARE ( k ) FIXED;
          {statement}
          END squares;
```

An input statement is generated by typing the command ".g" followed by <return>:

```
                                         editing   squares
        -------------------------------------------------
        /* print the square of an input value */
        squares: PROCEDURE OPTIONS (MAIN);
            DECLARE ( k ) FIXED;
            GET LIST ( list-of-variables );
            END squares;
```

and by typing "k" followed by <return>:

```
                                         editing   squares
        -------------------------------------------------
        /* print the square of an input value */
        squares: PROCEDURE OPTIONS (MAIN);
            DECLARE ( k ) FIXED;
            GET LIST ( k );
            {statement}
            END squares;
```

Typing ".pl" followed by <return> gives the skeleton of an output statement:

```
                                         editing   squares
        -------------------------------------------------
        /* print the square of an input value */
        squares: PROCEDURE OPTIONS (MAIN);
            DECLARE ( k ) FIXED;
            GET LIST ( k );
            PUT LIST ( list-of-expressions );
            END squares;
```

An error in typed text is detected as soon as the user attempts to move the cursor to another program element. For example, suppose we next type "k*kk". Then, as soon as <return> is typed, the bell rings, an error message appears on the top line, and the cursor is positioned at the point of the error:

```
          undeclared variable    editing   squares
        -------------------------------------------------
        /* print the square of an input value */
        squares: PROCEDURE OPTIONS (MAIN);
            DECLARE ( k ) FIXED;
            GET LIST ( k );
            PUT LIST ( k*kk );
            END squares;
```

Typing <clear> erases the extra "k" and corrects the program.

3. Execution

The program can be executed by simply striking the <execute> key. Since compilation into intermediate interpretable code has already taken place during the creation of the program, execution begins immediately. The display of the program disappears and the screen responds:

```
                type input data      executing squares
        ------------------------------------------------

        _
```

We type "2" followed by <return> and the program responds "4". The system then waits for instructions before returning to the editor. This allows the user to review output before it is overwritten by the program display.

Alternatively, by using the command ".split" before program execution, the screen can be split into a program display screen and an execution input/output screen. In this case, the system returns to the editor immediately after execution; the program output remains on the execution screen.

```
                                     editing   squares
        ------------------------------------------------
        /* print the square of an input value */
        squares: PROCEDURE OPTIONS (MAIN);
            DECLARE ( k ) FIXED;
            GET LIST ( k );
            PUT LIST ( k*k );
            END squares;

        ------------------------------------------------
        2
              4
```

Several symbolic debugging aides are available. The command ".trace" causes the flow of execution through the program to be traced using the screen cursor to indicate the location of the instruction pointer at each moment. The command ".pace n" causes the program to run with a delay of n/60 seconds per program segment. The statement "pause;" can be inserted into the program wherever a breakpoint is desired. The command ".check k" reserves a screen position for the value of k. This display is then updated on each assignment to k. "Single cycle" and "structured single cycle" execution is also available.

Incomplete programs can be executed at any stage of their development. Execution is suspended whenever an unexpanded placeholder is encountered. Control returns to the editor with the cursor positioned at the unexpanded placeholder. After the required code has been inserted, execution can be resumed.

4. Moving the cursor

All modifications of program text occur relative to the current cursor position. Using the cursor control keys of the terminal, it is possible to position the cursor wherever insertions and deletions are permitted. The cursor can only be positioned where modifications are allowed.

A program template generated by command is denoted by its left-most character. For example, the "G" in "GET LIST (k);" designates the entire statement. When the cursor is positioned at such a point, editing actions refer to the entire program element. For example, striking the <delete> key at that point would delete the whole statement, replacing it with the placeholder "{statement}".

The cursor control keys move the cursor *forward* and *backward* through the program. For want of better names, we refer to the keys as <left>, <right>, <up> and <down>. Despite this nomenclature, the effect of the control keys is defined with respect to the one-dimensional reading order of a program, not the two-dimensional coordinate system of its display. Thus, both <right> and <down> move the cursor forward through the program; <left> and <up> move it backwards.

Since much of the program text is immutable, the cursor jumps in logical increments, not character by character. Although <right> and <down> both move the cursor forward, their units of increment differ.

<Up> and <down> move the cursor one program element at a time, stopping only once per syntactic segment. Possible stopping points for the cursor using the <up> and <down> keys are indicated by underscores in the sample program below:

```
/* print the square of an input value */
squares: PROCEDURE OPTIONS (MAIN);
    DECLARE ( k ) FIXED;
    GET LIST ( k );
    PUT LIST ( k*k );
    END squares;
```

Cursor stopping points for <up> and <down>

<Left> and <right> differ from <up> and <down> by also stopping at every character of modifiable text. These places are indicated by underscores in the sample program below:

```
/* print the square of an input value  */
squares: PROCEDURE OPTIONS (MAIN);
    DECLARE ( k  ) FIXED;
    GET LIST ( k  );
    PUT LIST ( k*k  );
    END squares;
```

Cursor stopping points for <left> and <right>

Thus, <right> from the "G" of "GET LIST (k);" moves the cursor to "k", then to the position after "k", then to the "P" of "PUT ...". <Left> from the "G" moves the cursor to the position after the "k" in "DECLARE ...", then to the "k", then to the "D" of "DECLARE ...".

Some templates contain optional components; for example, DO-loops contain an optional loop-name. Two mechanisms serve to minimize the visibility of optional language features:

1) The only cursor motion that positions the cursor at an optional component is the command ".o", meaning "move to optional part". Optional statement elements are transparent to all other cursor controls.

2) Placeholders for optional components are only displayed when the cursor is positioned there.

<Return> is also a cursor motion key, similar in function to <down>. It differs, however, by also stopping everywhere that a program element can be inserted into a list: that is, at the beginning, at the end, and inbetween adjacent list elements. A placeholder appears at such a "list insertion point" whenever the cursor is positioned there; it disappears when the cursor is moved away. The possible stopping points for the cursor using <return> are indicated by underscores below. At most one of the placeholders shown would ever be visible at a time.

```
/* print the square of an input value */
squares: PROCEDURE OPTIONS (MAIN);
    {declaration}
    DECLARE ( k ) FIXED;
    {declaration}
    {statement}
    GET LIST ( k );
    {statement}
    PUT LIST ( k*k );
    {statement}
    END squares;
```

Cursor stopping points for <return>

Although the illustrated cursor motions are sufficient for novices, more experienced users will welcome several additional commands particularly in large, deeply nested programs.

The two key sequence <long><down> advances the cursor to the next element not structurally deeper in the program. The sequence <long><up> moves backward similarly. <Long><return> is like <long><down> but also stops at "list insertion points".

Thus, from the "G" of "GET ...", <long><down> move the cursor to the "P" of "PUT ..."; <long><up> moves to the "D" of "DECLARE ..."; <long><return> moves to "{statement}" inbetween "GET ..." and "PUT ...".

The <diagonal-arrow> key moves the cursor to the immediately enclosing program element. Thus, it has the effect of moving the cursor diagonally up and to the left.

5. Modifying a program

Phrases typed by the user can be modified by first repositioning the cursor within the desired line. Characters can then be deleted or inserted in place. <Rubout> erases the character to the left of the cursor, <clear> erases the character at the cursor. Of course, the syntactic correctness of re-edited text must be re-verified.

A single phrase or an entire template and its subordinate parts can be "clipped" or "deleted". The entire section of code disappears from the program and is replaced by the original placeholder.

Clipped code can be reinserted at any syntactically suitable place by repositioning the cursor and pressing <insert>. Thus, <clip> and <insert> are analogous to the functions "move-to-memory" and "insert-from-memory" on a one-register pocket calculator. In point of fact, clipped code is actually moved to an object of the file system named "CLIPPED".

Similarly, <delete> (usually used to permanently erase a code segment) actually moves it to an object of the file system named "DELETED". Thus, deletion is reversible: in the event of an inadvertent deletion, the original segment can be recovered by the command ".ins DELETED".

A separate command is available to clip a list of consecutive statements. The two-key sequence <long><clip> serves this purpose. The user positions the cursor at the first element of the desired sublist and types the command. The user then positions the cursor at the last element of the desired list and completes the command by typing ".".

The clipping mechanism is used to enclose existing code in the scope of a new template. For example, in order to enclose the "GET..." and "PUT..." statements in a loop, one clips the list of statements,

```
                                   editing   squares
---------------------------------------------
/* print the square of an input value */
squares: PROCEDURE OPTIONS (MAIN);
    DECLARE ( k ) FIXED;
    {statements}
    END squares;
```

generates the loop template by command,

```
                                   editing   squares
---------------------------------------------
/* print the square of an input value */
squares: PROCEDURE OPTIONS (MAIN);
    DECLARE ( k ) FIXED;
    DO WHILE ( condition );
        {statement}
        END;
    END squares;
```

enters the condition and inserts the clipped code into the body of the loop. Clipped code is automatically reindented with respect to its new context when reinserted.

```
                                   editing   squares
---------------------------------------------
/* print the square of an input value */
squares: PROCEDURE OPTIONS (MAIN);
    DECLARE ( k ) FIXED;
    DO WHILE ( '1'b );
        GET LIST ( k );
        PUT LIST ( k*k );
        END;
    END squares;
```

Similarly, to enclose the loop in a comment, one clips it, inserts the comment, types the comment and then reinserts the clipped code into the scope of the comment.

```
                              editing   squares
    ---------------------------------------------
    /* print the square of an input value */
    squares: PROCEDURE OPTIONS (MAIN);
        DECLARE ( k ) FIXED;
        /* for each input value k, print the square of k */
            DO WHILE ( '1'b );
                GET LIST ( k );
                PUT LIST ( k*k );
                END;
        END squares;
```

<Clip>, <delete> and <insert> are used in concert to delete a statement while preserving one of its subordinate parts. One merely clips the segment to be preserved, deletes the statement to be discarded and reinserts the clipped piece.

The display of code subordinate to a comment can be suppressed by typing the <...> key.

```
                              editing   squares
    ---------------------------------------------
    /* print the square of an input value */
    squares: PROCEDURE OPTIONS (MAIN);
        DECLARE ( k ) FIXED;
        /* for each input value k, print the square of k */
            ...
        END squares;
```

The <...> key serves as a toggle and is thus used to redisplay suppressed text. By suppressing the display of a refinement, irrelevant implementation details are concealed while increasing the "field of view" on the screen. Thus, the system rewards the use of meaningful precise comments with a condensed yet readable presentation of the program outline.

Declarations can be re-edited in the same way as the rest of the program, but such re-editing may be subject to non-local errors. For example, in order to change the declaration of k from FIXED to FLOAT, one first deletes the FIXED declaration. This leads to several segments in the program that contain an undeclared variable. These erroneous segments are highlighted on the screen (using the complemented character set, if available). When the FLOAT declaration for k is inserted, these highlighted areas are redisplayed in the normal font.

6. Ending a session

The command ".off" ends a session. The next time the system is reinvoked, it resumes in exactly the same state as if there had been no logoff.

7. Acknowlegements

It is a pleasure to acknowlege the special role of Thomas Reps who has worked closely with me on the development of the Synthesizer from the beginning. I am deeply indebted to Alan Demers for our many stimulating discussions and for writing the LSI-11 operating system kernel. I am also extremely grateful for the valuable contributions of Richard Conway, Jim Archer, Carl Hauser, Dean Krafft and Ron Olsson.

8. References

[1] Conway, R. and R. Constable, "PL/CS -- A Disciplined Subset of PL/I," Technical Report 76-293, Department of Computer Science, Cornell 1976.

[2] Conway, R., _Primer on Disciplined Programming Using PL/CS_, Winthrop 1978.

[3] Teitelbaum, T., "A Formal Syntax for PL/CS", Technical Report 76-281, Department of Computer Science, Cornell 1976.

[4] Conway, R. and D. Gries, _An Introduction to Programming_, Third Edition, Winthrop, 1979.

Cornell Program Synthesizer -- PL/CS Version

```
-----------------------------------------------------------------

PLACEHOLDER    COMMAND   TEMPLATE
-----------------------------------------------------------------

object         .main     /* comment */
                         file-name: PROCEDURE OPTIONS ( MAIN );
                            {declaration}
                            {statement}
                            END file-name;

               .proc     /* comment */
                         file-name: PROCEDURE ( list-of-parameters );
                             {parameter declaration}
                             {declaration}
                             {statement}
                             RETURN;
                             END file-name;

               .text      file-name: TEXT
                             {text}

               .seg proc-name
                          file-name: SEGMENT OF proc-name
                             {statement}


{declaration}  .fx       DECLARE ( list-of-variables ) FIXED;

               .fl       DECLARE ( list-of-variables ) FLOAT;

               .bt       DECLARE ( list-of-variables ) BIT ( 1 );

               .ch       DECLARE ( list-of-variables )
                            CHARACTER ( expression ) VARYING;

               .c        /** comment */
                            {declaration}


{parameter     .fx       DECLARE ( list-of-parameters ) FIXED;
  declaration}
               .fl       DECLARE ( list-of-parameters ) FLOAT;

               .bt       DECLARE ( list-of-parameters ) BIT ( 1 );

               .ch       DECLARE ( list-of-parameters )
                            CHARACTER ( * ) VARYING;

               .c        /** comment */
                            {parameter declaration}
```

Cornell Program Synthesizer

```
---------------------------------------------------------------------

PLACEHOLDER    COMMAND   TEMPLATE
---------------------------------------------------------------------

 statement     .i        IF ( condition )
    or                      THEN  statement
{statement}                 ELSE  statement

               .it       IF ( condition )
                            THEN  statement

               .s        SELECT;
                            {when-clause}
                            OTHERWISE statement
                            END;

               .d        DO;
                            {statement}
                            END;

               .dw       [loop-name:] DO WHILE ( condition );
                            {statement}
                            END;

               .du       [loop-name:] DO UNTIL ( condition );
                            {statement}
                            END;

               .di       [loop-name:] DO var= exp to exp by exp;
                            {statement}
                            END;

               .p        PUT SKIP LIST ( list-of-expressions );

               .pl       PUT LIST ( list-of-expressions );

               .psl      PUT SKIP(expression) LIST ( list-of-expressions );

               .ps       PUT SKIP(expression);

               .g        GET LIST ( list-of-variables );

               .c        /**  comment  */
                            {statement}


{when-clause}  .w        WHEN ( condition )
                            statement
```

PLACEHOLDER	PHRASE
comment	any sequence of characters not including "*/"
{text}	any sequence of characters
statement or {statement}	variable= expression; (semi-colon optional)
	leave loop-name;
	goto label;
	label:;
	call proc-name(argument list);
	call proc-name;
	pause;
expression or condition	any expression
var = exp to exp by exp	variable= expression to expression
	variable= expression to expression by expression

SYSTEM COMMANDS

<boot-switch>	Restore the system as it was at the time of the last ".off" or ".save" command.
.off	Save the status of the system and terminate session. Destroys previous saved status.
.save	Save the status of the system without terminating session. Destroys previous saved status.
.list	List a directory of all non-empty files.
.ed file	Edit the file named "file". If the parameter "file" is omitted, the file name defaults to "CLIPPED".
.pr	Print the currently edited file on a hard-copy printer.

CURSOR MOTION COMMANDS

<down>	Move to next program element.
<up>	Move to previous program element.
<return>	Move to next program element including points in between list elements.
<right>	Move to the next character.
<left>	Move to the previous character.
<diagonal-arrow>	Move to enclosing program element.
<long><right>	Move to last character of current program element. If already at last character, move to next element.
<long><left>	Move to first character of current program element. If already at first character, move to previous element.
<long><down>	Move to next program element not subordinate to the current program element.
<long><up>	Move to the previous program element not subordinate to the current program element.
<long><return>	Move to the next program element not subordinate to the current element but including points in between list elements.
<long><diagonal-arrow>	Move to the top of the file.
<long><long >	Reposition the program with respect to the screen so cursor is centered.
.o	Move cursor to the optional part of the template.

EDITING COMMANDS

If the parameter "file" is omitted, the file name defaults to "CLIPPED".

<rubout>	Delete the character to the left of the cursor.
<clear>	Delete the character at the cursor.
<delete>	Delete the current program element. Save it as a SEGMENT in the file named "DELETED".
<long><rubout>	Delete the prefix of the current phrase from the beginning up to (but not including) the character at the cursor.
<long><clear>	Delete the suffix of the current phrase from the character at the cursor to the end.
.mv file	Move the current program element to "file" after first saving the previous contents of "file" in "DELETED".
.ml file	Move a list of program elements beginning with the current element to "file" after first saving the previous contents of "file" in "DELETED".
<clip>	Same as ".mv CLIPPED".
<long><clip>	Same as ".ml CLIPPED".
.ins file	Remove the object contained in "file" and insert it at the current position of the cursor.
<insert>	Same as ".ins CLIPPED"
<...>	Suppress the display of code subordinate to the immediately enclosing statement comment. If it is already suppressed, then display it.

EXECUTION COMMANDS

<execute>	Execute the most recently edited main procedure with input from keyboard and output to display screen.
<control><break>	Interrupt execution. [Two keys must be pressed simultaneously.]
<control>d	End-of-file on PL/CS input from terminal. [Two keys must be pressed simultaneously.]
<resume>	Resume where execution was interrupted.
.ex in	Execute the most recently edited main procedure with input from text file named "in" and output to display screen.
.ex in PRT	Execute the most recently edited main procedure with input from text file named "in" and output to both display screen and printer.
.ex - PRT	Execute the most recently edited main procedure with input from the keyboard and output to both display screen and printer.

DEBUGGING COMMANDS

Command parameters enclosed in braces are optional.

.split [#1] [#2]	Simultaneous display: program to have #1 lines, output to have #2 lines, check to have (20 - #1 - #2) lines. defaults: 11 10 [no arguments] or #1 (21-#1) [1 argument]
.nosplit	Program and output each displayed on full screen. No check screen.
.trace	Cursor traces position in the program during execution. If the screen has not been split, it splits with the 11 10 default.
.notrace	Cursor not to trace position in the program during execution.
.pace [#]	Program will wait #/60 seconds at each program element. [Default: # = 30]
.nopace	Full speed execution.
.step	Set single step execution mode: when program is executed or resumed, control returns to the editor after one step of execution.
.nostep	Turn off single step execution mode.
.show	Print calling sequence of currently suspended environments.
.show var	Print the value of var. Var restricted to scalar variable or parameter.
.check var	Display the current value of the variable var during execution. If var is omitted, check all scalar variables of the current procedure.
.nocheck var	Don't display the value of the variable var during execution. If var is omitted, uncheck all scalar variables of the current procedure.

SPECIAL FUNCTION KEYS

Assignment of special function keys on the Terak 8532-2 keyboard:

key number	normal graphic	special function
(1)	ESC	<execute>
(15)	BACKSPACE	<diagonal-arrow>
(19)	¦NULL	<break> (with control)
(20)	LINEFEED	<clip>
(36)	`	<resume>
(38)	DC3	<insert>
(52)	DC1	<...>
(53)	DC2	<delete>
(54)	US	<clear>
(70)	ETX	<long>
(71)	DEL	<rubout>

```
(1)     1   2   3   4   5   6   7   8   9   0   -   =   (15)  ↑  {   }  (19)
    TAB
(20)    Q   W   E   R   T   Y   U   I   O   P   [   \        ↓  ~  (36) EOM

(38) LOCK A   S   D   F   G   H   J   K   L   ;   '  RETURN  <  (52)(53)(54)

CTRL SHIFT Z  X   C   V   B   N   M   ,   .   /     SHIFT    >   (70)  (71)

                          SPACE BAR
```