# THE CORNELL PROGRAM SYNTHESIZER: A TUTORIAL INTRODUCTION*

Tim Teitelbaum

TR 79-381

Revised June 1980

Department of Computer Science
Cornell University
Ithaca, New York 14853

* The development of the Cornell Program Synthesizer was supported in part by the National Science Foundation under Grant MCS77-08198.

CONTENTS

# The Cornell Program Synthesizer: A Tutorial Introduction *

Tim Teitelbaum
Department of Computer Science
Cornell University

## 1. Purpose

This tutorial introduces a novice student to the basic facilities of the Cornell Program Synthesizer for developing programs written in the PL/CS dialect of PL/I. No knowledge of programming is assumed or required. It is assumed that you possess a Synthesizer diskette and have access to a TERAK microcomputer.

## 2. Getting started

The TERAK consists of three physically separate components:

keyboard
video display screen
disk drive and processor

By itself, the computer is useless. A fourth component, the programming system, is required before a session can begin. The process of loading the TERAK with your copy of the Cornell Program Synthesizer is known as <u>booting</u>. The following steps are required to boot the system:

If necessary, open the door to the disk drive by pressing the horizontal bar just below the door. The door should spring open by sliding up. Remove any diskette that remains in the drive.

The <u>boot-switch</u> is in the upper-right corner of the disk drive. Press the boot-switch up.

Insert your own diskette into the disk drive. The diskette should be oriented so that the label is on the top surface, with lettering upside-down. Hold the labeled edge and insert the opposite edge first. The entire diskette should disappear into the drive and remain there. Close the drive door by sliding it down until it latches. If your diskette is not fully inserted into the drive, the door will not close. Your disk will begin booting as soon as the door is closed. The process of loading your copy of the Synthesizer into the TERAK takes about 12 seconds.

* The development of the Cornell Program Synthesizer was supported in part by the National Science Foundation under Grant MCS77-08198.

## 3. **The keyboard**

Several of the keys on the TERAK keyboard have a special meaning for the Synthesizer.* The layout of these keys is the following:

```
 ------------.                 .---------------------------------------.
| execute |   |               | ↖  | Δ  |  {  |    }    | break   |
|---------|    `-------.       `---|----|-----|---------|---------|
| clip    | command •  |           | ∇  |  ~  | resume  | unused  |
|---------|------------|        .--|----|-----|---------|---------|
| insert  | LOCK       |       | RETURN | ◁  | ... | delete  | clear   |
|---------|------------|       |--------|----|-----------------|---------|
| CTRL    | SHIFT      |       | SHIFT  | ▷  |      long       | rubout  |
 ----------------------         ---------------------------------------
     (left side)                            (right side)
```

Note that in this document, a word enclosed in angle brackets (like <RETURN>) denotes one particular key on the keybord. The four keys labeled with triangles are known as <up>, <down>, <left> and <right>. The key labeled with a diagonal arrow is known as <diagonal>.

Upper and lower case letters are considered distinct characters. Thus, words like "max", "MAX", and "Max" would each be considered distinct symbolic names in a program. All predefined PL/I names and keywords are expected to be typed in lower case.

Typing the <LOCK> key locks the keyboard in upper case. The red light on this key indicates that the keyboard is locked. Typing either <SHIFT> key unlocks the keyboard. Some of the special function keys have different meanings in upper case. If the keyboard does not seem to be responding normally, check to see if the keyboard is locked in upper case -- you may have typed <LOCK> by accident.

On the TERAK, there is no key for "¬", the not-sign. A tilde "~" is used instead. Thus, "¬=" is typed as "~=".

Be careful to distinguish between visually similar characters:

> "i", "I" (letter "I"), "1" (one), "l" (letter "L") and "|" (bar)
> "o", "O" (letter "O") and "0" (zero)
> "-" (arithmetic minus) and "_" (underscore)
> "." (decimal point), "•" (boldface dot), and "..." (ellipsis)

As you type, use short key-strokes since a pressed key begins to repeat after about a second.

---

* The Appendix contains a duplicate copy of the keyboard layout that can be removed and affixed to the keyboard for reference.

## 4. Creating a data file

A file is a named place where you store either a program or data. The Synthesizer can retain many files simultaneously, each distinguished by a different name. In this step of the tutorial, you will create a file named "pairs" containing a list of numbers.

The Synthesizer is controled by commands. All commands begin with the special symbol "•", a boldface dot. The key for boldface dot is labeled <command •>. Note that the boldface dot that identifies commands is not the same key as the decimal point.

Creating or modifying a file is known as editing the file. The command "•ed pairs" says that you wish to edit a file named "pairs". The contents of the file will be displayed below the dashed line as the file is created. A command is displayed above the dashed line because it is not part of the file per se.

> Type "•ed pairs" and then strike <RETURN>. (Do not type the quotation marks; type the characters that are enclosed in quotation marks.)

After issuing the command "•ed pairs", the display will look as follows:

```
                                   editing     pairs
----------------------------------------------------
object
```

Since "pairs" is a new file, it is initially empty. The word "object" is a placeholder that says, in effect, "an object goes here".

The square of light around the letter "o" of the word "object" is known as the cursor. The cursor serves as a pointer into the file being edited.

A data file consists of arbitrary lines of text. The command "•text" says that the currently edited file will consist of arbitrary text rather than a PL/I program:

> Type "•text" and then strike <RETURN>.

After issuing the command "•text", the placeholder "object" will be replaced by the following:

```
pairs: TEXT

{text}
```

The heading line "pairs: TEXT" is descriptive information and is not considered part of the file per se. The word "{text}" is called a placeholder. Placeholders identify locations in the file where insertions can be made. The braces in "{text}" mean that a list of zero or more text lines is permitted at this point of the file.

Now type the following list of number pairs. At the end of each line, use the <RETURN> key to go on to the next line. If necessary, use <rubout> to erase typographical errors. The <rubout> key erases the character just typed. Be careful to type a numeric "1", not a lower case "L":

```
2, 2
19, 77
56, -3
1234, 4321
50, 50
0, 0
```

## 5. Modifying a data file

Files are modified by making insertions and deletions at the position of the editing cursor. The editing cursor can be moved wherever a change is desired by using special control keys.

### 5.1. Cursor motion

Four keys labeled with triangles are known as <up>, <down>, <left> and <right>. They are used to change the position of the cursor in the file. The automatic repeat feature of the keyboard is often useful with the cursor control keys. While a cursor control key is held down, the cursor moves rapidly through the file.

The cursor control keys move the cursor [illegible] and backward through a file: <down> and <right> move the cursor forward, <up> and <left> move the cursor backward. The control keys move the cursor in different size increments: the keys <up> and <down> move the cursor one line at a time, the keys <left> and <right> move the cursor one character at a time:

> Repeatedly strike <up> until the cursor is positioned in the first line of the file. Then strike <down> until reaching the last line again.
>
> Repeatedly strike <left> until the cursor is positioned in the first line of the file. Then strike <right> until reaching the end of the file.

The <RETURN> key also controls the cursor. <RETURN> moves the cursor like <down> but also positions the cursor between lines. Note that as the cursor moves in between lines, the placeholder "{text}" appears; it disappears when the cursor moves onto an existing line. Thus, if you ever wish to insert a new line between two lines, position the cursor on the first of the two lines and then strike <RETURN>.

> Move the cursor to the top of the file using <up> and then repeatedly strike <RETURN> until reaching the bottom again.

## 5.2. Deleting and inserting

The <delete> key erases an entire line in a single operation. The placeholder "{text}" reappears when the line is deleted:

> Position the cursor anywhere on the line "1234, 4321" and strike <delete>. Then retype the line.

Text can be inserted in the middle of a line. Typed text is inserted to the left of the cursor position:

> Use <left> to position the cursor on the "," of the line "1234, 4321" and then type "wxyz".

Text can be deleted in the middle of a line. <Rubout> erases the character to the left of the cursor. Repeated use of <rubout> has the effect of erasing backward through the line:

> Erase "wxyz" by typing <rubout> four times.

<Clear> can be used to erase the character at the cursor position. Repeated use of <clear> has the effect of erasing forward through the line:

> Once again type "wxyz" in the middle of the line. This time, use <left> to reposition the cursor at the "w". Erase "wxyz" by typing <clear> four times.

## 5.3. Moving lines

It is possible to move entire lines in order to reorganize a file. The <clip> key temporarily removes a line from the file. The cursor can then be repositioned at a "{text}" placeholder where you wish to reinsert the line. The <insert> key inserts the "clipped" line at the position of the cursor.

We illustrate the use of <clip> and <insert> to move the line "50, 50" immediately after the line "2, 2":

> Position the cursor on "50, 50", the line to be moved.
>
> Clip the line out of the file by striking <clip>. (This actually moves the line to a file named "CLIPPED".)
>
> Position the cursor on a "{text}" placeholder at the desired destination after the line "2, 2". Remember that the way to make a placeholder appear between two lines is to put the cursor on the first of the two lines and then strike <RETURN>.
>
> Reinsert the line into the file by striking <insert>.

The <clip> key, by itself, clips only one line at a time. In order to move several consecutive lines, you must specify the first and last lines to be clipped. The two keys <long> and <clip>, struck one after the other, mean "clip a list of lines starting at the current cursor position". After typing <long><clip>, you reposition the cursor on the last of the list of lines to be clipped and strike ".".

We illustrate the use of <long><clip> to move the lines

```
19, 77
56, -3
1234, 4321
```

immediately after the line "2, 2". This will restore the list to its original order:

> Position the cursor at "19, 77", the first line to be moved. Type <long><clip>.
>
> Position the cursor at "1234, 4321", the last line to be moved. Strike <command .>. (The specified list of lines is actually moved to the file named "CLIPPED".)
>
> Position the cursor on a "{text}" placeholder at the desired destination after the line "2, 2". Remember that the way to make a placeholder appear between two lines is to put the cursor on the first of the two lines and then strike <RETURN>.
>
> Type <insert>.

## 6. Creating a program file

Program files are created in a different way from data files. You cannot type the program text into the file character-by-character. Rather, you must use commands to insert predefined templates for the various PL/I language constructs. Only a fraction of the total program is explicitly typed; the rest is "synthesized" as a result of the commands. A command is not part of the file; rather, it is a directive to the Synthesizer to insert a given template into the file. For example, the command ".dw" inserts the template

```
DO WHILE ( condition );
    {statement}
    END;
```

into the file. "Condition" and "{statement}" are placeholders that identify locations where subsequent insertions will be made.

The parts of a program that are typed explicitly are known as phrases. The editing of a phrase is exactly like the editing of a {text} line in a data file. PL/I constructs for which templates are provided can not be typed as phrases -- they must be entered as templates by command.

Although, at first, it may seem circuitous to enter templates by command, the commands allow you to enter the program quickly and accurately. Since the template is predefined, you can not make a mistake in typing it. The placeholders of a template serve as prompts that guide your subsequent insertions and allow the Synthesizer to validate the syntactic correctness of your program as it is created phrase-by-phrase and template-by-template. The indenting of the program is provided automatically as part of the predefined template.

A complete list of the commands for inserting program templates appears in the Appendix.

In this section of the tutorial, a complete script is provided to create the following sample program in the file named "adder":

```
/* print each input data pair and its sum */
adder: PROCEDURE OPTIONS (MAIN);
   DECLARE ( x, y  /* number pair to be summed */ ) FIXED;
   /** process data until x not positive */
      GET LIST ( x, y );
      DO WHILE ( x>0 );
         PUT SKIP LIST ( x, y, x+y );
         GET LIST ( x, y );
         END;
   END adder;
```

It is not necessary for you to understand how this program works. Our current interest is how programs are created using the Synthesizer.

The command ".ed adder" says that you want to edit a file named "adder". The display of the "pairs" file is then replaced by the display of the "adder" file:

Type ".ed adder" and then strike <RETURN>.

By following the script below, you will create the desired program. In the event of a typographical error, an individual typed phrase can be modified exactly like a "{text}" line in a data file. In particular, you can use <rubout> to erase the last character typed or <delete> to erase the entire phrase.

Be careful not to omit the <RETURN> in the fifth line of the script. It is essential that the ".c" in the sixth of the script be typed with the cursor positioned at "{statement}", not "{declaration}".

| WITH THE CURSOR POSITIONED AT: | YOU TYPE: |
|---|---|
| object | .main <RETURN> |
| comment | print each input data pair and its sum <RETURN> |
| {declaration} | .fx <RETURN> |
| list-of-variables | x,y /* number pair to be summed */ <RETURN> |
| {declaration} | <RETURN> |
| {statement} | .c <RETURN> |
| comment | process data until x not positive <RETURN> |
| {statement} | .g <RETURN> |
| list-of-variables | x,y <RETURN> |
| {statement} | .dw <RETURN> |
| condition | x>0 <RETURN> |
| {statement} | .p <RETURN> |
| list-of-expressions | x,y,x+y <RETURN> |
| {statement} | .g <RETURN> |
| list-of-variables | x,y <RETURN> |

When you have completed the script, verify that the program on the display screen matches the program above. (The one exception to a perfect match is that the cursor will be positioned at a "{statement}" placeholder). If you have made mistakes, read the section "Modifying a program file" and correct the program before running it.

## 7. **Running a program**

The purpose of the "adder" program is to add pairs of numbers. You will now run the "adder" program on the data stored in the file named "pairs". Running a program is also called <u>executing</u> the program.

The command ".ex pairs" initiates the execution of the "adder" program with input data taken from the file "pairs". Notice that you only mention the name of the data file in the command. The Synthesizer assumes that the program you wish to execute is "adder", the most recent program to have been edited. Notice also that it is not necessary to make the "{statement}" placeholder disappear before executing the program.

Type ".ex pairs" and then strike <RETURN>.

While the program is executing, the display of the "adder" file is replaced by the program's output. Each pair of numbers and its sum appears on a separate line.

When the program has finished, it pauses and you are prompted to "type any character to continue". This provides an opportunity for you to review the output on the screen. Notice that no line of output appears for the pair of numbers "0, 0". The program, as written, uses "0, 0" only to signal the end of the data file. Thus, the omission of output for the "0, 0" is deliberate.

The Synthesizer is waiting for a signal from you before it continues. When any character is typed, the execution output is erased and the adder file reappears:

Strike the <space> key once.

## 8. Getting a printout

From time to time, you may wish to have a paper printout of your program or its output. At Cornell, each TERAK is numbered. The printer is always switched to one particular TERAK.

The command ".pr" prints the file that is currently being edited on the printer. The command ".ex pairs PRT" executes the program with input from the file named "pairs" and output directed to the printer (as well as the video display).

Verify that the printer is not in use; then switch it to your TERAK.

Type ".pr" and then strike <RETURN>.

After the printer stops, type ".ex pairs PRT" and then strike <RETURN>.

When the execution of your program terminates, you will be prompted to "type any character to continue". Strike the <space> key once and then retrieve your printout from the printer.

## 9. Getting off

The next step of this tutorial is to terminate the session at the TERAK as if you were finished.

Type the command ".off" and then strike <RETURN>. After a few seconds, the Synthesizer should respond with the message "Session ended".

Open the door to the disk drive by pressing the horizontal bar just below the door. The door should spring open by sliding up. Remove your diskette.

Unless the TERAK is about to be used, turn the power off by pressing the boot-switch down.

Never remove your diskette or turn the power off without first issuing the ".off" command and receiving the "Session ended" message. You can type ".off" whenever commands are permitted.

## 10. Modifying a program file

To continue with the tutorial, get back on the TERAK by following the booting instructions in Section 2 above. The system will be restored exactly as it was at the time you issued the ".off" command, editing the file "adder".

### 10.1. Cursor motion

The cursor motion keys <up> and <down> move the cursor backwards and forwards through the program one phrase, template, or placeholder at a time:

> Repeatedly strike <down> until the cursor advances no further. Then strike <up> until reaching the top again.

The cursor motion keys <left> and <right> also move the cursor to the next phrase, template, or placeholder but within a phrase, advance the cursor one character at a time:

> Repeatedly strike <right> until the cursor advances no further. Then strike <left> until reaching the top again.

<RETURN> moves the cursor like <down> but also positions the cursor between consecutive declarations or statements. When the cursor moves between declarations, the placeholder {declaration} appears; when it moves between statements, the placeholder {statement} appears. These placeholders identify locations in the file where insertions can be made. The placeholders disappear when the cursor moves onto an existing declaration or statement. Thus, when you wish to insert a new declaration or statement into your program, position the cursor above the desired destination and strike <RETURN>.

> Repeatedly strike <RETURN> until the cursor advances no further.

### 10.2. Deleting, inserting and moving program elements

In a data file, <delete> or <clip> removes one line of text at a time. In a program file, <delete> or <clip> removes a whole structural unit from the program. A given structural unit may be only* part of a line or it may extend accross many lines of the program.

When the cursor is positioned anywhere in a phrase, <clip> or <delete> removes the entire phrase. When the phrase has been removed, the appropriate placeholder reappears:

> Position the cursor in the phrase "x>0" and strike <delete>. Then retype the phrase "x>0".

When the cursor is positioned at a template, the entire template and all its components are removed:

> Position the cursor at "DO" and strike <clip>. Reinsert the clipped template at the same location by typing <insert>.

---

* There is actually no difference between editing a text file and editing a program file. The "structural units" of a text file just happen to be lines of {text}.

Position the cursor at "/**" and strike <clip>. Reinsert the clipped template at the same location by striking <insert>.

When the cursor is positioned at a template, the entire template and all its components can be deleted in a single step. However, the most recently deleted phrase or template is saved in a file named "DELETED". Thus, if you delete a section of code by accident, it can be recovered by typing ".ins DELETED":

With the cursor still positioned at "/**", strike <delete>. Recover the deleted template by typing ".ins DELETED" and then strike <RETURN>. (The capital letters in "DELETED" are essential since the names "DELETED" and "deleted" denote two different files.)

The way to delete an entire file is to position the cursor at the very top at the outermost template of the file and strike <delete>.

### 10.3. **Hidden placeholders**

Placeholders for optional program elements are not normally displayed. For example, the full form of a "DO WHILE" template is

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

but the "[loop-name:]" placeholder is normally hidden from view. In order to enter a loop-name, it is first necessary to make the "[loop-name:]" placeholder appear. The special command ".o" ("move to option") is used to make a hidden placeholder appear:

Position the cursor at "DO", then type ".o" and then strike <RETURN>.

Type "readloop" and then strike <RETURN>. Note that the loop-name "readloop" is automatically inserted after the appropriate "END".

Suppose we now wish to change "readloop" to "Readloop":

Position the cursor at "readloop".

The cursor now denotes the entire "DO WHILE" template. This can be verified as follows:

Strike <clip>. (Note that the entire template is clipped.) Then reinsert the template by typing <insert>.

In order to position the cursor on the individual phrase "readloop" you must type the command ".o":

Type ".o" and then strike <RETURN>.

Note that the cursor did not change position when you typed ".o". This is because the same cursor position on the screen denotes both the entire "DO WHILE" template and also denotes the individual phrase "readloop". Fortunately, this is the only example in PL/CS where the meaning of the cursor position on the screen is ambiguous.

Strike <clear>. Then type "R". Then strike <RETURN>.

## 11. **Error prevention**

A program that is ungrammatical is said to contain a "syntax error". The Synthesizer will detect syntax errors on a phrase-by-phrase basis and will only permit you to create grammatically correct programs. We illustrate the Synthesizer's error prevention mechanism by making some intentional mistakes.

Position the cursor at the beginning of the phrase "x>0" and strike "/". Then strike <RETURN>.

The phrase "/x>0" is not a syntactically correct condition. The syntax error is detected: the buzzer sounds, an error message appears on the top line of the screen and the cursor is positioned at the point of error.

The error can be corrected immediately:

Strike <clear> to erase the "/" and then strike <RETURN>.

Errors do not have to be corrected immediately. However, if allowed to remain in the program, an incorrect phrase is highlighted to alert you to its presence:

Position the cursor at the phrase "x>0" strike "/". Then strike <RETURN>. When the syntax error is detected, strike <RETURN> again without first correcting the phrase.

An incorrect, highlighted phrase can be re-edited like any other phrase. Thus, you can now go back to the invalid phrase and correct it:

Strike <up> to position the cursor at the incorrect phrase. Strike <clear> once to erase the leading blank on the invalid phrase. Then strike <clear> again to erase the "/". Then strike <RETURN>.

Sometimes, a change in one part of the program causes errors in another part of the program. For example, if the declaration

```
DECLARE ( x, y /* number pair to be summed */ ) FIXED;
```

is deleted, the names "x" and "y" are said to be "undeclared". Since it is an error to use an undeclared name, all remaining phrases containing "x" and "y" are then highlighted to indicate that they contain errors. The display returns to normal when the declaration is restored:

Position the cursor at the word "DECLARE" and strike <clip>. (Notice that phrases containing "x" or "y" are highlighted.)

Reinsert the declaration by typing <insert>.

## 12. Abbreviated file display

When programs get large, it is useful to be able to suppress the display of selected parts. The display of code subordinate to a "/**" type of comment can be suppressed by typing the ellipsis key <...>. This is not the same as "clipping" or "deleting" which actually removes code from the program. <...> just suppresses the display of code that remains in the file:

Position the cursor at the comment symbol "/**" and strike <...>. Strike <...> again and the code reappears.

## 13. Program execution features

This section of the tutorial introduces features of the Program Synthesizer that allow you to monitor and control the execution of a program. These features will be especially valuable to you when your program doesn't work and you are trying to determine why.

The "trace feature" allows you to monitor the steps the computer takes as it moves through a program during execution. The command ".trace" turns on the feature. A "T" in the upper right corner of the screen indicates that the trace feature is on. When the trace feature is enabled, the program is displayed in the upper partition of the screen; the execution output is displayed in the lower partition. As the program executes, the cursor indicates the position of the computer's "instruction pointer":

Type ".trace" and then strike <RETURN>.

Now that the trace feature has been turned on, run the "adder" program again on the data in "pairs". You should be able to observe the cursor tracing through the program as it executes although it happens too rapidly to follow in detail:

Type ".ex pairs" and then strike <RETURN>.

The "pace feature" causes the Synthesizer to pace itself so that you can follow the separate steps of a computation. The pace is measured in 1/60 seconds per program step. Thus, the command ".pace 60" sets a pace of 1 second per program step. A "P" in the upper right corner of the screen indicates that the pace feature is on:

Type ".pace 60" and then strike <RETURN>.

Now run the program again. This time you will be able to observe the step-by-step operation of the program:

Type ".ex pairs" and then strike <RETURN>.

The "check feature" allows you to monitor the values of the program variables during execution. The command ".check" turns on the feature for all variables of the current procedure. A third partition of the screen (in the middle) is used to display the values of the variables:

Type ".check" and then strike <RETURN>.

Now run the "adder" program again on the data in "pairs". You should be able to observe the values in the variables changing as the program runs:

Type ".ex pairs" and then strike <RETURN>.

You will often need to suspend the execution of a program before it terminates on its own. For example, an error in the program may cause it to "run forever". Or the pace may be far too slow. Or you may want to stop to examine the contents of some program variables.

Suspending the execution of a program prematurely is called <u>interrupting</u> or <u>breaking</u> out of execution. To interrupt the execution of the program, press the key labeled <CTRL> and, while holding <CTRL>, strike the key labeled <break>. <CTRL> is in the lower left corner of the keyboard; <break> is in the upper right corner:

Type ".ex pairs" and then strike <RETURN>.

After the program has run for a while, hold <CTRL> and strike <break>.

It is possible to enquire about the values of individual variables of a program suspended in the middle of execution. The command ".show x" requests display of the value of the variable named "x". Although unnecessary in the present example, since all variables are displayed on the check screen, this feature will be useful when the check feature is not in use.

Type ".show x" and then strike <RETURN>.

The "step feature" allows you to execute the program one step at a time. The command ".step" turns on the feature. An "S" in the upper right corner of the screen indicates that the step feature is on:

Type ".step" and then strike <RETURN>.

When a program is executed with the step feature enabled, the program stops after each step. You explicitly request execution of each step with the step-size of each resumption specified by command:

| | |
|---|---|
| <resume> | execute the current statement one step at a time, |
| <long><resume> | execute the current statement as the next step, |
| <return> | complete a list of statements as the next step, |
| <diagonal> | complete the enclosing template as the next step. |

We now execute the "adder" program and illustrate the effects of the different single-step resume commands:

Type ".ex pairs" and then strike <RETURN>.

Repeatedly strike <resume> until the line "2 2 4" appears in the output. (Note that the cursor stops once for every template and phrase of the program.)

Now, repeatedly type <long><resume> until the line "19 77 96" appears in the output. (Note that the PUT-statement is executed as one step; similarly, the GET-statement is executed as one step.)

Now, repeatedly strike <RETURN> until the line "1234 4321 5555" appears in the output. (Note that both statements in the WHILE-loop are executed as one step.)

Finally, strike <diagonal>. (Note that this completes the execution of the WHILE-loop as the next step, which, in turn, completes execution of the entire program.)

Although there are individual commands that turn off the separate debugging features one at a time, the command ".nodebug" (i.e. no debug) turns off all debugging features at once:

Type ".nodebug and then strike <RETURN>.

It is possible to run the program with input data supplied directly from the keyboard rather than from a text file. Typing the <execute> key invokes execution of the "adder" program with input from the keyboard. When the program requires a pair of numbers, you will be prompted with the message "type input data". Type a pair of numbers followed by <RETURN> each time you are requested for data. You can suspend execution of the program by holding <CTRL> and then typing <break>.

Strike <execute>.

When prompted for input, type "2, 2" and then strike <RETURN>.

When next prompted, type "19, 77" and then strike <RETURN>.

Enter as many number pairs as you wish. When you are ready to terminate execution, hold <CTRL> and strike <break>. Then strike the <space> key once to restore the program display.

## 14. **End of exercise**

Terminate your session by following the instructions given in Section 9.

## 15. On Comments

The Synthesizer constrains your use of comments to the following contexts:

a) the "comment" field of a procedure-template
b) the "comment" field of a comment-template
c) the end of a list-of-variables in a declaration
   or parameter-declaration

In all other contexts, a comment is considered syntactically incorrect.

In the first two cases, "/*" and "*/" are provided as part of templates; you just type the comment text. In the third case, you type the entire comment including "/*" and "*/". The comment is only permitted at the end of the list-of-variables, not in between. Thus, the first example below is correct but the second is incorrect:

```
DECLARE ( x, y /* comment text */ ) FLOAT;
DECLARE ( x, /* comment text */ y ) FLOAT;
```

### 15.1. Comment-templates

A comment-template is inserted by typing the command ".c" when the cursor is positioned at one of the placeholders:

```
{statement}
statement
{declaration}
{parameter-declaration}
```

We shall consider the case of a comment-template inserted at "{statement}"; the other cases are similar.

When the cursor is positioned at "{statement}", the command ".c" inserts the template:

```
/** comment */
    {statement}
```

A comment-template includes two placeholders: "comment" (for the comment text) and "{statement}" (for the list of statements being commented). The extra "*" is part of the template and serves to identify this as a comment-template. (As far as PL/I is concerned, the extra "*" is just the first character of the comment.)

In a comment-template, the "{statement}" placeholder is indented to show that it is subordinate to the comment. It is important to understand that "comment" and "{statement}" are part of one template and that "/* comment */" is not just a line inserted in front of "{statement}". Thus, if you have a list of statements to be commented, you do not just insert a comment-template in front of those statements; rather, you must insert the statements into the "{statement}" part of the comment-template.

### 15.1.1. Entering comments first

It is easiest to enter comment-templates as the program is first created, as follows.

Type the command ".c" and then strike <RETURN>. This inserts the template and positions the cursor at the "comment" placeholder:

```
/** {comment */
    {statement}
```

Then type a comment that specifies _what_ you want to accomplish. Typing <RETURN> at the end of the comment text positions the cursor at "{statement}":

```
/** what the following statements do */
    {statement}
```

Now enter the statements that say _how_ to accomplish the goal you have specified in the comment. Each statement you enter will be inserted into the template. The <RETURN> typed after each statement will position the cursor at a new "{statement}" placeholder still within the template:

/** what the following statements do */
    $S_1$;
    $S_2$;
    ...
    $S_n$;
    {statement}

When you want to insert a statement _after_ the entire comment-template, not within it, just type an extra <RETURN>. This will position the cursor at a "{statement}" placeholder after the entire comment-template:

/** what the following statements do */
    $S_1$;
    $S_2$;
    ...
    $S_n$;
{statement}

The indenting level tells you whether a statement or placeholder is _within_ the template (subordinate to the comment) or _after_ the template.

### 15.1.2. Adding comments later

Suppose you wish to comment a program section that has already been entered into a file. For example, consider commenting the following code:

```
        temp= x;
        x= y;
        y= temp;
```

If you just insert a comment-template in front of the section to be commented, the section will not be subordinate to the comment:

```
        /** comment */
            {statement}
        temp= x;
        x= y;
        y= temp;
```

The best way to comment an existing section of code is to temporarily remove the section from the program, generate the comment-template, and then reinsert the section within the template.

In order to remove a list of statements from your program temporarily:

position the cursor on the first statement to be removed,
type <long><clip>,
position the cursor on the last statement to be removed,
type <command .>.

This moves the statements to the file named CLIPPED and leaves the cursor positioned at a "{statement}" placeholder. If the section to be commented consists of just one statement, then that one statement can be moved by positioning the cursor on it and typing <clip>.

Generate the comment-template by typing ".c" followed by <RETURN>. Then type the desired comment text followed by <RETURN>. The cursor will then be positioned at the "{statement}" placeholder within the template:

```
        /** exchange x and y */
            {statement}
```

Now reinsert the statements stored in CLIPPED by typing <insert>:

```
        /** exchange x and y */
            temp= x;
            x= y;
            y= temp;
```

### 15.1.3. **Ellipsis**

The separate sections of a program work together to implement some algorithm. However, the importance of any particular section depends only on <u>what</u> it does, not <u>how</u> it does it.

Each logical section of a program should be the subordinate part of a comment-template:

```
/** comment */
    {statement}
```

The comment should describe precisely *what* the subordinate statements do; the subordinate code says *how* to do it:

```
/** what */
    how
```

Of course, the "how" part is needed because the computer is unable to understand the English specification provided in the comment. However, for the purpose of human understanding, the "how" part has no bearing on the relationship of this section to the rest of the program. Provided that the comment is sufficiently precise and descriptive, the "how" part is superfluous and could be displayed as an ellipsis "...". Hiding details within an ellipsis will enable you to see the more global structure of your program on the display screen.

The Synthesizer provides the ellipsis feature in a single key-stroke: typing <...> hides the code of the innermost comment-template enclosing the present cursor position. For example, suppose the cursor were positioned as shown by the box below:

```
/** exchange x and y */
    temp= x;
    x= y;□
    y= temp;
```

Then typing <...> would result in the display:

```
/** exchange x and y */
    □..
```

The code can be revealed by typing <...> again. A good use of this feature is to hide all declarations of a program within a single comment template:

```
/** declarations */
   ...
```

Then they can be displayed only when you want to see them.

Programs can be executed even though some sections are displayed as "...". The code is still part of your program even though it is not displayed. When tracing, pacing, or stepping the execution of a program, the "..." is considered a single operation. This feature will be useful when you wish to trace through a program but don't wish to see a detailed trace of some inner loop. In this case, just put the inner loop in a comment-template and display it as "...".

## 16. **Acknowledgements**

I am grateful to Sidney Siskin for her editorial assistance in the preparation of this tutorial.

Many people have contributed to the development of the Synthesizer. It is a pleasure to acknowledge the special role of Thomas Reps who has worked closely with me 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, Ron Olsson, Steve Mahaney and Ralph Johnson.

## APPENDIX

---

CONTENTS

## A. TEMPLATE INSERTION COMMANDS

```
---------------------------------------------------------------------
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}
                           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 [attributes];

             .fl       DECLARE ( list-of-variables ) FLOAT [attributes];

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

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

             .c        /** comment */
                          {declaration}


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

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

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

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

## B. PHRASE SYNTAX

| PLACEHOLDER | PHRASE |
|---|---|
| comment | any sequence of characters not including "*/" |
| {text} | any sequence of characters |
| list-of-variables<br>(in a declaration) | v1,v2, ..., vn (optionally followed by a comment)<br>(both lower and upper subscript bounds required for arrays) |
| list-of-parameters<br>(in a parameter declaration) | v1,v2, ..., vn (optionally followed by a comment)<br>(*'s required for subscript bounds of arrays) |
| [attributes] | static, external |
| statement<br>or<br>{statement} | variable= expression; ( semi-colon optional )<br>array_name= expression;<br>array_name= array_name;<br>substr( variable, expression, expression )= expression;<br>substr( variable, expression )= expression;<br><br>leave loop-name;<br><br>call proc-name( argument list );<br>call proc-name;<br>return;<br><br>pause; |
| statement | ; |
| expression<br>or<br>condition | any expression |
| list-of-expressions<br>(in a PUT) | a list of scalar expressions or array names separated by commas. |
| list-of-variables<br>(in a GET) | a list of scalar variables, subscripted variables or array names separated by commas. |
| var= exp to exp by exp | variable= expression to expression<br>variable= expression to expression by expression |
| [loop-name:] | any name: (colon optional) |

```
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
```

## C. OPERATORS and BUILTIN FUNCTIONS

---

Operators:

```
    arithmetic:          +    -    *    /    **

    character string:    ||

    relational:          =    ~=   <    <=   >    >=   ~<   ~>

    logical:             &    |    ~
```

Functions:

```
    abs(x)
    ceil(x)
    fixed(x)
    float(x)
    floor(x)
    index(s1,s2)
    length(s)
    max(x1,x2,...,xn)
    min(x1,x2,...,xn)
    mod(x,y)
    sign(x)
    substr(s,f,n) and substr(s,f)
    translate(s1,s2,s3)
    trunc(x)
    verify(s1,s2)
```

## D. INCOMPATIBILITIES with PL/I, PL/C, and Batch PL/CS

LEXICAL:

Variable names can contain both upper and lower case letters.
Builtin functions have lower case names only.
E.g. substr(...), not SUBSTR(...)
Reserved keywords are lower case only.
E.g. call, not CALL
Arithmetic constants are either FIXED(5,0) or FLOAT(6).

```
E.g.  2      behaves as 00002
      2.     behaves as 2.00000e0
      2.0e0  behaves as 2.00000e0
```

FLOAT constants -- can use either 'e' or 'E'.
BIT constants -- can use either 'b' or 'B'.

STRINGS:

No particular upper limit on string length.
ASCII Character set and collating sequence is used instead of EBCDIC.

ARITHMETIC:

FIXED range is -32,767 to +32,767.
FIXED divide is integer division.
E.g. PUT LIST ( 1/2 ); prints 0, not .5

BIT:

BIT arrays are initialized to 0.
String operations are not permitted on BIT variables (e.g. ||, substr, etc. )

CONVERSIONS:

Conversions between all types are permitted.
Number to BIT conversion is not standard PL/I:

```
      0  -->  '0'b          1  -->  '1'b
  0.0e0  -->  '0'b      1.0e0  -->  '1'b
  other  -->  run time error
```

INPUT:

GET LIST ( ... );
In addition to blanks and commas, <RETURN> delimits an input field.
No data is passed to GET until a <RETURN>.

OUTPUT:

PUT LIST ( ... );
The screen is divided into five fields of 16 characters.
PUT SKIP(0) LIST ( ... );
An overstruck character erases the overwritten character.
PUT SKIP(expression) LIST ( ... );
Negative skip expression skips backward (i.e. upward) on the screen.

DO-INDEX STATEMENT:

Index variables of any type are allowed but the limit is coerced to numeric.
E.g. DO string_variable= 1 TO '10' BY 1
does an arithmetic relation test, not a string relation test.
The index variable is not READONLY (unlike Batch PL/CS).

PROCEDURES:

Procedures without parameters are allowed but there is no way to eliminate "( list-of-parameters )" from the procedure template.

UNIMPLEMENTED LANGUAGE FEATURES (of PL/CS dialect):

- ASSERT
- attributes
  - INITIAL
  - READONLY
- GET DATA, GET EDIT
- functions
  - user defined function procedures
  - remaining builtin functions
- GOTO
- label:;
- ON
- PUT DATA, PUT EDIT

## E. 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 on disk and terminate session. Destroys previous saved status. |
| .save | Save the status of the system on disk without terminating session. Destroys previous saved status. |
| .list | List a directory of all non-empty files. |
| .ed file | Edit the file named "file". |
| .pr | Print the currently edited file on a hard-copy printer. |
| .sc | (Terak only.) Turn on the raster graphics screen. An n by 320 BIT array declared at the beginning of the main procedure will be displayed in the bottom third of the screen. (In the present implementation, there is only room for n<70.) |
| .nosc | (Terak only.) Turn off the raster graphics screen. |
| .name id | The identifier "id" is recorded as the disk name. This name is printed at the head of output for identification. The name "id" must have the form of an identifier; i.e. a letter followed by zero or more letters, digits, or underscore characters. Spaces are not permitted within the id. The ".name" command can only be used once. Thus, the name is indelible. |

## F. CURSOR MOTION COMMANDS

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

## G. 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. |
| <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". |
| <clip> | Same as ".mv CLIPPED". |
| <delete> | Same as ".mv 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". |
| <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. |
| .rn v1 v2 | Rename the variable or parameter named "v1" as "v2". |

## H. EXECUTION COMMANDS

| | |
|---|---|
| .ex | Execute the most recently edited main procedure with input from the keyboard and output to the display screen. |
| <execute> | Same as ".ex". |
| .ex in | Execute the most recently edited main procedure with input from the text file named "in" and output to the display screen. |
| .ex in PRT | Execute the most recently edited main procedure with input from the text file named "in" and output to both the display screen and printer. |
| .ex - PRT | Execute the most recently edited main procedure with input from the keyboard and output to both the display screen and printer. |
| <CTRL><break> | Interrupt execution. (Hold <CTRL> while striking <break>.) |
| <CTRL>d | End-of-file on PL/CS input from terminal. (Hold <CTRL> while striking d.) |
| <resume> | Resume where execution was interrupted. |

When the single-step execution mode is enabled, control returns to the user after each "step". Execution is resumed by a command that specifies the size of the next step:

| | |
|---|---|
| <resume> | Execute the current statement one step at a time, returning to the user before executing each of its subordinate parts. |
| <long><resume> | Execute the current statement and all its subordinate parts as the next step. |
| <return> | Complete the current list of statements as the next step. |
| <diagonal> | Complete the enclosing template as the next step. |
| <long><diagonal> | Complete the enclosing procedure as the next step. |

## I. DEBUGGING COMMANDS

Command parameters enclosed in braces are optional.

.nodebug — Turn off all debugging features.

.split [#1] [#2] — Partition display screen into windows:
program trace window: #1 lines,
program output window: #2 lines,
variable check window: (20-#1-#2) lines.
No parameters defaults to ".split 11 10".
One parameter defaults to ".split #1 (21-#1)".

.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. During execution, control returns to the user after each step. See EXECUTION COMMANDS for a list of the different resumption commands when the single-step feature is enabled.

.nostep — Turn off single step execution mode.

.show — Print the calling sequence of currently suspended procedures.

.show var — Print the value of var. Var restricted to scalar variable or parameter.

.check [var] — Display the value of the variable var during execution. If var is omitted, check all variables of the current procedure.

.nocheck [var] — Don't display the value of the variable var during execution. If var is omitted, uncheck all variables of the current procedure.

## J. ADVISORY MESSAGES

The following advisory messages appear on the screen at various times. They require no response by you.

CORNELL PROGRAM SYNTHESIZER logo
While the Synthesizer diskette is booting, a logo appears on the screen providing version date and other information. If the logo is not replaced on the screen within 20 seconds, your disk has failed to boot. Your disk may be dirty or damaged; the microcomputer may be dirty or damaged. Try several times to boot the disk; try to boot the disk on a different microcomputer.

editing
A file is being edited; its name appears in the upper-right corner of the screen. During editing, you can examine and modify the current file, redirect the editor to another file, change debugging options, examine variables of a suspended execution, resume execution of a suspended execution, or invoke a new execution.

executing
A file is being executed; its name appears in the upper-right corner of the screen. During execution, you can provide the program with input data when requested to, you can interrupt execution, and if the single step execution mode is enabled, you can manually trigger the execution of the next program step.

please wait
Going back and forth between editing and execution requires a short wait.

Session ended
This is the normal response to the ".off" command. When this message is on the screen, the system can only be rebooted. If you do not receive this message before removing your diskette, you are in danger of loosing information.

## K. PROMPT MESSAGES

The following prompt messages appear on the screen at various times. They require a response from you before editing or execution can continue.

position cursor at end of sublist and strike <command .>

Moving some of the elements of a list of declarations, statements, or when-clauses requires that the first and last element of the desired list be specified. The position of the cursor when <long><clip> or the command ".ml" is typed specifies the first element of the list and prints this prompt message. You are now expected to reposition the cursor on the last element of the list to be moved after which you should strike <command .>. While this prompt is on the screen, the cursor can only move to meaningful positions in the file.

strike a resume key, or any other to interrupt

When the single-step execution mode is enabled, you receive this prompt after each step is executed. If you wish to execute the next step, strike <resume> or one of the other resumption commands listed in the section EXECUTION COMMANDS. Otherwise, strike any other character and you will suspend execution.

type any character to continue

You are being given an oportunity to examine the output on the screen before it is erased. If the word "executing" appears in the upper-right of the screen, then the reason for the delay is that the next line of output will "turn" the output page. If the word "executing" does not appear, then execution has terminated or suspended and you are about to return to editing a file.

type input data

Your program has executed a GET LIST statement and is now waiting for you to provide a line of input data. You can type any number of data items. A <RETURN> signals that the line of input data is prepared and ready for the program. If the GET LIST statement needs more data items than you have provided, you will be asked for more data. If the GET LIST statement needs fewer data items than you have provided, then the surplus items will be saved for the next GET LIST executed.

The line can be corrected as if it were a phrase any time before typing <RETURN>. In particular, use <left> and <right> to move the cursor, use <clear> or <rubout> to erase characters, use <delete> to delete the entire line, type wherever insertions are desired.

## L. ERRORS

Three kinds of errors can be distinguished:

1) errors in the use of commands and special function keys,
2) errors during program execution,
3) syntax errors in typed phrases.

1) If you use a command or special function key incorrectly, you receive an explanatory error message and the request is ignored.

2) If your program performs an illegal operation during execution, then execution is suspended, you receive an explanatory error message, and the editing cursor is positioned _at the first character_ of the offending phrase. This cursor position is sometimes misleading because the cursor does not pinpoint the position of the error in the phrase. For example, suppose variables x and y have been assigned values but z has not yet been assigned any value. Then attempting to execute the statement

```
PUT SKIP LIST ( x, y, z );
```

will result in the message

```
error: uninitialized variable
```

with the cursor positioned on the "x". Of course, it is not the "x" that is uninitialized but the "z". (Cursor positioning after execution errors will be improved in subsequent versions of the Synthesizer.)

3) If you type an ungrammatical phrase into a file, then as soon as you direct the cursor to leave the phrase, an error message will be printed and the cursor will be positioned as close to the error as possible. You can then either re-edit the phrase or immediately redirect the cursor away from the phrase. If you re-edit the phrase, its syntactic correctness will again be validated possibly resulting in another error message. If you immediately redirect the cursor, the erroneous phrase will be highlighted on the screen.

You can return to a highlighted phrase anytime to correct it. If you do not know why a highlighted phrase is highlighted, just re-edit it slightly (for example, by typing a blank at the beginning) and strike <RETURN>. This will cause the error message to be printed.

## M. ERROR MESSAGES

already declared
The same variable name has been declared earlier in this procedure or is a builtin function.

arg-parm mismatch
An argument in a procedure call is not consistant with the corresponding parameter in the procedure definition. The argument can be a scalar variable or expression if and only if the corresponding parameter is declared scalar. The argument can be an array name if and only if the corresponding parameter is declared as an array parameter with the same number of dimensions.

arithmetic overflow
The result of an arithmetic operation is too large.

bad character
You have typed a character that is only permitted in {text}, comments, and character string constants.

bad data: please correct
The input data to your program is not grammatical and must be corrected before the program execution can continue. Input data already read by the program is not displayed. The line can be corrected as if it were a phrase. In particular, use <left> and <right> to move the cursor, use <clear> or <rubout> to erase characters, use <delete> to delete the entire line, type wherever insertions are desired. Any cursor motion that normally causes the cursor to leave a phrase, resumes execution with the re-edited line as input.

bad output param
The second parameter of the ".ex" command, if provided, must be PRT.

bad syntax
The phrase you have typed is not grammatical. Its punctuation might be incorrect, parenthesese might be mismatched, a scalar variable might be used as an array, etc.

cannot edit
The first argument of the command ".ed" must be the name of a file.

cannot insert
You have asked to insert the contents of a file (perhaps CLIPPED) at the position of the editing cursor. You cannot insert it either because it does not exist or because it has the wrong syntactic structure. A temporary restriction forbids inserting a SEGMENT of one procedure into the body of another procedure.

cannot remove
You have asked to clip, move, or delete a file element but are positioned at a placeholder so there is nothing to remove.

char variable required
The first argument of the pseudo-variable substr must be declared as a CHARACTER variable.

dimensions mismatch
The assignment of one array to another is only permitted if both arrays have the same number of dimensions.

divide by zero
Your program has attempted to divide by 0.

empty object
You have asked to insert the contents of some file (perhaps CLIPPED) at the position of the editing cursor. The file to be inserted is currently empty.

end of file on input
Your program has read all the way through the input file and is still requesting more input data.

external variable has wrong type
An external variable has already been declared to have one type and now a procedure has been called in which that external variable is declared to have some other type.

illegal divide
Your program has attempted to divide by 0.0e0.

illegal parameter
You have typed a command with an incorrect parameter.

improper constant
You have typed a constant that is not well-formed or is too large.

incr is 0
The increment of a counting DO-loop must not be zero.

input not text
The file you have specified as the input file for an execution is not a text file or does not exist. Recall that the ".ex" command assumes that the main procedure you wish to execute is the most recent one to have been edited. Thus, you do not mention its name in the command. The only file name the ".ex" command expects is the name of the input file, which must be a TEXT file.

invalid phrase
The execution of your program has encountered a syntactically invalid phrase. Execution is suspended.

line limit exceeded
You have printed too many lines on the printer. Execution is suspended.

lo bound > hi bound
In the declaration of an array, you have specified a low subscript bound that is larger than the high bound.

lower bound required
In the declaration of an array, the specification of both a low and a high subscript bound is required for each dimension.

max length < 0
In the declaration of a CHARACTER variable, the maximum length cannot be negative.

mismatched array bounds
The assignment of one array to another is only permitted if both arrays have the same subscript bounds. Your program has attempted to assign the contents of one array to an array with different subscript bounds.

missing */
You have typed "/*", the beginning of a comment, but have not provided the "*/" required at the end of a comment.

missing parameter
You have typed a command requiring a parameter you have not provided.

missing program element
The execution of your program has encountered a placeholder that must be filled in before the execution can continue. It is possible to fill it in and then resume execution.

missing '
You have typed "'", the beginning of either a CHARACTER or BIT constant, but have not provided the "'" required at the end the constant.

mod by 0
The second argument of the builtin mod function cannot be zero.

most recent main proc gone
The most recent main procedure to have been edited has subsequently been deleted. Edit the file you wish to execute and then request execution.

negative ** float
The left operand of exponentiation cannot be negative when the right operand is a FLOAT expression.

no space left

The Synthesizer has run out of space. This can occur at during either editing or execution.

During editing: you must delete files to make room. Use the ".list" command to list the names of your files. (If the message recurs in response to ".list", delete the phrase of the file being edited.) Then delete each unnecessary file f by typing ".ed f" and striking <delete>.

During execution: the cursor will automatically be positioned at the offending declaration or statement. Most likely, you have tried to declare an array too large for the available space or a CHARACTER variable that is too long. You may have called a recursive procedure an unbounded number of times. Space for variables is allocated from a different storage pool from files. Thus, deleting files will <u>not</u> provide more space for variables. You will have to uses smaller arrays or shorter CHARACTER variables. (External and static variables are an exception to this rule. They are allocated from the same pool as files. Thus, by deleting files, you can free more space for such variables.)

no suspended env. for this proc

You have tried to look at the value of a variable, but there is no suspended environment for this procedure. When a procedure returns, its variables go away. Also, some editing modifications can cause a suspended environment to go away. You will have to execute the program again from the beginning.

nonexistent procedure called

Your program has called a procedure that does not exist.

not a command

You have typed a command that does not exist. You may have typed a space between the "." and the command name. You may have mistyped the command name.

not a possible split

The parameters to the ".split" command are illegal.

not at placeholder

An insertion from the file system is only permitted when the cursor is positioned at a placeholder. The cursor is currently positioned at a phrase or template. If you wish to make an insertion in front of the current cursor position, move the cursor <up> and then strike <RETURN> until reaching a placeholder where you wish to make the insertion.

not implemented yet

You have used a feature of PL/I that will eventually be incorporated in the Synthesizer, but is not yet there.

not permitted now
You have used a command to insert a template that is not syntactically correct at the current cursor position.

not possible to resume
You have tried to resume execution, but because of certain modifications made since the execution was suspended, it is necessary to execute the program from the beginning again.

on conversion: char to bit
An operation in your program requires that a value be converted from CHARACTER to BIT. The string value does have the form of a BIT constant. You may have unintentionally used the operator that requires BIT values. You may have incorrectly declared a variable to have type CHARACTER. You may not have realized that some earlier operation produced CHARACTER results.

on conversion: char to fixed
An operation in your program requires that a value be converted from CHARACTER to FIXED. The string value does not have the form of a FIXED constant. You may have unintentionally used the operator that requires FIXED values. You may have incorrectly declared a variable to have type CHARACTER. You may not have realized that some earlier operation produced CHARACTER results.

on conversion: char to float
An operation in your program requires that a value be converted from CHARACTER to FLOAT. The string value does not have the form of a FLOAT constant. You may have unintentionally used the operator that requires FLOAT values. You may have incorrectly declared a variable to have type CHARACTER. You may not have realized that some earlier operation produced CHARACTER results.

on conversion: fixed to bit
An operation in your program requires that a value be converted from FIXED to BIT. The FIXED value is neither 0 nor 1. You may have unintentionally used the operator that requires BIT values. You may have incorrectly declared a variable to have type FIXED. You may not have realized that some earlier operation produced FIXED results. Note that the Synthesizer's conversion from FIXED to BIT is not standard PL/I.

on conversion: float to bit
An operation in your program requires that a value be converted from FLOAT to BIT. The FLOAT value is neither 0.0e0 nor 1.0e0. You may have unintentionally used the operator that requires BIT values. You may have incorrectly declared a variable to have type FLOAT. You may not have realized that some earlier operation produced FLOAT results. Note that the Synthesizer's conversion from FLOAT to BIT is not standard PL/I.

on conversion: float to fixed
An operation in your program requires that a value be converted from FLOAT to FIXED but the magnitude of the FLOAT value is too large to

represent in FIXED.

only commands permitted here

When the cursor is positioned at a template, you are not permitted to type phrases -- you may only type commands or use the special function keys. Similarly, some placeholders can only be replaced by templates, not phrases. You may have forgotten to strike <command .> or may have typed a regular period by mistake. If you intended to insert a phrase in front of the current cursor position, then move the cursor <up> and then strike <RETURN> until reaching a placeholder where you wish to make the insertion.

out of space - phrases may be truncated

The Synthesizer has run out of space. You must delete something in order to make room. The change in a declaration just made has changed the validity of at least one phrase in the current procedure. Unfortunately, it may have been necessary to truncate characters from the end of some of those phrases for which there was no room.

phrase too long

You have typed a phrase that is too long for one of the Synthesizer's internal storage areas. Please make it smaller.

reserved word

You have typed a word that occurs in one of the predefined templates. PL/I statements that can be inserted as templates, cannot be typed as phrases. If you intended to insert that statement, use the appropriate command to insert the corresponding template. If you intended the word as a variable name, use a different name.

subscript out of bounds

One of the subscripts of an array reference is outside the bounds with which the array was declared.

substr length < 0

The third argument of substr, the length of the desired substring, must not be negative.

substr length too big

The third argument of substr, the length of the desired substring, is too big. The substring specified must be part of the string given in the first argument; it cannot extend past the end of the original string.

substr start < 1

The second argument of substr, the position of the first character of the desired substring, is less than one. The substring specified must be part of the string given in the first argument; it can not start before the original string.

substr start too big

The second argument of substr, the position of the first character of the desired substring, is too big. The substring specified must be part of the string given in the first argument; it can not start after the end of the original string.

this phrase invalid in a segment
Declarative phrases are invalid in a SEGMENT even if syntacticly correct. Just strike <RETURN>, you did nothing wrong. The phrase will be validated when inserted into a procedure.

too many symbols
The Synthesizer has run out of space for symbolic names. You must delete some names in order to make room. Use the ".list" command to list the names of all your files. Delete each unnecessary file by editing it and striking <delete>. Delete unnecessary variables from procedures.

trap
If you were booting a disk when you encountered this message, then your disk may be dirty or damaged; the microcomputer may be dirty of damaged. Try several times to boot the disk; try to boot the disk on a different microprocessor.

If your disk had already successfully booted, then you have encountered either a hardware malfunction or an internal error in the Synthesizer. You are now in its internal debugging routine and must re-boot your disk to continue. All work since the last ".off" or ".save" is lost.

unable to write on disk
There has been an error trying to save your work on disk. You may have removed your disk from the drive too soon; your disk may be dirty or damaged; the disk drive may be dirty or damaged.

undeclared parameter
You have used a parameter that is listed in the PROCEDURE heading but was never declared.

undeclared variable
You have used a name that was never declared. The declaration of a name must be provided before you can use that name in a program. You may have forgotten to declare the variable, you may have mistyped the name of a declared variable, or you may have used a PL/I builtin function not implemented in the Synthesizer.

unimplemented builtin
You have used a builtin function that has not yet been implemented in the Synthesizer.

unimplemented feature
You have used a feature of PL/I that is not yet implemented in the Synthesizer.

uninitialized variable
Your program has tried to use the value of a variable that does not currently contain a value. Two explanations are possible: 1) a

value was never assigned to the variable, 2) the variable was used as the control variable of a DO-loop and was set to uninitialized after that DO-loop terminated.

wrong # arguments
    You have used a builtin function with the wrong number of arguments.

wrong # subscripts
    The number of subscripts used does not match the declaration of the array.

#args ~= #parms
    The number of arguments given in the call of a procedure does not match the number of parameters in the definition of the procedure.

1, 2, 3, 4, 5, or 6
    Errors internal to the Synthesizer. You have done nothing wrong but the Synthesizer has.

## E. KEYBOARD LAYOUT

(cut here)

| | |
|---|---|
| execute | |
| clip | command . |
| insert | LOCK |
| CTRL | SHIFT |

(left side)

| | | | | |
|---|---|---|---|---|
| ↖ | Δ | { | } | break |
| | ∇ | ~ | resume | unused |
| RETURN | ◁ | ... | delete | clear |
| SHIFT | ▷ | long | | rubout |

(right side)

(cut here)

**INDEX**