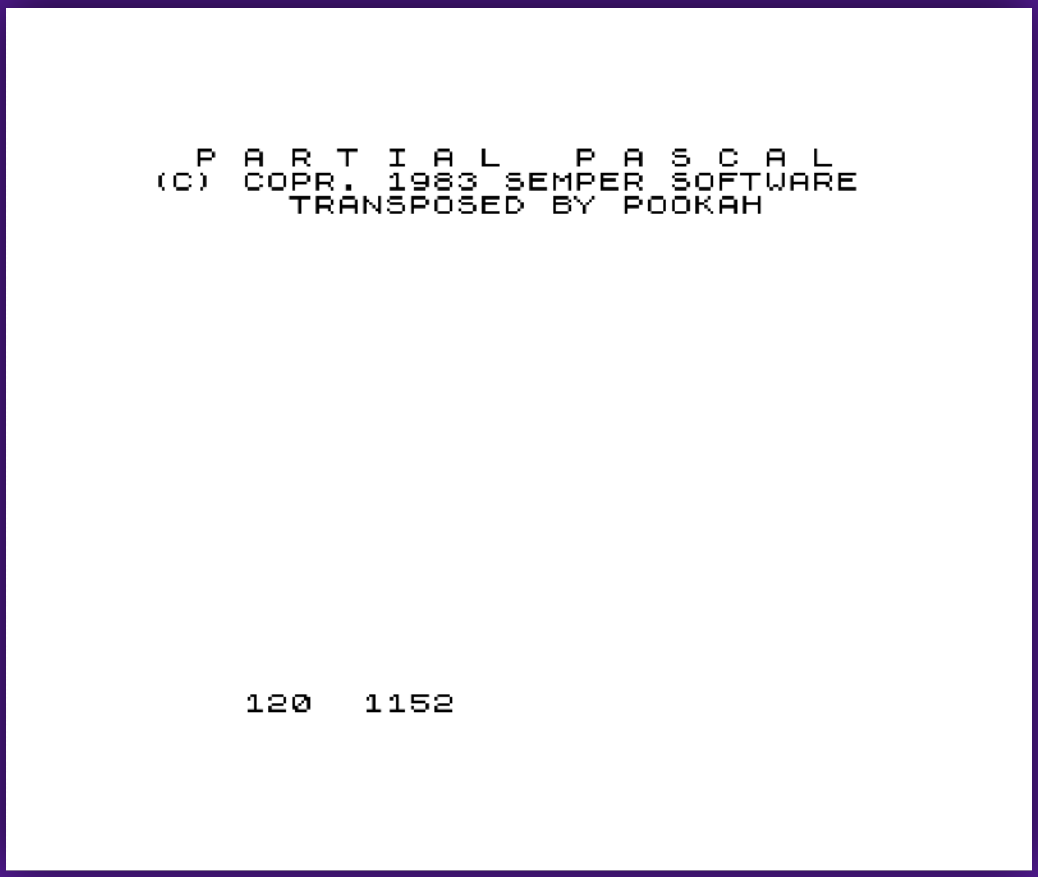
Semper Software

Partial Pascal

for the
Sinclair ZX-81, Timex/Sinclair 1000
and Timex/Sinclair 1500



Semper Software
585 Glen Ellyn Place
Glen Ellyn, IL 60137

Partial Pascal

Pascal is a computer programming language, very popular on microcomputers, invented by Professor Niklaus Wirth of the Swiss Institute of Technology. Partial Pascal is a subset of Pascal for the ZX81, Timex Sinclair 1000 and 1500.

Partial Pascal includes **IF, THEN, ELSE, CASE, OF, OTHERWISE, WHILE, DO, REPEAT, UNTIL, FOR, TO, DOWNTO, BEGIN** and **END** for program control; *read readln, write, writeln, reset, rewrite, eoln, eof, inkey* and *text* for input and output; +, -, *, **DIV, MOD,** *abs, chr, odd, ord, pred, succ* and *sqr* for calculations; **NOT, AND** and **OR** for decisions; **PROCEDURE, FUNCTION** and **FORWARD** for subroutines; **CONST, TYPE, VAR, ARRAY,** *Boolean, char* and *integer* for data; *copy, fast, slow, pause* and *halt* for computer control; *plot* and *point* for graphics; and *mem, mem2, memw, move* and *usr* for machine language.

Partial Pascal executes much faster than BASIC because, as a compiled language, it doesn't have to search thru tables to find variables or search thru line numbers as BASIC does for each **goto, gosub** or **next.** Partial Pascal's 16-bit integer calculations are much faster than BASIC's arithmetic.

Please note our new address. Partial Pascal is supplied on cassette tape with instruction manual. 16K RAM required. $30 postpaid from

Semper Software
585 Glen Ellyn Place
Glen Ellyn, Illinois 60137

Chapter 1. Introduction and Loading

1.1 Manual Design

Chapters 1, 2 and 3 should be read in order at the computer while using Partial Pascal for the first time. Chapters 1, 2 and 3 cover the mechanics of using the editor, compiler and loader by demonstrating the composition, compilation and execution of a sample program. Chapters 4 and 5 complete the description of the Partial Pascal language and chapter 6 complete the description of the editor. You may want to skip chapter 6 early to see all the editor has to offer.

1.2 Terminology and Fundamentals

Pascal is a computer programming language invented by Professor Niklaus Wirth of the Swiss Institute of Technology. Intended as a systematic programming language to improve the teaching of computer science, it has been a tremendous success both in teaching and in practical applications and is very popular on microcomputers.

A Pascal compiler connects two classes of "program." A "source program" is composed of readable characters: letters, digits, spaces and punctuation marks. An editor, itself a computer program, reads in a source program and writes out an "object program." The object program is executable by your computer. When this manual talks about "your program" in the context of composing, it means the source program that you write using the editor. In the context of execution, it means the object program written out by the compiler.

Pascal programs write all their output to files and read all their input from files. A file has a name used in a source program to represent an input or output device connected to your computer. "Input" and "read" apply to transfer of date to the computer from some less central (i.e. peripheral) device. "Output" and "write" apply to transfer of data from the computer to a peripheral device.

In Partial Pascal, the selection of which devices will be used with which files is postponed until the program in its object form is about to begin execution.

Data saved on tape by Partial Pascal programs, including that written by the editor and compiler, is given a name at the time it is written out. When a program writes data to a file for which you've selected the tape recorder as the "device," Partial Pascal saves the data in a buffer in memory until either the buffer is full or the program indicates it has no more date to write. Partial Pascal then asks you to supply a 12-character name for the data. Partial Pascal records a header, which includes the name you supply, and the data. This manual calls the header and data on the tape a

"data set," and the name in the header is the "data set name." The data set name serves to distinguish among the data sets on a tape.

When a program has to read data from a tape, Partial Pascal asks you for the data set name you provided when the data was recorded. Partial Pascal searches the tape for a data set with that name and reads the data from it into a buffer and provides the data to the program from the buffer.

1.3 Loading Partial Pascal

For a ZX81 or Timex Sinclair 1000, install the 16K RAM pack on the back of the computer. Connect the TV and power cables normally. Connect the cable between the tape recorder's earphone jack and the computer's EAR jack. Insert the tape and rewind fully. See section 5.11 if you have more than 16K of RAM or you wish to reserve space for your own machine language routines.

Type:

```
LOAD ""PASCAL""
```

Using the LOAD keyword on the J key. Press ENTER, then press play on the tape recorder. The first recording on the tape is a BASIC program called "PASCAL". The BASIC program starts automatically and invokes a machine language routine that reads in Partial Pascal from the second recording on the tape.

During loading four patterns should display on the TV.

1. The TV display should first show the pattern it normally shows when searching for a BASIC program.
2. It should then show the pattern normally seen when loading a BASIC program.
3. The third pattern is new. It lasts for two seconds and consists of narrow black lines narrowly spaced a few degrees clockwise from horizontal. This is the pattern produced by Partial Pascal searching for a recording.
4. The fourth pattern is produced by Partial Pascal reading data from a tape. It consists of unstable basically horizontal, thick black lines liberally peppered with short (1") thin horizontal black lines.

1.4 Loading Problems

Partial Pascal may display an error message during loading. If the message is ERROR 1 or ERROR 2 then 16389 (RAMTOP) was poked with a value less than 111 before Partial Pascal was loaded. Loading Partial Pascal without the 16K RAM

pack in place has the same effect. If you haven't poked RAMTOP then check that the 16k RAM pack is in place by issuing the BASIC command:

```
PRINT PEEK 16389
```

The value printed should be 128.

Other loading problems are due to incorrect data transfer from the tape to the computer. The first thing to do is to try again from the beginning using the other side of the tape.

If the message `ERROR 3` is displayed then Partial Pascal has recognized that one or more bits were read incorrectly from the tape. Try the load over again from the beginning. If this error happens more than once, return the tape to Semper Software for replacement. The loading of Partial Pascal consists of two main tasks:

1. Loading a BASIC program called "PASCAL" which contains a small machine language loader and
2. Using that machine language loader to load the bulk of Partial Pascal.

Most loading problems occur when loading the BASIC program. If the BASIC program has not been loaded correctly, then the third and fourth patterns displayed on the TV will not be as described in section 1.3 but rather will be the patterns of BASIC reading the tape since the

```
LOAD ""PASCAL""
```

Will still be looking for the BASIC program called "PASCAL".

To determine unambiguously whether the BASIC program has indeed loaded, press the SPACE key while the 4½ minute part of the tape is playing. If BASIC is still looking for "PASCAL", this will cause a BREAK and BASIC will clear the screen and display the inverse K. In this case, retry with different volume settings. You may want to retry using:

```
LOAD
```

Rather than:

```
LOAD ""PASCAL""
```

Although this seldom helps.

If the BASIC program has loaded correctly, the SPACE key will have no effect because the machine language loader ignores it. In this case, the tape may have been stretched or exposed to a magnetic field during shipping. The TV would show

the characteristic pattern of Partial Pascal reading data from the tape, the fourth pattern described in section 1.3, for about 4½ minutes, and then revert to the third pattern.

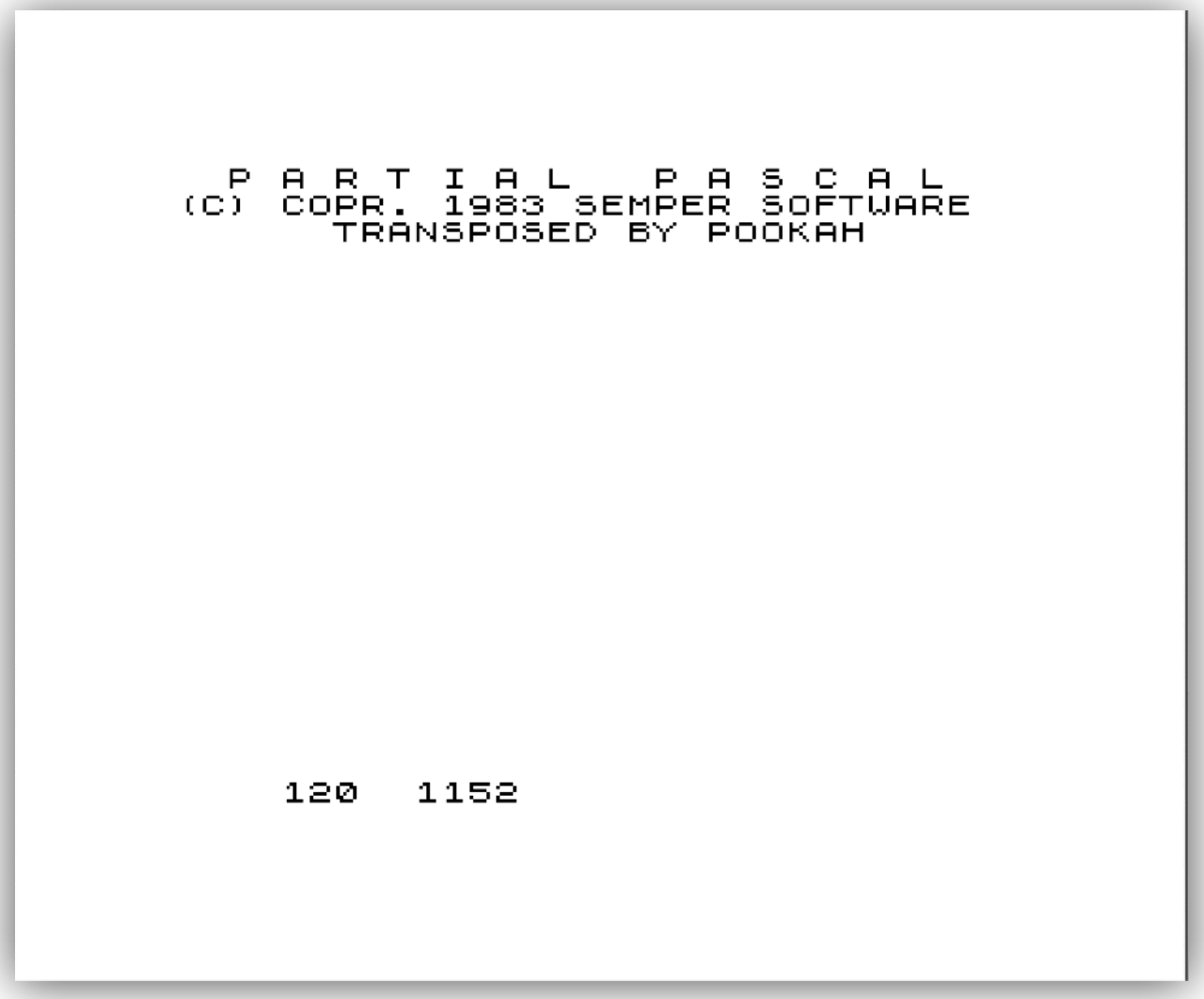
If neither side of the tape will load, return it for replacement to Semper Software. To aid in isolating the problem, we would appreciate a description of the computer and tape recorder you are using what happens when you try to load Partial Pascal.

Chapter 2. Start with the Editor

2.1 Selecting the Editor

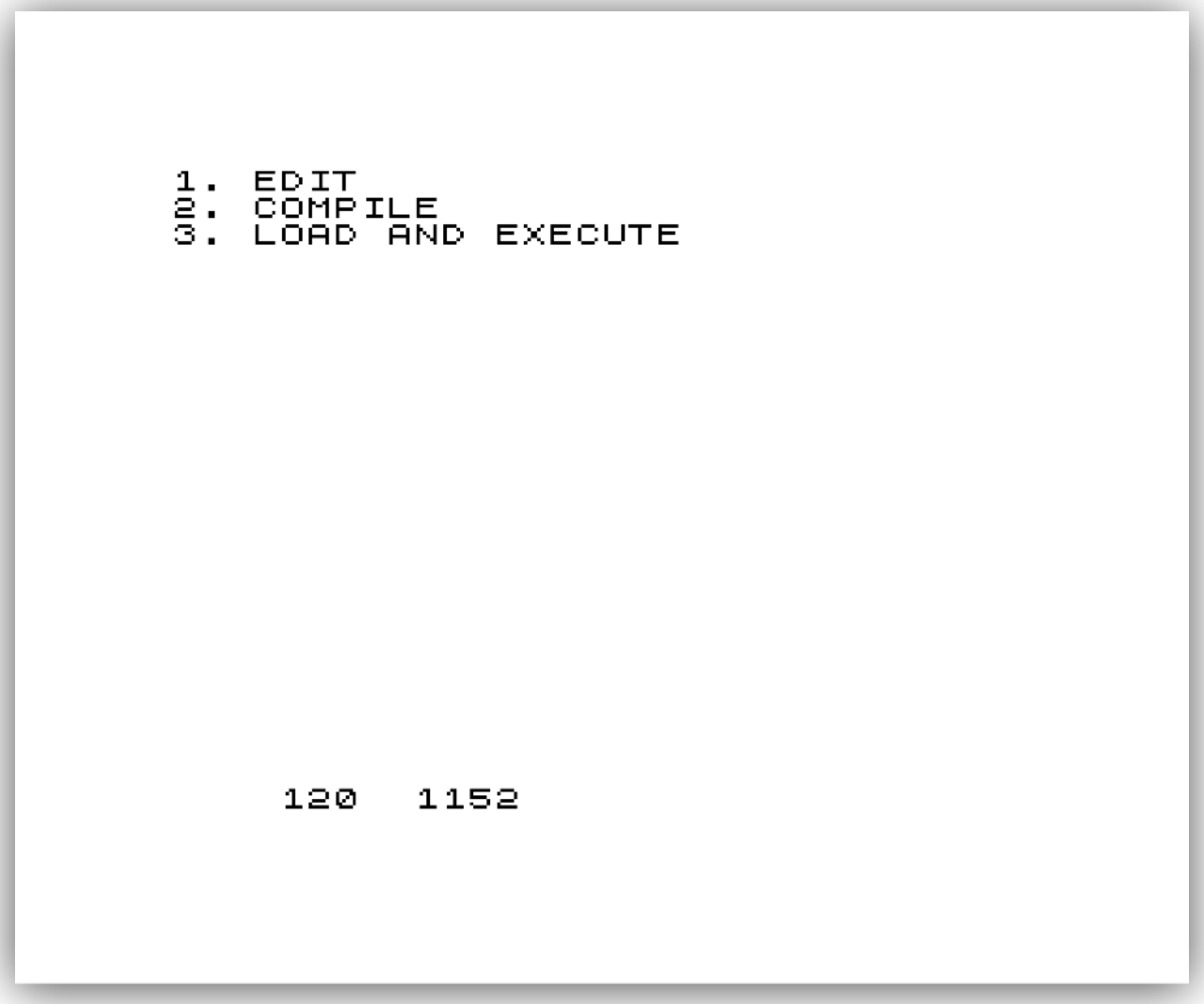
In this chapter we will begin to describe the editor. The editor is more fully described in chapter 6. This chapter will explain just enough of the editor to write some programs.

When loaded, Partial Pascal shows the message:



On the top two lines of the TV and some numbers on the bottom line. These numbers will be explained in chapter 5.

Press any key to continue with Partial Pascal. The original display is replaced by a selection display.



Press the 1 key to select the editor, Partial Pascal then, as it does when beginning any program, asks you to select devices for this execution of the editor. It does this by showing a pair of quotes on the last line of the TV with three spaces between them.

'' ''

Type ''SNT'' inside the quotes, (''SNT'' will also work for the compiler and the loader. The devices you can select are described fully in section 4.15.) The slowly flashing cursor is the normal Partial Pascal cursor. Every key repeats automatically when typing with the slowly flashing cursor, so be careful not to hold a key down too long unless you want it to repeat.

The editor will start as soon as you type the third character of ''SNT''. If you've mistyped the first or second character, you can backspace over them by pressing 0 while holding down SHIFT. If you've mistyped the third character, the editor will start with the wrong device and you should press Q while holding down SHIFT to cause it to stop, then press any key to get the selection and 1 to start the editor again.

The editor takes about 4 seconds to measure the amount of memory available, then starts by displaying the contents of its memory (there will be no content if this is the first time since Partial Pascal was loaded) and rapidly flashing its cursor. Keys do not automatically repeat when the editor is executing.

2.2 Controlling the Editor

The editor uses the first 22 lines of the TV display to show 22 lines of the source program being executed. On the 23rd line, the editor shows two numbers. The first is the number of memory bytes still available for use by the editor. The second is the number of source program lines that are not visible because they come before the top line displayed on the TV. The 24th line is used by Partial Pascal and is not under the control of the editor.

The editor uses a rapidly flashing "cursor" to show where the next character you type will appear. Press the alphabetic and numeric keys without shift to enter letters and digits. Type:

```
PROGRAM XYZ
```

Using the space key after the letter M. Unlike BASIC, all words in Partial Pascal must be spelled out letter by letter. Certain words, like PROGRAM, have reserved meaning in Pascal programs. These words are printed in boldface in this manual. They are typed in letter by letter just like words that are not reserved.

The keys with red punctuation marks (black on the Timex Sinclair 1500) on them are pressed while holding down the SHIFT key to enter the punctuation marks. Type I while holding the SHIFT key.

```
PROGRAM XYZ(
```

Release the SHIFT key and type

```
PROGRAM XYZ(INPUT
```

The comma is entered by pressing the period key while holding SHIFT. Release SHIFT to type output.

```
PROGRAM XYZ (INPUT, OUTPUT
```

Then use the O and X keys with SHIFT to finish the first line of our first program.

```
PROGRAM XYZ (INPUT,OUTPUT);
```

The 5, 6, 7 and 8 keys, when used with SHIFT, move the cursor without changing the display or any data. Move the cursor back several characters by using the 5 key with SHIFT held down until the cursor is over the letter X. As it stands, XYZ is the name of the program. Change the name of the program to your initials by typing these over the X, Y and Z.

```
PROGRAM SPP (INPUT, OUTPUT);
```

SPP are the initials of Semper's Partial Pascal. Letters, digits, punctuation marks and spaces replace the characters they are typed over. The 6 key, pressed with SHIFT held down, moves the cursor down one line. Use it to move the cursor down one line, then use the 5 key with SHIFT to move the cursor to the left margin. Now type:

```
PROGRAM SPP(INPUT,OUTPUT);
BEGIN
```

BEGIN is used in Pascal to mark the end of the declarations (there aren't any in this program) and the beginning of the executable statements. Now press the ENTER key. The ENTER key moves the cursor in a friendlier way than the arrow keys. The ENTER key always moves the cursor down exactly one line, but also moves it left or right to the editor's best guess as to where you want to start typing or that next line. Sometimes the editor's guess is not very good. In this case, it takes the cursor just below of B of BEGIN, only one space away form where the next line begins, and saves us four presses of the 5 key. Now use the space key once and type:

```
PROGRAM SPP(INPUT,OUTPUT);
BEGIN
WRITE("HELLO, WORLD")
```

Press the ENTER key, then hold SHIFT while pressing the 5 key once. Then type:

```
PROGRAM SPP(INPUT,OUTPUT);
BEGIN
WRITE("HELLO, WORLD")
END.
```

As an exercise in cursor agility, move the cursor to the BEGIN line and type:

```
PROGRAM SPP (OUTPUT);
BEGIN (* FIRST PROGRAM *)
WRITE ("HELLO, WORLD")
```

```
END.
```

What we have just added is a comment. Comments allow the author of a program to communicate to the readers of a program without any interference from the compiler. There are no rules about the contents of comments because the compiler ignores them. A comment begins with a `(*` symbol (no space is allowed between the opening parenthesis and the asterisk) and continues through the following `*)` symbol (again, no space), possibly spanning several lines. A comment may be placed in a program anywhere outside of a quotation that a space may be placed. This completes our first Pascal program.

2.3 Save Me

Now that the first program has been written, we should save it on tape. Place a blank tape in the recorder, play it for 5 seconds to get past the first few inches where tape quality problems are more common. Connect the tape recorder's microphone input jack to the computer's MIC output jack.

Now press `S` while holding down SHIFT. The editor will write the contents of its memory to its third file. When the editor started, we selected the tape recorder for editor's third file by typing the `T` in SNT.

Partial Pascal stores the date from the editor in a buffer in memory, then, before recording, asks what name to give this data.

```
O3 ''          ''
```

The last line of the TV display may not look like a question, the letter `O` indicates a query for an output data set name, the digit 3 indicates that the output data was written to the editor's third file, and the 12 spaces between the quotes are for you to type in the name for the data set.

It's a good idea to have a convention for naming data sets on a tape, to make the names easier to recall. In this manual the example data set names are composed from four parts: the name of a program, a sequence number to make the name unique, a period and a trailer that distinguishes source program from object program. For this data set the name is "SPP1.PAS." You may of course use any name of up to 12 characters. Unlike names in Pascal programs, punctuation marks, inverse video and graphics characters are allowed.

If the name you choose, like this one, has fewer than 12 characters, Partial Pascal will start recording when you press ENTER after the last character. If the name has a full 12 characters, Partial Pascal will start recording when you press the 12th character.

```
O3 "SPP1.PAS"
```

Type the name press RECORD and PLAY simultaneously on the tape recorder, then press ENTER or the 12th character of the data set name. While Partial Pascal is recording on tape the TV should look as it looks when saving a BASIC program. When the editor's display returns (in about 8 seconds), turn off the tape recorder.

2.4 Get That Source Program

Type over something, anything, in the program. (It will be restored when the editor reads the program from tape.) Now press G while holding down SHIFT. The editor asks or the top line.

```
LOSE CHANGES (Y/N)?
```

This question is a safety feature of the editor. The SHIFT G asks the editor to throw away the current contents of its memory and replace it with what it will read from tape. Since we have made changes to the editor's memory and not saved those changes the editor is verifying that we really want to discard memory and didn't press SHIFT G accidentally.

Press Y, then press ENTER. This confirms to the editor that we really meant what we told it. If you press N then ENTER, or just press ENTER, the editor will leave its memory intact and act as if SHIFT G had never been pressed.

Partial Pascal asks for the name of the data set we want to read. The last line becomes

```
I3"    "
```

The I3 asks what data set name Partial Pascal should read for input to the editor third file, the tape file. In the quotes, type the name you just typed when we saved the source program.

```
I3 "SPP1.PAS "
```

Press ENTER on the computer, and the TV should display the same narrowly spaced lines it showed for two seconds when we loaded Partial Pascal. Rewind the tape and press play. Partial Pascal will find the data set and provide the data to the editor. The pattern on the TV should be similar to the on displayed during the 4½ minutes Partial Pascal was loading. When that loading pattern disappears, stop the tape recorder.

If you type the wrong name, you can use the Ø key while holding down SHIFT to backspace within the quotes. If you discover you've typed the wrong name after

Partial Pascal has begun searching the tape, press the A key and Partial Pascal will ask on the last line for a new date set name.

When the editor has the source program back from the tape, it will display it on the TV and the overtyping we did just before reading from the tape should be gone.

2.5 I Quit

Now stop the editor by pressing Q while holding down SHIFT. If we had made any changes to the editor's memory since our most recent Save or Get, the editor would have asked if we really want to quit saving the changes on tape by asking

```
LOSE CHANGES (Y/N)?
```

When the editor ends, Partial Pascal displays two numbers on the last line of the display. Partial Pascal does this when any program ends. The first number gives a reason why the program is ending. 255 means the program ended normally. Any other number means the program ended due to an error (or by invoking the halt built-in procedure. See section 7.2 for program completion codes). Partial Pascal leaves the top 23 lines of the display as they were when the program ended. They usually have output from the program that just ended.

Press any key to return to the same selection display from which we selected the editor in section 2.1. if you press 1, the editor will resume editing the source program in its memory (it's still there) and you can finish this chapter now. If you're rather advance to chapter 3 and come back to finish this chapter later, press 2 to invoke the compiler.

2.6 A Change of Scene

With the source program safely on tape we can be more cavalier with it. Press 3 while holding SHIFT. Did the program disappear? Yes. Will we have to read it back in from tape? No. the editor had been displaying the first 22 lines of the program. Now it is displaying the 12th thru 33rd lines, which are all spaces. Note that the 0 on the 23rd line of the TV has changed to an 11. There are now 11 lines "above" the TV display which cannot be seen and the program is among them. Press 4 while holding SHIFT to return to the previous point of view.

The 3 and 4 keys from a pair like the 6 and 7 keys. In both pairs, the left key is to access lower lines in the program and the right key to access higher lines in the program.

2.7 Inversions

Inverse video (white on black) characters and the graphics characters may be entered using the editor. Pressing 9 while holding SHIFT puts the keyboard into graphics mode. Whenever the keyboard is in graphics mode, Partial Pascal shows the inverse video G as the first character of the last line of the TV display. When keyboard is in normal mode, the first character of the last line is a space.

When the keyboard is in graphics mode letters, digits punctuation marks and even space key are entered in inverse video and the graphics characters may be entered pressing the keys with graphics characters on them while holding the SHIFT key. (See section 4.15 for a note on using the A key with SHIFT.)

To return to normal mode press the 9 key again while holding the SHIFT key. The editor cannot recognize any keys normally used to control it, such as the cursor control keys 5, 6, 7 and 8, when the keyboard is in graphics mode, because these keys are used to enter the graphics characters. The keyboard must first be turned to normal mode by pressing 9 while holding SHIFT.

2.8 Summary

These keys control the editor if used with SHIFT when not in graphics mode:

3: Move the displayed area 11 lines farther from the top.

4: Move the displayed area 11 lines closer to the top.

5: Move the cursor one space to the left. If the cursor is already at the left margin, move it to the right margin one line up.

6: Move the cursor one line down. If it is already on the 23rd line, move it to 1st the line.

7: Move the cursor one line up. If it is already on the top line, move it to the 23rd line.

8: Move the cursor one space to the right. If it is already at the right margin take it to the left margin one line down.

9: If in normal mode, enter graphics mode. If in graphics mode, enter normal mode.

0: Removes the character at the cursor.

These keys that control the editor when used with SHIFT only function in normal mode.

Q: Quit editing. If any changes have been made since the last save or get (or since editor started, if there haven't been any saves or gets), the editor will ask if changes should be lost.

S: Save whatever is being edited. To perform the save, the editor copies from its memory to its third file, the one for which you selected tape by typing T as the third character in SNT.

G: Get something to edit from tape. If any changes have been made since the last save or get (or since the editor started if there haven't been any saves or gets), the editor asks if the changes should be lost. To perform the get, the editor erases everything it has in memory and replaces it with what it reads from its third file. The third file is the one for which you selected tape by typing the T in SNT.

The 1, 2, 0, E, F, T, Y, A, and D keys control the editor in ways described in chapter 6.

Chapter 3. Minimal Partial Pascal

3.1 Compilation

To compile the program in the editor's memory, press `2` from the selection display from which you previously pressed `1` to select the editor. Then type `SNT` to select devices for the compiler's three files:

```
"SNT"
```

The compiler writes each character of the source program to the TV screens as it comes to it. The compiler executes in fast mode so most of the time it's running there will be nothing displayed on the TV. At the end of each 22 lines of output, however Partial Pascal pauses for 3½ seconds to make sure lines don't scroll off the screen before you have a chance to see them.

This first program is not long enough to cause a pause so the first display from the compiler comes when it's done.

```
PARTIAL PASCAL COMPILER
(C) COPR. 1983 SEMPER SOFTWARE
PROGRAM SPP (INPUT, OUTPUT);
BEGIN (* FIRST PROGRAM *)
WRITE("HELLO WORLD")
END.
(*23*)
```

If you get only part of this display and an error number, you've just gotten a little ahead of the lesson. Proceed to section 7.1 which describes what to do when the compiler writes an error message.

What's that `(* 23*)`? The compiler inserts a comment into its listing of the program, which it writes to its first file, at the end of every subprogram and at the end of the main program giving the length, so far, of the object program. The total length of the object form of this program is 23 bytes.

Partial Pascal will be asking for a name for the output data set on the last line of the TV for the compiler's third file.

```
03 "   "
```

The conventional name we suggest is the name of the program, a sequence number, a period and the suffix OBJ to denote the object form of a program. If the last thing

you did was read SPP1.PAS in to the editor, the tape will be positioned to record SPP1.OBJ after it. Type the name inside the quotes:

```
O3 "SPP1.OBJ"
```

Press RECORD and PLAY on the recorder, then ENTER and Partial Pascal will record the object form of your program on tape.

3.2 The Loader

When the recording is complete, the compiler ends. Partial Pascal leaves the first 23 lines of the display as they were written by the compiler and puts the compiler's completion code (the first number should be 255 to denote normal completion) on the last line of the TV. Press any key to get the selection display.

Press 3 from the selection display to select the loader. The loader reads object data sets from tape and executes them. Partial Pascal will ask for devices for the loader on the last line of the TV.

```
"      "
```

Type `SNT` inside the quotes and the loader will begin. The first thing it will do is read in the object data set the compiler just wrote out. Partial Pascal asks for the name of the date set to be read in now on the last line of the TV:

```
I3 "            "
```

Type in the same name you specified when the compiler wrote out the object form of your program:

```
I3 "SPP1.OBJ"
```

Rewind the tape. You really need not rewind any further back then the beginning of the compiler's recorded output but this time rewind all the way, to before the beginning of the editor's output. Press ENTER then press PLAY. Partial Pascal will search the tape for the data set named "SPP1.OBJ".

When Partial Pascal comes across a data set other than the one it is looking for it puts the name of the data set it has found on the right side of the last line. When Partial Pascal finds SPP1.PAS, the source program saved by the editor displays.

```
I3 "SPP1.OBJ" SPP1.PAS
```

For 3½ seconds. This feature is helpful of you've forgotten the names of data sets on tape.

After skipping over SPP1.PAS the loader reads in SPP1.OBJ and starts it. The program SPP has only two files, so when it starts Partial Pascal asks you to select two devices for those files.

'' ''

Press S twice to select the Standard input and output devices, the keyboard and TV screen. The object program will immediately write.

```
HELLO, WORLD
```

On the first line of the TV. You have now composed, compiled and executed your first Partial Pascal program. Congratulations!

Partial Pascal freezes the display when the program ends. Press any key to return on the selection display.

3.3 Partial Pascal the Language

```
PROGRAM SPP (INPUT, OUTPUT);
BEGIN (* FIRST PROGRAM *)
WRITE (‘HELLO, WORLD’)
END.
```

The Pascal program header consists of the word PROGRAM, the name of the program, the parenthesized list of names of the program's files and a semicolon.

A name in Pascal begins with a letter. The second and succeeding characters of a same way he either alphabetic letters or numeric digits. A name may be as long as you like. (32 characters is a practical limit because a name cannot be continued from one line to another.) All characters of a name in Partial Pascal are equally significant.

The following words are not allowed as names because they have reserved meanings in Partial Pascal: AND, ARRAY, BEGIN, CASE, CONST, DIV, DO, DOWNTO, ELSE, END, FOR, FORWARD, FUNCTION, IF, MOD, NOT, OF, OR, PACKED, PROCEDURE, PROGRAM, REPEAT, WHEN, TO, TYPE, UNTIL, VAR and WHILE. Partial Pascal also has an extension to the CASE statement, OTHERWISE, that may not be used as a name. All of these reserved words are described in chapters 3, 4 and 5.

Unlike full Pascal, Partial Pascal does not have records, sets or gotos. The reserved words used for them in full Pascal, GOTO, IN, LABEL, RECORD, SET and WITH may not be used in Partial Pascal. The words FILE and NIL may not be used in Partial Pascal, but their purpose may be accomplished in Partial Pascal without using them.

After the program's header come the declarations. The next section begins the VAR declaration. The declarations are always ended by BEGIN, which marks the beginning of the program's executable statements. And END followed by a period must be used to mark the end of the program.

3.4 Declaring Integer Variables

A variable is a name for a portion of memory that will be used to store some piece of data when the program is executed. Integer variables can take on values that are whole numbers in the range -32768 thru +32767. They are used if your program needs to do any arithmetic. Variables in Pascal must be declared before they can be used.

A variable declaration starts with the reserved word VAR. It is followed by one or more names separated by commas, a colon, the type desired for the variables named in the list and finally a semicolon. Multiple lists of variable names and their types are allowed.

```
PROGRAM SOMEVARS (INPUT, OUTPUT);
VAR X, Y, Z: INTEGER; AB, C:
INTEGER;
BEGIN
END.
```

This program declares five variables (x, y, z, ab, c) of type integer. Variables of types other than integer are allowed in Partial Pascal. The other types are postponed to section 4.5.

3.5 Added Value

Variables in Pascal are not initialized to any particular value.

The assignment statement gives a value to a variable. The variable's name is followed by the assignment symbol, :=, which is followed by the new value. The new value may be specified by:

1. A constant, such as 15 or -127
2. A variable which has already been given a value.
3. A function, such as the absolute value function, abs, or
4. A formula combining any of the preceding.

```
PROGRAM ASSIGN (INPUT, OUTPUT);
VAR X, Y, Z: INTEGER;
AB, C: INTEGER;
BEGIN
X:= 365; (* A CONSTANT *)
Y:= X ;(* A VARIABLE *)
Z:= ABS (-Y);(* A FUNCTION *)
AB:= X+Y-Z; (* A FORMULA *)
C:= AB+37 DIV (ABS(X-Y)+1); (* ANOTHER FORMULA
*)
X:= Y MOD C; (* A NEW VALUE FOR X *)
END.
```

Formulas may be constructed using the symbols + for addition, – for subtraction, * for multiplication, DIV for division, MOD for finding a remainder and, if necessary, parentheses for indicating the order in which these operations should be performed. DIV performs a division and throws away any fraction in the result, e.g. `-7 DIV 3` is -2. The divisor of a DIV must not be zero. MOD is not a commonly seen operation. MOD gives the reminder of a division. The result of `A MOD B` gives the remainder of A divided by B. The result of MOD may be obtained by starting with the value of A and adding or subtracting the value of B as many times as necessary to get a result which is less than B and greater than or equal to 0. The divisor of a MOD must be greater than or equal to 1.

"Expression" is the general term for constant variable, function or formula. Expressions occur frequently in Pascal. An assignment statement has a variable name to the left of the assignment symbol (:=) and an expression to the right of it. The function `abs (x-y)` takes the absolute value of x-y. ABS may take the absolute value of any integer expression.

3.6 Write it out

The first example program showed Partial Pascal's write statement used to write a quotation. The write statement may also be used to write numbers whose value is given by an expression.

```
PROGRAM WRITEITOUT(INPUT,
OUTPUT);
VAR Y; INTEGER;
BEGIN
Y:=30;
WRITE ('IF EVERY MONTH HAD', Y,
```

```
" DAYS THERE WOULD BE ",
365 DIV Y, "MONTHS PER YEAR")
WRITE ("AND", 365 MOD Y, "DAY",
"S LEFT OVER")
END.
```

This program produces the following when the TV screen is selected for the output file.

```
IF EVERY MONTH HAS 30 DAYS, T
HERE WOULD BE 12 MONTHS PER Y
EAR AND 5 DAYS LEFT OVER
```

If you have more than one thing to write, use comma to separate them. Each "thing" must be either a quotation enclosed in quotes or an expression. (An expression may be a character as well as a number described in section 4.2.)

The above program shows `write` statements with both quotations and expressions. Since a quotation must end on the same line of the source program that it starts on, a `write` statement will sometimes have two quotations in a row. Each `write` statement picks up where the previous one left off, in mid-line or even in mid-word. The quote mark itself is represented in a quotation by two quote marks (i.e. press `P` twice while holding down SHIFT)

The `writeln` (pronounced write line) statement may also be used to write quotations and expressions. The difference between `write` and `writeln` is that `writeln`, after writing out all its data, finishes the line it ends on. A succeeding `write` or `writeln` will state on the first space of the next line. (`Write` behaves like BASIC's `PRINT` with a semicolon after everything printed while `writeln` behaves like `PRINT` without a trailing semicolon.)

```
PROGRAM WRITEITOUT(INPUT, OUTPUT);
VAR Y: INTEGER;
BEGIN
T:=29;
WRITELN ("IF EVERY MONTH HAD", T,
"DAYS");
WRITELN ('THERE WOULD BE", 365
DIV Y, "MONTHS PER");
WRITELN ("YEAR AND', Y MOD 365,
"DAYS LEFT OVER");
WRITE (" ""A"" IS THE LETTER",
"BEFORE" ""B"" ")
```

```
END.
```

Using `writeln` to improve the appearance of the output produces"

```
IF EVERY MONTH HAD 29 DAYS,
THERE WOULD BE 12 MOTHS PER
YEAR AND 17 DAYS LEFT OVER
"A" IS THE LETTER BEFORE "B"
```

Note that each number is displayed using 6 characters (4 or 5 spaces and 1 or 2 digits). You may specify the minimum number of characters to be used for an expression in a `write` statement by following the expression with a colon and another expression. Do not add an extra comma. For example,

```
PROGRAM WRITEITOUT (INPUT, OUTPUT);
VAR Y : INTEGER;
BEGIN
Y:=28;
WRITELN ("IF EVERY MONTH HAD",
Y:1, "DAYS ");
WRITELN ("THERE WOULD BE", 365
DIV Y:2, "MONTHS PER");
WRITE ("YEAR AND", 365 MOD Y:2,
"DAYS LEFT OVER")
END.
```

The expression only gives the minimum number of characters Pascal will use to represent the first expression. At least as many characters as are needed will be used. Y:1 for example, uses two characters when Y is 28.

```
IF EVERY MONTH HAD 28 DAYS,
THERE WOULD BE 13 MONTHS PER
YEAR AND 1 DAYS LEFT OVER
```

3.7 Blaise Can Read

Partial Pascal's read statement can read in numbers (and individual characters, as we shall see later). Each number read in has its value assigned to a variable. The variable's name is specified inside of the parentheses following the word `read`.

```
PROGRAM READANDWRITE
(INPUT, OUTPUT);
VAR D:INTEGER;
BEGIN
WRITE ("HOW MANY DAYS PER",
" MONTH?");
```

```
READ(D);
WRITE (365 DIV D, " MONTHS OF");
WRITELN (D:1, "DAYS EACH");
WRITE ("PLUS", 365 MOD D, "DAY",
"S MAKE ONE YEAR")
END.
```

When this program is executed the first thing it does is write a question on the TV screen.

```
HOW MANY DAYS PER MONTH?
```

Having executed the `write` statement, it goes on to the `read` statement. The cursor will blink on the screen where your answer will appear. This is on the same line as the question since the question was written using `write` rather than `writeln`. Type, say, `27` and press ENTER.

```
HOW MANY DAYS PER MONTH? 27
```

Partial Pascal assigns variable D the value you typed in, 27, and goes on to the next statements, which write the following.

```
HOW MANY DAYS PER MONTH? 27
13 MONTHS OF 27 DAYS EACH
PLUS 14 DAYS MAKE ONE YEAR
```

A single `read` statement can read values for several variables. Just separate the variables names with commas.

```
PROGRAM READS (INPUT, OUTPUT);
VAR DAYS, MONTHS; INTEGER;
BEGIN
WRITE ("HOW MANY DAYS SHOULD",
"MOST MONTHSHAVE AND HOW MANY",
"MONTHS SHOULD THERE BE?");
READ (DAYS, MONTHS):
WRITELN(MONTHS-1:1, " MONTHS",
"OF ", DAYS:1 " DAYS");
WRITELN("1 MONTH OF",
365-DAYS*(MONTHS-1): 4, " DAYS");
WRITE ("WOULD MAKE ONE YEAR")
END.
```

Like the previous program, this one when it executes, first writes a question.

```
HOW MANY DAYS SHOULD MOST MONTHS
```

```
HAVE AND HOW MANY MONTHS SHOULD
THERE BE?
```

There is no space between months and have. When a quotation or a number doesn't all fit on one line, Partial Pascal continues it on the next line.

The numbers to be read by the `read` statement may be typed in many different formats. First, you may press ENTER or SPACE as many times as you like. Then you may type a + or – sign. Finally, the number itself may begin with as many zeroes as you like. In any event, ENTER must be pressed after the last digit of the last number read. Here are several acceptable ways to enter the values 32 and 10 for DAYS and MONTHS into this program.

```
THERE BE? 32 10
```

Or

```
THERE BE? +32+10
```

Or

```
THERE BE? 32 +0000010
```

Or

```
THERE BE? 032
10
```

Or

```
THERE BE?
+032
+10
```

In the fourth example above, `10` may be typed on the second line either by pressing ENTER after the `32` or by pressing SPACE 10 times after the 32. The fifth example shows ENTER pressed before and after the `+032`, or the equivalent number of SPACEs.

After the value for DAYS and MONTHS are read, the program's execution continues with the `writeln` statements.

```
HOW MANY DAYS SHOULD MOST MONTHS
HAVE AND HOW MANY DAYS SHOULD THERE BE? 30 10
10 MONTHS OF 32 DAYS AND
1 MONTH OF 45 DAYS
WOULD MAKE ONE YEAR
```

The `readln` (pronounced read line) statement does everything that `read` does and one thing more: it causes any remaining characters on the line containing the last character read to be ignored.

```
PROGRAM READ(INPUT, OUTPUT);
VAR M1, M2, M3: INTEGER;
BEGIN
READ(M1);
READ(M2, M3);
WRITE(M1 :1, "*", M2:1, "*", M3 : 1,
"=", M1*M2*M3)
END.
```

Changing a `read` to `readln` gives a slightly different program.

```
PROGRAM READLINE (INPUT, OUTPUT);
VAR M1, M2, M3 :INTEGER;
BEGIN
READLN(M1);
READ(M2, M3);
WRITE(M1:1, "*", M2:1, "*", M3:1,
"=", M1*M2*M3)
END.
```

If we execute the above two programs, providing each with the same input.

```
2 3 4
```

Pressing ENTER after the `4`, the first program reads `2` as the value for M1, `3` as the value for M2, `4` as the value for M3 and writes its output.

```
2 3 4
2*3*4=24
```

In second program needs `2` as the value for M1, then ignored the rest of that line. Values for M2 and M3 must still be typed and another ENTER typed after them.

```
2 3 4
5 6
2*5*6=60
```

3.8 IFs, ANDs, OR

Pascal has several statements whose sole purpose is to decide whether or not to perform other statements. The `IF` statement decides whether one or two other statements or groups of statements should be executed.

```
PROGRAM TESTIF (INPUT, OUTPUT);
BEGIN
IF 1=2 THEN
WRITELN ("UH OH. 1 EQUALS 2");
WRITE ("COMPARISON COMPLETE")
END.
```

After the word `IF` comes a new kind of expression, an expression whose value is not numeric, but rather is either true or false. Expressions whose value is either true or false are called Boolean expressions after George Boole, an English logician who invented ways of working with expressions and variables of this type. In Pascal, Boolean expressions may be specified by:

1. A constant, like true.
2. A variable which has already been given a value of true or false (Boolean variables are further described later).
3. A function, such as the odd-number function `odd`, which tells whether the numeric expression it is given is odd or even (e.g. odd (6+7) is true and odd (0-12) is false).
4. A comparison, like = or <>. There are six possible comparisons: =, <>, <, <=, > and >=. <=, <> and >= must be typed into the editor using the L, M and N keys in combination with SHIFT held down. The R, T and Y keys control the editor and are not used to enter comparison symbols.
5. A formula combining any of the preceding with `AND`, `OR` and `NOT`. If comparisons are used in a formula they must be enclosed in parentheses. This is because Pascal treats something like"

```
A<12 AND ODD(A)
```

as

```
A<(12 AND ODD(A))
```

If you want to use the formula

```
(A<12) AND ODD (A)
```

Then you must code the parentheses.

After the Boolean expression comes the word THEN, and after the word THEN comes a statement or a group of statements that are only executed when the Boolean expression is true. The example above shows a writeln statement that will never be executed because 1 is never equal to 2.

An IF statement may also specify an ELSE and with it a statement or group of statements to be executed only when the Boolean expression is false.

```
PROGRAM IFELSE (INPUT, OUTPUT);
VAR TESTNUMBER: INTEGER;
BEGIN
WRITE("PLEASE ENTER A NUMBER:");
READ(TESTNUMBER);
IF ODD(TESTNUMBER) THEN
WRITE(TESTNUMBER, " IS ODD")
ELSE
WRITE (TESTNUMBER, " IS EVEN")
END.
```

The above program, when executed, first writes a request.

```
PLEASE ENTER A NUMBER:
```

The read statement waits for a number to be typed. Try -231.

```
PLEASE ENTER A NUMBER: -231
-231 IS ODD
```

The read statement gives the value – 231 to the variable TESTNUMBER. The IF statement uses the odd function to determine whether the number read is odd or not. Odd is true for -321, so the statement after THEN is executed and the statement after ELSE is not executed. For an even number:

```
PLEASE ENTER A NUMBER: 18
18 IS EVEN
```

The read statement gives the value 18 to variable TESTNUMBER, then odd is false for TESTNUMBER, the statement after the THEN is skipped over and the statement following ELSE is executed.

To execute a group of statements rather than just one statement after a THEN or ELSE, the group of statements must be bracketed using BEGIN and END. When BEGIN follows THEN or ELSE, it marks the beginning of a group of statements

that will either all be executed or all skipped over. END marks the end of the group of statements begun by BEGIN.

```
PROGRAM EVENGROUP (INPUT, OUTPUT);
VAR NUM:INTEGER;
BEGIN
WRITE(…PLEASE ENTER NUMBER. …);
READ(NUM);
WRITELN(… NUMBER ODD…);
IF NOT ODD(NUM) THEN BEGIN
WRITELN(NUM, … NO…);
WRITELN (NUM+1, … YES…);
WRITELN (NUM + 2, … NO…)
END ELSE BEGIN
WRITELN (NUM-1, … NO…);
WRITELN (NUM, … YES…)
END
END.
```

In this example, three writeln statements are executed and two writeln statements are skipped over when number is even NOT odd. When the number is odd, three writeln statements after THEN are skipped and the two after ELSE are executed. An execution of the program might produce the following display:

```
PLEASE ENTER NUMBER : 25
NUMBER ODD
24 NO
25 YES
```

Or like this:

```
PLEASE ENTER NUMBER 9876
NUMBER ODD
9876 NO
9877 YES
9878 NO
```

3.9 A Little While

One thing computers are very good at is looping: executing some sequence of statements over and over. Pascal provides three statements for looping. The first is the WHILE statement. The WHILE statement begins with the word WHILE followed by a Boolean expression followed by the word DO followed by a statement or a group of statements marked by BEGIN and END.

```
PROGRAM ONETOTEN (INPUT, OUTPUT);
VAR COUNTER: INTEGER;
BEGIN
COUNTER :=1;
WHILE COUNTER <= 10 DO BEGIN
WRITE (COUNTER);
COUNTER:= COUNTER+1
END;
WRITELN(''(ONE THRU TEN)'')
END.
```

The execution of a WHILE statement is controlled by its Boolean expression. When the Boolean expression is true, the statement(s) following the DO are executed and the Boolean expression evaluated again. When the Boolean expression is false, the statement(s) after the DO are skipped and execution continues with whatever follows.

In the above program, the assignment statement gives variable COUNTER the value 1. When the WHILE's Boolean expression is evaluated. 1<=10 is true, so the write statement is executed and the assignment statement after it gives COUNTER the value 1+1. The Boolean expression is evaluated again. 2<=10 is also true, so the write statement is executed again to write the number 2, the assignment statement gives COUNTER the value 3 and the Boolean expression is evaluated again.

This looping continues until on the last loop 10<=10 is true, the write statement writes the number 10, the assignment statement gives COUNTER the value 11 and the Boolean expression 11<=10 is false. Now that Boolean expression is false, the write and assignment statements are skipped and execution continues with the writeln statement. The entire output written by this program is

```
 1 2 3 4 5
6 7 8 9 10(ONE
THRU TEN)
```

3.10 Half a Colon is Better than None

Pascal statements have no particular relationship to lines in the source. You may place as many statements as well fit onto a line (although your program is usually cleaner with no more than the statement per line) or string out a single statement over many lines.

Pascal programs are always sprinkled with semicolons. A semicolon separates the PROGRAM heading from the rest of the declarations, the declarations from each other and the declarations from the word BEGIN. In the statement area, a

semicolon is used to separate any two consecutive statements. Sometimes a program has many statements but no two of them are consecutive. The statements contained in an IF THEN ELSE statement, for instance, are not consecutive with it or each other.

```
PROGRAM NONCONSECUTIVE
(INPUT, OUTPUT);
BEGIN
IF EOF (INPUT) THEN
WRITE ("NO DATA TO READ")
ELSE
IF MEMW (.16388.)<18000 THEN
WRITE ("MEMORY TOO SMALL")
ELSE
WRITE ("CONDITIONS OK")
END.
```

This program has two IF statements and three write statements, but no two of these five statement are consecutive so there are no semicolons.

Putting in an extra semicolon that is not required is not allowed in the declaration area. In the statement area extra semicolons are usually harmless, with four exceptions. The compiler will not allow a semicolon before the ELSE of an IF statement, before the OTHERWISE of a CASE statement, or before the END of a CASE statement. The fourth place is particularly dangerous because it is not caught by the compiler: after the DO of a WHILE statement. Much computer time can be wasted waiting for the following WHILE statement to stop looping.

```
PROGRAM FOREVER (INPUT, OUTPUT);
VAR COUNT, SUM. LIMIT : INTEGER;
BEGIN
WRITELN("HOW MANY NUMBERS",
"SHOULD I ADD");
READ (LIMIT);
COUNT:= 0;
SUM:= 0;
WHILE COUNT <= LIMIT DO; (*BAD*)
BEGIN
SUM:= SUM+COUNT;
COUNT:=COUNT+1;
END;
WRITE ("THE SUM OF INTEGERS")
"0 THRU", LIMIT, "IS", SUM)
```

```
END;
```

Its author intended this program to find the sum of some integers, but the semicolon after the word DO marks the end of the WHILE statement. When the read statement reads a value of zero or more. the WHILE's Boolean expression is true, the first time it is evaluated so the statement after the DO and before the semicolon is executed and the Boolean expression evaluated again. The statement after the DO and before the semicolon is a very small statement that doesn't do anything, so the Boolean expression will be true every time it is evaluated and the WHILE statement never stops looping. The pairing of the BEGIN and END that were intended to mark the group of statements for the WHILE statements doesn't help. Since it doesn't come after a THEN, DO, or ELSE, the compiler doesn't know that BEGIN was intended to group statements.

Chapter 4. Intermediate Partial Pascal

4.1 True or False

Variables of type Boolean may be declared in the same way as integer variables. A Boolean variable may have one of two values, *true* and *false*. An assignment statement for a Boolean variable has a Boolean expression after the assignment symbol, the same type of expression that follows an IF or WHILE. Boolean variables may be used in Boolean expressions either by themselves or combined with companions and other Boolean variables using AND, OR and NOT.

```
PROGRAM ANDNOT (INPUT, OUTPUT);
VAR EVEN, LARGE: BOOLEAN;
CANDIDATE ; INTEGER;
BEGIN
WRITE ("PLEASE ENTER SOME",
" NUMBERS");
READ(CANDIDATE);
WHILE CANDIDATE <> 0 DO BEGIN
EVEN:=NOT ODD (CANDIDATE);
LARGE:=CANDIDATE>999;
IF EVEN AND LARGE THEN
WRITELN ("BOTH")
ELSE
IF NOT EVEN AND NOT LARGE
THEN
WRITELN("NEITHER")
ELSE
WRITELN("ONE OR THE OTHER",
" BUT NOT BOTH");
READ (CANDIDATE)
END
END.
```

This program categorizes numbers as being even, large, both or neither. It has four Boolean expressions in it. The first, NOT odd (CANDIDATE) applies NOT to a Boolean function. The second assignment statement may look a little strange at first, with both an assignment symbol and a greater than sign in it. Its Boolean expression is a simple comparison, CANDIDATE >999. LARGE is made true if the number read as CANDIDATE is at least one thousand. The third Boolean expression combines two Boolean variables using AND. The fourth Boolean expression combines the two

Boolean variables in a different way. Both EVEN and LARGE must be false for the fourth Boolean expression to be true. An execution of the program could produce the following on the TV screen:

```
PLEASE ENTER SOME NUMBERS
```

The first `write` statement asks whoever is executing the program to type in numbers. We type in three numbers before pressing ENTER.

```
PLEASE ENTER SOME NUMBERS 9999 9
9 1234
```

The first read statement gives CANDIDATE the value 9999, since 9999>0, the group of statements following DO are executed. EVEN becomes false. LARGE becomes true. And EVEN AND LARGE is false. The first `writeln` statement is skipped and the second `IF` statement is executed. Its Boolean expression is also false, so the second `writeln` statement is skipped and the `writeln` statement following the second `ELSE` is executed.

```
PLEASE ENTER SOME NUMBERS 9999 9
9 1234
ONE OR THE OTHER BUT NOT BOTH
NEITHER
```

That completes the execution of the first `IF` so the next statement of the `WHILE`'s group is executed, the `read` that finishes the loop. That `read` begins reading where the first `read` left off and finds the value 99 for candidate.

Now the `WHILE`'s Boolean expressions is again found true. EVEN and LARGE are both false, the first `writeln` is again skipped and the second `writeln` executed.

```
PLEASE ENTER SOME NUMBERS 9999 9
9    1234
ONE OR THE OTHER BUT NOT BOTH
NEITHER
```

The `read` at the end of the loop gives candidate the value 1234. The `WHILE` loop is executed again because its Boolean expression is again true. The time the first writeln is executed and the other two is skipped over.

```
PLEASE ENTER SOME NUMBERS 9999 9
9    1234
ONE OR THE OTHER BUT NOT BOTH
NEITHER
```

```
BOTH
```

This time the read at the end of the loop has nothing to read, so the cursor flashes slowly again as we type last number and press ENTER.

```
PLEASE ENTER SOME NUMBERS 9999 9
9    1234
ONE OR THE OTHER BUT NOT BOTH
NEITHER
BOTH
```

4.2 What a Character

A char variable has a value that is a character (Partial Pascal does not have string variables, but there are `ARRAY`s of char described later). A character in Pascal is a space, a letter, a digit or a punctuation mark. A Partial Pascal char variable can also be an inverse video character, graphics character or other values that are not displayable on the TV screen.

A char expression can be specified by:

1. a constant single character (with one exception) between quote marks. Such as "A" or "?". (To specify quote mark as the character, use two consecutive quote marks inside quotes i.e """")
2. a char variable that has already been given a value or,
3. a function whose value is of type char, such as the built-in function `chr`, which gives the character corresponding to a Sinclair code value.

The `chr` function produces the character that corresponds to a number according to the Sinclair code used by the ZX81. Timex-Sinclair 1000 and 1500.
`Chr(25)=";"` and `chr(38)="A"`, for instance. The `ord` function gives the Sinclair code for a character. `Ord("C")=40` and `ord(".")=27`.

```
PROGRAM CHARACTER (INPUT OUTPUT) :
      VAR C:CHAR; S:INTEGER;
BEGIN
S:=ORD("7";
WHILE S<=ORD ("A") DO
  WRITELN (S:3," IS THE"
      "SINCLAIR CODE FOR ", CHR(S) );
WRITE (" PLEASE ENTER A" ,
    "CHARACTER");
READ (C);
WRITE ("SINCLAIR CODE FOR """, C,
```

```
""" IS ", ORD(C):3)
END.
```

This program writes:

```
35 IS THE SINCLAIR CODE FOR 7
36 IS THE SINCLAIR CODE FOR 8
37 IS THE SINCLAIR CODE FOR 9
38 IS THE SINCLAIR CODE FOR A
PLEASE ENTER A CHARACTER
```

If we press the comma key and then ENTER, it continues:

```
35 IS THE SINCLAIR CODE FOR 7
36 IS THE SINCLAIR CODE FOR 8
37 IS THE SINCLAIR CODE FOR 9
38 IS THE SINCLAIR CODE FOR A
PLEASE ENTER A CHARACTER
SINCLAIR CODE FOR "," IS 26
```

4.3 Just My Type

Partial Pascal does not limit you to the built-in types integer, char and Boolean (and text, which is described later). You may also define your own types! The simplest type to define is a subrange type. A subrange is written as two constants giving the lower and upper bounds of a range separated by the `..` symbol. `1..4` is a subrange of the integer type. In Partial Pascal, the integer type has values -32786 thru 32767.

1..4 has just 4 different values, 1 thru 4. The subrange type -32768..32767 is equivalent to integer in Partial Pascal. "A"..."C" is a subrange of the type char, with just three different values: "A", "B" and "C".

One way of declaring variables to be of a subrange type is to give the subrange in place of the name of a built-in type.

```
PROGRAM SUBRANGES (INPUT, OUTPUT):
VAR X1, X2, X3; "A"..."Z";
 NEWTONIAN:1..3;
 EINSTEINIAN:1..4;
```

Variables X1, X2 and X3 are all of the same unnamed type, the letters. Variable NEWTONIAN is of an unnamed type that has three values 1, 2 and 3. EINSTENIAN is also of an unnamed type, one that has the values 1,2,3 and 4.

The enumerated types are favorites of Pascal programmers. In an enumerated type, the values that form the types are names. In their declaration the names are separated by commas inside a set of parentheses.

```
PROGRAM ADVICE (INPUT,OUTPUT);
VAR BUILDING:(GARAGE, OFFICE,
HOUSE) ;
CH: CHAR;
BEGIN
WRITE ("IS THERE A CAR INSIDE",
  " (Y OR N)? ");
READ (CH);
IF CH="Y" THEN
BUiLDING:= HOUSE
ELSE
 BUILDING:= OFFICE
END;
IF BUILDING<OFFICE THEN
WRITE ("WATCH OUT FOR THE ",
"OIL SPOTS");
IF BUILDING=OFFICE THEN
  WRITE ("TURN OFF THE LIGHTS",
"AT 5 PM";
IF BUILDING <> HOUSE THEN
WRITE ("DO NOT SLEEP HERE")
END.
```

The expression of an enumerated type may be specified by:

1. a constant (one of the names used in declaration),
2. a variable of the type that has already been given a value, or
3. a function that produces a value of the enumerated type.

Note that like expression of type char, these three may not be cabined into formulas.

The TYPE declaration gives a name to a built-in or a simple type you define. If a program has both a TYPE and VAR declaration, the VAR declaration comes last.

After the word TYPE, the program declares each new name for a type in the same way: the name, an equals sign, the specifications of a simple type (integer, Boolean, char, a subrange of enumerated type or a previously declared name of any preceding), and finally a semicolon.

```
PROGRAM NEWTYPE (INPUT OUTPUT);
TYPE SAMEOLD=INTEGER; C=CHAR;
YETAGAIN=SAMEOLD;
DAYS=(SUNDAY, MONDAY, TUESDAY,
WEDNESDAY, THURSDAY, FRIDAY, SATURDAY);
WEEKDAYS= MONDAY..FRIDAY;
ZEROTHRU16= 0..16;
LETTERS="A".."Z";
VAR PAYDAY: WEEKDAYS;
  GAMESWON, GAMESLOST : ZEROTHRU16;
TEMP: SAMEOLD; CH;C;
BEGIN
   PAYDAY:= FRIDAY;
   CH: = "B";
 END.
```

4.4 Array of Sunshine

The variables described so far have been simple variables. Integer, char and Boolean variables, variables of enumerated types and variables of subrange types are called simple variables because they have no internal structure.

Arrays in Partial Pascal are variables that have an internal structure. They are composed of several simple variables, all of the same type. The individual simple variables are called the elements of the array. In Partial Pascal only one element of the array may be referred to at a time (Full Pascal has a few operations that work on an entire array at once). Each element of an array has a unique "index" that serves to distinguish that element from the other elements of the array. The indices of an array's element are consecutive values of a simple type, often consecutive integers.

An array element is denoted by the name of the array followed by the symbols `(.` and `.)` enclosing an expression that gives the value of the index. `MYARRAY(.37.)`, `VECTOR (.X+Y.)` and `TABLE (.⋯X⋯.)` are array elements of the arrays MYARRAY, VECTOR and TABLE. TABLE's index is of type char. To declare arrays, follow a list of one or more variable names in the `VAR` declaration by a colon, the word ARRAY, the type of the index enclosed in `(. .)` symbols, the word `OF` and the type of the elements.

```
PROGRAM ARRAYEXAM (INPUT, OUTPUT);
VAR VECTOR:ARRAY(.0..3.)
OF BOOLEAN;
NORMAL: ARRAY(.CHAR.) OF
INTEGER;
```

```
INTIND; INTEGER; INCHAR; CHAR;
BEGIN
INTIND:=0.
WHILE INTIND<=3 DO BEGIN
VECTOR(.INTIND.):=INTIND<3
END ;
INTIND:=0;
WHILE INTIND<=255 DO BEGIN
NORMAL(.CHR(INTIND).):=
(CHR (INTIND)>=… …) AND
(CHR (INTIND) =…Z…);
TABLE(.CHR(INTIND).):=0;
INTIND:=INTIND+1
END;
READ (INCHAR);
WHILE INCHAR <>"." DO BEGIN
TABLE(.INCHAR.) :=
TABLE(.INCHAR.) +1
READ(INCHAR)
END;
INTIND:=0;
WHILE INTIND<=255 DO BEGIN
IF (TABLE(.CHR(INTIND).)<>0)
AND NORMAL(.CHR(INTIND).)
THEN
WRITELN(CHR(INTIND),
" APPEARED ",
TABLE(.CHR(INTIND).),
" TIMES";
INTIND := INTIND +1
END
END.
```

VECTOR is an array with four Boolean elements VECTOR(.0.) , VECTOR(.1.), VECTOR(.2.) and VECTOR(.3.). NORMAL is an array of 250 Boolean elements, one for each of the characters Partial Pascal recognizes, including 128 that can't be displayed. TABLE is an array of 256 integers. The first WHILE statement assigns false to VECTOR(.0.) and VECTOR(.2.) and true to the other two elements.

The second WHILE statement initialize all the elements of the array named TABLE to zero, and assigns true to the elements of NORMAL that corresponds the 54 displayable non-inverse-video characters and the 10 graphics characters in the range " "…"0" and false to the 192 other elements of NORMAL.

The third WHILE statement reads characters until it comes to a period, using one element of TABLE for each possible character to keep track of how many times that character has appeared in the input.

The fourth WHILE statement writes a report of how many times each of 64 characters appeared in the input, if at all, before the first period.

A few restrictions apply to arrays in Partial Pascal that do not apply to full Pascal. The reserved word ARRAY may be used only in VAR declaration. Full Pascal also allows it in a TYPE declaration. Arrays in Partial Pascal have only one dimension. The elements of an array in Partial Pascal must be of a simple type.

The lowest index for an array may not be less than zero. If the lowest index is greater than zero, Partial Pascal treats it as zero during execution, both when allocating memory for the array and when checking its indices. The higher array index allowed is 4095.

This program is legal in full Pascal, but has two errors in Partial Pascal.

```
PROGRAM WONTWORK (INPUT,OUTPUT);
VAR TOOLOW: ARRAY (.-1..1.) OF
CHAR (* LOWER BOUND IS LESS
THAN ZERO; ILLEGAL IN PARTIAL PASCAL*);
TWODIM:ARRAY(.1..3,1..3.) OF
INTEGER (* TWO DIMENSIONS NOT
ALLOWED IN PARTIAL PASCAL*) ;
HILO:ARRAY (.1..100.) OF CHAR;
( * ALLOWED IN PARTIAL PASCAL;
TREATED AS ARRAY (.0..100.) OF
CHAR*)
```

PARTIAL PASCAL checks during execution that every index actually used with an array is greater than or equal to zero and less than or equal to the highest index declared for the array.

4.5 Constants

Partial Pascal recognizes several types of built-in constants: decimal numbers, characters in quotes, true and false. You can define your own named constants in an enumerated type. As a convenience, Pascal allows you to declare yet more names for constants in a CONST declaration. If a program has both a CONST declaration and either a TYPE or VAR declaration, the CONST declaration comes first.

After the word CONST, the program declares each new name for a constant in the same way: the name, an equal sign, the constant value and a semicolon. The constant value may be any built-in constant, or any constant name declared previously in a CONST or in an enumerated type.

```
PROGRAM CONSTANTINOPLE (INPUT, OUTPUT);
CONST CHECKING=FALSE;
MAXINDEX=72; LASTLETTER= "Z";
LISTING=CHECKING;
VAR DATA, CHECKS:
ARRAY(.0..MAXINDEX.) OF INTEGER;
COUNTER:INTEGER;
BEGIN
WRITELN( "THE ALPHABET GOES" ,
"FROM A TO ", LASTLETTER, ".");
COUNTER:=0;
WHILE COUNTER<=MAXINDEX DO
BEGIN
DATA(.COUNTER.):=0;
IF LISTING THEN
WRITELN ("DATA (. " ,COUNTER:1,
":=0");
COUNTER:=COUNTER+1
END;
IF CHECKING THEN
WHILE COUNTER>0 DO BEGIN
COUNTER:=COUNTER-1;
CHECKS(.COUNTER.):=1
END
END.
```

The CONST declaration makes CHECKING a synonym for false and LISTING a synonym for CHECKING, hence also false. Neither the writeln nor the second WHILE statement will ever be executed. The entire output written by this program is

```
THE ALPHABET GOES FROM A TO Z.
```

LASTLETTER, since it is declared as a char constant, is written as a single character by writeln. MAXINDEX makes changing the size of the DATA and CHECKS arrays easy. Both their declaration and values used as their indices in the statement are automatically changed when the value for MAXINDEX is changed.

4.6 I Repeat

The REPEAT statement forms a loop like the WHILE loop. The difference is that the statements in a REPEAT loop are guaranteed to be executed at least once.

The REPEAT statement begins with the word REPEAT followed by one or more statements separated by semicolons, the word UNTIL and a Boolean expression. After each execution of the statement, the Boolean expression is evaluated and if it is false the statements are executed again.

```
PROGRAM INITONLY (INPUT OUTPUT) ;
TYPE EINSTEINIAN=0..3;
VAR DIMENSION: EINSTEINIAN;
BEGIN
REPEAT
WRITE (" PLEASE ENTER A NUMBER",
" FROM 0 TO 3: ");
READLN(DIMENSION)
UNTIL (DIMENSION>=0) AND
(DIMENSION<=3)
END.
```

The two statements inside the REPEAT statement are executed at least once and may be executed many more times, depending on the input supplied to the read statement. An execution of this program could look like:

```
PLEASE ENTER A NUMBER FROM 0 TO 3: 5
PLEASE ENTER A NUMBER FROM 0 TO 3: -32768
PLEASE ENTER A NUMBER FROM 0 TO 3: 1984
PLEASE ENTER A NUMBER FROM 0 TO 3: 2
```

On the first three executions of the REPEAT loop the value entered for DIMENSION makes the Boolean expression after UNTIL false, and the statements are re-executed. The fourth time the expression is >=0 AND <=3, which is true and the execution of the REPEAT statement ceases.

4.7 For To Do

The FOR statement loops a predetermined number of times. Each loop is executed with a new value of a control variable. The FOR statement starts with the word FOR followed by the name of the control variable, an assignment symbol, an expression giving the initial value of the control variable, the word TO or the word DOWNTO, an expression giving the final value of the control variable, the word DO and a

statement or a group of statements marked by BEGIN and END. The control variable must be of a simple type. Full Pascal does not allow parameters to be used as the control variable, requires that the variable used be declared in the same program or subprogram that contains the FOR statement and does not allow the control variable to be modified by an assignment or read statement during the time FOR statement is executed. Partial Pascal does not check for these things, but to make your program compatible with full Pascal, you should.

If the final value is less than the initial value, greater than the DOWNTO value, the statement or group of statements is not executed at all, otherwise the control variable is assigned the initial value for the first execution of the statement(s) and on each succeeding execution its value is increased (or decreased if DOWNTO is used) by one. The last execution of the statement(s) is the one during which the control variable has the final value.

The initial and final values are calculated once, before the statements are executed. The control variable should not appear in the expressions for the initial and final value (Partial Pascal doesn't actually check for this), your program should not depend on the control variable having any particular value after the FOR statement is executed, since that value may be different for different Pascal compilers.

```
PROGRAM FORTODO (INPUT, OUTPUT);
TYPE NEWTONIAN=0..2;
VAR INIT:NEWTONIAN;
VECTOR:ARRAY(.NEWTONIAN.) OF INTEGER;
ALPH:CHAR;
BEGIN
FOR INIT:=0 TO 2 DO
VECTOR(.INIT.):=INIT:
FOR INIT:=2 DOWNTO 0 DO
IF VECTOR(.INIT.)<>INIT THEN
WRITE("UH OH", INIT);
FOR ALPH:="A" TO "Z" DO
WRITE (ALPH)
END.
```

The first FOR statement initializes the three elements of VECTOR. The second FOR statement checks that initialization. The third FOR statement is the only one that produces any output:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

4.8 Just in Case

The CASE statement chooses one statement for group of statements marked by BEGIN and END for execution from several candidates, based on the value of an expression. It starts with the word CASE followed by an expression, the word OF, one or more "labelled" statements (or groups) separated by semicolons, an optional OTHERWISE with a statement (or group) and finally the word END. The "labelled" statements each have one or more constants and a colon preceding them. The constants should all have different values (Partial Pascal does not check for this but a full Pascal compiler does).

If the value of the expression is the same as the value of one of the constants, the statement(s) that follow that constant are executed and the rest of the statements are not executed. If the value of the expression does not match any constant, the statement following OTHERWISE is executed. If the value of the expression does not match any of the constants and the word OTHERWISE is not used, none of the statements are executed.

The word OTHERWISE is a Partial Pascal extension to full Pascal. Several Pascal compilers recognize it, but it is not part of standard full Pascal.

Full Pascal allows an optional semicolon before the END that ends the CASE statement. Partial Pascal does not allow a semicolon there or before the word OTHERWISE.

```
PROGRAM TRAVEL (INPUT, OUTPUT);
CONST FOURSQUARE=16;
TYPE MEANS=(TRAIN,PLANE,CAR, BOAT, BIKE);
VAR NUM:INTEGER; WAY:MEANS;
BEGIN
(* PUT SOMETHING HERE TO ASSIGN VALUES TO NUM
AND WAY*)
WRITE(NUM, " ");
CASE NUM-2 OF 0,1,4,FOURSQUARE, 9: BEGIN
WRITE ("IS 2 MORE THAN A", "PERFECT SQUARE");
CASE NUM OF 2, 3,11: WRITE ( " AND A PRIME", "
NUMBER")
(* NO SEMICOLON HERE*)
END (* OF INTERNAL CASE *)
END (* OF STATEMENT GROUP*);
5,12: WRITE ("MOD 7=0"):
17, 15, 11, 3: WRITE ("IS A PRIME")
```

```
(* NO SEMICOLON HERE*)
OTHERWISE
WRITE ("IS NOT AN INTERESTING", " NUMBER")
END;
WRITELN;
CASE WAY OF
BIKE: WRITE ("NO FUEL NEEDED.");
TRAIN: WRITE ("NO STEERING " , " REQUIRED.");
PLANE, BOAT: WRITE ("TRAVELS ", "THRU A
FLUID.");
CAR: WRITE ("USES GASOLINE.")
END
END.
```

If way and num are given the values PLANE and 15 , this program writes:

```
15 IS NOT AN INTERESTING NUMBER
TRAVELS THRU A FLUID.
```

When way and num have the values BIKE and 11, the output is:

```
11 IS 2 MORE THAN A PERFECT SQUARE AND A PRIME
NUMBER.
NO FLUID NEEDED.
```

4.9 Subprograms

Pascal has two kinds of subprograms. A procedure is a subprogram that can be invoked as a statement. A function is a subprogram that produces a value. A subprogram is a program within a program. After its heading a subprogram looks just like a full program. Procedures and functions are declared after any `CONST`, `TYPE` and `VAR` declarations. A subprogram must be declared before it can be used.

Procedures and functions differ in their headings. The procedure heading is the word `PROCEDURE` followed by the name that will be used to invoke it, optionally followed by a list of parameters. Here is one procedure with no perimeters

```
PROGRAM USESPROC (INPUT, OUTPUT);
PROCEDURE THISISMYNAME;
BEGIN
WRITE(" LUDWIG VAN BEETHOVEN")
END;
BEGIN
WRITELN ("THIS IS FIRST");
THISISMYNAME;
```

```
WRITELN ("THIS IS LAST")
END.
```

A semicolon separates the procedure's heading from its declarations. Another semicolon after the procedure separates it from the PROCEDURE, FUNCTION or BEGIN that follows. A procedure's name may be used as a statement to cause the execution of the procedure.

The above program begins execution at the second BEGIN because that is the PROGRAM's BEGIN. Execution of the program does not start at the first BEGIN in the program but rather at the BEGIN that ends the declarations. The first statement executed is the first writeln. The next statement, THISISMYNAME, starts the execution of the procedure of that name at the BEGIN that ends its declarations. It executes the write statement, its only statement, and ends. When it ends, execution resumes with the next statement in the main program, the second writeln. The output from this program is:

```
THIS IS FIRST
LUDWIG VAN BEETHOVEN THIS IS LAST
```

4.10 With Parameters

A procedure with two integer parameters and one char parameter:

```
PROGRAM PARAMS3 (INPUT, OUTPUT);
VAR BADGUESSES, THIS: INTEGER;
PROCEDURE CHECKGUESS (GUESS, CORRECT: INTEGER;
CHANCE: CHAR);
VAR WRONG: INTEGER;
BEGIN
WRONG:=0;
IF ORD(CHANCE)<>CORRECT THEN
WRONG:=WRONG+ 1;
IF CHANCE<>CHR(CORRECT) THEN
WRONG:=WRONG+1;
IF WRONG=2 THEN
WRITE("INVOKER MADE ERROR");
IF WRONG=1 THEN
WRITE (" HELP PARTIAL PASCAL",
"IS INCONSISTENT");
IF (WRONG=0) AND
(GUESS<>ORD (CHANCE)) THEN
BADGUESS:= BADGUESS+1
```

```
END;
BEGIN
BADGUESSES:=0;
WRITE (" WHAT IS THE SINCLAIR" ,
" CODE FOR ""A""? ");
READ(THIS);
CHECKGUESS(THIS, ORD("A"), "A");
WRITE ("WHAT IS THE SINCLAIR ",
"CODE FOR "","""? ");
READ(THIS);
CHECKGUESS(THIS, ORD(","), " ");
WRITELN ("CORRECT ANSWERS",
2-BADGUESS)
END.
```

The heading of CHECKGUESSES declares three parameters. GUESS and CORRECT are both integers and CHANCES is a char. The parameters are declared inside parenthesis. Their format is the same as in a `VAR` declaration except that the types may be specified only by name. Parameters may be any of type built in to Partial Pascal (integer, char, Boolean, or text, which is described later) or any type whose name has been declared in a `TYPE` declaration. The parameters may be used in the same ways as variables declared in the subprogram's own `VAR` declaration. The only difference is that variables declared in `VAR` declaration have no particular value when the subprogram begins execution, but the parameters are initialized to values provided in the statement that invokes the subprogram.

The above program conducts a quiz and uses the procedure CHECKGUESS to check the answers. First the program uses `writeln` to pose a question and `read` to get an answer. Then the program causes CHECKGUESS to be executed by using its name as a statement. Since CHECKGUESS has parameters, the name must be followed by a parenthesis list of expressions separated by commas, one expression for each parameter of CHECKGUESS. The values of the expressions become the initial values of CHECKGUESS's parameters when it begins execution. The statement is called a procedure invocation and expressions are called the arguments of the invocation.

CHECKGUESS's purpose is to check the answer provided by whoever is taking the quiz but first it does a checking of its own. CHECKGUESS starts by zeroing its own variable, WRONG, whose value becomes unpredictable every time CHECKGUESS begins execution. Then CHECKGUESS tests the invoker and the authors of PARTIAL PASCAL for consistency. Finally, CHECKGUESS checks the answer. An incorrect guess is only recorded in the BADGUESS variable if the invoker and Partial Pascal were both found to be consistent. After CHECKGUESS ends, the main program asks

another quiz question, gets another answer, invokes CHECKGUESS a second time and finally writes a message to the quiz taker with his score.

Note that BADGUESSES is not one of CHECKGUESS's variables. Subprograms may use their own variables, the main program's variables and the variables of any other subprogram in which they are contained. Similarly, the CONST, TYPE, PROCEDURE and FUNCTION declarations of the program and of any enclosing subprogram may also be used. In the case of procedures and functions, the heading of the invoked procedure or function must have already appeared in the source program.

```
PROGRAM NESTING(INPUT OUTPUT):
CONST TRACING=TRUE;
PROCEDURE P1
BEGIN
IF TRACING, THEN
WRITE ("P1)"
(* P1 COULD BE INVOKED HERE *)
END:
PROCEDURE P2:
PROCEDURE P3:
BEGIN
IF TRACING, THEN
WRITE ("P3");
(*  P3 COULD BE INVOKED HERE *)
(* P2 COULD BE INVOKED HERE *)
P1
END;
BEGIN
IF TRACING, THEN
WRITE ("P2");
P3;
(* P3 COULD BE INVOKED HERE *)
P1
END;
PROCEDURE P4:
BEGIN
IF TRACING THEM
WRITE ("P4");
(* P3 CANNOT BE INVOKED HERE *)
P2;
P1
END;
```

```
BEGIN
WRITE ("START");
P1;
P2;
(* P3 CANNOT BE INVOKED HERE *)
P4;
WRITE ("END")
END.
```

When this program executes. It produces the following output:

```
STARTP1P2P3P1P1P4P2P3P1P1P1END
```

4.11 Functions

Functions are the other kind of subprogram. Their headings use the word FUNCTION in place of the word PROCEDURE and they have one item more than procedure heading. After the parameter list (if any) comes a colon and the name of the type of value the function produces. A declared function is invoked in the same way as a built-in function, as part of an expression.

```
PROGRAM MORE (INPUT OUTPUT);
VAR NUMBER:INTEGER;
FUNCTION MAX (ONE, THEOTHER:
INTEGER):INTEGER;
BEGIN
MAX:= ONE;
IF THEOTHER>ONE THEN
MAX:= THEOTHER
END:
FUNCTION FIRSTLARGEST
(ONE, TWO, THREE: INTEGER):
BOOLEAN;
BEGIN
FIRSTLARGEST:= ONE >= MAX (TWO, THREE)
END;
FUNCTION FACTORIAL (N:INTEGER):
INTEGER;
BEGIN
IF MAX(N, 0)=MAX(-N, 0) THEN
FACTORIAL:=1
ELSE
IF FIRSTLARGEST(-1,N,N) THEN
BEGIN
```

```
WRITE ("FACTORIAL OF ",
 "NEGATIVE NUMBER CANNOT ",
 "BE CALCULATED");
HALT (1)
END ELSE
IF N>= 8 THEN BEGIN
WRITE ("FACTORIAL OVERFLOW");
HALT (1)
END ELSE
FACTORIAL:= N*FACTORIAL(N-1)
END;
BEGIN
WRITE
("PLEASE ENTER A NUMBER");
READ (NUMBER):
WRITELN
(NUMBER:1 " FACTORIAL IS ",
FACTORIAL(N))
END.
```

MAX is a function that produces the larger of the value of its two arguments. The value a function produces is the value it assigns to its name. A function must assign a value to its name at least once before ending (a full Pascal compiler would check this, but Partial Pascal doesn't).

FIRSTLARGEST produces the value true when the first of its three parameters is at least as large as each of the other two.

FACTORIAL is the example invariably used to illustrate a recursive function, one that invokes itself. This version of FACTORIAL puts in two checks not usually seen in the textbooks. Factorials of negative numbers are not calculable numbers, and 9 factorial, 1*2*3*4*5*6*7*8 = 40,320, is not representable using Partial Pascal's integers.

4.12 In the Files

A program must do several things to write output to a device. It must declare a file name, it must execute the `rewrite` statement to prepare the file for output, and it must execute `write` or `writeln` to actually perform the output.

Similarly, to read input data from a device, a program must declare a file name in its reading, prepare the file for input using the `reset` statement and get the data with `read` and `readln` statements.

None of the programs so far has used `reset` or `rewrite`, so how is it that they read or written anything?

Every program reading so far has had input, output as the file names. Those names have special reading in Pascal. They are the default files. `Read` and `readln` may operate on any file but if they do not specify a file name, the file they use is input. `Write` and `writeln` use output if no file name is specified. Finally, if a file named input is declared, Partial Pascal issues reset for it before the program begins execution and if a file named output is declared, Partial Pascal issues rewrite for it before the program begins.

So, if you want to use more than one input file or more than one output file, you have a little extra work to do in our program. The rewrite statement takes one argument which must be the name of the file. It prepares the file for output. The reset statement takes one argument, which must be the name of a file. It prepares the file for input, unless the file is a printer file, in which case it halts it executes one of the programs.

```
PROGRAM SKIP1(SELBST,ANDERER);
ONECHARACTER: CHAR;
BEGIN
RESET(SELBST);
REWRITE (ANDERER);
READ(SELBST,ONECHARACTER);
WRITE (ANDERER,ONECHARACTER);
READ (SELBST,ONECHARACTER,
ONECHARACTER):
WRITELN(ANDERER,ONECHARACTER)
END.
```

The first argument of a read, readln, write or writeln may be a file name used for the reminder of the statement. The preceding program is equivalent to the following program.

```
PROGRAM (1AND2: INPUT, OUTPUT);
VAR C: CHAR;
BEGIN
READ (C);
WRITE (C);
READ (C);
READ (C);
WRITELN (C)
END.
```

The first program uses its own file names while the second uses the default. Both programs copy the first and third characters of their first file to their second file.

4.13 End of the Line

Pascal provides two functions that makes input easier to use. The `EOF` function takes one argument, which must be the name of a file (full Pascal allows `EOF` to be use with no arguments, but Partial Pascal does not), and produces a value of True or False. `EOF` stands for End of File. `EOF` is true for a file from which all the data has been read. If `read` or `readln` is invoked for a file for which `EOF` is true, execution of the program halts. `EOF` is also true (and attempts to read are fatal) for a file for which reset has been issued and for a file for which reset has been issued but for which `rewrite` has been issued since the most recent reset. When reading data from the keyboard, press ENTER while holding down SHIFT to indicate End of File.

```
PROGRAM COPY (INPUT,OUTPUT);
VAR FUZZY: CHAR;
BEGIN
WHILE NOT EOF (INPUT) DO BEGIN
READ (FUZZY);
WRITE (FUZZY)
END
END.
```

This program copies all of the characters of its first file to its second file. It can't be making a good copy, since, though, since it never issues writeln. It needs help from another built-in function, `EOLN`. `EOLN` stand for end of line. It takes one argument, which must be the name of a file (full Pascal allows `EOLN` to be used without an argument, but Partial Pascal does not). Like `read` and `readln`, `EOLN` halts program execution if it is issued for a file for which `EOF` is true. `EOLN` is true if and only if the next character to be read from the file (and there must be a next character, since `EOF` is false) is an end-of-line. When reading from the keyboard, the ENTER key is an end-of-line. When reading from tape, the end-of-line is a special character that the `writeln` statement writes to the tape. The end-of-line cannot be written to tape in any other way.

In either case, the end-of-line, when need in as a character, is indistinguishable from a space. An end-of-line can only be distinguished from a space before it is read.

```
PROGRAM ELINE (INPUT, OUTPUT);
VAR CH: CHAR;
BEGIN
IF EOLN (INPUT) THEN BEGIN
WRITELN ("FIRST CHARACTER IS", "AN");
WRITELN ("END-OF-LINE");
READ (CH);
IF CH <>" " THEN
WRITELN ("I NEVER WRITE THIS",  MESSAGE")
END ELSE
READ(CH);
READ(CH);
IF CH = " " THEN
WRITE ("THE SECOND CHARACTER ",
"IS EITHER A SPACE OR ",
"END-OF-LINE, BUT NOW IT IS",
"TOO LATE TO TELL WHICH")
END.
```

A program that correctly copies one file to another follows.

```
PROGRAM PERFECTCOPY (INFILE, OUTFILE);
VAR CH: CHAR;
BEGIN
RESET (INFILE);
REWRITE (OUTFILE);
WHILE NOT EOF (INFILE) DO BEGIN
IF EOLN (INFILE) THEN BEGIN
READ (INFILE, CH); (* CH = " " *)
WRITELN (OUTFILE)
END ELSE BEGIN
READ (INFILE, CH)
WRITE (OUTFILE,CH)
END
END
END.
```

4.14 Passing Files

Subprograms may have files as parameters. Each file parameter must be declared in the subprogram's header as type text, one of Pascal's built-in types.

Partial Pascal requires that all of a subprogram's file parameters to be among the subprogram's first 10 parameters.

```
PROGRAM DECTHEXT(INPUT, DECOUT, HEXOUT);
VAR CONVERT: INTEGER;
PROCEDURE INITIAL (NAME: TEXT);
BEGIN
REWRITE (NAME);
WRITELN (NAME, '-BEGINNING-')
END;
PROCEDURE WRITEHEX (WHERE: TEXT; WHAT: INTEGER);
BEGIN
IF WHAT > = 16 THEN
WRITEHEX(WHERE,WHAT DIV 16);
WRITE (WHERE, CHR (WHAT MOD 16 + ORD('0')))
END;
BEGIN
INITIAL (DECOUT);
INITIAL (HEXOUT);
REPEAT
READ (CONVERT):
WRITE (DECOUT, CONVERT);
IF CONVERT>= 0 THEN
WRITEHEX(HEXOUT, CONVERT)
UNTIL CONVERT<0
END.
```

Procedures INITIAL and WRITEHEX each have a text file as a parameter. A `reset` or `rewrite` issued by a subprogram is still valid after the subprogram ends.

WRITEHEX is a recursive program that writes out a non-negative number in a decimal notation. (Hexadecimal notation is base 16; there are 16 hexadecimals digits, 0 thru 9 and A thru F).

This program first uses procedure INITIAL to issue `rewrite` and write a message to two of its files. It then enters a loop that reads numbers and writes them in decimal to its second file and in hexadecimal to its third file.

4.15 Devices

As we have seen, Partial Pascal asks you to select a device for each of a program's files just before it begins execution. There are four different devices recognized by Partial Pascal, represented by four different letters when selecting devices.

`S` selects the standard devices, the keyboard for input and the TV screen for output.

`T` selects the tape recorder for input or output.

P selects the printer for output only.

N selects nothing for input or output.

When a program is reading input from the keyboard, Partial Pascal displays the typed input on the TV screen. A slowly flashing cursor shows where the next character will appear. 0 pressed with SHIFT deletes the previously typed character. Neither the SHIFT 0 nor the deleted character is provided to the program executing. None of the typed data is provided to the program until ENTER is pressed. ENTER pressed while holding SHIFT is the end of the input file. (Actually, if you type 33 characters before pressing ENTER, Partial Pascal will allow the program to read the first character and it can no longer be deleted). Undisplayable characters are converted to something displayable when written to the TV devices, with the exception of nullkey (chr(64)), which has no effect, ENTER (chr(118)), which acts like a writeln with no data, and SHIFT 0 (chr(119)), which erases the preceding character written.

Partial Pascal temporarily stops executing when the A key is held down while holding down the SHIFT key. Partial PASCAL does not recognize the BREAK key (which allows inkey to accept it as the SPACE key, something which BASIC's INKEY$ cannot do), but SHIFT A does allow you to stop a program to see what is on the screen. This use of SHIFT A makes it difficult to enter SHIFT A as a character. If you need to use SHIFT A to enter an inverse video A or the graphic on the A key or to control the editor, hold down SHIFT and press A several times. It will be recognized eventually.

The 9 key pressed with SHIFT, may be used to change from normal mode to graphics or back any time the slowly flashing cursor shows.

Output written to tape is read back in verbatim character by character. Integers are converted to decimal digits, just as they are when written to the TV. If you want to store binary data on tape, you will be glad to know that any of the 256 values chr(0) thru chr(255) may be written to tape and not confused with the end-of-line, which is a 257th value.

Each data set written to tape gets its own name. When a data set is read back in thru a file, EOF becomes true for the file if the data set was written to tape because the program doing the writing ended or the program doing the writing issued rewrite or reset for the file, indicating there was no more data for the data set. EOF does not become true at the end of the data set that was written because the output became full.

Program execution halts if reset is executed for a printer file.

After executing reset for a nothing file, `EOF` is still true. `Read`, `readln` and `EOLN` always halt the program when executed for a nothing file. After a `rewrite`, `write` and `writeln` may be executed freely for a nothing file, but the data they write just disappears.

Chapter 5. Advanced Partial Pascal

5.1 Abs, Pred, Succ, Sqr, Lsl

Partial Pascal has five built-in arithmetic functions. The built-in function `abs` produces the absolute value of its integer argument.

The built-in function `pred` produces the value are less than its argument, which may be of any simple type.

The built-in function `succ` produces the value one more than its argument, which may be of any simple type.

The built-in function `sqr` produces the square of its integer argument. Full Pascal's square root function, `sqrt` is not available in Partial Pascal.

The built-in function `lsl` performs a logical shift of its first argument. The second argument specifies how far to shift. The binary representation of the first argument is shifted in the left by the number of bits specified as the second argument. If the second argument is negative, the SHIFT is to the right.

5.2 No more

Partial Pascal's `maxint` built-in integer constant has the value +32767, the largest number that can be represented in Partial Pascal.

5.3 Clear Screen

Partial Pascal built-in procedure page takes one argument, the name of a file for which `rewrite` has been issued. `Page` clears the TV display, if the file is standard service file. Otherwise, `page` does nothing.

5.4 Write Here

Partial Pascal's `at` built-in procedure determines the next screen location to be used for screen output or keyboard input. The two arguments are integer expressions that gives the line (0..22) and column (0..31). Since, the 23rd line does not participate in automatic scrolling, only data writen after an `at` with its first argument set to 22 can appear on the 23rd line. `At` affects all the files that write to the TV screen.

5.5 Inkey and Nullkey

Partial Pascal's `inkey` built-in function returns the character value of the current key being pressed, if any. If no key is being pressed, `inkey` has the value `nullkey`, a built-in char constant. Unlike the BASIC function `INKEY$` and keyboard input using `read` or `readln` keys held down continuously do not automatically repeat for `inkey`. Any character having the value `nullkey` is ignored when written to the TV screen using `write` or `writeln`.

5.6 Fast, Slow, Copy, Pause

The built-in procedure `fast` and `slow` set the computer for compute only (fast mode) or compute-and-display (slow mode). `Fast` and `slow` take no arguments. All Partial Pascal programs begin execution in slow mode.

The built-in procedure `copy` prints the contents of the TV screen (all 24 lines) of printer is connected. `Copy` does not use a file to write to the printer. `Copy` takes no arguments.

The built-in procedure `pause` has one argument, an integer expression a time interval in units of 60th of a second (50ths in the United Kingdom). The computer stops processing until a key is pressed (other than just SHIFT) or until the specified time interval elapses, whichever comes first.

5.7 Graphics

Partial Pascal's built-in procedure `plot` may be used to control the display of graphics. `Plot` takes three arguments, a Boolean expression that determines if a graphics pixel is to be turned on (true) or off (false), and two integer expressions that give the horizontal (0..63) and vertical (0..43) coordinates of the pixel. If the display at the pixel is a non-graphic character or a graphic character that contains gray, the character is set to a blank space before `plot` operates.

Partial Pascal's built-in function `point` takes two arguments, the horizontal and vertical coordinates of a graphic pixel, and produces true if the pixel is on and false if the pixel is off or is in a non-graphics or gray graphics character.

5.8 Machine language programs

Partial Pascal's built-in functions `usr` causes a machine language program to be executed. `Usr` takes 5 integer arguments. The first is the address of the machine language program. When the machine language program is entered, the return

address to Partial Pascal is on the stack and the AF, BC, DE, and HL register pairs have the values specified as the second thru fifth arguments of usr. The machine language program may use up to 50 bytes of space on the stack. It may modify the AF, BC, DE, HL, BC', DE' and HL' registers without restoring them. Registers I, R, IX and IY should not be modified. The instruction EX AF, AF' should not be executed. The value produced by usr is the contents of the BC register pair when the machine language program returns to Partial Pascal.

5.9 Memory Manipulation

Three built-in arrays allow the direct examination and manipulation of the computer's memory. `Mem` operates one byte at time like BASIC's `PEEK` and `POKE`. `Mem` treats the computer's memory as a large `ARRAY` of char. The index of `mem` gives the address of the memory byte to be used. `Mem` may be used in an expression in the same way as a declared array to examine the contents of memory. Memory may be modified by using `mem` in a `read` or `readln` or by using `mem` on the left side of an assignment statement. `Mem`(.0.) thru `mem`(.8192.) are the ROM and `mem`(.16384.) thru `mem` (.32767.) are the 16k RAM.

```
PROGRAM PEEKANDPOKE (INPUT, OUTPUT)
VAR LOC, VALUE: INTEGER:
BEGIN
FOR LOC:= 0 TO 9 DO
WRITELN ("PEEK (",LOC:1,") = ", ORD(MEM
(.LOC.)));
REPEAT
WRITE ("LOCATION FOR",
"UPDATE? ");
READ (LOC);
WRITE ("NEW VALUE FOR ", LOC.1,
"?");
READ(VALUE);
MEM(.LOC.):= VALUE
UNTIL EOF (INPUT):
MEMW(.16400.)=MEM2(.16400.)
END.
```

This program first displays what would be `PEEK(0)` thru `PEEK(9)` in BASIC, then does `POKE LOC, VALUE` as long as the input file has any input left. (The user would press SHIFT ENTER after the last value.)

The built-in arrays MEMW and MEM2 access two consecutive bytes at a time. MEMW takes the two bytes in the Z80 microprocessor's customer order, the less significant byte at the lower address. MEM2 takes the bytes at the lower address the more significant byte.

The last statement in the above program exchanges the contents of the two bytes at 16400 and 16401.

The built-in procedure move is for wholesale movement of data in memory. Moves takes three arguments. The first is the lowest address in the block of memory to be covered, the second is the lowest address in a block of memory to be overwritten with data from first block. The third gives the number of bytes to be copied or the negative of the number of bytes to be copied. In the third operand is positive, the lowest address in a block is copied first. If the third operand negative, the lowest address in a block is copied last.

The built-in function ram produces the address of a work area for your program to use with mem, memw, mem2, and move. The argument of ram is the number of bytes of the space you require. If the number of bytes requested is not available, ram produces the value zero. Ram may be used as many times as necessary in a program producing the same address every time (except when it produces zero). it is used in a given execution of a program. The work area provided by ram is not modified by Partial Pascal until the program ends. The length of the work area is the value of the argument of the most recent use of ram that produced a non-zero address.

5.10 Hexadecimal

The Partial Pascal compiler recognizes hexadecimal constants indicated by a dollar sign. E.g., $400c is hexadecimal 400c. No space is permitted after the dollar sign.

5.11 Reserving space

If you wish to reserve some memory for your own machine language continuous or for other uses, POKE 16389 and optionally 16388 with a new value for RAMTOP, and execute BASIC's NEW before loading Partial Pascal.

RAMTOP must also be poked if you wish Partial Pascal to use more than 16k. If you have, for instance, a 16k RAM pack attached to a Timex Sinclair 1500, giving a total of 32k of RAM, then POKE 16389,192 and execute NEW from BASIC before loading Partial Pascal.

Partial Pascal does not modify or examine memory at or above the address gives in RAMTOP. 16389 must not be poked with the value less than 111.

5.12 Long Programs

For long programs, the compiler can be made in read some of a source program from tape. To do this, the devices must be selected as "STT" (for compiler listing on the TV screen) or "PTT" (for compiler listing to the printer). The compiler reads source from its second file when it encounters a $I in the program in the editor's memory ($I is not recognized if it is in a comment or in a quotation.)

The remainder of the source line that contains the $I is a comment. It may be used to indicate which tape date set is to be read.

```
PROGRAM LARGE (INPUT, OUTPUT);
VAR HUGE: INTEGER: BIG: BOOLEAN;
$INCLUDE SUBPROG
FUNCTION MYSTERY(A) INTEGER:
INTEGER;
$INCLUDE MYSTERY
BEGIN
SUBPROG (MYSTERY(1))
END.
```

Assume that the editor had previously been used to compose and save on tape the following procedure.

```
PROCEDURE SUBPROG (ONEARG:INTEGER)
BEGIN
WRITE ("THE ARGUMENT IS ",
 ONEARG)
END;
```

Using the data set name SUBPROG1.PAS and that the following function was saved using the data set name MYSTERY2.PAS.

```
BEGIN
MYSTERY:= A1*ABS (A1)
END;
```

The compiler, compiling LARGE from the editor's memory displays:

```
PROGRAM LARGE (INPUT, OUTPUT)
VAR HUGE : INTEGER; BIG: BOOLEAN.
$INCLUDE SUBPROG
```

And the 24th line asks:

```
I2 "     "
```

You type "SUBPROG1.PAS", play that previously saved data set on the recorder, and the compiler continuous compiling from tape displaying.

```
PROGRAM LARGE (INPUT, OUTPUT)
VAR HUGE: INTEGER: BIG: BOOLEAN
$INCLUDE SUBPROG
PROCEDURE SUBPROG/ONEARG
INTEGER
BEGIN
WRITE (←THE ARGUMENT IS←, ONEARG:1)
END
FUNCTION MYSTERY(A1:INTEGER):
INTEGER;
$INCLUDE MYSTERY
```

And the 24th line asks again

```
I2 "    "
```

Type in "MYSTERY.PAS", position the tape, press play, and the compiler finishes compiling.

```
PROGRAM LARGE (INPUT OUTPUT)
VAR HUGE: INTEGER: BIG: BOOLEAN
$INCLUDE SUBPROG
PROCEDURE SUBPROG (ONEARG: INTEGER)
BEGIN
WRITE ("THE ARGUMENT IS",
 ONEARG:1)
END;
(*27*)
FUNCTION MYSTERY(A1:INTEGER):
INTEGER;
$INCLUDE MYSTERY
BEGIN
MYSTERY:= A1*ABS(A1)
END:
(*35*)
BEGIN
SUBPROG (MYSTERY(1))
END.
(*43*)
```

5.13 Partial Pascal Built-ins

Constants: nullkey, maxint, false, true.

Types: text, integer, char, Boolean.

Vars: mem, memw, mem2.

Procedures: write, writeln, read, readln, rewrite, reset, atplot, halt, fast, slow, copy, pause.

Functions: eof, eoln, inkey, abs, pred, succ, sqr, chr, odd, ord, var, point.

Chapter 6. The Complete Editor

6.1 More Control

The Y, E, 1, 2, 0, R and A keys control the editor when used with SHIFT. They cannot be used when the cursor is on the 23rd line of the TV display or when the keyboard is in graphics mode (indicated by the inverse video G on the last line).

The Y key, used with SHIFT, makes room for an insertion into an already existing line of data. The data at the cursor and to its right is moved one space to the right, leaving a vacant space at the cursor. The move does not take place if it would shift data off the screen to the right.

The E key used with SHIFT, erases the line the cursor is on and moves everything below it up one line. Once the line is gone, the editor does not provide any way to get it back.

The 1 key, used with SHIFT, inserts a new line at the cursor location. All the data at or below the cursor is moved down one line.

The 2 key, used with SHIFT, makes a duplicate of the line the cursor is on. All the data below that line is moved down one line.

The 0 key, used with SHIFT, deletes the character the cursor is on. Characters to the right of the cursor move one space to the left.

The R key used with SHIFT, restores the line the cursor is on to what it was when the cursor moved to it. This is helpful if you've made some mistakes on the line and want to start over with it. Once the cursor has left a line, it can no longer be restored.

The T key, used with SHIFT, sends the entire contents of the editor's memory to the editor's second file. If devices were selected as "SNT" the second file is nothing and SHIFT T has no effect. To use SHIFT T to print data, specify "SPT" when starting the editor.

The A key, used with SHIFT, adds more data to that already in memory. The editor reads from its third file (tape) until end of file. Everything in the editor's memory before the SHIFT A is kept. If there is not enough memory to hold the additional data read in, only as many lines as will fit are read in and any remaining data in the data set is ignored. The line the cursor is on and all lines below it are moved down as far as necessary to accommodate the new data. Normally the data read in is data

previously saved by the editor, but it can be any tape data set written by any Partial Pascal program.

The D key, used with SHIFT, makes the editor issue the copy built in procedure to print the 24 lines of data currently displayed on the TV screen. If no printer is attached to the computer, SHIFT D does nothing.

6.2 Summary

These keys control the editor if used with SHIFT when not in graphics mode. Some require that the cursor be somewhere in the first 22 lines of the display of data being edited. The 23rd line shows the number of characters of the editor's memory remaining for used and the number of lines of data that are above those displayed on the TV.

1: 1 new line. Insert a new line consisting of all spaces. The line the cursor is on and all lines after it move down. Complements SHIFT E.

2: 2 of the same. Duplicate the line the cursor is on. All lines below it move down.

3: Display down. Move the displayed area 11 lines father from the top. Complements SHIFT 4.

4: Display up. Move the displayed area 11 lines closer to the top. Complements SHIFT 3.

5: Cursor left. Move the cursor one space to the left. If the cursor is already at the left margin, move it to the right margin one line up. Complements SHIFT 8.

6: Cursor down. Move the cursor one line down. If it is already on the 23rd line move it to the top line. Complements SHIFT 7.

7: Cursor up. Move the cursor one line up. If it is already on the top line. Move it to the 23rd line. Complements SHIFT 6.

8: Cursor right. Move the cursor one space to the right. If it is already at the right margin, move it to the left margin one line down. Complements SHIFT 5.

9: Graphics. If in normal mode enter graphics mode. If in graphics mode, enter normal mode. Note: SHIFT 9 controls graphics for any Partial Pascal program. SHIFT 9 cannot read be read in as data by read, `readln` or `inkey`.

Ø: Delete. Delete the character the cursor is on. Characters to the right of the cursor move left. Complements SHIFT Y.

Q: Quit editing. If any changes have been made since the last Save or Get (or since the editor started if there haven't been any Saves or Gets), the editor will ask if the changes should be lost.

E: Erase line. Erase the entire line the cursor is on. All lines below it move up. Complements SHIFT 1.

R: Restore. Restore the line the cursor is on to the way it looked when the cursor moved to it.

T: Type on the printer. Write everything in the editor's memory to the editor's second file. If you have a printer specify ''SPT'' when starting the editor. The compiler may be used to get a printed listing by specifying ''PNT'' when starting the compiler.

Y: Insert. Insert a space at the current location. Characters at and to the right of the cursor move right. Does nothing if a character other than a spare would be moved off the TV display. Complements SHIFT Ø.

A: Add more data. Read previously saved editor output back into memory. The line the cursor is on and all lower lines move down. Stops reading at the end of file or when memory is full.

S: Save whatever is being edited. To perform the save, the editor copies from its memory to its third file, the one for which you selected tape by typing T as the third character in SNT.

D: Copy. Use the copy built-in procedure to print all 24 lines of the TV display.

G: Get something to edit from tape. If any changes have been made since the last Save or Get (or since the editor started if there haven't been any Saves or Gets), the editor asks if the changes should be lost. To perform the get, the editor erases everything it has in memory and replaces it with what it reads from its third file. The third file is the one for which you selected tape by typing the T in ''SNT''.

Chapter 7. Diagnostic Messages

7.1 During Compilation

When the compiler cannot understand something in a program, it writes the word `ERROR` (a presumption on its part) and a number that describes the compiler's reason for not finishing the compilation. When this happens, look to the number in the following list of explanation, see how it applies to your program (the compiler's display of the source program continues for one or two symbols after it detects the error, but sometimes, as with an undeclared variable, this is long after the actual error), press any key to get the selection display, press `1` to select the editor and you can change your program, which is still in the editor's memory, immediately.

1. A name is used before it is declared. This is sometimes due to misspelling the name in the declaration, sometimes due to misspelling the name when it is used and sometimes due to omitting a declaration.
2. A name is declared twice at the same meeting level. Example:

```
VAR SOMENAME: CHAR;
PROCEDURE SOMENAME;
```

3. A procedure or function is used with more or fewer parameters than are in its declaration. Example:

```
K:=ABS(I,J)
```

4. Part of expression is invalid. Where the compiler expected to find a constant, variable or function call, it found something else. This can be caused by specifying a text file's name in an expression. Example:

```
WRITE (VAR)
```

5. The lower bound of index of an array is less than 0. This is legal in full Pascal, but not allowed in Partial Pascal. Example:

```
VAR TOOLOW: ARRAY (. -1..1.) OF
BOOLEAN;
```

6. A statement is not recognizable because the statement does not begin with IF. WHILE, REPEAT, CASE, FOR or a name. Example:

```
IF A<B THEN
B:=A;  (*ILLEGAL SEMICOLON*)
ELSE
B:=0
```

7. A statement is not recognizable because the statement begins with a name that is not the name of a variable, not the name of a procedure and not the name of a function. Example:

```
PROGRAM WRONG (INPUT, OUTPUT);
BEGIN
INTEGER:=1
```

8. A statement is not recognizable because it begins with the name of a function but is outside of the function's declaration. A value may be assigned to a function only by the function's own statements and the statements of any nested procedure or function.

9. The source program does not have a constant where it is required in one of the following situations:

    1. After an equal sign in a CONST declaration
    2. As the lower bound of a range
    3. As the upper bound of a range
    4. After the OF in a CASE statement
    5. After a semicolon in a CASE statement. Although full Pascal allows a semicolon before the END that ends a CASE statement, Partial Pascal allows a semicolon neither before the END nor before the (optional) OTHERWISE in a case statement. Example:

```
VAR A2: ARRAY (.4..SQR(4).) OF
INTEGER;
```

10. The upper bound of a range is lower than the lower bound. Example:

```
AR MIXED: ARRAY(.2..1.) OF
INTEGER;
```

a. A write or writeln does not specify a text file to write to and the default text file output is not declared in the program header or a read or readln does not

specify a text file to read from and the default text file input is not declared in the PROGRAM header. Example:

```
PROGRAM UHOH (INFILE, OUTFILE);
BEGIN
WRITE ("HELLO")
```

11. A closing quote does not appear on the same line as the corresponding opening quote. In Pascal, quotations must be contained on a single line. Example:

```
WRITE (OUTFILE, "THIS IS TOO LONG
TO FIT ON ONE LINE")
```

12. A write statement does not specify any data to write or a read statement does not specify any variable to be read. Example:

```
WRITE(OUTPUT)
```

13. A variable is declared to be of type text or of a type equivalent to text. This is legal in full Pascal, but not allowed in Partial Pascal. In Partial Pascal only parameters may be text files. Example:

```
TYPE MYFILES=TEXT;
VAR OOPS: MYFILES; OOPS2: TEXT;
```

14. A procedure or function is used with a parameter list that does not have a text file required by the procedure or function's declaration. Example:

```
VAR ABC: INTEGER;
BEGIN
REWRITE(ABC)
```

16: The name following the word FOR is not the name of a simple variable. Example:

```
VAR PAYROLL: ARRAY (.0..9.) OF
INTEGER;
BEGIN
FOR PAYROLL:=0 TO 9 DO
```

18: A parameter in a read or readln parameter list is not a variable. Example:

```
READ (-X)
```

19: The name of a type was required, but not found, in one of these situations:

1. Following a colon in a procedure or function's parameter list declaration.

2. Following a colon in a function's type declaration (after the function's parameter list)

Subranges and enumeration types are not allowed in procedures and function headers. Example:

```
PROCEDURE SENDDIGITS (X,Y:0..9);
```

19. The value returned by a function is declared to be of type text. Pascal allows functions to return simple types only.

```
FUNCTION CURRENT ( CH : CHAR) : TEXT;
```

20. A decimal constant is greater than 32767 or less than -32768. ALL decimal numbers in a Partial Pascal program must be in the range -32768 thru 32767. Example:

```
XYZ := 45000
```

21. An alphabetic character immediately follows a decimal number. Example:

```
X:= 100DIV Y
```

22. An opening quote mark is immediately followed by a closing quote. In Pascal, when quotes are used, at least one character must be between the quotes. Example:

```
NOCHAR:=''
```

23. The character following a dollar sign was not an "I" (FOR INCLUDE) and was not a hexadecimal digit ("0" .. "9", "A".."F"). Note that no space is allowed after the dollar sign. Example:

```
THIRTYTWO:=$ 20
```

24. The source program ended before the terminating period. A Pascal program mustend with the reserved word END followed by a period. This error is sometimes caused by omitting a $INCLUDE when compiling from tape.
25. A character constant is preceded by a + or – sign. Example:

```
CONST LAST= 'Z' ; FIRST=- LAST;
```

26. An invalid character is in the source program. Inverse video characters graphic characters unprintable characters, "?" and "/" may appear in a Partial Pascal program only if quoted or in a comment. Example:

```
WRITE ('HALF OF SIX IS' , 6/2)
```

27. A procedure was declared forward, but not redeclared within the immediately enclosing procedure, function or program. Example:

```
PROCEDURE ENCLOSE;
PROCEDURE DEFER: FORWARD;
BEGIN (# MARKS THE BEGINNING OF
THE STATEMENTS AND END OF
THE DECLARATIONS OF ENCLOSE #)
```

28. A procedure or function header was followed by FORWARD but the forward declaration is not allowed for one of the following reasons:

    1. The procedure or function was previously declared forward.
    2. The procedure or function was declared with parameters. Parameters are allowed for a procedure or function declared forward in full Pascal, but not in Partial Pascal.
    3. The procedure or function is a function. Functions may be declared forward in full Pascal but not in partial Pascal.

Example (reason 2):

```
PROCEDURE RESOLVE (J:INTEGER):
FORWARD;
```

29. The nesting level of a function or procedure is more than eight. Partial Pascal supports nesting of functions and procedures only to depth eight , counting the main program as depth one and the first procedure or function declared as depth two.

30. The upper bound of the index of an array is declared as more than 4095. Arrays in Partial Pascal may have no more than 4096 elements. Since the first index of an array in Partial Pascal is always zero, the largest index allowed in Partial Pascal is 4095. Example:

```
VAR TOOBIG:ARRAY(.0..5000.) OF
INTEGER;
```

31. A parameter in a procedure or function declaration is declared to be of type text but the parameter is not one of the first 10 parameters declared for that procedure or function. In Partial Pascal, procedures and functions are allowed no more than 10 text parameters, and all text parameters must declared among the first 10 parameters of a procedure or function. Example:

```
PROCEDURE OUTTEN( A,B,C,D, E,
F,G,H,I,J:INTEGER;OUT:TEXT);
```

32. The Pascal main program, or one of the functions or procedures, declares variables that take up a total of more than 65534 bytes of memory.

33. More names are declared then can be held in the memory available to the compiler. This problem can be avoided in two ways. Fewer names can be declared in the program (this is not usually a good idea), or memory can be made available to the compiler. The editor's use of memory can be decreased to leave more for the compiler by saving some or all of the program on tape and making the compiler read it from tape by using the $INCLUDE facility.

34. More than 255 procedure and/or functions are declared.

35. A procedure or function is declared with more than 255 parameters.

36. More than 26 text files are declared in the PROGRAM header .

37. The combined nesting of CASE statements, FOR statements and parentheses in expressions is too great.

42. A name is expected in one of these situations:

    1. 1. The name of a program in a PROGRAM header.
    2. 2. The name of a text file in a PROGRAM header.
    3. 3. The name of a variable being declared after the VAR reserved word.
    4. 4. The name of a FUNCTION or PROCEDURE being declared.
    5. 5. The name of a parameter in a FUNCTION or PROCEDURE heading.
    6. 6. The name of a type after a colon in the parameter list of a FUNCTION or PROCEDURE declaration.
    7. 7. The name of a type after a colon following the parameter list of a FUNCTION declaration.

Remember that Pascal reserved words may not be used as names in Pascal programs and that Partial Pascal, unlike full Pascal, does not allow the use of VAR, FUNCTION or PROCEDURE in a parameter list.

Example:

```
PROCEDURE UPDATE (VAR  X:INTEGER );
```

49. THEN does not follow the condition in an IF statement. Example:

```
IF A<0
   A := -A
```

52. Neither TO nor DOWNTO follows the first expression in a FOR statement. Example:

```
FOR. X :=1 TOO 2 DO
```

55. DO does not follow the condition of a WHILE statement or DO does not follow the second expression of a FOR statement. Example:

```
FOR X:=1 TO 2 BEGIN
```

58. OF does not follow the expression in a CASE statement or OF does not follow the word ARRAY in a VAR declaration. Example:

```
CASE ABS(Y) FO
```

62. A program does not begin with the word PROGRAM.

71. One of the following applies:

    1. Neither a semi-colon nor the word OTHERWISE nor the word END follows a statement in a CASE statement.

    2. A statement in a series of statements beginning with BEGIN is not followed by a semicolon or the word END. Example:

```
CASE ABS(Y) OF
1, 2: Y: = 0   ( +MISSING SEMICOLON+ )
3,
```

72. A statement following the word REPEAT is followed neither by a semicolon nor by the word UNTIL. Example:

```
REPEAT
 X:=X-1
WHILE A(.X.)<>Y
```

74. The word following PACKED in a VAR declaration is not ARRAY. Example:

```
VAR X:PACKED INTEGER,
```

84. An equal sign is expected after a name being declared in a CONST or TYPE declaration. Example:

```
CONST TWO :=2,
```

92. Something other than a comma or a closing parenthesis follows the name of a TEXT file in a read, readln, write or writeln statement. Example:

```
WRITELN(OUTPUT:4)
```

93. One of the following applies:

    1. Something other than a comma or colon follows a name being declared in a VAR declaration.
    2. Something other than a comma or colon follows a constant in a CASE statement. Example:

```
VAR X,Y,Z, INTEGER;
```

94. A required semicolon is not present in one of the following places:

    1. After the constant value in a CONST declaration.
    2. After the type in a VAR declaration.
    3. After a PROGRAM, FUNCTION or PROCEDURE header.
    4. After the END ending a function or procedure.
    5. After the word FORWARD. Example:

```
VAR X,Y: INTEGER
BEGIN
```

95. An assignment symbol, :=, is not present after a variable being assigned a value. Example:

```
    VAR B:BOOLEAN;
BEGIN
 B=FALSE
```

96. A type began with a constant which was not followed by an ellipsis. Example:

```
VAR ONETOTEN:1.10;
```

97. The terminating END of a program is not followed by a period. Example:

```
PROGRAM SPP(OUTPUT);
BEGIN
 WRITE(‘HELLO, WORLD’)
END;
```

98. One of the built-in procedures , write or read , is not followed by an opening parenthesis. Example:

```
WRITE;
```

99. A closing parenthesis is expected at the end of an expression in parenthesis or a comma or closing parenthesis is expected in one of the following situations:

1. In the list of parameters in a FUNCTION or PROCEDURE invocation.
2. In the list of names of an enumerated type.
3. In the parameter list of a FUNCTION, PROCEDURE or PROGRAM heading. Example:

```
PROCEDURE SUB(HOWMUCH,F OM):
```

100.An opening bracket, (., is expected in one of the following places:

1. After the word ARRAY in a VAR declaration.
2. After the name of an array in an assignment statement or in an expression. Example:

```
VAR ABCD:ARRAY(.1..4.) OF CHAR;
BEGIN
ABCD(1):="A"
```

101.A closing bracket is expected in one of the following places:

1. After the index in the declaration of an array available.
2. After the index of an array variable in an assignment statement or expression. Example:

```
VAR ABCD:ARRAY(.1..4) OF CHAR;
```

7.2 Completion codes

One of these codes is displayed by Partial Pascal as the first number on the last line of the TV every time a program ends execution. (The second number will determine in a future release of Partial Pascal, where in the program execution ended.) If the program ended by executing the halt procedure, the program determines its own completion code by the value of the parameter supplied to the halt procedure. Partial Pascal displays that value MOD 256. Otherwise, Partial Pascal assigns one of the following completion codes.

28. The program ended because it attempted to divide a number by zero. The value following the word DIV was 0.

29. The program ended because it attempted to take a number module zero or a negative number. The value following the word MOD was less than or equal to 0.

30. The program ended because it ran out of memory. Try to make more memory available to the program by deleting as much as possible from the editor's memory.

31. The program ended because an array index was out of the bounds allocated for that array. The index was either less than 0 or greater than the upper bound declared for that array.

32. The program ended because a non-numeric character (other than a space) was found when the program tried to read a number from a text file.

33. The program ended because a number read from a text file was either greater than 32767 or less than -32768.

34. The program ended because it tried to read from a text file that was at end of file. That is, either reset had not been issued for the file or rewrite for the file is been issued since its most recent reset, or the file had no data to be read (null file), or all the data had already been read at the tape of the read or readln that caused the program to end.