when the boolean expression is false. In any case, after the *then* or *else* clause (if present) is executed, control falls through to the next statement following the *IF* statement. Following are some examples:

```
IF I < J THEN I := I + 1 ELSE J := J + 1;
IF J*2 = 50 THEN BEGIN
  J := 5;
  I := I*3
  END
```

The last example shows how the *BEGIN* statement can be used with the *IF* statement.

9.3 WHILE Statement

The *WHILE* statement provides the capability of *repetitively* executing a given statement while a boolean expression is *true.* This is one of Pascal's *structured looping constructs.* The general form of the *WHILE* statement is as follows:

WHILE <boolean expression> DO <executable statement>

In the WHILE statement the <boolean expression> is evaluated and if found to be *true,* the <executable statement> following the *DO* is executed and the process is repeated. This continues until the <boolean expression> is found to be *false.* At that time, the <executable statement> is not executed and control falls through to the statement following the *WHILE* statement. Note that if the <boolean expression> is *false* when the *WHILE* statement is first executed, the <executable statement> following the *DO* is not executed at all.

Normally, the <executable statement> will change the value of one or more of the variables used in the <boolean expression>. Following are some examples:

```
WHILE I < J DO I := I + 3;
WHILE J > I+3 DO BEGIN
  J := J / 3;
  I := I + 1
  END
```

9.4 REPEAT Statement

The *REPEAT* statement provides the capability of repetitively executing a given statement until a boolean expression is *false*. The general form of the *REPEAT* statement is as follows:

```
REPEAT <executable statement> ;
                    .
                    .
                    .
            <executable statement>
UNTIL <boolean expression>
```

In the *REPEAT* statement the <executable statement>s following the *REPEAT* are executed. The <boolean expression> is then evaluated and if *false* the process is repeated. This continues until the <boolean expression> is found to be *true*. At that time, control falls through to the statement following the *UNTIL*. Note that if the <boolean expression> is *true* when the *REPEAT* statement is first executed, the <executable statement>s following the *REPEAT* are still executed one time.

Normally, the <executable statement>s will change the value of one or more of the variables used in the <boolean expression>. The following are some examples:

```
REPEAT I := I + 3 UNTIL I > J;
REPEAT
  J := J / 3;
  I := I + 1
UNTIL J < I + 3
```

9.5 FOR Statement

The *FOR* statement provides the capability of repetitively executing a statement while explicitly varying an ordinal variable. The general form of the *FOR* statement is as follows:

```
FOR <assignment statement> TO <expression> DO
      <executable statement>

                    or

FOR <assignment statement> DOWNTO <expression> DO
      <executable statement>
```

In both the *TO* and *DOWNTO* versions the <assignment statement> is executed first. The ordinal variable identifier to which the assignment is made is used as the *loop counter*. The testing and varying of the loop counter is different in the *TO* and *DOWNTO* versions.

In the TO version, the following sequence is performed:

1. If the loop counter is greater than the <expression>, processing in the *FOR* loop is terminated and control falls through to the next statement following the *FOR* loop. Otherwise, the following additional steps are performed.
2. The <executable statement> (which may be a compound statement) is executed.
3. The loop counter is advanced to the next higher value (see SUCC built-in function).
4. Control goes back to the first item in this sequence.

In the DOWNTO version, the following sequence is performed:

1. If the loop counter is less than the <expression>, processing in the *FOR* loop is terminated and control falls through to the next statement following the *FOR* loop. Otherwise, the following additional steps are performed.
2. The <executable statement> (which of course may be a compound statement) is executed.
3. The loop counter is reduced to the next lower value (see PRED built-in function).
4. Control goes back to the first item in this sequence.

Normally the <executable statement> will reference the loop counter although this isn't always the case. Following are some examples:

```
FOR I := 1 TO 3 DO MyColors[i] := Red;
FOR J := 0 TO 200 DO
  FOR I := 1 TO 6 DO OurColors[J,I] := Yellow;
FOR ColorVar := Green TO Orange DO
  NumbersVar[ColorVar] := 3;
```

In the second example, the <executable statement> of the first *FOR* statement was itself a *FOR* statement. The second *FOR* loop will execute to completion (6 iterations) for each iteration of the first

FOR loop. In the last example, the loop counter is an enumerated type and is used as the subscript of an *array* type variable.

9.6 CASE Statement

The *CASE* statement provides the ability to execute one of several statements depending of the value of an ordinal expression. This ordinal expression is called a *selector*. Following is the general form of the *CASE* statement:

```
CASE <ordinal expression> OF
  <constant list> : <executable statement>;
                    .
                    .
                    .
  <constant list> : <executable statement>
  ELSE <executable statement>
  END
```

The <constant list> is a list of type compatible constants separated with commas. The <ordinal expression> is evaluated and compared with each constant sequentially in each <constant list>. If the <ordinal expression> is found to equal a constant, the comparing is stopped and the <executable statement> immediately following that particular constant is executed and control is then passed to the next statement following the *CASE* expression. If none of the constants match the <ordinal expression> and the *ELSE* clause is present, then the statement following the *ELSE* is executed.

The *ELSE* clause is a common extension found in most Pascals (sometimes as an OTHERWISE clause). It is optional, but if present must follow the last case and precede the *END*. Following is an example:

```
CASE I*5+J OF
  7,9 : J := 15;
  11,12,13,14 : BEGIN I := 3; J := 2 END;
  1 : I := J + 5
  ELSE J := 0
  END;

CASE MyColors[I] OF
  Red, Orange : MyColors[I] := Green;
  Blue : I := 3
  END
```

In both examples, you will notice at least one case which has only 1 constant in its <constant list>. In the second example, the ordinal expression is of an enumerated type.

9.7 GOTO Statement

The *GOTO* statement provides the ability to cease program execution at the point of the *GOTO* statement, and then resume program execution at the point in the program identified with the corresponding label specified in the *GOTO* statement. For those used to programming in BASIC, this feature is very familiar. The **DEFT Pascal** Compiler, however, only allows a *GOTO* to reference a *label* that is defined within the same block as the *GOTO*. The following is an example:

```
GOTO 580;
```

Where 580 is a label used to identify an executable statement within the same block as the GOTO statement.

9.8 EXIT Statement

The *EXIT* statement provides the ability to deactivate a block before coming to the block's *END* statement. The *EXIT* statement is not part of standard Pascal but a form of it is found in a number of commercially available compilers. The syntax is as follows:

```
EXIT
```

When this statement is encountered, the active block in which it is found is deactivated and no further statements within that block are executed. Note that the block being referred to is one associated with a *procedure*, *function* or *program*.

Typically, the *EXIT* statement is used in conjunction with one of the other control statements in order to conditionally continue execution within a block. *EXIT* can be used to deactivate the *program* block in which case program execution terminates and control returns to BASIC.

9.9 WITH Statement

The *WITH* statement provides the ability to reference multiple fields within the same record with one statement. One or more fields of a record can be referenced within a *WITH* statement by their field names alone provided the remaining part of the name, i.e. the record name (eventually qualified by field names), is mentioned in the *WITH* statement. The syntax is as follows:

```
WITH <variable> DO <statement>;
```

For example:

```
WITH RecordName DO Field1 := X;
```

This example is equivalent to:

```
RecordName.Field1 := X;
```

The following is another example:

```
WITH RecordName.GroupName DO
  BEGIN
    Field1 := X;
    Field2 := Y;
    Field3 := Z;
  END;
```

This example is equivalent to:

```
RecordName.GroupName.Field1 := X;
RecordName.GroupName.Field2 := Y;
RecordName.GroupName.Field3 := Z;
```

Several *WITH* statements can be nested. But since field identifiers are local to the record in which they are defined, different records can have identical field identifiers. In the case of nested *WITH*s, ownership of like field identifiers is determined by the innermost *WITH* statement. This is consistent with the Pascal rules of scope. An example of nested *WITH* is as follows:

```
WITH Record1 DO
  WITH Record2 DO
    WITH Record3 DO Field1 := X;
```

DEFT Pascal allows up to eight levels of *WITH* nesting. Also, the <variable> in a *WITH* statement cannot contain a pointer dereference or a subscripted array.

10 Input/Output

Any program is totally useless unless it can, in some way, change something external to the processor. Input/Output statements allow a program to receive outside stimulus (Input) and provide a response (Output).

With **DEFT Pascal**, the primary input statements are *reset, get, read, readln* and the builtin functions *eof* and *eoln*. The primary output statements are *rewrite, put, write, writeln* and *close*. These statements and builtin functions provide a device independent mechanism for reading data from the keyboard, cassette and disks, and for writing data to the screen, printer, cassette and disks.

10.1 File Names

The device or file (a portion of the total storage on a cassette or disk) to be used in a series of Input/Output operations is identified with a *file name*. The format of a filename is as follows:

<filename>/<ext>:<device#>

This is the same format that BASIC uses for Disk files. However, by extending the device numbers, **DEFT Pascal** also uses it for the keyboard, screen, tape and printer. The <filename> is 0 to 8 ASCII characters. The extension is 0 to 3 ASCII characters. The device numbers range from -3 to 3 with the following meanings:

-3 Keyboard/Screen
-2 Printer
-1 Cassette Tape
0 Disk drive 0
1 Disk drive 1
2 Disk drive 2
3 Disk drive 3

As can be seen, the positive device numbers corresponds to BASIC's drive numbers. The negative device numbers correspond to BASIC's device numbers with the exception that the Keyboard/Screen is -3 rather than 0.

All of the fields are optional in different circumstances. When a device number of -3 or -2 is specified, there is no need for a <filename> or <extension>. When a device number of -1 is specified, the <extension> is not used. For device numbers 0 thru 3, a default <extension> is always present depending on the program being run. When a device number is not specified, 0 is assumed. Following are some examples:

:-3	**Keyboard/Screen**
:-2	**Printer**
MYFILE:-2	**Printer (filename ignored but allowed)**
TAPEFILE:-1	**Cassette Tape File**
DISKFILE/ASM	**Assembler source file on disk drive 0**
F2NAME:1	**File is on disk drive 1, default extension used**

10.2 File Variables

Rather than giving the file name in each Input/output statement and function, a *file type variable* is used. This file type variable is initialized by a *reset* or *rewrite* statement which associates it with a file name. Other statements and functions which subsequently reference this variable then cause operations to be performed to the corresponding device or portion thereof.

A file variable has a *window* which can be read (input) or written to (output) depending on how the file variable was originally initialized (using the *reset* or *rewrite* statements). You access this window by dereferencing the file variable much like the way a pointer variable is dereferenced. The procedures and functions described below provide the ability to move data between this file window and an external device or file.

10.3 INPUT and OUTPUT File Variables

There are two predefined *file of char* (text) variables available with **DEFT Pascal**. The variable *input* is pre-initialized for access to the keyboard as though a *RESET (INPUT, ':-3')* statement (see below) had been executed before your program began. The variable *output* is pre-initialized for access to the screen as though a *REWRITE (OUTPUT, ':-3')* statement (see below) had been executed before your program began.

The existence of these two pre-defined and pre-initialized variables provides the following benefits:

1. You do not need to use *reset* or *rewrite* to initialize these variables before using them in *readln*, *writeln*, etc.
2. When using *read*, *readln*, *eoln* and *eof* you can omit the <file variable> parameter in the statement and the default file variable *input* will be used.

3. When using *writeln*, *write*, *page* and *close* you can omit the <file variable> parameter in the statement and the default file variable *output* will be used. Note that although it is permissible to use *close* with *output*, it is not necessary.

NOTE: The *input* and *output* files are actually the same file which has been specially initialized to allow both input from the keyboard and output to the screen. For this reason, it is recommended that you do not use the *reset* or *rewrite* statements with these files. When you wish to do I/O to the printer, cassette or disk, setup a separate file variable as shown in the general I/O examples further on.

10.4 Overall Example

Below is an example of a simple program that prompts at the screen for a filename to be entered and then reads that file and writes it to the printer. The filename that is entered can be any of those described above in the section on *File Names*.

```
PROGRAM CopyFile (Input, Output);

VAR InFile, OutFile : Text;
    FileName : String;
    Data : String (255);

BEGIN
  Page; (* clear the screen *)
  WRITE ('FILE NAME: ');
  READLN (FileName);

  RESET (InFile, FileName);
  REWRITE (OutFile, ':-2');

  WHILE NOT EOF (InFile) DO BEGIN
    READLN (InFile, Data);
    WRITELN (OutFile, Data);
    END;

  CLOSE (OutFile);
  END;
```

In this example, *InFile* and *OutFile* are file *variables* and ':-2' is a string constant which contains a file *name*. The *reset* statement associates the file whose name has been entered into the string variable *FileName* with the file variable *InFile* and initializes it for reading. The *rewrite* associates the printer (device number -2) with

the file variable *OutFile* and initializes it for writing.

The *while* loop causes a check for end of file on *InFile* BEFORE reading the first record. The *close* statement at the end, forces any remaining buffered data to be written. When writing to the printer or the screen it is not absolutely necessary to do the *close*, but it is recommended in case the program may be changed to output to the disk or cassette.

10.5 Lazy Keyboard Input

In order to provide an easy to use interface for the keyboard, **DEFT Pascal** incorporates the concept of *lazy keyboard input*. This involves waiting until a *read* or *readln* statement is executed before actually performing an input from the keyboard.

Standard Pascal requires that the internal buffer be prefilled so that the *eof* and *eoln* and file dereferencing operations can be performed. If this were done for keyboard input, you would have to enter data into the keyboard immediately after executing any Pascal program (before your program actually begins executing any statements). This would make it very difficult for you to synchronize your prompts (via *write* and *writeln* statements) with the corresponding inputs (via *read* and *readln* statements).

The result of the *lazy keyboard input* is that *eof* and *eoln* reflect the status as of the end of the **last** *read* or *readln* statement. For example:

```
WHILE NOT EOF DO BEGIN
  WHILE NOT EOLN DO BEGIN
    READ (X);
    WRITE (X);
    END;
  READLN;
  WRITELN;
  END;
```

If the first key that you hit is the *CLEAR* key (to indicate *eof* and *eoln* on the keyboard) the inside loop will still execute once since the prompt does not appear until the *READ (X);* statement is being executed. *X* will retain whatever value it had before the *read* unless *X* is a *char* in which case it will contain a CHR(13).

Remember, *lazy keyboard input* is only used with the keyboard. Your cassette and disk input operations are pre-buffered and conform to the Pascal standard.

10.6 CLOSE Statement

This statement is required for output files (initialized via *rewrite*) to cassette or disk in order to ensure that all data has been written to the device and the directory or trailer has been written. It may also be used for screen and printer files but has no effect. Once this statement is executed, the file variable is considered uninitialized and must be initialized again (with either *rewrite* or *reset*) in order to be used. The format of the statement is:

CLOSE (<file variable>)

As mentioned above, if <file variable> is omitted, the *output* file is assumed.

10.7 EOF Function

This is a Boolean function which specifies whether *end of file* has been reached on a particular file. This function can be used on a file of any type. Its definition is:

FUNCTION EOF (VAR FileVar : Text) : Boolean;

It can also be used as though it had no parameter and the default file *input* will be assumed. Note that *eof* can be indicated from the keyboard by terminating the last line with the *CLEAR* key instead of the *ENTER* key.

10.8 EOLN Function

This is a boolean function which specifies whether an *end of line* character is next in the window on a *file of char*. Its definition is:

FUNCTION EOLN (VAR FileVar : Text) : Boolean;

It can also be used as though it has no parameter and the default file *input* will be assumed.

10.9 FILEERROR

This is an *integer* function which returns an indication of whether a file I/O error occurred on a particular file and what the error was if it did occur. This function can be used with a file of any type. Its definition is:

FUNCTION FILEERROR (VAR FileVar : Text) : Integer

It can also be used as though it had no parameter and the default file *input* will be assumed. The *integer* return is a number from 0 to -5. The possible error numbers are as follows:

- 0, *No Error*
- -1, *End of File* - The end of a given file has been reached.
- -2, *I/O Error* - This indicates that some hardware oriented problem occurred.
- -3, *File Not Found* - The file specified was not found.
- -4, *Illegal Operation* - This indicates that you attempted a read operation on an output file or a write operation on a input file. It can also occur if you attempt to do a *reset* to the printer.
- -5, *Device Full* - While doing a *rewrite* or other write operation, the device became full.

NOTE: *eof* will return a *true* anytime *fileerror* would return a non-zero. *Fileerror* is a **DEFT Pascal** extension and is not part of standard Pascal.

10.10 GET Statement

This statement (implemented as a built-in procedure) is used to input data from cassette or disk via a *file* that was previuosly initialized with the *reset* procedure. The format of the statement is:

GET (<file variable>)

The action of this procedure is to move the file window over the next element in the file. The *get* statement cannot be used in **DEFT Pascal** with a *file of char*.

10.11 PAGE

This procedure is used to output an ASCII formfeed to the specified file. When a formfeed is output to the screen, the equivalent of BASIC's *CLEAR* is performed. When a formfeed is output to the printer, it will skip to the top of the next page. The format of the statement is:

PAGE (<file variable>)

As mentioned previously, if <file variable> is omitted, the OUTPUT file variable is assumed.

10.12 PUT Statement

This statement (implemented as a built-in procedure) is used to output data to cassette or disk via a *file* that was previously initialized with the *rewrite* procedure. The format of the statement is:

PUT (<file variable>)

The action of this procedure is to output the contents of the file window to the external device or file and then empty the window. The *put* statement cannot be used in **DEFT Pascal** with a *file of char*.

10.13 RESET and REWRITE Statements

These statements are used to initialize *file* type variables for use with subsequent Input/Output statements and functions. You can think of these statements as procedures with the following definition:

```
PROCEDURE RESET  (VAR FileVar : Text;
                  VAR Filename : String;
                  VAR DefExtension : String);

PROCEDURE REWRITE (VAR FileVar : Text;
                   VAR Filename : String;
                   VAR DefExtension : String);
```

You will only need to use one or the other of the two statements. *Reset* initializes the *FileVar* for input from the specified *Filename*. When using *reset* with a disk or cassette file, a file by the name of *Filename* must already exist on that device.

Rewrite initializes the *FileVar* for output to the specified *Filename*. When using *rewrite* for output to disk, if the specified disk already has a file by the name of *Filename*, it will be deleted. A new file is then created by the name of *Filename*. On cassette, a file is created by the name of *Filename* at the current spot on the tape.

The *DefExtension* specifies the default filename extension to use if one is not included as part of the *Filename* string. This parameter is optional and if not present the default extension is blank.

10.14 READ Statement

This statement is used to input data from the keyboard, cassette or disk via a *file* that was previously initialized with the *reset* procedure. The format of the statement is:

READ ([<file variable>,] <variable>, ... ,<variable>)

As mentioned above, if <file variable> is omitted, the file *input* is assumed to be referenced.

Reading from a Typed File - When using *read* to read from a *file of <type>* where *<type>* is not *char*, the <any variable> must be of the same type as the *file*. When the *read* is executed, the size of the <type> is used to determine the number of bytes to transfer. Essentially, each *read* returns the next sequential occurance of the <type> in the *file*. For example, *READ(F, X)* is exactly the same as:

```
X := F^;
GET (F);
```

Note that *typed files* can only be used with cassette and disk.

Reading from a FILE OF Char - If the <file variable> is a *file of char* then the file is assumed to consist of a set of *lines* and the action of the *READ (F,X)* statement depends on the *type* of X. The following describes the legal *types* of *X* and the associated actions of the *read* statement:

1. *Char* - The next byte of the line is directly assigned.
2. *String* - The value of the string becomes the value of the remainder of the line. The line is truncated if necessary to fit in the string.
3. *Real* - The next group of characters delimited by blanks and/or *end of line* characters is processed by *encodereal* and the result is

stored in the variable.

4. *Integer* - The next group of characters delimited by blanks and/or *end of line* characters is processed by *encode* and the result is stored in the variable.

5. *Boolean* - The same as integer except that only the numbers 0 (for FALSE) and 1 (for TRUE) are legal. You will get unpredicable results with other values.

6. *Enumerated* - The same as integer except that only the subset of numbers 0 through 255 that apply to the given type are legal. Other values will convert to non-existent members of the type.

Some examples of use:

```
READ (IntVar, IntVar2);          (* integer from keyboard *)
READ (TapeFile, StringVar);      (* string from cassette  *)
READ (DiskFile, CharVar);        (* char from disk        *)
READ (KeyBoardFile, ColorVar)    (* enumerated from keyboard *)
```

10.15 READLN Statement

This statement is identical to the *read* statement when used with *typed files* and is almost the same when used with a *file of char* except that after all the variables are read, the window is moved past the next *end of line* character.

The *readln* statement can be used with no <variables> in order to position the file window past the next end of line character without reading a data before that point.

10.16 WRITE Statement

This statement is used to output data to the screen, printer, cassette or disk. The format of the statement is:

WRITE (<file variable>, <data>:<width>:<decimal>,...,
<data>:<width>:<decimal>)

As mentioned above, if <file variable> is omitted, the file *output* is assumed to be referenced. The action of the *write* statement depends on the type of the <file variable>.

Writing to a Typed File - If the <file variable> is *not* a *file of char* then the file is referred to as a *typed file* and the action of the *write (F,X)* statement is exactly the same as:

```
F ^ := X;
PUT (F);
```

where the *type* of the file *F* must be the same as the *type* of the variable *X*. Essentially, the current value of the variable *X* is assigned to the next element in the file *F*. The <width> and <decimal> parameters cannot be specified.

Note that *typed files* can only be used with cassette and disk.

Writing To a File of Char (Text) - If the <file variable> is a *file of char* then the action of the *write (F,X)* statement depends on the *type* of *X*. The following describes the legal *types* of *X* and the associated actions of the *write* statement:

1. *Char* - The next byte of the file is directly assigned from the contents of the character followed by <width>-1 blanks. The <decimal> parameter must not be specified.

2. *String* - The value of the string is output. If the <width> is zero or not present, the number of columns reserved will be the size of the string. If the <width> is less than the size of the string, only the first <width> characters will be output. If <width> is greater than the size of the string, blanks will be output following the string. The <decimal> must not be specified.

3. *Real* - The *decodereal* procedure is used to convert the real to the string of characters to output. The <width> specifies the size of the ASCII representation to output. If the <width> is too small to fit the number, asterisks are output to indicate overflow. The default value is 6.

   The <decimal> specifies the number of places to the right of the decimal point that should be output. If <decimal> is not present or negative, scientific notation will be used.

4. *Integer* - The *decode* procedure is used to convert the integer to the string of characters to output. The <width> parameter specifies the size of the string to output with the number right-justified within the string. If <width> is not specified it defaults to 6. The <decimal> parameter must not be specified.

5. *Boolean* - The same as integer except that only the numbers 0 (for *false)* and 1 (for *true)* are be output.

6. *Enumerated* - The same as integer except that only the subset of numbers 0 through 255 that apply to the given *type* are output.

Some examples of use:

```
WRITE (IntVar);                    (* Integer to screen          *)
WRITE (TapeFile, StringVar);       (* String to cassette         *)
WRITE (DiskFile, CharVar);         (* Char to disk               *)
WRITE (PrntrFile, ColorVar);       (* Enumerated to printer *)
WRITE (IntVar:3);                  (* 3 column output spec       *)
WRITE ('The answers are ',R*3.4:10:2,
       ' and ',S/4.2::0);          (* multiple items in one *)
WRITE (Printer, '':30, 'Centered Title');
                                   (* forcing blank padding *)
```

Since *write* does not output a carriage return at the end (as *writeln* does) it is usually used for prompting and for multiple *writes* to a single line followed by a *writeln* (see following section).

10.17 WRITELN Statement

This statement is used to perform the same operations as a *write* statement. When used with a *typed file* it is identical to the *write* statement. When used with a *file of char* (Text) it is almost the same except that after all the specified outputs have been made, a carriage return *(end of line* character) is also output. This statement also allows no <data> items at all to be specified so that only the carriage return will be output. All the examples shown for *write* also apply for *writeln*. Following are some additions:

```
WRITELN;                  (* carriage return to OUTPUT file     *)
WRITELN (DiskFile);       (* carriage return to DiskFile file *)
WRITE (CHR(13))           (* equivalent of WRITELN;             *)
```

11 Builtin Procedures and Functions

This section describes a number of predefined functions and procedures that are available with **DEFT Pascal.** Although definition statements are shown in each of the descriptions, these are purely for informational purposes and are not to be used in your program.

11.1 ABS

This is an *integer* or *real* function that returns the absolute value of the value parameter that is passed to it. The Function definition is:

```
FUNCTION ABS (Value : Integer) : Integer
                    or
FUNCTION ABS (Value : Real) : Real
```

11.2 ARCTAN

This is a *real* function which is used to compute the size of an angle whose tangent is passed to the function. The size of the angle returned by the funciton is in the form of a number of radians. The function definition is:

```
FUNCTION ARCTAN (Tangent : Real) : Real
```

11.3 CHR

This is a *character* function that returns the ASCII character for the binary value specified in the passed value parameter. The function definition is:

```
FUNCTION CHR (Value : Integer) : Char
```

This function allows you to *break* type from integer to char. See *Advanced Pascal* for a more general and structured type breaking language extension.

11.4 COS

This is a *real* function which is used to compute the cosine of an angle. The size of the angle is passed to the function in the form of a number of radians. The function definition is:

```
FUNCTION COS (Radians : Real) : Real
```

11.5 CURSOR

This is a builtin procedure that allows you to position the cursor to any of 512 positions on the screen. The upper left-hand corner is position 0. Consecutive positions proceed horizontally across the screen with the beginning of each line being a multiple of 32. The lower right-hand corner is position 511. The procedure definition is:

PROCEDURE CURSOR (Position : Integer)

Note that *position* is taken modulo 512 when used.

11.6 EXP

This is a *real* function which is used to compute the value of *e* (2.718281828) to a specific power. The function definition is:

FUNCTION EXP (Power : Real) : Real

11.7 LN

This is a *real* function which is used to compute the natural logarithm of a positive number. The function definition is:

FUNCTION LN (Number : Real) : Real

11.8 MARK

This is a *general* procedure which is used to *mark* the current state of the *heap* for use later by the *release* procedure. The procedure definition is:

PROCEDURE MARK (VAR PtrVar : Ptr)

where *PtrVar* can be a *pointer* to any *type*. Any variables allocated by the *new* procedure after saving the *heap* state in *PtrVar* will be deallocated when *release* is later called using the saved *Ptrvar*.

11.9 MEMAVAIL

This is an *integer* function which is used to determine the number of bytes of memory remaining in the *heap*. The *memavail* function is a **DEFT Pascal** extension and is not a part of standard Pascal. The function declaration is as follows:

FUNCTION MEMAVAIL : Integer;