For example:

```
WHILE MEMAVAIL >= SIZEOF (Struct) DO
  BEGIN
    NEW (Struct -Ptr);
    NumCopy := SUCC (NumCopy);
  END;
```

This example is using the value of *memavail* to determine how many copies of a data structure can be dynamically allocated. NOTE: If the *heap* available is greater than 32767 bytes, then *memavail* will return 32767.

11.10 NEW

This is a *general* procedure which is used to dynamically allocate a variable from the *heap* and assign its reference to a *pointer type variable*. The *type* of variable allocated is dependent on the *type* of the pointer. The procedure definition is:

```
PROCEDURE NEW (VAR PtrVar : Ptr)
```

where *PtrVar* can be a *pointer* to any *type*.

11.11 ODD

This is a boolean function that returns a *true* if the integer value that is passed is an odd number (not evenly divisible by 2) or a *false* if the value is even. The function definition is:

```
FUNCTION ODD (Value : Integer) : Boolean
```

11.12 ORD

This is a *general* integer function that can take *any* ordinal type expression and convert it to an integer. The internal binary value of the specified type is treated as though it were an integer. The function definition is:

```
FUNCTION ORD (Value : OrdinalType) : Integer
```

where *Value* can be any *type* of ordinal expression. See *Advanced Pascal* for a more general and structured type breaking language extension.

11.13 PRED

This is a *general* ordinal function that returns the next lower ordinal value of the same type as its argument. For *integers*, it is the equivalent of subtracting 1 from the number. For *chars*, it is the ASCII character with the next lower binary value. For *booleans*, it is only legal when the argument is *true* returning *false*. For *enumerated types*, it is the next preceding enumerated value. Using our *color type* example:

```
ColorVar := Green;
ColorVar := PRED (ColorVar)
```

ColorVar is now *Red*.

11.14 RELEASE

This is a *general* procedure which is used to deallocate variables from the *heap* which were orginally allocated via the *new* procedure. The procedure definition is:

```
PROCEDURE RELEASE (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* (via the *mark* procedure) will be deallocated when *release* is later called using the saved *PtrVar*.

11.15 ROUND

This is an *integer* function which is used to round a *real* value to the nearest *integer* value. The function definition is:

```
FUNCTION ROUND (VAR Value : Real) : Integer
```

11.16 SIN

This is a *real* function which is used to compute the sine 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 SIN (Radians : Real) ; Real
```

11.17 SIZEOF

This is a *general* integer function which is used to compute the size (in bytes) of a variable or type. The function definition is:

```
FUNCTION SIZEOF (GenVar : AnyType) : Integer
```

NOTE: *Sizeof* is a **DEFT Pascal** extension and is not part of standard Pascal.

11.18 SQR

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

```
FUNCTION SQR (Number : Real) : Real
```

11.19 SQRT

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

```
FUNCTION SQRT (Number : Real) : Real
```

11.20 SUCC

This is a *general* ordinal function that returns the next higher ordinal value of the same type as its argument. For *integers*, it is the equivalent of adding 1 from the number. For *chars*, it is the ASCII character with the next higher binary value. For *booleans*, it is only legal when the argument is *false* returning *true*. For *enumerated types*, it is the next succeeding enumerated value. Using our *color type* example:

```
ColorVar := Red;
ColorVar := SUCC (ColorVar)
```

ColorVar is now *Green*.

11.21 TRUNC

This is an *integer* function which is used to truncate a *real* value to its *integer* value. The function definition is:

```
FUNCTION TRUNC (VAR Value : Real) : Integer
```

12 DEFT vs. Standard Pascal

DEFT Pascal conforms closely to the ISO Draft Proposal for standard Pascal. The following list identifies the major areas where **DEFT Pascal** differs from the standard. In many of these areas, **DEFT Pascal** differs because it has features similar to those in **UCSD Pascal.**

- Program parameters are allowed but ignored. The predefined files INPUT and OUTPUT are always defined and opened.
- Heap management is via MARK and RELEASE rather than DISPOSE.
- Strings are variable length and implemented via the predefined type STRING(n) rather than being fixed length and implemented as PACKED ARRAY[1..n] OF CHAR.
- GOTOs may only reference LABELs within the current block.
- Lazy Keyboard Input is implemented in order to allow interactive I/O.
- RESET and REWRITE require a second parameter and may optionally take a third parameter for specifying external filenames and default extensions.
- Procedural parameters are not supported. A procedural parameter is a parameter which is itself a procedure.
- PACK and UNPACK statements are not supported. However, since **DEFT Pascal** does not have the standard restrictions on the use of PACKed variables, the major reason for the use of these statements is non-existent.
- GET and PUT cannot be used with TEXT files.
- The <record variable> in the WITH statement cannot include a subscripted array or pointer dereference.
- Conformant arrays are not supported. However, extensions to ARRAY typing provide a facility for passing actual parameters of varying numbers of elements to an individual procedure.

In addition, **DEFT Pascal** provides a number of minor and major enhancements. The minor enhancements are as follows:

- The bitwise integer operators XOR, LSL, LSR, AND and OR are supported.
- Equality comparisons between like structured types is allowed.
- I/O of enumerated types to and from TEXT files is allowed.
- An EXIT statement for prematurely returning from a procedure or function.
- FILEERROR and SIZEOF builtin functions and CURSOR builtin procedure.

A complete definition of all the major enhancements is contained in the section on *Advanced Pascal Language Extensions*.

13 Error Messsages

The **DEFT Pascal** compiler generates error messages in the source listing at those points where it detects either syntax errors or encounters I/O errors while processing a source file. The compiler prints a line of dashes followed by an up arrow and the message. The arrow indicates where the error was *detected* which is not necessarily where it *occurred*. There are a number of different error messages which are listed below:

COPY NESTING TOO DEEP

A *%C* compiler control has been found in a source file at the maximum copy depth. See *How To* for information on the *%C* compiler control.

DUPLICATE SYMBOL

The constant, type, variable, procedure or function name being defined has already been used at the current block level to define another constant, type variable, procedure or function.

EXPECTING ...

This message will have various words or symbols following it depending on what the compiler was expecting to find next in the source file. This token was not found at this point.

EXPR TYPE ERROR

This message indicates that the expression type expected at this point was not what was encountered. You may have to use a type transfer function (see *Advanced Pascal)* to let the compiler know that you want to use a different type.

FILE OPEN ERROR

The source file specified on the **DEFT Pascal** Compiler's screen, the *PASCALIB/EXT:0* file or a file specified in a *%C* compiler control resulted in an error when an open was attempted.

INVALID CONSTANT

At a point in the program where a constant was expected, a legal constant was not found.

INVALID FACTOR

While processing an expression, an invalid factor was encountered.

INVALID IDENTIFIER

At a point in the program where an identifier was expected, a legal identifier was not found.

INVALID ORDINAL TYPE

An error was detected at this point while processing an ordinal type.

INVALID SIGNED TERM

A signed term was encountered while processing an expression that was not of type integer or real.

INVALID STATEMENT

An unknown type of statement was encountered at this point in the program.

INVALID TYPE DECLARATION

An error was encountered while processing a type declaration.

INVALID TYPE IDENTIFIER

At a point in the program where a type identifier was expected, a legal type identifier was not found.

INVALID VARIABLE REFERENCE

At a point in the program where a variable identifier was expected, a legal variable identifier was not found.

LABEL ERROR

An illegal label declaration was encountered, or a label was incorrectly specified.

OBJ I/O ERROR

An I/O error was encountered while trying to open or write to the object file.

OUT OF RANGE

The explicit upper bound specified with an array type identifier was outside of the range of the original array declaration. See *Advanced Pascal* for more information.

SKIPPING TO ;

Due to a previous error, the compiler has begun skipping source code until a semicolon is encountered. This message may also indicate that a semicolon is expected at this point in the program.

SOURCE I/O ERROR

An I/O error was encountered while trying to read the current source file.

STRING CONSTANT TOO BIG

This error usually results from an unmatched quote. The maximum length of string constants is 128 bytes. The first quote is 128 bytes before the point of the error.

SYMBOL TABLE FULL

The number of symbols known at this point in the program has completely filled the available symbol table space. You must restructure your program to reduce the number of known symbols at this point in order to get it to compile within the current memory constraints.

SYNTAX ERROR

This is a catch-all error for any syntax error not explicitly covered with another error message.

UNDEFINED SYMBOL

A reference is being made to a symbol which has not been previously defined.

UNEXPECTED END

An END statement was encountered when it was not expected.

UNEXPECTED EOF

End of file on the main source file was reached before the end of the program was reached. This may be caused by a mis-matched BEGIN-END, unmatched (* or an unmatched quote (').

** UNDEFINED **

This message appears in the symbol table listing. A *procedure* was declared as *forward* but was never eventually defined, a *pointer* definition referenced a *type* identifier which was never defined or a *label* was declared but never defined.

WITH ERROR

Either the compiler's maximum *WITH* nesting level (8) was exceeded or the <record variable> portion of the statement was not a record variable or was an element of an *array* or the result of a *pointer* dereference.