```
zeroth          ColorList (elements 1 through 6)
first           ColorList (elements 1 through 6)
second          ColorList (elements 1 through 6)
   .
   .
   .
two hundredth   Colorlist (elements 1 through 6)
```

Alternate (equivalent) forms of multiple dimension ARRAY declarations are as follows:

TYPE ColorPlane = ARRAY[0..200] OF ARRAY[1..6] OF Color;

or

TYPE ColorPlane = ARRAY[0..200, 1..6] OF Color;

Note that there is no limit to the number of dimensions allowed and that each dimension can be of a different ordinal type.

The predefined type *string* is actually an *array[0..80] OF char.* **DEFT Pascal** supports a number of language extensions associated with this *type*. See *Advanced Pascal* for language extensions on both strings and arrays.

5.6 Records

A record is a collection of data of diverse types which are located contiguously in memory in the order in which they appear in the *record*. Each data element or item is referred to as a field. A field may be of any *type*. This means that a record field may be an array, another record, a set, and so on. The general form of a record definition is as follows:

```
RECORD

<field list>

END
```

Where the <field list> has a <fixed part> and/or a <variant part>. The <fixed part> is a group of fields which are declared very much like variables. The following is an example of a RECORD with only a <fixed part>:

```
TYPE Employee = RECORD
                    Name          : String (20);
                    Street, City  : String (20);
                    State         : String (2);
                    ZipCode       : String (5);
                    Number        : Integer;
                    END;
```

In addition to a <fixed part> a RECORD can also have a <variant part> This part describes several *alternative* <field list>s which are located in the *same area of memory.* This allows you to describe the same area of memory in more than one way. The general form of the <variant part> is as follows:

```
CASE [<identifier>:] <type identifier> OF

   <constant>, ... ,<constant> : ( <field list> );
                    .
                    .
                    .
   <constant>, ... ,<constant> : ( <field list> )
```

The <identifier>: following the CASE keyword is optional and if present defines the last *fixed* field in the *record.* The <constant>s must all be of the same *type* as the <type identifier>. Each (<field list>) begins at the same position in the *record.* The size of the *record* will be determined by the size of the largest (<field list>). The following example should make things more obvious:

```
TYPE JobType = (Manager, Worker, Secretary);
     Employee = RECORD

                   (* Fixed Part Starts Here *)

                     Name : String (20);
                     Address: RECORD
                                 Street, City : String (20);
                                 State : String (2);
                                 ZipCode : String (5);
                                 END;
                     Number : Integer;

                   (* Variant Part Starts Here *)

                     CASE EmployeeType : JobType OF

                       Manager : (TotalWorkers : Integer;
                                   SecName : String (20));
                       Worker : (ManagerNbr : Integer;
                                   TotalTools : Integer;
                                   RoomNumber : Integer);
                     END;
```

In this case we have a <variant part> based on the employee's job type. The fields following the *manager* constant describe the information required for a manager. The fields following the *worker* constant describe the information required for a worker. Only one <field set> or the other will be present in any given occurance of an *employee type*.

Note that the size of *Employee* is 21 (Name) + 51 (Address) + 2 (Number) + 1 (EmployeeType) + the size of the largest variant which is the one represented by the *manager* constant (which is 23). Although not shown here, the <field lists> in the <variant part> can themselves have <variant parts>.

5.7 Pointers

A *pointer* is a reference to a specific instance of a *type*. In standard Pascal, this instance is created via the *NEW* procedure. A pointer is basically the memory address of a variable of a specific *type*. You can create a pointer type by preceding any *type* definition with an uparrow (^). The general form of a *pointer type* is:

<type definition>

An example *pointer type* definition is:

```
    TYPE EmployeePtr = ^ Employee;
```

This defines a *type* called employeeptr which is a pointer to a *record type* called *employee.* you can create an instance of *employee* using the *NEW* procedure as follows:

```
    NEW (EmployeePtrVar);
```

This allocates memory for an instance of *employee* and sets the memory address of that instance in the variable called *employeeptrvar* which is of type *employeeptr*.

The size of a *pointer type* is always 2 bytes regardless of the size of the *type* that it is referencing. See *Advanced Pascal* for **DEFT Pascal** extensions on the use of *pointer types.*

5.8 Files

In Pascal, both *files* and *arrays* are lists of elements. With an *array* each element can be randomly accessed. With a *file* each element can be only sequentially accessed. Files are the structured *type* that represent periperal devices such as tape, disk, printer, keyboard and screen.

In Pascal, each element of a *file* can be of any type. File *types* other than *file of char* are used to transfer occurances of the binary image of the *type's* internal representation to and from I/O devices. A *file of char* has special (but standard Pascal) properties which provides for automatic conversion between the internal binary representation of data and the external ASCII representation. A complete explanation can be found in the section on *Input/Output.* The standard predefined type identifier *text* (file of char) can be used in *file* type declarations:

```
    TYPE ThisType = FILE OF Char;
         ThatFile = Text;
```

Both of these declarations define equivalent *type* identifiers. Note that a FILE of a given *type* has a size which is equal to the size of the *type* plus 286 bytes.

5.9 PACKED Types

The reserved word PACKED may precede either *set, array, record* or *file* in a *type* declaration. In standard Pascal, this reserved word indicates that the corresponding structured *type* should be organized to occupy the least possible amount of memory. There are subsequently some restrictions on the use of these *packed types*.

With the **DEFT Pascal** Compiler, the keyword PACKED is allowed but ignored in *set, array, record* and *file* type declarations. This means that the memory requirements don't change and the restrictions are not imposed on the resulting *types*. An example of use is as follows:

```
    TYPE ColorList = PACKED ARRAY[1..6] OF Color;
```

6 Variables

A variable in Pascal represents a specific memory allocation of a *type*. More important is when that memory allocation is made.

6.1 Automatic Allocation

In BASIC, a variable is allocated memory when it is first used. In assembly language a variable is allocated memory when the program is loaded into memory (provided it was declared with an RMB opcode).

In the section on *The Pascal Program* the block structure of Pascal is explained. Constants, types, procedures, functions and variables become known only when the *block* in which they are declared is activated. For variables, this also causes the memory for them to be allocated. When the block is deactivated, not only do the identifiers become unknown but the memory allocated to the variables is deallocated.

The implications of this allocation scheme are two-fold:

1. The value of any variable is undefined when the block is first activated. This is true even if the block was previously activated and deactivated. Variables will *not* assume the value that they had when the block was last deactivated.
2. An active block can activate itself causing a second allocation of its variables. Each concurrent activation of a block therefore has its own independent copy of each variable. This allows for recursive *procedures* and *functions*.

6.2 VAR Declaration

Variables are declared with the *var* statement. The general form of the statement is as follows:

```
VAR <identifier> : <type definition>;
                   .
                   .
                   .
```

For example:

```
VAR   I : Integer;
      ThisEmployee : Employee;
```

7 Procedures and Functions

The concept of a group of statements which perform a given operation is certainly not new to a BASIC programmer. The *gosub* statement allows exactly this type of operation. In Pascal, the *procedure* statement allows a programmer to set aside a group of statements explicitly for this purpose.

In BASIC the concept of a function is provided by the *DEF FN* statement. This statement provides the ability to define single line functions. In Pascal, the *function* statement (which is almost identical to the *procedure* statement) provides a general function definition capability.

The facilities found in Pascal for defining *procedures* and *functions* are very powerful and constitute one of the major characteristics of the Pascal language. As described in the section on *The Pascal Program* Pascal is a block structured language with *procedures* (and *functions)* at the heart of this structure. It is important to read and understand this section in order to use the features of the language to their fullest.

7.1 PROCEDURE Declaration

The *procedure* statement is a declaration statement which provides the ability to construct a complete subprogram which may itself contain subordinate subprograms *(procedures* and *functions).* The general form of the declaration is as follows:

```
PROCEDURE <identifier> <formal parameter definition>;

  <declaration statements>

  BEGIN
  <executable statements>
  END
```

As mentioned in the section on *Block Structure* the *<declaration statements> BEGIN <executable statements> END* constitute a <block> which is *exactly* the same as a *program's* <block>. The <formal parameter definition> can be null if there are no parameters to pass to the procedure or can have the following form if parameters are present:

```
(<parameter>; <parameter>; ... <parameter>)
```

Where the form of <parameter> is:

VAR <identifier>, ... ,<identifier> : <type identifier>

OR

<identifier>, ... ,<identifier> : <type identifier>

The *var* keyword is present when the parameter is a *reference* parameter and is not present when the parameter is a *value* parameter. The difference between these two classes of parameters is important and is discussed in full in the next section on *Procedure Invocation.* Following are some examples:

```
    PROCEDURE TestProc (VAR Parm1 : Integer; Parm2, Parm3 :
Integer);
    BEGIN
      Parm1 := Parm2 + Parm3
    END;

    PROCEDURE TestProc2;
    BEGIN
      IF GlobalVar1 > 0 THEN GlobalVar2 := 5;
      GlobalVar3 := GlobalVar1 + 3
    END;
```

You notice in the first example that *Parm1* is a reference parameter and *Parm2* and *Parm3* are value parameters. In the second example *GlobalVar1, GlobalVar2* and *GlobalVar3* are all variables declared outside the procedure *TestProc2.* See the section on *The Pascal Program* for a discussion of scope.

7.2 Procedure Invocation

Unlike BASIC's *gosub* statement, Pascal has no *call* statement for invoking a procedure. In Pascal, a procedure is invoked by name. That is, a *procedure* declaration implicitly defines a new executable statement which is the procedure name and is formatted according to the <parameter definition> provided in the declaration. The general form of a procedure invocation is:

<identifier> <actual parameters>

If the corresponding <formal parameter definition> in the *procedure* statement was null then the <actual parameters> must also be null. Otherwise the *actual* parameters must agree with the *formal* parameters in ordering, type and number. Some examples:

```
TestProc1 (I, 3, J*5);
TestProc2
```

Before explaining the above examples, it is necessary to define what *reference* and *value* parameters are. A formal *reference* parameter represents the actual *variable* used when the procedure is invoked. The parameter used in the procedure invocation *must* be a variable. In this case, all references to the *formal* parameter (the one in the *procedure* declaration statement) will reference the *actual* parameter (the one in the procedure invocation statement). This means that the actual parameter's value will be changed if the procedure modifies the formal parameter's value.

A formal *value* parameter represents the value of a general expression used when the procedure is invoked. In this case, any type compatible expression is allowed as the *actual* parameter since a separate allocation of memory is made when the procedure is invoked and is initialized to that value. The formal parameter in this case represents its own memory area rather than that of another variable. Changing the formal parameter in this case, does not change the value of any other variable.

In the first example above, *I* is a *reference* parameter and *3* and *J*5* are value parameters. When *TestProc1* is invoked in this case, *I* is assigned the value *3 + J*5*. Since *TestProc2* has no formal parameters, it therefore has no actual parameters.

7.3 FUNCTION Declaration

The *function* statement is almost identical to the *procedure* statement described above. This is because a *function* is a special type of *procedure* which is invoked in a different manner from a regular *procedure* and has a typed value associated with it. The syntax of the *function* statement is as follows:

```
FUNCTION <identifier> <formal parameter definition> :
                        <type identifier>;

  <declaration statements>

  BEGIN
  <executable statements>
  END
```

The only difference between the *function* statement and the *procedure* statement is the beginning keyword (*FUNCTION* instead of *PROCEDURE)* and the presence of the *<type identifier>* following the parameter definition. Following are some examples:

```
    FUNCTION TestFunc (VAR Parm1 : Integer; Parm2, Parm3 :
Integer)
                              : Boolean;
    BEGIN
      Parm1 := Parm2 + Parm3;
      TestFunc := (Parm2 > Parm3)
    END;

    FUNCTION TestFunc2 : Integer;
    BEGIN
      IF GlobalVar1 > 0 THEN GlobalVar2 := 5;
      GlobalVar3 := GlobalVar1 + 3;
      TestFunc2 := GlobalVar3 * 2;
    END;
```

You'll notice that these examples are similar to those used in the *Procedure* section except that there is an extra assignment statement at the end of each *function*. These statements use the *function* name on the left side of the assigment symbol to assign a value to be *returned* by the *function*. Every *function* is required to have at least one assigment statement which performs this task. If more than one assigment takes place, the last assigment made before the *function* terminates is the one that will be used. A function can only be of a simple type.

Pascal

7.4 Function Invocation

A function is invoked by referencing its name (and supplying any required actual parameters) in an expression. In this form the *function* reference is similar to a reference to a variable. Following are some examples:

```
    IF TestFunc (I, 3, J*5) THEN I := 0;
    GlobalVar2 := TestFunc2 * 5
```

Note that for purposes of recursion there is no ambiguity as to whether a function is being recursively invoked or having its returned value set for its current invocation. An *invocation* occurs when the *function's* name (and actual parameter list) are found in an expression. A *function's* returned value is set when its name alone is found on the left side of an assigment statement.

7.5 FORWARD References

In Pascal, a function or procedure may be referenced by another procedure or function only if the function or procedure being referenced has been defined previous to the procedure or function making the reference. There are times when this restriction is undesireable. The *forward* declaration in Pascal solves this little problem.

A forward reference is allowed only if the procedure or function being referenced has been defined using the *forward* declaration. The following is an example:

```
    PROCEDURE TestProc (VAR Parm1 : Integer; Parm2, Parm3 :
Integer);
    FORWARD;

    PROCEDURE TestProc2;
    VAR I, K, M : Integer
    BEGIN
      K := 17; M := 23;
      IF GlobalVar1 < 0 then TestProc (K, M);
      IF GlobalVar1 > 0 THEN GlobalVar2 := 5;
      GlobalVar3 := GlobalVar1 + 3
    END;

    PROCEDURE TestProc;
    BEGIN
      Parm1 := Parm2 + Parm3
      IF Parm1 <> 40 THEN TestProc2
    END;
```

Note that TestProc has been declared as *forward* and is referenced by *TestProc2*, even though *TestProc* is defined after *TestProc2*. The same rules and conventions apply for functions as well.

8 Expressions and Assignments

Expressions are the combination of constants, variables and functions with operators to form some result. This result can then be stored (assigned) in a variable, used as a parameter to a procedure or function, used as a subscript in an array specification, used to control the execution of the program or output to a file.

8.1 Factors

The fundamental elements of an expression are called *factors*. Factors are the constants, variables and functions previously mentioned. Following are some examples of factors:

```
(* Constants *)
2
'A'
'JOES''S PLACE'

(* Variables *)
I
MyColors[1]
OurColors[137,3]
MyRecord.ItsColor

(* Functions *)
CHR (65)
ABS (-3)
```

The *value* of a factor is dependent on what kind of factor that it is. A constant has a single given value that is always used whenever that constant is referenced.

A variable's value will be potentially different each time that it is referenced. The last value that was stored (assigned) to that variable before a given reference will be the value of that variable for that reference.

In the example above you can see a reference to an *array* type variable. The value contained in the square brackets ([]) (which can be a full expression) is called a *subscript* and identifies which element of the array is being referenced. Note that every element of an array is considered to be an independent variable. When an array has more than one dimension, the subscripts are ordered according to the *type* definition for that array and are separated from each other by commas.

Pascal

A reference to a field within a *record* is also a factor. This is done by naming the record, appending a period (.) and then naming the field. If the *record* is an element of an *array*, then the period follows the right bracket. For Example:

```
ArrayOfRec[i].Field1
Record1.ArrayField[i].SubField1
```

Notice that *arrays* of *records* and *records* of *arrays* can be referenced by following the above rules.

A reference to a function will actually cause the function to be invoked at the point of reference. The value returned by that invocation will be the function's value for that reference.

Another type of factor is the *inline set:*

```
(* In-Line Sets *)
[Green..Blue, Yellow]
['0'..'9', 'A'..'Z']
[1, 5, 7, I..50]
```

An inline set is a *set* value that is built from a list of itemized ordinal expressions and subranges as shown above. Note that an inline set must always be preceeded in an expression with some indication as to what type it should assume. Therefore, it cannot be used as the first factor in a boolean expression.

A final type of factor is a *dereferenced pointer*. This is a reference to a variable whose address is in a *pointer type* variable and can be made by naming the *pointer* variable and following it with an up-arrow (^). The same syntax is used to reference the window of a *file type* variable. For example:

```
PtrVar
FileVar
```

8.2 Arithmetic Operators

An expression does not have to have any operators so that a single factor can be considered to be a full expression. However, frequently we wish to combine one or more *integer* or *real type* factors arithmetically. This is done with the use of the following operators:

+	**Addition**
-	**Subtraction**
*	**Multiplication**
/	**Real Division**
DIV	**Integer Division - quotient result**
MOD	**Integer Division - remainder result**

In addition to the above standard arithmetic operators, the **DEFT Pascal** Compiler also provides the following additional arithmetic operators:

AND	**Bitwise logical AND**
OR	**Bitwise logical inclusive OR**
XOR	**Bitwise logical exclusive OR**
LSR	**Bitwise shift right (zero fill)**
LSL	**Bitwise shift left (zero fill)**

Some examples of simple arithmetic expressions are as follows:

```
I + R         (* sum of I and R, real result *)
2 * 3         (* product of 2 and 3 *)
J / 6         (* real quotient of J divided by 6 *)
J DIV 6       (* integer quotient of J divided by 6 *)
I AND $1FF    (* value of I with high 7 bits cleared *)
J LSL 3       (* value of J shifted left 3 bit positions *)
```

8.3 Integer/Real Expressions

All the above operators (except the slash) can be used with *integer types* to create *integer* type expressions. The plus (+), minus (-), asterisk (*) and slash (/) can also be used with *real* types to create *real* type expressions.

You can also include *integer* types in *real* expressions and **DEFT Pascal** will automatically convert the *integers* to *reals*. However, you must use either the *TRUNC* or *ROUND* built-in functions to convert from *real* to *integer*. These are described in the section on *Built-In Procedures and Functions*. Following are some examples of expressions mixing *integers* and *reals:*

```
R := 1;                  (* legal *)
I := 1.0;                (* illegal *)
R := I + R;              (* legal *)
IF R + I = 0 THEN ...    (* legal *)
IF I + R = 0 THEN ...    (* illegal *)
```

In **DEFT Pascal** the last expression is illegal because the expression started out as integer before the *R* was encountered. In standard Pascal, this would be a legal expression.

8.4 Arithmetic Precedence

In the above examples we saw how two factors could be combined with an arithmetic operator. In general, there is no limit to the number of factors that can be combined in a single expression. For example:

```
    I * J + 5 DIV 3 OR $FF00
```

The above example is a legal expression. Unfortunately it is not immediately clear how it might be evaluated. This is because it is not clear which order the operations are performed in. In Pascal, as in most languages, this is resolved via rules of *precedence*. For arithmetic expressions the operators are divided into two categories: *multiplying* operators and *addition* operators as shown below:

Multiplying Operators:	*** / DIV MOD AND XOR LSR LSL**
Addition Operators:	**+ - OR**

Expressions are generally evaluated from left to right with the multiplying operations performed before the addition operations. In the example above, the evaluation would occur in the following order:

```
  I * J                  (* result 1 *)
  5 DIV 3                (* result 2 *)
  result 1 + result 2    (* result 3 *)
  result 3 OR $FF00      (* final result *)
```

Parentheses can be used to change this *default* order of operations. In fact, the above expression, although legal, is generally considered poor programming practice since it is not immediately clear how the expression is to be evaluated. *All* operations (both multiplying and addition) within a set of parentheses are performed before the result is combined with operators outside the parentheses. By inserting parentheses in the above example we can change order of evaluation as follows:

```
    I * (J + 5) DIV (3 OR $FF00)
```

The parentheses have changed the order of evaluation to the following:

```
J + 5                      (* result 1 *)
I * result 1               (* result 2 *)
3 OR $FF00                 (* result 3 *)
result 2 DIV result 3      (* final result *)
```

Note in the above example that the * operation takes place before the OR operation. That is due to the left-right nature of the expression evaluation. Note that parentheses may be nested to form even a different evaluation as follows:

```
I * ((J + 5) DIV (3 OR $FF00))
```

The new parentheses have changed the order of evaluation to the following:

```
J + 5                      (* result 1 *)
3 OR $FF00                 (* result 2 *)
result 1 DIV result 2      (* result 3 *)
I * result 3               (* final result *)
```

Note that an expression inside a set of parentheses is actually considered a *factor* and is treated as such in all expressions.

8.5 Set Expressions

Set factors can be combined into expressions with the following operators:

+	**Union**
-	**Difference**
*	**Intersection**

As in arithmetic expressions, two set factors are combined with a single operator to produce a single set result. The above operators produce the following results:

- The Union of two sets produces a set which contains all the elements present in either the first or second set.
- The Difference of two sets produces a set which contains all the elements of the first set which are not also in the second set.
- The Intersection of two sets produces a set which contains only those elements which are in both the first and second sets.

Intersection has precedence over Union and Difference.

8.6 Boolean Expressions

A *Boolean* expression has a *true* or *false* boolean result (this is actually an 8 bit result). As in arithmetic and set expressions, boolean expressions are formed with factors and operators. The boolean operators are as follows:

NOT	**Logical NOT**	**(* Unary *)**
AND	**Logical AND**	**(* Multiplying *)**
OR	**Logical OR**	**(* Addition *)**
IN	**Set Membership**	
=	**Equals**	**(* Relational *)**
>	**Greater than**	
<	**Less than**	
>=	**Greater than or Equal**	**(* Simple Types *)**
	Containment	**(* Set Types *)**
<=	**Less than or Equal**	**(* Simple Types *)**
	Inclusion	**(* Set Types *)**
<>	**Not Equal**	

Unlike arithmetic and set expressions, boolean expressions can take *any* type factor as an argument. The only restriction is that they be combined with relational operators and that the types of both factors are the same. The *not*, *and* and *or* logical operators require boolean type factors (in order to produce a boolean result). For example:

```
BoolVar1 AND BoolVar2
Integer1 = Integer2
MyColor1 > MyColor2
```

The *in* operator is used to determine whether a given ordinal value in the range 0..255 is contained within a set of the same ordinal type. For example:

```
MyChar IN ['A'..'Z']
```

The <= and >= operators have a special meaning when applied to sets.

- Set Containment (>=) produces a *true* result if all the elements of the second set are also elements of the first set.
- Set Inclusion (<=) produces a *true* result if all the elements of the first set are also elements of the second set.

Precedence in boolean expressions is about the same as in arithmetic expressions with the following addition: after all multiplying and addition operations have been performed, a single relational or set membership operation may be performed. Note that as in arithmetic expressions, parentheses can be used to alter the order of evaluation and to break the expression down into a number of factors. The following examples illustrate this:

```
 J = I AND K <= L           (* illegal expression *)
(J = I) AND (K <= L)        (* legal expression   *)
```

Where *I*, *J*, *K* and *L* are all integer variables. The following evaluation takes place:

```
J = I                       (* boolean result 1 *)
K <= L                      (* boolean result 2 *)
result 1 AND result 2       (* final boolean result *)
```

In the following example, changing parentheses changes not only the order, but also the required intermediate expression types:

```
J = I AND (L <= K)
```

The above expression is illegal unless I and J are boolean type factors. Evaluation is as follows:

```
L <= K              (* boolean result 1 *)
I AND result 1      (* boolean result 2 *)
J - result 2        (* final boolean result *)
```

Not only factors, but arithmetic, set and boolean expressions may be combined via relational operators as follows:

```
I*3 >= J+2
Set1 <= Set2 + Set3
(I IN [5, 6, 20..30]) = OnOffVar
(L+2)*I >= K AND $1F0
```

In the last example, the AND operator is an arithmetic operator rather than a boolean operator.

8.7 Assignment Statement

This statement is similar to that found in BASIC. The symbol of assignment is different than BASIC's to distinguish it from the equals sign. The general form is as follows:

```
<identifier> := <expression>
```

The <identifer> on the left must be a variable whose value is to be set to that of the expression on the right *after* the expression is evaluated. Following are some examples:

```
I := I * ((J + 5) DIV (3 OR $FF00))
BoolVar1 := I = J
```

In the second example, BoolVar is assigned either a True or False value depending on whether I is equal to J.

9 Compound and Control Statements

Statement execution normally starts with the statement immediately following the *BEGIN* keyword in the main program block. Execution proceeds sequentially with each subsequent statement until the *END* at the end of the main program block is reached. If any other blocks are activated in the interim, execution within that block proceeds in a similar fashion.

This section primarily describes the statements that allow you to alter this general flow of execution.

9.1 BEGIN Statement

This statement allows a programmer to include more than one statement in a place in the program where normally only one statement would be allowed. This statement does not cause any change in the order of statement execution but is frequently used in conjunction with the control statements described below which do. The following is the general form of the *BEGIN* statement:

```
BEGIN
   <executable statement> ;
                .
                .
                .
   <executable statement>
END
```

Note that the semi-colon is used to separate rather than terminate statements. Since the **DEFT Pascal** Compiler supports a *null* statement, you can put a semi-colon after the last executable statement before the *END*.

9.2 IF Statement

The *IF* statement provides the capability to execute either one of two statements based on the value of a boolean expression. Following is the general form of an *IF* statement:

```
IF <boolean expression> THEN <executable statement>
                        ELSE <executable statement>
```

If the boolean expression is *true* then the <executable statement> following the *THEN* keyword is executed otherwise the <executable statement> following the *ELSE* is executed. The *else* clause is optional and if it is not present, no statement is explicitly executed