# Waterloo microPascal ©

commodore
COMPUTER

# Waterloo microPascal

# Tutorial and Reference Manual

F. D. Boswell

T. R. Grove

J. W. Welch

**Disclaimer**

# Preface

Pascal was originally developed in the late 1960's by Niklaus Wirth at ETH in Zurich Switzerland. In the 1970's it became a widely respected programming language, particularly for the teaching of Computer Science.

This document provides a tutorial and a reference manual for the Pascal language.

The Tutorial is intended to provide a quick introduction to the language. The serious user may wish to acquire one of the many Pascal textbooks available.

The Reference Manual is intended to be a concise definition of the language. It is based on the draft proposals produced by the Pascal standardization effort. The language is quite similar to what is described by Jensen and Wirth in Pascal User Manual and Report, Second Edition (Springer-Verlag 1974).

All members of the Computer Systems Group have made a significant contribution to the Waterloo microPascal interpreter. The design is based upon ideas evolved and proven over the past decade in other compiler projects in which the group has been involved. The actual design and programming of the processor was primarily performed by F. D. Boswell and T. R. Grove. Sharon Malleck assisted in the production of the manual.

This document was typeset in 10-point Times using the Waterloo SCRIPT text formatter and a Mergenthaler VIP photo typesetter.

F. D. Boswell
T. R. Grove
J. W. Welch
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
June 1981

Table of Contents

# Waterloo Computing Systems Newsletter

The software described in this manual was implemented by Waterloo Computing Systems Limited. From time-to-time enhancements to this system or completely new systems will become available.

A newsletter is published periodically to inform users of recent developments in Waterloo software. This publication is the most direct means of communicating up-to-date information to the various user. Details regarding subscriptions to this newsletter may be obtained by writing:

**Waterloo Computing Systems Newsletter**
**Box 943,**
**Waterloo, Ontario, Canada**
**N2J 4C3**

# Introduction

Waterloo microPascal is an interpretive implementation of the Pascal language. It is accompanied by Waterloo microEdit--a full-screen text editor. Waterloo microEdit is used to create and maintain both program source files and data files. This manual assumes familiarity with microEdit. A description of the editor may be found in a separate manual.

This document consists of two sections: a tutorial introduction and a reference manual. The tutorial introduction introduces the features of the Pascal language by a series of simple examples accompanied by notes. The reference manual defines the Pascal language and also explains specific features of Waterloo microPascal.

The remainder of this section is an overview of Waterloo microPascal.

### Language Supported

There is no offical standard for the Pascal programming language. The Waterloo microPascal implementation corresponds closely to *Pascal User Manual and Report, Second Edition* (Springer-Verlag, 1974) and the interim draft standards being produced by the international standardization effort.

### Enhancements and Features

- An interactive debugger allows single-step operation, breakpoints and interactive examination of variables at execution-time.

- Peek and poke procedures allow direct access to the user memory, including the screen.

- Reset and rewrite allow the specification of an actual filename as their second parameter.

- Lazy I/O is a feature permitting keyboard and screen I/O to behave in an intuitive way for interactive programs.

### Restrictions

- Sets may contain a maximum of 256 elements. The ordinal values of the elements of the base type of the set must be in the range 0..255.
- Pack and unpack are unimplemented.
- Variant record semantics are not checked.
- Passing procedure or function names as parameters is not supported.

# A Quick Tutorial Introduction to Pascal

## EXAMPLE 1 A First Program

This program writes a message on the screen.

```
(* this is our first Pascal example *)

program example1( output );
  begin
    writeln( 'This is my first Pascal program' );
  end.
```

Notes:

1. The first line in the program is called a *comment*, and may be recognized by the "(*" and "*)" characters. Comments have no effect whatsoever on the execution of a program; they are used as *documentation*.

2. Pascal programs consist of three *sections*: the *program heading*, the *declarations*, and the *program body*. The program heading gives a name to the program, and says that the program will produce some output. This program is too simple to have a declaration section (the next example will have one). The program body consists of the *keyword* "begin", followed by some *executable statements*, followed by the keyword "end", followed by a period ("."). The executable statements are separated from each other by a semi-colon (";").

3. The appearance of a Pascal program (spacing, indentation, blank lines etc.) is immaterial to the execution of a program, but is very important from a *programming style* point of view.

4. The "writeln" ("write a line") statement here outputs a *character string constant*. It will appear exactly as it appears in the program. A character string consists of a sequence of characters enclosed in "'" characters.

**EXAMPLE 2 Variables and Arithmetic**

Our second example declares two integer variables, performs some simple arithmetic, and outputs the results of that computation.

```
program example2( output );
  var
    x, xsquared : integer;

  begin
    x := 12;
    xsquared := x * x;
    writeln( x, xsquared );
  end.
```

Notes:

1. This program has a declaration section as follows:

   ```
   var
     x, xsquared : integer;
   ```

   Two variables, named "x" and "xsquared", are declared to be of *type* integer. This means that the range of possible values that these variables may have is restricted to the integers (..., −3, −2, −1, 0, 1, 2, 3, ...).

2. Variables names (or variable *identifiers*) start with a letter, but then may consist of any number of letters and numbers.

3. The ":=" that appears in the program is the *assignment operator*. It says that the variable on the left is assigned the value of the *expression* on the right.

4. "x*x" is an *arithmetic expression,* and "*" is the multiplication operator. Other arithmetic operators are:

   | | |
   |---|---|
   | + | addition |
   | − | subtraction |
   | div | integer division |
   | mod | integer remainder |
   | / | real division |

5. Arithmetic expressions are evaluated according to the usual rules of algebra.

6. Variables must be assigned a value before they may be used in an expression. Try removing the statement "x := 12;" from the program and then re-running it.

**EXAMPLE 3 Loops (the For Statement)**

The computations from Example 2 are placed in a loop, producing a table of squares.

```
program example3( output );
  var
    x, xsquared : integer;

  begin
    for x := 12 to 21 do
      begin
        xsquared := x*x;
        writeln( x, xsquared );
      end;
  end.
```

Notes:

1. A *for* statement is used to perform the looping.

2. A for statement may execute only one statement repeatedly (it is called the *object statement,* or the *for loop object*).

3. Because we need to repeat two statements (the assignment and the writeln), a *compound statement* is used. A compound statement is a sequence of statements enclosed by a begin-end pair.

4. The "x := 12" in the for statement is just like an assignment statement; "x" is referred to as the *for statement index,* and "12" is the *initial value.*

5. Each time the for repeats, the value 1 is added to the for index. This continues until the index is equal to the *final value* ("21" in this example).

6. After the for statement is finished executing, the value of the for index is undefined.

**EXAMPLE 4 More Loops (the While Statement)**

Example 4 produces the same output as Example 3. A while statement is used instead of a for statement.

```
program example4( output );
  var
    x, xsquared : integer;

  begin
    x := 12;
    while( x <= 21 )do
      begin
        xsquared := x*x;
        writeln( x, xsquared );
        x := x + 1;
      end;
  end.
```

Notes:

1. The *while* statement is another method by which looping may be done. Like the for statement, a while statement repeats only a single statement, so a compound statement is usually used.

2. The "x <= 21" is called a *relational expression,* and the "<=" ("less than or equal to") is a *relational operator*. The value of a relational expression is either *true* or *false*. True and false are called *Boolean constants.*

3. A while statement will repeat as long as the value of the relational expression is true.

4. The statement "x := x + 1;" causes the value of variable x to increase by 1 each time through the loop. X is said to be *incremented* by 1.

5. Some other relational operators are:

| | |
|---|---|
| <> | not equal to |
| = | equal to |
| > | greater than |
| >= | greater than or equal to |
| < | less than |

**EXAMPLE 5 While vs For**

A table of the squares and cubes of even numbers from 12 to 21 is output with a title.

```
program example5( output );
  var
    x, xsquared, xcubed : integer;

  begin
    writeln( 'A table of squares and cubes:' );
    x := 12;
    while( x <= 21 )do
      begin
        xsquared := x*x;
        xcubed := x*x*x;
        writeln( x, xsquared, xcubed );
        x := x + 2;
      end;
  end.
```

Notes:

1. An immediate advantage of the while statement over the for statement may be seen in the example. The problem requires us to increment x by 2 each time through the loop. The for statement, however, will not allow this. With a while statement we are free to choose any increment necessary.

2. The statement "xcubed := x*x*x;" could have been written as "xcubed := xsquared*x;".

**EXAMPLE 6 Column Titles**

The output from this example is the same as Example 5, except that titles are output above each column of numbers.

```
program example6( output );
  var
    x, xsquared, xcubed : integer;

  begin
    writeln( 'A table of squares and cubes:' );
    writeln( 'X':7, 'X**2':7, 'X**3':7 );
    x := 12;
    while( x <= 21 )do
      begin
        xsquared := x*x;
        xcubed := x*x*x;
        writeln( x, xsquared, xcubed );
        x := x + 2;
      end;
  end.
```

Notes:

1. If you examine the output from Example 5, you will see that the numbers are aligned in *zones* that are seven characters wide.

2. The new writeln statement outputs three character string constants: "X", "X**2" and "X**3". The ":7" to the right of each string is called a *field width modifier*. It tells Pascal that the string is to be output in a seven-character zone, *right-justified* with blanks on the left. This will cause the titles to appear directly above their respective columns.

3. The next example will show an easy way to create columns of any width.

**EXAMPLE 7 Variable-width Columns**

Once again, the output from this example is similar to that of Example 5. A programming technique is introduced that allows the columns of output to be any width.

```
program example7( output );
  const
    width = 10;
  var
    x : integer;

  begin
    writeln( 'A table of squares and cubes:' );
    writeln( 'X':width, 'X**2':width, 'X**3':width );
    x := 12;
    while( x <= 21 )do
      begin
        writeln( x:width, x*x:width, x*x*x:width );
        x := x + 2;
      end;
  end.
```

Notes:

1. A new kind of declaration, a *constant declaration,* appears in the program. Wherever the *constant identifier* "width" appears in the program, the number 10 will be used instead.

2. The constant declarations must be located between the program heading and the variable declarations.

3. All the relevant writeln statements use the constant as a field width modifier, so that changing the column width is a simple matter of changing the program in one place (the declaration of "width").

4. A variable could have been used instead of a constant (assuming that the variable was assigned a value).

5. This program does not have variables for x-squared and x-cubed. Instead, the values are calculated directly in the writeln statement. In general, Pascal will allow any arithmetic expression in a writeln statement.

**EXAMPLE 8 The Real Type**

The Pascal type real is introduced with a program that produces a table of square roots from 1 to 15.

```
program example8( output );
  var
    x : integer;
    rootofx : real;

  begin
    writeln( 'A table of square roots:' );
    x := 1;
    while( x <= 15 )do
      begin
        rootofx := sqrt( x );
        writeln( x, rootofx );
        x := x + 1;
      end;
  end.
```

Notes:

1. A new type, *real,* is used in the declaration of variable "rootofx".

2. "sqrt" is a *built-in function* that calculates the square root of its *parameter* (the value in parentheses), provided that the value of that parameter is not negative.

3. The type of the parameter to "sqrt" may be integer or real, but the result is always real.

4. There are many other built-in functions available in Pascal including sine and cosine, for example.

**EXAMPLE 9 More Real Numbers**

This example produces a table of sines and cosines for x ranging from pi/2 to pi radians, in increments of 0.1 radians.

```
program example9( output );
  const
    width = 15;
    pi = 3.1415926;
  var
    x, sineofx, cosineofx : real;

  begin
    writeln( 'A table of sines and cosines:' );
    writeln( 'X':width, 'Sin(x)':width, 'Cos(x)':width );
    x := pi/2;
    while( x <= pi )do
      begin
        sineofx := sin( x );
        cosineofx := cos( x );
        writeln( x:width, sineofx:width, cosineofx:width );
        x := x + 0.1;
      end;
  end.
```

Notes:

1. The value of "pi" is declared to be a real constant with the value of 3.1415926 .

**EXAMPLE 10 Input from the Keyboard**

An integer number is read from the keyboard, and its square and square root are output.

```
program example10( input, output );
  var
    x, xsquared : integer;
    rootofx : real;

  begin
    writeln( 'Enter an integer:' );
    readln( x );
    rootofx := sqrt( x );
    xsquared := x*x;
    writeln( x, 'xsquared =', xsquared,
      '; sqrt(', x, ') =', rootofx );
  end.
```

Notes:

1 The keyword *input* in the program heading indicates that the program will be reading from the keyboard.

2. The first writeln statement outputs a *prompt*. The purpose of this is to remind you to enter a number.

3. The readln statement reads a number from the input, and then assigns that number to "x".

4. Since the square root of a negative number is undefined (for the real numbers, at least), you will get odd results if you enter a negative number. This defect in the program will be corrected in a later example.

## EXAMPLE 11 Reading And Loops

This example places the computations of Example 10 into a while loop. The loop stops when a value of −999 is input.

```
program example11( input, output );
  var
    x, xsquared : integer;
    rootofx : real;

  begin
    writeln( 'Enter an integer:' );
    readln( x );
    while( x <> -999 )do
      begin
        rootofx := sqrt( x );
        xsquared := x*x;
        writeln( x, 'squared =', xsquared,
          '; sqrt(', x, ') =', rootofx );
        writeln( 'Enter an integer:' );
        readln( x );
      end;
  end.
```

Notes:

1. The while statement uses the "<>" ("not equal") relational operator.

2. The program contains two pairs of identical lines (the prompt, and the readln statement). In the next example, we will see a way to avoid this repetition.

**EXAMPLE 12 Procedures**

This example produces the same output as Example 11. A procedure is used to do the prompting and reading.

```
program example12( input, output );
  var
    x, xsquared : integer;
    rootofx : real;

  procedure getnumber;
    begin
      writeln( 'Enter an integer:' );
      readln( x );
    end;

  begin
    getnumber;
    while( x <> -999 )do
      begin
        rootofx := sqrt( x );
        xsquared := x*x;
        writeln( x, ' squared =', xsquared,
          '; sqrt(', x, ') =', rootofx );
        getnumber;
      end;
  end.
```

Notes:

1. A *procedure* by the name "getnumber" is defined in the declaration section of this program. A procedure is very similar in structure to a program: it consists of a procedure heading ("procedure getnumber"), a declaration section (this procedure doesn't have one) and a procedure body (the four lines following the heading).

2. Procedure declarations occur between the variable declarations and the body of the program.

3. The procedure body ends with a ";", whereas the program body ends with a ".".

4. The purpose of a procedure is to isolate a group of statements that performs a specific function. This is often referred to as *program modularization.*

5. Procedures are used by having a statement which consists of nothing but the procedure name. Whenever such a statement is encountered, the procedure is *invoked,* and all of the statements in the procedure body are executed. When the end of the procedure body is reached, execution resumes at the statement following the invocation statement.

**EXAMPLE 13 Boolean Variables and If Statements**

The "negative square root" problem from the preceding examples is corrected. Also, a more elegant way to stop the program is shown.

```
program example13( input, output );
  const
    endingvalue = -999;
  var
    x, xsquared : integer;
    rootofx : real;
    done : boolean;

  procedure getnumber;
    begin
      writeln( 'Enter an integer:' );
      readln( x );
      done := (x = endingvalue);
    end;

  begin
    getnumber;
    while( not done )do
      begin
        xsquared := x*x;
        write( x, ' squared =', xsquared,
          '; sqrt(', x, ') =' );
        if( x >= 0 )then
          begin
            rootofx := sqrt( x );
            writeln( rootofx );
          end
        else
          begin
            writeln( ' undefined' );
          end;
        getnumber;
      end;
  end.
```

Notes:

1. A new kind of type is used in the declaration of variable "done": it is the *Boolean* type. A Boolean variable may have either the value *true* or the value *false*. Note that true and false are *not* strings; they are *Boolean constants,* in the same sense that 27 and −3 are integer constants, for example.

2. The statement "done := (x = endingvalue)" in procedure getnumber may be interpreted as follows:

   - the relational expression "(x = endingvalue)" is evaluated, and gives the value true or false (the "=" is the "equal to" relational operator).

   - the resulting value is assigned to the Boolean variable "done".

3. The while statement in the program body uses a new kind of relational expression. As mentioned above, the variable "done" will have a value of either true or false; "not" is a *Boolean operator* that *inverts* the value (not false is true, and not true is false). Thus, the resulting value will still be either true or false, and the while statement works the same way as before.

4. A new statement, *write,* is used. It is very similar to writeln, the difference being that subsequent write or writeln statements will put their output on the same line. For example, all of the following groups of lines produce the same output:

```
write( 'a' );        write( 'a' );        writeln( 'abc' );
write( 'b' );        writeln( 'bc' );
writeln( 'c' );
```

   They all produce a line:

```
abc
```

5. Another new statement is the *if* statement. It is used to select between two alternatives. There are two forms of an if statement. The first form executes the first object statement (the "then part") if the value of the relational expression is true, and executes the second object statement (the "else part") if the value of the relational expression is false. In the second form, the "else part" is optional. In this case the "then part" is executed if the value of the relational expression is true, otherwise execution proceeds at the statement following the if.

6. As in the for and while statements, the object statement of an if statement may be a compound statement.

**EXAMPLE 14 A Loop Within a Loop**

This example summarizes many of the ideas presented so far. The program produces a set of tables of squares and square roots.

```
program example14( input, output );
  const
    width = 15;
    endingvalue = -999;
  var
    x, xvary : integer;
    loopcounter, tablelength : integer;
    done : boolean;

  procedure getstartingx;
    begin
      writeln( 'Enter the table starting value:' );
      readln( x );
      done := (x = endingvalue);
    end;

  begin
    getstartingx;
    while( not done )do
      begin
        writeln(
          'Enter the increment for x, and the table length:' );
(*1*)   readln( xvary, tablelength );
        writeln( 'X':width, 'X*X':width, 'Sqrt(X)':width );
        for loopcounter := 1 to tablelength do
          begin
(*2*)       write( x:width, (x*x):width );
            if( x >= 0 )then
(*2*)         writeln( (sqrt(x)):width )
            else
              writeln( 'undefined':width );
            x := x + xvary;
          end;
        writeln( 'End of table' );
(*3*)   writeln;
        getstartingx;
      end;
  end.
```

Notes:

1. This example uses a "loop within a loop". The *inside loop* is the for statement, and the *outside loop* is the while statement. The outside loop is said to include or enclose the inner loop, and the inner loop is sometimes referred to as a *nested* loop.

2. The readln statement indicated by (*1*) inputs two numbers. They may be entered on the same line (with a blank in between), or on separate lines.

3. The lines in the program indicated by (*2*) show that it is possible to use a length modifier on any expression value, and not just a variable.

4. The writeln statement indicated by (*3*) simply outputs a blank line.

**EXAMPLE 15 Output Formatting**

The field width modifiers used so far have all been constants. They may also be variables and expressions. Example 15 demonstrates this feature by drawing a triangle.

```
program example15( output );
  const
    width = 21;
  var
    leftside, middle, i : integer;

  begin
    leftside := width div 2;
    middle := leftside + 1;
    writeln( '*':middle );
    for i := middle+1 to width-1 do
      begin
        writeln( '*':leftside, '*':(i-leftside) );
        leftside := leftside - 1;
      end;
    for i := 1 to width do
      write( '*' );
    writeln;
  end.
```

Notes:

1. The object statement of the second for statement is a single statement, so no "begin-end" pair is required.

2. The last writeln statement is needed to "finish off" the output line created by the preceding for statement.

3. "div" is an arithmetic operator that performs an integer division (i.e. any fraction is thrown away). The resulting value is of type integer, and *both* of the operands must be of type integer.

### EXAMPLE 16 Subranges of Integers

The Pascal construct "subrange of integer" is introduced.

```
program example16( input, output );
  var
    thisdecade : 1980..1989;
    hour : 0..23;
    minute, second : 0..59;

  begin
    while( true )do
      begin
        writeln( 'What year is it?' );
        readln( thisdecade );
        writeln( 'What time is it (hh mm ss)?' );
        readln( hour, minute, second );
      end;
  end.
```

Notes:

1. All of the variables in this program are declared with a new kind of type: a *subrange of integer*. A subrange declaration tells Pascal that the variables may be assigned only the values specified by the subrange. For example, the variable "thisdecade" may have only the integer values 1980, 1981, 1982, ..., 1989. Any attempt to give subrange variables a value outside their declared range will result in an error.

2. The program executes an *infinite loop* (the relational expression has the constant value of true, so it never stops), so that you may try entering various values. When you want to stop, simply enter a value outside the range of the variable that is being prompted, and Pascal will give an error message.

**EXAMPLE 17 User-defined Types**

This example does the same thing as the previous example. The declarations of the variables are made with user-defined types.

```
program example17( input, output );
  type
    eighties = 1980..1989;
    validhours = 0..23;
    minorsec = 0..59;
  var
    thisdecade : eighties;
    hour : validhours;
    minute, second : minorsec;

  begin
    while( true )do
      begin
        writeln( 'What year is it?' );
        readln( thisdecade );
        writeln( 'What time is it (hh mm ss)?' );
        readln( hour, minute, second );
      end;
  end.
```

Notes:

1. There is a new kind of declaration in the program, namely a *type declaration*. A type declaration is used to give a name to some collection of values, in the same sense that "boolean" is the name for the collection of values "true" and "false". For example, the declaration for type "validhours" says that this type consists of the integer values in the integer subrange 0..23 (i.e. the values 0, 1, 2, ..., 23). Once a type has been declared, it may be used in a variable declaration just like real, integer etc. .

2. Type declarations occur between the constant declarations and the variable declarations.

3. The name of a type is called the *type identifier*.

**EXAMPLE 18 Arrays**

This program inputs 5 integers, and uses an array to store them. The list is then output in reverse order.

```
program example18( input, output );
  const
    lower = 1;
    upper = 5;
  type
    bounds = lower..upper;
  var
    index : bounds;
    list : array[ bounds ] of integer;

  begin
    for index := lower to upper do
      begin
        writeln( 'Enter an integer:' );
        readln( list[ index ] );
      end;
    writeln( 'The list of numbers (backwards) is:' );
    for index := upper downto lower do
      writeln( list[ index ] );
  end.
```

Notes:

1. The variable "list" is declared to be an *array of integers*. The number of elements in the array is specified by the subrange in the square brackets (in this case, the subrange of integer "bounds"), so that "list" has 5 elements: list[1], list[2], ..., list[5].

2. "list" also could have been declared as

```
var
  list : array[ 1..5 ] of integer;
```

but the method used is preferable from a programming style point of view.

3. Any subrange may be used to define the size of an array, for example:

   list : array[ 10..15 ] of integer

   would be a six-element array (list[10], list[11], ..., list[15]), and

   list : array[ −5..5 ] of integer

   would be an *eleven-element* array (list[−5], list[−4], ..., list[0], list[1], ..., list[5]).

4. A new kind of for statement is used. It uses the keyword "downto" instead of "to". Instead of being incremented by 1 each time, the for statement index is *decremented* by 1 each time. The loop ends when the index is equal to the final value.

**EXAMPLE 19 Two-dimensional Arrays**

The program prompts for "rows" of integers. A two-dimensional matrix is constructed from the input, and then displayed.

```
program example19( input, output );
  const
    width = 10;
    rowmin = 1;
    rowmax = 5;
    firstonrow = 1;
    lastonrow = 5;
  type
    numberofrows = rowmin..rowmax;
    rowsize = firstonrow..lastonrow;
    rows = array[ rowsize ] of integer;
  var
    rownumber : numberofrows;
    rowindex : rowsize;
    matrix : array[ numberofrows ] of rows;

  begin
    for rownumber := rowmin to rowmax do
      begin
        writeln( 'Enter a row of', lastonrow:3,
          ' numbers:' );
(*1*)   for rowindex := firstonrow to lastonrow do
          read( matrix[ rownumber ][ rowindex ] );
        readln;
      end;
    writeln;
    writeln( 'The complete matrix is:' );
    for rownumber := rowmin to rowmax do
      begin
        for rowindex := firstonrow to lastonrow do
          write( matrix[ rownumber ][ rowindex ]:width );
        writeln;
      end;
  end.
```

Notes:

1. The variable matrix is declared to be an "array of arrays"; each element of the first (or "outer") array is itself an array (the "inner" array).

2. The elements of "matrix" are referred to by specifying the outer array subscript followed by the inner array subscript: matrix[3][5] for example.

3. The entire array is referenced as follows:

| | | | |
|---|---|---|---|
| matrix[1][1] | matrix[1][2] | ... | matrix[1][5] |
| matrix[2][1] | matrix[2][2] | | . |
| . | | | . |
| . | | | . |
| matrix[5][1] | ... | ... | matrix[5][5] |

4. This method of *nesting* arrays may be used to create matrices of any dimension. For example, an "array of array of array" would be a three-dimensional array.

5. The for statement indicated by (*1*) has a *read* statement. Read is like readln, except that subsequent read's followed by a final readln will get their input from the same line. The for statement is followed by a readln so that the next row may be read from a new input line (after the prompt).

## EXAMPLE 20 User-defined Functions

A two-dimensional matrix is created as in the previous example. A user-defined function is declared and used to compute the largest element in each row of the matrix.

```
program example20( input, output );
  const
    rowmin = 1;
    rowmax = 5;
    firstonrow = 1;
    lastonrow = 5;
  type
    numberofrows = rowmin..rowmax;
    rowsize = firstonrow..lastonrow;
    rows = array[ rowsize ] of integer;
    matrixshape = array[ numberofrows ] of rows;
  var
    rownumber : numberofrows;
    rowindex : rowsize;
    matrix : matrixshape;

  function maxrowelement( thisrow : rows ) : integer;
    var
      max : integer;
      index : rowsize;
    begin
      max := -maxint;
      for index := firstonrow to lastonrow do
        if( thisrow[ index ] > max )then
          max := thisrow[ index ];
      maxrowelement := max;
    end;

begin
  for rownumber := rowmin to rowmax do
    begin
      writeln( 'Enter a row of', lastonrow:3,
        ' numbers:' );
      for rowindex := firstonrow to lastonrow do
        read( matrix[ rownumber ][ rowindex ] );
      readln;
    end;
```

```
        writeln;
        for rownumber := rowmin to rowmax do
           writeln( 'The maximum element in row', rownumber:3,
              ' is', maxrowelement( matrix[ rownumber ] ) );
    end.
```

Notes:

1. A *function*, "maxrowelement" is declared. Functions are similar to procedures except that they *return* a value. The function is invoked by using it in an expression, and the value that is returned replaces the function name in the computation of the expression.

2. Somewhere in the body of the function there must be a statement that assigns the value to be returned to the function name.

3. Functions are declared following the variable declarations (the same place as procedures). The ":integer" after the function header says what type of value the function will return.

4. The function in this program is declared with a *parameter*. This is done so that the function may be used with different data. In this case, the parameter is an entire row from the matrix. Parameters may also be used with user-defined procedures.

5. The function has its own declaration section, and declares some *local variables*. They are called local because they are "created" when the function is invoked (and "destroyed" when the function returns), and because only the function in which the variables are declared may refer to them. For example, if you were to refer to variable "max" in the body of the program, an error would occur.

6. Functions and procedures may also declare local constants and types (and even local procedures and functions).

7. "maxint" is a built-in constant that represents the largest integer that can be represented on the computer. Thus, "−maxint" is the smallest integer.

**EXAMPLE 21 Character Variables**

This program introduces Pascal character manipulation.

```
program example21( input, output );
  const
    stringlength = 5;
    greeting = 'Howdy';
  type
    stringtype = packed array[ 1..stringlength ] of char;
  var
    string : stringtype;
    index : 1..stringlength;
    characterindex : char;

  begin
    string := 'Hello';
    writeln( 'string has the value ''', string, '''' );
    string := greeting;
    writeln( 'string now has the value ''', string, '''' );
    for index := 1 to stringlength do
      writeln( 'string[', index:2, '] is ''',
        string[ index ], '''' );
    writeln( 'Enter a ', stringlength:2,
      '-character string' );
    for index := 1 to stringlength do
(*1*)  read( string[ index ] );
    readln;
    writeln( 'You entered ''', string, '''' );
    writeln(
      'Here are the lower-case alphabetic characters:' );
    for characterindex := 'a' to 'z' do
(*2*)  write( characterindex );
    writeln;
  end.
```

Notes:

1. A new type is used in this program: the *char* type. The set of values specified by this type is the character set of the computer being used to run your Pascal programs.

2. A new type is defined: "stringtype". It is a five-element array of single characters. The keyword *packed* indicates something to Pascal; for all intents and purposes it may be ignored (although it must be there!).

3. String constants such as 'Howdy' may be assigned to variables which are declared to be of type "stringtype"; however the constants must be *exactly* the same length as the array.

4. The for statement indicated by (*1*) shows how to read a character string one character at a time.

5. The for statement indicated by (*2*) demonstrates that a character variable may be used as a for statement index, in which case the initial and final values must be characters.

**EXAMPLE 22 Arrays of Strings**

A list of strings is read by the program and saved in an array. The program prints each string according to the reply to the prompt, and stops when an invalid number is entered.

```
program example22( input, output );
  const
    stringstart = 1;
    stringend = 20;
    liststart = 1;
    listend = 5;
    blankstring = '
  type
    stringsize = stringstart..stringend;
    stringtype = packed array[ stringsize ] of char;
    listsize = liststart..listend;
    listtype = array[ listsize ] of stringtype;
  var
    list : listtype;
    requestedstring, listindex : listsize;
    done : boolean;

  function getrequest : listsize;
    var
      n : integer;
    begin
      writeln(
        'Enter the number of the string you wish to see:' );
      readln( n );
      done := ((n < liststart) or (n > listend));
      if( done )then
        n := liststart;  (* return anything valid *)
      getrequest := n;
    end;

  procedure getstring( which : listsize );
    var
      index : integer;
      junk  : char;
    begin
      list[ which ] := blankstring;
      index := stringstart;
```

```
        while( not eoln )do
           if( index > stringend )then
              read( junk )   (* get rid of unwanted chars *)
           else
              begin
                 read( list[ which ][ index ] );
                 index := index + 1;
              end;
        readln;
     end;

  begin
     for listindex := liststart to listend do
        begin
           writeln( 'Enter a string:' );
           getstring( listindex );
        end;
     requestedstring := getrequest;
     while( not done )do
        begin
           writeln( list[ requestedstring ] );
           requestedstring := getrequest;
        end;
  end.
```

Notes:

1. The parameter to procedure "getstring" indicates which string is to be read. The string is read one character at a time, up to a maximum of 20 characters.

2. "getstring" uses a built-in Boolean function called "eoln" ("end-of-line"). "Eoln" returns true if all the characters on the line you typed have been read in, otherwise it returns false.

3. Variable "list" is usually thought of as an array of strings. It could be thought of as a two-dimensional character matrix, however.

4. The method used to determine if the reply to the prompt should stop the program is somewhat more complicated than before. A *compound Boolean expression* with an "or" operator is used to determine if the reply is a valid index into the array. If it isn't, variable "done" is set to true, and the program stops.

**EXAMPLE 23 Enumerated Types**

Some simple properties of Pascal's enumerated types are demonstrated.

```
program example23( output );
  type
    colour = (red, yellow, blue, orange, green, purple);
    primary = red..blue;
  var
    shade : colour;
    basic : primary;

  begin
    shade := orange;
    basic := yellow;
    if( shade = green )then
      writeln( 'The value of shade is green.' )
    else if( shade < green )then
      writeln( 'The value of shade is less than green.' )
    else
      writeln( 'The value of shade is greater than green.' );
    if( shade > blue )then
      writeln(
        'The value of shade is not in the primary subrange.' )
    else
      writeln(
        'The value of shade is in the primary subrange.' );
    shade := pred( shade );
    basic := succ( basic );
    if( shade = basic )then
      writeln( 'Shade and basic have the same value.' );
    basic := purple;
  end.
```

Notes:

1. The user-defined type "colour" is called an *enumerated* type. An enumerated type defines *all* the constant values that make up the type. For example, the standard type "Boolean" is really an enumerated type defined as follows:

```
type boolean = (false, true);
```

As you recall, false and true are constants of the Boolean type.

2. In this example, red, yellow, blue, orange, green and purple are constants of the colour type.

3. An enumerated type also specifies an *ordering* of the constants. In particular, false is less than true, and red < yellow < blue < orange < green < purple.

4. Subranges of enumerated types may be declared. This means that enumerated types may be used as array indices, for example.

5. Two new built-in functions are used in the program. They are *pred* and *succ*. Pred ("predecessor") returns the value that precedes its parameter, according to the ordering defined by the type declaration. Succ ("successor") returns the next value in the ordering. Succ and pred may also be used with integers, so that "pred( 12 )" would be 11, and "succ( −15 )" would be −14, for example.

6. The last assignment statement in the program causes an error. Variable "basic" is of type "primary", which has only three values (red, yellow and blue); purple is not one of these values, so an error occurs.

**EXAMPLE 24 Set Types and the Case Statement**

One of the more unusual type constructs in Pascal is the set type. This example demonstrates the use of sets.

```
program example24( output );
  type
    colour = (red, yellow, blue, orange, green, purple);
    blend = set of colour;
  var
    shade : colour;
    rainbow : blend;

  procedure colourstring( requested : colour );
    begin
      case requested of
        red: write( 'red' );
        yellow: write( 'yellow' );
        blue: write( 'blue' );
        orange: write( 'orange' );
        green: write( 'green' );
        purple: write( 'purple' );
      end
    end;

  procedure whatsintheset( s : blend );
    var
      colourindex : colour;
    begin
      for colourindex := red to purple do
        if( colourindex in s )then
          begin
            colourstring( colourindex );
            writeln( ' is in the set.' );
          end;
    end;

  begin
    writeln(
      '*** Initial definition: red, yellow, blue and purple.' );
    rainbow := [red, yellow, blue, purple];
    whatsintheset( rainbow );
    writeln( '*** Orange is added.' );
```

```
    rainbow := rainbow + [orange];
    whatsintheset( rainbow );
    writeln( '*** Yellow is removed.' );
    rainbow := rainbow - [yellow];
    whatsintheset( rainbow );
    writeln( '*** Intersected with purple.' );
    rainbow := rainbow * [purple];
    whatsintheset( rainbow );
    writeln( '*** Nothing is in a null set:' );
    whatsintheset( [] );
    writeln( '*** Was there anything?' );
    writeln( '*** Everything should be in this one:' );
    whatsintheset( [red..purple] );
  end.
```

Notes:

1. A *set* may be thought of as a collection of elements of some other type (called the *basetype* of the set). In this example, we have a set (or collection) of colours. The operations that may be performed on a set include: the ability to add elements to a set (set union, denoted by "+"); removing elements from a set (set difference, denoted by "−"); finding out what elements are common to two sets (set intersection, denoted by "*"); and testing to see if a particular element is in a set (set membership, denoted by "in").

2. Set constants are formed by enclosing constants of the set basetype in square brackets, for example [red] or [yellow..orange]. The latter example means all elements from yellow to orange. A set with no elements, the *null set*, is formed with empty brackets: [].

3. The case statement used in procedure "colourstring" is used to select one of a number of alternatives. The first line of the statement contains a *selector expression* (in this case, just a variable value). The case statement attempts to find a match between the selector expression value and one of the *case constants* that follow. If a match is found, then the statement (or compound statement) beside the case constant is executed. If no match is found, an error occurs.

# Introduction to the Reference Manual

The Reference Manual consists of two main sections (A and B) followed by several brief sections (C, D, E and F) containing quick-reference summaries. The last section (G), gives details particular to Waterloo microPascal. All other sections refer to the Pascal language in general.

Section A describes the features of the Pascal language. It specifies the syntax and meaning of each construct and statement in the language. This section describes declarations for constants, types, variables, functions and procedures. The rules for executable statements such as assignment statements and control statements are also defined in this section.

Section B describes the standard (predefined) constants, types, variables, procedures and functions. This includes the standard types *integer, char, Boolean* and *real*. The standard procedures and functions of Pascal provide much of the capability of the language; for example, input/output is accomplished in this way.

The next sections (C, D, E and F) provide brief summaries of the reserved words, delimiters, operators and syntax.

# Reference Section A

# Syntax and Semantics Definition

## A.1 Notation

The following notation is used in the syntax definition of Pascal.

| | | |
|---|---|---|
| | ⟨*abc*⟩ | *abc* is optional |
| | $\{abc\}^0$ | *abc* may be repeated 0 or more times |
| | $\{abc\}^1$ | *abc* must be repeated 1 or more times |
| | *abc* \| *def* | choose *abc* or *def* |
| | *abc* | |
| or | *def* | choose *abc* or *def* |
| | **abc** | **abc** is a keyword |

Definitions will be enclosed like the definitions above. The item being defined will be shown in *italics* and the definition of the item will follow, beginning on the next line and indented. The style of definition is based on a modification of Backus-Naur form.

## A.2 Basics

*digit*
"0" | "1" | "2" | ... | "9"

*letter*
"a" | "b" | "c" | ... | "z"
or "A" | "B" | "C" | ... | "Z"

*number*
    {digit}[1]⟨.{digit}[1]⟩⟨exponent⟩

*exponent*
    e⟨+|−⟩{digit}[1]

*id*
    letter{letter|digit}[0]

*string*
    '{any character}[1]'

There are four basic classes of symbols which constitute the vocabulary of the Pascal language:

(1) numbers (e.g. 1, 1.2, 1.2e34, 1.2E−21)

(2) id's (short for identifiers) (e.g. *X, y, abc, z21*)

(3) quoted strings (e.g. 'a', 'abc')

(4) special symbols (e.g. **begin, end,** :=, ;)

These symbols are also known as tokens. A single token must be completely contained on a single line. The maximum length of an identifier is bounded only by the rule that tokens may not span lines. It is only guaranteed that the first eight characters of an identifier will be used to distinguish it from other identifiers.

Special symbols include delimiters such as comma (,), semicolon (;), and reserved words such as **if, while** and **begin.** Some special symbols are not available on all computer hardware so alternate representations are available for them; see Reference Section D for definitions of these alternate representations.

Letters outside quoted strings are case insensitive (i.e. capitalized and uncapitalized letters are treated as being equivalent); **Begin** is equivalent to **begin** and the variable *A* is the same as *a*.

In a number an "e" means "times 10 to the power of". Reference section A.4.2 describes numeric constants in detail.

In a quoted string, two consecutive quotes are used to represent each quote character which is to be part of the string. For example, the quoted string

    'it''s'

contains the word

    it's

A comment consists of an opening brace ({) followed by any string of characters, followed by a closing brace (}). Comments may not contain closing braces.

Blanks, comments and ends of lines are known as token separators. The following rules apply:

(1) at least one token separator must occur between any consecutive pair of id's, keywords or numbers,

(2) token separators may appear only between tokens, never within tokens; blanks within quoted strings are not considered to be token separators,

(3) token separators do not otherwise affect the meaning of a program.

Thus, a Pascal program may be entered in "free format" as long as tokens do not span lines and tokens are properly separated from one another.

## A.3 Programs and Blocks

---

*program*
    program-heading;
        block

*program-heading*
    **program** program-name ⟨program-parameter-list⟩

*program-name*
    id

*program-parameter-list*
    ( id-list )

*id-list*
    id $\{, \text{id}\}^0$

*block*
    declarations
    **begin**
        $\{\text{statement};\}^0$
        statement
    **end**

---

A program consists of a program heading followed by a block. This block is called the main block; program execution begins with the activation of the main block. The main block is followed by a period (.).

The program heading gives a name to the program and optionally declares a list of identifiers which is the program parameter list. The program name is the identifier directly following the keyword **program.** It has no meaning within the program although some implementations may choose to give it a meaning outside the program.

Program parameters refer to variables in the main block (usually files) which may correspond (in some implementation-defined manner) to entities which exist outside the program. They facilitate communication between a Pascal program and the system under which it is running. If a program is to reference files which were in existence before the program is executed, or if files which the program processes are to remain in existence after the program terminates execution, then these files are called external files and their names must occur in the program parameter list. (External files must also be global; i.e. declared in the main block.) If the standard files *input* and *output* are mentioned in the program parameter list then they are declared and automatically initialized prior to program execution.

*example*:

> **program** *example( input, output );*

A block is a basic unit in the Pascal language. Programs, functions and procedures each consist of a heading followed by a block. The heading associates a name with the block. A block consists of two parts:

(1) declarations,

(2) executable statements.

The declarations define the items to be operated upon, such as variables. The executable statements define the actions to be performed when the block is activated.

### A.4 Declarations and Scope

---

*declarations*
  ⟨label-declarations⟩
  ⟨constant-declarations⟩
  ⟨type-declarations⟩
  ⟨variable-declarations⟩
  ⟨procedure-and-function-declarations⟩

---

Every entity which is referenced in a Pascal program (i.e. labels, constants, types, variables, functions and procedures) must be defined in a declaration.

Since the declarations for a block can themselves contain procedure and function declarations, blocks can be nested to an arbitrary depth. Entities defined in a particular block are said to be *local* to that block. A nested block inherits all of the declarations from the parent block in which it is contained. Any inherited definitions may be superceded by local definitions.

Entities defined in the main block are said to be *global*, since all procedures in the program can potentially inherit them.

The set of blocks over which a particular definition of an identifier or label applies is called the *scope* of the definition.

An identifier or label may have only one definition for each block. Once an identifier has been defined in a declaration or used to reference an inherited definition, the meaning of the identifier for that block is determined. An identifier must be defined prior to its use except in the following case: a pointer type declaration may reference a type identifier which is defined subsequently in the type declarations.

### A.4.1 Labels

---

*label-declarations*
  **label**
    label {, label}$^0$;

---

A label declaration defines a symbol to be a statement label.

---

*label*
  {digit}$^1$

---

A label must identify exactly one statement in the executable statements of the block in which the label is local.

The label may be referenced by any **goto** statement within its scope.

### A.4.2 Constants

---

*constant-declarations*
**const**
{id = constant;}[1]

---

A constant declaration defines an identifier to represent a constant value. The constant identifier may then be used in place of the constant value, anywhere within the scope of the identifier.

---

*constant*
⟨+ | −⟩ number
or ⟨+ | −⟩ id
or string

---

Constant values have one of four data types.

(1) A number which has no decimal point or exponent is of type *integer* (e.g. 12345).

(2) A number which has a decimal point, an exponent, or both is of type *real* (e.g. 123.45, 123e45, 123.45e67).

(3) A quoted string of length one is of type *char* (e.g. 'a' ).

(4) A quoted string of length greater than one is of type
**packed array** [ 1 .. *length* ] **of** *char*
(e.g. 'Hello' is of type **packed array** [1..5] **of** *char*).

In *real* numeric constants an "e" means "times 10 to the power". (An upper case "E" may be used instead of the lower case "e".) This is called exponential notation or scientific notation.

An identifier used as a constant must have been previously defined as a constant. Note that where a constant value is necessary (e.g. in a subrange type declaration), a variable will not suffice.

A constant preceded by a sign (+,−) must be of type *integer* or *real*. The plus sign (+) has no effect and the minus sign (−) denotes negation (change of sign).

### A.4.3 Types

*type-declarations*
**type**
{id = type;}[1]

A type declaration defines an identifier to be the name of a type.

*type*
type-id
or enumerated-type
or subrange-type
or ⟨**packed**⟩ array-type
or ⟨**packed**⟩ set-type
or ⟨**packed**⟩ file-type
or pointer-type
or ⟨**packed**⟩ record-type

Types are used to describe data.

**Packed** indicates that the compiler should store the data in a compact manner, possibly at the cost of less efficient access to the data.

*Definitions Relating to Types*

*Ordinal Type*

A type is ordinal if it is any of the following:

(1) *integer,*

(2) *char,*

(3) enumerated (including *Boolean*),

(4) subrange.

Note that this does not include type *real*.

Ordinal types all define an *ordered* set of values.

*Identical Types*

A type, *t1*, is identical to another type, *t2*, if they have been declared to be equivalent in a declaration of the form

**type**
  *t1* = *t2;*

Any type is naturally identical to itself.

*String Type*

A type is a string type if it is of the form

**packed array** [ 1 .. *n* ] **of** *char*

In particular, the following are *not* string types:

- **packed array** [ 0 .. *n* ] **of** *char*
- **packed array** [ ( *red, green, blue* ) ] **of** *char*
- **packed array** [ 1 .. *n* ] **of** 'a'..'z'
- **array** [ 1 .. *n* ] **of** *char*

*Compatible String Types*

Two string types are compatible if they have the same number of elements.

*examples*:

'abc' is type compatible with any **packed array** [ 1..3 ] **of** *char*

'abc' is *not* type compatible with 'ab' or 'abcd'

*Compatible Types*

Two types, *t1* and *t2*, are compatible if at least one of the following holds:

(1) *t1* and *t2* are identical,

(2) *t1* is a subrange of *t2*,

(3) *t2* is a subrange of *t1*,

(4) *t1* and *t2* are both subranges of another type,

(5) *t1* and *t2* are compatible string types,

(6) *t1* and *t2* are sets with compatible base types,

(7) *t1* is compatible with type *integer* and *t2* is *real*,

(8) *t2* is compatible with type *integer* and *t1* is *real*.

*Assignment Compatible*

A type, *t1*, is assignment compatible to another type, *t2*, if *t1* and *t2* are compatible, provided that if *t1* is *real* then *t2* must be *real*. In other words, *real* values may not be assigned to *integer* variables.

### A.4.3.1 Simple Types

---

*type-id*
    id

---

A type may be defined simply by a reference to a previously defined type identifier. There are five predefined type identifiers:

(1) *integer,*

(2) *char,*

(3) *real,*

(4) *Boolean,*

(5) *text.*

Reference Section B.3 describes predefined types.

*example*:

**type**
  *temperature = real;*

---

*enumerated-type*
    ( id-list )

---

A type may be defined by enumerating a list of identifiers which are to denote the values of the type. Each id in the list is then a constant of the enumerated type which is being defined.

*example*:

**type**
  *spectrum = ( infrared, red, green, blue, ultraviolet );*

*subrange-type*
constant .. constant

A type may be defined by specifying a range of values within a previously defined ordinal type. The new type is denoted by the low and high bounds for the range of values. The low and high bounds must be constants of the same type, called the *base type,* and the low bound must be less than or equal to the high bound.

*example*:

```
type
  visible = red..blue;
  days = 0..365;
```

### A.4.3.2 Arrays

*array-type*
**array** [ index-type {, index-type}$^0$ ] **of** type

*index-type*
type-id
or enumerated-type
or subrange-type

An array is a fixed-length list of data items, all of the same specific type (called the constituent type). Each element in the list is identified by an element from the set defined by the index type, which must be ordinal. The number of elements in an array is therefore the number of elements in the ordered set defined by its index type.

*example*:

**array** [ 1..10 ] **of** *char*

The above defines a list of ten characters, which might be viewed as a ten-character word. The construct

**array** [ 1..20 ] **of array** [ 1..10 ] **of** *char*

defines a list of twenty words of ten characters each.
Pascal permits this to be denoted more conveniently as

**array** [ 1..20, 1..10 ] **of** *char*

Reference Section A.5.2.1 describes the access of array elements.

Arrays of the form

**packed array** [ 1 .. *n* ] **of** *char*

are called strings, and the relational operators are defined for them.

#### A.4.3.3 Sets

---

*set-type*
**set of** enumerated-type
**set of** subrange-type
**set of** type-id

---

A set type is defined in terms of an ordinal *base type*, and represents a collection of elements from its base type. Each element in a set can have one of two states: present or not present.
Consider the example:

```
type
  fruit = ( apple, orange, peach );
  basket = set of fruit;
```

The type *fruit* has three values denoted by *apple*, *orange* and *peach*.

The type *basket* has eight possible values denoted by

| | |
|---|---|
| [ ] | [ *apple, orange* ] |
| [*apple*] | [ *apple, peach* ] |
| [*orange*] | [ *orange, peach* ] |
| [ *peach* ] | [ *apple, orange, peach* ] |

The possible values for a set are all the combinations of the elements from its base type, including the empty set. This is the set of all subsets of the base type, and is called the *powerset* of the base type. The ordinal positions of the largest and smallest elements in the base type of a set are implementation-defined.

The set operators are described in Reference Section A.5.2.2. They include set union, set difference, set intersection and tests for set inclusion.

### A.4.3.4 Files

---

*file-type*
  **file of** type

---

File types are lists of elements, all of one particular type (called the constituent type). There are several significant differences between files and arrays that make files particularly suitable for representation on terminals or printers, or for storage on disk or tape.

Before a file may be used it must be initialized. This is done by an activation of a standard procedure. The elements may then be accessed sequentially; this means that one element only is available at any given time. The element that is available is called the current element. This access scheme may be viewed as having a window on the file from which one element may be seen. Every file has a *buffer variable* associated with it which contains the value of the current element. Whenever access to a file is initiated, the current element is the first element in the file. The next element after the current element can become the current element (the window may be moved ahead one element) by an activation of a standard procedure. No other movement of the window, such as ahead more than one at a time, is provided.

A file may be accessed for either reading or writing at any one time. Reading means that the elements may be examined but not modified. Writing means that the contents of the file are deleted and new elements may then be added to the empty file. A file may be accessed an arbitrary number of times by a program.

The number of elements in a file is not specified in the declaration. There is a standard procedure to detect when the window has been advanced past the last element when reading a file.

If a file existed before a program using it was executed, or if a file is to remain in existence after a program processing it has terminated execution, then the file is said to be external to the program. External files permit communication between programs. The names of the file variables corresponding to external files must occur in the program parameter list and the file variables must be global (i.e. declared in the main block). Files which are not external are said to be internal and exist only for the duration of the program.

See Reference Section A.5.2.1 for a description of the access of a file buffer variable. See Reference Section B.5.7 for a description of the file manipulation procedures and functions. Reference Section B.3.5 describes the standard type *text,* which is the type of the standard files *input* and *output.*

### A.4.3.5 Pointers

---

*pointer-type*
    ↑ type-id

---

A pointer type has values which "point to" variables of a one particular type (called the base type).
For example:

**type**
  *x* = ↑ *integer;*

defines *x* to represent a type whose values denote *integer* variables.

The variables pointed to by pointers are created and destroyed at execution time by the standard procedures *new* and *dispose.*

Reference Section A.5.2.1 describes the use of pointer variables. Reference Section B.5.2 describes the procedures *new* and *dispose.*

### A.4.3.6 Records

---

*record-type*
**record**
field-list
**end**

*field-list*
fixed-fields ⟨;⟩
or fixed-fields; variant-part ⟨;⟩
or variant-part ⟨;⟩

*fixed-fields*
{id-list : type;}$^0$
⟨id-list : type⟩

*variant-part*
**case** ⟨tag-name :⟩ tag-type **of**
{variant;}$^0$
⟨variant⟩

*tag-name*
id

*tag-type*
type-id

*variant*
variant-label-list : (field-list)

*variant-label-list*
constant {, constant}$^0$

---

A record type is a fixed-length list of elements *not* necessarily all of the same type. The elements are called fields and each has a field name which designates it.

*example*:

```
type
  employee =
    record
      name : packed array [ 1..20 ] of char;
      age : integer;
      sex : ( male, female );
    end
```

A record may be defined to have a *variant* part. This allows a choice in the definition of the record at execution time. At any time during execution, only *one* of the variants of the record may exist. The value of the tag field indicates which variant is currently in existence.

A tag field name may be specified by including an identifier followed by a colon directly after the keyword **case**. The tag type must always be specified following the optional tag field name. The types of the case label constants must be compatible with the tag type.

If the tag field name is specified, then assignment of one of the case label values to it activates the variant corresponding to that case label. Assignment of a value which is not a variant case label to the tag field is an error.

If the tag field name is not specified in the record definition then assignment to a field which is not in the currently-active variant activates the newly-referenced variant. When a variant is activated, the previous variant ceases to exist and the fields in the new variant have undefined values.

*example*:

```
type
  employee =
    record
      name : packed array [ 1..20 ] of char;
      sex : ( male, female );
      case employed : Boolean of
        true : ( jobname : array [ 1..20 ] of char );
        false : ( unemploymentamount : integer );
    end
```

Reference Section A.5.2.1 describes the access of fields within record variables.

### A.4.4 Variables

*variable-declarations*
    **var**
        {id-list : type;}[1]

A variable is used to store a value. Each variable has a type and can store only values of that type.

Variable declarations define one or more identifiers to represent variables of a particular type.

### A.4.5 Procedures and Functions

*procedure-and-function-declarations*
    {procedure-or-function-declaration}[1]

*procedure-or-function-declaration*
    procedure-heading;
        body;
or  function-heading;
        body;

*procedure-heading*
    **procedure** id ⟨formal-parameters⟩

*function-heading*
    **function** id ⟨ ⟨formal-parameters⟩ : type-id⟩

*body*
    block
or  directive

*directive*
    id

A procedure or function is a named block. A procedure is activated by a procedure invocation statement. A function is activated by a function reference in an expression and returns a value.

The definition of a procedure or function consists of:

(1) a heading which must specify the name of the function or procedure, its parameters, and the type of value it returns if it is a function,

(2) the block which is to be executed upon activation of the function or procedure.

When a function is activated, the value it returns is the value most recently assigned in an assignment statement which specifies the name of the function on the left-hand side. Within a function, if the name of the function is used in an expression, except on the left-hand side of an assignment, it indicates a recursive activation of the function.

Functions may only return values of ordinal type or of type *real*.

The block may be defined separately and subsequently to the heading by using the directive *forward* in place of the block. The block must then occur subsequently with a heading which specifies no parameters nor a return value type.

*example*:

```
procedure X( y, z : integer );
  forward;

function Y( u, v : integer ) : integer;
  forward;

procedure X;
  begin
    ...
    ... := Y( x, z-1 );
    ...
  end;

function Y;
  begin
    ...
    X( u, v );
    ...
  end;
```

Procedure $X$ and function $Y$ are said to be mutually recursive with respect to each other, since each may invoke the other. Because Pascal requires that entities be declared before they are referenced, the *forward* directive is essential for defining mutually recursive procedures or functions.

### A.4.5.1 Formal Parameters

---

*formal-parameters*
( parameter-group {; parameter-group}$^0$ )

*parameter-group*
id-list : type-id
or **var** id-list : type-id
or procedure-heading
or function-heading

---

The procedure or function heading permits declarations of *formal parameters*. Parameters allow information be passed to a block upon activation.

The parameters in a procedure or function heading are known as formal parameters. The entities in a parameter list in an *invocation* are known as *actual parameters* and correspond to the formal parameters when the procedure or function is activated.

Four classes of parameters exist:

(1) value parameters: This is the default class for parameters. This technique of parameter passing is referred to as *call by value*. The formal parameter is a variable in the block. It is assigned the value of an actual parameter upon activation. The actual parameter must be of assignment compatible type to the formal parameter. Since files may not be assigned, they may not be passed as value parameters.

(2) variable parameters: This class of parameters is designated by the **var** keyword. This technique of parameter passing is referred to as *call by reference* or *call by address*. An actual parameter must be a variable of identical type to the formal parameter. Within the block the formal parameter denotes the variable specified as the actual parameter.

(3) procedure parameters: A parameter of this class is declared by specifying a procedure heading as a formal parameter. An actual parameter must be a procedure with a compatible parameter list (as defined below) to the formal parameter. Within the block, the formal parameter denotes the procedure specified as the actual parameter. When a procedure is activated as a formal parameter it has the environment (inherited definitions) from which it was passed as an actual parameter.

(4) function parameters: A parameter of this class is declared by specifying a function heading as a formal parameter. An actual parameter must be a function with a compatible parameter list (as defined below) to the formal parameter, and of identical result type to the result type specified for the formal parameter. Within the block the formal parameter denotes the function specified as the actual parameter. When a function is activated as a formal parameter it has the environment (inherited definitions) from which it was passed as an actual parameter.

Two parameter lists are compatible if they have the same number of parameters and each corresponding pair of parameters is one of the following:

(1) value parameters of identical type,

(2) variable parameters of identical type,

(3) procedure parameters with compatible parameter lists,

(4) function parameters with compatible parameter lists and identical result types.

### A.5 Executable Statements

---

*statement*
⟨label :⟩
⟨unlabelled-statement⟩

---

Executable statements define actions to be performed. Each executable statement may have a label associated with it so that it can be referenced by a **goto** statement.

Note that by the above definition a statement may consist of nothing at all. A statement consisting of nothing is called a *null statement* and does not cause any action when it is executed. Null statements are in no way detrimental to a program and arise surprisingly often in Pascal programs. This is largely because semicolons (;) are used as *statement separators* instead of *statement terminators* in Pascal, and therefore no semicolon is required between the last statement in a block and the **end** keyword. If a semicolon is included after the last statement in a block then a null statement exists between that semicolon and the **end** keyword.

*example*:

```
begin
  x := 1;
end
```

In the above example a null statement occurs between the semicolon and the **end** keyword.

---

*unlabelled-statement*

procedure-invocation
or assignment-statement
or control-statement
or compound-statement

*compound-statement*

**begin**
{statement;}$^0$
statement
**end**

---

Executable statements are divided into four classes. Three classes (procedure invocation statements, assignment statements and control statements) are described in subsequent sections. The fourth class of statement is the compound statement.

A compound statement is simply a list of statements separated by semicolons and enclosed by a **begin-end** pair. Anywhere that a single executable statement may be used, a compound statement may be used. A compound statement can contain as many statements as necessary.

### A.5.1 Procedure Invocation and Parameters

---

*procedure-invocation*
    procedure-id
or  procedure-id ( actual-parameter {, actual-parameter}$^0$ )

*actual-parameter*
    procedure-id
or  function-id
or  variable
or  expression
or  write-parameters

*procedure-id*
    id

*function-id*
    id

*write-parameters*
    expression ⟨ ⟨ : expression⟩ : expression⟩

---

The procedure-invocation statement is used to activate a procedure and specify any actual parameters to the procedure. Reference section A.4.5 describes passing parameters to procedures.

Note that there is a special form of actual parameter which may be used only with *write* and *writeln* to specify the field width for textfile output. See Reference Section B.5.7 for further details.

### A.5.2 Assignment Statement (Variables and Expressions)

---

*assignment-statement*
    var := expression

---

The assignment statement is used to assign the value of an expression to a variable. The type of the expression must be assignment compatible with the type of the variable. Files may not be assigned.

*examples*:

*a* := 1;
*b*↑.*a* := *blue;*
*a*[ 1 ] := *abc;*

Note that, because of the rules for assignment compatibility, *integer* values may be assigned to *real* variables, but *real* values may not be assigned to *integer* variables without the use of the *round* or *trunc* functions.

### A.5.2.1 Variables

---

*variable*
id
or subscripted-variable
or variable-with-field-selection
or indirectly-referenced-variable

*subscripted-variable*
variable [ expression {, expression}$^0$ ]

*variable-with-field-selection*
variable . field-name

*field-name*
id

*indirectly-referenced-variable*
variable ↑

---

A variable is used to store a data value. Variables may be referenced in different ways depending on their type.

*Simple Variables*

A simple variable (not within an array or record, and not dynamically created) is specified by its identifier.

*example*:

```
var
   a : integer;
...
a := 1;
```

*Elements In Array Variables*

An element of an array type variable is specified using a *subscript* enclosed by square brackets ( [] ) following the array variable name. The subscript is a value from the index type of the array variable, and indicates which element is to be selected from the array.

*example*:

```
var
   a : array [ 1..10 ] of integer;
...
a[ 5 ] := 1;
```

An array *a* with *n* subscripts *s1, s2, ..., sn* may be referenced in either of the following ways:

*a[s1][s2]...[sn]*

or

*a[ s1, s2, ..., sn ]*

*Fields In Record Variables*

A field within a record variable is specified by the record variable, followed by a dot (.) (the field selection operator), followed by the name of the field to be selected.

*example*:

```
var
  a :
    record
      r : real;
      i : integer;
    end
...
a.i := 1;
a.r := 10.4;
```

*Dynamically Created Variables*

A dynamically created variable (created by procedure *new*) which a pointer value identifies, may be specified by using the upward-pointing arrow (↑) ("points to" notation). This operation is sometimes called an "indirect reference" or "indirection".

*example*:

```
var
  a : ↑ integer;
...
new( a );
..
a↑ := 1;
```

*File Buffer Variables*

The file buffer variable (current element) for a file is also referenced using the "points to" notation. (The "points to" notation does not imply that the file variable contains a pointer to the file buffer; it is just a coincidence that the same notation is used to reference dynamically-created variables.)

*example*:

```
var
   a : file of integer;
...
rewrite( a );
a↑ := 1;
put( a );
```

### A.5.2.2 Expressions and Operators

---

*expression*
    simple-expr
or    simple-expr relational-operator simple-expr

*relational-operator*
    =
or    <>
or    <
or    <=
or    >
or    >=
or    **in**

*simple-expr*
    ⟨+|−⟩ term
or    simple-expr adding-operator term

*adding-operator*
    +
or    −
or    **or**

*term*
    factor
or    term multiplying-operator factor

*multiplying-operator*
    *
or    /
or    **div**
or    **mod**
or    **and**

---

An expression is a sequence of elements specifying data such as variables, and operators such as + and −. The elements specifying data are called *factors* and are described in the next section (the **not** operator is also described with factors). See Reference Section E for a table summarizing the operators and their valid operand types.

By the standard rules of algebra, the expression

$$a + b * c$$

is equivalent to the expression

$$a + (b*c)$$

rather than

$$(a+b) * c$$

This is because the operator * is of higher *priority* than the operator +, and higher priority operators are performed first.

Brackets also may affect the order of evaluation of an expression; see Reference Section A.5.2.3.

It can be seen from the above syntax definition for expressions that Pascal has four priorities of operators:

(1) relational operators (lowest priority)

(2) adding operators (same priority as +)

(3) multiplying operators (same priority as *)

(4) Boolean **not** operator (highest priority)

Observe that due to the syntax for expressions, the expression

$a < b$ **and** $c < d$

is a syntax error, which is not intuitively expected. Similarly, in the expression

$a < b$ **and** $c$

the **and** is evaluated first which is different from many other programming languages. Bracketing may be used to overcome these problems.

The following description of the operators is organized by the priorities of the operators.

*Relational Operators* ( $=$, $<>$, $<$, $<=$, $>$, $>=$, **in** )

The relational operators are used to compare values to determine if a particular relationship (e.g. "less than") holds between them. The result is always of type *Boolean;* it is *true* if the relationship specified by the operator holds, and *false* otherwise. (See Reference Section E for a definition of what relationship each operator denotes.) All of the relational operators take two operands.

The relational operators $=$, $<>$, $<=$, $>=$, $<$ and $>$ may be applied to compatible operands of type *real, integer, char,* enumerated (including *Boolean*), subrange, or string.

The relational operators $=$, $<>$, $<=$ and $>=$ may be applied to compatible set types, in which case $<=$ and $>=$ denote set inclusion.

The relational operators $=$ and $<>$ may be applied to pointer types with identical base types.

In arithmetic comparisons, if one operand is *real* and the other is *integer* then the *integer* operand is converted internally to a *real* value to be used in the comparison.

The operator **in** takes a set as its right operand and an expression of a compatible type to the base type of the set as its left operand. **In** yields *true* if the left operand value is included in the set value specified by the right operand, and *false* otherwise.

*examples*:

> $a < b$
> [1,2] <= [1,2,3]
> *character* **in** ['a'..'z']

*Adding Operators* ( +, −, **or** )

If the adding operators + and − have only a right operand (e.g. −5) they are said to be unary (or monadic). If they have two operands (e.g. 4−5) they are said to be binary (or diadic).

Unary + has no effect (the *identity* operation in algebraic terms). Unary − represents negation (change of sign). Both take an *integer* or *real* operand and yield a result of the same type as the operand.

The binary + and − denote addition and subtraction for numeric values and set union and difference for sets.

Addition and subtraction require two *integer* or *real* operands. The result is the same type as the operands. If one operand is *integer* and the other is *real* then the *integer* operand is converted internally to a *real* value to be used in the operation and the result is of type *real*.

Set union and set difference require compatible set operands. They yield a set value of appropriate type.

The **or** operator is a *Boolean* operator. It requires two *Boolean* operands and yields a *Boolean* result. The result is *true* if either or both of the operands are *true*, and *false* otherwise.

*examples*:

> $a + b$
> $c$ + [ *red, blue, green* ]
> $f$ **or** $g$
> [1..10] − [$a$..$b$]

*Multiplying Operators* ( *, /, **div, mod, and** )

The operator * represents multiplication with numeric operands and intersection with set operands.

Multiplication requires two operands of type *integer* or *real*. The type of the result is the same as the type of the operands. If one operand is *integer* and the other *real*, then the *integer* operand is converted internally to a *real* value to be used in the operation and the result is of type *real*.

Set intersection requires compatible set types as operands and yields a set value of appropriate type.

The operator / represents real division. The operands must be of type *real*. If one or both operands is of type *integer* the conversion to *real* takes place before the operation is performed. The result is always of type *real*.

The operator **div** represents integer division. The operands must be of type *integer* and the result is always of type *integer*. *A* **div** *b* yields the number of times the absolute value of *a* may be subtracted from the absolute value of *b* and still leave a positive quantity.

The **mod** operator represents integer remainder. The operands must be of type *integer* and the result is always of type *integer*. *A* **mod** *b* yields the remainder when *a* is divided by *b*.

The **and** operator is a *Boolean* operator. It requires two *Boolean* operands and yields a *Boolean* result. The result is *true* if both of the operands are *true*, and *false* otherwise.

*The* **not** *Operator*

**Not** is the highest priority operator. It is described in the next section.

### A.5.2.3 Expression Factors

*factor*

|  |  |
|---|---|
|  | variable |
| or | number |
| or | string |
| or | constant-id |
| or | **nil** |
| or | ( expression ) |
| or | set-constructor |
| or | **not** factor |
| or | function-invocation |

*constant-id*

|  |  |
|---|---|
|  | id |

*set-constructor*

|  |  |
|---|---|
|  | [ ] |
| or | [ set-item {, set-item}$^0$ ] |

*set-item*

|  |  |
|---|---|
|  | expression |
| or | expression .. expression |

*function-invocation*

|  |  |
|---|---|
|  | function-id |
| or | function-id ( actual-parameter {, actual-parameter}$^0$ ) |

Expression factors are the elements in expressions which represent values. For example, in the expression

$$a + b$$

$a$ and $b$ are factors and + is an operator. There are nine classes of expression factors.

(1) variables (e.g. $a$, $a\uparrow$, $a$[ 1 ], a.b)

These yield the value stored in the variable. Note that variable operators denoted by ↑, [ ] and . are performed before any expression operators such as +, −, *, /.

(2) numbers (e.g. 123, 12.34, 12e34, 12.34e56)

(3) strings and single characters (e.g. 'abc', 'a')

(4) constant identifiers

(5) **nil**

**Nil** is a keyword which designates a pointer value which means "this pointer variable does not contain a pointer value".

(6) (expression)

Parenthesis are used according to the standard rules of algebra to force the evaluation of an expression to take place in a particular order. For example, in the expression

$$a + b * c$$

if it was required to evaluate the $a + b$ first, rather than the $b * c$ which is the normal order, then

$$(a + b) * c$$

could be used.

(7) **not**

The **not** operator is a unary operator. It takes a *Boolean* operand and yields a *Boolean* result. If the value of the operand is *true* it yields *false*, and if the value of the operand is *false* it yields *true*.

(8) function invocation

This specifies the activation of a function and the actual parameters for that activation of the function. The value of this factor is the value returned by the function. Reference section A.4.5 describes passing parameters to functions and returning values from functions.

(9) set constructor

A list of set-items enclosed in square brackets ( [] ) is a set constructor. The set-items may specify individual elements in the set or ranges of elements.

*examples*:

*a*
*a*[*1*]
*a*↑
*a*.*b*
*a*↑.*b*
(*a* + *b*)
**not** *true*
**not** (*a* **or** *b*)
*f*( *x*, *y*, *z*, *t* )
[ *red*, *green* ]
[ *a*..*5*, *29* ]
[ *a*, *b*, *c*..*d* ]

### A.5.3 Control Statements

---

*control-statement*
if-statement
or case-statement
or while-statement
or repeat-statement
or for-statement
or with-statement
or goto-statement

---

Control statements are used to control the execution of a Pascal program in four ways.

(1) The **if-then-else** and **case** control statements choose between alternate actions to be executed.

(2) The **repeat-until, while-do** and **for** control statements cause some action to be executed repeatedly.

(3) The **goto** statement causes execution of statements to continue at a new place in the program.

(4) The **with** statement makes the fields within specified record variables accessible using only the field-name.

The **if-then-else, repeat-until** and **while-do** statements all use a *Boolean* expression, called the *control expression,* to determine their action. The statements (actions) which are caused to be executed by control statements are called *object statements.*

### A.5.3.1 IF Statement

---

*if-statement*
  **if** control-expression **then**
    statement
or **if** control-expression **then**
    statement
  **else**
    statement

*control-expression*
  expression

---

There are two forms of the **if** statement. The first form of the **if** statement performs its object statement if the value of the control expression is *true*.

*examples*:

**if** $a < 5$ **then**
  $a := a + 1$

**if** $x$ **in** $y$ **then**
  **begin**
    $x := 1$;
    $y := []$;
  **end**

The second form of the **if** statement performs the first object statement (called the "then part") if the value of the control expression is *true* and the second object statement (called the "else part") if the value of the control expression is *false*. Note that there is *no semicolon* (;) separating the first object statement from the keyword **else.**

For both forms of the **if** statement, the control expression must be of type *Boolean*. The object statements must each be single statements. If several statements are required as the object, they may be enclosed in a **begin-end** pair.

*examples*:

```
if 9 < y then
   y := 22
else
   y := 0

if test( y, j ) then
   begin
      fixup( y, j );
      writeln( y, j );
   end
else
   writeln( 'ok' )
```

It is often appropriate to use the **if** statement to select between one of many choices in the following way:

```
if expression-1 then
   statement-1
else if expression-2 then
   statement-2
else if expression-3 then
   statement-3
...
else if expression-n then
   statement-n
```

This construct will execute the action for the first *true* condition and then leave the **if** construct.

When **if** statements with **else** parts are nested a syntactic ambiguity may arise. The **else** part in the following statement could apply to either **if** statement, and is therefore called a "dangling else".

```
if expression then
   if expression then
      statament
   else
      statement
```

The rule for resolving the ambiguity is that above construct has the meaning of the following non-ambiguous **if** statement. The **else** is applied to the closest nested **if** statement.

```
if expression then
   begin
      if expression then
         statement
      else
         statement
   end
```

### A.5.3.2 CASE Statement

---

*case-statement*
    **case** selector-expression **of**
      {case-label-list : statement;}$^0$
      ⟨case-label-list : statement⟩
    **end**

*selector-expression*
    expression

*case-label-list*
    constant {, constant}$^0$

---

The **case** statement permits selection of one of many actions. Each possible action consists of one statement. Multiple statements may be enclosed in a **begin-end** pair. Each action is identified by one or more case-label values which are constants. If the value of the selector-expression is equal to the value of a case-label on a statement, then that statement is executed. If no case-label matches the value of the selector-expression then an error occurs. All case-labels must be unique over each **case** statement. Each time the **case** statement is executed exactly one of the actions will be chosen and performed. *All case-labels must be of compatible type to the* selector-expression. The selector-expression *must* be of ordinal type.

*example:*

```
case character of
   'a' : Process( y, y );
   'b', 'c' : ; {null action}
   'd' :
        begin
          a := 1;
          b := 2;
        end;
   'e' : Process( n, y );
end
```

### A.5.3.3 WHILE Statement

*while-statement*
**while** control-expression **do**
statement

The **while** statement performs its object statement repeatedly while the value of the control expression is *true*. If the control expression is initially *false* then the object statement will not be executed at all. The control expression must be of type *Boolean*. The object statement consists of a single statement. Multiple statements may be enclosed in a **begin-end** pair.

Note that if the statement part does not take some action which affects the value of the control expression, the statement will repeat endlessly. This situation is called an infinite loop.
The following is an example of a properly terminating **while** statement:

```
i := 1;
while i <= 10 do
  begin
    writeln( i );
    i := i + 1;
  end
```

### A.5.3.4 REPEAT Statement

---

*repeat-statement*
**repeat**
{statement;}$^0$
statement
**until** control-expression

---

The **repeat-until** statement executes its object statements repeatedly until the value of the control expression is *true*. The control expression must be of type *Boolean*. Note that there may be multiple object statements; a **begin-end** pair is *not* necessary. Also note that the object statements are always executed at least once since the control expression is evaluated after each iteration of the loop. This is different from the **while** statement which may not execute its object statement at all, since the control expression is evaluated before each iteration of the loop.

*example*:

```
repeat
   y := f( x);
   x := x + deltax;
   writeln( x, y );
until x >= limit
```

### A.5.3.5 FOR Statement

*for-statement*
  **for** control-variable := initial-value **to** final-value **do**
    statement
or **for** control-variable := initial-value **downto** final-value **do**
    statement

*control-variable*
  id

*initial-value*
  expression

*final-value*
  expression

The **for** statement executes its object statement once for each value in a sequence. The values in the sequence run from the specified initial value to the specified final value. The control variable contains the value of the current element in the sequence. The value of the control variable is undefined when the **for** statement terminates. The control variable must be locally declared as a variable of ordinal type. It may not be inherited from an enclosing scope and it may not be a value parameter or a **var** parameter. The control variable may not be modified during the execution of a **for** statement.

When the **for** statement is executed the initial-value expression and the final-value expression are evaluated first. If the loop is to be executed at least once then the initial value is assigned to the control variable. At the end of each iteration a new value for the control variable is calculated, if there is to be another iteration. When the **to** keyword is specified the *succ* function is applied to the value of the control variable to compute the new value for each iteration. When the **downto** keyword is specified the *pred* function is used instead. The object statement is executed until the value of the control variable reaches the final value.

*examples*:

```
for i := 1 to 10 do
   writeln( i )

for j := red downto blue do
   begin
      match( j, k );
      writeln( ord( j ) );
   end
```

The **for** statement:

```
for i := expr1 to expr2 do
   statement
```

is equivalent to:

```
initial := expr1;
final := expr2;
if initial <= final then
   begin
      i := initial;
      statement;
      while i <> final do
         begin
            i := succ( i );
            statement;
         end
   end
```

where *initial* and *final* are local variables of the base type of *i*.

The **for** statement:

> **for** *i* := *expr1* **downto** *expr2* **do**
>    statement

is equivalent to:

> *initial* := *expr1;*
> *final* := *expr2;*
> **if** *initial* >= *final* **then**
>    **begin**
>       *i* := *initial;*
>       statement;
>       **while** *i* <> *final* **do**
>          **begin**
>             *i* := *pred( i );*
>             statement;
>          **end**
>    **end**

where *initial* and *final* are local variables of the base type of *i*.

### A.5.3.6 WITH Statement

---

*with-statement*
  **with** var {, var}$^0$ **do**
    statement

---

The **with** statement executes its object statement with additional variables available to it. Several variables may be specified in a **with** statement; they must all be of type record. Within the object statement, fields in the record variables named in the **with** statement may be referred to by field name only. If a variable has the same name as a field in a record variable which was specified in a **with** statement, the variable will be inaccessable within the **with** statement. If the same name exists in two record variables which are named in the same or nested **with** statements then the latest definition applies.

*example*:

```
var
  a : Boolean;
  b :
    record
      a : integer;
    end;
...
with b do
  a := 1; {refers to b.a}
```

### A.5.3.7 GOTO Statement

---

*goto-statement*
  **goto** label

---

The **goto** statement is a primitive, low-level mechanism for controlling the flow of execution of a program. It is very unrestricted and can easily render programs excessively complex; however, in Pascal there are some instances where it is necessary. When the **goto** statement is executed the flow of control transfers to the point in the program designated by the label specified on the **goto.** Since labels have the same scope rules as identifiers, the **goto** can transfer out of procedures and functions.

*example*:

```
program gotodemo( output );

  label
    10, 20;

  var
    i : integer;

  begin
    i := 1;
  10:
    if i > 100 then goto 20;
      writeln( i );
      i := i + 1;
    goto 10;
  20:
  end.
```

# Reference Section B

# Predefined Identifiers

Every Pascal program has certain predefined identifiers available to it. Conceptually, they are defined in an imaginary outer block which encloses the entire program, and are referred to as *standard* identifiers.

## B.1 Predefined Labels

There are no predefined labels.

## B.2 Predefined Constants

### B.2.1 Maxint (Largest Integer)

*Maxint* is a predefined constant whose value is the largest integer magnitude representable by the computer hardware. It is implementation defined. The integer operators +, −, *, **div** and **mod** are guaranteed to be implemented correctly when the absolute values of the two operands and the result are all less than or equal to *maxint*.

## B.3 Predefined Types

### B.3.1 Integer

The range of values for the data type *integer* is

{ *−maxint*, *−maxint+1*, ..., *−1*, *0*, *1*, ..., *maxint−1*, *maxint* }.

The following operators are defined for the type *integer*:

| | |
|---|---|
| $+$ | addition, unary identity |
| $-$ | subtraction, unary negation |
| **div** | integer division |
| **mod** | integer remainder |
| $*$ | multiplication |
| $=$ | |
| $<>$ | |
| $<$ | |
| $<=$ | relational |
| $>$ | |
| $>=$ | |

### B.3.2 Char

The range of values for the data type *char* is at least the upper case letters (A-Z), the digits (0-9) and the space character, plus any additional characters provided by the character set underlying the implementation. The values of the data type *char* are therefore implementation defined.

The ordinal positions of the characters within the character set are implementation defined. Within each of the three subsets, A-Z, a-z and 0-9, the characters will be in alphabetical order but only the digits, 0-9, are guaranteed to be in consecutive ordinal positions.

The relational operators are the only ones defined for the data type *char*.

### B.3.3 Boolean

The data type *Boolean* is defined by

> **type**
> *Boolean* = ( *false*, *true* );

*Boolean* is a particular case of an enumerated data type. The relational operators =, <>, <=, >=, <, > and **in** all yield *Boolean* values. The **if, while,** and **repeat-until** statements all require *Boolean* control expressions.

### B.3.4 Real

The data type *real* allows approximations of real (in the mathematical sense) numbers to be represented, that is, *real* values may have a fractional part (digits to the right of the decimal point). The following operators are defined for the data type *real*.

| | |
|---|---|
| + | addition, unary identity |
| − | subtraction, unary negation |
| * | multiplication |
| / | real division |
| = | |
| <> | |
| < | |
| <= | relational |
| > | |
| >= | |

Real values typically have their precision and magnitude limited by the computer hardware. See Reference Section G for the limitations on *real* numbers in Waterloo microPascal. Since *real* numbers are approximations one should not rely on the results of *real* operations being absolutely correct. Comparisons between *real* numbers for strict equality (=) are very likely to produce unexpected results. It is a safer practice to program in such a way that *real* comparisons are expressed as <= or >=.

### B.3.5 Text

The data type *text* is an enhanced version of the type

```
type
  text = file of char;
```

Variables of type *text* are referred to as *textfiles* and have special features beyond files of all other types including ordinary files of *char*.

Textfiles have the property that they may be divided into lines. This page, if stored in a computer, could be represented conveniently as a textfile.

The following special features are included in Pascal to facilitate the processing of textfiles:

(1) In order that a program can determine where lines end and new lines begin when reading a textfile, the function *eoln*( *f* ) is included. It returns *true* if the textfile *f* is at the end of a line and *false* otherwise.

(2) In order that a program can indicate the end of the current line when writing a textfile, the procedure *writeln*( *f* ) is included. It writes a new-line marker on the textfile *f*.

(3) Since only data of the base type of a file may be used in operations to the file (i.e. assigned to the file buffer variable) textfiles are restricted to character data. In order to enhance the usefulness of textfiles, the procedures *read* and *write* will convert internal representations of some data types to character data. This allows values of type *integer*, *Boolean*, string and *real* to be written out in human-readable format, and also allows numbers in human-readable format to be read by a Pascal program.

Reference Section B.5.7 describes the standard procedures and functions for file manipulation.

## B.4 Predefined Variables

### B.4.1 Standard Input and Output Files

The standard files *input* and *output* are external files. The following declaration is assumed automatically if they are mentioned in the program heading.

```
var
  input, output : text;
```

They are declared to be local to the main block as distinct from in the conceptual block enclosing the entire program. This means that the identifiers *input* and *output* cannot be redefined accidentally in the main block. These files are automatically initialized before program execution is started (i.e. *reset(input)* and *rewrite(output)* are executed) provided they are mentioned in the program heading.

The standard procedures and functions *get, read, readln, eof* and *eoln* assume the standard file *input* if the optional parameter specifying the file is omitted. The standard procedures *put, write* and *writeln* assume the standard file *output* if the optional parameter specifying the file is omitted.

## B.5 Predefined Procedures and Functions

### B.5.1 Mathematical Functions

| | |
|---|---|
| *sin*($x$) | returns the sine of $x$ radians |
| *cos*($x$) | returns the cosine of $x$ radians |
| *arctan*($x$) | returns the arctangent in radians of $x$ |
| *ln*($x$) | returns the natural logarithm of $x$ |
| *exp*($x$) | returns $e$ raised to the power of $x$ |
| *sqrt*($x$) | returns the square root of $x$ |

All of the above functions take either an *integer* or *real* parameter and always return a *real* result.

| | |
|---|---|
| *abs*($x$) | returns the absolute value of $x$ |
| *sqr*($x$) | returns $x*x$ |

Both of the above functions take an *integer* or *real* parameter and return a result of the same type as the parameter.

### B.5.2 Dynamic Variable Creation Procedures

*new( x )*

*New* takes a pointer variable, say *x*, as a parameter. It creates a variable of the type to which *x* is a pointer. The pointer to the new variable is returned in *x*. If *x* points to a variant record then the variable created by *new* will be capable of storing any of the variants (except when the following form of *new* is used).

*new( x, t1, t2, ..., tn )*

In the case where *x* points to a record with a variant part, a value for each tag field may be specified. This extra information may permit the compiler to make some space-saving optimizations.
The following rules apply:

(1) *New* does not assign the tag field values to the tag fields.

(2) The values correspond to consecutive tag fields starting with the first one in the record.

(3) Only the values specified in the parameter list to *new* may be assigned to the tag fields by the program.

(4) The same tag field values *must* be specified on an activation of *dispose* for the variable created by this form of *new*.

*dispose( x )*

*Dispose* takes a pointer value parameter (which was originally returned by *new*) for which no *dispose* has previously been done, and destroys the variable which is pointed to by the parameter.

*dispose( x, t1, t2, ..., tn )*

In the case where tag field values were specified to *new*, the same tag field values must be specified to *dispose*.

### B.5.3 Real to Integer Conversion Functions

*trunc( x )*

*Trunc* takes a *real* parameter and truncates it to an *integer* value.

*round( x )*

*Round* takes a *real* parameter and rounds it to the nearest *integer* value. If the parameter is zero or positive then *round( x )* is equivalent to *trunc( x + 0.5 )*; otherwise it is equivalent to *trunc( x − 0.5 )*.

If the result of either of the above functions is not in the range of values for the type *integer* then an error occurs.

### B.5.4 Functions for Ordinal Types

*ord( x )*

*Ord* takes an ordinal type parameter and returns an *integer* value which is the ordinal position of the parameter value within the set defined by the type of the parameter.

The ordinal position of the first element in an enumerated type is zero. The rest of the elements occupy consecutive positions. The ordinal position of an element of the type *integer* is the value of the integer. The ordinal positions of the elements of the type *char* are implementation defined. The ordinal positions of the elements of a subrange type are the same as the ordinal positions of the elements of its base type.

*chr( x )*

*Chr* takes an *integer* parameter and returns a value of type *char* which is the character at the ordinal position indicated by the parameter value. If no such character exists an error occurs. *Chr( ord( x ) )* = *x* is always *true* if the character *x* is defined.

*succ( x )*

*Succ* takes a parameter which is of ordinal type and returns the next element in the ordered set of values defined by that type. An error occurs when the parameter value is the last item in the ordered set (i.e. no successor to the parameter value exists).

*pred( x )*

*Pred* takes a parameter which is of ordinal type and returns the previous element in the ordered set of values defined by that type. An error occurs when the parameter value is the first item in the ordered set (i.e. no predecessor to the parameter value exists).

### B.5.5 Miscellaneous Functions

*odd( n )*

*Odd* takes an *integer* type parameter and returns *true* if the value of the parameter is odd and *false* otherwise.

### B.5.6 Data Transfer Procedures

*pack( source, offset, dest )*

*Pack* copies data from the parameter *source* to the parameter *dest* under the following rules.

(1) *Source* must be an array which is not packed.

(2) *Dest* must be an array which is packed.

(3) *Source* and *dest* must have identical constituent types.

(4) The number of elements copied is the number of elements in the array *dest*.

(5) The first element copied is *source[offset]* which is assigned to the first element of *dest*.

(6) The remaining elements are copied to corresponding consecutive positions.

*unpack( source, dest, offset )*

*Unpack* copies data from the parameter *source* to the parameter *dest* under the following rules.

(1) *Source* must be an array which is packed.

(2) *Dest* must be an array which is not packed.

(3) *Source* and *dest* must have identical constituent types.

(4) The number of elements copied is the number of elements in the array *source*.

(5) The first element copied is the first element in the array *source* which is assigned to *dest[offset]*.

(6) The remaining elements are copied to corresponding consecutive positions.

### B.5.7 File Manipulation Procedures and Functions

*eof( f )*
*eof*

*Eof* takes a file variable as a parameter. The parameter may be omitted, in which case the standard file *input* is assumed. An error occurs if the file variable was not initialized by an activation of procedure *reset* or *rewrite*.

*Eof( f )* returns *true* if the file *f* is positioned at the end-of-file (past the last element) and *false* otherwise. When *eof( f )* is *true*, *f*↑ is undefined.

A file *f* may be written (i.e. an activation of procedure *put( f )*) only when *eof( f )* is *true*. A file *f* may be read (i.e. an activation of procedure *get( f )*) only when *eof( f )* is *false*.

*eoln(f)*
*eoln*

*Eoln* takes a file variable as a parameter. The file must be a textfile. The parameter may be omitted, in which case the standard file *input* is assumed. An error occurs if the file variable was not initialized by an activation of procedure *reset*.

*Eoln(f)* returns *true* if the textfile *f* is positioned at the end of the current line, and returns *false* otherwise. When *eoln(f)* is *true* the value of *f*↑ is a space. *Eof(f)* and *eoln(f)* will never be *true* at the same time.

*get(f)*
*get*

*Get* takes a file variable as a parameter. The parameter may be omitted, in which case the standard file *input* is assumed. An error occurs if the file variable was not initialized with an activation of procedure *reset*.

If *eof(f)* is *true* prior to the activation of *get(f)* then an error occurs. Otherwise, the current position of the file is advanced to the next element and *f*↑ receives the value of the new current element. If no next element exists (i.e. end-of-file is encountered) then *eof(f)* becomes *true* and the value of *f*↑ is undefined. If *f* is a textfile and the new current element is a new-line marker then *eoln(f)* becomes *true* and the value of *f*↑ is a space.

*put(f)*
*put*

*Put* takes a file variable as a parameter. The parameter may be omitted, in which case the standard file *output* is assumed. An error occurs if the file variable was not initialized by an activation of procedure *rewrite*. An error occurs if *eof(f)* is not *true*.

The value of the file buffer variable *f*↑ is appended to the file *f*, and the value of *f*↑ becomes undefined.

*reset*( *f* )

*Reset* takes a file variable as a parameter and initializes the file for reading. The file is positioned at the beginning and an initial *get*( *f* ) is performed. After executing *reset*( *f* ) the buffer variable *f*↑ contains the value of the first element of the file. If the file is empty then the value of *f*↑ is undefined and *eof*( *f* ) is *true*.

*rewrite*( *f* )

*Rewrite* takes a file variable as a parameter and initializes the file for writing. All the elements are deleted and the file is then empty. *Eof*( *f* ) becomes *true* and the value of *f*↑ is undefined.

*read*( *f*, *v* )
*read*( *v* )

This form of *read* takes an optional file variable parameter and one data variable parameter. Forms of *read* which take several data variable parameters are subsequently defined in terms of this form. If the file variable is omitted then the standard file *input* is assumed. If the file variable was not initialized by an activation of *reset* then an error occurs. If *eof*( *f* ) is *true* prior to the execution of *read* then an error occurs.

If *f* is not a textfile then *read*( *f*, *v* ) is equivalent to

```
begin
   v := f↑;
   get(f);
end
```

If *f* is a textfile and *v* is of type *char* then the above definition also applies.

If *f* is a textfile and *v* is of type *integer* or *real*, then characters forming a number according to the syntax of Pascal are collected (after starting at the current character and skipping blanks and new-line characters). If a number is found and is of a type which is assignment compatible with *v* then the value of the number is assigned to *v*. The value of *f*↑ is the character immediately after the last character in the number which was found. If no number was found because end-of-file was encountered then *eof*( *f* ) becomes *true* and the value of *f*↑ is undefined. If no number was found and end-of-file was not encountered then an error occurs.

*read( f, v1, v2, ..., vn )*

When the first parameter to *read* is a file variable then this form of *read* is equivalent to

```
begin
  read( f, v1 );
  read( f, v2 );
  ...
  read( f, vn );
end
```

*read( v1, v2, ..., vn )*

When the first parameter to *read* is not a file variable then this form of *read* is equivalent to

```
begin
  read( input, v1 );
  read( input, v2 );
  ...
  read( input, vn );
end
```

*readln( f )*
*readln*

This form of *readln* takes a file variable as a parameter. The file must be of type *text*. The parameter may be omitted in which case the standard file *input* is assumed. *Readln( f )* is equivalent to

```
begin
  while not eoln( f ) do
    get( f );
  get( f );
end
```

*readln(f, v1, v2, ..., vn )*

When the first parameter to *readln* is a file variable then this form of *readln* is equivalent to

> **begin**
> *read(f, v1, v2, ..., vn );*
> *readln(f );*
> **end**

*readln( v1, v2, ..., vn )*

When the first parameter to *readln* is not a file variable then this form of *readln* is equivalent to

> **begin**
> *read( input, v1, v2, ..., vn );*
> *readln( input );*
> **end**

*write(f, v )*
*write( v )*

This form of *write* takes an optional file variable parameter and one data value parameter. Forms of *write* which take several data value parameters are subsequently defined in terms of this form. The file variable may be omitted in which case the standard file *output* is assumed. If the file variable was not initialized with a call to *rewrite* or if *eof( f )* is not *true* then an error occurs.

If *f* is not a textfile then *write(f, v )* is equivalent to

> **begin**
> *f↑ := v;*
> *put(f );*
> **end**

If *f* is a textfile and *v* is a *real, integer, Boolean, char* or **packed array of** *char* variable, a sequence of characters representing the data is constructed and put on the textfile.

If *f* is a textfile then the data value parameter may have a *field-width specifier* and be of the form

*v* : *w1*
or *v* : *w1* : *w2*

For example:

*write( a : 2, b : 2 : 4 );*

The field-width specifiers, *w1* and *w2*, may not be specified unless *f* is a textfile. Field-width specifiers may be used with all forms of *write* and *writeln*.

The field-width specifier *w1*, may be used with all types of parameters; it is used to indicate the number of characters to be written. Both field width specifiers *w1* and *w2* may be specified only with *real* parameters, in which case *w1* indicates the total number of characters to be written and *w2* indicates the number of digits to the right of the decimal point. If either *w1* or *w2* are negative then an error occurs.

The formats of the sequences of characters for the various types of data are given as follows:

*char*

minimum field width: 1
default field width: 1
format: The character is right-justified with blanks to the left to generate a field with the required width.

*Boolean*

minimum field width: 1
default field width: 5
format: The string "TRUE" or "FALSE", as indicated, is written. If the field width is 5 or greater then the string is right-justified within the field with blanks to the left. If the field width, *w1*, is 4 or fewer then the first *w1* characters of the string are written.

**packed array of** *char*

minimum field width: 1
default field width: length of string
format: If the field width is greater than the length of the string then the string is written right-justified in the field with blanks to the left. If the field width, *w1*, is less than or equal to the length of the string then the first *w1* characters in the string are written.

*integer*

minimum field width: 2
default field width: implementation defined
format: The value is represented with no leading zeroes and a minus sign to its immediate left if the quantity is negative. If the resulting string will not fit in the field then the field is expanded to the size of the string. Otherwise the resulting string is right-justified in the field with blanks to the left.

*real* (without *w2* specified)

minimum field width: implementation defined
default field width: implementation defined
format: The quantity is formatted in exponential notation which consists of:

(1) a minus sign (−) if the quantity is negative, otherwise a space,

(2) one digit,

(3) a decimal point (.),

(4) as many digits as the field width will permit (at least one),

(5) an "e",

(6) the sign (+, −) of the exponent,

(7) an implementation dependent number of digits of exponent.

*real* (with *w2* specified)

minimum field width: 4
default field width: implementation defined
format: The quantity is formatted in fixed-point format which consists of:

(1) as many blanks as required to right-justify the remainder of the representation of the number in the field,

(2) the digits required to the left of the decimal point; with the first character a minus sign (−) if the quantity is negative,

(3) a decimal point (.),

(4) *w2* digits.

When the representation of a *real* or *integer* quantity will not fit in the field width specified by *w1* the field will automatically be expanded. When *real* numbers are to be formatted in fixed-point representation, the field will be expanded if necessary to allow *w2* digits to the right of the decimal point.

*write( f, v1, v2, ..., vn )*

When the first parameter to *write* is a file variable then this form of *write* is equivalent to

```
begin
   write( f, v1 );
   write( f, v2 );
   ...
   write( f, vn );
end
```

*write( v1, v2, ..., vn )*

When the first parameter to *write* is not a file variable then this form of *write* is equivalent to

> **begin**
> *write( output, v1 );*
> *write( output, v2 );*
> ...
> *write( output, vn );*
> **end**

*writeln( f )*
*writeln*

This form of *writeln* takes a file variable as a parameter. The file must be of type *text*. The file may be omitted in which case the standard file *output* is assumed. If the file *f* has not been initialized by an activation of procedure *rewrite* or if *eof( f )* is not *true* then an error occurs.

*Writeln( f )* indicates that the current line on textfile *f* should be ended. Conceptually, a new-line marker is written on the file.

*writeln( f, v1, v2, ..., vn )*

When the first parameter to *writeln* is a file variable then this form of *writeln* is equivalent to

> **begin**
> *write( f, v1, v2, ..., vn );*
> *writeln( f );*
> **end**

*writeln( v1, v2, ..., vn )*

When the first parameter to *writeln* is not a file variable then this form of *writeln* is equivalent to

```
begin
  write( output, v1, v2, ..., vn );
  writeln( output );
end
```

*page( f )*
*page*

*Page* takes a file variable as a parameter. The file must be of type *text*. The parameter may be omitted in which case the standard file *output* is assumed. If the file *f* has not been initialized by an activation of procedure *rewrite* or if *eof( f )* is not *true* then an error occurs.

*Page* indicates that the next line of textfile *f* should begin at the top of a new page, if the representation of the textfile permits this.

# Reference Section C

# Reserved Words

The following words have special meaning in Pascal and may not be used as identifiers.

| | |
|---|---|
| **and** | **nil** |
| **array** | **not** |
| **begin** | **of** |
| **case** | **or** |
| **const** | **packed** |
| **div** | **procedure** |
| **do** | **program** |
| **downto** | **record** |
| **else** | **repeat** |
| **end** | **set** |
| **file** | **then** |
| **for** | **to** |
| **function** | **type** |
| **goto** | **until** |
| **if** | **var** |
| **in** | **while** |
| **label** | **with** |
| **mod** | |

# Reference Section D

# Delimiters

The following delimiters are symbols used in the Pascal language. Alternate representations are shown to the right of the preferred representation. They will be recognized on systems where the preferred representation is unavailable. (Waterloo microPascal recognizes alternate representations for { and } only.)

| | |
|---|---|
| + | |
| − | |
| * | |
| / | |
| = | **EQ** |
| <> | **NE** |
| < | **LT** |
| <= | **LE** |
| > | **GT** |
| >= | **GE** |
| . | |
| ↑ | |
| ( | |
| ) | |
| [ | (. |
| ] | .) |
| ; | ., |
| : | % |
| , | |
| := | .= or %= |
| .. | |
| { | (* |
| } | *) |

# Reference Section E

# Summary of Operators

The following table summarizes the operators of Pascal.

| symbol | operation | operand types left | operand types right | result type |
|---|---|---|---|---|
| := | assignment | ordinal | ordinal | none |
| | | real | real | none |
| | | real | integer | none |
| | | array | array | none |
| | | string | string | none |
| | | record | record | none |
| | | set | set | none |
| | | pointer | pointer | none |
| + | identity | none | integer | integer |
| | | none | real | real |
| | addition | real | real | real |
| | | real | integer | real |
| | | integer | real | real |
| | | integer | integer | integer |
| | setunion | set | set | set |

| symbol | operation | operand types left | right | result type |
|---|---|---|---|---|
| – | negation | none | integer | integer |
| | | none | real | real |
| | subtraction | real | real | real |
| | | real | integer | real |
| | | integer | real | real |
| | | integer | integer | integer |
| | set difference | set | set | set |
| * | multi-plication | real | real | real |
| | | real | integer | real |
| | | integer | real | real |
| | | integer | integer | integer |
| | set intersection | set | set | set |
| / | real division | real | real | real |
| | | real | integer | real |
| | | integer | real | real |
| | | integer | integer | real |
| **div** | integer division | integer | integer | integer |
| **mod** | integer remainder | integer | integer | integer |

| symbol | operation | operand types left | operand types right | result type |
|---|---|---|---|---|
| **and** | Boolean and | Boolean | Boolean | Boolean |
| **or** | Boolean or | Boolean | Boolean | Boolean |
| **not** | Boolean not | none | Boolean | Boolean |
| = | equality | ordinal | ordinal | Boolean |
| | | real | real | Boolean |
| | | real | integer | Boolean |
| | | integer | real | Boolean |
| | | pointer | pointer | Boolean |
| | | string | string | Boolean |
| | | set | set | Boolean |
| <> | inequality | ordinal | ordinal | Boolean |
| | | real | real | Boolean |
| | | real | integer | Boolean |
| | | integer | real | Boolean |
| | | pointer | pointer | Boolean |
| | | string | string | Boolean |
| | | set | set | Boolean |
| <= | less or equal | ordinal | ordinal | Boolean |
| | | real | real | Boolean |
| | | real | integer | Boolean |
| | | integer | real | Boolean |
| | | string | string | Boolean |
| | set inclusion | set | set | Boolean |

| symbol | operation | operand types left | right | result type |
|---|---|---|---|---|
| >= | greater or equal | ordinal | ordinal | Boolean |
| | | real | real | Boolean |
| | | real | integer | Boolean |
| | | integer | real | Boolean |
| | | string | string | Boolean |
| | set inclusion | set | set | Boolean |
| < | less | ordinal | ordinal | Boolean |
| | | real | real | Boolean |
| | | real | integer | Boolean |
| | | integer | real | Boolean |
| | | string | string | Boolean |
| > | greater | ordinal | ordinal | Boolean |
| | | real | real | Boolean |
| | | real | integer | Boolean |
| | | integer | real | Boolean |
| | | string | string | Boolean |
| **in** | set membership | ordinal | set | Boolean |

# Reference Section F

# Syntax Summary

## F.1 Notation

The following notation is used in the syntax definition of Pascal.

| | | |
|---|---|---|
| | $\langle abc \rangle$ | *abc* is optional |
| | $\{abc\}^0$ | *abc* may be repeated 0 or more times |
| | $\{abc\}^1$ | *abc* must be repeated 1 or more times |
| | *abc* \| *def* | choose *abc* or *def* |
| | *abc* | |
| or | *def* | choose *abc* or *def* |
| | **abc** | **abc** is a keyword |

The item being defined will be shown in *italics* and the definition of the item will follow, beginning on the next line and indented. The style of definition is based on a modification of Backus-Naur form.

### F.2 Basics

*digit*

"0" | "1" | "2" | ... | "9"

*letter*

"a" | "b" | "c" | ... | "z"

or "A" | "B" | "C" | ... | "Z"

*number*

$\{\text{digit}\}^1\langle.\{\text{digit}\}^1\rangle\langle\text{exponent}\rangle$

*exponent*

$\text{e}\langle + | - \rangle\{\text{digit}\}^1$

*id*

$\text{letter}\{\text{letter}|\text{digit}\}^0$

*string*

$'\{\text{any character}\}^1{}'$

## F.3 Programs and Blocks

*program*
program-heading;
block

*program-heading*
**program** program-name ⟨program-parameter-list⟩

*program-name*
id

*program-parameter-list*
( id-list )

*id-list*
id {, id}$^0$

*block*
declarations
**begin**
{statement;}$^0$
statement
**end**

## F.4 Declarations and Scope

*declarations*
⟨label-declarations⟩
⟨constant-declarations⟩
⟨type-declarations⟩
⟨variable-declarations⟩
⟨procedure-and-function-declarations⟩

### F.4.1 Labels

*label-declarations*
**label**
label {, label}$^0$;

*label*
{digit}$^1$

### F.4.2 Constants

*constant-declarations*
**const**
{id = constant;}$^1$

*constant*
⟨+ | −⟩ number
or ⟨+ | −⟩ id
or string

### F.4.3 Types

*type-declarations*
**type**
{id = type;}$^1$

*type*
type-id
or enumerated-type
or subrange-type
or ⟨**packed**⟩ array-type
or ⟨**packed**⟩ set-type
or ⟨**packed**⟩ file-type
or pointer-type
or ⟨**packed**⟩ record-type

### F.4.3.1 Simple Types

*type-id*
    id

*enumerated-type*
    ( id-list )

*subrange-type*
    constant .. constant

### F.4.3.2 Arrays

*array-type*
    **array** [ index-type {, index-type}$^0$ ] **of** type

*index-type*
    type-id
or    enumerated-type
or    subrange-type

### F.4.3.3 Sets

*set-type*
    **set of** enumerated-type
    **set of** subrange-type
    **set of** type-id

### F.4.3.4 Files

*file-type*
    **file of** type

### F.4.3.5 Pointers

*pointer-type*
    ↑ type-id

### F.4.3.6 Records

*record-type*
    **record**
        field-list
    **end**

*field-list*
    fixed-fields ⟨;⟩
or  fixed-fields; variant-part ⟨;⟩
or  variant-part ⟨;⟩

*fixed-fields*
    {id-list : type;}$^0$
    ⟨id-list : type⟩

*variant-part*
    **case** ⟨tag-name :⟩ tag-type **of**
        {variant;}$^0$
        ⟨variant⟩

*tag-name*
    id

*tag-type*
    type-id

*variant*
    variant-label-list : (field-list)

*variant-label-list*
    constant {, constant}$^0$

### F.4.4 Variables

*variable-declarations*
    **var**
        {id-list : type;}$^1$

### F.4.5 Procedures and Functions

*procedure-and-function-declarations*
    {procedure-or-function-declaration}$^{1}$

*procedure-or-function-declaration*
    procedure-heading;
      body;
or  function-heading;
      body;

*procedure-heading*
    **procedure** id ⟨formal-parameters⟩

*function-heading*
    **function** id ⟨ ⟨formal-parameters⟩ : type-id⟩

*body*
    block
or  directive

*directive*
    id

### F.4.5.1 Formal Parameters

*formal-parameters*
    ( parameter-group {; parameter-group}$^{0}$ )

*parameter-group*
    id-list : type-id
or  **var** id-list : type-id
or  procedure-heading
or  function-heading

## F.5 Executable Statements

*statement*
  ⟨label :⟩
    ⟨unlabelled-statement⟩

*unlabelled-statement*
  procedure-invocation
or  assignment-statement
or  control-statement
or  compound-statement

*compound-statement*
  **begin**
   {statement;}$^0$
   statement
  **end**

### F.5.1 Procedure Invocation and Parameters

*procedure-invocation*
  procedure-id
or  procedure-id ( actual-parameter {, actual-parameter}$^0$ )

*actual-parameter*
  procedure-id
or  function-id
or  variable
or  expression
or  write-parameters

*procedure-id*
  id

*function-id*
  id

*write-parameters*
  expression ⟨ ⟨ : expression⟩ : expression⟩

**F.5.2 Assignment Statement (Variables and Expressions)**

*assignment-statement*
    var := expression

**F.5.2.1 Variables**

*variable*
    id
or    subscripted-variable
or    variable-with-field-selection
or    indirectly-referenced-variable

*subscripted-variable*
    variable [ expression {, expression}$^0$ ]

*variable-with-field-selection*
    variable . field-name

*field-name*
    id

*indirectly-referenced-variable*
    variable ↑

### F.5.2.2 Expressions and Operators

*expression*
simple-expr
or simple-expr relational-operator simple-expr

*relational-operator*
=
or <>
or <
or <=
or >
or >=
or **in**

*simple-expr*
⟨+|−⟩ term
or simple-expr adding-operator term

*adding-operator*
+
or −
or **or**

*term*
factor
or term multiplying-operator factor

*multiplying-operator*
*
or /
or **div**
or **mod**
or **and**

### F.5.2.3 Expression Factors

*factor*

| | |
|---|---|
| | variable |
| or | number |
| or | string |
| or | constant-id |
| or | **nil** |
| or | ( expression ) |
| or | set-constructor |
| or | **not** factor |
| or | function-invocation |

*constant-id*

| | |
|---|---|
| | id |

*set-constructor*

| | |
|---|---|
| | [ ] |
| or | [ set-item $\{$, set-item$\}^0$ ] |

*set-item*

| | |
|---|---|
| | expression |
| or | expression .. expression |

*function-invocation*

| | |
|---|---|
| | function-id |
| or | function-id ( actual-parameter $\{$, actual-parameter$\}^0$ ) |

### F.5.3 Control Statements

*control-statement*

| | |
|---|---|
| | if-statement |
| or | case-statement |
| or | while-statement |
| or | repeat-statement |
| or | for-statement |
| or | with-statement |
| or | goto-statement |

### F.5.3.1 IF Statement

*if-statement*

    **if** control-expression **then**
      statement
or **if** control-expression **then**
      statement
    **else**
      statement

*control-expression*

    expression

### F.5.3.2 CASE Statement

*case-statement*

    ***case*** *selector-expression* ***of***
      {case-label-list : statement;}$^0$
      ⟨case-label-list : statement⟩
    **end**

*selector-expression*

    expression

*case-label-list*

    constant {, constant}$^0$

### F.5.3.3 WHILE Statement

*while-statement*

    **while** control-expression **do**
      statement

### F.5.3.4 REPEAT Statement

*repeat-statement*

    **repeat**
      {statement;}$^0$
      statement
    **until** control-expression

### F.5.3.5 FOR Statement

*for-statement*
  **for** control-variable := initial-value **to** final-value **do**
    statement
or **for** control-variable := initial-value **downto** final-value **do**
    statement

*control-variable*
  id

*initial-value*
  expression

*final-value*
  expression

### F.5.3.6 WITH Statement

*with-statement*
  **with** var {, var}$^0$ **do**
    statement

### F.5.3.7 GOTO Statement

*goto-statement*
  **goto** label

# Reference Section G

# Waterloo microPascal Users Guide

## G.1 Introduction

This section addresses issues specific to Waterloo microPascal and also contains the hardware dependent specifications.

## G.2 Run-time Error Detection in Waterloo microPascal

Waterloo microPascal is designed to provide useful diagnostic information in the case of run-time errors. The classes of run-time errors that Waterloo microPascal detects are:

- attempts to use a variable that has not been assigned a value,
- attempts to assign a value that is outside the declared range of a variable,
- array subscripting errors,
- attempts to use a **nil** pointer, or to use previously "disposed" memory,
- dynamic storage resources exhausted,
- run-stack overflow (for example, infinite recursion),
- control statement semantics: branching into an inactive **for** or **with** statement; no case match in a **case** statement.

In the case of any run-time error, Waterloo microPascal displays:

- the name of the variable involved (if any),
- the source-file line where execution was taking place when the error occurred,

## G.3 Language Supported By Waterloo microPascal

Unlike most other programming languages there is no official standard for Pascal. The Pascal User Manual and Report, Second Edition (Kathleen Jensen and Niklaus Wirth, Springer-Verlag, New York, 1974, ISBN 0-387-90144-2) was the original definition of the Pascal language. An international standardization effort is now underway. In the absence of such a standard, Waterloo microPascal is an implementation of the language described herein, which is based on the draft proposals produced by the Pascal standardization effort. The language is very close to what is described by Jensen and Wirth.

## G.4 Implementation Defined Attributes

(a) *Maxint* is defined to be 32,767 (that is, 2**16−1).

(b) The largest *real* value is approximately 1.7e+38.

(c) The smallest positive *real* value (machine epsilon) is approximately 2.9e−39.

(d) The data type *char* is defined to be all 128 ASCII character codes. This includes all upper and lower case letters, and all special characters.

(e) Sets may have a maximum of 256 elements. The ordinal values of the elements must be in the range 0..255.

(f) The default field widths used by procedures *write* and *writeln* are 7, 5, and 15 for *integer, Boolean* and *real,* respectively.

(g) The default number of decimal places displayed by *write* or *writeln* for a floating-point number (exponential notation) is 8.

## G.5 Implementation Dependent Attributes

(a) The only procedure directive in Waterloo microPascal is the *forward* directive (in particular, there is no *external* directive).

(b) There are some additional standard functions/procedures (see Reference Sections G.10 and G.11)

(c) Attempting to write onto a file that was "opened" for reading will result in a run-time error.

(d) The operands of a binary operator are evaluated left-to-right so that in the following expression, the left-oprnd-expression is evaluated first:

left-oprnd-exprn operator right-oprnd-exprn

(e) *Boolean* expressions are always evaluated completely (there is no partial expression evaluation optimization).

(f) The order of evaluation and binding of function and procedure actual parameters is strictly left-to-right.

(g) The effect of resetting or rewriting a standard file is the same as for any other file.

(h) Data items of the type *char* are stored in one byte.

(i) *Integer*, enumerated types, and subrange types are stored in two bytes.

(j) Data items of the type *real* are stored in five bytes.

(k) Declaring a structured type to be **packed** has no effect on the internal representation.

## G.6 File I/O Considerations

Waterloo microPascal allows a more general form of the standard functions *reset* and *rewrite*. For example,

*reset*( *x*, 'testdata' )

would open the file named "testdata" for input. It is also possible to use

*reset*( *x*, *filename* )

where *filename* is a **packed array of** *char* containing a filename.

### G.7 Character-set Extensions

Because some of the special characters used in the Pascal language may not be available on some I/O devices, Waterloo microPascal recognizes the following escape sequences:

| | | |
|---|---|---|
| (* | ... | left brace bracket ({) |
| *) | ... | right brace bracket (}) |

### G.8 Miscellaneous Considerations

Identifiers and keywords are case-insensitive (that is, $A=a$ always yields *true*).

Waterloo microPascal should not be used with source files that have a record length greater than 128 bytes.

Sequence numbers are not part of the Pascal language; thus, Waterloo microPascal will not accept programs that have them.

### G.9 Restrictions

In order to ensure the security of the run-time environment of Waterloo microPascal (that is, to allow complete run-time semantic checking), the restriction that file types may not contain file types or pointer types is enforced.

The semantics of variant records are not checked at execution time.

Pack and unpack are not implemented.

Passing procedure or function names as parameters is not supported.

### G.10 The Interactive Debugger

An integral part of Waterloo microPascal is an interactive debugger. It may be used to trace program execution, temporarily suspend program execution and examine and/or change program variables.

The debugger is invoked when:

(1) the break key is hit,

(2) the standard procedure pause is executed, or

(3) a run-time error is detected.

When the debugger is invoked, microPascal displays the source line at which the program was executing, and prompts for a command. The user may proceed either by pressing "return", which simply continues as if the debugger had not been invoked, or may enter a debugger command. These commands are described in the next section.

*Debugger Commands*

All debugger commands are represented by a single character.

*Quit*

Syntax: q

Description: The quit command terminates execution and returns the user to the editor.

*Continue*

Syntax: c

Description: The continue command terminates the debugger, and resumes execution of the program at the point where the debugger was invoked.

*Execute*

Syntax: e <statement>

Description: The given statement is executed. It may be any Pascal statement that is legal in the current scope context. In particular, "writeln" may be used to examine the contents of variables, and assignment statements may be used to change values.

*Single-step*

Syntax: s

Description: The single-step command places microPascal in a state such that each source statement is displayed, but not executed until the "return" key is pressed. This allows the user to trace the execution of a program. Single-step mode may be terminated either by allowing the program to end normally, or by entering another debugger command. If a run-time error occurs while in single-step mode, the debugger is invoked as usual.

*Where-am-I?*

Syntax: w

Description: The "where-am-I?" command simply reviews what state microPascal is in (i.e single-stepping, break'ed, etc. ), and displays the source line where the program is currently executing.

## G.11 Additional Standard Functions and Procedures

address( variable )

The parameter to *address* may be of any type, and must designate a variable. A variable designator is a variable identifier, a subscripted variable, a field-selection variable or an indirectly-referenced variable. *Address* returns the machine address of the storage allocated to that variable. The address is returned as an *integer* (i.e. *address* is of type *integer*).

pause

The *pause* procedure may be used to invoke the interactive debugger under program control.

peek( machine_address )

The *peek* function takes an *integer* parameter and returns an *integer* value which is the contents of the byte specified by the parameter.

poke( machine_address, value )

> The *poke* procedure requires two *integer* parameters. The value of the second parameter is stored in the byte at the address specified by the first parameter.

sysfunc( machine_address, parm1, parm2, ..., parm10 )

> The *sysfunc* function is used to invoke assembler-language or library routines. It may have from one to eleven parameters. The first parameter must be an *integer* value which is the address of the routine to be invoked. The other parameters (if any) are passed to the invoked routine, and must be of ordinal type. The *address* function described above may be used to facilitate passing non-ordinal variables (by address). The parameters to the invoked routine are passed using the same conventions as the Waterloo System Library (as described in the System Library Reference *Manual). Sysfunc returns an integer value (i.e. the value returned* by the invoked routine is considered to be an *integer* value by microPascal).

sysproc( machine_address, parm1, parm2, ..., parm10 )

> Procedure *sysproc* operates in a similar fashion to *sysfunc*, except that no value is returned from the invoked routine.

Waterloo microPascal is an interpretive implementation of the Pascal language. It is accomplished by Waterloo microEdit—a full-screen text editor. This manual assumes familiarity with microEdit.

This document consists of two sections: a tutorial introduction and a reference manual. The tutorial introduction introduces the features of the Pascal language by a series of simple examples accompanied by notes. The reference manual defines the Pascal language and also explains specific features of Waterloo microPascal.

## Language Supported

The Waterloo microPascal implementation corresponds closely to **Pascal User Manual and Report, Second Edition** (Springer-Verlag, 1974) and the interim draft standards being produced by the international standardization effort.

## Enhancements and Features

- An interactive debugger allows single-step operation, break-points and interactive examination of variables at execution-time
- Peek and poke procedures allow direct access to the user memory, including the screen
- Reset and rewrite allow the specification of an actual filename as their second parameter
- Lazy I/O is a feature permitting keyboard and screen I/O to behave in an intuitive way for interactive programs

DISTRIBUTED BY

Howard W. Sams & Co., Inc.

4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

$10.95/21905

ISBN: 0-672-21905-0