Appendix A An introduction to Oberon-2

A.1 Overview of Oberon-2

The programming language we use for the examples throughout the book is Oberon-2. Oberon-2 was designed by Niklaus Wirth, well known as the designer of many programming languages, in particular Pascal, one of Oberon-2's predecessors. Oberon-2 is an object-oriented programming language and has features which allow us to use the important software engineering principles of abstraction, data hiding (which is central to data abstraction and object-oriented design), problem decomposition and modularisation. Also important for teaching, Oberon-2 has a very simple syntax, and is a small language, even though the concepts it implements are central to advanced software engineering.

We will work through the basic features of Oberon-2 in this appendix, using simple programs to illustrate each new idea. The more complex features which allow abstraction, data hiding and modularisation are dealt with in detail in the main body of the book. The treatment of Oberon-2 here is particularly suitable for those who have little or no experience of programming. It is suggested that beginners read this appendix immediately after Chapter 2.

We will assume that you have a computer, a Oberon-2 compiler and you know how to type in, compile and run a program. The examples in the book are implemented in POW! Oberon-2 for Windows, on a PC. Those of you using other Oberon-2 compilers may find that the ProgMain procedure is not required, and you can simply leave it out of the examples.

We start straight away with the problem of displaying a simple message on the terminal screen – displaying messages or results in some way or another is very important, since there is little point in writing a program to calculate the answer to life, if there is no way of finding out the result!

```
MODULE First;
   (* Display an important message on the terminal. *)
IMPORT Out;
```

```
PROCEDURE ProgMain*;
BEGIN
   Out.Ln;
   Out.String("Hello Oberon-2 World!");
   Out.Ln
END ProgMain;
END First.
```

This program (when compiled, linked and run) gives the following output:

```
Hello Oberon-2 World!
```

Now we know what the program does, let's look inside to see how the internal workings display a message. Each line in the program has a particular meaning and use, so we will examine each in turn.

The program commences with the line

```
MODULE First;
```

which states the name of our program or *module* – **`First`**. In Oberon-2 all programs consist of one or more modules; this program is quite simple and so has only one module.

Next we have a piece of text enclosed by **`(*`** and **`*)`**. This is a comment: comments are used to describe what a program does, or how it does it, to a human reader. They are ignored by the compiler, and play no part in the program itself – they are simply a means of conveying information to the reader. The use of comments is obligatory to all rational programmers. Just bear in mind that any reader of your program is unlikely to be psychic and may not be able to work out what your program is meant to do solely from the code. Additionally, as you write programs to solve more complex problems, you will find that you need the comments as much as anyone else, to remind you how your own code works. Oberon-2 allows comments to be placed in just about any position inside a program, and to extend over multiple lines. Some examples of commented code are given below.

```
Out.String(" the total is "); (* Print message. *)
(* Calculate the new value for total. *)
total := total + 1;
```

We are also allowed to *nest* comments inside each other. This can be useful if we want to place comment brackets around a section of code to prevent it being executed. For example,

```
number := 5;
(* By-pass reading in of number
Out.String("enter a number : ");
In.Int(number); (* Get number from user. *)  *)
square := number * number;
```

In this program fragment, the **`Out.String`** and **`In.Int`** statements are commented out and will not be executed when the program is run.

Now we come to a feature of Oberon-2 which is very important for software engineering – the *import* statement. The line

```
IMPORT Out;
```

is an example of an import statement. Oberon-2 is a language which is designed to allow existing problem solutions to aid in the solving of new problems. One way of doing this with *library modules*. Library modules can be thought of as a lending library, from which a program can *borrow* ready-made problem solutions (usually called procedures) by using import statements. These solutions are really sets of Oberon-2 instructions, which perform a specific task or tasks.

A typical Oberon-2 program will contain imports from several library modules. The import statement in **`First`** is a request to the standard library **`Out`**. The import statement must come near the top of the program text, so that the required procedures are imported before any attempt is made to use them. Every implementation of Oberon-2 provides a set of standard library modules, which contain procedures for solving some very common problems, such as getting data into a program, printing out results and performing mathematical calculations. In addition, the programmer can create his or her own libraries of procedures, which can be reused in many different programs.

A single Oberon-2 program can be divided up into several, separate modules which are eventually joined together to make a single executable program. One module has a special role as the main module of the program, and will be executed first. Since the main module itself can contain several procedures, one needs to be designated the main procedure, which will be executed first. The main or first procedure in a program is always called **`ProgMain`**, and the line:

```
PROCEDURE ProgMain*;
```

indicates its start. The rest of the statements, up to the line

```
END ProgMain;
```

are the body of procedure **`ProgMain`**, and these are the statements which will be run when the main module is executed. We will look at the meaning of the **`*`** later.

We have now reached the line consisting only of the word **`BEGIN`**. Although this seems a small and inconsequential statement it is in fact rather important, since it marks the start of the main workings of the main procedure **`ProgMain`** – the part which does the job of printing out the message using the imported procedures from library Module **`Out`**. The statements in the procedure body are executed sequentially when the program is run.

The first statement inside the procedure body, **`Out.Ln`**, is a use of the procedure, **`Ln`**, imported from library module **`Out`**. **`Ln`** sends a newline command to the terminal screen, causing the cursor to move down from its current position to the beginning of the next line. Notice that we must preface each use of the procedure **`Ln`** by the name of the library module

from which it comes. This is true for every procedure borrowed from every library. You may consider this to be a waste of typing, but the reason for it is to prevent the compiler being confused by name clashes. For example, module **Out** contains an **Ln** command which sends a newline to the terminal, but another module, **Printer** say, might also contain an **Ln** command to send a newline to an attached printer, and both these libraries might be imported into the same program. These two **Ln** procedures will be very different from each other, so we prefix the library module name onto the procedure name (separated by a full stop) so that the compiler knows which one we are referring

The next statement uses procedure **String** (also imported from **Out**). **Out.String** displays a string of characters on the terminal screen, where the string is indicated by enclosing the characters to be displayed in either double or single quotes. Further examples of valid **Out.String** statements are:

```
Out.String("  Hello ");
Out.String("54000 black space ships.");
Out.String("  ");
Out.String('Answer "Yes" or "No" ');
```

The module **First** uses only **Out.Ln** and **Out.String** to produce the output shown earlier.

Examining program **First**, you should see that all the statements except **BEGIN**, the last **Out.Ln**, **END First**, and the comment, are terminated by a semicolon. This is not done simply to increase the amount of typing needed to write a program, but because semicolons are required by Oberon-2 to separate one statement or language block from another. The semicolons are therefore used as *statement separators*. It is difficult at first to understand when a semicolon is needed, but the general rule is that a semicolon is not needed before the first statement in a sequence or after the last statement in a sequence. Thus the word **BEGIN** and the last **Out.Ln** do not have semicolons after them. The rules of Oberon-2 do allow an extra semicolon to be inserted after the last statement of a sequence; this has the effect of inserting an extra, empty statement. Try leaving out each of the semicolons in turn in the **First** module, and seeing when the compiler complains.

Having statement separators allows us to place several statements on one line, such as:

```
Out.Ln; Out.String("Hello");
Out.Ln; Out.String("Oberon-2 World");
```

However, this is not generally considered to be good programming style. In fact, rather than putting several statements on one line, you should consider leaving blanks lines at strategic positions in your programs, to make them easier to read and understand.

You may still be wondering why the **END First** and the comment do not have semicolons at the end of them. The comment has its own delimiters, **(*** and ***)**; once the compiler recognises the first **(*** it ignores everything

until it sees the corresponding ***)**. The **END First** statement is the last statement of the module body, and also the last statement in the entire module. The syntax of Oberon-2 dictates that it must have the name of the module after it, and it must be terminated by a full stop. This full stop indicates the end of the entire module.

Before we go any further into the details of Oberon-2 let's take a quick look at how program **First** is laid out on the page. Note the way some lines are indented, and some blank lines have been left in the text of the program. The indentation is to highlight the structure of the program and make it easier to read. If we write **First** without any indentation:

```
MODULE First;
(* Display an important message on the terminal. *)
IMPORT Out;
PROCEDURE ProgMain*;
BEGIN
Out.Ln;
Out.String("Hello Oberon-2 World!");
Out.Ln
END ProgMain;
END First.
```

it isn't so easy to understand. You should always indent your programs to display their structure, and leave blank lines where necessary to divide up the program into logical parts. Try to use the styles of indentation and layout that we have used throughout this chapter, and which are given in detail in Appendix E. Indentation styles tend to vary very little between programmers (except in the size of the indent), and you will see very similar styles in other books on Oberon-2 (and, indeed, most other programming languages).

Exercises

1. Alter module **First** so that it displays your name on the screen.
2. Write a program which prints your name in a diamond pattern, as shown :

   Claire

   Claire Claire

   Claire

3. Find all the syntax errors in the following program:

```
MODULE Test;
(* Print a test message
PROCEDURE ProgMain*;
  Out.Ln
  Out.String("Testing, testing);
END ProgMain;
END Test
```

   Once you thing that you have found all the errors, type the program in exactly as it is, and use the Oberon-2 compiler to check if your were

right. Each time the compiler finds an error, correct it and recompile the program, until you have removed them all.

4. Which of the following are valid strings:

```
"abc"
"12 23  "34" 78"
"   forty-two!
"'hello', she said"
"please enter a number
'a dark, stormy night'
```

You can check if you're right by substituting each one for the string in the original **First** program.

5. What output does the following program produce?

```
MODULE Message;
(* demo output program
PROCEDURE ProgMain*;
BEGIN
  Out.String("Oberon-2 is a descendent of Pascal");
END;  That's all *)
END Message.
```

6. Write a program that prints out your initials in large block letters. Use a 6 by 6 grid for each letter, and print out 6 strings. Each string should consist of a row of asterisks interspersed with blanks.

A.2 Reserved words and syntax

We haven't yet explained why some of the words in **First** are in upper case, or why they are arranged in a certain way in the program. All the uppercase words are *reserved words* which have special meanings to Oberon-2. For example,

Reserved word	Description
MODULE	Indicates the start of a module.
BEGIN	Indicates the start of a procedure body.
END	Indicates the end of a module or a procedure.
IMPORT	Used to borrow procedures from a library module.

The reserved words of Oberon-2 are always written in uppercase. Unlike English, Oberon-2 is case-sensitive, so words like

```
BEGIN, begin, Begin
```

are taken as different words in Oberon-2, although they all have the same meaning in English. Appendix C gives all the reserved words of Oberon-2, which you have to avoid using as names for items in your own programs.

Just as English has rules of syntax which tell us how to construct correct sentences, so each programming language has its own syntax, which governs how we construct statements and programs. If we don't use the syntax correctly our statements and programs will not make any sense to the compiler, and will not therefore be executable.

The syntax of Oberon-2 insists on a certain ordering of parts inside a module or program. Thus, a program always commences with a **MODULE** statement, this is followed by any **IMPORT**s required, then any declarations of data items (which we will meet later in this appendix), the **ProgMain** procedure if it is the main program module, any other procedures, and finally the **END** statement.

Exercises

1. Which of the following are reserved words of Oberon-2?

```
STOP   end  ELSIF  CHAR    CH    OR
case   False       TRUE
```

2. Do you think the following program is correct Oberon-2? If not, why not? How would you alter it to make it correct? Type in your corrected version and see if you are right.

```
MODULE Test;
PROCEDURE ProgMain*;
IMPORT Out;
BEGIN
  Out.String("This is a test program.")
END
END Test.
```

A.3 Data handling

Most programs need data, which is manipulated by the statements in the module body to produce results of one form or another. This data comes in many forms, from weather pictures transmitted as a sequence of digits by an orbiting satellite, to a piece of text typed in by a user. Oberon-2 insists that we *declare* any data before we use it. Declaring data means that we state, usually at the top of a module or the main procedure, all the data items we are going to use in the module or procedure body. Each data item is declared by giving it a unique name by which it can be referred to in the program and stating what sort of data item it is. For example, if it is an array of real numbers or a character or an integer etc.

A.3.1 Constants

Let's consider how we might write a program which calculates the number of working hours in a typical week, and prints out the result. We will assume that a working day runs from 9am to 5pm, so that the number of working hours in a day is eight. Here is one way of using declared data in such a program:

```
MODULE Hours;
(* Calculate and display the number of working hours in
a week. *)

IMPORT Out;
PROCEDURE ProgMain*;
CONST
   WorkingDays = 5; (* Working days in week. *)
   DailyWorkHours = 8; (* Working hours in day. *)
   WeeklyWorkHours = WorkingDays * DailyWorkHours;
               (* Number of working hours per week. *)

BEGIN
   Out.String("Number of working hours in a week is: ");
   Out.Int(WeeklyWorkHours, 3);
   Out.Ln
END ProgMain;
END Hours.
```

This program illustrates the simplest form of data in Oberon-2, constants. Constants are data items which have a fixed value, for example, the integer 42 is a constant. **`WorkingHours`** declares three constants, using the reserved word **`CONST`**, in the declaration section of **`ProgMain`**. Each data item is given a name and this name is then used to refer to the data item throughout **`ProgMain`**. The program uses procedures imported from **`Out`** to print out the number of working hours in a week. Procedure **`Out.Int`** displays an integer number on the terminal: it must be given the integer number to display (**`WeeklyWorkHours`** in this example) and the minimum field width (number of spaces) in which to print this number (three spaces in this case).

We could have produced the same result as the above program without using declared constants:

```
MODULE Hours2;
(* Calculate and display the number of working hours in
a week. *)
IMPORT Out;
PROCEDURE ProgMain*;
BEGIN
   Out.String("Number of working hours in a week is: ");
   Out.Int(40, 3);
   Out.Ln
```

```
END ProgMain;
END Hours2.
```

Comparing this to the earlier version you should agree that the first program is far easier to understand. **Out.Int(40, 3)** could mean anything, but **Out.Int(WeeklyWorkHours, 3)** is much clearer. It would be even better to have declared the field width as a constant too, perhaps called **FieldWidth**, so that the **Out.Int** statement would look like this:

```
Out.Int(WeeklyWorkHours, FieldWidth);
```

This is also a good reason for choosing meaningful names for modules, procedures and data items.

The three constants used in **WorkingHours** are quite straightforward: **WorkingDays** and **DailyWorkHours** are given the values 5 and 8 respectively; **WeeklyWorkHours** is a constant expression, since it contains only constants, and will have the value 40. Here are some further examples of valid Oberon-2 constants and constant expressions:

```
pi = 3.1416;
FirstLetter = "A";
Greeting = "Good Morning";
TwoPi = 2.0 * pi;
```

A.3.2 Variables

Declaring constants in our programs is useful, but doesn't allow us to use data items whose values can change. For example, if we wish to write a program to read in a character from the terminal keyboard, we need to declare a data item in which to store the character. We need a data item whose value is allowed to change because we cannot know in advance which character the user of the program will type at the terminal, so we cannot set up the data item to be a constant.

Data items whose values are allowed to change are called variables; they are declared using the Oberon-2 reserved word **VAR**. Here is an example of how we might use a variable in a program which reads a single character from the terminal keyboard and prints it out again:

```
MODULE ReadChar;
(* Read character from the keyboard, and write out
again. *)
IMPORT In, Out;
VAR
   ch : CHAR;  (* the variable to read the character
               into *)
PROCEDURE ProgMain*;
BEGIN
   Out.String("Type a character and press return..  ");
   In.Char(ch);
```

```
   Out.Ln;
   Out.String("You typed the character ..  ");
   Out.Char(ch);
   Out.Ln
END ProgMain;
END ReadChar.
```

This program contains a declaration of one variable data item called **ch**. The value of **ch** becomes the character typed on the keyboard, which is read in by the procedure **Read** (**Read** reads a single character; **Out.Char** displays a single character). The **CHAR** after **ch** declares what class of data objects **ch** belongs to – the class of a data object is more usually called its *type*. Type **CHAR** means that **ch** is a character (rather than an integer or a real number etc.) and can therefore take the value of any valid character, such as "!" or "g". Most implementations of Oberon-2 use a set of characters called the ASCII character set. They are defined in Appendix B. The ASCII character set includes all the characters you would expect (upper and lowercase letters, digits, punctuation), and a few unusual ones.

The other thing to note about this program is that we are importing from another library, called **In**, which contains the procedures for reading data items into our programs. In this instance we use the procedure for reading in characters, **In.Char**.

The type of a data item is essential because it tells us two very important things about the item:

- The range of values which the data item can assume. For example, if we are using the ASCII character set there are 128 possible characters for a data item of type **CHAR**.
- The operations which can be performed on or with the data item. For example, addition and subtraction are valid operations for integer numbers, but not for characters.

Oberon-2 provides us with some ready-made classes or types for our data items which already have their range of values and operations defined. These ready-made types are as follows:

Type name	Meaning
CHAR	Characters, for example, a, !, F.
INTEGER	Integer numbers, for example, 0, 56, 23100, -4567
REAL	Real numbers, for example, 2.13, 0.0, -32.009, 2.3E2.

A.3.3 Identifiers

The names which we use when declaring constants and variables, such as **ch** and **WorkingDays**, are called *identifiers*. We also use identifiers for the names of procedures and modules, such as **In**, **Out**, **First** and **Out.String.** Oberon-2 places certain restrictions on the characters we can use in identifiers, and the way in which those characters are arranged. The Oberon-2 system you are using may have to place restrictions on the length of module names (since each module is stored as a file on the computer, and the operating system may restrict the length of filenames). Here are some valid identifiers:

```
ReadChar    newWord     STOP2
count20       dailyPayRate   taxcode
```

The following identifiers are illegal for the reasons given:

BEGIN This is a Oberon-2 reserved word and reserved words cannot be used as identifiers.

day temp Spaces are not allowed in identifiers.

night-temp Illegal character "-".

20stop An identifier is not allowed to start with a digit.

The valid characters for identifiers are: alphabetic, "a"– "z" and "A"– "Z", and numeric, "0"– "9" (although a numeric character cannot be the first character in an identifier). Most Oberon-2 implementations do not place any restrictions on the length of identifiers for procedure names and variables, but you should be sensible about how long (or how short) you make them.

As Oberon-2 is a case-sensitive language it is customary for programmers to take advantage of this by using both uppercase and lowercase characters in identifiers, to make their programs more readable. Here are some of the most commonly used ways of writing different kinds of identifiers:

- Module names, procedure and function names usually start with a capital letter, and every new word within the identifier starts with a capital letter. For example,

  ```
  ReadWord
  WriteTable
  ```

- Variables can either be entirely in lowercase or can start with a lowercase letter and have each word within the identifier starting with a capital letter. For example,

  ```
  newword
  dailyPayRate
  ```

  We use the latter method for variables in all our program examples.

- Constants are often entirely in uppercase, or can be defined in the same way as library and procedure names. For example,

  ```
  MAXSIZE
  ```

```
FieldWidth
```

We have chosen to use the latter method in our program examples.

See Appendix D for a more detailed description of the rules you should following when deciding upon identifier names.

A.3.4 Expressions and assignments

So far we have only described how to declare constant values and expressions, read a data value into a variable and write things out. This doesn't allow us to do more complex operations or calculations with data. Consider the task of creating a short program to read in a number from the user and print out its square and its cube.

The algorithm for this problem is given below:

```
Ask user to enter a number
Read in the number
Square the number
Print out the square
Cube the number
Print out the cube
```

Here is the Oberon-2 solution:

```
MODULE SqrCube;
(* This program reads in an integer number typed by the
user, and prints out its square and cube. *)
IMPORT In, Out;
PROCEDURE ProgMain*;
CONST
   FieldWidth = 6; (* Width of number display. *)

VAR
   number, square, cube : INTEGER;
BEGIN
   (* Get the number from the user. *)
   Out.String("Enter a number: ");
   In.Int(number);
   Out.Ln;

   (* Calculate the square and cube. *)
   square := number * number;
   cube := square * number;

   (* Display the results. *)
   Out.String("The square is:  ");
   Out.Int(square, FieldWidth);
   Out.Ln;
```

```
   Out.String("The cube is: ");
   Out.Int(cube, FieldWidth);
   Out.Ln
END ProgMain;
END SqrCube.
```

The following two lines,

```
square := number * number;
cube := square * number;
```

are examples of assignment statements. The right hand sides of these assignment statements are expressions. Note that the calculation of the cube uses the previously calculated square, which saves a multiplication. The **:=** can be read as *becomes* or *takes the value of*, thus square takes the value of **number*number**, and **cube** takes the value of **square*number**. Other examples of valid assignments and expressions are:

```
circumference := 2.0 * pi * radius;
answer := 6 * 7;
total := 0;
total := total + 1;
```

The final one of these calculates **total+1**, and assigns this value back to **total** itself, thus increasing the value of **total** by 1.

Exercises

1. Which of the following identifiers are invalid, and why?

helloworld	**11temp**	**REAL**
PrintList	**last item**	**sports-car**
USER	**END**	**"help"**
!stop	**input**	**a**

A.4 Simple data types

We have already said that Oberon-2 provides several ready-made data types, so we will now look at them in more detail.

A.4.1 Type INTEGER

Oberon-2's type **INTEGER** consists of integer numbers which may be positive, negative or zero. It does not include numbers with decimal points. The range of integers that can be used is restricted by the particular computer on which you are running Oberon-2. A 16-bit computer will typically provide a range of -32768 to +32767.

We can do all the expected operations on data items of type **INTEGER**:

+	addition
-	subtraction and negation

*****	multiplication
DIV	integer division
MOD	modulus/remainder

The operations **+**, **-** and ***** are straightforward, they act on integers to produce integer results, but why do we have **DIV** instead of the more obvious **/** operator? **DIV** is integer division and it always gives an integer result. The statement

```
x := y DIV z;
```

assigns to **x** the truncated result of dividing **y** by **z**. Consider the result of 5/2, which is 2.5. This is a real number and cannot be assigned to a variable of type **INTEGER** (we will meet type **REAL** later). However **5 DIV 2** produces an integer result, 2, which can be assigned to a variable of type **INTEGER**. It should also be remembered that division by zero is undefined on a computer (because computers have considerable trouble with the concept of infinity) and any attempt to do this will cause a program to fail as soon as the computer tries to execute the division.

The other strange operation for type **INTEGER** is called **MOD**, which is short for modulus.

```
x := y MOD z;
```

This gives the (integer) remainder of **y** divided by **z**. Here are some examples of **DIV** and **MOD**,

```
4 DIV 2 = 2
10 DIV 3 = 3
10 MOD 3 = 1
4 MOD 2 = 0
2 MOD 5 = 2
```

Let's consider a small problem where we might need to use integer data. Let us take the problem of calculating a person's annual income tax bill. First we must define the problem more clearly.

We want to calculate a tax payer's yearly tax bill, given his yearly income and the number of children he has. Assume that one third of the income (less any allowances) must be paid as tax. There is a personal allowance of £6000, and an allowance of £1000 per child. An initial algorithm for solving this problem will be:

```
Get annual income from user
Get number of children from user
Calculate total amount of allowances
Calculate total amount of taxable income
Calculate tax payable
Display tax payable
```

Get annual income and *Get number of children* are quite straightforward, consisting simply of displaying appropriate messages and reading in numbers. The three calculations are also simple:

Calculate total amount of allowances:

```
total allowances = (£1000 * number of children) + £6000
```

Calculate total amount of taxable income:

```
taxable income = income - total allowances
```

Calculate tax payable:

```
tax payable = 1/3 * taxable income
```

We can put all this together in a Oberon-2 program:

```
MODULE Taxes;
(* Program to calculate amount of tax payable per year
by a tax payer, given his yearly income and the number
of children he has.  The program calculates the tax
payable to be one third of the annual income after any
allowances have been deducted.  There is a personal
allowance of £6000 and an allowance of £1000 per child.
*)
IMPORT In, Out;
PROCEDURE ProgMain*;
CONST
   ChildAllowance = 1000; (* Allowance per child. *)
   PersonalAllowance = 6000;
   FieldWidth = 6; (* Number display width. *)
VAR
   income : INTEGER; (* Total yearly income. *)
   numberOfChildren : INTEGER;
   totalAllowances : INTEGER; (* Personal + children. *)
   taxableIncome : INTEGER; (* Income - allowances. *)
   taxPayable : INTEGER; (* Total tax payable. *)
BEGIN
   (* Get income and number of children from user. *)
   Out.String("Enter total yearly income (Eg 12500): ");
   In.Int(income);
   Out.Ln;
   Out.String("Enter number of children:  ");
   In.Int(numberOfChildren);
   Out.Ln;
```

```
   (* Calculate total allowances. *)
   totalAllowances := (numberOfChildren *
            ChildAllowance) + PersonalAllowance;
   (* Calculate taxable income. *)
   taxableIncome := income - totalAllowances;

   (* Calculate tax payable. *)
   taxPayable := taxableIncome DIV 3;
   (* Tell user tax payable. *);
   Out.Ln;
   Out.String("Tax payable is:  ");
   Out.Int(taxPayable, FieldWidth)
END ProgMain;
END Taxes.
```

Note the use of **DIV** to obtain an integer result, and how the algorithm description provides some of the comments for the final implementation in Oberon-2.

Note the use of parentheses (round brackets) in the assignment,

```
totalAllowances := (numberOfChildren * ChildAllowance)

                   + PersonalAllowance;
```

to show how the expression on the right-hand side should be evaluated.

You should always use parentheses to indicate the order in which parts of an expression should be evaluated. The parts of an expression within the innermost parentheses are always evaluated first. So, in the above example, **(numberOfChildren * ChildAllowance)** is evaluated first, and the result is added to PersonalAllowance, which is exactly what is required.

Without parentheses it can be difficult to determine the result of an expression. For example, what is the result of the following expression?

```
2 * 3* 4 + 5
```

If we evaluate it from left to right we get

```
2 * 3 = 6
6 * 4 = 24
24 + 5 = 29
```

But if we add brackets as follows:

```
2 * ((3 * 4) + 5)
```

the result is as follows (remember that the innermost parentheses are evaluated first):

```
3 * 4 = 12
12 + 5 = 17
```

```
2 * 17 = 34
```

a completely different result!

Most programming languages, including Oberon-2 have a default order of precedence for operators. Thus, if you leave the following expression without parentheses:

```
2 * 3 + 4 * 5
```

Oberon-2 gives precedence to the multiplication operators, so we obtain the result

```
6 + 20 = 26
```

However, since operator precedence varies from language to language you should **never** write expressions which rely upon it — always use parentheses to state the precedence of operators that you require in your expression, in a clear, unambiguous fashion. That way you will ensure that your expression are always evaluated in the order **you** want, rather than the order the compiler wants!

A.4.2 Type **REAL**

Many applications require real numbers, and Oberon-2 provides type **REAL** for such situations. Some examples of valid **REAL** numbers in Oberon-2 are:

0.0032

-5.6

8.

897.0

Note that there must always be at least one digit before the decimal point, but there don't have to be any digits after the decimal point.

Oberon-2 also allows us to represent real numbers using exponential notation. This can be very useful for representing very large or very small numbers. Here are some examples of exponential notation:

32.124E2	which is	3212.4
1E10	which is	10000000000.0
12.2E1	which is	122.0
2.3E-6	which is	0.0000023

Most of the operations available for **REAL** are the same as those for **INTEGER** except for the division operator which returns a real result:

+	addition
-	subtraction and negation

***** multiplication

/ real division, this gives a **REAL** result.

Modules **In** and **Out** respectively provide procedures for reading in and printing out real numbers, and there is also a module **Float** which contains mathematical functions such as sin, cos etc.

We need to use real numbers in many mathematical problems, for example, consider writing a program which will calculate the circumference and area of a circle when given its radius. The algorithm for this problem is quite simple:

```
Get the radius from the user
Calculate the circumference
Print out the result
Calculate the area
Print out the result
```

In Oberon-2:

```
MODULE Circles;
(* Given the radius of a circle, calculate its
circumference & area. *)

IMPORT In, Out;
PROCEDURE ProgMain*;
CONST
   FieldWidth = 6; (* Number display width. *)
   Pi = 3.1416;
VAR
   radius, circumference, area : REAL;

BEGIN
   (* Ask user to enter radius. *)
   Out.String("Enter radius of circle: ");
   In.Real(radius);
   Out.Ln;
   (* Calculate and display circumference. *)
   circumference := 2.0 * Pi * radius;
   Out.String("The circumference is:  ");
   Out.Real(circumference, FieldWidth);
   Out.Ln;
   (* Calculate and display area. *)
   area := Pi * radius * radius;
   Out.String("The area is:  ");
   Out.Real(area, FieldWidth);
END ProgMain;
END Circles.
```

Precision

We have to be very careful when using numbers (particularly real numbers, very large numbers, and very small numbers) on a computer, because their precision depends on the number of significant digits to which the computer can store them. When using a 32-bit word, seven significant digits is normal. This means that two numbers which differ in only the eighth significant digit will be indistinguishable to the computer. For example,

2000000.0 + 0.00000001

will give the result

2000000.0

You should never expect calculations done on a computer to be exact – and in complex calculations expect errors that are caused by rounding errors, which are accumulating at every stage of the calculation. When accuracy is needed you may have to increase the word length for the arithmetic and use type **LONGREAL** In addition, always carefully consider the order in which calculations are made. Errors can be cancelled out or compounded by choosing the appropriate order of operations.

A.4.3 Type **CHAR**

Oberon-2's type **CHAR** allows our programs to use character data. Characters are not stored as such inside the computer: instead each character is assigned a number, called its *ordinal value*, which is used to represent it inside the computer. Each character has a different ordinal value. Most implementations of Oberon-2 use a standard set of ordinal values called the ASCII set. Many different programming languages and computers represent characters using the ASCII representation. Appendix B contains the ASCII representations of the 128 characters of type **CHAR**, giving their ordinal values in decimal, octal (base 8) and hexadecimal (base 16).

If you look at the ASCII table you will see that the different characters are grouped together in sensible blocks, so that all the digits are together, as are the lowercase alphabet and the uppercase alphabet. Because of this, Oberon-2 allows us to use the relational operators to do direct comparisons between the characters within each grouping, thus

"A" < "B" < "C" < .... < "Z"

"a" < "b" < "c" < .... < "z"

"0" < "1" < "2" < .... < "9".

It is sometimes useful to know the ordinal value of a character, so Oberon-2 provides an operation, **ORD**, for doing this.

```
ORD("x") = 120

ORD("A") = 65
```

The ordinal values range from 0 to 127, one for each of the 128 characters, and **ORD** returns the values as **INTEGER** numbers. But remember, not all of these characters are printable, especially those with the lower ordinal values. We can also go in the opposite direction, starting with an ordinal value and obtaining the corresponding character, using an operation called **CHR**. For example,

```
CHR(120) = "x",
CHR(65) = "A".
```

CHR and **ORD** are mutually inverse, so

```
CHR(ORD(120)) = "x",
ORD(CHR("x")) = 120.
```

We can use ordinal values to write a program which reads a lowercase letter, and writes it out again in uppercase. How can we do this? If we look at the ASCII table, we can see that for each letter the ordinal value of the lowercase letter is 32 higher than that of the corresponding uppercase letter. For example, "A" has ordinal value 65, and "a" has ordinal value 92. Thus the assignment,

```
upperCh := CHR( ORD(lowerCh) - 32 );
```

where **upperCh** and **lowerCh** are of type **CHAR**, and **lowerCh** is a lowercase letter, will assign the corresponding uppercase letter to **upperCh**. Here is the algorithm:

```
Read a character
Convert to uppercase character
Display uppercase character
```

The complete program is very short:

```
MODULE ToUpper;
(* Read in a lower case letter and print it out again in
uppercase.*)

IMPORT In, Out;
PROCEDURE ProgMain*;
VAR
   lowerCh, upperCh : CHAR;
BEGIN
   (* Read in character. *)
   Out.String("Enter a lowercase letter:  ");
   In.Char(lowerCh);
   Out.Ln;
   (* Find corresponding uppercase character. *)
   upperCh := CHR( ORD(lowerCh) - 32 );
   (* Write out resulting character. *)
   Out.String("The character in uppercase is: ");
   Out.Char(upperCh);
```

```
   Out.Ln
END ProgMain;
END ToUpper.
```

You will probably have noticed that this program can't cope with invalid input. What happens if the user enters a space or a question mark instead of a lowercase letter? (Hint: look at the ASCII table in Appendix B.)

A.4.4 Relationships between types and type compatibility

Oberon-2 has a number of built-in types for representing numbers. Along with **INTEGER** and **REAL**, which we have already met, there are also **SHORTINT**, **LONGINT**, and **LONGREAL**.

All variables must be declared to be of a particular type so that the compiler knows how much to space to allocate them in the computer's memory. The amount of memory allocated to each variable of a particular type can differ between implementations of Oberon-2. On my system an **INTEGER** is stored in 2 bytes, a **SHORTINT** in 1 byte, and a **LONGINT** in 4 bytes. Thus a variable of type **SHORTINT** cannot hold such large integers as a variable of type **INTEGER**, which in turn cannot hold such large integers as a variable of type **LONGINT**. Similarly, **LONGREAL** variables are stored in more memory than **REAL** variable, and so a **LONGREAL** variable can store a real number to more precision (more significant digits) than a **REAL** variable. For most of what you do you are likely to need only **INTEGER** and **REAL**, but it is worth understanding how to use the other types in your programs.

You can find out the ranges of the integer types (that is, the minimum and maximum values that a variable of the type can have) on your particular implementation of Oberon-2 by using the built-in **MIN** and **MAX** functions. **MIN** and **MAX** expect to be given a type name, and will return the minimum (or maximum) value of that type. You can find out the value by printing it out, for example,

```
Out.Int(MIN(INTEGER), 1);
```

will print out (on my computer) -32768. (Note that the second parameter given to **Out.Int**, the 1, is the fieldwidth in which the number will be printed. If the value needs more than the fieldwidth given, it will be printed anyway).

I have run **MIN** and **MAX** for each of the integer types on my version of Oberon-2, and I get the following values:

	minimum value	maximum value
INTEGER	-32768	32767
SHORTINT	-128	127
LONGINT	-2147483648	2147483647

The built-in numeric types of Oberon-2 are related to each other in the following way:

SHORTINT is a subset of **INTEGER**

INTEGER is a subset of **LONGINT**

LONGINT is a subset of **REAL**

REAL is a subset of **LONGREAL**

This is sometimes called "type inclusion", and affects the way in which we can use variables of different numeric types together in an expression.

Since the "smaller" types are included within the "larger" types, a data item belonging to a "smaller" type may be assigned to a variable of a "larger" type at any time. For example, assume that we have the following variables:

```
VAR
   s : SHORTINT;
   i : INTEGER;
   l : LONGINT;
```

then the following assignments are acceptable:

```
   i := s;
   l := i;
   l := s;
```

However, if you need to assign a data item of a "larger" type to a variable belonging to a "smaller" type, you must convert the "larger" type's data item into the "smaller" type using a special, built-in type conversion function called **SHORT**.

Without using **SHORT** the following assignments will all give "incompatible assignment" compilation errors:

```
   i := l;
   s := i;
   s := l;
```

We use the **SHORT** function to allow such assignments in the following way:

```
i := SHORT(l);
```

turns **LONGINT** data item **l** into an **INTEGER**, and assigns it to **i**, which is itself an **INTEGER**.

```
s := SHORT(i);
```

turns **INTEGER** data item **i** into a **SHORTINT**, and assigns it to **s**, which is itself a **SHORTINT**.

To convert a **LONGINT** to a **SHORTINT** we must use the **SHORT** function twice, for example,

```
s := SHORT(SHORT(l));
```

SHORT(l) returns the value of **l** as an **INTEGER**. Then we apply **SHORT** to this **INTEGER** value, and obtain a **SHORTINT** value which can be assigned to **s**.

In general, **SHORT(x)** returns the value of **x** as the next smaller type, which is why it must be used twice to move from a **LONGINT** to a **SHORTINT**.

x : INTEGER	**SHORT(x)**	gives a **SHORTINT**
x : LONGINT	**SHORT(x)**	gives an **INTEGER**
x : LONGREAL	**SHORT(x)**	gives a **REAL**

You must be very careful when using **SHORT**. If you use it to try and assign a value to a variable which is outside the minimum and maximum range of values that the variable you are assigning to can hold, then your program will crash with a runtime error. For example, given the following, where **l** is a **LONGINT** variable, and **i** is an **INTEGER** variable,

```
l := 32768;
i := SHORT(l);
```

The assignment of 32768 to **l** is perfectly acceptable, since this value is well within the range of values that a **LONGINT** variable can hold. However, the second assignment will cause a runtime error because it is impossible for the system to store the value 32768 in an **INTEGER** variable (the maximum allowed **INTEGER** is 32767). So only use **SHORT** when you are sure that the value you are trying to convert will fit into a variable of the type you are converting it to.

If you need to convert from a **LONGREAL** value to a **REAL** value you can use the **SHORT** function to do so. However, if the **LONGREAL** has more significant digits than can be held in a **REAL** variable, then the **LONGREAL** value will be truncated to fit.

If you need to mix integer and real numbers in your programs then you must also follow certain rules. **SHORTINT**, **INTEGER** and **LONGINT** variables can all be directly assigned to a **REAL** or **LONGREAL** variable. Thus, given the following variable declarations:

```
VAR
   s : SHORTINT;
   i : INTEGER;
   l : LONGINT;
   r : REAL;
```

the following assignments are all allowed:

```
r := s;
r := i;
r := l;
```

However, we cannot directly assign a **REAL** (or **LONGREAL**) number to any type of integer variable. This is quite obvious when you consider that integers are always whole numbers, but reals are not. If you need to assign

a **REAL** number to an integer variable you must use another built-in function called **ENTIER**. The **ENTIER** function, when given a real value, will truncate it, chopping off everything to the right of the decimal point, and returning the remaining integer value. In fact **ENTIER** returns the truncated real number as a **LONGINT**.

So, to assign the contents of the **REAL** variable **r** to the **LONGINT** variable **l**, we must do the following:

```
l := ENTIER(r);
```

If, for example, **r** contained the value 2.031, then **ENTIER(r)** would give the value 2. If r contained the value 2.95, then **ENTIER(r)** would also give the value 2 — remember that **ENTIER** does not round the real number it is given, it simply truncates it.

Mixed type arithmetic always requires great care. Do not be afraid to print out intermediate values produced by your program, so that you can check they are of the correct size.

Exercises

1. Given the following types declarations and assignments:

```
VAR
   x, y, z : INTEGER;
.....
x := 15;
y := 3;
z := 2;
```

what are the values of the following expressions?

```
x DIV yx MOD z   y DIV z    y MOD x
x MOD yx DIV z   z DIV y    Z MOD x
```

2. Write a program which asks the user for two integer numbers and calculates first number **MOD** second number and first number **DIV** second number, and prints out the results. Try to make your program as user-friendly as possible, so that the user is told exactly what to enter, and can understand what the outputs mean.

3. Write a program, using real numbers, to,

   (i) Calculate 1.0 / 3.0

   (ii) Multiply the calculation in (i) by 3.0, and subtract 1.0 from the result.

   (iii) Display the final result using **Out.Real** with a field width of 15.

   Is the result equal to 0? If not, why not? try the same calculation on a hand-held calculator. Do the results differ? What does this suggest?

4. Write a program, **ToLower**, which asks the user to enter an uppercase letter, and prints out its lowercase equivalent.

A.5 Sequence control

Our Oberon-2 programs have so far only used direct sequencing, where statements are executed in the order in which they appear. In the earlier section on algorithm design we discussed some other methods of sequence control:

Conditional branching	if x then do y otherwise do z
Bounded iteration	do y exactly n times
Unbounded iteration	1. while x do y 2. do y until x.

Oberon-2 provides constructs which match these forms of sequence control. We will look at how these constructs are implemented in Oberon-2 in the following sections.

A.6 Conditional branching

This is done with the **IF** statement, which is of the form:

```
IF condition 1 THEN
   statement sequence 1
ELSE
   statement sequence 2
END;
```

Where statement sequence 1 is evaluated only if its associated Boolean expression, condition 1, is **TRUE**, otherwise statement sequence 2 following **ELSE** is executed.

You can have as many different branches as you need in an **IF** statement, so you can have multiple-way choices in your programs. The more general form of the **IF** statement is therefore:

```
IF condition 1 THEN
   statement sequence 1
ELSIF condition 2 THEN
   statement sequence 2
ELSIF condition 3 THEN
   statement sequence 3
- - - -
ELSIF condition n - 1 THEN
   statement sequence n - 1
ELSE
   statement sequence n
```

```
END;
```

A statement sequence is only executed if its associated Boolean condition is **TRUE**. Note that at most one statement sequence will be executed.

You do not have to have an **ELSE** at the end of an **IF** statement if you have no need of it. In most cases the **ELSE** acts as a catch-all, performing some set of actions if none of the other branches of the **IF** statement have been executed. However, in some cases this is not required. It is therefore perfectly acceptable to have an **IF** statement of the form:

```
IF temperature >= 85 THEN
   Out.String("Unusually hot.");
   Out.Ln;
END;
```

You will find several more examples of IF statements in the programs in the remainder of this Appendix.

A.6.1 Boolean expressions and type BOOLEAN

The conditions we use in **IF** statements are called *Boolean expressions* – the value of a Boolean expression is either true or false, there are no other possibilities. The Boolean value of an expression is its *truth value* – whether it is true or whether it is false. For example,

3 > 2	has Boolean value	true
1 = 1	has Boolean value	true
10 > 12	has Boolean value	false
7 <= 3	has Boolean value	false.

The operators used here are called relational operators, some of which we met briefly in the section on type **CHAR**. Here is a full list:

Symbol	Meaning
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equals
#	not equal to.

The result of a relational operator is always a Boolean value. Oberon-2 allows the use of Boolean values by providing type **BOOLEAN** which has

two possible values, **TRUE** and **FALSE**. A variable of type **BOOLEAN** can therefore only be **TRUE** or **FALSE**. There are also three operators for **BOOLEAN**: **&** (Boolean AND), **OR**, and **~** (Boolean negation or NOT), which can be used on **BOOLEAN** variables and expressions. We can define these operations with a truth table:

x	y	~ x	x & y	x OR y
TRUE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE

Here are some examples of **BOOLEAN** expressions and operators in use:

```
In.Int(x);
In.Int(y);

IF x >= y THEN
  max := x;
ELSE
  max := y;
END;

noBugs := TRUE;
programWorks := FALSE;
IF programWorks & noBugs THEN
  programmerHappy := TRUE;
ELSE
  programmerHappy := FALSE;
END;

summer := (temperature >= 75) & sunny;
IF summer THEN
  Out.String("Time for a holiday!");
END;
```

In the above code fragments **noBugs**, **programWorks**, **programmerHappy** and **summer** are all **BOOLEAN** variables.

A common mistake made when using **BOOLEAN** variables is to assume that a statement such as:

```
Out.String(programmerHappy);
```

will print out either **TRUE** or **FALSE**, depending on the Boolean value of the given variable, in this case **programmerHappy**. This does **not** work – **TRUE**

and **FALSE** are simply names for some internal representation of type **BOOLEAN**. **Out.String** expects to be given a string of characters between quotes, such as "hello !", which it will write out on the terminal, or a variable of type **ARRAY OF CHAR**, which we will meet later. The variable **programmerHappy** is neither a string of characters between quote marks nor of type **ARRAY OF CHAR**, so **Out.String** cannot print it. If you want to print out the Boolean value of an expression or variable you must write the code to do so yourself, for example:

```
IF programmerHappy THEN
   Out.String("TRUE");
ELSE
   Out.String("FALSE");
END;
```

You must remember that the string of characters represented by "TRUE" is not the same as the Boolean value **TRUE**. Try experimenting to see what error messages your compiler gives when you give **Out.String** something unexpected to display.

A.6.2 An alternative to **IF** – the **CASE** statement

The **CASE** statement is another form of conditional branch, which is generally thought to be more elegant than the **IF** statement in some situations. Here is an example of a **CASE** statement:

```
CASE programmerHappy OF
     0 : Out.String("FALSE"); |
     1 : Out.String("TRUE")
END;
```

This has exactly the same effect as the previous **IF** statement. A **CASE** statement can also have a trailing **ELSE**, which acts exactly like the **ELSE** of an **IF** statement, being a catch-all for cases which don't meet any of the given conditions:

```
CASE day OF
      1 : Out.String("Monday")|
      2 :    Out.String("Tuesday")|
      3 :    Out.String("Wednesday")|
      4 :    Out.String("Thursday")|
      5 :    Out.String("Friday")|
      6 :    Out.String("Saturday");
             Out.Ln;
             Out.String("Don't get up today!")|
      7 :    Out.String("Sunday");
             Out.Ln;
             Out.String("Mow the lawn.");

ELSE
   Out.String("Invalid day number.");
```

```
   Out.Ln;
END;
```

The **ELSE** statement will be entered if day is not 1, 2, 3, 4, 5, 6, or 7. The general form of a **CASE** statement is given below:

```
CASE expression OF
       case-label-1 : statement-sequence-1; |
       case-label-2 : statement-sequence-2; |
       case-label-3 : statement-sequence-3;
ELSE
   statement-sequence;
END;
```

The vertical bar separates each choice in the **CASE**, and is needed because the statement sequence associated with each choice can be one statement or many statements.

The case labels must be of type **INTEGER** or **CHAR**, or expressions which evaluate to either of these two types, so the following **CASE** statement is illegal:

```
CASE price OF
    <= 20 : canBuy := TRUE; |
    > 20 : canBuy := FALSE;
END;
```

If more than one label applies to a particular case, then the labels can be separated by commas, that is

```
CASE character OF
       "Y", "y" : Out.String("Yes") |
       "N", "n" : Out.String("No");

ELSE
   Out.String("Invalid character");
END;
```

The main difference between the **CASE** and the **IF** concerns the **ELSE** clause. In an **IF** statement, if the **ELSE** clause is missing and all the choices in the **IF** fail, control simply falls through to the statement which comes after the end of the **IF** statement. However, in a **CASE** statement without an **ELSE** clause, if all the choices in the **CASE** fail when the code is run, the program itself will fail immediately.

Exercises

1. Given the following Boolean variables:

```
   IsRaining := TRUE;
   Cold := FALSE;
```

what is the value of each of the following Boolean expressions?

(i) `IsRaining & Cold`

(ii) `(~IsRaining) OR Cold`

(iii) `Cold & (6 < 4)`

2. Write the following **IF** statements:

   (a) Assign a value of **TRUE** to **positive**, if the value of **n** lies between 1 and **Max** inclusive; otherwise assign a value of **FALSE**.

   (b) Assign a value of **TRUE** to **uppercase** if **ch** is an uppercase letter; otherwise assign a value of **FALSE**.

   (c) Assign a value of **TRUE** to **divisor** if **M** is a divisor of **N** (ie. goes into **N** an exact number of times); otherwise assign a value of **FALSE**.

   (d) If **item** is non-zero, then multiply **product** by **item**, and save the result in **product**; otherwise skip the multiplication. In either case print out the value of **product**.

3. Write a program that will read in a character value and a real number. Depending on what is read, certain information will be printed. If the character is an "S" and, for example, the number is 500.50 then the program will print out:

   ```
   Send money! I need £500.50
   ```

   If the character is a "T" then the program will print out:

   ```
   The temperature last night was 500.50 degrees.
   ```

   If any other character is read in the program will print out:

   ```
   Sorry!
   ```

4. Write a program which reads an exam mark typed in by the user, in the range 0 to 100 (inclusive) and prints out the equivalent grade as an uppercase letter. Grades are assigned as follows:

Exam score	Grade
Below 45	F
45 - 49	D
50 - 59	C
60 - 69	B
70 - 100	A

5. Write a program which uses a **CASE** statement to print out a message indicating whether **nextCh** (of type **CHAR**) is an operator symbol (**+ - * / # < > & ~**), an Oberon-2 punctuation symbol (**; . ( ) |**), a digit, a letter or something else. The character **nextCh**, which is to be classified, should be entered by the user in response to an appropriate message from the program.

6. Rewrite your program from 4. using an **IF** statement.

A.7 Bounded iteration

This type of iteration takes the form *do x exactly y times*. It is particularly useful when an algorithm needs to do a task or tasks a fixed number of times. You will also find bounded iteration is useful when using arrays (another means of storing data, which will be discussed later).

A.7.1 The **FOR** loop

We will use bounded iteration to display a message on the terminal a fixed number of times. The Oberon-2 form of bounded iteration is called the **FOR** loop. **FOR** loops often appear at first glance to be rather confusing constructs. Here is a **FOR** loop which will display the message "hello" five times:

```
FOR count := 1 TO 5 DO
  Out.String("hello");
  Out.Ln;
END;
```

The confusing part is at the top of the loop, where we have

```
FOR count := 1 TO 5 DO
```

You may well be wondering what **count** is for. **Count** will have been declared as a variable of type **INTEGER** , and it is called the *control variable* of the **FOR** loop. Each time round the loop **count** will be automatically incremented by 1.

The body of the loop is executed for each value of **count**, from 1 up to and including 5. Note that the control variable doesn't have to be called **count**, it can be called anything you like, so long as it is declared as a variable of type **INTEGER**.

The general form of a **FOR** loop is slightly more complicated:

```
FOR controlVariable := expression1 TO expression2
            BY constantExpression DO
     statement sequence
END;
```

Expression1 is the starting value for **controlVariable**, and **expression2** is the limit. **ConstantExpression** is the increment or decrement, to be applied to **controlVariable** each time round the loop; if **constantExpression** is positive it is an increment, if negative it is a decrement. If **BY constantExpression** is missing then an increment of 1 is assumed. Note that the control variable must be **INTEGER**, **not REAL**. The control variable is incremented (or decremented) automatically each time round the loop until its value passes that of the limit (expression-2).

The control variable can be referred to inside the **FOR** loop, for example:

```
 FOR c := 2 TO 10 BY 2 DO
  Out.Int(c,2);
  Out.Ln
END;
```

will print out:

```
     2     4     6     8     10
```

But you should **never** assign a value to the control variable inside the loop. In particular remember that you must **not** increment or decrement the control variable by an explicit statement within the loop. The incrementing or decrementing is done automatically when the **FOR** loop is run.

Exercises

1. Write a program to find the largest, smallest and average value in a collection of n integer numbers, where the value of n will be the first item read in.

A.8 Conditional iteration

Oberon-2 provides constructs which mirror both *while x do y* and *do y until x*. These are called **WHILE** loops and **REPEAT** loops respectively.

A.8.1 The **WHILE** loop

Consider the problem of reading in a string of characters one at a time, until some *end of text* character is read. We have already seen an algorithm for this problem:

```
Read a character
While character isn't "." do
   Process character
   Read next character
Endwhile
```

We can now write this in Oberon-2, using a **WHILE** loop:

```
In.Char(ch);
WHILE ch # "." DO
   (* Here we would process ch in some way. *)
   In.Char(ch);
END;
```

The **END** statement corresponds to the Endwhile statement in our algorithm. The Oberon-2 **WHILE** tests the condition at the top of the loop, before the statements in the body of the loop are executed. In our example, if the first character typed is "." the test **ch # "."** will fail on the first attempt and the body of the loop will never be executed, causing control to move to the statement following the end of the loop. Here is a complete program which counts the occurrences of a given letter in a string of characters typed in at the terminal. The user must enter both the letter to be counted and the text. The algorithm will be as follows:

```
Read letter to count
Set count to zero
```

```
Read text, adding one to count each time letter occurs
Display count
```

This can be refined further to produce a more detailed algorithm:

```
Ask user what letter to count
Ask user to enter text, terminated by a full stop
Set letterCount to zero
Read a character
While character isn't "." do
   If character = letter then
      Add one to letterCount
   Endif
   Read next character
Endwhile
Display letterCount
```

Writing this in Oberon-2 we obtain the following program:

```
MODULE Letters;
(* Count the occurrences of a given letter (entered by
the user) in a string of characters, which is typed in
at the terminal by the user.  The character string must
be terminated by a full stop.  A letter must be an
alphabetic character "a".."z" or "A".."Z". *)

IMPORT In, Out;
PROCEDURE ProgMain*;
CONST
   FullStop = "."; (* Terminates input string. *)
   FieldWidth = 3; (* Number display width. *)
VAR
   letter : CHAR; (* Letter to be counted. *)
   ch : CHAR; (* Character read in. *)
   letterCount : INTEGER; (* Occurrences of letter. *)
BEGIN
   (* Ask user to enter letter to be counted. *)
   Out.Ln;
   Out.String("Enter letter to be counted ('a'..'z' or
'A'..'Z'):  ");
   In.Char(letter);
   Out.Char(letter);
   Out.Ln;
   IF ((letter >= "a") & (letter <= "z")) OR
      ((letter >= "A") & (letter <= "Z")) THEN
      (* We have a valid letter. *)
      letterCount := 0;
      (* Ask user to enter text. *)
```

```
      Out.String("Enter text, ending with a full stop.");
      Out.Ln;

      (* Read in characters and count occurrences. *)
      In.Char(ch);
      WHILE ch # FullStop DO
        IF ch = letter THEN
          (* Add one to letterCount. *)
          letterCount := letterCount + 1;
        END;
        (* Read next character. *)
        In.Char(ch);
      END;
      (* Display total occurrences of letter. *)
      Out.Ln;
      Out.String("The number of occurrences is: ");
      Out.Int(letterCount, FieldWidth);
      Out.Ln;

    ELSE (* Not a valid character. *)
      (* Display an error message and finish. *)
      Out.Ln;
      Out.String("Not a valid character.");
      Out.Ln;
    END; (* IF *)
END ProgMain;
END Letters.
```

Note that we have added some error checking to make sure that the user only enters a valid character to be counted. This is done by an **IF** statement which controls entry to the main body of the program, so that the text is only requested and read if the user has given a valid character to count, that is, a letter. If the character given is invalid the **ELSE** clause will be entered, and an error message is printed out.

A.8.2 The **REPEAT** loop

This type of loop is most useful when some task has to be performed at least once. If we return to one of our earlier programs, **SqrCube**, which read in an integer and printed out its square and its cube, we can modify this program so that it can be used for more than one number in each execution. To do this we will alter the algorithm first. The initial algorithm for the program was:

```
Ask user to enter a number
Read in the number
Square the number
Print out the square
Cube the number
```

```
Print out the cube
```

Now we want an iterative construct which can go around this whole algorithm, so that the user is asked repeatedly to enter a number. This also suggests that we need to consider how the user is to stop the program, so that it doesn't keep asking for numbers until the next power cut. One way would be to ask the user each time round the iteration if another number is to be entered, and stop the iteration if the response is no. If we use a do-loop for the iteration we need a condition to test at the end of the loop.

```
Do the following
   Ask user to enter a number
   Read in the number
   Square the number
   Print out the square
   Cube the number
   Print out the cube
   Ask if user wants to enter another number
   Read in the response typed
Until the response is negative
```

Remember, the body of a do-loop is executed at least once, because the test for stopping the iteration is at the bottom of the loop. This is the opposite situation to a while loop, where the test is at the top, so that the body of the loop need not be executed at all.

This algorithm can now be written in Oberon-2, using the equivalent of do y until x – the REPEAT loop, which is of the form:

```
REPEAT
   (* Statements go here. *)
UNTIL condition;
```

where condition is a Boolean variable or expression.

Here is the program, incorporating such a loop:

```
MODULE SqrCube2;
(* Reads positive integers from the terminal and
displays their squares
and cubes. *)
IMPORT In, Out;
PROCEDURE ProgMain*;
 VAR
   number, square, cube : INTEGER;
   response : CHAR;
BEGIN
   number := 0;

   REPEAT
      (* Get the number from the user. *)
```

```
      Out.String("Enter a number: ");
      In.Int(number);

      (* Calculate the square and cube. *)
      square := number * number;
      cube := square * number;
      (* Display the results. *)
      Out.String("The square is:  ");
      Out.Int(square, FieldWidth);
      Out.Ln;
      Out.String("The cube is:  ");
      Out.Int(cube, FieldWidth);
      Out.Ln;

      (* Find out if user wants to stop. *)
      Out.String("Enter another number? (y/n): ");
      In.Char(response);
   UNTIL (response = "n") OR (response = "N");

END ProgMain;
END SqrCube2.
```

If you examine the code closely you will see that the iteration (and the program) will finish when the user types "n", and continue if any other character (not just a "y") is typed. The answer "n" causes Boolean variable **stop** to be set to **TRUE**, so the condition on the **UNTIL** is now true, stopping the iteration and exiting the **REPEAT** loop.

A.8.3 The LOOP

There is yet another loop in Oberon-2 which can provide unbounded iteration – this is called the **LOOP** statement. A **LOOP** statement looks like this:

```
LOOP
   statement-sequence
   IF condition THEN
      EXIT;
   END;
   statement-sequence
END;
```

When the condition to the **IF** statement is true, the **EXIT** statement will be executed, which causes control to leave the **LOOP** and move to the next statement after the end of the **LOOP**. The **LOOP** construct is useful in situations where you want to exit from the middle of a loop rather than from the top or the bottom. Alternatively, it can be used to produce an infinite loop by leaving out the **EXIT** statement. Of course, one usually wants to avoid programs with infinite loops which never terminate, but some programming situations, such as multi-programming, do need non-

exiting loops. However, in most cases loops are better written using **WHILE** or **REPEAT** loops. Here is an example where use of **LOOP** is relatively valid:

```
LOOP
   Out.String("Enter number to square: ");
   In.Int(number);
   Out.Ln;

   IF number = 0 THEN
      EXIT;
   END;

   (* Process number. *)
   number := number * number;
   Out.String("The square is:  ");
   Out.Int(number, 6);
   Out.Ln;
END;
```

However, in general you should avoid using the LOOP construct.

Exercises

1. Write a program, using a **REPEAT** loop, that will find the product (ie. a * b * c * d * ....) of a collection of integers typed in by the user. The user should be prompted to enter each integer on a separate line, terminating the data entry by typing the value 0. Echo the user's inputs on the screen.
2. Rewrite the above program using a **WHILE** loop instead of a **REPEAT** loop.
3. Write a program that converts Celsius temperatures to Fahrenheit, and Fahrenheit temperatures to Celsius. The conversion from Celsius to Fahrenheit is achieved by the following equation:

   F = 1.8C + 32

   where C is the given temperature in Celsius, and F is the temperature in Fahrenheit. Of course, the equation can easily be rearranged to convert from Fahrenheit to Celsius.

   The program should request an integer temperature from the user, and then ask:

   (C) Convert Celsius to Fahrenheit

   (F) Convert Fahrenheit to Celsius

   (Q) Quit?

   The program reads the user's response and converts appropriately, giving the result with a suitable message. The program should continue accepting temperatures and converting them until the user responds with a "Q". The program should accept uppercase or lowercase letters, and should use a **WHILE** loop or **REPEAT** loop as you prefer.

A.9 Structured data types

We have so far shown how to declare single data items of a particular type, but many situations need to use groups of data items. For example, if we want to read in a word typed at the terminal, we would really like to be able to store the characters as one data item, rather than putting each character into a separate variable. If we have to store the characters of a word in separate variables the problems are two-fold. Firstly we need as many variables as there are characters in the word, so this might mean anything from one to twenty variables (or more), and secondly we must declare all twenty variables, even though most of the time we will only be using five or ten, because we don't know beforehand how large each word will be.

Similarly, if we need to read in a list of daytime temperatures and perform several operations on this list, we would like to have a way of storing these temperatures all together, so that we can refer to them collectively by one name, and still access each one individually when needed.

A.9.1 The **ARRAY** type

An array is a list of data items:

7.0, 2.5, 12.31, 6.3, 0.05

All the data items in an array must have the same type, so they must be all **REAL** or all **CHAR** etc. We need to be able to refer to each item in an array individually – this is done by *indexing* the array with a suitable range of numbers:

0	1	2	3	4	Index
7.0	2.5	1.31	6.3	0.05	Array

The data item at index 0 is 7.0, the data item at index 3 is 6.3, and so on.

Here is a declaration of an array variable in Oberon-2, which could be used for storing a list of temperatures:

```
VAR
   temperatures : ARRAY 20 OF REAL;
```

This array holds 20 real numbers, which will be indexed 0, 1, 2,.. 19. So the first number is at index 0, the second is at index 1 and so on. This is an example of a one-dimensional array.

The general form of a one-dimensional array variable declaration is:

```
name : ARRAY size OF DataType;
```

DataType is the type of items which can be stored in the array. An array can only store items of one type, the type defined in its declaration.

Two-dimensional and multi-dimensional arrays are declared in a similar way:

```
name : ARRAY size1, size2, .... size3 OF Datatype;
```

where **size1**, **size2**, etc are the sizes of the successive dimensions of a N dimensional array (note, the maximum value of N may be dependent on the particular Oberon-2 system you are using). For example:

```
board : ARRAY 8, 8 OF BlackOrWhite;
```

gives an 8 x 8 two dimensional array of black or white values which could be used to represent a chess board.

We will use the declaration of **temperatures** for the following examples. How can we assign a value to an array location? This can be done with an assignment statement:

```
temperatures[0] := 65.3;
temperatures[1] := 62.7;
```

and so on. The square brackets indicate the index (that is, the location) at which the value should be stored. We can also use a variable (of the correct type) to refer to a location in the array:

```
VAR
   position : INTEGER;
   ....
BEGIN
   position := 5;
   temperatures[position] := 75.0;
   ....
END;
```

This assigns the value 75.0 to location 5 in the array (so it will be the sixth temperature in the array). Note that if an index is given that is not in the defined range (0 through to 19 inclusive, in this case), the program will fail when it is run, as soon as an attempt is made to refer to an invalid array location.

We can now use an array in a program which reads in up to twenty temperatures and prints out their average. We need to consider how the user can indicate that he has entered all the temperatures he wants averaged – we could ask the user to enter the number of temperatures he is going to give, before he starts typing them in, which would allow us to use a **FOR** loop.

```
Ask user how many temperatures he wants to enter
Store in number
Set index to 1
(* Read in the temperatures. *)
Do the following number times
   Read in a temperature
```

```
  Store in array at position index
  Add 1 to index
Enddo

Set total to 0
Set index to 1

(* Total the temperatures. *)
Do the following number times
  Get value stored in array at position index and add
  to total
  Add 1 to index
Enddo
Calculate average
Print out average
```

In Oberon-2 we get the following program:

```
MODULE Average;
(* Find the average of up to twenty temperatures. *)
IMPORT In, Out;
PROCEDURE ProgMain*;
CONST
  FieldWidth = 4; (* Number display width. *)
VAR
  c : Integer; (* loop control variable. *)
  number : INTEGER; (* Number of temperatures. *)
  temp : REAL; (* Temperature read in. *)
  total : REAL; (* Total of all temperatures. *)
  average : REAL; (* Average temperature. *)
  temperatures : ARRAY 20 OF REAL; (* Temp. list *)

BEGIN
  (* Ask user how many temperatures he will enter. *)
  Out.String("How many temps to average (up to 20)? ");
  In.Int(number);
  (* Check that it's a valid number of temperatures. *)
  IF (number <= 20) & (number > 0)THEN
    (* Read in the temperatures. *)
    Out.String("Enter temperatures, one per line:");
    Out.Ln;
    FOR c := 0 TO number-1 DO
      In.Real(temp);
      Out.Ln; (* Space out input for user. *)
      temperatures[c] := temp;
    END;
    (* Average temperatures. *)
```

```
    Out.String("Calculating average....");
    Out.Ln;
    total := 0.0;
    FOR c := 0 TO number-1 DO
      total := total + temperatures[c];
    END;
    average := total / number;
    (* Print out the result. *)
    Out.String("The average temperature is: ");
    Out.Real(average, FieldWidth);
  ELSE
    (* User entered invalid number of temperatures. *)
    Out.String("Invalid number of temperatures.");
    Out.Ln;
  END;
  END ProgMain;
  END Average.
```

There are several points to note in this program. Firstly, in transferring from the algorithm to Oberon-2 we find that a separate variable for the index is not needed – we can use the loop control variable **c** to index the array of temperatures.

Secondly, note how we have used a **FOR** loop to step through the array, one location at a time. Working through arrays is a common use of **FOR** loops, and is one of the reasons why **FOR** loops have explicit control variables (so that the control variable can be used for the array index).

A.9.2 Character arrays

Arrays of characters are very widely used, so we will take a more detailed look at them. We will consider the problem of reading in a word and storing it as a single data item – this is an obvious case for using an array of characters. A possible definition for a word is:

```
VAR
  word : ARRAY 25 OF CHAR;
```

which allows us words of up to 25 characters.

Before we go any further, it is worth mentioning that Oberon-2 regards strings as arrays of characters, where the lower bound of the array index is zero, and the upper bound is n-1 for a string of n characters. The following code fragments will produce exactly the same output:

```
Out.String("Hello");
```

```
CONST
  Greeting = "Hello";
```

```
BEGIN
  Out.String(Greeting);
```

```
VAR
  greeting : ARRAY 5 OF CHAR;
BEGIN
  greeting := "Hello";
  Out.String(greeting);
```

The length of a string of characters is the same as or less than the size of the array. Since a string may be shorter than the array in which it is stored we need some means of indicating where in the array the string ends. To do this we store a special character in the unused part of the array. This character is called *null*, and its special attribute is that it is not printable or displayable in any way whatsoever. Since we can't display the null character we can only refer to it by its ordinal value in the ASCII table, 0. Thus **`CHR(0)`** is the null character.

Let's look at what happens to array variable greeting when different length strings are assigned to it.

```
greeting := "Hello";
```

H	e	l	l	o

```
greeting := "Hi"
```

H	i	CHR(0)	CHR(0)	CHR(0)

```
greeting := "Salutations";
```

S	a	l	u	t

The Pow! compiler will complain if you try to assign a string constant to a character array which is too small to hold it.

```
greeting := "";
```

CHR(0)	CHR(0)	CHR(0)	CHR(0)	CHR(0)

This last example is called a null string, which is indicated by the two quote marks without any gap between them. It is a completely empty string, and will have no effect on the output when displayed using **`Out.String`**.

A.9.3 Array type declarations

Instead of just using arrays directly in variable declarations (as shown in the previous examples), we can declare new array types, in the **TYPE** declaration section of our programs. The following type declaration, declares a new, user-defined type, **TemperatureList**, which is an array type:

```
TYPE
   TemperatureList = ARRAY 31 OF REAL;
```

Now we can declare variables of this new type,

```
VAR
   dayTemps, nightTemps : TemperatureList;
```

This type declaration can help to clarify a program if a meaningful name is chosen for the type. It is also required if you are going to need multiple variables of your array type, as shown above.

Let's try and design a program which draws a simple bar chart of some given input data. The user must first state how many bars there are to be in the chart, and then enter the size of each bar. The program should then display the bar chart sideways (because this makes it much easier than displaying a vertical bar chart) with the size printed at the end of each bar. So, a typical input might look like this:

```
5  12  23  9  3  10
```

and the corresponding output should look like this:

```
*****5
************12
***********************23
*********9
***3
**********10
```

We need to use several of the concepts we have met in this chapter, such as arrays, **FOR** loops and **IF** statements. First we need to design the algorithm:

```
Get number of bars from user - store in barCount
Do the following barCount times
   Read a bar size.
   Store in bar chart array
Enddo

(* Print out bar chart. *)
Do the following barCount times
   Get a bar size from bar chart array
   Do the following bar size times
        Print a "*"
   Enddo
   Print bar size
```

```
  Move to next line
Enddo
```

Now here it is in Oberon-2:

```
MODULE Chart;
(* Displays a horizontal bar chart, given the number of
bars to display and the size of each bar (as positive
integers). *)
IMPORT In, Out;
PROCEDURE ProgMain*;
CONST
  FieldWidth = 3; (* Number of spaces to print out a
number. *)
  MaxChart = 24; (* Max number of bars allowed in
chart. *)
TYPE
  BarChart = ARRAY 20 OF INTEGER;
VAR
  barCount : INTEGER; (* Number of bars. *)
  barSize : INTEGER; (* Size of current bar. *)
  chart : BarChart; (* The bar chart array. *)
  c, d : INTEGER; (* Loop indexes. *)

BEGIN
  (* Get number of bars from user. *)
  Out.String("Enter number of bars (up to 20): ");
  In.Int(barCount);
  Out.Ln;

  (* In. in bar sizes. *)
  FOR c := 0 TO barCount-1 DO
    Out.String("Enter bar size: ");
    In.Int(barSize);
    Out.Ln;
    chart[c] := barSize;
  END;

  (* Print out bar chart. *)
  FOR c := 0 TO barCount-1 DO
    barSize := chart[c];
    FOR d := 1 TO barSize DO
      Out.Char("*")
    END;
    Out.Int(barSize, FieldWidth);
    Out.Ln;
  END;
```

```
END ProgMain;
END Chart.
```

To print out the bar chart we have used one **FOR** loop to print each bar in turn that is

```
FOR c := 0 TO barCount-1 DO
   ....
END;
```

We have used another **FOR** loop to print the asterisks for each individual bar that is

```
FOR d := 0 TO barSize-1 DO
   Out.Char("*");
END;
```

Using one loop or one sequence control structure within another is often called *nesting*. Since all control structures just contain statement sequences we can have a **WHILE** loop inside an **IF** statement or a **REPEAT** loop inside a **CASE** statement. If you look back at some of the programs demonstrating sequence control you will see some more examples of nesting.

A.9.4 Records

Arrays provide us with a means of grouping data items of the same type. However, consider the situation where we want to store all the details about each employee in a small computer company. The information to be stored for each employee consists of:

Name – String

Position – String

Salary – Real number

Payroll reference – Positive integer.

We can declare these data items as separate variables:

```
VAR
   name : ARRAY 20 OF CHAR;
   position : ARRAY 20 OF CHAR;
   salary : REAL;
   payrollRef : INTEGER;
```

but it would be preferable to place them all together under one name, as we did with the array of temperatures. Unfortunately, we cannot use an array because of the restriction that all data items in an array must be of the same type. However, Oberon-2 provides us with another do-it-yourself data type called the **RECORD**.

In Chapter 2, we used records in the car pool reservation system, where the details of each car in the car pool were stored as records containing several

pieces of information about the car. Similarly, a Oberon-2 **RECORD** consists of a group of named slots, each of which can hold a data item. When a **RECORD** is declared we also declare the name and type of each slot.

```
TYPE
  EmployeeRecord =
    RECORD
      name : ARRAY 20 OF CHAR;
      position : ARRAY 20 OF CHAR;
      salary : REAL;
      payrollRef : INTEGER;
    END;
VAR
  employee1, employee2 : EmployeeRecord;
```

Now we have declared a **RECORD**, we need a means of accessing each slot. In the car pool example of Chapter 2, we defined our own operations for accessing each slot in a record, but, luckily Oberon-2 provides the operations for us, although the syntax looks slightly strange at first:

```
variableName.slotName
```

For example,

```
employee1.salary
```

refers to slot **salary** in record variable **employee1**, and

```
employee2.salary
```

refers to slot **salary** in record variable **employee2**.

We can use the assignment statement to put a value into a record field, or use a field in an expression (provided it is of the correct type). For example:

```
bonus := 500.0;
employee1.salary := employee1.salary + bonus;
Out.String(employee1.name);
Out.Ln;
Out.Real(employee1.salary, 8);
```

where the record type and its variables are as defined above, and bonus is of type **REAL**.