Introduction to Systematic Programming
Unit 7 - Writing Larger Programs

7.1 Introduction

The first few units of this course have advocated a four stage approach to program writing:

i) General think through whole problem
ii) Identify and write down the most important features of a solution
iii) Get an algorithmic scheme
iv) Tidy up to give a complete Ada program.

This scheme is fine for an introduction to writing short, simple programs; but it soon becomes clear that for larger programs a more powerful method is needed. For example think about the factor calculation problem in Unit [4.6]. We looked at a stage (ii) analysis, an algorithmic scheme and at the final program, but did not consider in detail the question of how to get from (ii) to (iii) to (iv). Yet for you to learn to write larger programs, this question must be effectively answered. If we try to apply our current scheme, we will soon find that to make the direct leap from (ii) (features) to (iii) (algorithm) is too difficult (certainly if we want any assurance of correctness). It is also true that once you have some experience of Ada, the clear separation between (iii) and (iv) which is suggested above is unnecessarily restrictive. In fact what we need is a more gradual approach which enables us to make the transitions (ii) - (iii) - (iv) in 'easy stages'. To see how such a method could work, we shall consider the following problem.

7.2 Employees' pay problem - systematic development of a solution

Problem:

Data has been prepared about the employees of a particular company, and contains an employee code number (between 1 and 9999 inclusive), the number of hours and minutes worked during the week, the hourly rate of pay in pounds and pence, and the tax code, for each employee. The data is terminated by a zero employee code number. A program is to be written to determine the following:

- The gross amount (ie. before tax) earned for the week by each employee (in £ and pence)
- The net amount (ie. after tax) earned by each employee for the week (in £ and pence). The tax to be deducted from the gross pay can be calculated as follows:

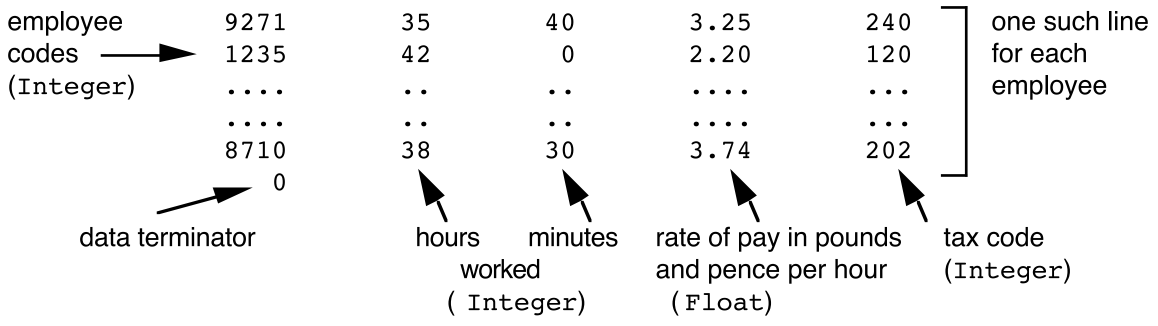
$$\text{annual tax due} = \left[\begin{array}{ll} 0 & \text{if the annual gross pay} \le \text{tax threshold} \\ \\ \text{(annual gross pay - tax threshold) x 0.25} & \text{otherwise} \end{array}\right.$$

where tax threshold = 10 x tax code
annual gross pay = 52 x weekly gross pay
annual tax due = 52 x weekly tax due

- The average actual (ie. net) rate of pay in pounds and pence per hour (to the nearest penny) - averaged over all the employees.

7.2.1 Describe the form of the input data

Start by considering the form that the data should take (making up some typical data values and annotating the values with explanatory comments and data types). For this problem we produce:



```
9271     35     40     3.25     240
1235     42      0     2.20     120
....     ..     ..     ....     ...
....     ..     ..     ....     ...
8710     38     30     3.74     202
   0
```

7.2.2 Describe the form of the output results

Repeat the above for the form of the results (though fewer comments will be necessary since the results will normally include explanatory headings and other text), producing results which correspond to the data values chosen earlier. We thus produce:

```
EmployeeΔΔΔΔGross PayΔΔΔTax DueΔΔΔΔΔNet Pay  ] heading

ΔΔΔ9271ΔΔΔΔΔΔΔ115.91ΔΔΔΔΔ17.44ΔΔΔΔΔΔΔ98.47  ]
   1235        92.40     17.33        75.07  ] one such line
   ....       ......     .....        .....  ] for each
   ....       ......     .....        .....  ] employee
   8710       143.99     26.29       117.70  ]

Average net rate of pay per hour is 2.97     ] summary
```

It is also useful at this stage to indicate the intended layout by the inclusion of spaces and blank lines, though this information will not be used until we write the final program. It is helpful if the values in this 'output diagram' are calculated (by hand), from the values in the corresponding 'input data daigram'.

7.2.3 First thoughts (optional)

We need to work through the data line-by-line, calculating the gross pay, tax due, and net pay for each employee from the time worked, hourly rate and tax code, *and outputting a line of the table for each line of the data* (this implies that we need for a repetition terminated by finding an employee-code whose value is 0). During this repetition, we need to increment a total of the overall net amount earned by all the employees, and to increment a total of the overall time worked by all employees (so that we can later work out the average net hourly rate of pay for all the employees).

When the main repetition is complete (and hence all the rows of the table have been output), we can then calculate and output the average net hourly rate of pay. Of course, the heading for the table will need to be output *before* the main repetition described above commences.

7.2.4 Write a top level design or outline algorithm

The form of the input and output diagrams, the hand calculations needed to produce the values in the output diagram from those chosen in the input diagram and the 'first thoughts' stages should have identified the overall structure of a solution to the problem. The next step is to start to express an algorithmic structure. Firstly, write down a combination of Ada steps and English phrases as appropriate describing (in their correct order) the main features which are important to the overall processing to be performed. Applying this prescription to the example we might obtain something like this:

```
Initialise TotalHrsWorked and TotalNetPay ................ A

Output heading for table ................................. B

Process all the employee records (ie. outputting the
     rows of the table and incrementing the values ......... C
    of TotalHrsWorked and TotalNetPay)

Produce average hourly rate of pay
    (ie. divide TotalNetPay by TotalHrsWorked) ............ D
```

We shall call a form such as the above a **top level design** or **outline algorithm**, ie. a description which captures the main essential structure of the algorithm, with all components in the correct order.

7.2.5 Identify any global variables

List the major variables which are identifiable in the top level design. These will be those variables which have been explicitly referred to (ie. by name) in the top level design, or those variables which are not referred to explicitly, but whose existence can be inferred from the intended steps. In the latter case, it should not include variables which are to be used wholly within a single step of the top level design (such as a variable to hold the calculated average net hourly rate of pay in step **D**), but rather those variables which are needed to hold data values which are required in more than one step, and hence are necessary to communicate information from one step to another.

Variables identified in this way will become the global variables of the (main) program.

In this case the variables which have been used to communicate information between the steps have also been explicitly identified, so we would list the variables:

```
     TotalHrsWorked : Float;  -- Total hours worked for all employees
     TotalNetPay    : Float;  -- Total net pay for all employees
```

Here we note that the variables `TotalHrsWorked` and `TotalNetPay` need to be initialised (to zero) in step **A**, to be updated with the details of individual employee payments in step **C**, and finally to be used to produce the output in step **D** (`TotalNetPay` is divided by `TotalHrsWorked` to calculate the average net rate of pay), and hence are used to communicate values between the steps of the top level design.

7.2.6 Development of top level design steps

The labelled English phrase steps in the top level design must now be expanded, ie. worked out to a greater level of detail. We may proceed by writing each top level design step as a procedure. For each procedure body we follow a similar scheme to that adopted in the formation of the top level design, using a combination of correctly ordered English phrases describing major activities, and Ada steps as appropriate. This typically involves writing what can immediately be determined in Ada (particularly using the formal notation `IF..THEN..ELSE..END IF` for expressing selections, and `WHILE..LOOP..END LOOP` for repetitions) and leaving steps which require greater elaboration as English phrases.

It is usually best to begin with the step that's most crucial to the problem: here that is obviously step **C** (*`Process all the employee records`*). Notice that to simplify organisation each undeveloped step has been given an identifying label. It is clear from what we have said above in [7.2.5] that this procedure is going to have to update the global variables `TotalHrsWorked` and `TotalNetPay`, ie. these variables are going to be `IN OUT` actual parameters of the procedure. Thus a possible development of **C** would be the call:

```
C:   ProcessEmployeeRecords ( TotalHrs => TotalHrsWorked,
                              TotalPay => TotalNetPay )
```
to the procedure:

```
     PROCEDURE ProcessEmployeeRecords ( TotalHrs : IN OUT Float;
                                        TotalPay : IN OUT Float ) IS
     BEGIN
          Get(Item => EmployeeCode)
          WHILE EmployeeCode /= 0 LOOP
               Deal with this employee (ie. calculate
                    the hours worked, gross pay, tax due ....... C1
                    and net pay - and output table row
                    for this employee)
               Update TotalHrs and TotalPay
                    with the details for this employee ......... C2
               Get(Item => EmployeeCode)
          END LOOP
     END ProcessEmployeeRecords
```

As each development proceeds we should list the major variables referenced, using exactly the same approach as was adopted in the top level design. The difference here though is that these variables will become the local variables of the procedure `ProcessEmployeeRecords`.

In this case we note the explicitly identified variable `EmployeeCode`, and the implicitly referred to variables which are worked out in step **C1** and needed in step **C2** to update the values of the variables `TotalHrs` and `TotalPay`. Giving:

```
     EmployeeCode : Integer;
     HrsWorked    : Float;   -- Number of hours worked by this employee
     NetPay       : Float;   -- Net pay of this employee
```

Having commenced the refinement of one major step it is usually better to continue its refinement, thus details of each major English phrase used in the refinement of **C** are then refined. Applying the same process as before we might refine step **C1** as the call:

```
C1: ProcessOneEmployee ( EmployeeNo => EmployeeCode,
                         TimeWorked => HrsWorked, TakeHomePay => NetPay )
```

to the procedure:

```
    PROCEDURE ProcessOneEmployee ( EmployeeNo  : IN Integer;
                                   TimeWorked  : OUT Float;
                                   TakeHomePay : OUT Float ) IS
    BEGIN

         Input Hours, Mins, Rate and TaxCode ........... C1a
         Calculate TimeWorked, GrossPay,
            TaxDue and NetTakeHomePay .................. C1b
         Output EmployeeNo, GrossPay,
            TaxDue and TakeHomePay ..................... C1c

    END ProcessOneEmployee
```

which uses the local variables:

```
    Hours            : Integer;  -- Number of whole hours worked
    Mins             : Integer;  -- Number of minutes worked
    Rate             : Float;    -- Rate of pay in pounds per hour
    TaxCode          : Integer;  -- Tax code
    GrossPay         : Float;    -- Gross pay (before tax)
    TaxDue           : Float;    -- Amount of tax to be paid
```

We don't always refine every step into a procedure, sometimes it is clearer to do so, sometimes it is better for the steps to be expressed directly without using a procedure. There is no hard and fast rule about which approach is best - but general advice is to procedurise if the step being refined is complicated. You might feel that introducing all these intermediate steps is unnecessary, and a tedious waste of time, but experience has shown that sticking to fairly small expansions, about whose correctness you can be reasonably certain, is the best policy in the long run. Having got this far, it is now quite straightforward to expand **C1a, C1b** and **C1c** into Ada:

```
C1a:    Get(Item => Hours)
        Get(Item => Mins)
        Get(Item => Rate)
        Get(Item => TaxCode)

C1b:    TimeWorked := Float(Hours) + Float(Mins) / 60.0
        TaxThreshold := Float(10 * TaxCode)
        AnnualGrossPay := 52.0 * TimeWorked * Rate
        GrossPay := AnnualGrossPay / 52.0
        IF AnnualGrossPay <= TaxThreshold THEN
            AnnualTaxDue := 0.0
        ELSE
            AnnualTaxDue := (AnnualGrossPay - TaxThreshold) * 0.25
        END IF
        TaxDue := AnnualTaxDue / 52.0
        TakeHomePay := GrossPay - TaxDue
```

which uses the additional local variables (local to the procedure `ProcessOneEmployee`):

```
    TaxThreshold   : Float;    -- Threshold for paying tax at all
    AnnualGrossPay : Float;    -- 52 times the weekly gross pay
    AnnualTaxDue   : Float;    -- 52 times the weekly tax to be paid
```

```
C1c:     Put(Item => EmployeeNo)
         Put(Item => GrossPay)
         Put(Item => TaxDue)
         Put(Item => TakeHomePay)
```

where for brevity the output formats have been omitted. Of course they will need to be added later when we write the final program.

Step **C1** is now completely refined. Step **C2** now becomes:

```
C2:      TotalHrs := TotalHrs + HrsWorked
         TotalPay := TotalPay + NetPay
```

Of the remaining undeveloped steps in the top level design (**A**, **B** and **D**), step **B** is almost trivial and hence is not considered further until we write the final program, while the others are relatively straightforward and can be expanded as:

```
A:       TotalHrsWorked := 0.0
         TotalNetPay    := 0.0
```

Having already worked out section **C**, which uses these variables, it is easy to ensure that nothing is overlooked in the initialisation step **A**. This demonstrates one advantage of dealing with the more central parts of a program first, rather than simply trying to work from 'top to bottom'. We can complete the design with the refinement of **D** as the call:

```
D:   OutputAverageRate ( TotalHrs => TotalHrsWorked,
                         TotalPay => TotalNetPay )
```

to the procedure:

```
     PROCEDURE OutputAverageRate ( TotalHrs : IN Float;
                                   TotalPay : IN Float ) IS
     BEGIN
         AverageNetRate := TotalPay / TotalHrs
         Put(Item => "Average net rate of pay per hour is ")
         Put(Item => AverageNetRate)
     END OutputAverageRate
```

which uses the local variable:

```
     AverageNetRate : Float; -- Average net rate of pay per hour
```

7.3 Input/Output using files

With the simple programs discussed so far, we have assumed that input comes from a keyboard and output is sent back to a VDU screen; however, this interactive form is often inconvenient. There may be a large amount of data which we don't want to re-type each time the program is tested, or we may wish to produce printed output, or to avoid input and output getting mixed up on the VDU screen. The solution is to use (magnetic disk) files. If so directed, an Ada program can read its input from a file and/or send its output to a file.

In order to use a file `Employees.dat` (say) for input it must first be opened with the command:

```
     OpenInput(FileName => "Employees.dat");
```

This causes the program to read its subsequent input data (using the `Get` procedure) from the specified data file rather than from the keyboard. Similarly, the command:

```
     OpenOutput(FileName => "Payroll.txt");
```

will cause the program to write its subsequent output results (using the procedures `Put`, `New_Line`, etc.) to the specified file: `Payroll.txt`, rather than to the VDU screen. Note that any previous contents of the output-file `Payroll.txt` will then be overwritten.

Note the convention which is used to identify data and results files which are used with a particular program, ie. use the extension `'dat'` for a data file, `'txt'` for a file containing text (which could be program results, or `'res'` is sometimes used instead).

When we have finished reading data from an input file we should 'tidy-up' by closing the file with the command:

```
CloseInput;
```

This closes the file and any subsequent input is then taken from the keyboard. Similarly an output file is closed with the command:

```
CloseOutput;
```

and any further output produced by the program is then sent to the VDU screen.

Sometimes we may want a program to read its input from a file and/or send its output to a file specified by the user at run-time (rather than a file whose name is specified by the programmer as above). In this case we can use versions of the `OpenInput` and `OpenOutput` procedures which take no parameters. For example if a program contains the step:

```
OpenInput;
```

then on execution, the program outputs:

```
 Input File>
```

on the VDU screen to prompt the user to enter the name of the file to be opened. The user then types the name of the desired file (not surrounded by double quote marks) and presses the return key. A similar prompt for the user to enter the name of the file to be used for output is produced by the step:

```
OpenOutput;
```

The procedures `OpenInput`, `OpenOutput`, `CloseInput` and `CloseOutput` are provided in the Ada library package `CS_File_IO` and before we can use them we must 'import' them into our program by means of the `WITH` and `USE` clauses:

```
WITH CS_File_IO;  USE CS_File_IO;
```

It is worth noting that if a program takes its input from a file, then obviously prompts to the user of the program (which might have been introduced if the program were to be used interactively) to input data values are not required (since a program which inputs its data values from a file is no longer interactive) and should therefore be omitted. Of course, all data values needed by the program should have been put into the input file (for example by using an editor) before the execution of the program is to commence.

You may have observed that the approach described here for input/output using files is different to that in the recommended course books by Feldman and Koffman or Rudd. This is due to the fact that many of the input and output procedures we use are not part of the Ada language, but can be made available to an Ada program in a fairly straightforward manner. In this course we have chosen to make them available through the library packages `CS_Int_IO`, `CS_Flt_IO` and `CS_File_IO`, rather than the approaches adopted by Feldman and Koffman or Rudd, because our approach combines simplicity and flexibility, and is therefore a little easier for you to use.

7.4 Final program

Finally we collect all the fully developed Ada procedures and program segments together, using the code letters (**A**, **B**, **C**, etc.) to ensure that everything is put in the correct place, add the punctuation, improve the layout of the results, and add further suitable explanatory comment when appropriate. We have also included the use of `OpenInput`, `OpenOutput`, etc. to input the data values from a file (since there may potentially be a large amount of data) and output the results to a file (since there may also potentially be a large number of results).

```
WITH Ada.Text_IO;
USE Ada.Text_IO;                  -- To import Put for text, and New_Line
WITH CS_Int_IO;  USE CS_Int_IO;  -- To import Get and Put for Integer I/O
WITH CS_Flt_IO;  USE CS_Flt_IO;  -- To import Get and Put for real I/O
WITH CS_File_IO; USE CS_File_IO; -- To import OpenInput, OpenOutput etc.

PROCEDURE EmployeesPay IS
    -- Example program for Unit 07 of Ada course.
    -- Written by L J Hazlewood, October 1993.
    -- Updated by A Barnes, August 1996.
    -- See Ada notes Unit [7.2] for the problem specification

    PROCEDURE ProcessOneEmployee ( EmployeeNo  : IN Integer;
                                   TimeWorked  : OUT Float;
                                   TakeHomePay : OUT Float ) IS
        -- To deal with this employee (ie. calculate the hours worked,
        -- gross pay, tax due and net pay - and output table row
        -- for this employee)
        Hours           : Integer;  -- Number of whole hours worked
        Mins            : Integer;  -- Number of minutes worked
        Rate            : Float;    -- Rate of pay in pounds per hour
        TaxCode         : Integer;  -- Tax code
        GrossPay        : Float;    -- Gross pay (before tax)
        TaxDue          : Float;    -- Amount of tax to be paid
        TaxThreshold    : Float;    -- Threshold for paying tax at all
        AnnualGrossPay  : Float;    -- 52 times the weekly gross pay
        AnnualTaxDue    : Float;    -- 52 times the weekly tax to be paid
    BEGIN
        -- Input Hours, Mins, Rate and TaxCode
        Get(Item => Hours);
        Get(Item => Mins);
        Get(Item => Rate);
        Get(Item => TaxCode);

        -- Calculate TimeWorked, GrossPay, TaxDue and TakeHomePay
        TimeWorked := Float(Hours) + Float(Mins) / 60.0;
        TaxThreshold := Float(10 * TaxCode);
        AnnualGrossPay := 52.0 * TimeWorked * Rate;
        GrossPay := AnnualGrossPay / 52.0;
        IF AnnualGrossPay <= TaxThreshold THEN
            AnnualTaxDue := 0.0;
        ELSE
            AnnualTaxDue := (AnnualGrossPay - TaxThreshold) * 0.25;
        END IF;

        TaxDue := AnnualTaxDue / 52.0;
        TakeHomePay := GrossPay - TaxDue;

        -- Output EmployeeNo, GrossPay, TaxDue and TakeHomePay
        New_Line;
        Put(Item => EmployeeNo, Width => 7);
        Put(Item => GrossPay, Fore => 10, Aft => 2);
        Put(Item => TaxDue, Fore => 7, Aft => 2);
        Put(Item => TakeHomePay, Fore => 9, Aft => 2);
        -- The above formats have been chosen to produce the output
        -- layout given in the output diagram in [7.2.2]

    END ProcessOneEmployee;
    PROCEDURE ProcessEmployeeRecords ( TotalHrs : IN OUT Float;
                                       TotalPay : IN OUT Float ) IS
        -- To process all the employee records (ie. outputting the rows of
        -- the table and incrementing the values of TotalHrs and TotalPay)
        EmployeeCode : Integer;
        HrsWorked    : Float;   -- Number of hours worked by this employee
        NetPay       : Float;   -- Net pay of this employee
    BEGIN
        Get(Item => EmployeeCode);
        WHILE EmployeeCode /= 0 LOOP
            -- Deal with this employee (ie. calculate the hours worked,
            -- gross pay, tax due and net pay - and output table row
            -- for this employee)
            ProcessOneEmployee ( EmployeeNo  => EmployeeCode,
                                 TimeWorked  => HrsWorked,
                                 TakeHomePay => NetPay );
            -- Update TotalHrs and TotalPay
            -- with the details for this employee
            TotalHrs := TotalHrs + HrsWorked;
            TotalPay := TotalPay + NetPay;

            Get(Item => EmployeeCode);
        END LOOP;
    END ProcessEmployeeRecords;

    PROCEDURE OutputAverageRate ( TotalHrs : IN Float;
                                  TotalPay : IN Float ) IS
        -- To produce average hourly rate of pay
        AverageNetRate : Float; -- Average net rate of pay per hour
    BEGIN
        AverageNetRate := TotalPay / TotalHrs;
        New_Line; New_Line;
        Put(Item => "Average net rate of pay per hour is ");
        Put(Item => AverageNetRate);
        New_Line;
    END OutputAverageRate;

    -- Global variable declarations
    TotalHrsWorked : Float;  -- Total hours worked for all employees
    TotalNetPay    : Float;  -- Total net pay for all employees

BEGIN -- EmployeesPay main program
    -- Open files for input and output
    OpenInput(FileName => "Employees.dat");
    OpenOutput(FileName => "Payroll.txt");

    -- Initialise TotalHrsWorked and TotalNetPay
    TotalHrsWorked := 0.0;
    TotalNetPay    := 0.0;

    -- Output heading for table
    Put(Item => "EmployeeΔΔΔGrossΔPayΔΔΔTaxΔDueΔΔΔΔNetΔPay");
    New_Line;

    -- Process all the employee records (outputting the rows of the table
    -- and incrementing the values of TotalHrsWorked and TotalNetPay)
    ProcessEmployeeRecords ( TotalHrs => TotalHrsWorked,
                             TotalPay => TotalNetPay );

    -- Produce average hourly rate of pay
    OutputAverageRate ( TotalHrs => TotalHrsWorked,
                        TotalPay => TotalNetPay );

    -- Close files used for I/O
    CloseInput;
    CloseOutput;
END EmployeesPay;
```

7.5 Summary of the methodology

We have introduced the above techniques as a framework for producing small to medium sized programs (ie. of the size you will encounter in a first course on programming). These techniques are not 'hard-and-fast', and after some practice you may find that you can vary the techniques somewhat to suit your own individual style. Although experienced programmers can produce correct programs in a less formal way altogether, to be able to do so it is necessary for them to draw upon many years of programming experience, and our 'semi-formal' approach should enable you to produce good software without extensive previous knowledge. Indeed a systematic methodology is so important in programming that for the purposes of this course and its associated examination you will be expected learn and apply the methodology described, so it is worth summarising its main points:

i) Think through problem.

ii) Consider the data and results by drawing and annotating input and output diagrams. If the program is to be an interactive one, these input and output diagrams could be combined into a single (interactive) 'conversation' diagram.

iii) Determine the main actions, particularly repetitions (optionally, note down).

iv) Write a top level design or outline algorithm which expresses the overall structure in a combination of English and Ada-like notation as appropriate.

v) Label each English phrase (ie. each undeveloped component), but do not label parts which are already developed using full Ada notation.

vi) Identify all global variables used by the steps of the top level design.

vii) Expand the undeveloped components (in any convenient order), often writing each step of the top level design as a procedure, writing Ada where what is needed is immediately clear, otherwise using further (but simpler and more precise) English phrase components. If the undeveloped components are expanded as procedures with parameters it is important to show both the procedure call (with actual parameters) as well as the procedure declaration (with formal parameters).

viii) As each component is refined note any variables which have been used. At the same time any input/output procedures used can be noted together with the package names from where they are to be imported.

ix) Continue the label/expand process until all non-trivial steps have been expanded into Ada.

As you become more experienced you will find that it is not always necessary to expand every English step into Ada, but rather the details of many 'obvious' components can be left until the whole program is assembled. English phrases describing simple input/output and initialisations are ones which could be treated in such a manner.

x) Collect all the Ada together, using the labels to ensure correct ordering and including comment where judged appropriate. It is usual to comment each major activity of the program (ie. each major step identified in the expansion process), and to acknowledge authorship of the program in a comment immediately after the program heading.

xi) Add the semi-colon punctuation, ensuring you have a legal Ada program.

xii) Modify any output steps to improve the layout of results. Here you will need to refer to the output diagram in order to determine the formatting parameters necessary to produce correctly aligned output.

xiii) Add any further explanatory comments.

xiv) Finally, read through the whole program to check for any obvious blunders, then ... test it.

Note Don't expect always to make the correct decisions about how to expand each step. If you get completely stuck, or if things seem to be getting very 'messy', you must be prepared to 'back-up', ie. to re-think and, where necessary, revise earlier decisions.