Computer Science 301 - 2015

Programming Language Translation

Practical 1, Week beginning 24 August 2015

This prac is due for submission by lunch time on your next practical day, correctly packaged in a transparent folder as usual (**unpackaged and late practical submissions will not be accepted - you have been warned**). Pracs should please be deposited in the hand-in box outside the lab. Only **one set of listings** is needed for each group, but please enclose as many copies of the cover sheet as are needed, one for each member of the group. These will be returned to you in due course.

Objectives:

In this practical you are to

- acquaint yourselves with some command line utilities, with various editors, interpreters and compilers;
- investigate various qualities of some computer languages and their implementations, including C, C++, C#, Pascal and Parva.
- obtain some proficiency in the use of the various library routines that will be used later in the course.

The exercises for this week are not really difficult, although they may take longer than they deserve simply because you may be unfamiliar with the systems.

Copies of this handout, the cover sheet, the Parva language report, and descriptions of the library routines for input, output, string handling and set handling in C# are available on the course web site at http://www.cs.ru.ac.za/courses/CSc301/Translators/trans.htm .

Outcomes:

When you have completed this practical you should understand

- how and where some languages are similar or dissimilar;
- how to use various command line compilers and decompilers for these languages;
- what is meant by the term "high level compiler" and how to use one;
- how to measure the relative performance of language implementations;
- the elements and limitations of programming in Parva;
- how to use I/O and set handling routines in C#;
- a little more about simple library design in C#.

To hand in:

This week your group is required to hand in, besides the individual cover sheets for each member:

- One copy of the listings of your solutions to the programming exercises in tasks 5, 12, 13, 14 and 15 produced by using the `LPRINT` utility from the command line (which prints listings economically).
- Electronic copies of your source code for those exercises, using the electronic submission system.
- Your commentary and solutions to the questions posed below. Part of this consists of results that you should be able to collect and record on the back of the cover sheet by the end of the first afternoon.

Keep the cover sheet and your solutions until the end of the semester. Check carefully that your mark has been entered into the Departmental Records.

You are referred to the rules for practical submission which are clearly stated in our Departmental Handbook. However, for this course pracs must be posted in the "hand-in" box outside the laboratory before the next practical session and not given to demonstrators during the session.

A rule not stated there, but which should be obvious, is that you are not allowed to hand in another student's or group's work as your own. Attempts to do this will result in (at best) a mark of zero and (at worst) severe disciplinary action and the loss of your DP. You are allowed - even encouraged - to work and study with other students, but if you do this you are asked to acknowledge that you have done so on **all** cover sheets and with suitable comments typed into **all** listings. You are expected to be familiar with the University Policy on Plagiarism, which you can consult at

`http://www.ru.ac.za/media/rhodesuniversity/content/institutionalplanning/documents/Plagiarism.pdf`

Before you begin

In this practical course you will be using a lot of simple utilities, and will usually work at the "command line" level rather than in a GUI environment. Note in particular:

- After logging on, get to the DOS command line level by using the `Start -> Command prompt` sequence if you don't already have a shortcut (it is probably worth creating a short cut).

- Listings are conveniently produced by using the `LPRINT` command from a command window, for example

  ```
  LPRINT  Queens.cs  Queens.pav
  ```

  The listings come out in a small font which enables long lines to be read easily and with narrow line spacing (so that you get more listing for your money). **Please use this utility, which prints listings in a small courier font to produce all listings submitted on this course, as it makes my job of reading the submissions much easier.** Program listings in "proportional font" are awkward to read.

- Before you can use `LPRINT` you may need to "capture" the printer, after opening a command window, by using the command `UNMAP` (if necessary) followed by `PRINTEAST` or `PRINTWEST` as appropriate.

Copies of software for home use

For this prac it is recommended that you simply work in the Hamilton lab, rather than begging, borrowing or stealing copies of a whole host of software for home use. In future pracs you will mostly use C# only, and the prac kits will, hopefully, contain all the extras you need.

Task 1 (a trivial one)

We shall make use of zipped prac kits throughout the course; you will typically find sources for each week's prac in a file `pracNN.zip` on the server. Copy `prac1.zip` as needed for this week, either directly from the server on `I:\CSC301\TRANS` (or by using the WWW link on the course page), and extract the sources when you need them, into your own directory/folder, perhaps by using `UNZIP` from a command line prompt.

```
j:> copy i:\csc301\trans\prac1.zip
j:> unzip  prac1.zip
```

In the past there has occasionally been a problem with running applications generated by the C# compiler if these are stored on the network drives. If you have difficulties in this regard, for those parts of the practical that involve the use of C#, work from the local D: drive instead. After opening a command window, log onto the D: drive, create a working directory and unpack a copy of the prac kit there:

```
j:> d:
d:> md d:\G13T1111
d:> cd d:\G13T1111
d:> unzip I:\csc301\trans\prac1.zip
```

In the prac kit you will find various versions of a famous program for finding a list of prime numbers using the method known as the Sieve of Eratosthenes. You will also find various versions of a program for solving the N

Queens problem, some "empty" programs, some other bits and pieces, including a few batch files to make some of the following tasks easier and a long list of prime numbers (`primes.txt`) for checking your own!

Task 2 The Sieve of Eratosthenes in Pascal

You may not be a Pascal expert, but in the kit you will find some Pascal programs, including `SIEVE.PAS` that determine prime numbers using a Boolean array to form a "sieve". Study and compile these programs - you can do this from the command line quite easily by issuing commands like

```
FPC SIEVE.PAS
FPC FIBO.PAS
FPC EMPTY.PAS
```

to use the 32-bit Windows version of the Free Pascal compiler. Make a note of the size of the executables (use the commands `DIR SIEVE.EXE`, `DIR FIBO.EXE` and `DIR EMPTY.EXE`).

How do the sizes of the executables compare? Why do you suppose the "empty" program produces the amount of code that it does?

```
PROGRAM Sieve (Input, Output);
(* Sieve of Eratosthenes for finding primes 2 <= N <= Max
   P.D. Terry, Rhodes University, 2015 *)

  CONST
    Max = 32000                                (* largest number allowed *);
  TYPE
    SIEVES = ARRAY [2 .. Max] OF BOOLEAN;
  VAR
    I, N, K, Primes, It, Iterations : INTEGER  (* counters *);
    Uncrossed : SIEVES                         (* the sieve *);
    Display : BOOLEAN;
  BEGIN
    Write('How many iterations '); Read(Input, Iterations);
    Display := Iterations = 1;
    Write('Supply largest number to be tested '); Read(Input, N);
    IF N > Max THEN BEGIN
      WriteLn(Output, 'N too large, sorry'); HALT
    END;
    WriteLn(Output, 'Prime numbers between 2 and ', N);
    WriteLn(Output, '-----------------------------------');
    FOR It := 1 To Iterations DO BEGIN
      Primes := 0 (* no primes yet found *);
      FOR I := 2 TO N DO                       (* clear sieve *)
        Uncrossed[I] := TRUE;
      FOR I := 2 TO N DO                       (* the passes over the sieve *)
        IF Uncrossed[I] THEN BEGIN
          IF Display AND (Primes MOD 8 = 0) THEN WriteLn;    (* ensure line not too long *)
          Primes := Primes + 1;
          IF Display THEN Write(Output, I:6);
          K := I;                              (* now cross out multiples of I *)
          REPEAT
            Uncrossed[K] := FALSE; K := K + I
          UNTIL K > N
        END;
      IF Display THEN WriteLn
    END;
    Write(Primes, ' primes')
  END.
```

Note that the Sieve programs are written so that requesting 1 iteration displays the list of the prime numbers; requesting a large number of iterations suppresses the display, and simply "number crunches". This is for use in Task 9. In all cases the program reports the number of prime numbers computed. So, for example, a single iteration with an upper limit of 20 will report that there are 8 primes smaller than 20 - 2, 3, 5, 7, 11, 13, 17 and 19.

Here is something more demanding:

Prime numbers are those with no factors other than themselves and 1. But the program does not seem to be looking for factors!

Look at the Pascal code carefully. How does the algorithm work? Why is it deemed to be particularly efficient? How much (mental) arithmetic does the "computer" have to master to be able to solve the problem?

Something more challenging - find out how large a prime number the program can really handle, given a limit of 32000 on the size of the Boolean array. What is the significance of this limit? How many prime numbers can you find smaller than 20000? *Hint*: you should find that funny things happen when the requested "largest number" N gets too large, although it may not immediately be apparent. Think hard about this one!

Task 3 The Sieve in C or C++

The kit also includes C and C++ versions of these programs. Compile these and experiment with them in the same way, with the 32-bit Windows compilers:

`BCC SIEVE.C`	(using the Borland compiler in C mode)
`BCC SIEVE.CPP`	(using the Borland compiler in C++ mode)
`CL SIEVE.C`	(using the WatCom compiler in C mode)
`CL SIEVE.CPP`	(using the WatCom compiler in C++ mode)

Once again, make a note of the size of the executables, and in particular, compare them with the earlier versions. Can you think of any reason why the differences are as you find them? How large a prime number can you handle now? How many prime numbers can you find smaller than 20000? If there is a difference, explain it.

Task 4 See C#

You can compile the C# versions of these programs from the command line, for example:

```
csharp Sieve.cs
```

Make a note of the size of the ".NET assemblies" produced (`SIEVE.EXE, EMPTY.EXE` and `FIBO.EXE`). How do these compare with the other executables? What limit is there now to the largest prime you can find?

Task 5 Progress to Parva

On the course web page you will find a description of Parva, a toy language very similar to C, and a language for variations on which we shall develop a compiler and interpreter later in the course. The main difference between Parva and C/Java/C# is that Parva is stripped down to bare essentials. Learn the Parva system by studying the language description where necessary, and trying the system out on the supplied code (`SIEVE.PAV`, `FIBO.PAV` and `EMPTY.PAV`).

There are various ways to compile Parva programs. The easiest is to use a command line command:

`Parva Sieve.pav`	simple error messages
`Parva -o Fibo.pav`	slightly optimized code
`Parva -l Sieve.pav`	error messages merged into `listing.txt`

The code you have been given for `Sieve.pav` has some deliberate errors, so you will have to find and correct these. All in a jolly afternoon's work! Hand in listings of your final corrected `Sieve.pav` produced with the `LPRINT` command.

Task 6 A blast from the past - some 1980s vintage 16 bit DOS compilers

We have some early compilers, two of which you might like to explore.

These compilers will not run directly on 64 bit Windows systems with 4 GB of memory. They were constructed at a time when 64 KB of memory was considered "large", and as such they are masterpieces of software engineering.

We can run 16 bit software in various ways. The simplest - adequate for our purpose - is to run the DOS 6.2 emulator known as `DosBox`, as a Windows application. To do this, first copy a shortcut to your desktop. The shortcut can be found by navigating to the `I:\utils` folder or in the unpacked kit. Once you have it on the desktop, clicking it will open an 80 x 25 text window, set up a few paths to executables, and present you with a DOS prompt.

(Advance warning - sometimes the mouse pointer disappears when you are using DOSBox. The Mouse does not work within the system at all. If you lose the mouse, CTRL+F10 usually gets it back again).

At this prompt you can execute various familiar DOS commands, like `DIR` and `DEL`, but you cannot execute 32 bit software designed for Windows. No matter - you can edit files on the D: drive using 32 bit software like NotePad++, and they will be visible in the DosBox window for further processing.

Turbo Pascal 6.0

Start by recompiling the Pascal source code mentioned previously, executing the code, and making the same measurements as before, using commands like

```
TP6 SIEVE.PAS
TP6 FIBO.PAS
TP6 EMPTY.PAS
```

and comment on any major differences that you notice from your use of Free Pascal. TP6 executes a version of Turbo Pascal last released in 1990, by which stage the Pascal language it compiled was quite a lot more complex than the original language of 1970. Then repeat the exercise using a slightly different setting of the same compiler which is supposed to produce slightly faster code.

```
TP6O SIEVE.PAS
TP6O FIBO.PAS
TP6O EMPTY.PAS
```

Turbo Pascal 1.0

For some real fun, try out the original Turbo Pascal system, by giving the command

```
TURBO
```

This system is all contained in 39 KB, and that includes the compiler, a full screen editor, and runtime support. Once the first screen loads you can import a source file, then press C to compile it and R to run the compiled program. E will invoke the Editor and F1 will take you back to the main screen.

Everything - even the object code - is kept in RAM, which partly explains the blazing speed. To save the machine code version as a .COM file (another blast from the past) you will have to use the `Options` available in a fairly obvious way.

Turbo Pascal, in its day (about 1984) revolutionized the teaching of Computer Science. It was possible at last to do really nice programming on the machines of that era, which by today's standards were really small and slow.

Task 7 High level translators

It may help amplify the material we are discussing in lectures if you put some simple Parva programs through a high-level translator, and then look at, and compile, the generated code to see the sort of thing that happens when one performs automatic translation of a program from one high-level language to another.

We have a home-brewed system that translates Parva programs into C#. The system is called `Parva2ToCSharp`. It is still under development - meaning that it has some flaws that we might get you to repair in a future practical. Much of the software for this course has been designed expressly so that you can have fun improving it.

You can translate a Parva program into C# using a command of the form exemplified by

```
Parva2ToCSharp Sieve.pav
Parva2ToCSharp Fibo.pav
Parva2ToCSharp Empty.pav
```

A C# source file is produced with an obvious name; this can then be compiled with the C# compiler by using commands of the form:

```
csharp Sieve.cs
```

and executed with the usual commands of the form

```
Sieve
```

Take note of, and comment on, such things as the kind of C# code that is generated (is it readable; is it anything like you might have written yourself?), and of the relative ease or difficulty of using such a system. You might also like to comment on the performance of the programs that are produced, by running timing tests as in Task 9.

Another program in the kit is a variation on the example found in the book on page 90. This has an intentional weakness. See if you can spot it!

Run the Parva compiler directly:

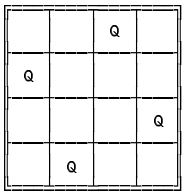
```
Parva voter.pav
```

Then try translating the program to C# and compiling and running that:

```
Parva2ToCSharp voter.pav
csharp voter.cs
voter
```

Task 8 The N Queens problem

In the kit you will find various equivalent programs that attempt to solve the famous *N Queens* problem. These use a back-tracking approach to determine how to place N Queens on an N * N chess board in such a way that no Queen threatens or is threatened by any other Queen - noting that a Queen threatens another Queen if the two pieces lie on a common vertical, horizontal or diagonal line drawn on the board. Here is a solution showing how 4 Queens can be placed safely on a 4 * 4 board:



Compile one or more of these programs and try them out. For example

```
FPC  QUEENS.PAS
QUEENS
```

There are three versions written in each of Pascal, Parva and C#. One version uses parameters to pass information between the routines, another version uses global variables, and the third tabulatess the number of solutions for a range of board sizes. As in the case of the Sieve, one can get these to "number crunch" by specifying a large number of iterations. At some stage you could usefully spend a little time studying `Tutorial 1` on the web site, which explains the technique behind the solution.

In the kit you will also find some variations on Parva programs for solving the N-Queens problem. Not all of these programs are correct. Try compiling them with the Parva compiler first, and observe the outcome. Then try converting them to C# (without editing them in any other way), observe the outcome, and compile and run the

C# programs, such as are produced. Do the Parva programs need to be acceptable to the Parva compiler if they are to be acceptable to `Parva2ToCSharp`? What can you learn from this exercise about using a tool of this nature? Have we made `Parva2ToCSharp` "as simple as possible, but no simpler"? Do we have to, or could we, make it simpler still? Do we have to make it more complex? Why - or why not?

Summarize your thoughts in a short essay which should form part of your submission.

Task 9 - How fast/slow are various language implementations?

Different compilers - even for very similar programs - can produce code of very different quality. In particular "interpretive" systems (of which the Parva implementation is one example) produce programs that run far more slowly than do "machine" or "native" code systems. Carry out some tests to see these effects for yourselves, and how severe they are, by comparing the execution times of some of the programs.

Summarize your findings on page 2 of the cover sheet, explaining briefly how you come to the figures that you quote. Is Java better/worse than C# (the source code in each case is almost identical)? Do 16-bit compilers fare better or worse than 32-bit compilers?

Hint: the machines in the Hamilton Labs are *very* fast, so you should try something like this: Experiment choosing sizes for the sieve or chessboard (and a suitable large number of iterations) that will produce measurable times of the order of a few seconds. Use the same sizes/counts with each program - for best results "hard coding" them into the source code, and then use the "timer.bat" script to run the executables and time them using the computer's clock.

Task 10 - Reverse Engineering - disassembly

In lectures you were told of the existence of decompilers - programs that can take low-level code and attempt to reconstruct higher level code. There are a few utilities available for experiment.

`ildasm`	a decompiler that creates CIL assembler source from a .NET assembly
`ilasm`	an assembler that creates a .NET assembly from CIL assembler source
`peverify`	a tool for verifying .NET assemblies

Try out the following experiments or others like them:

(a) Compile `Sieve.cs` and then disassemble it

```
csharp Sieve.cs
Disassemble Sieve
```
(calls `ildasm` from a batch file, produces `Sieve.cil`)

and examine the output, which will appear in `Sieve.cil`

(b) Reassemble `Sieve.cil`

```
Reassemble Sieve
```
(calls `ilasm` from a batch file, produces new `Sieve.exe`)

and try to execute the resulting class file

```
Sieve
```

(c) Be malicious! Corrupt `Sieve.cil` - simply delete a few lines in the section that corresponds to the *Main* function (use lines with opcodes on them). Try to reassemble the file (as above) and to re-run it. What happens?

(d) Experiment with the .NET verifier after step (b) and again after step (c)

```
NetVerify Sieve
```
(calls `peverify` from a batch file)

Task 11 - A Cautionary Tale

A few years ago I set some simple programming exercises for this class to do in this practical, and provided an executable version of a solution to assist the class in understanding the problem. I had not reckoned with the guile of some of your predecessors who, rather than write their own program, decompiled my executable and

handed that in instead. Caution: After years of experience I can spot fraudulent behaviour very quickly. I'd thought I was safe because I had not let the class know about .NET decompilers, but my colleague Professor Google had obviously been consulted. The group pointed me very quickly to a tool written by JetBrains, known as dotPeek. This is very easy to use and quite fun, so I have installed it on the lab machines temporarily for use in this practical.

As a variation on task 10, try using it to decompile `Sieve.exe`, `Fibo.exe` and `Queens.exe` (having first recompiled the C# versions, of course):

```
dotPeek Sieve.exe                (runs dotPeek from a batch file)
dotPeek Fibo.exe                 (runs dotPeek from a batch file)
```

and navigate to the decompiled source code. Go on to save this under different names (for example `Sieve2.cs` and `Queens2.cs`) and then recompile these sources to see if you can get working executables.

What happens if you try to decompile an executable that was not produced from a .NET compatible compiler? Try it.

Caution: Don't try to run tools like this to decompile programs I might give you in executable form! Nevertheless, I am sure that you can see that they might have a great deal of use - for example in legitimately recreating source that might have got lost. I have not, myself, explored the options of dotPeek farther than I needed to make the suggestions above, but feel free to experiment.

Time to think for yourselves

And now for something completely different! (where have you heard that before?)

- and don't use a search engine!

Nothing you have done so far should have extended your programming talents very much. To get the old brain cells working a little harder, turn your minds to the following.

It is important that you learn to use the IO libraries `InFile, OutFile` and `IO`. These will be used repeatedly in this course. Please do not use other methods for doing I/O, or spend time writing lots of exception handling code.

Pat Terry's problems are sometimes reputed to be hard. They only get very hard if you don't think very carefully about what you are trying to do, and they get much easier if you think hard and spend time discussing the solutions with the tutors or even the Tyrant himself. His experience of watching the current generation of students suggests that some of you get beguiled by glitzy environments and think that programs just "happen" if you can guess what to click on next. Don't just go in and hack. It really does not save you any time, it just wastes it. Each of the refinements can be solved elegantly in a small number of lines of code if you think them through carefully before you start to use the editor, and I shall be looking for elegant solutions.

Remember a crucial theme of this course - "Keep it as simple as you can, but no simpler".

I am looking for imaginative, clear, simple solutions to the problems.

Task 12 Creative Parva programming - the "hailstone" sequence

Suppose N is a positive number that starts a sequence defined by the following rules: If a term M is odd, the next term in the sequence is $3M + 1$. If a term M is even, the next term is $M / 2$. The sequence terminates when $M = 1$. For example, the sequence that starts with $N = 6$ is as follows:

6 3 10 5 16 8 4 2 1

and in this particular case the length of the sequence is 9. Write a function procedure that, given N, will return the length L of the sequence beginning with N. Then continue to write a little program that uses this function to determine the smallest positive integer N that produces a sequence length L greater than K, where K is a number

used as input data. For example, for $K = 12$, the result should be $N = 7$. 7 is the smallest positive integer that generates a sequence with more than 12 members.

Task 13 Something more creative - "Pig Latin" in C#

fIay ouyay ancay eadray histay, ouyay ancay robablypay igurefay utoay hatway hetay roblempay siay. uorYay eallyray eednay otay riteway wotay rogramspay; neoay otay ncodeeay ndaay neoay otay ecodedday enntencessay!

Hint: To read in a sequence of words, make use of the `ReadWord` method in the `InFile` library - there is no need to read a whole line and "tokenize" it yourself.

Task 14 Something more creative - Another look at the infamous Sieve in C#

The code you have been given for the Sieve of Eratosthenes makes use of a Boolean array. A little thought should convince you that conceptually it is using this array as a "set". In the weeks ahead we shall use the set concept repeatedly, so to get you into this frame of mind, develop a C# version of the Sieve idea using objects of the `IntSet` type, details of which can be found on the web site - and some simple examples of use can be found in the program at the end of this handout. You should find this very simple, once you get the idea.

Task 15 Something more creative - A simple graphics class in C#

Some months ago, when looking for new ideas for this practical, I thought it might be fun to draw some simple graphs from within a C# program - in particular to graph the number of solutions to the N-Queens problem as a function of board size. I did what all good/bad code thieves do - searched the web for a simple graphics library that I could use, and to my amazement could not find one that could be used from the simple console based applications that I favour in this course. Oh, yes, if one wanted to live inside Visual Studio there was probably something, but I have never lived in Visual Studio and didn't wish to do so now.

Peter Wentworth came to my rescue, and after hacking around inside and outside of Visual Studio, came up with the classes supplied in the kit, which allow one to write simple programs like the following:

```
using System;
using System.Drawing;
using GraphicsLib;

class Program {
  // Elementary use of GraphicsLib
  // P.D. Terry, 2015

  static void Main(string[] args) {
  // The calls to Console routines are merely to allow the graphics window to remain
  // "open" so that the user can study the progress of the calls to the graphics routines.

    Console.Write("\nHit Enter to Start!");
    Console.ReadLine();

    GraphicsWindow gw = new GraphicsWindow(400, 600);
    gw.DrawLine(30, 60, 100, 100);

    Console.Write("\nHit Enter to Continue!");
    Console.ReadLine();

    gw.SetPenColor(Color.Red);
    gw.DrawLine(100, 100, 300, 400);

    Console.Write("\nHit Enter to Exit!");
    Console.ReadLine();

  } // Main

} // Program
```

You will note that there is a low-level class which deals with threads and the like, and then a higher-level class which might be deemed incomplete, in that other useful methods might be added to it - for example to draw a circle centred on a given point and with a given radius, and a method to draw a graph linking a set of x,y coordinates stored in two parallel arrays. See if you can modify the C# program Sieve2.cs to produce the sort of

graph I had in mind. Hint: it would be better to plot *log(solutions)* against board size rather than *solutions* directly.

Ah, but there is another, hidden agenda. Keep it as simple as you can, but no simpler! Every year some students come up with improvements to the code they see in this course. Maybe some of you are much better acquainted with graphics in C#? Can you improve on Peter Wentworth's core library - while still striving for greater simplicity? Can you criticize this class and point out any real shortcomings in it?

Demonstration program showing use of InFile, OutFile and IntSet classes

This code is to be found in the file `SampleIO.cs` in the prac kit.

```
// Program to demonstrate Infile, OutFile and IntSet classes
// P.D. Terry, Rhodes University, 2015

using Library;
using System;

class SampleIO {

  public static void Main(string[] args) {
  //                                      check that arguments have been supplied
    if (args.Length != 2) {
      Console.WriteLine("missing args");
      System.Environment.Exit(1);
    }
  //                                      attempt to open data file
    InFile data = new InFile(args[0]);
    if (data.OpenError()) {
      Console.WriteLine("cannot open " + args[0]);
      System.Environment.Exit(1);
    }
  //                                      attempt to open results file
    OutFile results = new OutFile(args[1]);
    if (results.OpenError()) {
      Console.WriteLine("cannot open " + args[1]);
      System.Environment.Exit(1);
    }
  //                                      various initializations
    int total = 0;
    IntSet mySet = new IntSet();
    IntSet smallSet = new IntSet(1, 2, 3, 4, 5);
    string smallSetStr = smallSet.ToString();
  //                                      read and process data file
    int item = data.ReadInt();
    while (!data.NoMoreData()) {
      total = total + item;
      if (item > 0) mySet.Incl(item);
      item = data.ReadInt();
    }
  //                                      write various results to output file
    results.Write("total = ");
    results.WriteLine(total, 5);
    results.WriteLine("unique positive numbers " + mySet.ToString());
    results.WriteLine("union with " + smallSetStr
                      + " = " + mySet.Union(smallSet).ToString());
    results.WriteLine("intersection with " + smallSetStr
                      + " = " + mySet.Intersection(smallSet).ToString());
    results.Close();
  } // Main

} // SampleIO
```