Computer Science 301 - 2017

Programming Language Translation

Practical 1, Week beginning 17 July 2017

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 library routines for input, output, string, list, dictionary 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 make simple measurements of 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 10 to 12, 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 attached 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 giving these names 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 executing `CMD.exe`. You will be doing this many times in the next few weeks, so set yourself up a shortcut on your desktop that points to the program `c:\windows\system32\cmd.exe` and starts execution in your home folder, or in `D:\`. (You may already have such a shortcut?)

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

```
LPRINT  SieveCS.cs  SievePas.pas
```

  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" can be very awkward to read.

- Before you can use `LPRINT` you will need to "capture" the printer, after opening a command window, by using the command `UNMAP` (if necessary) followed by `PRINTHAMILTON` (only once).

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:\G14T1111
d:> cd d:\G14T1111
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 tabulating the Fibonacci sequence, some "empty" programs, three programs for investigating the N-Queens problem, and some other bits and pieces, including a few batch files to make some of the following tasks easier, as well as 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 `SievePas.pas`, one that determines prime numbers using a Boolean array to form a "sieve". Study and compile this program - you can compile this from the command line quite easily by issuing the command

```
FPC SievePas.pas   or  FPO SievePas.pas   or  FPS SievePas.pas
```

to use a Windows version of the Free Pascal compiler (the three variations set up various compile-time options). After compilation you can execute the program with the command:

```
SievePas
```

The Pascal source reads as follows.

```
PROGRAM Sieve (Input, Output);
(* Sieve of Eratosthenes for finding primes 2 <= N <= Max
   P.D. Terry, Rhodes University, 2017 *)

  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 all the Sieve programs in this kit have been 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 a later task. 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, namely 2, 3, 5, 7, 11, 13, 17 and 19.

Here is something more interesting and, perhaps, puzzling:

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? What aspects of (mental) arithmetic does the "computer" have to master to be able to solve the problem?

The prac kit contains a file `PRIMES.TXT` listing all the 9592 prime numbers below 100,000.

As something more challenging - find out how large a prime number this program can really handle, given a limit of 32000 on the size of the Boolean array. What is the significance of this largest prime? 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 - 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 (`SievePav.pav`)

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

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

The code you have been given has some deliberate errors, so you will have to find and correct these. All in a jolly afternoon's work! How does the Parva version behave when you try to find large prime numbers?

By the end of the afternoon - Hand in a listing of your final corrected `SievePav.pav` - produced with the `LPRINT` command as always.

Task 4 - A blast from the past - some 1980s vintage 16 bit DOS compilers

Pascal came into favour in the early 1970s. and was the dominant academic language until the late 1980s. and is still popular in some circles. We have some early Pascal compilers, two of which we would like you to explore.

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

We can run 16 bit software in various ways, the simplest - adequate for our purpose - being to run a "DOS emulator" as a Windows application. We have two of these; try both. **DosBox** was apparently developed to allow users of 64 bit Windows to run old MS-DOS games, while **VDosPlus** was developed from an earlier **VDos** to allow users of 64 bit Windows systems to run non-gaming (ie serious) old MS-DOS applications. VDosPlus also seems capable of running 32 bit applications.

To run the **DosBox** emulator, 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).

To run the **VDosPlus** emulator simply give the command `VDosPlus` from within a Command window. It will also open an 80 x 25 text window, set up a few paths to executables, and present you with a DOS prompt.

At the DOS prompt in either emulator window you can execute various familiar DOS commands, like `DIR` and `DEL`. You cannot execute 32 bit software designed for Windows under DosBox although you seem to be able to do so under VDosPlus. 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 SievePas.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 SievePas.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 (KiloBytes, not MegaBytes!), and that includes the compiler, a full screen editor, and runtime support. Once the IDE screen loads you can, firstly, import a source file as a "workfile" (press W to do this), then press C to compile it and R to run the compiled program. E will invoke the Editor and F1 will take you back from the editor 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 very small and slow. if you are interested in computer history, look up the Wikipedia article on it.

Task 5 - How much code do various compilers generate?

Different compilers - even for the same source program - can produce very different sizes of executable versions of the programs they compile. To explore this, fill in the chart on the sheet provided of the sizes of the .exe files which the various compilers produce. Compile each program in turn and then use the DIR command to see the size of the executable, for example

```
FPC SievePas.pas
DIR SievePas.exe

FPC EmptyPas.pas
DIR EmptyPas.exe
```

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

The kit also includes C#, C and C++ versions of these programs. Compile these with the 32-bit Windows compilers:

```
CSHARP SieveCS.CS      (using the C# career)
DIR SieveCS.exe

BCC SieveC.C           (using the Borland compiler in C mode)
DIR SieveC.exe
BCC SieveCPP.CPP       (using the Borland compiler in C++ mode)
DIR SieveCPP.exe

CL  SieveC.C           (using the WatCom compiler in C mode)
DIR SieveC.exe
CL  SieveCPP.CPP       (using the WatCom compiler in C++ mode)
DIR SieveCPP.exe
```

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 6 - 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 `Parva2ToCSharp` that translates Parva programs into C#. 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 SievePav.pav
Parva2ToCSharp FiboPav.pav
Parva2ToCSharp EmptyPav.pav
```

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

```
csharp SievePavp2c.cs
```

and executed with the usual sort of command:

```
SievePavp2c
```

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.

Another program in the kit is a variation on the example found in the book at the end of Chapter 7. This has an intentional weakness. See if you can spot it!

Run the Parva compiler directly:

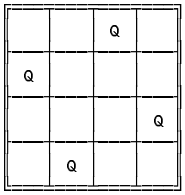
```
Parva voter.pav
```

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

```
Parva2ToCSharp voter.pav
csharp voterp2c.cs
voterp2c
```

Task 7 - 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 these programs and try them out. For example

```
FPC  Queens1.pas
Queens1
```

There are three versions written in each of Pascal and C#. One version uses parameters to pass information between the routines, another uses global variables, the third calculatess the number of solitions for an increasing sequence of board sizes. At some stage you could usefully spend a little time studying `Tutorial 1` on the web site, which explains the technique behind the solution. The idea is to number-crunch!

In the kit you will also find these variations in Parva. Not all of these programs are correct. Try compiling them with the Parva compiler first, observe the outcome and make any corrections needed. Do Parva programs need to be acceptable to the Parva compiler if they are also 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 8 - How efficient is the code generated by 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 the attached page, explaining briefly how you come to the figures that you quote. Do C or C++ produce better/worse performance than C# (the source code in each case is almost identical)? Do 16-bit compilers fare better or worse than 32-bit compilers? Why do you think this is?

Hint: the machines in the Hamilton Labs are *very* fast, so you should try something like this: Experiment choosing sizes for the Sieve or N-Queens or Fibonacci program (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 so as not to measure the I/O time and reaction time, and then use the "TIMER.BAT" script to run the executables and time them using the computer's clock.

Task 9 - A Cautionary Tale

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

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.

Try using dotPeek to decompile `SieveCS.exe` and `FiboCS.exe` (having first compiled these from the source code of the C# versions, of course). Give commands like:

```
dotPeek SieveCS.exe          (runs dotPeek from a batch file)
dotPeek FiboCS.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 `Fibo2.cs`) and then recompile these sources to see if you can get working executables.

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

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.

Interlude : Time to write some decent code 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 or using other ways to extract tokens from complete lines.**

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 exercises 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.

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 10 - One way of speeding up tedious recursion

Consider the following program, which is the Parva version of one of those in the kit: (`FiboPav.pav`):

```
// Print a table of Fibonacci numbers using (slow) recursive definition
// P.D. Terry,  Rhodes University, 2017

  int fib(int m) {
  // Compute m-th term in Fibonacci series 0,1,1,2 ...
    if (m == 0) return 0;
    if (m == 1) return 1;
    return fib(m-1) + fib(m-2);
  } // fib

  void main() {
    int limit;
    read("Supply upper limit ", limit);
    int i = 0;
    while (i <= limit) {
      write(i, "\t", fib(i), "\n");
      i = i + 1;
    } // while
  } // main
```

If you compile and run this for a fairly large *Limit* you will easily see that it takes longer and longer as *i* increases. In fact, the algorithm is easily shown to be $O(1.6^N)$.

Of course, if all you want is a simple table of Fibonacci numbers and not a text-book demonstration of a recursive function, it is much simpler and faster to proceed as follows (`fib1.pav`):

```
// Print a table of Fibonacci numbers using (fast) iterative method
// P.D. Terry,  Rhodes University, 2017

  void main() {
    int
      term   = 0,
      first  = 0,
      second = 1,
      limit;

    read("Supply upper limit ", limit);
    write(term, "\t", first, "\n");
    while (term < limit) {
      term = term + 1;
      write(term, "\t", second, "\n");
      int next = first + second;
      first  = second;
      second = next;
    } // while
  } // main
```

or, perhaps, with the *while* loop expressed using only two variables, as follows (`fib2.pav`):

```
    while (term < limit) {
      term = term + 1;
      write(term, "\t", second, "\n");
      second = first + second;
      first  = second - first;
    }  // while
```

Well, suppose we *do want* a recursive function (perhaps *you* don't, but this is not a democracy!). The reason that the program runs ever slower is, of course, that evaluating the function for a large argument effectively sets up a tree of calls to functions whose values have already been computed previously many, many times. Try drawing the tree, if you need convincing.

A way of improving on `FibPav.pav` is as follows. Each time a value is computed for a hitherto unused value of the argument, store the result in a (global) array indexed by the value of the argument as well as "returning" it. Then, if another call is made for this value of the argument, obtain the value of the function from the array rather than making the two interior recursive calls. So, for example, if one had evaluated `fib(2)` ... `fib(5)` the array would contain:

0	1	1	2	3	5	0	0	0	0	0	0	0	.....
0	1	2	3	4	5	6	7	8	9	10	11	12	.....

and evaluating `fib(6)` would amount to a single addition only, while evaluating `fib(10)` at this point still be much faster than in the simple version in `FiboPav.pav` (why?).

Wait! This is a very well known exercise. You are bound to be able to find many solutions on the web. Resist the temptation. Work out your own solution, please.

(The same goes for some other exercises in this course. You will get far more out of them if you try them by yourselves, or within your group, rather than hunting for previous solutions.)

Try running the program with the `while` loop in the `main` function going aup, and then try it going down. Does the direction make a difference? Why or why not?

This sort of optimization can be very useful in cases where a recursive function might be called many thousands of times in a system for values of an argument within a known range (so that the array size is easily fixed). The technique has a special name. After you have developed your solution, see if you can find out what this name is.

Task 11 - Creative programming - Goldbach's conjecture

Goldbach's conjecture is that every even number greater than 2 can be expressed as the sum of two prime numbers. Write a program that examines every even integer *N* from 4 to *Limit*, attempting to find a pair of prime numbers (*A* , *B*) such that $N = A + B$. If successful the program should write *N, A* and *B*; otherwise it should

write a message indicating that the conjecture has been disproved. This might be done in various ways. Since the hidden agenda is to familiarize you with the use of a class for manipulating "sets", you **must use a variation on the sieve method** suggested by the code you have already seen: create a "set" of prime numbers first in an object of the `IntSet` class, and then use this set intelligently to check the conjecture.

Task 12 - 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.

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, 2017

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());
  /* or simply
    results.WriteLine("union with " + smallSetStr + " = " + mySet.Union(smallSet));
    results.WriteLine("intersection with " + smallSetStr + " = " + mySet.Intersection(smallSet));
  */

    results.Close();
  } // Main

} // SampleIO
```