Computer Science 301 - 2010

Programming Language Translation

Practical for Week 19, beginning 23 August 2010

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#, Java, Pascal, Modula-2 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 Java and C# are available on the course web site at `http://www.cs.ru.ac.za/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 Java.

To hand in:

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

- Listings of your solutions to the programming exercises in tasks 7 and 11, produced by using the `LPRINT` utility or by using UltraEdit in "small Courier font" mode.
- 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 suuitable comments typed into **all** listings. You are expected to be familiar with the University Policy on Plagiarism, which you can consult at:

```
http://www.scifac.ru.ac.za/plagiarism_policy.pdf
```

Before you begin

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

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

- UltraEdit is probably your editor of choice. The version in the lab is configured to run various of the compilers easily, and it is possible to tweak it to run others in the same sort of way. *To get this to work properly, start UltraEdit from a command window by giving the command* `UEDIT32`*, rather than by clicking on an icon on the desktop or start menu.*

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

  ```
  LPRINT Fibo.pav Fibo.java
  ```

  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 or the standard UltraEdit configuration with 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 will 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 Java or 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 `prac19.zip` and `xtacy.zip` 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`.

```
copy i:\csc301\trans\prac19.zip
unzip  prac19.zip
```

Use `UNZIP` or `WINZIP`, as the file contains files with long file names which `PKUNZIP` cannot handle.

In the past there has 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:

```
d:
md d:\G01T1111
cd d:\G01T1111
unzip I:\csc301\trans\prac19.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 computing Fibonacci numbers recursively, some "empty" programs, and some other bits and pieces, including a few batch files to make some of the following tasks easier.

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 determines 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 executable (use the command `DIR SIEVE.EXE` and `DIR FIBO.EXE` and `DIR EMPTY.EXE`).

You may be able to produce a slightly faster version of the executable program for the Sieve example by suppressing the index range checks that Pascal compilers normally include for code that accesses arrays:

```
FPO SIEVE.PAS
```

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

Here is something more demanding: By experimenting with the `CONST` declaration, find out how large a sieve the program can handle. What is the significance of this limit? *Hint*: you should find that funny things happen when the sieve gets too large, though it may not immediately be apparent.

Task 3 The Sieve in Modula-2

You may not be a Modula-2 expert either, but examine, and then compile and run the equivalent Modula-2 code supplied in the files `SIEVE.MOD, EMPTY.MOD` and `FIBO.MOD`. You can do this quickly using commands like

`M2C FIBO`	(note that the `.MOD` extension is not quoted here) or
`M2O SIEVE`	(for the version that suppresses subscript checks)

Make a note of the size of the executables produced. How do they compare with the Pascal executables? Approximately how big a sieve can the compiler handle? Why do you suppose there is a difference, when the source programs are all so similar?

Task 4 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:

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

Task 5 Jolly Java, what

There are two Java compilers available for your use. The JDK one is called `javac` and there is also the (much faster) one called `jikes` (Jikes will only handle Java 1.4 level source, but that covers most things). Both of these are conveniently invoked from within UltraEdit. You can also compile a Java program directly from the command line with commands like

```
javac Sieve.java       (using the (slow) JDK compiler)
jikes Sieve.java       (using the (fast) Jikes compiler)
```

Task 6 See C#

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

```
csharp Sieve.cs
```

(You may have to do this on the local D: drive) 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?

Task 7 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` and `FIBO.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 Sieve.pav         slightly optimized code
parva -l Sieve.pav         error messages merged into listing.txt
```

You will have to do this on the local D: drive.

More conveniently, we have set up UltraEdit to allow for an option to compile Parva programs. If you want to add this to your home systems, use the `Advanced->Tool Configuration` pull down, then set the following fields

Command Line	Parva %n%e
Working Directory	%p
Menu Item Name	Parva
Save all files first	Selected
Output to List Box	Selected
Capture Output	Selected

and then click `Insert`. After this you can choose the Parva option on the Advanced menu to compile (and, when successful, run) the program in the "current window". The demonstration programs `Sieve.pav` and `Fibo.pav` in the kit have a few fairly obvious errors. Learn the syntax and semantics of Parva by correcting the errors until the programs run correctly. Once again, experiment to see how large a sieve you can set up.

Task 8 High level translators

It may help amplify the material we are discussing in lectures if you put some simple Modula-2 programs through a high-level translator we have available, and then look at, and compile, the C 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 demonstration copy of a system (Russian in origin), that translates Modula-2 or Oberon-2 source code into C. The system is called Extacy (a poor pun on "X to C", it seems). Whether or not the C one obtains is usable depends, obviously, on having C translations of all of one's Modula-2 libraries as well. In principle all one has to do is convert these libraries using the same system. Some very simple libraries came with the demonstration kit, and we have produced one or two more, but we would have to pay many Roubles and do an awful lot of work to get the system fully operational.

- Create a further subdirectory under your G0xAxxxx directory, say `XTACY`. The reason for working in another directory is to ensure that you don't edit, change, or otherwise get corrupted versions of other files with similar names in other sections of the prac kit.

- Log into this directory.

- Unpack the demo conversion program with the command `UNZIP XTACY.ZIP`. This will create a whole lot of other files for you.

- A command of the form

```
XC   =m   SOURCE.MOD
```

will produce all the `.H` and `.C` files needed for a "make" of the parent program `SOURCE.MOD`

Convert the sample programs in this kit (`SIEVE.MOD` and `FIBO.MOD`) and the various support modules to C, and then use a C++ compiler to compile and run the resulting code. Most simply, run the C compiler directly from the command line:

```
BCC SIEVE.C EASYIO.C X2C.C
BCC FIBO.C EASYIO.C X2C.C
```

or

```
CL SIEVE.C EASYIO.C X2C.C
CL FIBO.C EASYIO.C X2C.C
```

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 size of the object code produced.

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.

Hint: the machines in the Hamilton Labs are *very* fast, so you should try something like this: modify the sieve programs to comment out nearly all the output statements (since you are not interested in seeing a zillion lists of prime numbers, or measuring the speed of I/O operations), and then run the programs and time them with a stop watch. Similarly, choose an upper limit for the list of Fibonacci numbers (and a suitable number of iterations and sizes for the sieve) that will produce measurable times.

Although Java is often touted as being an interpreted language, in fact the latest versions of the Java "interpreter" - the program executed when you give the `java` command - actually indulge in "just in time" compiling (see textbook page 32) and "JIT" the code to native machine code as and when it is convenient - which results in spectacularly improved performance. It is possible to frustrate this by issuing the `java` command with a directive `-Xint`:

```
javac Sieve.java
java -Xint Sieve
```

to run the program in interpretive mode. Try this out as part of your experiment.

Summarise your findings on page 2 of the cover sheet, and go on to explain briefly how you come to the figures that you quote. For example, 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?

Task 10 Reverse Engineering and Decompiling

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 of these available for experiment.

`jad`	a decompiler that tries to construct Java source from Java class files
`javap`	a decompiler that creates pseudo assembler source from a Java class file
`gnoloo`	a decompiler that creates JVM assembler source from a class file
`oolong`	an assembler that creates Java class files from JVM assembler source
`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) After compiling `Sieve.java` to create `Sieve.class`, decompile this:

```
jad   Sieve.class
```

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

(b) Disassemble `Sieve.class`

```
javap  -c Sieve >Sieve.jvm
```

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

(c) Disassemble `Sieve.class`

```
gnoloo Sieve.class
```

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

(d) Reassemble `Sieve.j`

```
oolong Sieve.j
```

and try to execute the resulting class file

```
java Sieve
```

(e) Be malicious! Corrupt `Sieve.j` - simply delete a few line with opcodes on them. Try to reassemble the file (as above) and to re-run it. What happens?

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

```
csharp Sieve.cs
decompile Sieve
```
(calls ildasm from a batch file)

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

(g) Reassemble `Sieve.cil`

```
ilasm Sieve.cil
```

and try to execute the resulting class file

```
Sieve
```

(h) Be malicious! Corrupt `Sieve.cil` - simply delete a few line with opcodes on them. Try to reassemble the file (as above) and to re-run it. What happens?

(i) Experiment with the .NET verifier after (h)

```
exeverify Sieve.exe
```
(calls peverify from a batch file)

Task 11 Something more creative - Play Sudoku

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.

You will need to become acquainted with various library classes to solve this task, which is preferably to be done in Java (C# if you prefer), and is designed to emphasize some useful techniques. Descriptions of these relevant library routines can be found on the course website.

A simple sample program using some of the library routines can be found in the kit as the program `SampleIO.java` (listed below).

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".

The game of Sudoku has achieved cult status, and hours are spent each day by thousands of commuters solving the puzzles that appear in their newspapers.

In its simplest form, the player is presented with a 9 x 9 grid, in some cells of which appear single digits. The aim of the game is to deduce how to fill all the remaining cells, subject to the constraint that each row, column and 3 by 3 sub matrix contains each of the numbers 1 to 9.

When playing the game one cannot get very far without carefully maintaining a **set** of *assignable values* for candidates for each blank cell. One starts from the assumption that each blank cell might have any value between 1 and 9, constructs the corresponding small sets, and then then removes from each set all values which have already been assigned to other cells in its respective row, column and 3 x 3 box. Doing this by hand is laborious and prone to error, and often detracts from the fun of solving these puzzles. So, for something of a challenge, develop a program to help in this regard.

The program can begin by reading in a 9 x 9 matrix of values similar to that shown here (use 0 to denote a blank cell):

```
0  0  1  0  0  0  8  0  0
0  7  0  3  1  0  0  9  0
3  0  0  0  4  5  0  0  7
0  9  0  7  0  0  5  0  0
0  4  2  0  5  0  1  3  0
0  0  3  0  0  9  0  4  0
2  0  0  5  7  0  0  0  4
0  3  0  0  9  1  0  6  0
0  0  4  0  0  0  3  0  0
```

From this one can compute the initial assignable sets for each element of a 9 x 9 matrix of small sets. For the above example this would lead to the following (spend a moment thinking about this and verifying the values of some of the sets for yourself):

```
           0   1   2    3   4   5    6   7   8
        |============+============+============|
        |    . 2 .   | 2 . 2 . 2  |    . 2 . 23|
     0  | 456. 56.   |   6.  6.  6|    . 5 . 56|
        |   9.   .   |   9.   .7  |    .   .   |
        |----+---+----+----+---+----+----+---+----|
        |    .   .   |    .   . 2 |  2 .   . 2 |
     1  | 456.   . 56|    .   .  6| 4 6.   . 56|
        |  8 .   . 8 |    .   . 8 |    .   .   |
        |----+---+----+----+---+----+----+---+----|
        |    . 2 .   |  2 .   .   |  2 .12 .   |
     2  |    .  6.  6|   6.   .   |   6.   .   |
        |    . 8 . 89| 89.   .    |    .   .   |
        |============+============+============|
        | 1  .   .   |    . 23. 23|    . 2 . 2 |
     3  |   6.   .  6|    .  6.4 6|    .   .  6|
        |  8 .   . 8 |    . 8 . 8 |    . 8 . 8 |
        |----+---+----+----+---+----+----+---+----|
        |    .   .   |    .   .   |    .   .   |
     4  |   6.   .   |   6.   .  6|    .   .  6|
        | 78 .   .   |  8 .   . 8 |    .   . 89|
        |----+---+----+----+---+----+----+---+----|
        | 1  .1  .   | 12 . 2 .   |  2 .   . 2 |
     5  |  56. 56.   |   6.  6.   |   6.   .  6|
        | 78 . 8 .   |  8 . 8 .   | 7  .   . 8 |
        |============+============+============|
        |    .1  .   |    .   .  3|    .1  .   |
     6  |    .  6.  6|    .   .  6|    .   .   |
        |    . 8 . 89|    .   . 8 |   9. 8 .   |
        |----+---+----+----+---+----+----+---+----|
        |    .   .   |  2 .   .   |  2 .   . 2 |
     7  |  5 .   . 5 | 4  .   .   |    .   . 5 |
        | 78 .   .78 |  8 .   .   | 7  .   . 8 |
        |----+---+----+----+---+----+----+---+----|
        | 1  .1  .   |  2 . 2 . 2 |    .12 .12 |
     8  |  56. 56.   |   6.  6.  6|    . 5 . 5 |
        | 789. 8 .   |  8 . 8 . 8 |    .78 . 89|
        |============+============+============|

    31 squares known
```

After displaying this structure, the program can then invite the user repeatedly to input triplets of numbers:

```
Your move - row[0..8] col [0..8] value [1..9] (0 to give up)?
```

If the combination is valid (that is, if the requested value is indeed in the assignable set for the designated cell of the grid), the program should assign the value to that cell, exclude it from the assignable sets of all other cells sharing the same row, column and 3 x 3 box, and then display the resulting new state of the game.

Begin by writing a program that will do this. As already mentioned, make use of the `InFile` and `IO` libraries to handle input and output - keeping it simple! Make use of the `IntSet` class to manipulate the sets you need. Please resist the temptation to do I/O in any other way, or to use other classes for manipulating sets. Details of these classes can be found on the course web page.

A sample executable is in the prac kit, and you can run this with the command

```
sudoku0 datafile
```

where datafile is one of the sample files `s1, s2, s3, s4` ... (It is easy to find other puzzles - but to make it easy to check without spending hours playing the game, these data files have the solution appended as well.)

Once you have got that working, go on to something more challenging - consider how you can get the program to make suggestions as to what values can be supplied, and where. Here you can be guided by what some of the literature call "singles" and "hidden singles".

Singles: Any cell which has only one element left in its assignable set can safely be assigned that value.

Hidden Singles: Very frequently, there is really only one candidate for a given row, column or 3 x 3 box, but it is hidden among other candidates. For example, in the extract below, the number 6 is only found in the middle right cell of a 3 x 3 box. Since every 3 x 3 box must have a 6, this cell must be that 6.

(4)	(7)	1 5 9
(3)	(8)	1 56 9
(2)	1 9	1 5 9

In the bigger example given earlier the number 1 is a "hidden single" in the top right 3 x 3 box, and also in the middle 3 x 3 box (verify this for yourself).

Modify the program to determine and display the singles and hidden singles in parentheses, perhaps giving output on the lines suggested here:

```
      0   1   2   3   4   5   6   7   8

 0:  (4) ..   1  ..  ..  (7)  8  ..  (3)
 1:  ..   7  ..   3   1  ..  (4)  9  ..
 2:   3  ..  ..  ..   4   5  ..  (1)  7
 3:  (1)  9  ..   7  (3) (4)  5  ..  ..
 4:  (7)  4   2  ..   5  ..   1   3  (9)
 5:  ..  ..   3  (1) ..   9  (7)  4  ..
 6:   2  ..  ..   5   7  (3) (9) ..   4
 7:  ..   3  (7) (4)  9   1  ..   6  ..
 8:  ..  ..   4  ..  ..  ..   3  (7) (1)

      0   1   2   3   4   5   6   7   8

31 squares known. 18 predictions
```

From here it is an easy extension to write a program that will either solve the game completely or get you pretty close to a solution. In the above example, after identifying the 11 predictions, the program could apply them all, rather than asking for input from the user, and then display the outcome. In this example this would lead to

```
      0   1   2   3   4   5   6   7   8

 0:   4  ..   1  (9) ..   7   8  (5)  3
 1:  ..   7  (5)  3   1  ..   4   9  ..
 2:   3  ..  (9) ..   4   5  (6)  1   7
 3:   1   9  ..   7   3   4   5  ..  ..
 4:   7   4   2  ..   5  ..   1   3   9
 5:  ..  ..   3   1  (2)  9   7   4  ..
 6:   2  (1) ..   5   7   3   9  (8)  4
 7:  ..   3   7   4   9   1  (2)  6  (5)
 8:  (9) ..   4  ..  ..  ..   3   7   1

      0   1   2   3   4   5   6   7   8

49 squares known. 11 predictions
```

and after a few more iterations the final solution will emerge

```
      0   1   2   3   4   5   6   7   8

 0:   4   2   1   9   6   7   8   5   3
 1:   6   7   5   3   1   8   4   9   2
 2:   3   8   9   2   4   5   6   1   7
 3:   1   9   8   7   3   4   5   2   6
 4:   7   4   2   8   5   6   1   3   9
 5:   5   6   3   1   2   9   7   4   8
 6:   2   1   6   5   7   3   9   8   4
 7:   8   3   7   4   9   1   2   6   5
 8:   9   5   4   6   8   2   3   7   1

      0   1   2   3   4   5   6   7   8

81 squares known. 0 predictions

No more moves possible
```

A sample executable with these refinements has been supplied in the prac kit and can be executed with the command

```
sudoku1 datafile
```

For some puzzles you may find that the tips suggested here cannot make any predictions; in such cases the user can be invited to make his or her own move, as in the simpler version of the program.

Demonstration program showing use of InFile, OutFile and IntSet classes

This code is in the file `SampleIO.java` in the prac kit. There is an equivalent C# one in the file `SampleIO.cs`.

```
import library.*;

class SampleIO {

  public static void main(String[] args) {
  //                                        check that arguments have been supplied
    if (args.length != 2) {
      IO.writeLine("missing args");
      System.exit(1);
    }
  //                                        attempt to open data file
    InFile data = new InFile(args[0]);
    if (data.openError()) {
      IO.writeLine("cannot open " + args[0]);
      System.exit(1);
    }
  //                                        attempt to open results file
    OutFile results = new OutFile(args[1]);
    if (results.openError()) {
      IO.writeLine("cannot open " + args[1]);
      System.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());
  } // main
} // SampleIO
```