Computer Science 301 - 2012

Programming Language Translation

Practical for Week 19, beginning 27 August 2012

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/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 Java.

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 6 and 11 - 14, 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 by following a link from

http://www.scifac.ru.ac.za/ or http://www.ru.ac.za/

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).

- If UltraEdit is your editor of choice, note that the version in the lab can be configured to run various of the compilers easily. *To get this to work properly, you may need to 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.cs  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, 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 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` from a command line prompt.

```
j:> copy i:\csc301\trans\prac19.zip
j:> unzip  prac19.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:\G01T1111
d:> cd d:\G01T1111
d:> 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. 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 compliers:

```
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 4 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 5 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 6 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 you might like to set up UltraEdit to allow for an option to compile Parva programs. If you want to do this, 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 7 A blast from the past - two 1980s vintage 16 bit DOS compilers

We have three early compilers which we want you 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, use the `Start -> Programs-> All Programs -> DosBox` sequence, which will open an 80 x 25 text window 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, 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 the commands

```
TPC SIEVE.PAS
TPC FIBO.PAS
TPC EMPTY.PAS
```

and comment on any major differences that you notice from your use of Free Pascal. TPC executes a version of Turbo Pascal last developed in 1990, by which stage the Pascal language it compiled was quite a lot more complex than the original language of 1970.

Turbo Pascal 1.0

For some real fun, try out the early 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.

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.

TopSpeed 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 under DOSBox 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?

This compiler - known as TopSpeed Modula-2 - was another brilliant implementation, done partly by the same team as developed Turbo Pascal. Once again, it was designed to run on small systems and exploit every ounce of speed possible. Rhodes used this compiler for teaching between 1986 and about 1994, and I occasionally still do. Nostalgia ain't what it used to be.

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.

Once again, this is a 16-bit system, and you will have to execute it under DOSBox (and then compile the generated C code using a 32-bit compiler under Windows).

- Unpacking the ZIP file onto the D: drive will have created a further subdirectory XTACY under the first directory, with the parts of the Xtacy system ready for use.

- Log into this directory under DOSBox and then give the command

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

This should 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 under Windows 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 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?

(f) 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`

(g) Reassemble `Sieve.cil`

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

and try to execute the resulting class file

```
Sieve
```

(h) 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?

(i) Experiment with the .NET verifier after step (g) and again after step (h)

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

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 tasks.

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 problems can be solved elegantly in a relatively 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 11 Creative Parva programming - the "hailstone" sequence

Nothing so far should have extended your programming talents very much. To get the brain cells working a little harder, and to learn more about a language we shall meet again, solve the following problem using Parva:

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 12 Something more creative - Lucky Numbers

Consider a sequence originally containing the N ordinal numbers

1 2 3 4 5 6 7 8 9 10 11 12 13 14 … N

Removing every second number produces the new sequence

1 3 5 7 9 11 13 15 17 19 21 23 25 27 …

and now removing every third number from that sequence generates another new sequence

1 3 7 9 13 15 19 21 25 27 …

and removing every fourth number from this sequence produces another new sequence

1 3 7 13 15 19 25 27 …

This process continues indefinitely, or more exactly, as long as numbers are still there to be removed. For example if $N = 12$ the process would terminate once we had generated the sequence 1 3 7.

The numbers which survive the culling process indefinitely are deemed to be "lucky".

Write a Parva program to determine the lucky numbers less than some number provided as input data.

Hint: Write a function that will generate a list of such numbers in an array parameter when supplied with an upper limit N, and use this in a program that displays such a list.

Please read the next section carefully before you continue:

You will need to become acquainted with various library classes to solve these next problems - classes that we shall use a great deal from here on as the rest of the software depends crucially on them. A simple sample program using some of the library routines can be found in the kit as the program `SampleIO.java`, which is listed below. Descriptions of these and other library routines can be found on the course website.

So please don't just rush into this, and please don't just "do your own thing" and use `System.out.write` and `Console.Write` and all those old friends. In particular the I/O must be done using the `IO, InFile` and `OutFile` classes and not all sorts of other classes and I/O and parsing methods that you might know of or read up about. Study the `SampleIO` or `SampleIO.cs` program carefully and familiarize yourself with routines like `readChar, readString` and `readWord`.

And now, to refresh your Java or C# skills, here are two more exercises:

Task 13 Something more creative - "Pig Latin"

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!

Task 14 Timetable clashes

A recurrent problem that students have is to check timetables for clashes. Wouldn't it be nice if we had programs that could do this for us?

Well, of course we have, if we are prepared to write them. In the prac kit you will find a data file that describes the Rhodes Timetable, and a program `CLASH.EXE` (originally written in C#; you may have to run it from the D: drive) that reads this file, and then allows you to type in pairs of subjects, for each of which it will report whether there are any clashes, and when these occur. A session with the program might go something like this

```
D:>CLASH

Subject 1 (or STOP) ? csc 1
Subject 2 (or STOP) ? gog 2
CSC 1     Mon 1 Tue 2 Wed 3 Thu 4 Fri 5
GOG 2     Mon 3 Tue 4 Wed 5 Wed 7 Wed 8 Wed 9 Wed 10 Thu 1 Fri 2
0 clashes

Subject 1 (or STOP) ? csc 1
Subject 2 (or STOP) ? gog 1
CSC 1     Mon 1 Tue 2 Wed 3 Thu 4 Fri 5
GOG 1     Tue 2 Wed 3 Thu 4 Fri 5
4 clashes Tue 2 Wed 3 Thu 4 Fri 5

Subject 1 (or STOP) ? stop
D:>
```

Your biggest task this week is to write a Java or C# program that achieves the same effect.

Warning - although this exercise is actually quite easy, I suspect that this may be a non-trivial problem for many of you. So make sure you understand what is required before you rush in to writing code. Discuss the approach you will take with the demonstrator team, and with each other.

The program involves various aspects of reading and *parsing* data - that is, recognizing components of data buried in a large input file and extracting them for further processing. Parsing is a fundamental component of programming language translation as well.

The problem is neatly solved if you make use of sets - each subject has a set of fixed time table slots (and a possible set of alternative time-table slots, but we can ignore those; you can also ignore any periods out of the

range 1 … 10 in any day). So the program can sensibly make use of an array of suitable records or structures, one field of which is a set of integers.

Part of the data file is listed out below. You will need to study it before you begin. Notice that there is a lot of information on each line that is effectively ignorable for this exercise.

```
22 February 2012                                                ; date of last modification
;
; Data file used by the CLASH program 2008 (all semesters too)
; and by WHEN, VENUE, VENALL and WHERE
;
; This file should only be edited with a "flat ASCII" editor, not a word
; processor.  Use QEdit if all else fails (a very easy editor!)
;
; Format is as follows - spacing important before the "periods" start
;
; Line starting ; in column 1 is just commentary.  Blank lines not allowed!!
;
; Sem = blank (both) or 1/F (1st) or 2/S (second)
; 1 and 2 for courses only offered in one semester
; F and S for components of a year course
; A and B for components of a year course for venues only
;
; * or + are subjects ignored by DUMPVEN and DUMPTIM
; col 2 non blank if you want to highlight changes
;
; Mnemonic of 7 chars is the one that is typed into Pat's programs
; Mnemonic of 7 chars is the one that is useful for venue loading/display
; Protea course code
; Mnemonic of 7 chars is the one that is used by Protea programs
; Followed by guesstimate of size and 5-letter venue code
;
; counts 3 or >4 are used in the venue programs   (mixed venue subjects need a count of 2)
; counts 3 and 4 mean the subject is ignored when testing for clashes in Pat's helper options
; counts 2 or >4 are used in the timetable print program (only for blank, 1 or 2 type subjects)
;
; List of timetabled periods followed, terminated with .
; This is a little cryptic, but not too bad once you get used to it
; It's all linked to the TIMES file if there is no + or -
;  L = alternative lecture
;  T = alternative tut
;  A = alternative prac
;  P = fixed prac
;  no letter - fixed period
;
; 11 = day 1 period 1; 35 = day 3 period 5;  59 = day 5 period 9; 1A = day period 10
;
; Mnemonic Booked   Code     Protea  Size Venue Full Name                     Cryptic stuff with . at end
;2345678901234567890123456789012345678901234567890123456789012345678901234567890
; vvvvvvv  vvvvvvv  vvvvvvv  vvvvvvv  vvv vvvvv vvvvvvvvvvvvvvvvvvvvvvvvvvvvv  vvvvvvvvvvvvvvvvv
  ACC 1    ACC 1    4101100  ACC 1     20 Barr2 Accounting 1                  L31 L35 L42 L46 L52 L53 A17 A18 A27 A28 A4
1 ACC 101  ACC 101  4101101  ACC 101   20 Barr2 Accounting 101 (Sem 1)        L31 L35 L42 L46 L52 L53 A17 A18 A27 A28 A4
2 ACC 102  ACC 102  4101102  ACC 102   20 Barr2 Accounting 102 (Sem 2)        L31 L35 L42 L46 L52 L53 A17 A18 A27 A28 A4
* CSC 1    CSC 1    5101100  CSC 1      1 CheMa Computer Science 1            11 22 33 44 55 A17 A18 A19 A1A A27 A28 A29
* CSC 2    CSC 2    5101200  CSC 2      5 Glg11 Computer Science 2            13 24 35 41 52 P37 P38 P39 P3A .
* CSC 3    CSC 3    5101300  CSC 3      5 Glg11 Computer Science 3            12 23 34 45 51 P47 P48 P49 P4A .
 .DRA 1    DRA 1    2301100  DRA 1     20 Gog10 Drama 1                       22 33 44 A29 A2A A39 A3A A49 A4A .
;
1 EAR 101  EAR 101  26011G1  EAR 101   20 GLT   Earth Science 101 (Sem 1)     11 22 33 44 55 A17 A18 A19 A27 A28 A29 A2A
;
;  These next are not really "subjects" but are useful in marking special
;  periods when fiddling
+ MON  1   MON  1                      1                                     11 .
+ TUE  1   TUE  1                      1                                     21 .
+ WED  1   WED  1                      1                                     31 .
+ MON  10  MON  10                     1                                     1A .
+ SLOT 1   SLOT 1                      1                                     11 22 33 44 55 .
+ SLOT 2   SLOT 2                      1                                     12 23 34 45 51 .
```

You may also like to experiment with the program `WHEN.EXE` in the prac kit, which creates a graphical output of a timetable for selected subjects and explores whether apparent clashes can be resolved. It does this by using a recursive "backtracking" approach to that used to solve the N Queens problem that you may have seen elsewhere.

The file `KEY` summarizes the mnemonic codes used for choosing subjects in the `WHEN` and `CLASH` programs.

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
```