Computer Science 301 - 2016

Programming Language Translation

Practical 1, Week beginning 18 July 2016

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

Objectives:

In this practical you are to

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

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

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

Outcomes:

When you have completed this practical you should understand

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

To hand in:

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

- One copy of the listings of your solutions to the programming exercises in tasks 5 and 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 back of the cover sheet by the end of the first afternoon.

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

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

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

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

Before you begin

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

- After logging on, get to the DOS command line level by executing `CMD.EXE` to open a command window if you don't already have a shortcut (it is probably worth creating a short cut).

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

  ```
  LPRINT  SieveCS.cs  SiecePAS.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" 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 `PRINTHAMILTON`.

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

Task 2 The Sieve of Eratosthenes in Pascal

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

```
FPC SIEVEPAS.PAS
FPC FIBOPAS.PAS
FPC EMPTYPAS.PAS
```

to use the 32-bit Windows version of the Free Pascal compiler. Make a note of the size of the executables (use the commands `DIR SIEVEPAS.EXE`, `DIR FIBOPAS.EXE` and `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?

```
PROGRAM Sieve (Input, Output);
(* Sieve of Eratosthenes for finding primes 2 <= N <= Max
   P.D. Terry, Rhodes University, 2016 *)

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

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

Here is something more demanding:

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

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

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

Task 3 The Sieve in C or C++

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

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

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

Task 4 See C#

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

```
csharp SieveCS.cs
```

Make a note of the size of the ".NET assemblies" produced (`SIEVECS.EXE`, `SIEVESET.CS`, `SIEVESET2.CS`, `EMPTYCS.EXE` and `FIBOCS.EXE`). How do these compare with the other executables? What limit is there now to the largest prime you can find?

Task 5 Progress to Parva

On the course web page you will find a description of Parva, a toy language very similar to C, and a language for variations on which we shall develop a compiler and interpreter later in the course. The main difference between Parva and C/Java/C# is that Parva is stripped down to bare essentials. Learn the Parva system by studying the language description where necessary, and trying the system out on the supplied code (`SIEVEPAV.PAV`, `FIBOPAV.PAV` and `EMPTYPAV.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! Hand in listings of your final corrected `SievePAV.pav` - produced with the `LPRINT` command as always.

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

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

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

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

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

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

Turbo Pascal 6.0

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

```
TP6 SIEVEPAS.PAS
TP6 FIBOPAS.PAS
TP6 EMPTYPAS.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
TP6O FIBOPAS.PAS
TP6O EMPTYPAS.PAS
```

Turbo Pascal 1.0

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

```
TURBO
```

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

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

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

Task 7 High level translators

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

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

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

```
Parva2ToCSharp 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 commands of the form:

```
csharp SievePAVp2c.cs
```

and executed with the usual commands of the form

```
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. You might also like to comment on the performance of the programs that are produced, by running timing tests as in Task 9.

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

Run the Parva compiler directly:

```
Parva voter.pav
```

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

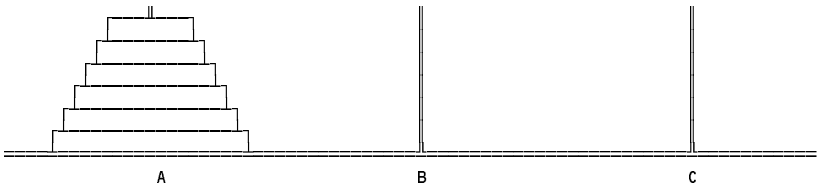
```
Parva2ToCSharp voter.pav
csharp voterp2c.cs
voterp2c
```

Task 8 The Towers of Hanoi problem

In the kit you will also find some variations on Parva programs for solving the Towers of Hanoi problem, which I am sure you must have seen previously as an example of the deft use of recursion.

One of the problems with the presentation of a recursive algorithm is that often one is not shown how the ideas come about, and yet when one is presented with the *fait accompli* it is hard to see how to implement it in any other way (as may have been the reader's experience with tree manipulation algorithms presented earlier), or where the catch lies. A classic example of this is the (rather ridiculous) Towers of Hanoi problem, which is a time wasting game that is supposed to have originated in Hanoi many years ago, and which occupies folk as follows:

We are given three pegs, imaginatively labelled *A, B, C*. On peg *A* we initially have *N* disks, placed in decreasing order of size from the bottom. The object of the exercise is to transfer disks from peg to peg, one at a time, so that eventually the disks are all on peg *B*, in the original order. We can use pegs *A, B, C* as we please, but at no stage can a disk of larger diameter lie on top of one of smaller diameter.



A recursive solution to this appears below, and is often presented with no clue given as to how it was derived. This is a pity, for it is not too much to ask that the clue be provided - the insight that leads to a correct solution is to think about the bottom disk on peg *A* rather than the top one. We then solve the problem as follows:

Move *N - 1* disks from peg *A* to peg *C*
Move the *Nth* disk from peg *A* to peg *B*
Move *N - 1* disks from peg *C* to peg *B*.

Yes, but how do we move $N - 1$ disks from A to C. Well, in the same way!

> Move $N - 2$ disks from peg A to peg B
> Move the $N - 1st$ disk from peg A to peg C
> Move $N - 2$ disks from peg B to peg C.

Yes, but how do we move the $N - 2$ disks? Well in the same way again! But surely we get to a stage where we don't have anything to move, if we are just moving smaller piles of disks at each turn? Yes, exactly - we have to build in a termination criterion, of course. In this way we might come up with

```
// Parva recursive solution to the Tower of Hanoi problem
// P.D. Terry, Rhodes University, 2016

  int moves;   // for expediency - count the moves
  bool display;

  void Hanoi (int n, int a, int b, int c) {
  // Move n disks from a to b using c as an intermediate peg
    if (n > 0) {
      Hanoi(n-1, a, c, b);
      if (display) write("Move disk ", n, " from ", a, " to ", b, "\n");
      moves = moves + 1;
      Hanoi(n-1, c, b, a);
     }
  } // Hanoi;

  void Main() {
    int disks;
    read("How many disks? ", disks);
    int iter;
    read("How many iterations ", iter);
    display = iter == 1;
    while (iter > 0) {
      moves = 0;
      Hanoi(disks, 1, 2, 3);
      write(moves, " moves\n");
      iter = iter - 1;
    }
  } // Main.
```

Notice that we have to specify the pegs as parameters to the function, as the role of source, destination and intermediate pegs changes at each call of the function.

This recursive solution has a time complexity which is easy to determine. The time complexity of our algorithm is $O(2^N)$ which is most unfortunate. It is easy to see that we cannot do better than this, and that, in a sense, the algorithm is optimal. Before we can move the Nth disk we *must* have moved off $N - 1$ disks, and after we move the Nth disk we *must* move all the other $N - 1$ disks again. Since the same rules apply for all movements, we cannot hope to find an algorithm for which $W_N < 2W_{N-1} + 1$.

If we, as humans, wanted to solve this problem, would we *really* have to proceed as given above? Keeping track of $2^N - 1$ moves looks like it will be a mind-blowing problem. As it happens, there is a much easier way of solving the problem, better suited to the lowly minds of us mortals, rather than the ethereal thoughts of the monks in Hanoi who are supposed to have been presented with 64 disks, told to get on with it, and then warned that when they finished the task the world would come to an end. (We are still here. 2^{64} is an awfully large number of moves!)

The trick, since you are doubtless wishing to have it revealed, is to think of the pegs as being placed in a circle, rather than in a straight line. One then proceeds as follows:

Simple solution to Towers of Hanoi

```
BEGIN
  Move smallest disk one peg x-wise
  WHILE another disk must be moved DO
    Move that disk
    Move smallest disk one peg x-wise
  END
END
```

where *x-wise* is clockwise if the number of disks is even, and anti-clockwise if the number of disks is odd.

It is not at all obvious that this leads to the same moves as before; it is probably not even obvious how it might best be programmed in Parva. However, being a kind old soul I have included an iterative version, leaving you to puzzle out how it works.

If you run the Parva versions of these programs you should find that for large N they do indeed take a fair amount of time. Try converting them to C# using the tool, and then see if you can be blown away with the speed of the computers in Hamilton. Try to sense how the two versions compare.

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

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

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

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

Summarize your findings on page 2 of the cover sheet, explaining briefly how you come to the figures that you quote. 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 Towers of Hanoi or Fibonacci sequence (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 10 - Reverse Engineering - disassembly

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

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

Try out the following experiments or others like them:

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

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

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

(b) Reassemble `SieveCS.cil`

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

and try to execute the resulting class file

```
SieveCS
```

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

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

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

Task 11 - A Cautionary Tale

A few years ago I set some simple programming exercises for this class to do in this practical, and provided an executable version of a solution to assist the class in understanding the problem. I had not reckoned with the guile of some of your predecessors who, rather than write their own program, decompiled my executable and handed that in instead. Caution: After years of experience I can spot fraudulent behaviour very quickly. I'd thought I was safe because I had not let the class know about .NET decompilers, but my colleague Professor Google had obviously been consulted. The group pointed me very quickly to a tool written by JetBrains, known as dotPeek. This is very easy to use and quite fun, so I have installed it on the lab machines temporarily for use in this practical.

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

```
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.

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

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

Task 12 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.

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 value 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. Details of these classes can be found on the course web page. Please resist the temptation to do I/O in any other way, or to use other classes for manipulating sets.

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, most of 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 (like `S9`) you may find that, after a few moves, 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. These sorts of puzzles are usually labelled "hard" in the puzzle books.

Solving Sudoku puzzles is one thing. I have never attempted to create one, and, while I am not suggesting you do so, you might like to ponder on how it could be done (or ask Dr Google).

I am looking for an imaginative, clear, simple solution to the problem.

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, 2016

using Library;
using System;

class SampleIO {

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

} // SampleIO
```