The University of Melbourne
Department of Computer Science and Software Engineering

Declarative Programming
COMP90048

Semester 2, 2011

Project Specification

Project due 24 October 2011 at 5pm
Worth 20%

The objective of this project is to practice and assess your understanding of logic programming and Mercury. You will write code to solve Sudoku puzzles.

The (Generalised) Game of Sudoku

A generalised sudoku puzzle of rank n consists of n^4 cells arranged in a $n^2 \times n^2$ grid. The cells of this grid are also grouped into $n \times n$ cell squares, with these larger squares arranged into an $n \times n$ grid. The standard Sudoku puzzle is rank 3.

When a Sudoku puzzle is initially supplied, some of the cells of the puzzle are filled in with numbers between 1 and n^2, and the rest are left blank. To solve the puzzle, the blank cells must be filled with numbers between 1 and n^2. The rules are simple: each of the n^2 cells in each row, column, and $n \times n$ cell square must be filled with a distinct number. Therefore, each row, column, and $n \times n$ cell square must contain all of the numbers between 1 and n^2. A correctly constructed Sudoku puzzle has only one correct solution.

Here is an example rank 3 Sudoku puzzle, together with its solution, taken from the Wikipedia page for Sudoku:

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

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

The Program

You will write Mercury code to solve Sudoku puzzles. Your program should take one command line argument naming a file containing the puzzle, and should write the solved puzzle to the

standard output stream. Rank n Sudoku puzzles are formatted, for both input and output, as text files following these instructions:

- The first and last lines, and after every n lines of numbers, should consist of n runs of n hyphens ("`-`") separated by plus signs ("`+`"), and with a plus sign and the start and end of the line.
- Each line of numbers should show a single row of the puzzle, with a single character for each cell in the row. For cells filled with a number, the number character should be shown. Where a number greater than 9 is to be shown, "`a`" should be shown for 10, "`b`" for 11, and so on. An un-filled cell should be shown as a period ("`.`") character.
- Each line of numbers should begin and end with a vertical bar character ("`|`"), and should have a vertical bar after every n cells.

This should produce a reasonable ASCII depiction of a Sudoku puzzle. For example, the sample puzzle above, and its solution, would be written as follows:

```
+---+---+---+                   +---+---+---+
|53.|.7.|...|                   |534|678|912|
|6..|195|...|                   |672|195|348|
|.98|...|.6.|                   |198|342|567|
+---+---+---+                   +---+---+---+
|8..|.6.|..3|                   |859|761|423|
|4..|8.3|..1|                   |426|853|791|
|7..|.2.|..6|                   |713|924|856|
+---+---+---+                   +---+---+---+
|.6.|...|28.|                   |961|537|284|
|...|419|..5|                   |287|419|635|
|...|.8.|.79|                   |345|286|179|
+---+---+---+                   +---+---+---+
```

I will post sample Mercury code to read and write Sudoku puzzles in this format on the LMS. You may feel free to use, adapt, or ignore this code. If you do use or adapt this code, you should preface it with a **prominent comment clearly explaining where you got it**.

The Mercury release comes with a Sudoku program written using Mercury's constraint logic programming feature. Your program **must not use** Mercury's constraint logic programming feature. However, you may want to look at this program, if only to see how simple it is as a constraint logic program. I will post this program on the LMS.

I will compile and link your code for testing using the command:

```
mmc -O6 --make sudoku
```

or similar. This will compile your program with maximum optimisation.

Assessment

Your project will be assessed on the following criteria:

30% Quality of your code and documentation;

30% The ability of your program to correctly solve Easy rank 3 Sudoku puzzles;

20% The ability of your program to correctly solve Medium rank 3 Sudoku puzzles;

20% The ability of your program to correctly solve Difficult (Killer, Diabolical) Sudoku puzzles of sizes up to rank 5.

Note that timeouts will be imposed on all tests. You will have at least 1 second to solve each puzzle, regardless of difficulty. Executions taking longer than that may be unceremoniously terminated, leading to that test being assessed as failing. One second should be ample with careful implementation.

See the Project Coding Guidelines on the LMS for detailed suggestions for coding style. These guidelines will form the basis of the quality assessment of your code and documentation.

Submission

The project submission deadline is 24 October 2011 at 5pm.

You must submit your project from any one of the student unix servers, such as `cat`, `lister`, `rimmer`, or `holly`. Make sure the version of your program source files you wish to submit is on these machines (your files are shared between all of them, so any one will do), then `cd` to the directory holding your source code and issue the command:

```
submit 686 proj2 sudoku.m
```

If your code spans multiple source files, add the extra ones to the end of that command line.

Important: you must wait a minute or two (or more if the servers are busy) after submitting, and then issue the command

```
verify 686 proj2 | less
```

This will show you the test results from your submission, as well as the file(s) you submitted. If the test results show any problems, correct them and submit again. You may submit as often as you like; only your final submission will be assessed.

If your program compiles and runs properly, you should see the line "`Running tests`", followed by several test runs. If you do not see test runs like this, then your program did not work correctly.

If you wish to (re-)submit after the project deadline, you may do so by adding "`.late`" to the end of the project name (*i.e.,* `proj2.late`) in the `submit` and `verify` commands. But note that a penalty, described below, will apply to late submissions, so you should weigh the points you will lose for a late submission against the points you expect to gain by revising your program and submitting again.

It is your responsibility to verify your submission.

Windows users should see the LMS Resources list for instructions for downloading the (free) Putty and Winscp programs to allow you to use and copy files to the department servers from windows computers. Mac OS X and Linux users can use the `ssh`, `scp`, and `sftp` programs that come with your operating system.

Late Penalties

Late submissions will incur a penalty of 0.5% per hour late, including evening and weekend hours. This means that a perfect project that is much more than 4 days late will receive less than half the marks for the project. If you have a medical or similar compelling reason for being late, you should contact the lecturer as early as possible to ask for an extension (preferably before the due date).

Hints

1. There is a huge amount of information on solving Sudoku puzzles on the net. You may find useful suggestions there, as well as programs written in other languages.

2. A good first strategy to employ is to compute the set of allowable values in each cell; when only one value is allowable in a particular cell, you can confirm that as the value for that cell. To compute the allowable values for each cell, begin by assigning each un-filled cell the set of all integers between 1 and n^2. Then go through each row, column, and $n \times n$ cell square of the puzzle (these are called the "countries" of the puzzle) first collecting the set of all filled-in values of cells in that country, and then removing the set of filled-in values in the country from all the sets of possibilities for the country. For example, the center cell in the sample Sudoku puzzle above would initially appear to have all values from 1 to 9 as possibilities, but after removing all the filled values of that row, only 2, 5, 6, 7, and 9 are possible. After removing the filled values of that column, only 5 is still a possibility. At that point, we can fill that cell in with 5.

   This process should be repeated as long as it continues to fill in values in previously un-filled cells. This technique alone will be sufficient to solve easy puzzles.

3. It will be well worth your while to define your own type to represent Sudoku cells.

4. If you represent a whole Sudoku puzzle as a one-dimensional (flat) list of cells, you can break up the puzzle into rows, columns, and squares with a single operation that converts a list into a list of lists according to repeating patterns. For example, for a rank 3 Sudoku puzzle, the puzzle can be broken into rows by taking groups of 9 elements in a row; it can be broken into columns by taking groups of 1 element in a row over a cycle of 9 groups, and repeating this 9 times; and it can be broken into 3×3 squares by taking 3 elements at a time over a cycle of 3 groups, and repeating this 3 times. I will post code to break a list into groups (that is, a list of lists) in this way, as well as code to perform the inverse operation, on the LMS. You may use this code or ignore it, but if you use it, you should preface it with a **prominent comment clearly explaining where you got it**.

   Also note that the Mercury library has many operations you may find useful for this project, and you are welcome to use the libraries (as long as you don't use the constraint logic programming modules). It is well worth reading through the documentation of some of the library modules to see what operations they provide. In particular, you may find the `list` and `set` and possibly the `map` modules helpful.

5. A good second strategy to employ when the first strategy has been exhausted is to fill in cells that are the only cell in a country that could possibly have a particular value.

Since each value must appear (once) in each country, if only one cell could possibly hold that value, then it must hold that value. For example, in the central column of the sample Sudoku puzzle, only the topmost un-filled cell could possibly be a 4, so it must be a 4. If employing this strategy is able to fill in any squares, then the first strategy may be able to fill more squares, so it should be tried again.

6. To solve the most difficult puzzles, it will be necessary to "guess" values for unfilled squares. This is the least efficient way to solve a Sudoku puzzle, so it should only be used when all other strategies have failed. Also, to make it most likely to guess correctly with the fewest guesses, you should pick a cell with the fewest possible alternative values. After guessing a value for a cell and filling it in, you can employ the other strategies again. You may discover that you have guessed wrong, in which case you need to backtrack your guess and make a different guess. Since backtracking is a natural feature of logic programming languages, Mercury is a good language in which to implement Sudoku.

7. There are other strategies you can employ. A little research on the web may suggest a few. However, these three strategies are enough to create a very efficient solution.

Note Well:

This project is part of your final assessment, so cheating is not acceptable. Any form of material exchange between teams, whether written, electronic or any other medium, is considered cheating, and so is the soliciting of help from electronic newsgroups. Providing undue assistance is considered as serious as receiving it, and in the case of similarities that indicate exchange of more than basic ideas, formal disciplinary action will be taken for all involved parties. If you have questions regarding these rules, please ask the lecturer.