Western Washington University
Computer Science Department

CSCI 145 Computer Programming and Linear Data Structures
Winter 2012

Assignment 1

Submitting Your Work

This assignment is worth 15% of the grade for the course. In this assignment you will create an Ada program to solve Sudoku puzzles. Save your program file (the .adb file) in the zipped tar file WnnnnnnnnAssg1.tar.gz (where Wnnnnnnnn is your WWU W-number) and submit the file via the **Assignment 1 Submission** item on the course web site. You must submit your assignment by 3:00pm on Friday, February 10.

Sudoku

A sudoku puzzle consists of a 9 x 9 grid of squares, subdivided into nine 3 x 3 boxes. Some of the squares contain numbers. The object of the puzzle is to fill in the remaining squares, so that every row, every column and every box contains each of the numbers from 1 to 9 exactly once.

For example, consider the following puzzle:

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

Some of the entries are obvious. For example, in the top-center box, the only place that a 6 can occur is in row 3, column 6. Similarly, there must be a 4 in row 4, column 5, a 1 in row 4, column 7, and a 3 in row 8, column 7.

Filling in the spaces one by one eventually leads to the solution:

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

Notice that in each row, each column and each box, each of the numbers 1 through 9 appears exactly once.

Your sudoku solver problem does not need to apply logic to determine the contents of the empty squares in the puzzle. It will simply use a *brute force* method of trying all possible combinations of values in the empty squares until it finds a combination that solves the puzzle.

Solution Method

It would actually be quite a difficult problem to write a sudoku solver using only iteration, somehow looping though all possible combinations. However, solution using recursion is remarkably easy.

The basis for any recursive solution to a problem is to solve the problem by identifying and solving an easier or smaller version of the same problem. Eventually, we must reach a version of the problem which is so simple or small that we can solve it immediately, without reference to an even easier problem. This is the base case.

Given a sudoku problem, we can find an empty square and make an arbitrary choice of a number to place in that square. Now we can attempt to solve the resulting puzzle, which is easier or smaller than the original puzzle because it has one less empty square. It may be that there is no solution to our easier puzzle, because our choice of number for that empty square was wrong. In this case our attempt to solve the easier puzzle fails , so we simply try another number in that same empty square and attempt to solve the resulting easier puzzle. Eventually, we will get to the base case: a puzzle with no empty squares, which is already solved.

Choosing an Empty Square and a Number for that Square

Given a sudoku puzzle to solve, we could identify an easier, smaller puzzle by repeatedly making a random choice of puzzle squares until we find one that is empty and then making a random

choice of the number to place in that square for the easier puzzle. This would eventually lead to a solution, but not as fast as it could be if we are more methodical in our choice of empty square and the value to be placed in that square.

To choose an empty square, we can simply search through the puzzle, row by row, until we find the first empty square.

To choose a number to place in that square we can try each of the numbers 1 through 9, one at a time. However, having chosen a number for the empty square, we can gain greater efficiency if, before attempting to solve the easier puzzle with that number in the empty square we check whether that number is compatible with the numbers already in the puzzle. If that number already appears in the row, column or box of the empty square, it cannot be used there.

If we reach the situation where we cannot find any number which can be placed in the empty square without conflicting with the numbers already in the puzzle, our attempt to solve the puzzle has failed. This would be because of an incorrect choice to fill an empty square in some harder puzzle that led to this easier puzzle.

Program Requirements

1. Your program must read a puzzle from a text file. The name of the file may be given as a command line argument. If there is no command line argument, the program must prompt the user for the name of the file. In either case, if the file cannot be opened for input (for example, if it cannot be found in the current directory), the program is to display an appropriate message and terminate.

2. Each puzzle file contains 9 rows of characters, with 9 characters in each row. Each character represents the value in one square of the puzzle. Empty squares are represented by space characters. Several puzzle files are available on the course web site.

3. Once the program has read the puzzle from the puzzle file, the program is to attempt to find a solution to the puzzle. If a solution can be found, the program must display the solution – just the final state of the solved puzzle, not the steps taken to reach that solution. If no solution can be found, the program is to display a message stating that there is no solution for that puzzle.

What you must submit

You must submit the Ada source file (the .adb file) for the sudoku solver program. This file must be included in a zipped tar file.

Saving your files in a zipped tar file

Note: you will probably have just one Ada source file for this lab, so these instructions may seem like an overkill, but this is the method you are to use for later labs and assignments, so treat this as practice!

You only submit .adb and .ads files. First you need to bundle them up into a single tar file. The term "tar" is an abbreviation of "tape archive" and goes back to the days when people would save a back-up copy of their files on magnetic tape. Nowadays, with the price of large disk drives so low, nobody uses tape anymore, but the concept of tar files survives.

Use the command:

```
tar -cf WnnnnnnnnAssg1.tar *.adb *.ads
```

(where Wnnnnnnnn is your W-number).

The -cf specifies two options for the tar command: 'c' means create and 'f' means that the name of the resulting tar file comes next in the command.

This will include every file in the current directory whose name ends with ".adb" or ".ads".

If you now use the command ls you should now see the file WnnnnnnnnAssg1.tar in your directory.

If you use the command

```
tar -tf WnnnnnnnnAssg1.tar
```

(where Wnnnnnnnn is your W-number), it will list the files within the tar file.

Now compress the tar file using the gzip program:

```
gzip WnnnnnnnnAssg1.tar
```

By using the ls command again, you should see the file WnnnnnnnnAssg1.tar.gz in your directory. This is the file that you need to submit through the **Assignment 1 Submission** link in the moodle web site.

Coding Standards

1. Use meaningful names that give the reader a clue as to the purpose of the thing being named.

2. Use comments at the start of the program to identify the purpose of the program, the author and the date written.

3. Use comments at the start of each procedure to describe the purpose of the procedure and the purpose of each parameter to the procedure.

4. Use comments at the start of each section of the program to explain what that part of the program does.

5. Use consistent indentation:

- The declarations within a procedure must be indented from the **procedure** and **begin** reserved words. The body of the procedure must be indented from the **begin** and **end** reserved words. Example of procedure indentation:

```
procedure DoStuff is
     Count : integer;
     Valid : boolean;
begin
     Count := 0;
     Valid := false;
end DoStuff;
```

- The statements within the then-part, each elsif-part and the else-part of an if statement must be indented from the reserved words if, elsif, else and end.

```
if Count > 4 and not Valid then
     Result := 0;
     Valid := true:
elsif Count > 0 then
     result := 4;
else
     Valid := false;
end if;
```

- The statements within a loop must be indented from the loop and end loop.

```
loop
     Count := Count + 1;
     Get (Number);
     exit when Number < Count;
end loop;
```

- The exception handlers and statements within each exception handler must be indented.

```
begin
     ...  -- normal processing statements
exception
     when Exception1 =>
          Put_Line ("An error has occurred");
          Total := 0;
     when others =>
          Put_Line ("Something weird happened");
end;
```