The University of Melbourne
Department of Computing and Information Systems

Declarative Programming
COMP90048

Semester 2, 2014

Project Specification

Project due 22 October 2014 at 5pm
Worth 15%

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

Fillin Puzzles

A fillin puzzle (sometimes called a fill-it-in) is like a crossword puzzle, except that instead of being given obscure clues telling which words go where, you are given a list of all the words to place in the puzzle, but not told where they go.

The puzzle consists of a grid of squares, most of which are empty, into which letters or digits are to be written, but some of which are filled in solid, and are not to be written in. You are also given a list of words or multi-digit numbers to place in the puzzle. Henceforth we shall discuss the puzzles in terms of words and letters, but keep in mind that they can be numbers and digits, too. You must place each word in the word list exactly once in the puzzle, either left-to-right or top-to-bottom, filling a maximal sequence of empty squares. Also, every maximal sequence of non-solid squares that is more than one square long must have one word from the word list written in it. Many words cross one another, so many of the letters in a horizontal word will also be a letter in a vertical word. For a properly constructed fillin puzzle, there will be only one way to fill in the words (in some cases, the puzzle is symmetrical around a diagonal axis, in which case there will be two symmetrical solutions).

Here is an example 7 by 7 fillin puzzle, together with its solution, taken from the Wikipedia page for fillin puzzles. In this example, one word is already written into the puzzle, but this is not required.

The word list for this puzzle is: GI IO ON OR DAG EVO OED REF ARID CLEF CLOD DAIS DENS DOLE EDIT SILO ARTICLE VESICLE.

The Program

You will write Prolog code to solve fillin puzzles. Your main predicate should be written in the file `fillin.pl`, but you may submit other prolog files if you like. If you do use multiple files, make sure `fillin.pl` loads the other files.

Your program should supply a predicate `main(`*PuzzleFile, WordlistFile, SolutionFile*`)` that reads in the puzzle file whose name is *PuzzleFile* and the word list file whose name is *WordlistFile*, solve the puzzle, and print out the result to the file *SolutionFile*.

The puzzle file will contain a number of lines each with the same number of characters, to form a rectangle. The characters in this file should all be either an underline character (`_`) indicating a fill-able square, a hash character (`#`) indicating a solid, non-fill-able square, or a letter or digit, indicating a pre-filled square. The output *SolutionFile* must have the same format (except that it should be filled, so it should not contain underlines). The word list file is simply a text file with one word per line.

You will be provided with code to read in these files and to print out the result in the form of a file `fillin_starter.pl`. You may use this code, modify it as you like, or ignore it, but you must provide the predicate `main(`*PuzzleFile, WordlistFile, SolutionFile*`)`, and your code must handle input files, and produce an output file, in the specified format. The starter file gives a trivial definition of the predicate `solve_puzzle`/3; you need only give a working definition of this predicate, and of course properly *document the predicates you are given* as well as the predicates you write. You should begin by renaming the starter file to `fillin.pl`.

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 fill-in puzzles of varying sizes that can be solved without search (see hint 7);

20% The ability of your program to correctly solve small to medium size puzzles that do require search;

20% The ability of your program to correctly solve puzzles up to 15 by 15 that do require search.

Note that timeouts will be imposed on all tests. You will have at least 20 seconds to solve each puzzle, regardless of difficulty. Executions taking longer than that will be unceremoniously terminated, leading to that test being assessed as failing. Twenty seconds 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. Be sure to document the predicates you take from the starter file you are given, too.

Submission

The project submission deadline is 22 October 2014 at 5pm. You must submit your project from one of the Linux servers `dimefox.eng.unimelb.edu.au` or `nutmeg.eng.unimelb.edu.au`. Note that this host is not directly accessible from outside the university campus, so if you wish to submit from home, you must use the university's VPN; see the LMS Resources page for guidance. Make sure the version of your program source files you wish to submit is on the server, `cd` to the directory holding your source code and issue the command:

```
submit COMP90048 proj2 fillin.pl
```

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

Important: you must wait a minute (more if the servers are busy) and do:

```
verify COMP90048 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 this, then your program did not work correctly. The version of SWI Prolog installed on the servers is older than you probably have on your own computer or the lab computers, so you should be sure to test your code on the servers. Do not just assume it will run 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. The basic strategy for solving these puzzles is to select a "slot" (*i.e.*, a maximal horizontal or vertical sequence of fill-able and pre-filled squares, and a word of the same length from the word list, fill the word into the slot in the puzzle, and remove the word

from the word list. It is important that the word selected should agree with any letters that are already filled into the slot. Repeat this process until the puzzle is completely solved and the word list is empty.

2. To avoid writing much code to handle filling slots vertically, you can just transpose the puzzle, use operations to handle horizontal puzzle slots, and then transpose the puzzle back. It is easier to detect and handle horizontal slots than vertical ones.

   The SWI Prolog library provides two different, incompatible `transpose`/2 predicates, and unfortunately autoloads the wrong one by default. If you wish to use the correct one, you should put the line

   ```
   :- ensure_loaded(library(clpfd)).
   ```

   in your source file. This defines the predicate `transpose(`*Matrix0, Matrix*`)` that holds when *Matrix0* and *Matrix* are lists of lists, and the "columns" of each are the "rows" of the other.

3. One of Prolog's features, known as the "logical variable" can make this job easier. You can construct a list of "slots" where each slot is a list of Prolog variables representing one square in the puzzle. If you make sure that the same variable is used for the same square, whether it appears in a vertical or horizontal slot, then Prolog will ensure that the letter placed in the box for a horizontal word is the same as for an intersecting vertical word.

   For example, suppose you have a 3×3 puzzle with the four corners filled in solid. This would be represented as a file with these lines:

   ```
   #_#

   ---
   #_#
   ```

   The slots in this puzzle could be represented as `[[A,B,C], [X,B,Z]]`, in which case the filled-in puzzle will be `Filled = [['#',X,'#'], [A,B,C], ['#',Z,'#']]`. Then unifying `[A,B,C]` with `[c,a,t]` will bind `B` to `a` in both lists, ensuring that the first list is only unified with a word that has an `a` as second letter. Note that after the second list is unified with `[c,a,t]`, `Filled` will automatically become `[['#',c,'#'], [A,a,C], ['#',t,'#']]`. Once the first list is unified, binding `A` and `C`, this will be the final solved puzzle.

   With this approach, the list of slots is simply a permutation of the list of words. For small puzzles, it is sufficient to backtrack over permutations of the wordlist until it can be unified with the slot list.

4. As an alternative to Hint 3, you may instead approach the problem by repeatedly transforming the puzzle by filling in slots in the puzzle. With this approach, the 3×3 puzzle discussed above would be represented as

   ```
   [['#','_','#'], ['_','_','_'], ['#','_','#']]
   ```

   Filling in "cat" in the vertical slot would leave:

```
[['#',c,'#'], ['_',a,'_'], ['#',t,'#']]
```

(again, it is easiest to work with horizontal slots; a vertical slot can be handled by transposing the puzzle, handling a horizontal slot, and then transposing again). As words are filled in, some of the letters of some slots will be filled in. Care must be taken when filling partially-filled slots to ensure the word being placed matches the letters already placed in that slot. Care must also be taken that a slot that has had all its letters filled by filling perpendicular words is in fact a word, and that word is removed from the word list.

This approach is simpler to understand than that of Hint 3, but requires considerably more code, and more work.

5. The number of permutations of a list of length n is $n!$. Even a 7×7 puzzle may have 18 words, which has $18! > 10^{15}$ permutations. Careful use of Prolog can substantially reduce this search space through the use of fast failure. That is, as each word is selected for its slot, it should be unified immediately with the slot. If this fails, then all further permutations involving that word in that slot can be discarded (Prolog will automatically do that).

6. Hint 5 will allow medium-size puzzles to be solved. For larger puzzles, the power of the factorial is too great. To solve those, it is necessary to avoid more of the search space. This can be done by carefully choosing the order in which to place words in slots. For example, if there is only one 6-letter word, then place that first. Once that word is placed, some letters in others words will be filled in; perhaps one of them will have only one matching word, and so on. Each time a word is to be placed, you should count the number of words that match each slot, and select (one of) the slot(s) with the fewest matching words to fill. This minimises the search space.

7. For some puzzles, as they are being filled in, there is always at least one slot that has only one word that can possibly fit. These puzzles are easier to solve, because no search is actually required. 30% of the test cases will be like this.

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.