ECS 140A Programming Languages

Winter 2019

Homework 4

About This Assignment

- This assignment asks you to complete programming tasks using the SWI-Prolog programming language.
- To complete the assignment (i) download `hw4-handout.zip` from Canvas, (ii) modify the `.pl` and `.plt` files in the `hw4-handout` directory as per the instructions in this document, and (iii) zip the `hw4-handout` directory into `hw4-handout.zip` and upload this `zip` file to Canvas by the due date.

  Do not change the file names, create new files, or change the directory structure in `hw4-handout`.
- This assignment has to be worked on individually.
- We will be using the SWI-Prolog implementation of the Prolog programming language, version 7.6.4, which can be installed from http://www.swi-prolog.org/download/stable?show=all.

  Use the command `swipl --version` to verify that you have the correct version installed:

  ```
  $ swipl --version
  SWI-Prolog version 7.6.4<other output>
  ```

- SWI-Prolog 7.6.4 is also installed on *all* CSIF machines. For instance,

  ```
  $ ssh <kerberos-id>@pc2.cs.ucdavis.edu
  <ssh output>
  $ swipl --version
  SWI-Prolog version 7.6.4<other output>
  ```

- Information about using CSIF computers, such as how to remotely login to CSIF computers from home and how to copy files to/from the CSIF computers using your personal computer, can be found at http://csifdocs.cs.ucdavis.edu/about-us/csif-general-faq.
- Begin working on the homework early.

- Apart from the description in this document, look at the unit tests provided to understand the requirements for the code you have to write.

  We are using the `plunit` test framework.[1]

  You may need to modify/write new tests in order to achieve full code coverage.

- Post questions on Piazza if you require any further clarifications. Use private posts if your question contains part of the solution to the homework.

  Ensure that your questions on piazza are specific. Even when using private posts, do not post your code and ask someone else to debug it for you.

General Tips

- When developing your program, you might find it easier to first test your predicate interactively before using the test program. You might find trace predicate useful in debugging your predicate. You can find information on tracing and debugging here: http://www.swi-prolog.org/pldoc/man?section=debugger.

- The command `swipl myFile.pl` runs the swipl interpreter with functions defined in `myFile.pl` already loaded (but not run).

- You can start swipl interactively using:

  ```
  $ swipl
  ```

- To load function definitions from `myFile.pl` in the current directory (notice the `.` at the end of the command, this should be at the end of every function call and at the end of the last line of every function definition):

  ```
  ?- [myfile].
  ```

- To exit error mode (i.e. an exception was thrown), type `a` (for abort):

  ```
  ?- bad_func(parameters).
  <error output>
     Exception: <error output> ? abort
  % Execution Aborted
  ?-
  ```

- You can exit the interactive swipl interpreter using:

  ```
  ?- halt.
  ```

[1] http://www.swi-prolog.org/pldoc/package/plunit.html

1 query (30 points)

- You are given a set of facts of the following form in `hw4-handout/query/facts.pl`:
  - `novel(name, year).`

    Here `name` is an atom denoting the name of the novel and `year` denotes the year that the novel has been published.

    For example, the fact `novel(the_kingkiller_chronicles, 2007).` says that the novel named `the_kingkiller_chronicles` was published in the year 2007.
  - `fan(name, novels_liked).`

    Here `name` is an atom denoting the name of the person and `novels_liked` denotes the list of novels liked by that person.

    For example, the fact `fan(joey, [little_women]).` says that the person named `joey` is a fan of the novel named `little_women`.
  - `author(name, novels_written).`

    Here `name` is an atom denoting the name of the author and `novels_written` denotes the list of novels written by that author.

    For example, the fact `author(george_rr_martin, [a_song_of_ice_and_fire_series]).` says that the author named `george_rr_martin` has written the novel named `a_song_of_ice_and_fire_series`.
- Complete the definition of the predicate `year_1953_1996_novels(Book)` in `hw4-handout/query/query.pl`, which is true if `Book` is a novel that has been published in either 1953 or 1996.
- Complete the definition of the predicate `period_1800_1900_novels(Book)` in `hw4-handout/query/query.pl`, which is is true if `Book` is a novel that has been published during the period 1800 to 1900.
- Complete the definition of the predicate `lotr_fans(Fan)` in `hw4-handout/query/query.pl`, which is true if `Fan` is a name of a person that is a fan of `the_lord_of_the_rings`.
- Complete the definition of the predicate `author_names(Author)` in `hw4-handout/query/query.pl`, which is true if `Author` is an author whose novels `chandler` is a fan of.
- Complete the definition of the predicate `fans_names(Fan)` in `hw4-handout/query/query.pl`, which is true if `Fan` is a person who is a fan of the novels authored by `brandon_sanderson`.
- Complete the definition of the predicate `mutual_novels(Book)` in `hw4-handout/query/query.pl`, which is true if `Book` is a novel that is common between either of `phoebe`, `ross`, and `monica`.

- Use the following commands to run the unit tests provided in `hw4-handout/query/query.plt`:

```
$ cd hw4-handout/query/
$ swipl -s query.plt
```

- Ensure that the coverage for the `hw4-handout/query/query.pl` file is 100%. See the `%Cov` column in the output of the unit tests above.

  Write additional tests, if needed, in `hw4-handout/query/query.plt`.

- Note that a different set of facts will be used while grading this question.

2 set (40 points)

- In this question, we will represent sets as lists, where each element of a set appears exactly once on its list, but in no particular order. Do not assume you can sort the lists. Do assume that input lists have no duplicate elements, and do guarantee that output lists have no duplicate elements.

- Complete the definition of the predicate `isUnion(Set1,Set2,Union)` in `hw4-handout/set/set.pl` which is true if `Union` is the union of `Set1` and `Set2`.

  Do not use the predefined list predicate `union`. Your predicate may choose a fixed order for `Z`. If you query `isUnion([1,2],[3],Z)` it should find a binding for `Z`, but it need not succeed on both `isUnion([1],[2],[1,2])` and `isUnion([1],[2],[2,1])`. Your predicate need not work well when `X` or `Y` are unbound variables.

  For example, `isUnion([1,2],[3],Z).` returns `Z = [1,2,3]`.

- Complete the definition of the predicate `isIntersection(Set1,Set2,Intersection)` in `hw4-handout/set/set.pl`, which is true if `Intersection` is the intersection of `Set1` and `Set2`.

  Do not use the predefined list predicate `intersection`. Your predicate may choose a fixed order for `Z`. Your predicate need not work well when `X` or `Y` are unbound variables.

  For example, `isIntersection([1,2],[3],Z).` returns `Z = []`.

- Complete the definition of the predicate `isEqual(Set1,Set2)` in `hw4-handout/set/set.pl`, which is true when `Set1` is equal to `Set2`. Two sets are equal if they have exactly the same elements, regardless of the order in which those elements are represented in the set. Your predicate need not work well when `X` or `Y` are unbound variables.

  For example, `isEqual([a,b],[b,a]).` returns `true`.

- Complete the definition of the predicate `powerSet(Set,PowerSet)` in `hw4-handout/set/set.pl`, which is true if `PowerSet` is the powerset of `Set`.

  The powerset of a set is the set of all subsets of that set. For example, consider the set `A={1,2,3}`. It has various subsets: {1},{1,2} and so on. And of course the empty set ∅ is a subset of every set. The powerset of A is the set of all subsets of A:

  $$P(S) = \{\emptyset,\{1\},\{2\},\{3\},\{1,2\},\{2,3\},\{1,3\}, \{1,2,3\}\}.$$

  For your `powerSet` predicate, if `X` is a list(representing the set), `Y` will be a list of lists (representing the set of all subsets of the original set). So `powerSet([1,2],Y)` should produce the binding `Y = {{1,2},{1},{2},{}}` (in any order). Your predicate may choose a fixed order for `Y`. Your predicate need not work well when `X` is an unbound variable.

- Use the following commands to run the unit tests provided in `hw4-handout/set/set.plt`:

```
$ cd hw4-handout/set/
$ swipl -s set.plt
```

- Ensure that the coverage for the `hw4-handout/set/set.pl` file is 100%. See the `%Cov` column in the output of the unit tests above.

  Write additional tests, if needed, in `hw4-handout/set/set.plt`.

3 nfa (10 points)

- An non-deterministic finite automaton (NFA) is defined by a set of states, symbols in an alphabet, and a transition relation.

  A state is represented by an integer. A symbol is represented by a character.

  Given a state `St` and a symbol `Sym`, a transition relation `transition(St, Sym, States)` defines the list of states `States` that the NFA can transition to after reading the given symbol. This list of next states could be empty.

  A graphical representation of an NFA along with the corresponding transition facts are shown below.

```
transition(0, a, [1,2]).
transition(0, b, [2]).
transition(1, a, []).
transition(1, b, [0]).
transition(2, a, []).
transition(2, b, []).
```

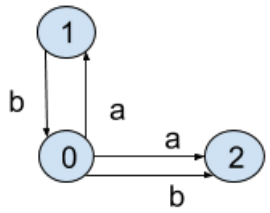

Figure 1: Example NFA diagram with corresponding `transition` facts.

- In this example, $\{0, 1, 2\}$ are the set of states, $\{a, b\}$ are the set of symbols, and the transition function is represented by labelled arrows between states.
  - If the NFA is in state 0 and it reads the symbol a, then it can transition to either state 1 or to state 2.
  - If the NFA is in state 0 and it reads the symbol b, then it can only transition to state 2.
  - If the NFA is in state 1 and it reads the symbol b, then it can only transition to state 0.
  - If the NFA is in state 1 and it reads the symbol a, it cannot make any transitions.
  - If the NFA is in state 2 and it reads the symbol a or b, it cannot make any transitions.
- A given final state is said to be *reachable* from a given start state via a given input sequence of symbols if there exists a sequence of transitions such that if the NFA starts at the start state it would reach the final state after reading the entire sequence of input symbols.
- In the example NFA above:
  - The state 1 is reachable from the state 0 via the input sequence *abababa*.
  - The state 1 is *not* reachable from the state 0 via the input sequence *ababab*.
  - The state 2 is reachable from state 0 via the input sequence *abababa*.
- Complete the definition of the predicate `reachable(StartState, FinalState, Input)` in `hw4-handout/nfa/nfa.pl`, which which is true if state `FinalState` is reachable from the state `StartState` after reading the input list of symbols in `Input`.

  The `transition` facts are listed in `hw4-handout/nfa/transition.pl`.
- Use the following commands to run the unit tests provided in `hw4-handout/nfa/nfa.plt`:

```
$ cd hw4-handout/nfa/
$ swipl -s nfa.plt
```

- Ensure that the coverage for the `hw4-handout/nfa/nfa.pl` file is 100%. See the `%Cov` column in the output of the unit tests above.

  Write additional tests, if needed, in `hw4-handout/nfa/nfa.plt`.
- A different set of the `transition` facts will be used when grading.

4 puzzle (30 points)

- This question entails a farmer, a wolf, a goat, and a cabbage. All four are near a river. The farmer has a boat that can carry only one of the other three.

  Each of the four might initially be on either the left or right bank of the river.

  The farmer is tasked with moving all four of them from some initial configuration to a given final configuration.

  For example, all four of them might initially be on the left bank, and the farmer needs to use the boat so that they end up in the configuration in which all four of them are on the right bank of the river.

- However, if the wolf and the goat are on the same river bank without the farmer, the wolf will eat the goat. If the goat and the cabbage are on the same river bank without the farmer, the goat will eat the cabbage.

  Thus, the farmer should avoid such *unsafe* configurations.

- For example, it is possible to move farmer, wolf, goat and cabbage from left bank to right bank by avoiding the above unsafe configurations.

  It is NOT possible to start with the farmer and cabbage on the left bank and the wolf and goat on the right bank, and then end with farmer, wolf, goat and cabbage on the right bank. (The wolf will eat the goat when they are alone on the right bank without the farmer.)

  It is NOT possible to start with farmer, wolf, goat and cabbage on the left bank, and end up with the farmer and wolf on the right bank, and the goat and cabbage on the left bank. (The goat will eat the cabbage when they are alone on the left bank without the farmer.)

- The terms `left` and `right` denote the left and right banks of the river.

- Complete the definition of the predicate `solve(F1, W1, G1, C1, F2, W2 ,G2 ,C2)` in `hw4-handout/puzzle/puzzle.pl`, which is true if and only if there exists a way to go from the initial state of the farmer (`F1`), wolf (`W1`), goat (`G1`), and cabbage (`C1`) to their respective final state `F2, W2, G2` and `C2`.

  For example, `solve(left, left, left, left, right, right, right, right)` returns `true`.

- Use the following commands to run the unit tests provided in `hw4-handout/puzzle/puzzle.plt`:

```
$ cd hw4-handout/puzzle/
$ swipl -s puzzle.plt
```

- Ensure that the coverage for the `hw4-handout/puzzle/puzzle.pl` file is 100%. See the `%Cov` column in the output of the unit tests above.

  Write additional tests, if needed, in `hw4-handout/puzzle/puzzle.plt`.

- Hint: defining the following terms might help when constructing your solution:

  - Define a term `state(F,W,G,C)` that denotes on which bank the farmer (F), the wolf (W), the goat (G), and the cabbage (C) are on.
    For example, `state(left,left,right,left)` denotes the state in which the farmer, the wolf, and the cabbage are on the left bank, and the goat is alone on the right bank.
  - Define a term, `opposite(A,B)`, that is true if `A` and `B` are different banks of the river.
  - Define a term, `unsafe(A)`, to indicate if state `A` is unsafe.
  - Similarly define a term, `safe(A)`.
  - Define a term, `take(X,A,B)`, for moving object `X` from bank `A` to bank `B`.
  - Define a term, `arc(N,X,Y)`, that is true if move `N` takes state `X` to state `Y`. Define the rule for the above terms. Now, the solution involves searching from a certain initial state to a final state. You may look at relevant examples in the textbook on how you can write your search algorithms.