Middle East Technical University Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring 2020-2021

Programming Exam 9

Due date: June 24, 2021, Thursday, 23:59

1 Problem Definition

In this lab exam, you are going to implement predicates that operate on a knowledge base with people. These people have their names, ages and hobbies.

1.1 Knowledge Base Predicate

The **person** predicate is used to define people and their attributes. Its format is given below:

```
person(Name, Age, Hobby).
```

The **word** predicate is used to define words in the knowlegde base. Its format is given below:

```
word(Word).
```

Some of the facts from the example knowledge base given to you can be seen below. (**Please note that it is not complete.** To see the full version, you should see the kb.pl file.)

```
person(joseph, 27, reading).
person(jessica, 32, crafting).
person(michael, 22, reading).
person(william, 33, reading).
person(elizabeth, 30, television).
person(jennifer, 38, crafting).
person(robert, 22, television).
word(acaba).
word(adana).
word(alaca).
word(araba).
word(eleme).
word(fiili).
word(ikili).
word(ama).
word(masa).
word(maya).
word(uyku).
word(vana).
```

2 Specifications

In this lab exam, you are expected to code 3 different predicates with varying difficulties.

2.1 The bigram predicate - 30 pts

It has 2 arguments. The first one is a word in the format of an atom. The second one is the Result. The result should be an atom and it should be one of the bigrams of the given word. A bigram of a word is 2 adjacent characters in it. For example, the bigrams of the word hello are he, el, ll, and lo. The order of returning bigrams is not important. You can split an atom and obtain a list of character atoms by using **atom_chars** (https://www.swi-prolog.org/pldoc/man?predicate=atom_chars/2). You can also do the reverse procedure using the same predicate.

Format of the predicate is given below:

```
bigram(Word, ResultBigram).
```

Examples:

```
?- bigram(h, R).
false.

?- bigram(he, R).
R = he ;
false.

?- bigram(hel, R).
R = he ;
R = el ;
false.

?- bigram(hello, R).
R = he ;
R = el ;
R = ll ;
R = lo ;
false.

?- bigram(hellolo, R).
R = he ;
R = el ;
R = ll ;
R = lo ;
R = ol ;
R = lo ;
false.
```

2.2 The num_hobbies predicate - 35 pts

It has 2 arguments. The first one is a list containing names. The second one is the result list.

The second argument is the hobby list belonging to the people in the first list but reported in a more compact way. It is a list containing hobby predicates where each hobby predicate contains the name of the hobby, and the number of occurrences of that hobby. hobby predicate can be seen below:

```
hobby(HobbyName, NumberOfOccurrences).
```

For example, let's say that our query is num_hobbies([joseph, jessica, michael], ResultList). Their hobbies are reading, crafting, and reading, respectively. Therefore, we have 2 reading and 1 crafting hobbies and the ResultList will consist of hobby(reading, 2) and hobby(crafting, 1). The order is **not** important. (You may want to use \= operator to test for not-matching of two terms. For example, X\=Y evaluates to true if and only if X=Y evaluates to false.)

Format of the predicate is given below:

```
num_hobbies(NameList, ResultList).
```

Examples:

```
?- num_hobbies([], ResultList).
ResultList = [].

?- num_hobbies([joseph], ResultList).
ResultList = [hobby(reading, 1)] ;
false.

?- num_hobbies([joseph, jessica], ResultList).
ResultList = [hobby(reading, 1), hobby(crafting, 1)] ;
false.

?- num_hobbies([joseph, jessica, michael], ResultList).
ResultList = [hobby(reading, 2), hobby(crafting, 1)] ;
false.

?- num_hobbies([joseph, jessica, michael, william, patricia, karen], ResultList).
ResultList = [hobby(reading, 3), hobby(crafting, 1), hobby(bird_watching, 2)] ;
false.
```

2.3 The sentence_match predicate - 35 pts

It has 2 arguments. The first one is a list containing words given as atoms. The second one is the result list which also contains words in the format of atoms.

In the knowledge base, all possible words are given as facts. For example, we have words like word(ama) or word(tava). Let's say that we have an encrypted word "tzt" and if we map t↔a and z↔m, then from "tzt", a word in our knowledge base can be obtained (word(ama)).

The sentence_match predicate tries to map every word in the first argument into a word in the knowledge base character by character and while doing that it tries to find a consistant mapping thoughout the words. So if we map a↔b, then we cannot do mappings like a↔c (a cannot be mapped both b and c) or c↔b (both a and c cannot be mapped to b). Of course, b↔c is still consistant with a↔b since b is on the left-hand side in the first one. So the character mapping should be one-to-one and onto, although some of the characters may remain unmatched if they do not occur.

Some important information:

- Only lower-case characters are used as words.
- The order in the ResultList is important but the order in your output after pressing semicolon (;) does not matter.
- The encrypted words in the first argument list doesn't have to appear in the knowledge base.
- You can split an atom and obtain a list of character atoms by using **atom_chars** (https://www.swi-prolog.org/pldoc/man?predicate=atom_chars/2).

- You may want to first define a predicate that matches only one word and maybe also returns the map it used for that.
- You may also want to use **not**, **!** (cut), or **member**.

Format of the predicate is given below:

```
sentence_match(WordList, ResultList).
```

Examples:

```
?- sentence_match([] ,R).
R = [].

?- sentence_match([xejeye],R).
R = [badana] ;
R = [binici] ;
R = [birisi] ;
R = [deneme] ;
R = [derece] ;
R = [duyuru] ;
R = [kabaca] ;
R = [kamara] ;
R = [kasaba] ;
R = [lahana] ;
R = [nerede] ;
R = [salata] ;
R = [tabaka] ;
R = [yarasa] ;
R = [yasama] ;
false.

?- sentence_match([irki,indi],R).
R = [arka, abla] ;
R = [abla, arka] ;
false.

?- sentence_match([irki,indi,icik],R).
R = [arka, abla, ayak] ;
false.

?- sentence_match([xejeye, mem, mexebe],R).
R = [badana, tat, tabaka] ;
false.
```

3 Regulations

- **Implementation and Submission**: The template files are available in the Virtual Programming Lab (VPL) activity called "PE8" on ODTUCLASS. At this point, you have two options:
  - You can download the template files, complete the implementation and test it with the given sample I/O on your local machine. Then submit the same file through this activity.
  - You can directly use the editor of VPL environment by using the auto-evaluation feature of this activity interactively. Saving the code is equivalent to submitting a file.

Please make sure that your code runs on ODTUCLASS. There is no limitation in running your code online. The last save/submission will determine your final grade.

- **Programming Language**: You must code your program in Prolog. Your submission will be run with swipl on ODTUCLASS. You are expected make sure your code runs successfully with swipl on ODTUCLASS.
- **Modules**: You are not allowed to import any modules. However, you are allowed to use the base predicates present in prolog or define your own predicates.
- **Cheating**: Using code from any source other than your own is considered cheating. This includes but not limited to internet sources, previous homeworks/exams, and friends. In case of cheating, the university regulations will be applied.
- **Grading**: You can evaluate your code interactively on Cengclass. However, note that the evaluation test cases are only provided to assist you and are different from the test cases that will be used during grading. Thus, the grade resulting from the evaluation may not be your final grade.
- **Late Submission**: There is no late submission.