Department of Computer Science
University of Pretoria

Programming Languages
COS 333

Practical 3: Logic Programming

April 6, 2024

1 Objectives

This practical aims to achieve the following general learning objectives:

- To gain and consolidate some experience writing logic programs in Prolog;
- To consolidate the concepts covered in Chapter 16 of the prescribed textbook.

2 Plagiarism Policy

Plagiarism is a serious form of academic misconduct. It involves both appropriating someone else's work and passing it off as one's own work afterwards. Thus, you commit plagiarism when you present someone else's written or creative work (words, images, ideas, opinions, discoveries, artwork, music, recordings, computer-generated work, etc.) as your own. Note that using material produced in whole or part by an AI-based tool (such as ChatGPT) also constitutes plagiarism. Only hand in your own original work. Indicate precisely and accurately when you have used information provided by someone else. Referencing must be done in accordance with a recognised system. Indicate whether you have downloaded information from the Internet. For more details, visit the library's website: `http://www.library.up.ac.za/plagiarism/`.

3 Submission Instructions

Upload your practical-related source code file to the appropriate assignment upload slot on the ClickUP course page. You will be implementing all the practical's tasks in the same file. Name this file `s99999999.pl`, where `99999999` is your student number. Multiple uploads are allowed, but only the last one will be marked. The submission deadline is **Monday, 22 April 2024, at 12:00**.

4 Background Information

For this practical, you will be writing programs in SWI-Prolog version 9.0.4:

- Write all your Prolog programs (consisting of facts and rules) in a single source file. To do this, launch the SWI-Prolog interpreter in Windows, and select "New..." under the "File" menu. Type a file name, and click "Save". An edit dialogue will then pop up, through which you can write your program source file. Source code files are saved by selecting "Save buffer" under the "'File' menu. Note that SWI-Prolog is sensitive to end of line characters, so it is strongly recommended that you use this built in editor to write your programs. Also, be sure to place facts and rules on separate lines. Note that only Prolog predicate implementations are provided in your source file. You should not provide queries in your program source code.
- In order to test your implementation, select "Consult..." under the "File" menu, then choose the program source file you have written. You can then type in queries in the main SWI-Prolog window, which is in interactive mode.
- In interactive mode, a `true` response means that the interpreter can prove your query to be true, while a `false` response means that the interpreter cannot prove your query to be true. If you include variables in your query, the interpreter will respond with a value that makes the query true, or `false` if it cannot find such a value. Pressing `r` after a query response will re-query the interpreter (you can do this repeatedly, until Prolog runs out of responses), while pressing `Enter` will end the query.
- The course ClickUP page contains documentation related to SWI-Prolog [1], which contains detailed information on the operation of the SWI-Prolog interpreter, and details on the implementation of the Prolog programming language that SWI-Prolog provides.
- **Note that you may only use the simple constants, variables, list manipulation methods, and built-in predicates discussed in the textbook and slides. In particular, do not use if-then, if-then-else, and similar constructs. You may NOT use any more complex predicates provided by the Prolog system itself. In other words, you must write all your own propositions. Failure to observe this rule will result in all marks for a task being forfeited.**
- **You may implement and use the propositions defined in the textbook and slides (e.g. `member`, `append`, and `reverse`). Note that you must provide the implementation for any of these propositions in your source file. Also note that there are some built-in propositions that correspond to the propositions defined in the textbook, which you may not use.**
- You may implement helper propositions if you find them necessary.

5 Practical Tasks

For this practical, you will need to explore and implement logic programming concepts, including list processing. All of the following tasks should be implemented in a single source code file.

5.1 Task 1

Define a series of facts relating to marriage and pet ownership, such as the following:

```
married(peter, mary).
married(lilly, joseph).

ownsPet(peter, rover).
ownsPet(mary, fluffy).
ownsPet(joseph, tweety).
ownsPet(lilly, fluffy).
```

Note that the **only** facts you are allowed to define are the `married` and `ownsPet` proposition. If you define any facts using additional propositions, you will **forfeit all marks for this task**. The `married(X, Y)` proposition means that person `X` is married to person `Y`. The `ownsPet(X, Y)` proposition means that person `X` owns pet `Y`. For simplicity, assume that only two people can be married to one another. Also note the following:

- There need not be facts covering inverse marriage relationships. For example, the `married(mary, peter)` and `married(joseph, lilly)` facts need not be present, although they are implied.
- Pet ownership does not have to extend to the second party in a marriage. For example, the facts `ownsPet(mary, rover)` and `ownsPet(lilly, fluffy)` are not implied, and need not be provided.

You must define the following propositions by means of rules that use the fact propositions listed above:

- The proposition `household(X, Y)`, which is true when person `X` is in the same household as person `Y`. Two people are considered to be in the same household if they are married. Given the facts in the example above, the queries `household(peter, mary)`, `household(mary, peter)`, `household(lilly, joseph)`, and `household(joseph, lilly)` are all true.
- The proposition `householdPet(O1, O2, P)`, which is true when pet `P` is owned by someone in the household of persons `O1` and `O2`. It is suggested that you use the `household` proposition in the rule (or rules) for the `householdPet` proposition. Given the facts in the example above, `rover` is a household pet in the household of `peter` and `mary`, and `tweety` is a household pet in the household of `lilly` and `joseph`. Additionally, `fluffy` is a household pet both in the household of `peter` and `mary`, and in the household of `lilly` and `joseph` (because `fluffy` is a pet of both `mary` and `lilly`). Note that `O1` and `O2` are reversible, so both `householdPet(peter, mary, rover)` and `householdPet(mary, peter, rover)` are true.
- The proposition `wanderingPet(P)`, which is true when pet `P` is a household pet of two households. It is suggested that you use the `householdPet` proposition in the rule (or rules) for the `wanderingPet` proposition. Given the facts in the example above, `fluffy` is the only wandering pet, because it is a household pet of two households. Neither `rover` nor `tweety` are considered wandering pets.

Hint 1: Be sure to test all your propositions thoroughly, including both propositions that should be true, as well as propositions that should be false.

Hint 2: To test whether invalid objects are included in your proposition, try entering queries involving variables, such as `wanderingPet(X)`. If you re-query repeatedly, this will list all objects for `X` that satisfy the proposition.

5.2 Task 2

Write a Prolog proposition named `addPositives` that has two parameters. The first parameter is a simple numeric list (i.e. a list containing only integers), while the second parameter is an integer. The proposition defines the second parameter to be the sum of the positive non-zero values contained in the first parameter. To illustrate the use of the `addPositives` proposition, consider the following queries and responses:

```
?- addPositives([], X).
X = 0.

?- addPositives([-1, -5, 0], X).
X = 0.

?- addPositives([-1, 5, 0, 2, -5, 1], X).
X = 8.
```

Test the `addPositives` proposition using the provided example queries, as well as your own test input, and verify that the proposition works as you expect.

Hint: You can use comparison operators as terms in Prolog. These operators are similar to what you are used to in imperative languages. The operators `<` (less than), `=<` (less than or equal to), `>` (greater than), and `>=` (greater than or equal to) are all valid. As a very simple example, you could use a comparison operator as follows:

```
lessThan(X, Y) :- X < Y.
```

The following queries are then possible:

```
?- lessThan(1, 2).
true.

?- lessThan(2, 1).
false.
```

Note that this is just an example, and does not mean you should use this `lessThan` proposition in your implementation (although you may if you wish to).

5.3 Task 3

Write a Prolog proposition named `getEverySecondValue` that has two parameters. The first parameter is a simple list (i.e. a list containing only atoms or integers), while the second parameter is also a simple list. The proposition defines the second parameter to be a list containing every second value in the first parameter. To illustrate the use of the `getEverySecondValue` proposition, consider the following queries and responses:

```
?- getEverySecondValue([], X).
X = [].

?- getEverySecondValue([a], X).
X = [].

?- getEverySecondValue([a, b, c, d, e], X).
X = [b, d].
```

Test the `getEverySecondValue` proposition using the provided example queries, as well as your own test input, and verify that the proposition works as you expect.

Hint 1: You can represent a list containing a single element in a proposition using standard list notation, without including a tail. As a very simple example, you could define the following proposition:

```
singleElementList([H]).
```

In this proposition, the variable `H` could also be replaced with an underscore, because it is not used. The proposition is true for any list containing only a single element. The following queries are then possible:

```
?- singleElementList([a]).
true.

?- singleElementList([a, b]).
false.
```

Note that this is just an example, and does not mean you should use this `singleElementList` proposition in your implementation (although you may if you wish to).

Hint 2: You can represent multiple elements in a list, beyond just the head. As a very simple example, you could define the following proposition:

```
secondElement([First, Second|Tail], Second).
```

In this proposition, the variables `First` and `Tail` could also be replaced with underscores, because they are not used. The proposition is true if the second parameter is the second value contained in the first parameter list. The following query is then possible:

```
?- secondElement([a, b, c], X).
X = b.
```

Note that this is just an example, and does not mean you should use this `secondElement` proposition in your implementation (although you may if you wish to).

6 Marking

Each of the tasks will count 5 marks for a total of 15 marks. Submit both tasks implemented in the same source code file. Do not upload any additional files other than your source code. Both the implementation and the correct execution of the propositions will be taken into account. **You will receive zero for a task that uses a language feature you are not allowed to use**. Your program code will be assessed during the practical session in the week of **Monday, 22 April 2024**.

References

[1] Jan Wielemaker. *SWI-Prolog Reference Manual*. University of Amsterdam, January 2018.