Department of Computer Science
University of Pretoria

Programming Languages
COS 333

Practical 3: Logic Programming

March 17, 2025

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 `u99999999.pl`, where `99999999` is your student number. Multiple uploads are allowed, but only the last one will be marked. The submission deadline is **Monday, 31 March 2025, 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 personhood and marriage, such as the following:

```
person(peter).
person(mary).
person(tom).
person(lilly).
person(joseph).
person(rob).

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

invited(mary, tom).
```

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

- Personhood does not imply a marriage relationship or an invited relationship. For example, the fact `person(rob)` is possible, even though `rob` does not appear in a `married` fact or a `invited` fact.
- 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.
- There need not be facts covering invitations from both parties in a marriage relationship. For example, the `invited(peter, tom)` fact need not be present, although it is implied.

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

- The proposition `liveTogether(X, Y)`, which is true when person `X` is married to person `Y`. Given the facts in the example above, the queries `married(peter, mary)`, `married(mary, peter)`, `married(lilly, joseph)`, and `married(joseph, lilly)` are all true.
- The proposition `visiting(X, Y)`, which is true when person `X` is visiting person `Y`. A person is visiting anyone who lives in a house from which an invitation has been extended. Given the facts in the example above, `visiting(tom, mary)` and `visiting(tom, peter)` are both true (even though only `mary` has invited `tom`, `tom` is also visiting `peter` because `mary` and `peter` are married). It is suggested that you use the `liveTogether` proposition in the rule (or rules) for the `visiting` proposition.
- The proposition `livesAlone(X)`, which is true when person `X` is not married and has not been invited to live with anyone. Given the facts in the example above, only `livesAlone(rob)` is true. It is suggested that you use the `liveTogether` and `visiting` propositions in the rule (or rules) for the `livesAlone` proposition.

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 `livesAlone(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 `getZeroValues` that has two parameters. Both parameters are simple numeric lists (i.e. lists containing only integers). The proposition defines the second parameter to be list containing only the zero values contained in the first parameter list. To illustrate the use of the `getZeroValues` proposition, consider the following queries and responses:

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

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

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

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

Hint: You may find that when you re-query a successful result, you get an unexpected response. For example, you may test your proposition on a list, and get the correct instantiation of the second parameter, but then re-querying produces a `false` response.

This usually happens because, on backtracking, a base case or recursive case is re-evaluated using an inappropriate recursive case, which then fails. This isn't a problem for this question. However, if you wish to avoid this behaviour, you can add a cut. A cut is a goal that is automatically satisfied, but does not allow backtracking past it. It can be used as follows:

```
proposition([]) :- !.
```

Assuming that the empty list denotes a base case, the `!` symbol is a cut. What this means is that once the base case of an empty list is reached, we can't get Prolog to backtrack and find another way to satisfy the base case. This also works for recursive cases, where the cut is typically provided as the last sub-goal in the antecedent.

5.3 Task 3

Write a Prolog proposition named `monotonicallyDecreasing` that has one parameter. The parameter is a simple numerical list (i.e. a list containing only integers). The proposition is true if the list contains integers that are monotonically decreasing (i.e. moving from left to right in the list, values either decrease in value or stay the same, but never increase in value). To illustrate the use of the `monotonicallyDecreasing` proposition, consider the following queries and responses:

```
?- monotonicallyDecreasing([]).
true.

?- monotonicallyDecreasing([5]).
true.

?- monotonicallyDecreasing([5, 5, 10]).
false.

?- monotonicallyDecreasing([5, 3, 3, 1]).
true.
```

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

Hint: Refer to the slides for an example of how to represent a list containing at least two elements at the start of the list.

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, 31 March 2025**.

References

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