Computer Science 3MI3 – 2020 homework 2

Solving problems in Prolog

Mark Armstrong

September 18th, 2020

Contents

Introduction

Prolog is a *logical programming* language, in which the task of the programmer is to precisely describe the possible solution space, rather than an algorithm to generate the solution. The language runtime then performs a search to find a solution fitting the description.

Put simply, the programmer describes the problem, not the solution.

Boilerplate

Submission procedures

Submission method

Homework should be submitted to your McMaster CAS Gitlab respository in the `cs3mi3-fall2020` project.

Ensure that you have **pushed** the commits to the remote repository in time for the deadline, and not just committed to your local copy.

Naming requirements

Place all files for the homework inside a folder titled `hn`, where `n` is the number of the homework. So, for homework 1, the use the folder `h1`, for homework 2 the folder `h2`, etc. Ensure you do not capitalise the `h`.

Unless otherwise instructed in the homework questions, place all of your code for the homework in a single file in the `hn` folder named `hn.ext`, where `ext` is the appropriate extension for the language used according to this list:

- For Scala, `ext` is `sc`.
- For Prolog, `ext` is `pl`.
- For Ruby, `ext` is `rb`.
- For Clojure, `ext` is `clg`.

If multiple languages are used in the homework, submit a `hn.ext` file for each language.

If the language supports multiple different file extensions, you must still follow the extension conventions above.

Incorrect naming of files may result in up to a 10% deduction in your grade.

Do not submit testing or diagnostic code

Unless you are instructed to do so in the homework questions, **you should not submit testing code with your homework submission**.

This includes

- any `main` function,
- any `print` statements which output information **that is not directly requested as console output in the homework questions**.

If you do not wish to remove diagnostic print statements manually, you will have to find a way to ensure that they disabled in your final submission.

For instance, by using a wrapper on the print function or macros.

Due date and allowance for technical difficulties

Homework is due on the second Sunday following its release, by the end of the day (midnight). Submissions past 00:00 may not be considered.

If you experience technical difficulties leading up to the submission time, please contact Mark **ASAP** with the details of the problem and, if possible, attach the current state of your homework to the communication. This information will help ensure we are able to accept your submission once the technical difficulties are resolved.

Proper conduct for coursework

Individual work

Unless explicitely stated in the homework questions, all homework in this course is intended to be *individually completed*.

You are welcome to discuss the content of the homework in the public forum of the class Microsoft Teams team homework channel, though obviously solutions or partial solutions should not be posted or described.

Private discussions about the homework cannot reasonably be forbidden, but such discussions should follow the same guidelines as public discussions.

Inappopriate collaboration via private discussions which is later discovered by course staff may be considered academic dishonesty.

When in doubt, make the discussion private, or report its contents to the course staff by making a note of it in your homework.

To clarify what is considered appropriate discussions of homework content, here are some examples:

1. Discussing the language features introduced or needed for the homework.
    - Such as relevant builtin datatypes and datatype definition methods and their general use.
    - Code snippets that are not partial solutions to the homework are welcome and encouraged.
2. Questions of the form "What is meant by `x`?", "Does `x` really mean `y`?" or "Is there a mistake with `x`?"
    - Of course, questions of those form which would be answered by partial solutions are not considered appropriate.
3. Questions or advice about errors that may be encountered.
    - Such as "If you see a `scala.MatchError` you should probably add a catch-all `_` case to your `match` expressions."

Language library resources

Unless explicitely stated in the questions, it is not expected that you will use any language library resources in the homeworks.

Possible exceptions to this rule include implementations of datatypes we discuss in this course, such as lists or options/maybes, if they are included in a standard library instead of being builtin.

Basic operations on such types would also be allowed.

- For instance, `head`, `tail`, `append`, etc. on lists would not require explicit permission to be used.
- More complex operations such as sorting procedures would require permission before you used them.

Part 0.1: Installing Prolog [0 points]

In this course, we will be targetting SWI Prolog version 8.2.0, as used in the swipl Docker image.

If there is any update to the Docker image, or if for any other reason we change our targeted versions, we will make an announcement on the homepage.

That said, any recent version should suffice.

Installation guides

- Linux and MacOS users should be able to install SWI Prolog via their package manager.
- The SWI Prolog website provides downloads of prebuilt binaries for SWI Prolog 8.2.1.

Part 0.2: Basic Prolog programming [0 points]

Some basic tutorial on Scala will be given in an upcoming lecture, and also in the tutorials, and should provide you with the knowledge you need to complete this homework.

Prolog specific instructions

In your Prolog code in this course, you are allowed to use all builtin predicates; in particular, see the following entries of the SWI Prolog documentation:

- Arithmetic and
- Built-in list operations

Musa AlHassy's Prolog cheat sheet (developed in part for this course last year.) is also a useful resource, and includes links to several other resources.

Part 1: Prime checker [5 points]

Here we define a predicate "has divisor less than or equal to" which checks, for a given `X`, if there exists a divisor of `X` less than **or equal to** the given `Y` which is greater than 1. (Edited September 20th: previously the preceding description incorrectly said only "less than".)

```
hasDivisorLessThanOrEqualTo(_,1) :- !, false.
hasDivisorLessThanOrEqualTo(X,Y) :- 0 is X mod Y, !.
hasDivisorLessThanOrEqualTo(X,Y) :- Z is Y - 1,
↪  hasDivisorLessThanOrEqualTo(X,Z).
```

Roughly, this works as follows:

1. If `Y` is 1, end the search (don't backtrack to look for other proofs) and fail (search turns up `false`).
2. Otherwise, if `Y` does divide `X`, end the search immediately (don't backtrack looking for other proofs) (and the search will return `true`).
3. Otherwise, check if there is a divisor of `X` which is less than `Y - 1`.

Using this predicate, define a `isPrime` predicate such that `isPrime(X)` returns true if `X` is a prime number.

Part 2: From number to list of digits (and vice versa) [15 points]

Complete the following definition of predicate that checks if a given list is a list of the digits in a given integer. We have filled in code to fetch the last element of the list as a starting point.

```
isDigitList(_,[]) :- !, false.
isDigitList(X,[X]). % Your code here; change the . to :-
isDigitList(X,L) :-
  last(L,F). % F is the final element of the list
  % Your code here; change the above . to a ,
```

Your definition must enforce that

- the last element of the list (`F`) is the same as the first digit of `X`, and
- the remaining portion of the list matches the remaining digits of `X`.

(Note this predicate can be thought of as checking if the given list contains the digits for the given integer, or as checking if the given integer has the digits contained in a given list; hence, the "vice versa".)

You may use this code to "drop the last element" of the list if you wish (in this part and the rest of the homework.)

```
% dropLast(L1,L2) if L2 is the list L1, leaving off the last
↪  item.
dropLast([_],[]). % The last element is dropped.
dropLast([H|T],[H|T2]) :-
  % Aside from the base case above, the lists must match.
  dropLast(T,T2).
```

Part 3: Palindrome [10 points]

Define a predicate `isPalindrome` which checks that a given list is a palindrome.

Note you can use pattern matching with the list "cons" operator, `|`, to refer to the head and tail of a list, as in

```
palindrome([H|T]) :- ??? % H is the first element, L is the rest
```

Part 4: Prime palindromes [15 points]

Define a predicate `primePalindrome` which checks if a given integer is both

1. prime, and
2. a palindrome (in terms of the list of its digits).

Part 5: Efficiency [10 bonus_points]

For extra marks, research how to make your above definitions more efficient.

In particular, look into the use of the "cut" operator, `!`, to prevent backtracking.

The unit tests will include some (optional for your use) "large" versions of these problems in order to test the efficiency of your solutions.

Testing

:TODO: