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 explicitly 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.

Additionally, the standard *higher-order* operations including `map`, `reduce`, `flatten`, and `filter` are permitted generally, unless the task is to implement such a higher-order operator.

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, with the first element of the list being the "ones" digit, the second element of the list being the "tens" digit, etc.

Updated September 23rd: the given code below has been changed, and the description of the list has been "reversed". Previously, the ones digits would have been the last element. Now, it is the first element.

One further edit was added later; remove the cut in the first case below.

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,[H|T]). % Your code here; change the . to :-
```

(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.) Updated September 23rd: this predicate is unlikely to be necessary for the updated version of this question.

```
% 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, T 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.

Updated September 25th: these test scripts, separate from the provided unit testing, can be used to evaluate the efficiency of your solution. The `time` predicate used will report the number of inferences used and how long the test takes.

(Updated September 26th: I've run the tests again and provided here the number of inferences my sample solution takes. as a better metric.)

On my naive, unoptimised sample solution, the first test makes 222,020,399 inferences (and takes 45 seconds on my machine). The second test makes 1,401,332,191 inferences (and takes 5 minutes on my machine). The last one did not halt before I gave up on it .

```prolog
:- consult(h2).

:- initialization(writeln('Timing prime palindrome')).
:- initialization(time(primePalindrome(111010111))).

:- consult(h2).

:- initialization(writeln('Timing prime palindrome with a
↪   beastly prime palindrome')).
:- initialization(time(primePalindrome(700666007))).

:- consult(h2).

:- initialization(writeln('Timing prime palindrome with
↪   Belphegor\'s prime palindrome')).
:-
↪   initialization(time(primePalindrome(1000000000000066600000000000001))).
```

Testing

As of September 23rd, unit tests for the requested predicates can be found here. The contents of the unit test file are also repeated below.

Important notes regarding this testing:

1. Unlike homework 1, for homework 2 it is *mandatory* that your code pass these tests.

2. There is no interface file that you need to complete. You still only need submit your `h2.pl` file.

3. Passing these tests does not guarantee a particular grade. What has been provided is intended to act as a *sanity check*, and more thorough tests may be used in the grading process.

The tests can be run by placing the `h2.plt` unit test file in the same directory as your `h2.pl`, and running the command

```
swipl -t "load_test_files([]), run_tests." -s h2.pl
```

in your shell in that directory.

Automated testing via Docker

Once you have Docker and Docker Compose (installation guides linked here) installed, you can easily run the unit tests by copying the files

- `Dockerfile`,
- `docker-compose.yml`,
- `setup.sh` and
- `run.sh`

into your `h2` directory where your `h2.pl` file exists, and running `setup.sh` and `run.sh`.

See the README for this testing setup and those files on the course GitHub repo.

Note that the use of the `setup.sh` and `run.sh` scripts assumes that you are in a `bash` like shell; if you are on Windows, and not using WSL or WSL2, you may have to run the commands contained in those scripts manually.

The tests

The contents of the testing script are repeated here.

```
:- begin_tests(h2).
:- include(h2).


% isPrime tests
```

```prolog
test(five_is_prime, nondet) :- isPrime(5).
test(sixty_seven_is_prime, nondet) :- isPrime(67).
% The \+ means "it is not provable that."
test(four_is_not_prime, nondet) :- \+ isPrime(4).
test(one_hundred_is_not_prime, nondet) :- \+ isPrime(100).
% Also test there are no "bad" proofs that primes are not
↪   prime.
test(not_five_is_not_prime, nondet) :- \+ not(isPrime(5)).




% isDigitList tests

test(digit_list_12345, nondet) :-
↪   isDigitList(12345,[5,4,3,2,1]).
test(not_digit_list_12345_grouped, nondet) :- \+
↪   isDigitList(12345,[5,43,2,1]).




% isPalindrome tests

test(empty_palindrome, nondet) :- isPalindrome([]).
test(pair_palindrome, nondet) :- isPalindrome([1,1]).
test(steponnopets_palindrome, nondet) :-
↪   isPalindrome([s,t,e,p,o,n,n,o,p,e,t,s]).
test(palindrome_elements_atomic, nondet) :- \+
↪   isPalindrome([12, 21]).




% primePalindrome tests

test(eleven_prime_palindrome, nondet) :- primePalindrome(11).
test(nine_twenty_nine_prime_palindrome, nondet) :-
↪   primePalindrome(929).
test(not_thirteen_prime_palindrome, nondet) :- \+
↪   primePalindrome(13).
```

```prolog
test(not_twenty_two_prime_palindrome, nondet) :- \+
↪   primePalindrome(22).
```

Sample solution

The source code can be downloaded here.

```prolog
% Part 1, prime checker

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

isPrime(X) :- X < 2, !, false.
isPrime(2) :- !.
isPrime(X) :- not(hasDivisorLessThanOrEqualTo(X,X-1)).



% Part 2, number and lists of their digits.

% Note that the list may have trailing zeroes
% by this definition.
% This is not forbidden by the question.
isDigitList(_,[]) :- false.
isDigitList(X,[X]) :- member(X,[1,2,3,4,5,6,7,8,9,0]).
% I include some more documentation here to explain
% how this can solve the problem "both ways";
% that is, it will work no matter which argument is unknown.
isDigitList(X,[H|T]) :-
  isDigitList(Y,T),
  % Using member to restrict the value of H avoids
  % uninstantiated errors we would get if we tried to use
  % less than, etc., while H is unknown.
  member(H,[1,2,3,4,5,6,7,8,9,0]),
  % This "is" statement is carefully put together
  % so that only one argument is ever unknown.
  % This also prevent uninstantiated errors.
  X is Y * 10 + H.
```

```prolog
% Part 3, palindromic lists

dropLast([_],[]). % The last element is dropped.
dropLast([H|T],[H|T2]) :-
  % Aside from the base case above, the lists must match.
  dropLast(T,T2).

isPalindrome([]). % An empty list is a palindrome.
isPalindrome([_]). % Any singleton is a palindrome.
isPalindrome([H|T]) :-
  last(T,H), % H must also be the last element of T.
  dropLast(T,TN),
  isPalindrome(TN). % check the remainding elements.




% Part 4, Prime palindromes

% Just put together our predicates from above.
primePalindrome(X) :-
  isPrime(X),
  isDigitList(X,Digits),
  isPalindrome(Digits).
```