Computer Science 3MI3 – 2020 homework 3

Revisiting homework 1 in Prolog and homework 2 in Scala

Mark Armstrong

September 25th, 2020

Contents

Introduction

We have now worked in Scala for one homework, representing different kinds of trees and defining operations on them.

We have also worked in Prolog solving problems about palindromes and prime numbers.

For our third homework, we gain further experience in both languages by tackling our previous problems in the other language. Since they are from different paradigms, although the problems are the same, your solutions will out of necessity be quite different.

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, 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.

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 1: Flattening and ordering trees in Prolog [20 points]

Using the representations (not technically types) of `BinTree` and `LeafTree` trees discussed in the "Trees in Prolog" lecture (links below), write predicates `flatten` and `elemsOrdered` in Prolog.

- (Each predicate is assigned 7.5 points.)

Each should be a binary predicate, taking a tree and a list as parameters. So `flatten(T,L)` means that `L` is the result of flattening `T`, and similarly `elemsOrdered(T,L)` means that `L` is a list of elements of `T` in order.

Implement your own sorting predicate to use in `elemsOrdered`; do not use a predefined one.

- (The sorting predicate is assigned 5 points.)

Refer to the "Trees in Prolog" lecture notes, available as

- HTML,
- PDF, or
- plaintext Org source,

Part 2: Predicates in Scala [20 points]

In Scala and other non-logical languages, we can represent predicates as boolean-valued functions/methods (functions/methods which return booleans). Note that in Scala the type of booleans is `Boolean` (not `Bool`, as it is in some languages.)

In Scala, implement the predicates from homework 2:

1. `isPrime` on the `Int` type.
    - (assigned 5 points.)
2. `isPalindrome` on the type `List[A]` for any `A`.
    - (assigned 5 points.)
3. `primePalindrome` on the type `Int`.
    - (assigned 5 points.)

In the process, also define a method `digitList` (not `isDigitList`) which converts an `Int` into a `List[Int]` consisting of its digits, with the first element of the list being the "ones" digits, the second element being the "tens" digit, etc.

- (this method is assigned 5 points.)

Testing

As of September 28th, unit tests for the requested predicates are available through the following links.

- Scala unit tests: h3t.sc
- Prolog unit tests: h3.plt.

The contents of the unit test file are also repeated below.

The tests can be run by placing the `h3t.sc` and `h3.plt` files in the same directory as your `h3.sc` and `h3.pl` files, and running the following commands.

```
amm h3t.sc
```

and

```
swipl -t "load_test_files([]), run_tests." -s h3.pl
```

to test your Prolog code.

If you use implementations other than Ammonite and SWI Prolog, you may need to adjust the tests to be able to run them on your system.

In particular, those using IntelliJ for writing Scala may need to remove the import statement at the top of that file, or move the tests to their source file (if so, be sure to remove them before submission!)

Automated testing via Docker

(Updated October 1st.)

The Docker setup and usage scripts are available at the following links.

- Dockerfile
- docker-compose.yml
- setup.sh
- run.sh

Place them into your `h3` directory where your `h3.sc` and `h3.pl` files and the `h3.plt` and `h3t.sc` (linked to above) files exist, then run `setup.sh` and `run.sh`.

You may also refer to 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 scripts are repeated here.

The Scala tests

```
import $file.h3, h3._

/* Given an expected result and a computed result,
   check if they are equal in value.
   If so, return 0. Otherwise, inform the user, and return 1,
   so the number of failures can be counted. */
```

```
def test[A](given: A, expected: A, the_test: String) =
  if (!(given equals expected)) {

    println("+-------------------------------------------------")
    println("| " + the_test + " failed.")
    println("| Expected " + expected + ", got " + given + ".")

    println("+-------------------------------------------------")
    1
  } else {
    0
  }

// The tests are saved as tuples, the pieces of which will be passed
// to test.
val tests = List(
  (isPrime(2), true, "Two is prime"),
  (isPrime(53), true, "Fifty three is prime"),
  (isPrime(8), false, "Eight is not prime"),
  (isPrime(63), false, "Sixty-three is not prime"),
  (isPalindrome(List()), true, "Empty palindrome"),
  (isPalindrome(List(1,1)), true, "Pair palindrome"),

  (isPalindrome(List('s','t','e','p','o','n','n','o','p','e','t','s')),
  true, "step on no pets palindrome"),
  (isPalindrome(List(12,21)), false, "Palindrome elements are atomic"),
  (primePalindrome(11), true, "Eleven is prime palindrome"),
  (primePalindrome(929), true, "Nine twenty nine is prime palindrome"),
  (primePalindrome(13), false, "Thirteen is not prime palindrome"),
  (primePalindrome(22), false, "Twenty two is not prime palindrome"),
  )

// Apply test to each element of tests, and sum the return values.
// This is essentially a for loop.
```

```
val failed = tests.foldLeft(0) {
  (failures, next) => next match {
    // Deconstruct the tuple to get its parts
    case (given, expected, the_test) => failures + test(given,
    ↪  expected, the_test)
  }
}

println("+-------------------------------------------------")
println("| " + failed + " tests failed")
println("+-------------------------------------------------")
```

The Prolog tests

```
:- begin_tests(h3).
:- include(h3).

% One way to "name" values is to use predictes such as these.
% Then a variable can be bound to the values here using the
↪  predicate.
largeLeafTree(T) :-
    T = branch(
         branch(
          branch(leaf(10),leaf(-2)),
          branch(leaf(3),leaf(400))),
         branch(
          branch(leaf(55),leaf(6)),
          branch(leaf(777),leaf(88888888)))).
largeLeafTreeFlat(L) :-    L =
↪  [10,-2,3,400,55,6,777,88888888].
largeLeafTreeOrdered(L) :- L =
↪  [-2,3,6,10,55,400,777,88888888].

largeBinTree(T) :-
    T = node(
         node(
          node(empty,1,empty),
          22,
          node(empty,-3,empty)),
         404,
```

```
          node(
           node(empty,50,node(empty,6,empty)),
           7777777,
           node(node(empty,-88,empty),9001,empty))).
largeBinTreeFlat(L) :-    L =
↪  [1,22,-3,404,50,6,7777777,-88,9001].
largeBinTreeOrdered(L) :- L =
↪  [-88,-3,1,6,22,50,404,9001,7777777].


% The tests

test(flatten_leaf, nondet) :- flatten(leaf(5),[5]).
test(flatten_branch, nondet) :-
↪  flatten(branch(leaf(1),leaf(2)),[1,2]).
test(flatten_large_leaftree, nondet) :- largeLeafTree(X),
↪  largeLeafTreeFlat(L), flatten(X,L).

test(order_leaf, nondet) :- elemsOrdered(leaf(5),[5]).
test(order_branch, nondet) :-
↪  elemsOrdered(branch(leaf(1),leaf(2)),[1,2]).
test(order_large_leaftree, nondet) :- largeLeafTree(X),
↪  largeLeafTreeOrdered(L), elemsOrdered(X,L).

test(flatten_empty, nondet) :- flatten(empty,[]).
test(flatten_node, nondet) :-
↪  flatten(node(empty,5,empty),[5]).
test(flatten_large_bintree, nondet) :- largeBinTree(X),
↪  largeBinTreeFlat(L), flatten(X,L).

test(order_empty, nondet) :- elemsOrdered(empty,[]).
test(order_node, nondet) :-
↪  elemsOrdered(node(empty,5,empty),[5]).
test(order_large_bintree, nondet) :- largeBinTree(X),
↪  largeBinTreeOrdered(L), elemsOrdered(X,L).
```

Sample solution

The source code can be downloaded

- for the Scala portion, here, and
- for the Prolog portion, here.

It is also shown below.

```
// An imperative-style prime check.
def isPrime(n: Int): Boolean = {
  if (n < 2) return false

  // For efficiency, we can check up to only sqrt of n.
  val sqrt_n = (math.sqrt(n.toDouble)).toInt

  for (i <- 2 to sqrt_n if n % i == 0) return false

  return true
}

def isPalindrome[A](xs: List[A]): Boolean = xs match {
  case Nil => true
  case _ :: Nil => true
  case (x :: xs) => x == xs.last &&
  ↪  isPalindrome(xs.dropRight(1))
}

def digitList(n: Int): List[Int] = {
  // Treat negatives as if they were positive.
  val abs_n = n.abs

  if(0 <= abs_n && abs_n <= 9)
    List(abs_n)
  else
    digitList(abs_n / 10) ++ List(abs_n % 10)
}

def isPrimePalindrome(n: Int): Boolean = isPrime(n) &&
↪  isPalindrome(digitList(n))

flatten(leaf(X),[X]).
flatten(branch(L,R),Xs) :- flatten(L,Ls), flatten(R,Rs),
↪  append(Ls,Rs,Xs).
```

```
flatten(empty,[]).
flatten(node(L,A,R),Xs) :- flatten(L,Ls), flatten(R,Rs),
↪  append(Ls,[A|Rs],Xs).

isSorted([]).
isSorted([_]).
isSorted([H,N|T]) :- H =< N, isSorted([N|T]).

bubble([],[]).
bubble([X],[X]).
bubble([X1,X2|Xs],[X1|Ys]) :- X1 =< X2, bubble([X2|Xs],Ys).
bubble([X1,X2|Xs],[X2|Ys]) :- X1 > X2, bubble([X1|Xs],Ys).

bubbleSort(X,X) :- isSorted(X), !.
% Note: this is very inefficient, but hopefully easy to
↪  understand.
bubbleSort(X,Y) :- bubble(X,Z), bubbleSort(Z,Y).

orderedElems(leaf(X),[X]).
orderedElems(branch(L,R),X) :- flatten(branch(L,R),Y),
↪  bubbleSort(Y,X).

orderedElems(empty,[]).
orderedElems(node(L,A,R),X) :- flatten(node(L,A,R),Y),
↪  bubbleSort(Y,X).
```