Homework 4
COMP 302 Programming Languages and Paradigms

Francisco Ferreira and Brigitte Pientka
McGill University: School of Computer Science

Due Date: 17 November 2017

Your homework is due at the beginning of class on Nov 17, 2017. All code files must be submitted electronically using my courses. **Your program must compile.** Solutions should be correct (i.e. produce the correct result), be elegant and compact, and take advantage of OCaml's pattern matching. Please consult the style guides posted on the course website to get more information regarding what constitutes good style.

Q1. A Rose by any Other Name Would Smell as Sweet (30pts)

As any avid gardener will know, rose trees have very large branching factors. In this question we will consider such trees (minus the flowers).

We explore backtracking through an n-ary tree using exceptions and using continuations. We define an n-ary tree as follows:

```
type 'a rose_tree = Node of 'a * ('a rose_tree) list
```

Q1.1 (10 points) Implement a function `find:('a -> bool)-> 'a rose_tree -> 'a option` using auxiliary functions which uses the exception `Backtrack` to backtrack through the tree.

Q1.2 (10 points) Implement a function `find' :('a -> bool)-> 'a rose_tree -> 'a option` using auxiliary functions which uses a continuation of type `unit -> 'a option` to backtrack through the tree.

Q1.3 (10 points) Implement a function `find_all : ('a -> bool)-> 'a rose_tree -> 'a list` using auxiliary functions which uses a success continuation of type `'a list -> 'b` to collect all the nodes that satisfy the predicate in the tree.

Q2. Rational Numbers Two Ways (20pts)

Floating point numbers can be considered as a computer approximation of rational numbers. In many situations they are exactly what you need. However, certain applications require other representations for rational numbers. For example, Donald Knuth uses fixed

point arithmetic for the TEX typesetting system (that we used to generate this document for example). In this question we will explore modules, and in particular how to abstract the representation or rational number and its arithmetic that we use (spoiler alert: we will use functors to do this).

First, we define a representation of rationals using fractions:

```
type fraction = int * int
```

Second, we define a module type to perform arithmetic, that may use different number representations.

```
module type Arith =
  sig
    type t
    val epsilon : t (* A suitable tiny value, like epsilon_float for floats *)

    val plus : t -> t -> t (* Addition *)
    val minus : t -> t -> t (* Substraction *)
    val prod : t -> t -> t (* Multiplication *)
    val div : t -> t -> t (* Division *)
    val abs : t -> t (* Absolute value *)
    val lt : t -> t -> bool (* < *)
    val le : t -> t -> bool (* <= *)
    val gt : t -> t -> bool (* > *)
    val ge : t -> t -> bool (* >= *)
    val eq : t -> t -> bool (* = *)
    val from_fraction : fraction -> t (* conversion from a fraction type *)
    val to_string : t -> string (* generate a string *)
  end
```

A trivial example to implement is the module that uses floats internally to implement these operations.

```
module FloatArith : Arith =
struct
  type t = float
  ... (* check hw4.ml for the complete implementation *)
end
```

Q2.1 (10 points) Implement a module named: `FractionArith` of type `Arith` that uses a value of type `fraction` as its internal representation. Note 1/1000000 is a suitable tiny number for this case.

Rational approximations of real valued functions are a very important problem in numerical computing. One example is the Newton-Raphson method. This method can be used to find the roots of a function; in particular, it can be used for computing the square root of a given integer. Given a good initial approximation, it converges rapidly and is highly effective for computing square roots, solving the equation

$$a - x^2 = 0$$

To compute the square root of a, choose any positive `x0`, say `1`, as the first approximation. If `x` is the current approximation then the next approximation `next` is

$$(a/x + x)/2$$

Stop as soon as the difference becomes too small.

We can implement solutions of this problem using a function that is parametrized by modules of type `Arith`. The type of a module that uses Newton-Raphson's method is:

```
module type NewtonSolver =
  sig
    type t

    val square_root : t -> t
  end
```

To compute the square root of a, implement a function `findroot x acc` where `x` approximates the square root of a with accuracy `acc`, i.e. the absolute difference between the current approximation and the next approximation is smaller than `acc`. We use `epsilon` as the desired accuracy.

Your function `square_root` should have type `t -> t`. Note that we made `findroot` a local function to be defined inside the function `square_root`.

Q2.2 (10 points) Implement a functor named: `Newton` that takes a module of type `Arith` to produce a module of type `NewtonSolver`. This module should allow us to approximate the square root function using two internal representations (floats and fractions) with a great deal of code reuse.

Q3. Real Real Numbers, for Real! (50pts)

In question 2 we used Newton-Raphson's method to compute approximate rational answers of real valued functions. In this question we will explore the first steps of having a representation of real numbers (with their infinite number of decimals) inside our computer. The definitions of this question are based on a paper by David Lester [1], the paper contains how to extend this question to real arithmetic and some transcendental functions using continued fractions. If you feel like implementing a calculator with real numbers that can be approximated to any precision, you can always check the algorithms in the paper, it could be fun!

Real numbers can be represented in many ways, Cauchy sequences or Dedekind cuts for example. In this question we explore the representation of reals as continued functions. We can represent a real number n as a continued function like this:

$$r = x_0 + \cfrac{1}{x_1 + \cfrac{1}{x_2 + \cfrac{1}{x_3 + \cdots}}}$$

Where x_n are integers, moreover all x_n for $n >= 1$ are positive (the first integer may be negative). Real numbers may be infinitely long, while rationals are finite in length.

Because this is an infinite sequence, we will use laziness to represent a real number as a stream of integers that represent all the x_n. In a very precise sense, constructive real numbers are represented by a lazy computer program. Therefore, we represent a real number r as the stream of integers: $r = [x_0, x_1, \cdots, x_n, \cdots]$.

In our implementation we will change some details. In order to use an infinite sequence, if the number we are representing has a finite representation, we will just continue the stream with zeros. Be mindful of this when implementing your answers.

Using Euler's Q-polynomials we can define the following expansion, for the approximation with the first n terms:

$$r_n = x_0 + \frac{1}{q_0 q_1} + \frac{-1}{q_1 q_2} + \cdots + \frac{-1^{(n-1)}}{q_{n-1} q_n}$$

where:

$$q_0 = 1$$
$$q_1 = x_1$$
$$q_{n+2} = (x_{n+2} q_{n+1}) + q_n$$

The functions r_n and q_n allow us to get rational expansions, for simplicity we will use floating point values to store our approximations.

Q3.1 (5 points) Implement the function `q : int stream -> int -> int` Using the definition of q above.

Q3.2 (5 points) Implement the function `r: int stream -> int -> float` Using the definition r above.

So the function r allows us to get the n term approximation of a real number, but it does not say how precise it is (i.e.: different streams will converge at different speeds). The error of an approximation is defined as:

$$|r - r_n| < \frac{1}{q_n(q_n + q_{n-1})}$$

Q3.3 (5 points) Implement the function `error: int stream -> int -> float` Using the definition above. This function takes the stream, the number n of terms in the approximation and returns a bound for the error.

Q3.4 (15 points) Implement the function **val** `rat_of_real : int stream -> float -> float` that takes a string and an error bound and computes a rational approximation of the real number using the functions `r`, `q` and `error`. Because we are getting an approximation, our function will always inspect finitely many elements of the stream.

The last step is to convert floating point numbers (that is rationals as we explained above) into reals. To calculate a continued fraction representation of a number, write down the integer part using `floor` of r (this will become the head of the stream). Subtract this integer part from r. If the difference is 0 (or less than `epsilon_float`), the stream will continue with 0s from now on; otherwise find the reciprocal of the difference and repeat to compute the tail of the stream.

Q3.5 (20 points) Implement the function `real_of_rat : float -> int stream` that computes the real number representation of floating point value.

References

[1] David R. Lester. Vuillemin's exact real arithmetic. In *Functional Programming, Glasgow 1991: Proceedings of the 1991 Glasgow Workshop on Functional Programming, Portree, Isle of Skye, 12–14 August 1991*, pages 225–238, London, 1992. Springer London.