Homework #2

Theory

In the following questions, you can freely make use of λ-terms defined in the lectures.

For example, you may write TRUE, SECOND or NOT without repeating their definition, and you do not have to spell out all these terms in your answers.

Question 1 *Numbers and lists* (2 points)

(a) Write a λ-function EVEN? that takes as input a Church numeral n, and reduces to TRUE if n is even, and FALSE otherwise. For example, EVEN? c_5 should reduce to FALSE.

(b) In the lecture it was discussed how lists can be represented in the λ-calculus. Write a λ expression that corresponds to a list containing the value FALSE three times in a row.

(c) In the lecture we represented numbers using Church numerals, but we also considered representing numbers in the λ-calculus as a list containing a number of copies of the value FALSE. We will call this a "list numeral". Write a function CHURCHTOLIST that takes a Church numeral n as input, and outputs the corresponding list numeral.

Question 2 *Negative numbers* (3 points)

We have seen how to do arithmetic with the nonnegative integers, either using Church numerals or list numerals. With that as a starting point, how could you represent *any* integer, including negative ones? And how would you adapt the basic operations to work with your new scheme? *Hint: you may want to take inspiration from how negative integers are formally defined in mathematics. Wikipedia can help.*

(a) Provide a simple, one line definition specifying how an integer should be represented as a λ-expression.

(b) Give a λ-expression for NZERO?: the function that yields TRUE if a give integer is zero and FALSE otherwise.

(c) Give a λ-expression for NSUCC: the function that takes an integer and returns the successor of that integer.

(d) Give a λ-expression for NPRED: the function that takes an integer and returns the predecessor of that integer.

Question 3 *Y combinator* (3 points)

(a) Write a function SUM that takes a list of Church numerals as input, and outputs their sum.

(b) Write a function COUNTDOWN that takes a Church numeral n as input, and produces a list containing n, $n-1$, ..., 0.

(c) Write a function LESSTHAN? that takes two Church numerals n and m as inputs; it should reduce to TRUE if $n < m$, and FALSE otherwise.

(d) Write a function FILTER that takes two inputs, g and l. The second argument l is a list; the first argument g is a function that takes any list element as input and produces a boolean that identifies whether the list element satisfies some criterion. The output should be the list containing all elements from the list l for which the function g resulted in the value TRUE. For example, FILTER (λx.NOT (EVEN? x)) (CONS c_2 (CONS c_3 (CONS c_4 NIL))) should reduce to the list CONS c_3 NIL. Tip: to avoid confusion, write the true-case and the false-case of conditionals on separate lines.

Question 4 *Recurrences* (2 points)

The Y combinator satisfies the recurrence $YF = F(YF)$. This question asks you to find similar recurrences. You may use Y in your answers, or you can come up with an answer directly. Note: this can be a bit of a puzzle. It's important to try to do this for yourself, so take a reasonable amount of time (say, at least half an hour per item) to experiment with different expressions before asking the TA for hints if you don't see the answer immediately.

(a) An "eater" is a function F such that $Fx = F$, for all x. Show such a function.

(b) A "flipper" is a function F such that $Fx = xF$, for all x. Show such a function.

Practice

In this homework we use an explicit Prelude. This means you may only use the functions mentioned below. Do not use any other Haskell libraries and do not import anything else. We also tell GHC and hlint to ignore certain warnings and hints, respectively.

```
{-# LANGUAGE RankNTypes #-}
{-# OPTIONS_GHC -Wall -Wno-unused-imports #-}
```

```
import Prelude (Integer,Bool(False,True),(-),(+),(==),undefined,error,head,null,tail)
{-# ANN module "HLint: ignore Redundant lambda" #-}
{-# ANN module "HLint: ignore Avoid lambda" #-}
{-# ANN module "HLint: ignore Use foldl" #-}
{-# ANN module "HLint: ignore Use foldr" #-}
{-# ANN module "HLint: ignore Use list literal" #-}
{-# ANN module "HLint: ignore Eta reduce" #-}
```

Question 5 *Church numerals* (3 points)

This exercise is about defining the Church numerals in Haskell. Unfortunately standard Haskell does not allow us to define the type `Church` without any type parameter. Hence above we activated the language extension `ImpredicativeTypes` so that we can do the following.

```
type Church = (forall a . (a -> a) -> (a -> a))
```

Note that we cannot `show` something of type `Church` because Haskell has no way of printing functions. Similarly, we also cannot use `==` or `compare` on our `Church` type. (Later we will see how to teach Haskell which things are equal, using the `Eq` type class.)

Define conversion functions to make a Church numeral, and to convert it back to an `Integer`:

```haskell
toChurch :: Integer -> Church
toChurch 0 = undefined
toChurch n = undefined

toInteger :: Church -> Integer
toInteger cn = undefined -- hint: use the (+1)  function
```

Define addition, multiplication and exponentiation for Church numerals. Note: You should not use `toInteger` or `toChurch` here! Also no that `pow` should take the exponent as second argument, so for example `five `times` five `times` five` should be the same as `five `pow` three`.

```haskell
plus :: Church -> Church -> Church
plus cn cm = undefined

times :: Church -> Church -> Church
times cn cm = undefined

pow :: Church -> Church -> Church
pow cn cm =  undefined
```

Finally, give three example queries that you can use to test whether your definitions are correct and work as expected. For example, how would you check that $3*7 = 7+7+7$ in Church numerals? Hint: you may use `toInteger` here.

```haskell
tests :: [Bool]
tests =
  [ undefined
  , undefined
  , undefined
  ]
```

Question 6 *Fixpoints* (3.5 points)

Haskell's type system does not allow the Y combinator as introduced in the lecture. However, it is easy to define an equivalent fixpoint operator in Haskell that actually looks a lot simpler, and that does work. It can be defined like this:

```haskell
fix :: (a -> a) -> a
fix f = x where x = f  x
```

Using this fixpoint operator, and using Haskell numbers and lists, we'll define functions in Haskell that are equivalent to the recursive functions that we've seen in the lecture.

(a) Write a function `countdown1` using `fix`. The function should take a nonnegative `Integer` and count down to zero. For example, `countdown1 5` should yield `[5, 4, 3, 2, 1, 0]`.

(b) Now make a second version `countdown2` that does not use the fixpoint operator. Instead, write the base case and the recursive case on two lines as two separate definitions for your function, where you use pattern matching to identify the appropriate definition. Use the fact that Haskell functions are allowed to be defined recursively.

(c) Now write a function `sumlist1` that should take a list of Integers, and outputs the sum of these Integers, using the fixpoint operator `fix`.

(d) As before, write a second version `sumlist2` that uses pattern matching and direct recursion instead of the fixpoint operator.

(e) Finally, write a function `lessthan` that takes two nonnegative integers, and outputs `True` if the first is less than the second, and `False` otherwise. Use pattern matching and a recursive definition.

```
-- (a)
countdown1 :: Integer -> [Integer]
countdown1 = undefined

-- (b)
countdown2 :: Integer -> [Integer]
countdown2 = undefined

-- (c)
sumlist1 :: [Integer] -> Integer
sumlist1 = undefined

-- (d)
sumlist2 :: [Integer] -> Integer
sumlist2 = undefined

-- (e)
lessthan :: Integer -> Integer -> Bool
lessthan =  undefined
```

Question 7 *Recursion practice* (3.5 points)

For this question, remember to not use anything not in the explicit Prelude above.

Moreover, do not use the built-in interval syntax `[0..7]`.

(a) Write a recursive function `addEnd` that takes as inputs a value `x` of any type `a`, and a list `lst` of type `[a]`, and outputs the same list except with `x` appended at the end. For example, `addEnd 3 [1,2,5]` should yield `[1,2,5,3]`.

(b) Earlier you've written a function `countdown2`. Of course it is also possible to count *up*. Write a function `countup1` that takes a nonnegative integer n and outputs the list `[0, 1, …, n]`. Use the function `addEnd`.

(c) Now implement `countup2` that does the same as `countup1`, but in a different way: instead of using `addEnd` it uses a different helper function `countfromto` that takes *two* integers `n` and `m` as inputs, with $0 \leq$ `n` $\leq$ `m`, and outputs the list `[n, n+1, n+2, …, m]`. Add a comment line stating which implementation of counting up is more efficient, and explain why.

(d) Now, considering what you have just seen, write a function `rev` that takes a list of type $[a]$ and outputs that list in reverse order, again without using anything from the prelude. Your function may use a (non-global) helper function, but it should work in linear time, that is, the number of β-reductions required by `rev` applied to a list of length n should be proportional to n. Do *not* use `addEnd` here.

```
-- (a)
addEnd :: a -> [a] -> [a]
addEnd = undefined

-- (b)
countup1 :: Integer -> [Integer]
```

```haskell
countup1 = undefined

-- (c)
countup2 :: Integer -> [Integer]
countup2 = undefined where
  countfromto = undefined

-- (d)
rev :: [a] -> [a]
rev =  undefined
```