Assignment 3

CS 4400 / CS 5400 Programming Languages

General

Due: Friday, November 1, 11:59pm.

Instructions:

1. Submissions are handled via the Khoury Handin server: https://handins.ccs.neu.edu/
2. Download the assignment pack from https://vesely.io/teaching/CS4400f19/m/cw/03/Assignment3.zip.
3. You can choose to complete this assignment as a pair. If you work as a pair, submit as a pair. Completing an assignment with a partner but submitting individually is considered cheating.
4. Complete the information in `Assignment3.hs` and submit all `.hs` files included in the original pack. You should only need to modify `Church.hs` and `ABLF.hs`.
5. You are free to add top-level definitions, but add them to the bottom of each file, below the separator line.
6. Each top-level function must include a type signature, followed by one or more defining equations.
7. Make sure your file loads into GHCi or can be compiled by GHC without any errors.
8. Use the provided `SimpleTests` module to write your own tests in the `tests` function. Some exercises state properties that your implementation should satisfy. You can either write test cases for these manually, or you can use Haskell's QuickCheck library for property-based testing. I will give an overview of QuickCheck in an upcoming lecture.

Grade: To calculate your grade, we will take the following into account:

a) Quality of submission: Does your code compile without errors? Did you follow the above steps?
b) Correctness: How well does it implement the specification?
c) QA: How well did you test your code?
d) Is your code readable?

Pure Lambda Calculus as a Core Language

As explained in the lecture, pure lambda calculus can is Turing complete and can thus be used to implement features of much richer languages. This assignment explores this concept. In the first few exercises, you will be expected to implement Church encodings for booleans and natural numbers, together with a fixpoint combinator. This will provide scaffolding for translating a more complex language into pure lambda calculus, which is explored in Exercise 7. The relevant lecture notes are here.

The assignment pack contains the following:

- `Assignment3.hs`, containing meta information and a main function for running all tests. This file can be compiled to produce an executable `Assignment3` which will run all tests defined in the `Church` and `ABLF` modules.
- `Lambda.hs`, the syntax for `Lambda`, together with a pretty-printing function `showLambda` (for easier debugging).
- `Reduce.hs`, a module implementing normal-order reduction and normalization. I recommend taking a look here, although you do not have to do anything in this file. It provides stepping functions that may be helpful when debugging solutions (`stepNormal` and `step`).
- `Church.hs`, a module where you will complete definitions for Church encodings-related functions and for the fixpoint combinator.
- `ABLF.hs`, which contains the abstract syntax for ABLF and the skeleton for a translation function from ABLF to `Lambda`.
- `SimpleTests.hs` – same as in Assignment 2

Church encodings

Exercise 1 Implement a function converting Haskell booleans to Church-encoded booleans and its reverse, a function for converting a Church boolean (in normal form) into its Haskell counterpart:

```
toChurchBool :: Bool -> Lambda
fromChurchBool :: Lambda -> Maybe Bool
```

The function `fromChurchBool` should assume that the `Lambda` term is normalized, that is, the term will be in the canonical form presented in the lecture (or the notes) for true or false. However, due to alpha-equivalence, do not assume anything about the names of bound variables. For any `Lambda` term that is not a canonical Church boolean, the function should return `Nothing`.

The two conversion functions should satisfy the following property, for all `b`

```
fromChurchBool (toChurchBool b) == Just b
```

Exercise 2 Implement a function converting an integer ≥ 0 to a Church numeral and its reverse, a conversion from a Church numeral to an integer.

```
toNumeral :: Integer -> Lambda
fromNumeral :: Lambda -> Maybe Integer
```

The function `fromNumeral` should assume that the `Lambda` term is normalized, that is, the Church numeral will be in the canonical form presented in the lecture (or the notes). However, do not assume anything about the names of bound variables. For any `Lambda` term that is not a canonical Church numeral, the function should return `Nothing`.

These operations should satisfy the following, for all `i` ≥ 0:

```
fromNumeral (toNumeral i) == Just i
```

Exercise 3 Complete the following with the `Lamabda`-terms implementing operations on Church-encodings of naturals and booleans. The terms for successor (`csucc`) and predecessor (`cpred`) are predefined for you.

```
-- operations on numerals
cplus :: Lambda    -- addition
cminus :: Lambda   -- subtraction
ctimes :: Lambda   -- multiplication

-- operations on Church booleans
cand :: Lambda
cor :: Lambda
cnot :: Lambda

-- operations on numerals returning Church booleans
ciszero :: Lambda  -- is numeral equal to zero?
cleq :: Lambda     -- less or equal
ceq :: Lambda      -- equal

-- conditional expression
cifthen :: Lambda
```

Exercise 4 `Reduce.hs` contains an implementation of normal reduction and provides the function

```
normalize :: Lambda -> Lambda           -- fully normalize a term
```

Use `normalize` to test operations from Exercises 3 by comparing them to their Haskell counterparts. For example, `plus` should satisfy the following:

```
fromNumeral (normalize (App (App cplus (toNumeral n)) (toNumeral m)))
  == Just (n + m)
```

You can either write unit tests manually, using the `test` function from `SimpleTests`, or you can use QuickCheck to state properties and have them verified. I will give an overview of QuickCheck in the next lecture.

Exercise 5 Write the term `fix`, expressing the Y combinator.

```
fix :: Lambda
```

Translating into Pure Lambda Calculus

`ABLF.hs` contains the abstract syntax definition for a language with booleans, arithmetic expressions, let bindings and recursive definitions of functions with multiple arguments. You will be asked to translate this language into pure lambda calculus. The following sections five a brief overview of the intended semantics of this language. Note, that you are not expected to implement an evaluator for this language, just its translation to pure lambda calculus.

Arithmetic Operations

First few constructs are addition, subtraction, and multiplication on natural numbers. It is assumed that the constructor `Num` will only ever be applied to positive integers or zero. Subtraction is assumed to be a total operation: `Sub (Num 0) (Num m)` should evaluate to `Num 0` for any `m`.

```
data ABLFExpr = AVar Variable
              | Num Integer
              | Add ABLFExpr ABLFExpr
              | Sub ABLFExpr ABLFExpr
              | Mul ABLFExpr ABLFExpr
```

Boolean Operations

Constructs for boolean operations behave as usual. `And` should evaluate to true only if both its arguments evaluate to true, otherwise it evaluates to false. `Or` should evaluate to true if and only if at least one of its arguments evaluates to true. `Not` evaluates to false if its argument evaluates to true and vice-versa.

```
               | Bool Boolean
               | And ABLFExpr ABLFExpr
               | Or ABLFExpr ABLFExpr
               | Not ABLFExpr
```

Comparison of Natural Numbers

`Leq e1 e2` should evaluate to true if `e1` evaluates to a natural number that is less or equal to the natural number resulting from evaluating `e2`. It evaluates to false otherwise. `Eq e1 e2` evaluates to true if `e1` and `e2` evaluate to the same natural number.

```
               | Leq ABLFExpr ABLFExpr
               | Eq ABLFExpr ABLFExpr
```

Conditional Expression

The expression `IfThen e1 e2 e3` evaluates `e2` if `e1` evaluates to true, otherwise it evaluates `e3`.

```
               | IfThen ABLFExpr ABLFExpr ABLFExpr
```

Let Bindings

`Let x e1 e2` then evaluates `e2` with the variable `x` replaced by `e1`.

```
               | Let Variable ABLFExpr ABLFExpr
```

Function Definitions

ABLF allows defining recursive functions, with the given formal arguments, which can be called from within their own body. `LetFun "f" ["x", "y", "z"] e1 e2` corresponds to

```haskell
let f x y z = e1
in e2
```

in Haskell, or

```scheme
(letrec ((f (lambda (x y z) e1)))
  e2)
```

in Scheme.

```
               | LetRecFun Variable [Variable] ABLFExpr ABLFExpr
```

Function Invocations

Functions are invoked using their name and providing a list of arguments: `Call "f" [e1, e2, e3]` corresponds to `f e1 e2 e3` in Haskell, or `f(e1, e2, e3)` in Python.

```
               | Call Variable [ABLFExpr]
```

Exercise 6 In `ABLF.hs`, Implement the translation function

```
translate :: ABLFExpr -> Lambda
```

translating from the `ABLF` language to `Lambda`, using Church encodings for numbers and booleans. Operations on booleans and numerals, the comparison operators, and `IfThen` should be straightforward translations, using the functions and terms you have implemented in previous exercises. `Let` should follow the expansion given in the lecture.

Translating `LetFun` and `Call` involves two concepts:

- defining and applying abstractions with multiple arguments (currying), and
- (for `LetFun`) recursion using the fixpoint combinator.

Exercise 7 Define

```
factorialOf :: Integer -> ABLFExpr
```

`factorialOf n` should output an ABLF program which:

- defines a factorial function using `LetFun`
- calls the factorial function using `Call` with `n` as the argument

Test this function against a Haskell definition of factorial using the pipeline:

> `factorialOf n` → `translate` → `normalize` → `fromNumeral`

I recommend starting with numerals corresponding to 1, 2 and 3, or you might end up waiting a very long time wondering if the computation terminates.

The function `normalizeWithCount` from the `Reduce` module, behaves as normalize, except it returns a pair containing the normalized term together with the number of reduction steps it needed. In `Assignment3.hs`, fill in how many reduction steps were needed for factorials of 1, 2, 3, 4, and 5. Warning: normalizing `factorialOf 5` will take a *VERY long* time, so have something to do in the meantime and only try it once you are sure everything else works.

Exercise 8 (required for CS5400, optional for CS4400) In `Church.hs`, write a `Lambda`-term implementing exponentiation. That is, `expn n m` should correspond to n^m.

```
expn :: Lambda    -- exponentiation
```

In `ABLF.hs`, extend the `ABLFExpr` definition with an exponentiation construct

```
               | Exp ABLFExpr ABLFExpr
```

and extend `translate` with a corresponding translation case. Provide tests for this new construct.