CSCI 490 problem set 5

Due Tuesday, February 23

Revision 2: compiled Thursday 18th February, 2016 at 16:47

While working on this problem set you may find it helpful to make use of the λ-calculus evaluator available at `https://fling.seas.upenn.edu/~cis39903/cgi-bin/LC.cgi`. There is also a command-line λ-calculus interpreter; you can download an installer from the course website.

You will not need to formally prove your answers on this problem set, but you should justify them, *e.g.* by giving an example reduction sequence that illustrates the behavior of some λ-calculus term you have defined, or by giving an informal argument explaining why your solution is correct.

What to turn in

- A document in both `.lhs` and `.pdf` form, as usual. In particular the `.lhs` document should load cleanly in `ghci` and allow your solution to Exercise 1 to be run.
- A text file with the definitions of your lambda calculus terms, in a format suitable for loading into the lambda calculus evaluator.

To typeset your lambda calculus terms in LaTeX, you can just use `verbatim` environments; there's no need to get fancy typesetting them with actual lambdas and so on. However, if you *do* want to typeset them in a fancy way, you can use the `\lam` and `\app` commands defined in `hshw.sty`. For example,

```
\lam{x}{\lam{y}{\app{x}{y}}}
```

produces

$$\lambda x.\, \lambda y.\, x\ y.$$

These commands ensure proper spacing after the period and between terms in an application.

Rubric

For full credit, your solutions should demonstrate a proficient understanding of the following topics:

- Lambda calculus syntax and semantics (*e.g.* $\alpha\beta\eta$-equivalence, bound and free variables, substitution, reduction)
- Church numerals
- Generalized Church encoding

The untyped λ-calculus

Exercise 1 Consider the Haskell data type

```
data Term where
  Var :: String -> Term
  Lam :: String -> Term -> Term
  App :: Term -> Term -> Term
```

which represents a naïve encoding of λ-calculus terms as Haskell values. Write a function

For extra (brownie) points, make sure it takes linear time...

```
freeVars :: Term -> [String]
```

which computes the set of all free variables of a term. For example,

```
freeVars (App (Var "z") (Lam "y" (App (Var "y") (Var "x")))) = ["z","x"].
```

Natural numbers

Recall from lecture that we can represent natural numbers in the λ-calculus by their *Church encoding*, that is, the natural number n is represented by the λ-calculus term

$$\lambda s.\, \lambda z.\, s\ (s\ \dots (s\ z))$$

where the s is repeated n times. In other words, a natural number is *represented by its own fold*, that is, a function which takes as arguments a function s and starting value z, and applies s to z a certain number of times.

We will abbreviate Church-encoded natural numbers as n_λ. For example,

$$3_\lambda = \lambda s.\, \lambda z.\, s\ (s\ (s\ z)).$$

The following exercises ask you to build up facilities for doing computation with natural numbers.

Exercise 2 Define the natural number 0_λ, and define a function *succ* which takes a (Church-encoded) natural number and yields its (Church-encoded) successor.

In order to test your natural number functions in the λ-calculus evaluator, you will want to evaluate things like, *e.g.*, `plus two three S Z` instead of just `plus two three`. The reason is that reduction gets "stuck" when the outermost term constructor is a λ. In order to "fully reduce" a Church-encoded number like `plus two three`, you can apply it to some arguments, in this case, just two free variables `S` and `Z` to stand in for successor and zero.

Exercise 3 Define a λ-calculus term *plus* that adds Church numerals. That is, *plus* should have the property that

$$\mathit{plus}\ m_\lambda\ n_\lambda \equiv (m+n)_\lambda,$$

where $\equiv$ denotes $\alpha\beta\eta$-equivalence of λ-calculus terms.

Exercise 4 Define a λ-calculus term *mul* that multiplies Church numerals.

Exercise 5 Define a λ-calculus term *exp* that exponentiates Church numerals, that is,

$$exp\ m_\lambda\ n_\lambda \equiv (m^n)_\lambda.$$

Feel free to define other named λ-calculus terms if it makes your solutions more modular/elegant/readable.

Exercise 6 Define a λ-calculus term *iszero* that decides whether a Church numeral is zero. That is, when applied to a Church numeral, it should evaluate to an appropriate Church-encoded boolean.

Church pairs

Exercise 7 Define λ-calculus terms *pair*, *fst*, and *snd* such that

$$fst\ (pair\ x\ y) \equiv x$$

(and similarly for *snd*).

Hint:

Exercise 8 Define a λ-calculus term *pred* such that when n is positive, *pred* applied to n_λ is equivalent to $(n-1)_\lambda$ (*pred* applied to zero can just yield zero).

This problem is tricky! If you are stuck, feel free to ask me for a hint.

Exercise 9 Now define a λ-calculus term *sub* that subtracts Church numerals (truncating at zero in the case of subtracting a larger number from a smaller).

Church lists

Exercise 10 Define λ-calculus terms *nil* and *cons* which represent the constructors for (Church-encoded) lists.

Exercise 11 Define a λ-calculus term *sum* such that, for example,

$$sum\ (cons\ 3_\lambda\ (cons\ 1_\lambda\ (cons\ 4_\lambda\ nil))) \equiv 8_\lambda.$$

Exercise 12 Define a λ-calculus term *filter* which works similarly to Haskell's standard `filter` function.