CSCI 365 Module 4

Due Wednesday 23 February, 2022

Specification

- To receive **Level 1** credit for this module:
  - Complete exercises 1 and 3–10.
  - Your code must adhere to the style guide linked from the course web page.
  - You must complete Quiz 4 for credit, which will test your understanding of the lambda calculus.
- To receive **Level 2** credit for this module:
  - Complete everything required for Level 1.
  - Complete exercises 2 and 11–13.

Instructions

While working on this problem set you should use the command-line λ-calculus interpreter, which is available from the course website. Download and unzip, then execute `stack run` at a command prompt from within the unzipped directory. Please try it early and let me know if you have trouble getting it installed!

Note that a file `bool.lc` is available with some of the definitions for Booleans that we went over in class.

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 `.hs` file with your solution to exercise 1.
- A text file with the definitions of your lambda calculus terms, in a format suitable for loading into the lambda calculus evaluator.

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

```
freeVars :: Term -> Set 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")))) = fromList ["z","x"].
```

The `Set` type can be found in the `Data.Set` module, which you can read about on Hackage: `https://hackage.haskell.org/package/containers-0.6.5.1/docs/Data-Set.html`. To import it into your `.hs` file, you'll want to put something like

```
import Data.Set (Set)
import qualified Data.Set as S
```

at the top. Then you can use the `Set` type and call functions from the library like `S.insert` and `S.union`.

Exercise 2 (Level 2) Ensure that your `freeVars` function takes $O(n \lg n)$ time or better, where n is the size (*i.e.* the total number of constructors) of the `Term`.

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

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

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 3 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 4 Define a λ-calculus term *plus* that adds Church numerals. That is, *plus* should have the property that

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

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

Exercise 5 Define a λ-calculus term *mul* that multiplies Church numerals.

Do not look any of these up on the internet! Not even to "help you get started" or for "inspiration". It is easy to find solutions, and as soon as you do, you have lost a learning opportunity.

Exercise 6 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 7 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 lists

Exercise 8 Define λ-calculus terms *nil* and *cons* which represent the constructors for (Church-encoded) lists.

Remember that we will encode lists as their own folds! You may find it helpful to write out the recipe in Haskell first.

Exercise 9 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 10 Define a λ-calculus term *filter* which works similarly to Haskell's standard `filter` function.

Church pairs and subtraction (Level 2)

Exercise 11 (Level 2) Define λ-calculus terms *pair*, *fst*, and *snd* such that

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

(and similarly for *snd*).

Hint:

Exercise 12 (Level 2) 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. Do not try to find hints online.

Exercise 13 (Level 2) 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).