HW 6

50pts

Posted October 16

Due Monday, December 1 at midnight

A Normal-Order Normal-Form Interpreter for the Lambda Calculus

We now return to our favorite topic of interpreters.

Part 1: Normal-Order to Normal-Form Interpreter in Pseudocode. Your first task is to write a normal-order to normal-form interpreter in pseudocode in the style of Lecture 20. Please include your answer in comments in your Haskell file under heading Problem 1.

Part 2: Coding an Interpreter in Haskell. Next, you will code a normal-order interpreter in Haskell. Note that you will not be coding the recursive interpreter from Problem 1 but a step-by-step interpreter. The key function is the *one step* function. It takes a lambda expression and either (i) carries out a single step of evaluation returning the next expression in the normal-order reduction sequence, or (ii) yields Nothing, meaning there is no redex to reduce.

A lambda expression is of the following form, as we discussed in class:

```
data Expr =
  Var Name --- a variable
  | App Expr Expr --- function application
  | Lambda Name Expr --- lambda abstraction
  deriving (Eq,Show) --- the Expr data type derives from built-in Eq and
             --- Show classes; thus, we can compare and print expressions

type Name = String --- a variable name
```

To test more conveniently, create a few expressions and call your functions on them:

```
e1 = Lambda "x" (Var "x") -- Lx.x
e2 = App e1 e1 -- (Lx.x) (Lx.x)
e3 = Lambda "y" (App (Var "x") (Var "y")) -- Ly.x y
e4 = Lambda "x" e3 -- Lx.Ly.x y
e5 = App (Var "y") (Var "w") -- y w
e6 = App e4 e5 -- (Lx.Ly.x y) (y w)
```

Implement the following functions.

Free variables. `freeVars` takes an expression and returns a list of its free variables:

```
freeVars :: Expr -> [Name]
freeVars = undefined

> freeVars (App (Var "x") (Var "x"))
["x"]
```

Substitution. subst is the substitution function:

```
subst :: (Name,Expr) -> Expr -> Expr
subst = undefined

> subst ("x",e5) e3 -- replace "x" with (y w) in Ly.x y
Lambda "v1" (App (App (Var "y") (Var "w")) (Var "v1")) -- Lv1.y w v1
```

First, remember that all functions in Haskell are curried: i.e., they take just one argument. The above function takes a tuple of a variable, say x, and an expression, say M, and returns a function that takes an expression, say E and returns an expression, namely E[M/x], the result of the substitution of M for every free occurrence of x in E.

Note that you can implement subst as you wish, as long as it renames bound variables when necessary. You can implement the aggressive algorithmic substitution from class (Lecture 15) if you wish, making use of the infinite list of potential fresh variables ["v1","v2","v3"..] filtering variables appropriately. We are testing a normalized version of subst on Submitty, so as long as the expression you produce is correct it earns all points.

Our normalization function uses De Bruijn indices (https://en.wikipedia.org/wiki/De_Bruijn_index) but with the important simplification that we leave free variables as is, rather than turning them into indices. Thus, our function turns the above expression into L.y w 1. (On Submitty we render lambda expressions with L instead of λ.) In De Bruijn, there are no variables under lambda binders and 1 in the above example refers to the (missing) variable under the first lambda binder (and in this case the only lambda binder). We can "restore" the expression into the usual notation by naming 1, e.g., Lz.y w z, Lv1.y w v1, etc.

One step. Now write the one step function:

```
normNF_OneStep::Expr -> Maybe Expr
normNF_OneStep = undefined

> normNF_OneStep e6 -- (Lx.Ly.x y) (y w)
Just (Lambda "v1" (App (App (Var "y") (Var "w")) (Var "v1")))
```

Here the built-in Maybe type is defined as follows:

```
data Maybe a =
 Nothing
 | Just a
```

normNF_OneStep takes an expression and returns a Maybe Expr, where the Maybe type can be thought of as an optional type: if there is a redex, normNF_OneStep selects the normal-order redex and carries out a reduction to expression E returning Just E. (Here Just is part of the Maybe type indicating that the computation does produce a value.) If there is no redex, i.e., the input expression is already in normal form, normNF_OneStep returns Nothing. (Nothing is also part of the Maybe part indicating that the computation does not produce a value.)

Repetition. Now implement a function that does n steps of normal-order reduction:

```
normNF_n :: Int -> Expr -> Expr
normNF_n = undefined
```

```
> normNF_n 100 e1 -- Lx.x
Lambda "x" (Var "x")
```

The function applies one step n times, and note that if the expression is in normal form, `normNF_n` returns it as is.

Pretty printing. The final step is a pretty-printing function that takes a `Maybe Expr` and produces a normalized string such that:

(1) There are no variables under lambda binders replacing them with De Bruijn indices in the function body (again, see `https://en.wikipedia.org/wiki/De_Bruijn_index`). Importantly, we simplify De Bruijn by allowing free variables to remain as is rather than turning them into indices.
and
(2) There are no redundant parentheses. Recall the syntactic conversion we used in class: application is left-associative and it takes precedence over abstraction.

```
prettyPrint :: Maybe Expr -> String
prettyPrint = undefined

> prettyPrint Nothing
""
> prettyPrint (Just e1)
"L.1"
> prettyPrint (Just e6)
"(L.L.2 1) (y w)"
```

In the above example `e1`, representing `Lx.x` turns into `"L.1"` as we eliminate the variable under the binder and have index 1 refer to that (missing) variable. In the second example `e6` representing `(Lx.Ly.x y) (y w)` turns into `(L.L.2 1) (y w)` where index 2 refers to the (missing) variable under the second enclosing lambda binder, and index 1 refers to the (missing) variable under the first (closest) enclosing lambda binder. You can turn it into `(L2.L1.2 1) (y w)` and then to a standard notation: `(Lx.Lz.x z) (y w)`. Note that free variables `y` and `w` remain as is.

Please read this before starting to code:

- Download `Interpreter.hs`, code the function and submit the file in Submitty. We run Haskell 8.8.4 and we are testing `freeVars`, `subst`, `normNF_OneStep`, `normNF_n` and `prettyPrint`.
- Do not include additional imports.
- Part 1 is worth 5 points. Part 2 is 45 points with 40 points autograded on Submitty and 5 TA points for style. Follow the Haskell style guide as much as possible.

Haskell Style Guide. Adapted from Stephanie Weirich's class at UPenn.

- **Write a type signature for every function.** We will be strict about this when grading. Hint: try writing the signature *before* writing the function. If you do write the function first, try deducing the signature and if this doesn't work, there is always the `:t` command.
- Make sure that your code produces no errors or warnings. Code with errors receives 0 on Submitty and we will mark down warnings during TA grading.

- Use consistent indentation.
- Do not use tab characters, use space for indentation. GHC should be flagging tabs, but nevertheless, be careful.
- No line should have more than **80 characters**.
- Use whitespace to make your code readable. Add whitespace on either side of binary operators, e.g., write `3 * n + 1` instead of `3*n+1`.

- Use descriptive names.
- Follow standard Haskell naming conversions: (1) use camelCase for compound names, and (2) use `x` and `xs` when you pattern-match lists.

- Use comments. Each function definition should be preceded by a comment.
- Comment should say what the function does, not how.
- Comments should be concise. Do not overcomment.
- Use full English sentences.

- Do not leave incomplete pattern matches. They will be marked down.
- Tuples, records and datatypes can be decomposed. You can also use the `@` operator if you need a reference to both the object and its components.
  For example, do not use this:

  ```
  f arg1 arg2 = ...  where
     x = fst arg1
     y = snd arg1
     z = fst arg2
  ```

  Use this instead:

  ```
  f (x,y) (z,_) = ...
  ```
- Combine nested case expressions.
  For example, do not use this:

  ```
  case x of
        Red -> case y of
                 Red  -> True
                 Blue -> False
        Blue -> case y of
                 Red  -> False
                 Blue -> True
  ```

  Use this instead:

  ```
  case (x,y) of
  ```

```
(Red,   Red) -> True
(Blue, Blue) -> True
(   _,    _) -> False
```

- Use library functions, unless the assignment explicitly forbids them. Use Haskell's search engine Hoogle to look up library functions.