COMS 6998 TLC: Homework 2

Stephen A. Edwards, Columbia University

Due Sunday, April 2 at 11:59 PM, 2023

This homework involves implemnting various lambda calculus operations in Haskell. Download and install the Haskell Tool Stack from `https://www.haskellstack.org`.

Download and unpack the skeleton code for the homework from
`https://www.cs.columbia.edu/~sedwards/classes/2023/6998-spring-tlc/lambda.tar.gz`

Unpack, build, and test the Lambda calculus parser we have provided. `lcp` is the simple command-line parser, which parses and pretty-prints untyped lambda calculus expressions; `runtests.sh` runs `lcp` on the examples in the `tests/examples/` directory and verifies they produce the expected result.

```
$ tar zxf lambda.tar.gz
$ cd lambda
$ stack build
Building all executables for `lambda' once. After a successful build ...
...
Building library for lambda-0.1.0.0..
[1 of 4] Compiling AST
...
Installing library in ...
Installing executable plc in ...
$ stack run lcp < tests/examples/y.lam
\f . (\x . f (x x)) (\x . f (x x))
$ cd tests
$ ./runtests.sh
beta1...OK
beta2...OK
...
weak-normal-form...OK
y...OK
$
```

The parser recognizes the following grammar

expr ::= *id*
| **** *id*$^+$ **.** *expr*
| *expr expr*
| **(** *expr* **)**

and parses it into the following data type, defined in `src/AST.hs`

```
data Expr = Var String
          | Lam String Expr
          | App Expr Expr
```

A variable identifer is any string that doesn't include whitespace or `\ . ( ) -`
Single-line comments start with `--`

Note that lambdas may take more than one argument; the parser desugars these into curried functions.

Invoking GHCi with `stack` imports the parser library so you can experiment with it. Note that backslashes need to be doubled in string literals.

If you edit a source file while GHCi is running, use `:r` to reload/recompile your code.

```
$ stack ghci
...
Ok, five modules loaded.
ghci> :t parse
parse :: String -> Expr
ghci> :i Expr
type Expr :: *
data Expr = Var String | Lam String Expr | App Expr Expr
ghci> parse "\\f.(\\x.f(x x))(\\x. f (x x))"
\f . (\x . f (x x)) (\x . f (x x))
ghci> parse "\\a b c. a b c"
\a . \b . \c . a b c
ghci> :r
...
Ok, five modules loaded.
ghci>
```

1. Complete the implementation of the `fv` function in `src/HW.hs` so that it computes the free variables of an expression.

   Follow the definition of the free variable function from lecture:

$$\begin{aligned} \mathrm{FV}(x) &= \{x\} \\ \mathrm{FV}(\lambda x\ .\ M) &= \mathrm{FV}(M) \setminus \{x\} \\ \mathrm{FV}(M_1\ M_2) &= \mathrm{FV}(M_1) \cup \mathrm{FV}(M_2) \end{aligned}$$

   Implement this function in Haskell with the following type signature.

```
fv :: Expr -> Set.Set String
```

   It should behave like this

```
ghci> fv $ parse "\\x.x"
fromList []
ghci> fv $ parse "\\x.x y z"
fromList ["y","z"]
ghci> fv $ parse "x (\\x.x y) z"
fromList ["x","y","z"]
```

2. Complete the implementation of the `subst` function in `src/HW.hs` so that it safely substitutes a given variable for an expression while avoiding name capture.

   Follow these substitution rules, which adds one to those presented in class

$$
\begin{aligned}
x[x := N] &= N \\
y[x := N] &= y \\
(M_1\ M_2)[x := N] &= (M_1[x := N]\ M_2[x := N]) \\
(\lambda x\ .\ M)[x := N] &= (\lambda x\ .\ M) \\
(\lambda y\ .\ M)[x := N] &= (\lambda y\ .\ M[x := N]) \text{ if } y \notin \mathrm{FV}(N) \\
(\lambda y\ .\ M)[x := N] &= (\lambda y'\ .\ M[y := y'][x := N]) \text{ where } y' \notin \mathrm{FV}(N) \cup \mathrm{FV}(M)
\end{aligned}
$$

   The tricky case is when you enter the scope of a lambda term whose argument is a free variable of what you're substituting, i.e., $(\lambda y\ .\ M)[x := N]$ when $y \in \mathrm{FV}(N)$. Substituting in N in this scope would be incorrect because the argument y will capture a free variable in N, changing its meaning.

   Consider $(\lambda y\ .\ x)[x := y]$. Ignoring name capture would give $\lambda y\ .\ y$, which means something else. We would like to produce a correct new expression for this since it would be awkward to force the caller to choose a different expression to substitute.

   The solution is to alpha rename $(\lambda y\ .\ M)$ into $(\lambda y'\ .\ M[y := y'])$ before substituting N for x in the new body. The important thing is that y' must avoid capture, i.e., not be a free variable in either M or N.

   To this end, we have provided `pickFresh`, which takes a set of names it must avoid and a root name and returns a distict name. Use this along with your `fv` function to choose a satisfactory y'.

Here are test cases that illustrate the rules:

```
ghci> subst (parse "\\x.x") "x" $ parse "x"
\x . x
ghci> subst (parse "\\x.x") "x" $ parse "y"
y
ghci> subst (parse "\\x.x") "x" $ parse "x y x"
(\x . x) y (\x . x)
ghci> subst (parse "\\x.x") "x" $ parse "x (\\x.x y z)"
(\x . x) (\x . x y z)
ghci> subst (parse "\\z.z") "x" $ parse "\\y. x (\\x. y x) x"
\y . (\z . z) (\x . y x) (\z . z)
ghci> subst (parse "y") "x" $ parse "\\y.x"
\y' . y
ghci> subst (parse "y") "x" $ parse "\\y.x y'"
\y'' . y y'
ghci> subst (parse "y") "x" $ parse "\\y.y x y'"
\y'' . y'' y y'
ghci> subst (parse "y") "x" $ parse "\\y.y x y' (\\y. y y y')"
\y'' . y'' y y' (\y'' . y'' y'' y')
ghci> subst (parse "y") "x" $ parse "\\y.y x y' (\\y''. y'' y y')"
\y'' . y'' y y' (\y''' . y''' y'' y')
ghci> subst (parse "y") "x" $ parse "\\y.y x y' (\\y''. y'' y y' x)"
\y'' . y'' y y' (\y''' . y''' y'' y' y)
```

3. Complete the `normalstep` function, which should either return the result of a normal order reduction step or `Nothing` indicating that no further reduction is possible.

   Implement the following rules, using your `subst` function for β reduction:

$$\frac{M[x := N] = M'}{(\lambda x\,.\,M)\;N \longrightarrow M'}\ \text{beta} \qquad \frac{M \longrightarrow M'}{\lambda x\,.\,M \longrightarrow \lambda x\,.\,M'}\ \text{body}$$

$$\frac{M \longrightarrow M'}{M\;N \longrightarrow M'\;N}\ \text{func} \qquad \frac{M \not\longrightarrow M' \quad N \longrightarrow N'}{M\;N \longrightarrow M\;N'}\ \text{arg}$$

```
ghci> normalstep $ parse "x"
Nothing
ghci> normalstep $ parse "(\\y. x y z) (\\x. x)"
Just x (\x . x) z
ghci> normalstep $ parse "(\\x. \\y.x) y"
Just \y' . y
ghci> normalstep $ parse "\\z. (\\x. \\y.x) y"
Just \z . \y' . y
ghci> normalstep $ parse "(\\x. \\y.y x y' (\\y''. y'' y y' x)) y"
Just \y'' . y'' y y' (\y''' . y''' y'' y' y)
ghci> normalstep $ parse "y (\\x.\\y.x y z)(\\z.\\y.(\\x.x y) w)"
Just y (\x . \y . x y z) (\z . \y . w y)
ghci> normalstep $ parse "y(\\x.\\y.x(\\x.x)x z)(\\z.\\y.(\\x.x y) w)"
Just y (\x . \y . x (\x . x) x z) (\z . \y . w y)
ghci> normalstep $ parse "y(\\x.\\y.(\\x.x)x z)(\\z.\\y.(\\x.x y) w)"
Just y (\x . \y . x z) (\z . \y . (\x . x y) w)
ghci> normalstep $ parse "(\\x.\\y.y)((\\z.z z)(\\z.z z))"
Just \y . y
```

We have provided `printnormal`, which parses a lambda calculus string then prints each step of its reduction to normal form using the `repeatedly` function and `normalstep`. This makes it easy to verify some proofs from class.

Here's `fst (pair a b)` and `snd (pair a b)`.

```
ghci> printnormal "(\\p.p(\\x y.x)) ((\\x y f. f x y) a b)"
(\p . p (\x . \y . x)) ((\x . \y . \f . f x y) a b)
(\x . \y . \f . f x y) a b (\x . \y . x)
(\y . \f . f a y) b (\x . \y . x)
(\f . f a b) (\x . \y . x)
(\x . \y . x) a b
(\y . a) b
a
ghci> printnormal "(\\p.p(\\x y.y)) ((\\x y f. f x y) a b)"
(\p . p (\x . \y . y)) ((\x . \y . \f . f x y) a b)
(\x . \y . \f . f x y) a b (\x . \y . y)
(\y . \f . f a y) b (\x . \y . y)
(\f . f a b) (\x . \y . y)
(\x . \y . y) a b
(\y . y) b
b
```

```
ghci> lzero="(\\f x.x)"
ghci> lone="(\\f x.f x)"
ghci> ltwo="(\\f x.f(f x))"
ghci> lthree="(\\f x.f(f(f x)))"
ghci> lsucc="(\\n f x.f(n f x))"
ghci> printnormal $ lsucc ++ lzero
(\n . \f . \x . f (n f x)) (\f . \x . x)
\f . \x . f ((\f . \x . x) f x)
\f . \x . f ((\x . x) x)
\f . \x . f x
ghci> printnormal $ lsucc ++ lone
(\n . \f . \x . f (n f x)) (\f . \x . f x)
\f . \x . f ((\f . \x . f x) f x)
\f . \x . f ((\x . f x) x)
\f . \x . f (f x)
ghci> printnormal $ lsucc ++ ltwo
(\n . \f . \x . f (n f x)) (\f . \x . f (f x))
\f . \x . f ((\f . \x . f (f x)) f x)
\f . \x . f ((\x . f (f x)) x)
\f . \x . f (f (f x))
ghci> lplus="(\\m n f x.m f(n f x))"
ghci> printnormal $ lplus ++ lthree ++ ltwo
(\m . \n . \f . \x . m f (n f x)) (\f . \x . f (f (f x))) (\f . \x . f (f x))
(\n . \f . \x . (\f . \x . f (f (f x))) f (n f x)) (\f . \x . f (f x))
\f . \x . (\f . \x . f (f (f x))) f ((\f . \x . f (f x)) f x)
\f . \x . (\x . f (f (f x))) ((\f . \x . f (f x)) f x)
\f . \x . f (f (f ((\f . \x . f (f x)) f x)))
\f . \x . f (f (f ((\x . f (f x)) x)))
\f . \x . f (f (f (f (f x))))
ghci> lmult="(\\m n f.m(n f))"
ghci> printnormal $ lmult ++ ltwo ++ lthree
(\m . \n . \f . m (n f)) (\f . \x . f (f x)) (\f . \x . f (f (f x)))
(\n . \f . (\f . \x . f (f x)) (n f)) (\f . \x . f (f (f x)))
\f . (\f . \x . f (f x)) ((\f . \x . f (f (f x))) f)
\f . \x . (\f . \x . f (f (f x))) f ((\f . \x . f (f (f x))) f x)
\f . \x . (\x . f (f (f x))) ((\f . \x . f (f (f x))) f x)
\f . \x . f (f (f ((\f . \x . f (f (f x))) f x)))
\f . \x . f (f (f ((\x . f (f (f x))) x)))
\f . \x . f (f (f (f (f (f x)))))
```

```
ghci> lpred="(\\n f x.n (\\g h.h(g f)) (\\u.x) (\\u.u))"
ghci> printnormal $ lpred ++ lzero
(\n . \f . \x . n (\g . \h . h (g f)) (\u . x) (\u . u)) (\f . \x . x)
\f . \x . (\f . \x . x) (\g . \h . h (g f)) (\u . x) (\u . u)
\f . \x . (\x . x) (\u . x) (\u . u)
\f . \x . (\u . x) (\u . u)
\f . \x . x
ghci> printnormal $ lpred ++ lone
(\n . \f . \x . n (\g . \h . h (g f)) (\u . x) (\u . u)) (\f . \x . f x)
\f . \x . (\f . \x . f x) (\g . \h . h (g f)) (\u . x) (\u . u)
\f . \x . (\x . (\g . \h . h (g f)) x) (\u . x) (\u . u)
\f . \x . (\g . \h . h (g f)) (\u . x) (\u . u)
\f . \x . (\h . h ((\u . x) f)) (\u . u)
\f . \x . (\u . u) ((\u . x) f)
\f . \x . (\u . x) f
\f . \x . x
ghci> printnormal $ lpred ++ ltwo
(\n . \f . \x . n (\g . \h . h (g f)) (\u . x) (\u . u)) (\f . \x . f (f x))
\f . \x . (\f . \x . f (f x)) (\g . \h . h (g f)) (\u . x) (\u . u)
\f . \x . (\x . (\g . \h . h (g f)) ((\g . \h . h (g f)) x)) (\u . x) (\u . u)
\f . \x . (\g . \h . h (g f)) ((\g . \h . h (g f)) (\u . x)) (\u . u)
\f . \x . (\h . h ((\g . \h . h (g f)) (\u . x) f)) (\u . u)
\f . \x . (\u . u) ((\g . \h . h (g f)) (\u . x) f)
\f . \x . (\g . \h . h (g f)) (\u . x) f
\f . \x . (\h . h ((\u . x) f)) f
\f . \x . f ((\u . x) f)
\f . \x . f x
```

I removed the spaces around the dots to make the following fit on the page:

```
ghci> printnormal $ lpred ++ lthree
(\n.\f.\x.n (\g.\h.h (g f)) (\u.x) (\u.u)) (\f.\x.f (f (f x)))
\f.\x.(\f.\x.f (f (f x))) (\g.\h.h (g f)) (\u.x) (\u.u)
\f.\x.(\x.(\g.\h.h (g f)) ((\g.\h.h (g f)) ((\g.\h.h (g f)) x))) (\u.x) (\u.u)
\f.\x.(\g.\h.h (g f)) ((\g.\h.h (g f)) ((\g.\h.h (g f)) (\u.x))) (\u.u)
\f.\x.(\h.h ((\g.\h.h (g f)) ((\g.\h.h (g f)) (\u.x)) f)) (\u.u)
\f.\x.(\u.u) ((\g.\h.h (g f)) ((\g.\h.h (g f)) (\u.x)) f)
\f.\x.(\g.\h.h (g f)) ((\g.\h.h (g f)) (\u.x)) f
\f.\x.(\h.h ((\g.\h.h (g f)) (\u.x) f)) f
\f.\x.f ((\g.\h.h (g f)) (\u.x) f)
\f.\x.f ((\h.h ((\u.x) f)) f)
\f.\x.f (f ((\u.x) f))
\f.\x.f (f x)
```

Applying the Y combinator to an abstract function H correctly gives
`H (H (H (H (H (H (H ...` which quickly becomes tedious.

```
ghci> ly="(\\f.(\\x.f(x x))(\\x.f(x x)))"
ghci> mapM_ print $ take 8 $ repeatedly normalstep $ parse $ ly ++ "H"
(\f . (\x . f (x x)) (\x . f (x x))) H
(\x . H (x x)) (\x . H (x x))
H ((\x . H (x x)) (\x . H (x x)))
H (H ((\x . H (x x)) (\x . H (x x))))
H (H (H ((\x . H (x x)) (\x . H (x x)))))
H (H (H (H ((\x . H (x x)) (\x . H (x x))))))
H (H (H (H (H ((\x . H (x x)) (\x . H (x x)))))))
H (H (H (H (H (H ((\x . H (x x)) (\x . H (x x))))))))
```

Factorial works, although it's a little slow

```
ghci> liszero="(\\n.n(\\x.(\\x y.y)) (\\x y.x))"
ghci> lfac="((\\f n.("++liszero++"n)"++lone++
                            "("++lmult++"n(f("++lpred++"n)))))"
ghci> printnormal $ ly ++ lfac ++ lzero
...
\f . \x . f x
ghci> printnormal $ ly ++ lfac ++ lone
...
\f . \x . (\f . \x . f x) f x
\f . \x . (\x . f x) x
\f . \x . f x
ghci> printnormal $ ly ++ lfac ++ ltwo
...
\f . \x . f (f x)
ghci> print $ last $ repeatedly normalstep $
         parse $ ly ++ lfac ++ lthree
\f . \x . f (f (f (f (f x))))
ghci> print $ last $ repeatedly normalstep $
         parse $ ly ++ lfac ++ "("++lsucc++lthree++")"
\f.\x.f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f x))))))))))))))))))))))
```