1 Introduction

The purpose of this assignment is to learn more about the evaluation of lambda calculus.

Every function should be commented. At the very least, the comments should state what the function does, which arguments it takes, and what result it produces.

This assignment is graded out of 100. It is worth 10% of your final grade.

2 Lambda Calculus

Solve problems 1, 2 (a-e), 4, and 5 on page 158–159 in Syntax and Semantics of Programming Languages, by Ken Slonneger and Barry Kurtz.

Hand in your results in as a text file called theory. Use “L” to signify the lambda operator.

3 Pure Lambda Expressions in Haskell

In the remainder of this assignment we are going to manipulate pure lambda expressions. These are defined in Haskell like this:

```haskell
data LExpr = Var String |
            Comb LExpr LExpr |
            Abstr String LExpr

-- This declaration allows us to print out
-- lambda expressions easily, for example:
--   > (Abstr "v" (Comb (Var "v") (Var "v")))
--       (L v.(v v))
--   > [(Abstr "v" (Comb (Var "v") (Var "v")))]
--       [(L v.(v v))]

instance Show LExpr where
  show (Var n) = n
  show (Comb e1 e2) = "(" ++ (show e1) ++ " " ++ (show e2) ++ ")"
  show (Abstr v e) = "(L " ++ v ++ "." ++ (show e) ++ ")"
```

1
-- This declaration allows us to compare two
-- lambda expressions for (deep) equality, for
-- example
-- > (Abstr "v" (Comb (Var "v") (Var "v"))) == (Abstr "v" (Comb (Var "v") (Var "v")))
-- True
-- Note, the comparison is *exact*; two expressions are
-- not considered equal even if some variable renaming
-- would make them so.

instance Eq LExpr where

  Var a == Var b = a == b
  Comb a b == Comb c d = (a == c) && (b == d)
  Abstr a b == Abstr c d = (a == c) && (b == d)
  _ == _ = False

4 Lambda Calculus Evaluator [50 points]

Write a lambda calculus evaluator in Haskell. Page 162–165 in Syntax and Semantics of Programming Languages, by Ken Slonneger and Barry Kurtz may be somewhat helpful, for those who know Prolog.

My implementation of the three functions below is 17 lines long, including function signatures but excluding comments.

1. Start by defining a function normal which returns True if an expression is in normal form, i.e. if it contains no beta redexes. Remember that a redex is an expression of the form ((lx.E) y)

Example:

-- return True if the expression is in normal form, i.e.
-- if it contains no redexes.
normal :: LExpr -> Bool
normal ... = ...

> normal (Var "x")
True
> normal (Comb (Var "x") (Var "y"))
True
> normal (Comb (Abstr "x" (Var "x")) (Var "y"))
False

2. Then write a function reduce which finds the leftmost, outermost redex, and, if one is found, applies it: [30 points]

-- Find the leftmost, outermost redex, and apply it.
-- If no redex can be found, return the original expression.
reduce :: LExpr -> LExpr
reduce ... = ...
Reduce (Comb (Var "x") (Var "y"))
(x y)
Reduce (Comb (Abstr "x" (Var "x")) (Var "y"))
y
Reduce (Comb (Abstr "x" (Var "x")) (Comb (Abstr "z" (Var "z")) (Var "w")))
((L z.(z)) w)

A function substs (similar to the one from assignment 2) will be provided for you to do the actual beta reduction.

3. Finally, write a function eval which repeatedly reduces an expression, until no more redexes can be found (i.e. until it is in normal form): [10 points]

-- return the normal form of an expression, if one exists
-- i.e. repeatedly apply 'reduce':
-- reduce (reduce (reduce ... (reduce e)))
-- until the resulting expression is in a normal
-- form. Note: if no normal form exists, this function
-- will not terminate.
eval :: LExpr -> LExpr
eval = ...

> eval (Comb (Abstr "x" (Var "x")) (Comb (Abstr "z" (Var "z")) (Var "w")))
w
HINT: The Haskell prelude has a function until that might be helpful.

5 Pure Lambda Calculus Functions [20 points]

Add functions to your evaluator that perform arithmetic in pure lambda calculus.

1. Define a function nat n that returns the pure lambda calculus representation of the natural number n: [5 points]
nat :: Int -> LExpr
nat ... = ...

> nat 6
(L f.(L x.(f (f (f (f (f x)))))))

2. Define pure lambda calculus constants zero, one, ..., five: [5 points]

> zero
(L f.(L x.x))
> five
(L f.(L x.(f (f (f (f x))))))

3. Define lambda calculus functions succL and addL for arithmetic: [5 points]
Main> eval (Comb succL one)
(L f.(L x.(f (f x))))
Main> eval (Comb (Comb addL two) three)
(L f.(L x.(f (f (f (f (f x)))))))

4. Define a lambda calculus expression `inftyL` whose evaluation never terminates: [5 points]

Main> eval inftyL
^C{Interrupted!}

6 Submission and Assessment

The deadline for this assignment is noon, Mon Mar 7. You should submit the assignment (a text-file containing the function definitions) electronically using the Unix command `turnin cs520.3 theory reduce.hs`. This assignment is worth 10% of your final grade.

Don’t show your code to anyone, don’t read anyone else’s code, don’t discuss the details of your code with anyone. If you need help with the assignment see the instructor or TA.