University of Arizona, Department of Computer Science

CSc 520 — Assignment 2 — Due noon, Wed Feb 16 — 10%

Christian Collberg
February 7, 2005

1 Introduction

The purpose of this assignment is to become familiar with Lambda Calculus and Haskell programming.

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 [20 points]

Solve problems 1,2, and 3 on page 146–147 in *Syntax and Semantics of Programming Languages*, by Ken Slonneger and Barry Kurtz.

For problem 2 show the steps in which the substitutions are carried out, as shown in page 146 in Slonneger and Kurtz.

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

3 Set operations [30 points]

In the next section we will need operations on sets. So, start by giving Haskell definitions for `member`, `union`, and `delete`.

1. Implement the operation `member x l`, where x is an arbitrary type and l a list of this type:[10 points]

```
member :: (Eq a) => a -> [a] -> Bool
member x l = ...
```

   The "`(Eq a) =>`" part of the signature just means that `member` can only work on types for which equality is defined.

   Don't use recursion for this function! (Hint: use some combination of `or` and `map`.)

Examples:

```
> member 1 []
False
> member 2 [1,2,3]
True
> member "hoi" ["hi","hoi","hej"]
True
```

2. Define a function `union a b` that implements set union: [10 points]

   ```
   union :: (Eq a) => [a] -> [a] -> [a]
   union a b = ...
   ```

   Examples:

   ```
   > union [] []
   []
   > union [1] [1]
   [1]
   > union [1] [1,2]
   [1,2]
   > union [1,3] [1,2]
   [3,1,2]
   ```

   For this and other functions in this assignment you will find it convenient to use the following syntax:

   ```
   fun :: ....
   fun arg1 arg2 ... argM
      | condition1 = expr1
      | condition2 = expr2
              ...
      | conditionN = exprN
      | otherwise = expr0
   ```

   Here, each condition is evaluated until one evaluates to `True`, after which the corresponding expression is evaluated and its value returned.

3. Define a function `delete a b` that removes all occurences of `b` from the list `a`: [10 points]

   ```
   delete :: Eq a => [a] -> a -> [a]
   delete l x = ...
   ```

   Examples:

   ```
   > delete [1,2,3,3,2] 4
   [1,2,3,3,2]
   > delete [1,2,3,3,2] 1
   [2,3,3,2]
   > delete [1,2,3,3,2] 3
   [1,2,2]
   > delete [1,2,3,3,2] 2
   [1,3,3]
   ```

You should download `lambda.hs` from the class home page: `http://www.cs.arizona.edu/~collberg/Teaching/520/2005/Assignments/lambda/hs`. To run your program on `lectura`, do:

```
> /usr/local/hugs98/bin/hugs
> :load lambda.hs
> showLExpr e1
"5"
```

4 Lambda Expressions in Haskell

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

```
data LExpr = Const Int |
             Var String |
             Comb LExpr LExpr |
             Abstr String LExpr

showLExpr :: LExpr -> String
showLExpr (Const n) = show n
showLExpr (Var v) = v
showLExpr (Comb e1 e2) = "(" ++ (showLExpr e1) ++ " " ++ (showLExpr e2) ++ ")"
showLExpr (Abstr v e) = "(lambda " ++ v ++ " . " ++ (showLExpr e) ++ ")"
```

Note the similarity between this definition of a Lambda expression and the one given on page 141 of Slonneger and Kurtz.

Using this definition we can define some lambda expressions:

```
e1 = (Const 5)
e2 = (Var "v")
e3 = (Comb (Const 5) (Var "v"))
e4 = (Abstr "v" (Var "v"))
e5 = (Abstr "v" (Comb (Var "v") (Var "v")))
e6 = (Abstr "y"
          (Comb
             (Abstr "f" (Comb (Var "f") (Var "x")))
             (Var "y")
          )
     )
```

and print them out:

```
> showLExpr e1
"5"
> showLExpr e3
```

```
"(5 v)"
> showLExpr e4
"(lambda v . v)"
> showLExpr e5
"(lambda v . (v v))"
> showLExpr e6
"(lambda y . ((lambda f . (f x)) y))"
```

5 Free Variables [20 points]

Implement the function `fv e`

```
fv :: LExpr -> [String]
```

that returns the set of free variables in a lambda expression. The result should be a list of strings. The algorithm is given on page 145 of Slonneger and Kurtz:

- The free variables of an expression E, $\mathsf{FV}(E)$ is computed as
  1. $\mathsf{FV}(c) = \emptyset$ for any constant c.
  2. $\mathsf{FV}(x) = \{x\}$ for any variable x.
  3. $\mathsf{FV}((E_1\ E_2)) = \mathsf{FV}(E_1) \cup \mathsf{FV}(E_2)$.
  4. $\mathsf{FV}((\lambda x.E)) = \mathsf{FV}(E) - \{x\}$.

Examples:

```
> fv e6
["x"]
> fv e1
[]
> fv e2
["v"]
```

6 Substitutions [30 points]

Implement the function `subst` $E\ v\ E_1$ that performs the substitution $E[v \to E_1]$. The result should be a new lambda expression. The algorithm is given on page 146 of Slonneger and Kurtz:

a) $v[v \to E_1] = E_1$ for any variable v

b) $x[v \to E_1] = x$ for any variable $x \neq v$

c) $c[v \to E_1] = c$ for any constant c

d) $(E_1\ E_2)[v \to E_3] = (E_1[v \to E_3]\ E_2[v \to E_3])$

e) $(\lambda v.E)[v \rightarrow E_1] = (\lambda v.E)$

f) $(\lambda x.E)[v \rightarrow E_1] = (\lambda x.E[v \rightarrow E_1])$, when $x \neq v$ and $x \notin \mathsf{FV}(E_1)$

g) $(\lambda x.E)[v \rightarrow E_1] = (\lambda z.E[x \rightarrow z][v \rightarrow E_1])$, when $x \neq v$ and $x \in \mathsf{FV}(E_1)$ where $z \neq v$ and $z \notin \mathsf{FV}((E\ E_1))$

Here's the beginning of the `subst`-function and a function `testSubst` that can be used to test `subst`:

```
-- The expression
--     subst e v e1
-- performs the substitution
--     e[v->e1]
-- where e and e1 are lambda exressions
-- and v is a (string) variable.
subst :: LExpr -> String -> LExpr -> LExpr
subst (Const x) _ _ = (Const x)
        .....

testSubst :: LExpr -> String -> LExpr -> String
testSubst e v e1 = (showLExpr e) ++ "[" ++ v ++ "->" ++
                   (showLExpr e1) ++ "] ==> " ++
                   (showLExpr (subst e v e1))
```

This is the example worked on page 146 of Slonneger and Kurtz:

```
> testSubst e6 "x" (Comb (Var "f") (Var "y"))
"(lambda y . ((lambda f . (f x)) y))[x->(f y)] ==> (lambda x1 . ((lambda x1 . (x1 (f y))) x1))"
```

Note that the result is different from what's in the book, due to how new fresh variables are chosen in the g)-part of the algorithm.

As part of your implementation you may want to define a function

```
fresh :: [String] -> String
```

which, given a set of variables returns a new ("fresh") variable not in the set.

7 Submission and Assessment

The deadline for this assignment is noon, Wed Feb 16. You should submit the assignment (a text-file containing the function definitions) electronically using the `Unix` command ⌜`turnin cs520.2 theory lambda.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.