COMP105 Assignment: Lambda Calculus

Due Sunday, April 10, at 11:59PM.

Setup and Instructions

You will need

```
git clone /comp/105/git/lambda
```

which chould give you a directory Lambda with files `linterp.sml`, `Lhelp.ui`, `Lhelp.uo`, `Makefile`, and `basis.lam`.

Do all six problems below. There are three problems on implementing the lambda calculus, which you'll do with a partner, and three problems on programming with Church numerals, which you'll do on your own. For problems 1-3, modify the `linterp.sml` file. Put your answers for the Church numerals problems, 4-6, in a file called `church.lam`.

Introduction to the Lambda interpreter

You will be working with a small, interactive interpreter for the lambda calculus. This section explains what syntax to use and how to interact with the interpreter.

Concrete syntax

Every definition must be terminated with a semicolon. Comments are C++ style line comments, starting with the string `//` and ending at the next newline. A definition can be a term, a binding, or a `use` statement.
A `use` statement loads a file into the interpreter as if it had been typed in directly. A `use` statement is of the form
`-> use filename;`
When a term is entered at the toplevel, any free variables in the term which appear in the environment are substituted with their bindings in the environment, then the term is reduced to normal form (if possible) and the result is printed.
`-> term;`
A binding first evaluates the term on the right hand side as above, and then binds the name on the left hand side to the resulting term in the environment. It is an error for the term on the right hand side to contain any free variables which are not bound in the environment. Bindings are of the form:
`-> name = term;`
or
`-> noreduce name = term;`
If the `noreduce` keyword appears, the term on the right-hand side is not normalized. This tactic can be useful for terms that have no normal form, such as

```
noreduce bot = (\x.x x)(\x.x x);
noreduce Y   = \f.(\x.f(x x))(\x.f(x x));
```

A lambda term can be either a variable, a lambda abstraction, an application, or a parenthesized lambda term. Precedence is as in ML. A lambda abstraction is written as follows. Note that each lambda abstraction can abstract over only one variable.
`\name.term`
Application of one term to another is written:
`t1 t2`

The lambda interpreter is very liberal about names of variables. A name is any string of characters that contains neither whitespace, nor control characters, nor any of the following characters: `\ ( ) . = /`. Also, the string `use` is reserved and is therefore not a name. So for example, the following are all legal:

```
1    = \f.\x.f x;
True = \x.\y.x;
one  = True 1;
```

A short example transcript

A healthy lambda interpreter should be capable of something like the following:

<transcript>=
```
-> true  = \x.\y.x;
-> false = \x.\y.y;
-> pair  = \x.\y.\f.f x y;
-> fst = \p. p (\x.\y.x);
-> snd = \p.p(\x.\y.y);
-> true;
\x.\y.x
-> fst (pair true false);
\x.\y.x
-> snd (pair true false);
\x.\y.y
-> if = \x.\y.\z.x y z;
if
-> (if true fst snd) (pair false true);
\x.\y.y
-> (if false fst snd) (pair true false);
\x.\y.y
```

For more example definitions, see the `basis.lam` file distributed with the assignment.

Modifying the Lambda Interpreter: Exercises for pairs

The purpose of these problems is to help you learn about substitution and reduction, the fundamental notions of the lambda calculus. We also give you a little more practice in continuation passing, which is an essential technique in lambda-land. **You may do these exercises by yourself or with a partner.**

For these problems, define appropriate types and functions in `linterp.sml`. When you are done, you will have a working lambda interpreter. Some of the code we give you (`Lhelp.ui` and `Lhelp.uo`) is object code only, so you will have to build the interpreter using Moscow ML. Just typing `make` should do it.

1. Evaluation—Basics (difficulty *). This problem has two parts:

a. Using ML, create a type definition for a type `term`, which should represent a term in the untyped lambda calculus. Using your representation, define the following functions with the given types:

```
lam : string -> term -> term    (* lambda abstraction *)
app : term -> term -> term      (* application        *)
var : string -> term            (* variable           *)
cpsLambda :                     (* observer *)
  term ->
  (string -> term -> 'a) ->
  (term -> term -> 'a) ->
  (string -> 'a) ->
  'a
```

These functions must obey the following algebraic laws:

```
cpsLambda (lam x e)  f g h = f x e
cpsLambda (app e e') f g h = g e e'
cpsLambda (var x)    f g h = h x
```

b. Using `cpsLambda`, define a function `toString` of type `term -> string` that converts a term to a string in uScheme syntax. Your `toString` function should be *independent* of your representation. That is, it should work using the functions above.

My solution to this problem is under 30 lines of ML code.

2. Evaluation—Substitution (difficulty **). Implement substitution on your `term` representation. Use a function `subst` of type `string * term -> term -> term`. To compute the substitution *M[x|–>N]*, you should call `subst (x, N) M`.
Also define a function `freeVars` of type `term -> string list` which returns a list of all the free variables in a term.

My solution to this problem is under 40 lines of ML code.

3. Evaluation—Reductions (difficulty ***). In this problem, you use your substitution function to implement two different evaluation strategies.

a. Implement normal-order reduction on terms. That is, write a function `reduceN : term -> term` that takes a term, performs a single reduction step (either beta or eta) in normal order, and returns a new term. If the term you are given is already in normal form, your code should raise the exception `NormalForm`, which you should define.
b. Implement applicative-order reduction on terms. That is, write a function `reduceA : term -> term` that takes a term, performs a single reduction step (either beta or eta) in applicative order, and returns a new term. If the term you are given is already in normal form, your code should raise the exception `NormalForm`, which you should reuse from the previous part.

My solution to this problem is under 20 lines of ML code.

Work with Church Numerals: Exercises to do on your own

The purpose of the problems below is to give you a little practice programming in the lambda calculus. My solutions to all three problems total less than fifteen lines of code. **You must do these exercises by yourself.**

4. Church Numerals—predecessor (difficulty **). Implement the predecessor function for the Church numerals. That is, a function `pred` such that `pred(succ n)=n`, and `pred(0)=0`. Ultimately, you will write your function in lambda notation acceptable to the lambda interpreter, but you may find it useful to try to write your initial version in Typed uScheme (or ML or uScheme) to make it easier to debug. **To get full credit, you must explain the reasoning behind your answer**.

Remember,

<church numerals>=
```
0    = \f.\x.x;
succ = \n.\f.\x.f (n f x);
+    = \n.\m.n succ m;
*    = \n.\m.n (+ m) 0;
```

You can load the initial basis with these definitions already created by typing: `use basis.lam;` in your interpreter.

Hint: Define a function `lag`, so that `lag(n,m) = (n+1,n)`. Now, what is `lag(lag(lag(lag(0,0))))`?.

5. Church Numerals—equality (difficulty *). Assume you have a predecessor function. Use it to implement an equality function on Church numerals. Again, turn in your answer using (untyped) lambda notation, but feel free to start off in uScheme or Typed uScheme if that makes it easier to debug. (Don't expect to be able to use ML here; the Hindley-Milner type system is not powerful enough to express the types of all of the functions you need to write.) **Do not use recursion or the Y-combinator in your solution.**

Remember the definitions of the Booleans:

<*>=
```
true  = \x.\y.x;
false = \x.\y.y;
```

Hints: Write a function to test if a Church numeral is zero. Consider that n = m if and only if n <= m and m <= n. Remember to use the predecessor function.

6. Church Numerals—division and modulus (difficulty ***). Write a function `divmod` such that given two Church numerals `m` and `n`, `divmod m n` returns a lambda-term representing the pair (m div n, m mod n). Again, use pure untyped lambda notation. You may use `pair`, `fst`, and `snd` as defined in class, and you will use the functions from the previous problems. You may not use explicit recursion; if you want a recursive solution, use the Y combinator.

Common mistakes, alarums, and excursions

One common mistake is to forget the eta rule:

```
\x.Mx --> M   provided x is not free in M
```

Here is an reduction in two eta steps:

```
\x.\y.cons x y --> \x.cons x --> cons
```

Your interpreters should be capable of this reduction.

If you test an interpreter before implementing reduction, you may see some alarming-looking terms that have extra lambdas and applications. This is because the interpreter uses lambda to implement the substitution that is needed for the free variables in your terms. Here's a sample:

<transcript of interpreter without reductions>=

```
-> thing = \x.\y.y x;
thing
-> thing;
(\thing.thing) \x.\y.y x
```

Everything is correct here except that the code claims something is in normal form when it isn't. If you reduce the term by hand, you should see that it has the normal form you would expect. A~term that refers to even more defined variables could start to look very exciting indeed.

Extra Credit

Solutions to any extra-credit problems should be placed in your README file.

Extra Credit—Normalization. Write a higher-order function that takes as argument a reducing strategy (e.g., `reduceA` or `reduceN`) and returns a function that normalizes a term. Your function should also count the number of reductions it takes to reach a normal form. As a tiny experiment, report the cost of computing using Church numerals in both reduction strategies. For example, you could report the number of reductions it takes to reduce ``three times four'' to normal form.

This function should be doable in about 10 lines of ML.

Extra Credit—Normal forms galore. Discover what Head Normal Form and Weak Head Normal Form are and implement reduction strategies for them. Explain, in an organized way, the differences between the four reduction strategies you have implemented.

Extra Credit—Recursive functions. Although the Y combinator cannot be typed in Hindley-Milner, it is actually fairly easy to write an explicit fixed-point operator in ML. Do so. Your operator should be called `fix` and should have the following type:

```
fix : forall 'a, 'b . (('a -> 'b) -> ('a -> 'b)) -> ('a -> 'b)
```

Use your explicit fixed-point operator to define a recursive factorial function and see if it works.

For a larger bonus, define `fix` *without* using `fun` or `val rec` (hard).

Extra Credit—Typed Equality. For extra credit, write down equality on Church numerals using Typed uScheme, give the type of the term in algebraic notation, and explain why this function can't be written in ML. (By using the ``erasure'' theorem in reverse, you can take your untyped version and just add type abstractions and type applications.)

What to submit

For this assignment,

- Please use `submit105-lambda-pair` to submit your modified `linterp.sml` for the parts on evaluation.
- Please use `submit105-lambda-solo` to submit a README file and file `church.lam` for the parts on Church Numerals.