COMP105 Assignment: Lambda Calculus

~~Due Monday, April 8, at 11:59PM.~~
Deadline extended! Due Tuesday, April 9, 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 problems below.

- There are three problems on implementing the lambda calculus, which you can do with a partner, For problems 1-3, modify the `linterp.sml` file from the git repository.
- There are four problems on programming with Church numerals, which you'll do on your own. Put your answers for to these problems, 4-7, 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 abstracts over exactly one variable.

```
  \name.term
```

Application of one term to another is written:

```
  term1 term2
```

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 operations 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 (estimated 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 *)
  forall 'a .
  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 (estimated difficulty **). Implement substitution on your `term` representation. Define 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 (estimated 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 four problems total less than fifteen lines of code. **You must do these exercises by yourself.**

4. Church Numerals—parity (estimated difficulty *). **Without using recursion or a fixed-point combinator**, define a function `even?` which, when applied to a Church numeral, returns the Church encoding of `true` or `false`, depending on whether the numeral represents an even number or an odd number.

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 uML or uScheme) to make it easier to debug.

Remember,

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

true  = \x.\y.x;
false = \x.\y.y;
```

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

5. Church Numerals—division by two (estimated difficulty ***). **Without using recursion or a fixed-point combinator**, define a function `div2` which divides a Church numeral by two (rounding down). That is `div2` applied to the numeral for *2n* returns *n*, and `div2` applied to the numeral for 2*n*â— +â— 1 also returnsÂ *n*.

Hint: For ideas, revisit the programming problems on the midterm.

6. Church Numerals—`binary` (estimated difficulty **) Implement the function `binary` from the Impcore homework. The argument and result should be Church numerals. For example,

<transcript for `binary`>=
```
-> binary 0;
\f.\x.x
-> binary 1;
\f.f
-> binary 2;
\f.\x.f (f (f (f (f (f (f (f (f (f x)))))))))
-> binary 3;
\f.\x.f (f (f (f (f (f (f (f (f (f (f x))))))))))
```

For this problem, you may use the `Y` combinator. If you do, remember to use `noreduce` when defining `binary`, e.g.,

```
  noreduce binary = ... ;
```

Note: This problem, although not so difficult, may be time-consuming. If you get bogged down, go forward to the next problem, which requires similar skills in recursion, fixed points, and Church numerals. Then come back to this problem.

EXTRA CREDIT. Write a function `binary-sym` that takes three arguments: a symbol for zero, a symbol for one, and a Church numeral. Function `binary-sym` reduces to a term that "looks like" the binary representation of the given Church numeral. Examples:

<transcript for `binary-sym`>=
```
-> binary-sym _ | 0;
_
-> binary-sym _ | 1;
|
-> binary-sym _ | 2;
| _
-> binary-sym _ | (+ 2 4);
| | _
-> binary-sym Zero One (+ 2 4);
One One Zero
```

Note: I can't identify any practical value to `binary-sym`. It is purely for showing off your new skills.

7. Church Numerals—list selection (estimated difficulty ***). Write a function `nth` such that given a Church numeral `n` and a church-encoded list `xs` of length at least `n`+1, `nth n xs` returns the `n`th element of `xs`:

```
-> 0;
\f.\x.x
-> 2;
\f.\x.f (f x)
-> nth 0 (cons Alpha (cons Bravo (cons Charlie nil)));
Alpha
-> nth 2 (cons Alpha (cons Bravo (cons Charlie nil)));
Charlie
```

If you want to define `nth` as a recursive function, use the Y combinator, and use `noreduce` to define `nth`.

Don't even try to deal with the case where `xs` is too short.

Hint: One option is to go on the web or go to Rojas and learn how to tell if a Church numeral is zero and if not, and how to take its predecessor. There are other, better options.

Common mistakes, alarums, and excursions

The lambda intepreter

Here are some common mistakes to avoid in the first part:

- Don't forget **the eta rule**:

  ```
  \x.M x --> 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 *must* eta-reduce when possible.
- Don't forget to **reduce under lambdas**.
- Don't forget to **use normal-order reduction** in the code you submit.
- **Don't clone and modify** your code for reduction strategies; too many people who do this wind up with wrong answers. The code should not be that long; just type in every case.
- Don't think that an application is in normal form just because one subterm is in normal form—you have to check **both** subterms. Most students do this using exception handlers, but if you are not comfortable with exception handlers, you can write your reducer in continuation-passing style instead.
- Don't try to be clever about a divergent term; just reduce it. (It's a common mistake to try to detect the possibility of an infinite loop. Mr Turing proved that you can't detect an infinite loop, so please don't try.)

  Do make sure to use normal-order reduction, so that you don't reduce a divergent term unnecessarily.

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.

Programming with Church numerals

Here are some common mistakes to avoid in the second part:

- Don't forget the question mark in the name of `even?`.
- When using a fixed-point combinator, don't forget `noreduce`.
- **Don't use the list representation or primitives from Wikipedia.** We will test your code using the representation and primitives described in class, which you will also find in the file `basis.lam`.
- **Don't include any `use` directives** in `church.lam`.
- **Don't copy basis functions** from `basis.lam`. We will load the basis before running your code.

  To make sure your code is well formed, load it using

  ```
  cat basis.lam church.lam | ./linterp
  ```

  If you want to build a test suite, run your tests by putting them in file `test.lam`, then running

```
cat basis.lam church.lam test.lam | ./linterp
```

More Extra Credit

If you solve the `binary-sym` extra credit, please put the solution in your `church.lam` file.

Solutions to any of the extra-credit problems below should be placed in your README file. Some may be accompanied by code in your `linterp.sml` 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—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 these files:
  - ♦ `README`
  - ♦ `church.lam` for the parts on Church Numerals

How your work will be evaluated

Your ML code will be judged by the usual criteria, emphasizing

- Correct implementation of the lambda calculus
- Good form
- Names and contracts for helper functions
- Structure that exploits standard basis functions, especially higher-order functions, and that avoids redundant case analysis

Your lambda code will be judged on correctness, form, naming, and documentation, but not so much on structure.
In particular, because the lambda calculus is such a low-level language, we will especially emphasize **names and contracts for helper functions**.

- This is low-level programming, and if you don't get your code exactly right, the only way we can recognize and reward your learning is by reading the code. It's your job to make it clear to us that even if your code isn't perfect, you understand what you're doing.
- Try to write your contracts in terms of higher-level data structures and operations. For example, even though the following function does some fancy manipulation on terms, it doesn't need much in the way of a contract:

```
double = \n.\f.\x. n (\y.f (f y)) x;  // double a Church numeral
```

Documenting lambda calculus is like documenting assembly code: it's often sufficient to say what's happening at a higher level of abstraction.

- Although it is seldom ideal, it can be OK to use higher-level code to document your lambda code. In particular, if you want to use Scheme or ML to explain what your lambda code is doing, this can work only because Scheme and ML operate at much higher levels of abstraction. Don't fall into the trap of writing the *same* code twice—if you are going to use code in a contract, it **must** operate at a significantly higher level of abstraction than the code it is trying to document.

In more detail, here are our criteria:

	Exemplary	**Satisfactory**	**Must improve**
Naming	• Each Î»-calculus function is named either with a noun describing the result it returns, or with a verb describing the action it does to its argument, or (if a predicate) as a property with a question mark.	• Functions' names contain appropriate nouns and verbs, but the names are more complex than needed to convey the function's meaning. • Functions' names contain some suitable nouns and verbs, but they don't convey enough information about what the function returns or does.	• Function's names include verbs that are too generic, like "calculate", "process", "get", "find", or "check" • Auxiliary functions are given names that don't state their contracts, but that instead indicate a vague relationship with another function. Often such names are formed by combining the name of the other function with a suffix such as `aux`, `helper`, `1`, or even `_`. • Course staff cannot identify the connection between a function's name and what it returns or what it does.
Documentation	• The contract of each function is clear from the function's name, the names of its parameters, and perhaps a one-line comment describing the result. • *Or*, when names alone are not enough, each function's contract is documented with a type (in a comment) • *Or*, when names and a type are not enough, each function's contract is documented by writing the function's operation in a high-level language with high-level data structures. • *Or*, when a function cannot be explained at a high level, each function is documented with a meticulous contract that explains what Î»-calulucs term the function returns, in terms of the parameters, which are mentioned by name. • All recursive functions use structural recursion and therefore don't need documentation.	• A function's contract omits some parameters. • A function's documentation mentions every parameter, but does not specify a contract. • A recursive function is accompanied by an argument about termination, but course staff have trouble following the argument.	• A function is not named after the thing it returns, and the function's documentation does not say what it returns. • A function's documentation includes a narrative description of what happens in the body of the function, instead of a contract that mentions only the parameters and result. • A function's documentation neither specifies a contract nor mentions every parameter. • A function is documented at a low level (Î»-calculus terms) when higher-level documentation (pairs, lists, Booleans, natural numbers) is possible. • There are multiple functions that are not part of the specification of the problem, and from looking just at the names of the functions and the names of their parameters, it's hard for us to figure out what the functions do. • A recursive function is accompanied by an argument about termination, but

	• *Or*, every function that does *not* use structural recursion is documented with a short argument that explains why it terminates.		course staff believe the argument is wrong. • A recursive function does not use structural recursion, and course staff cannot find an explanation of why it terminates.
Structure	• File `church.lam` loads without errors. • The code for reduction contains no redundant case analysis. • Where appropriate, ML code uses `map`, `List.filter`, `List.exists`, `List.all`, and `foldl` or `foldr`.	• File `church.lam` loads with an I/O error and/or possibly some divergent terms. • Some code for reduction contains redundant case analysis. • ML code uses standard higher-order functions, but some opportunities for using folds have been overlooked.	• File `church.lam` loads with an undefined identifier on the RHS or with some other error. • Both normal-order and applicative-order reduction contain redundant case analysis. • ML code defines recursive helper functions that could have been defined non-recursively using `map`, `filter`, `exists`, `all`, and so on.
Form	• The code has no offside violations. • *Or*, the code has just a couple of minor offside violations. • Indentation is consistent everywhere. • The submission has no bracket faults. • The submission has a few minor bracket faults. • *Or*, the submission has no bracketed names, but a few bracketed conditions or other faults.	• The code has several offside violations, but course staff can follow what's going on without difficulty. • In one or two places, code is not indented in the same way as structurally similar code elsewhere. • The submission has some redundant parentheses around function applications that are under infix operators (not checked by the bracketing tool) • *Or*, the submission contains a handful of bracketing faults. • *Or*, the submission contains more than a handful of bracketing faults, but just a few bracketed names or conditions.	• Offside violations make it hard for course staff to follow the code. • The code is not indented consistently. • The submission contains more than a handful of parenthesized names as in `(x)` • The submission contains more than a handful of parenthesized `if` conditions.