Lambda Calculus

COMP 105 Assignment

Due Tuesday, April 11, 2017 at 11:59PM

Contents

Overview

This assignment will give you practice with the lambda calculus:

- Using lambda calculus to write simple functions
- Implementing lambda calculus using substitution, reduction, and alpha-conversion

Substitution, reduction, and alpha-conversion are ubiquitous ideas in programming-language semantics.

Setup

You will build on an existing interpreter for lambda calculus, which you will get by cloning the following git repository:

```
git clone linux.cs.tufts.edu:/comp/105/git/lambda
```

Cloning should give you a directory `lambda` with files `linterp.sml`, `Lhelp.ui`, `Lhelp.uo`, `Makefile`, and `predefined.lam`.

Learning about the lambda calculus

There is no book chapter on the lambda calculus. Instead, we refer you to these resources:

1. Raúl Rojas's "A Tutorial Introduction to the Lambda Calculus[1]" is short, easy to read, and covers the same points that are covered in lecture:

   - Syntax
   - Free and bound variables
   - Capture-avoiding substitution
   - Addition and multiplication with Church numerals
   - Church encoding of Booleans and conditions
   - The predecessor function on Church numerals
   - Recursion using the Y combinator

   Rojas doesn't provide many details, but he covers everything you need to know in 8 pages, with no distracting theorems or proofs. When you want a short, easy overview to help you solidify your understanding, Rojas's tutorial is the best source.

2. Wikipedia offers two useful pages:[2]

   - The Lambda Calculus[3] page covers everything you'll find in Rojas and much more besides. In particular, it discusses reduction strategies.
   - The Church Encoding[4] page goes into more detail about how to represent ordinary data as terms in the lambda calculus. The primary benefit relative to Rojas is that Wikipedia describes more kinds of arithmetic and other functions on Church numerals.

[1] http://www.cs.tufts.edu/comp/105/readings/rojas.pdf
[2] At least, they looked useful as of March 2017. As always, Wikipedia pages are subject to change without notice.
[3] https://en.wikipedia.org/wiki/Lambda_calculus
[4] https://en.wikipedia.org/wiki/Church_encoding

You need to know that **the list encoding used on Wikipedia is not the list encoding used in COMP 105**. In order to complete all the homework problems successfully, **you must use the list encoding described in the lecture notes.**

When you want a quick, easily searchable reference to some particular point, Wikipedia is your best source. Wikipedia is particularly useful for explaining the difference between normal-order reduction and applicative-order reduction, both of which you will implement.

3. Prakash Panangaden's "Notes on the Lambda-Calculus[5]" cover the same material but with more precision and detail. Prakash is particularly good on capture-avoiding substitution and change of bound variables, which you will implement.

   Prakash also discusses more theoretical ideas, such as how you might prove inequality (or inequivalence) of lambda-terms. And instead of just presenting the Y combinator, Prakash goes deep into the ideas of fixed points and solving recursion equations—which is how you achieve recursion in lambda calculus.

   When you are getting ready to implement substitution and reduction strategies, Prakash's notes are your best source.

Introduction to the lambda interpreter

You will implement the key components of a small, interactive interpreter for the lambda calculus. This section explains how to use the interpreter and the syntax it expects. A reference implementation of the interpreter is available in `/comp/105/bin/linterp-nr`.

Syntax

The syntax of definitions

Like the interpreters in the book, the lambda interpreter processes a sequence of definitions.
The concrete syntax is very different from the book languages. Every definition must be terminated with a semicolon. Comments are line comments in C++ style, starting with the string `//` and ending at the next newline.

The interpreter supports four forms of definition: a binding, a term, the extended definition "`use`", and an extended definition "`check-equiv`".

Bindings

A binding has the form

```
-> noreduce name = term;
```

or

```
-> name = term;
```

[5] http://www.cs.tufts.edu/comp/105/readings/prakash.pdf

In both forms, every free variable in the term must be bound in the environment—if a right-hand side contains an unbound free variable, the result is a checked run-time error. The first step of computation is to substitute for each of the free variables: each occurrence of each free variable is replaced by that variable's definition.

In the first form, where `noreduce` appears, no further computation takes place. The substituted right-hand side is simply associated with the name on the left, and this binding is added to the environment.

The `noreduce` form is intended only for terms that cannot be normalized, such as

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

In the second form, after the free variables are replaced, the term on the right is reduced until there are no more beta-redexes or eta-redexes. (You will implement the two reduction strategies presented in class.) If reduction doesn't terminate, the interpreter might loop.

Loading files with `use`

The `use` extended definition loads a file into the interpreter as if it had been typed in directly. It takes the form

```
-> use filename;
```

Comparing normal forms with `check-equiv`

The `check-equiv` form **immediately** reduces two terms to normal form and compares them for equivalence. It has the form

```
-> check-equiv term = term;
```

And here are some examples:

```
-> check-equiv x = x;
The test passed
-> check-equiv \x.x = \y.y;
The test passed
-> check-equiv \x.x = \y.x;
The test failed: terms \x.x and \y.x do not have equivalent normal forms
-> check-equiv (\x.x)(\y.y) = \z.z;
The test passed
```

Unlike the `check-expect` in the other interpreters, `check-equiv` is *not* "saved for later"—the given terms are normalized right away.

Terms as definitions

As in the book, a term can be entered at the read-eval-print loop, just as if it were a definition. Every free variable in the term is checked to see if it is bound in the environment; if so, each free occurrence is replaced by its binding. Free variables that are not bound in the environment are permissible; they are left alone.[6] The term is reduced to normal form (if possible) and the result is printed.

[6] Try, for example, `(\x.\y.x) A B;`.

```
-> term;
```

The syntax of terms

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 abstracts over exactly one variable; it is written as follows:

```
\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. But a name made up entirely of digits is OK; the lambda calculus has no numbers, and names like `105` have no special status.

As examples, all the following definitions are 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 `predefined.lam`[7] file distributed with the assignment.

[7] ./predefined.lam

Software provided for you

Both capture-avoiding substitution and normal-order reduction can be tricky to implement.[8] So that you may have a solid foundation on which to write your lambda code, I provide an interpreter `linterp-nr`. Running `use comp105` should give you access to that interpreter.

Even with a correct interpreter, lambda code can be hard to debug. So I also provide an interpreter called `lamstep`, which shows every reduction step. Some reductions require a *lot* of steps and produce very big intermediate terms. Don't be alarmed.

All questions and problems

- There are four problems on programming with Church numerals, which you'll do on your own.
- There are four problems on implementing the lambda calculus, which you can do with a partner. Modify the `linterp.sml` file from the git repository described under the heading Setup, above.

Reading comprehension

These problems will help guide you through the reading. We recommend that you complete them before starting the other problems below. You can download the questions[9].

1. In this assignment, or in Rojas or Panangaden, read about the concrete syntax of lambda-terms. Now define, in Standard ML, an algebraic data type `term` that represents the *abstract* syntax of terms. Your data type should have one value constructor for a variable, one for a lambda abstraction, and one for an application.

   You are ready for exercise 5, and you have a foundation for exercises 6 and 8.

2. First read about reduction[10] on Wikipedia. Then in Panangaden[11], be sure you have an idea about each of these concepts:

   - Capture-avoiding *substitution* (Definition 1.3)
   - *Reduction* (Definition 1.5), including the example reduction (Example 1.3)
   - *Redex*, *contractum*, and *normal form* (Definitions 1.7 and 1.8)

   Showing each reduction step, reduce the following term to normal form. At each step, choose a redex and replace the redex with its contractum.

   ```
   (\n.(n(\z.T))F)(\f.\x.f x)
   →
   …
   ```

   The term contains more than one redex, but no matter which redex you choose at each step, you should reach the normal form after exactly four reductions.

[8] Over the course of my career, I have botched capture-avoiding substitution multiple times.
[9] ./cqs.lambda.txt
[10] https://en.wikipedia.org/wiki/Lambda_calculus#Reduction
[11] http://www.cs.tufts.edu/comp/105/readings/prakash.pdf

You are preparing to complete exercise 8. But first, you will need an implementation of substitution.

3. Read about capture-avoiding substitutions[12]. Review the algebraic laws from exercise 6 and their side conditions. (The same laws are presented by Prakash Panangaden[13], using different notation, as Definition 1.3.)

   If you want another example of variable capture, read about syntactic sugar for && in μScheme (Ramsey, Section 2.16.3, which starts on page 180), and read about substitution in Ramsey, Section 2.16.4.

   The lambda term `\x.\y.x` represents a (Curried) function of two arguments that returns its first argument. We expect the application `(\x.\y.x)y z` to return `y`.

   (a) Take the redex `(\x.\y.x)y` and reduce it one step, **ignoring the side conditions that prevent variable capture**. That is, substitute *incorrectly*, without renaming any variables.

   If you substitute incorrectly in this way, what term do you wind up with?

   (b) The final term in part (a) codes for a function. In informal English, how would you describe that function?

   (c) Now repeat part (a), but this time, renaming variables as needed to avoid capture during substitution.

   After a correct reduction with a correct substitution, what term do you wind up with?

   (d) The final term in part (c) codes for a function. In informal English, how would you describe that function?

   You are ready for exercise 6 (substitution).

4. Read about redexes in Wikipedia[14]. If you have read Panangaden, Definition 1.7, be aware that Panangaden mentions only one kind of redex, but you will be implementing two.

   (a) Name the two kinds of redex.

   (b) For each kind of redex, use the concrete syntax defined above, to show what form all redexes of that kind take.

   (c) For each kind of redex, use your algebraic data type from the preceding question to write a pattern that matches every redex of that kind.

   You are getting ready for exercise 8 (reductions).

5. Here's another question about redexes and reduction[15]. For each kind of redex, show the general form of the redex from part (b) of the preceding question, and show what syntactic form the redex reduces to (in just a single reduction step).

   You are getting ready for exercise 8 (reductions).

[12] https://en.wikipedia.org/wiki/Lambda_calculus#Capture-avoiding_substitutions
[13] http://www.cs.tufts.edu/comp/105/readings/prakash.pdf
[14] https://en.wikipedia.org/wiki/Lambda_calculus#Reduction
[15] https://en.wikipedia.org/wiki/Lambda_calculus#Reduction

6. Read about normal-order and applicative-order reduction strategies[16]. Using the concrete syntax defined above, write a lambda term that contains exactly two redexes, such that *normal-order* reduction strategy reduces one redex, and *applicative-order* reduction strategy reduces the other redex.

   You are (finally!) ready for exercise 8.

Programming in the lambda calculus (individual problems)

These problems give you a little practice programming in the lambda calculus. **All functions must terminate in linear time**, and **you must do these exercises by yourself.** You can use the reference interpreter `linterp-nr`.

Place your solutions in file `church.lam`.

Not counting code copied from the lecture notes, my solutions to all four problems total less than fifteen lines of code. And all four problems rely on the same related reading.

Related reading for lambda-calculus programming problems 1 to 4:

- For the basics, consult Wikipedia on Church Encoding[17] and section 2 of Panangaden[18], which is titled "Computing with Lambda Calculus" (from page 8 to the middle of page 10). These basics are sufficient for you to tackle problems 1 and 2.
- On problems 3 and 4 only, if you have the urge to write a recursive function, you may use a fixed-point combinator. You must understand the first paragraph under "Fixed-Point Combinators" on page 10 of Panangaden[19]. This explanation is by far the best and simplest explanation available—but it is very terse. For additional help, consult the examples on page 11.

  Another option around recursion is the section on "Recursion and fixed points[20]" in the Wikipedia article about lambda calculus. I find this section a little hard to follow, but I recommend it because it shows a step-by-step transition from a recursive function we would like to write to a real lambda-calculus term defined using a fixed-point combinator. I recommend **against** the Wikipedia "main article" on fixed-point combinators: the article is all math all the time, and it won't give you any insight into how to *use* a fixed-point combinator.

1. *Church Numerals—parity*. **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.

Your function must terminate in time linear in the size of the Church numeral.

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 μScheme (or ML or μML or μScheme) to make it easier to debug.

Remember these basic terms for encoding Church numerals and Booleans:

[16] https://en.wikipedia.org/wiki/Lambda_calculus#Reduction_strategies
[17] https://en.wikipedia.org/wiki/Church_encoding
[18] http://www.cs.tufts.edu/comp/105/readings/prakash.pdf
[19] http://www.cs.tufts.edu/comp/105/readings/prakash.pdf
[20] https://en.wikipedia.org/wiki/Lambda_calculus#Recursion_and_fixed_points

```
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 these definitions by typing `use predefined.lam;` in your interpreter.

2. *Church Numerals—division by two*. **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 $2n + 1$ also returns n.

Your function must terminate in time linear in the size of the Church numeral.

Hint: Think about function `split-list` from the Scheme homework[21].

3. *Church Numerals—conversion to binary.* Implement the function `binary` from the Impcore homework[22]. The argument and result must be Church numerals. For example,

```
-> binary 0;
\f.\x.x
-> binary 1;
\f.f
-> binary 2;
\f.\x.f (f (f (f (f (f (f (f (f x)))))))) // f applied 10 times
-> binary 3;
\f.\x.f (f (f (f (f (f (f (f (f (f x))))))))) // f applied 11 times
```

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

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

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.

Your function must terminate in time linear in the size of the Church numeral.

EXTRA CREDIT. Write a function `binary-sym` that takes three arguments: a name for zero, a name for one, and a Church numeral. Function `binary-sym` reduces to a term that "looks like" the binary representation of the given Church numeral. Here are some examples where I represent a zero by a capital `O` (oh) and a one by a lower-case `l` (ell):

```
-> binary-sym O l 0;
O
-> binary-sym O l 1;
l
-> binary-sym O l 2;
l O
```

[21] ./scheme.html
[22] ./impcore.html

```
-> binary-sym O l (+ 2 4);
l l O
-> binary-sym Zero One (+ 2 4);
One One Zero
-> binary-sym O l (+ 1 (* 4 (+ 1 2)));
l l O l
```

It may help to realize that `l l O l` is the application `(((l l) O) l)`—it is just like the example at the bottom of the first page of Rojas's tutorial[23].

Function `binary-sym` has little practical value, but it's fun. If you write it, please put it in your `church.lam` file, and mention it in your `README` file.

4. *Church Numerals—list selection*. 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`.

Provided `xs` is long enough, function `nth` must terminate in time linear in the length of the list. 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[24] 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.

Implementing the lambda calculus (possibly with a partner)

For problems 5 to 7 below, you may work on your own or with a partner. These problems help you learn about substitution and reduction, the fundamental operations of the lambda calculus. The problems also give you a little more practice in continuation passing, which is an essential technique in lambda-land.

For each problem, 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. Typing `make` should do it.

5. *Evaluation—Basics*. This problem has three 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        *)
```

[23] http://www.cs.tufts.edu/comp/105/readings/rojas.pdf
[24] http://www.cs.tufts.edu/comp/105/readings/rojas.pdf

```
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
```

My solution to this problem is under 15 lines of ML code.

(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.

(c) In file `string-tests.sml`, submit three test cases using `assert`. Here are some **updated updated** examples:

```
val _ = assert (toString (lam "x" (var "x")) = "(lambda (x) x)")
val _ = assert (toString (app (app (var "map") (var "f")) (var "xs")) =
        "((map f) xs)")
val _ = assert (toString (app (app (lam "x" (lam "y" (var "x"))) (var "1")) (var "2")) =
                "(((lambda (x) (lambda (y) x)) 1) 2)")
```

My solution is under 30 lines of ML code.

Related reading: The syntax of lambda terms[25] in this homework.

6. *Evaluation—Substitution*. Implement capture-avoiding substitution on your `term` representation. In particular,

- Define a function `subst` of type `string * term -> term -> term`. Calling `subst (`x`, `N`) M` returns the term $M[x \mapsto N]$.

  Function `subst` obeys these algebraic laws:[26]

  (a) `subst (x, N) x = N`
  (b) `subst (x, N) y = y`, provided `y` is different from `x`
  (c) `subst (x, N)` $(M_1\ M_2)$ = (`subst (x, N)` M_1) (`subst (x, N)` M_2)
  (d) `subst (x, N)` $(\lambda$x.M$) = (\lambda$x.M$)$
  (e) `subst (x, N)` $(\lambda$y.M$) = \lambda$y.(`subst (x, N)` M), provided `x` is not free in `M` or `y` is not free in `N`, and also provided `y` is different from `x`

  If none of the cases above apply, then `subst (x, n) M` should return `subst (x, n) M'`, where `M'` is a term that is obtained from `M` by renaming bound variables. Renaming a bound variable is called "alpha conversion."

[25] http://www.cs.tufts.edu/comp/105/homework/lambda.html#the-syntax-of-terms
[26] The laws, although notated differently, are identical to the laws given by Prakash Panangaden[27] as Definition 1.3.

You will need to rename bound variables only if you encounter a case that is like case (e), but in which `x` is free in `M` and `y` is free in `N`. In such a case, `subst (x, N)` $(\lambda y.M)$ can be calculated only by renaming `y`, which is bound in the lambda abstraction, to some new variable that is not free in `M` or `N`.

To help you implement `subst`, you may find it useful to define these helper functions:

- Function `freeIn`, which tells if a given variable occurs free in a given term
- Function `freeVars`, which produces a list of the variables free in a given term
- Function `freshVar`, which is given a list of variables and produces a variable that is different from every variable on the list

By using `freshVar` on the output of `freeVars`, you will be able to implement alpha conversion.

Define functions `subst`, `freeIn` and `freeVars` using `cpsLambda`.

When you test your interpreter after this problem, you may see some alarming-looking terms that have extra lambdas and applications. This is because the interpreter uses lambda to substitute for the free variables in your terms. Here's a sample:

```
-> 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.

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

Related reading:

- Panangaden[28] describes free and bound variables in Definition 1.2 on page 2. He defines substitution in Definition 1.3 on page 3. (His notation is a little different from our ML code, but the laws for `subst` are the same.)
- In his Definition 1.3, case 6, plus Definition 1.4, Panangaden[29] explains the "change of bound variables" that you need to implement if none of the cases for `subst` apply.
- Page 456 of your book defines an ML function `freshName` which is similar to the function `freshVar` that you need to implement. The `freshName` on page 456 uses an infinite stream of candidate variables. You could copy all the stream code from the book, but it will probably be simpler just to define a tail-recursive function that tries an unbounded number of variables.

  You can also study similar code on pages 1376 and 1377.

  Don't emulate function `freshtyvar` on page 503. It's good enough for type inference, but it's not good enough to guarantee freshness in the lambda calculus.

7. *Substitution tests*. As shown in the previous problem, function `subst` has to handle five different cases correctly. It also has to handle a sixth case, in which none of the laws shown above applies, and

[28] http://www.cs.tufts.edu/comp/105/readings/prakash.pdf
[29] http://www.cs.tufts.edu/comp/105/readings/prakash.pdf

renaming is required. In this problem, you create test cases for your `subst` function. They should look like this:

```
exception MissingTest of string
val N : term = app (app (var "fst") (var "x")) (var "y")
val test_a = subst ("x", N) (var "x") = N
val test_b = raise MissingTest "(b)"
val test_c = raise MissingTest "(c)"
val test_d = raise MissingTest "(d)"
val test_e = raise MissingTest "(e)"
val test_renaming = raise MissingTest "renaming"

val _ = ListPair.app
        (fn (t, name) => if t then () else print ("BAD TEST (" ^ name ^ ")\n"))
        ( [test_a, test_b, test_c, test_d, test_e, test_renaming]
        , ["a", "b", "c", "d", "e", "renaming"]
        )
```

To test substitution, complete these steps:

(a) Put the code above in your `linterp.sml` where it says COMMAND LINE, just above the definition of function `main`.

(b) Replace every line that raises the `MissingTest` exception with a test for the appropriate case.

(c) Remove the definition of the `MissingTest` exception.

(d) Verify that all tests are good by compiling the interpreter and running

```
./linterp < /dev/null
```

8. *Evaluation—Reductions*. In this problem, use your substitution function to implement two different reduction 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.

For debugging purposes, here is a way to print a status report after every `n` reductions.

```
fun tick show n f =  (* show info about term every n reductions *)
  let val count = ref 0
      fun apply arg =
        let val _ = if !count = 0 then
                      ( List.app print ["[", Int.toString (show arg), "] "]
                      ; TextIO.flushOut TextIO.stdOut
                      ; count := n - 1)
                    else
```

```
                    count := !count - 1
      in  f arg
      end
in  apply
end
```

I have defined a status function `size` that prints the size of a term. You can print whatever you like: a term's size, the term itself, and so on. Here is how I show the size of the term after every reduction. Some "reductions" make terms bigger!

```
val reduceN_debug = tick size 1 reduceN  (* show size after every reduction *)
```

My solution to this problem is under 20 lines of ML code, not counting my `size` function, which is another 3 lines.

Related reading:

- The simplest source on reduction is probably the lecture notes.
- Panangaden[30] describes the reduction relation in Definition 1.5. Although he treats it as a mathematical relation, not a computational rule, you may find his definitions helpful. But some commentary is required:
  - Rules α (change of variables) and ρ (reflexivity) have no computational content and should therefore play no part in `reduceN` or `reduceA`. (Rule α plays a part in `subst`.)
  - Rule τ (transitivity) involves multiple reductions and therefore also plays no part in `reduceN` or `reduceA`.

  The remaining rules are used in both `reduceN` and `reduceA`, but with different priorities.
  - Rule β is the key rule, and in normal-order reduction, rule β is always preferred.
  - In applicative-order reduction, rule μ (reduction in the argument position) is preferred.
  - In normal-order reduction, rule ν (reduction in the function position) is preferred over rule μ but not over rule β.

  Finally, Panangaden omits rule η, which like rule β is always preferred:
  - $\lambda x.Mx \to M$, provided x is not free in M

  You must implement the η rule as well as the other rules.
- Wikipedia describes some individual reduction rules in the Reduction[31] section of the lambda-calculus page. And it briefly describes applicative-order reduction and normal-order reduction, as well as several other reduction strategies, in the reduction strategies[32] section of the lambda-calculus page.

[30] http://www.cs.tufts.edu/comp/105/readings/prakash.pdf
[31] https://en.wikipedia.org/wiki/Lambda_calculus#Reduction
[32] https://en.wikipedia.org/wiki/Lambda_calculus#Reduction_strategies

More Extra Credit

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 and how to submit: Individual work

Using script `submit105-lambda-solo`, submit

- A `README` file containing
  - The names of the people with whom you collaborated
  - Any extra credit you may have earned
- File `cqs.lambda.txt`, containing your answers to the reading-comprehension questions
- File `church.lam` containing your solutions to the Church-numeral

As soon as you have the files listed above, run `submit105-lambda-solo` to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

What and how to submit: Pair work

Using script `submit105-lambda-pair`, submit

- File `linterp.sml`, containing your modified lambda-calculus interpreter and your tests of substitution
- File `string-tests.sml`, containing your tests of `var`, `app`, `lam`, and `toString`

As soon as you have the files listed above, run `submit105-lambda-pair` to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

Avoid common mistakes

Common mistakes with Church numerals

Here are some common mistakes to avoid when programming with Church numerals:

- 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 `predefined.lam`[33].
- **Don't include any `use` directives** in `church.lam`.
- **Don't copy predefined terms** from `predefined.lam`. We will load the predefined terms before running your code.

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

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

  or

  ```
  cat predefined.lam church.lam | linterp-nr
  ```

  If you want to build a test suite, put your tests in file `test.lam` and run

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

Common mistakes with the lambda interpreter

Here are some common mistakes to avoid in implementing the interpreter:

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

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

  Here is a 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 forget that in an application $M_1\ M_2$, just because M_1 is in normal form doesn't mean the whole thing is in normal form. In particular, if reducing M_1 raises the `NormalForm` exception, **you must catch the exception** and try to reduce M_2.

[33] ./predefined.lam

- **Don't clone and modify** your code for reduction strategies; people who do this wind up with wrong answers. The code should not be that long; use a clausal definition with nested patterns, and write every case from scratch.

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

- 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.)

- When implementing `freshVar`, don't try to repurpose function `freshTyvar` from Section 7.6. That function isn't smart enough for your needs.

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 for names:

	Exemplary	**Satisfactory**	**Must Improve**
Naming	● Each λ-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[34], 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.

And here are our criteria for contracts:

[34] http://www.cs.tufts.edu/comp/105/coding-style.html#contracts

	Exemplary	**Satisfactory**	**Must Improve**
Documentation	• The contract[35] 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[36] that explains what λ-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. • *Or*, every function that does *not* use structural recursion is documented with a short argument that explains why it terminates.	• A function's contract[37] omits some parameters. • A function's documentation mentions every parameter, but does not specify a contract[38]. • 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[39] 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 (λ-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 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.

[35] http://www.cs.tufts.edu/comp/105/coding-style.html#contracts
[36] http://www.cs.tufts.edu/comp/105/coding-style.html#contracts

Exemplary	Satisfactory	Must Improve

[37] http://www.cs.tufts.edu/comp/105/coding-style.html#contracts
[38] http://www.cs.tufts.edu/comp/105/coding-style.html#contracts
[39] http://www.cs.tufts.edu/comp/105/coding-style.html#contracts