Lambda Calculus

COMP 105 Assignment

Due Wednesday, April 17, 2019 at 11:59PM

Contents

Overview

Lambda calculus is not just a universal model of computation—it is also a language you can use to communicate with educated people around the world. In this assignment,

- You use lambda calculus to write simple functions
- You implement lambda calculus using substitution, reduction, and alpha-conversion

Substitution, reduction, and alpha-conversion are found all over programming-language semantics, not just in lambda calculus.

Setup

TL;DR: download the template solution[1] and compile with `compile105-lambda`. Everything will be fine.

Behind the curtain: For the first part, coding in lambda calculus, you will code things from scratch. For the second part, implementing lambda calculus, you will extend an interpreter I've written. But because you can work with ML modules now, you won't be stuck modifying a huge pile of code. Instead, you'll define several modules, for both implementation and testing, and you'll use several of my interfaces.

The ML module system is nice, but Moscow ML's module bureaucracy is not at all nice. I've hidden the bureaucracy behind a shell script, `compile105-lambda`. This script lives in `/comp/105/bin`, and if you run `use comp105` at the command line, you have access to it. But if something goes wrong, you may wish to know about the pieces of the assignment. Here are the source codes:[2]

`church.lam`	Your solutions to the first part
`solution.sml`	Your module implementing terms, substitution, and reduction
~~`client.sml`~~	~~Your module demonstrating term functions~~
~~`string-tests.sml`~~	~~Test cases for your classmates' code~~
`subst-tests.sml`	Test cases for substitution
`link-lambda.sml`	Instructions for linking your code with mine
`link-lambda-a.sml`	More instructions for linking your code with mine
`link-lamstep.sml`	Even more instructions for linking your code with mine

Using these sources, the `compile105-lambda` script will create binaries:

`./run-solution-unit-tests`	Runs some of your unit tests
~~`./run-client-unit-tests`~~	~~Runs more unit tests~~
~~`./run-string-tests`~~	~~Runs more unit tests~~
`./run-subst-tests`	Runs the last of your unit tests
`./linterp`	Runs your complete interpreter (normal-order reduction)
`./lamstep`	Runs your interpreter, showing each reduction
`./linterp-applicative`	Runs your complete interpreter (applicative-order reduction)

[1] ./solution.sml

[2] Files `link-lambda.sml` and `link-lambda-a.sml` are copied into your directory by the `compile105-lambda` script. The others are created by you.

Learning about the lambda calculus

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

1. The edited version of Raúl Rojas's "A Tutorial Introduction to the Lambda Calculus[3]" 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 9 pages, with no distracting theorems or proofs.

   When you want a short, easy overview to help you solidify your understanding, Rojas's tutorial is your best source.

2. I have written a short guide to coding in Lambda calculus[4]. It shows how to translate ML-like functions and data, which you already know how to program with, into lambda calculus.

   When you are solving the individual programming problems, this guide is your best source.

3. Prakash Panangaden's "Notes on the Lambda-Calculus[5]" cover the same material as Rojas, 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, Prakash's notes are your best source.

4. I have also written a short guide to reduction strategies[6]. It is more useful than anything that could be found online in 2018. As a bonus, it also explains *eta-reduction*, which is neglected by other sources.

   When you have finished implementing substitution and are ready to implement reduction, this guide is your best source.

5. Wikipedia offers two somewhat useful pages:[7]

[3] https://www.cs.tufts.edu/comp/105/readings/rojas.pdf
[4] ../handouts/lambda-coding.pdf
[5] https://www.cs.tufts.edu/comp/105/readings/prakash.pdf
[6] https://www.cs.tufts.edu/comp/105/readings/reduction.pdf
[7] They were more useful in 2017 then they are now—as always, Wikipedia pages are subject to change without notice.

- The Lambda Calculus[8] page covers everything you'll find in Rojas and much more besides. (If you wish, you can read what Wikipedia says about reduction strategies and evaluation strategies. But do not expect to be enlightened.)
- The Church Encoding[9] 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.

You need to know that **the list encoding used on Wikipedia is not the list encoding used in COMP 105**. In order to complete the homework problems successfully, **you must use the list encoding described in the guide to coding in lambda calculus[10].**

Introduction to the lambda interpreter

You will implement the key components of an 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 "bridge languages" in the book. 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 is something like a `val` form in μScheme. A binding has one of two forms: either

```
noreduce name = term;
```

or

```
name = term;
```

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

[8] https://en.wikipedia.org/wiki/Lambda_calculus
[9] https://en.wikipedia.org/wiki/Church_encoding
[10] ../handouts/lambda-coding.pdf

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

The `noreduce` form is also needed for definitions that *use* terms like `bot` and `Y`.

If `noreduce` is absent, the interpreter substitutes for free variables, then reduces the term on the right 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
-> check-equiv \f.f = \f.\x.f x;
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.[11] The term is reduced to normal form (if possible) and the result is printed.

```
-> 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.

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

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;
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`[12] file distributed with the assignment.

Software provided for you

Both capture-avoiding substitution and normal-order reduction can be tricky to implement.[13] 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.

[12] https://www.cs.tufts.edu/comp/105/homework/predefined.lam

[13] I have botched capture-avoiding substitution multiple times.

Even with a correct interpreter, lambda code can be hard to debug. So I also provide an interpreter called `lamstep-nr`, which shows every reduction step. Some computations require a *lot* of reduction steps and produce 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. Your solutions will go into a Standard ML module, which you will link with the rest of the interpreter.

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[14].

1. (NOT ON THE READING.) Throughout the term, your code's functional correctness has been assessed by automated testing. The automated test scripts are intended not only to assign a grade but to identify the most important fault in the code. Please answer these two questions:

   (a) How did you benefit from the feedback you received about functional correctness?

   (b) What were the drawbacks, if any, of the feedback you received about functional correctness?

2. *Syntax of lambda terms*. 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.

3. *Recognizing redexes*. Read about redexes in Wikipedia[15]. (You will then follow up with Panangaden[16].)

   (a) Wikipedia mentions two kinds of redex. What are their names?

   (b) In Panangaden, Definition 1.7 defines a redex. Which of the two redexes mentioned in Wikipedia is being defined here?

   Your code will have to recognize redexes, and it starts with knowing the form of each kind. As of Spring 2019, both forms are shown in Wikipedia. But if Wikipedia changes, one form can be found in Panangaden; for the other, look in the last section of my guide to reduction strategies.

   (c) For each of the two kinds of redex, use the concrete syntax for our lambda interpreter (see above) to show what form every redex of that kind takes.

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

[14] ./cqs.lambda.txt
[15] https://en.wikipedia.org/wiki/Lambda_calculus#Reduction
[16] https://www.cs.tufts.edu/comp/105/readings/prakash.pdf

You are getting ready for exercise 8 (reductions).

4. *Practicing reduction*. Read about reduction[17] on Wikipedia. Then in Panangaden[18], 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. Do not expand or replace the names `ZERO` and `NONZERO`.

   ```
   (\n.(n(\z.NONZERO))ZERO)(\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.

   You are preparing to complete exercise 8.

5. *Reduction: the general case*. For each kind of redex, repeat the general form of the redex from question ~~2(c)~~ 3(c) above, then show what syntactic form the redex reduces to (in just a single reduction step).

   You are getting ready for exercise 8 (reductions).

6. *When to reduce*. Read my handout on reduction strategies[19]. Using the concrete syntax accepted by the interpreter (and 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.

7. *Understanding Church numerals*. You may recognize the practice reduction above as a computation that tells if a Church numeral is zero. Read about Church numerals, either on pages 9 and 10 of Panangaden or in Section 2 of Rojas ("Arithmetic"). Then, say whether each of the following lambda-calculus terms is a Church numeral. If so, write the corresponding decimal representation. If not, write "not a Church numeral".

   ```
   \f.x
   \f.\x.x
   \f.\x.f
   \f.\x.f x
   \x.\x.f (f (f (f (f x))))
   ```

   You are ready for exercises 1 to 4.

[17] https://en.wikipedia.org/wiki/Lambda_calculus#Reduction
[18] https://www.cs.tufts.edu/comp/105/readings/prakash.pdf
[19] ../readings/reduction.pdf

Programming in the lambda calculus (individual problems)

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

Lambda-calculus programs work at the same intellectual level as assembly-language programs. Therefore, **every new helper function must be well named and must be accompanied by a contract.** Detailed guidance can be found below.

Helper functions listed in the assignment are exempt from the contract requirement, as are the helper functions in `predefined.lam`.

Complete all four problems below, and 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:

- My guide Coding in Lambda Calculus[20] should explain everything you need to know to write functional programs in lambda calculus. If not, or if the explanations there are a little too terse, consult the additional readings below.
- Basic techniques can be found in Wikipedia on Church Encoding[21] and in section 2 of Panangaden[22], 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.

  Another alternative is Section 2 of Rojas's tutorial, entitled "arithmetic." Rojas doesn't mention Church numerals by name, but that's what he's working with. You may find the examples useful and the presentation more accessible than what you see from Panangaden.
- On problems 3 and 4 only, if you have the urge to write a recursive function, you may use a fixed-point combinator. My guide ends with a few pages on recursion. You may also wish to consult the first paragraph under "Fixed-Point Combinators" on page 10 of Panangaden[23]. 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.

  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.

[20] ../handouts/lambda-coding.pdf
[21] https://en.wikipedia.org/wiki/Church_encoding
[22] https://www.cs.tufts.edu/comp/105/readings/prakash.pdf
[23] https://www.cs.tufts.edu/comp/105/readings/prakash.pdf

Remember these basic terms for encoding 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 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.

We don't know if this one can be done in linear time, but it is sufficient if your function terminates in time quadratic in the size of the Church numeral.

Hint: Think about function `split-list` from the Scheme homework[24], about the implementation of the predecessor function on natural numbers, and about the "window" example from recitation.

3. *Church Numerals—conversion to binary.* Implement the function `binary` from the Impcore homework[25]. 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 list-selection problem (`nth`),
which can benefit from similar skills in recursion, fixed points, and Church numerals. Then come back to this problem.

Your function must terminate in time quadratic 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>;
```

[24] ./scheme.html
[25] ./impcore.html

```
O
-> binary-sym O l <1>;
l
-> binary-sym O l <2>;
l O
-> 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 $E_1 E_2 E_3 \ldots E_n$ in the first section of Rojas's tutorial[26].

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
```

To get full credit for this problem, **you must solve it without recursion.** But 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[27] 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 8 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 first problem also gives you a little more practice in using continuation-passing to code an algebraic data type, which is an essential technique in lambda-land.

[26] https://www.cs.tufts.edu/comp/105/readings/rojas.pdf
[27] https://www.cs.tufts.edu/comp/105/readings/rojas.pdf

For each problem, you will implement types and functions described below. When you are done, the `compile105-lambda` script will link your code with mine to build a complete lambda interpreter. To simplify the configuration, most of the functions and types you must define will be placed in a module called `SealedSolution`, which you will implement in a single file called `solution.sml`. The module must be sealed with this interface:

```
signature SOLUTION = sig

  (************************ BASICS ************************)

  eqtype term
  val lam : string -> term -> term    (* lambda abstraction *)
  val app : term -> term -> term      (* application        *)
  val var : string -> term            (* variable           *)
  val cpsLambda :                     (* observer           *)
    (* forall 'answer . *)
    term ->
    (string -> term -> 'answer) ->
    (term -> term -> 'answer) ->
    (string -> 'answer) ->
    'answer

  (* These functions 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
   *)


  (********************** SUBSTITUTION **********************)

  val freeIn : string -> term -> bool

  val freeVars : term -> string list

  val subst : string * term -> (term -> term)
    (* subst (x, M) returns the capture-avoiding substitution
       of M for x ("x goes to M") *)


  (****************** REDUCTION STRATEGIES ******************)

  val reduceN : term Reduction.reducer  (* normal order *)
  val reduceA : term Reduction.reducer  (* applicative order *)

end
```

You can download a template solution[28].

5. *Evaluation—Coding terms*.

In your file `solution.sml`, create an ML type definition for a type `term`, which should represent a term in the untyped lambda calculus. Using your representation, define functions `lam`, `app`, `var`, and `cpsLambda`.

Compile this file by running `compile105-lambda` (with no arguments), then run any internal unit tests by running `./run-solution-unit-tests`.

My solution is under 15 lines of ML code.

Related reading: The syntax of lambda terms[29] in this homework.

6. *Evaluation—Substitution*. In file `solution.sml`, implement capture-avoiding substitution on your `term` representation. In particular,

- Define function `freeIn` of type `string -> term -> bool`, which tells if a given variable occurs free in a given term. (If you adapt your solution to the pair problem on the ML homework, or my model solution to that problem, acknowledge your sources!)
- Define function `freeVars` of type `term -> string list`, which returns the variables free in a given term. The list must have no duplicates.
- Define function `subst` of type `string * term -> term -> term`. Calling `subst (`x`, `N`) M` returns the term $M[x \mapsto N]$ ("M with x goes to N").

  Function `subst` obeys these algebraic laws,[30] in which x and y stand for variables, and N and M stand for terms:

  (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."

  You 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 this helper function:

- 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.

[28] ./solution.sml

[29] https://www.cs.tufts.edu/comp/105/homework/lambda.html#the-syntax-of-terms

[30] The laws, although notated differently, are identical to the laws given by Prakash Panangaden[31] as Definition 1.3.

To test this problem, you have three possible approaches:

- The next problem demands a minimal set of test cases. You can stick with this set and choose not to worry about further testing.
- You can add `Unit` tests to your `solution.sml` file. You would then compile it by running `compile105-lambda` with no arguments, and run the binary `./run-solution-unit-tests` that results.
- You can also build and run the full interpreter `./linterp`, again by running `compile105-lambda` without arguments. But be warned: 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[32] 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[33] explains the "change of bound variables" that you need to implement if none of the cases for `subst` apply.
- Page 470 of your book defines an ML function `freshName` which is similar to the function `freshVar` that you need to implement. The `freshName` on page 470 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.

  Don't emulate function `freshtyvar` on page 517. 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 renaming is required. In this problem, you create test cases for your `subst` function. They should go into a file `subst-tests.sml`, which should look like this:

```
structure S = SealedSolution
fun toString t = S.cpsLambda t
      (fn name => fn trm => "(lambda (" ^ name ^ ") " ^ toString trm ^ ")")
      (fn t1 => fn t2 => "(" ^ toString t1 ^ " " ^ toString t2 ^ ")")
      (fn name => name)
val N : S.term = S.app (S.app (S.var "fst") (S.var "x")) (S.var "y")
```

[32] https://www.cs.tufts.edu/comp/105/readings/prakash.pdf
[33] https://www.cs.tufts.edu/comp/105/readings/prakash.pdf

```
val checkExpectTerm = Unit.checkExpectWith toString
val () = checkExpectTerm "subst, case (a)" (fn () => S.subst ("x", N) (S.var "x")) N
val () = checkExpectTerm "subst, case (b)" ...
val () = checkExpectTerm "subst, case (c)" ...
val () = checkExpectTerm "subst, case (d)" ...
val () = checkExpectTerm "subst, case (e)" ...
val () = checkExpectTerm "subst, renaming" ...
```

To run these tests, run `compile105-lambda` without arguments, then run the resulting binary `./run-subst-tests`.

8. *Evaluation—Reductions*. In this problem, you use your substitution function to implement two different reduction strategies, called `reduceN` and `reduceA`.

A reduction strategy is a function that takes a term M and produces a one of the following two values:

- `Reduction.DOESN'T_STEP`, if it is not possible to reduce M
- `Reduction.ONE_STEPS_TO` N, if M reduces to N in a single step

The relation "M reduces to N in a single step" is written $M \to N$, and it is explained in the handout on reduction strategies[34] as well as in many other sources on the lambda calculus.

Each function takes a term and tries to perform a **single** reduction step, using any rule that applies: Beta, Eta, Mu, Nu, or Xi. (The rules are shown in the handout on reduction strategies.) Each function is specified as follows:

a. Function `reduceN` implements normal-order reduction: it tries the leftmost, outermost redex first. In other words, it prefers Beta over Nu and Nu over Mu).

b. Function `reduceA` implements applicative-order reduction: it uses the Beta rule only when the argument is normal form. In other words, it prefers Mu over Beta.

Both functions must also implement Eta reduction.

To compile and test this code, run `compile105-lambda` without arguments, then test using `./linterp` (normal-order reduction), `./lamstep` (normal-order reduction, showing each step), and `./linterp-applicative` (applicative-order reduction). You may also wish to consult the hints below.

I've written two solutions to this problem. One solution uses only first-order functions: it implements `reduceN` and `reduceA` directly, by extensive case analysis. My first-order solution is about 30 nonblank lines of ML code. The other solution uses higher-order functions to define `reduceN` and `reduceA`. It implements each *rule* as its own function, then combines them using the `>=>` operator described below. My higher-order solution is about 25 nonblank lines of ML code.

Related reading:

- Start with my guide, "Reduction Strategies for Lambda Calculus[35]."
- For implementation, read the Hints on the implementation of reduction section below.
- Consider consulting Panangaden[36], who 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

[34] https://www.cs.tufts.edu/comp/105/readings/reduction.pdf
[35] ../readings/reduction.pdf
[36] https://www.cs.tufts.edu/comp/105/readings/prakash.pdf

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.

- If you want to know more, or you want a different formulation, go (cautiously) to Wikipedia. Wikipedia describes some individual reduction rules in the Reduction[37] 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[38] section of the lambda-calculus page.

Hints on the implementation of reduction

The return type of `reduceA` and `reduceN` is `term Reduction.result`, where `Reduction.result` is defined by this interface, which also defines some useful helper functions:

```
signature REDUCTION = sig
  datatype 'a result
    = ONE_STEPS_TO of 'a
    | DOESN'T_STEP

  val rmap : ('a -> 'b) -> ('a result -> 'b result)
     (* laws: rmap f (ONE_STEPS_TO e) = ONE_STEPS_TO (f e)
              rmap f DOESN'T_STEP     = DOESN'T_STEP          *)

  type 'a reducer = 'a -> 'a result

  val nostep : 'a reducer

  val >=> : 'a reducer * 'a reducer -> 'a reducer
    (* Sequential composition: try left, then right.
       Associative, with identity nostep *)
end
```

[37] https://en.wikipedia.org/wiki/Lambda_calculus#Reduction
[38] https://en.wikipedia.org/wiki/Lambda_calculus#Reduction_strategies

The helper functions `rmap`, `nostep`, and `>==` are used to implement the second of two possible implementation options:

- The first-order option is simply to take a term, break its representation down by cases, and in each case, define a right-hand side that combines all the rules for that case, including the Eta rule. The advantage of this option is it's concrete, and the programming techniques are ones you've been using all along—break down the data by cases, apply the rules. The disadvantage is that there are a lot of cases, and the logic on the right hand side is complicated. Once you've written the code, it will be hard to understand and hard to debug. Students choosing this option often forget cases or botch cases.
- The higher-order option is to define each *rule* as its own function, then to compose the functions using the `>==` operator in the `Reduction` module.[39] The advantage of this option is that the construction of the reduction strategy makes it crystal clear what is going on and in what order—it becomes very hard to forget or botch a case. This option also makes it easy to implement and test one rule at a time. The disadvantage of this option is that it is abstract, and it is aggressively higher-order: you are using the `>==` arrow to compose simple functions into more complex functions. Understanding the "reducer" abstraction well enough to implement it will take a little time.

Notes on the higher-order option

If you want to try the `REDUCTION` interface and the higher-order option, here are some notes:

- The type called `'a reducer` is really a *partial* reducer: a function of this type implements some sequence of rules. Function `nostep` implements no rules, and the composition arrow `>==` combines two functions to implement a combined sequence of rules. Your implementation task breaks down into two steps: first, define rule functions; second, compose them.
- The `rmap` function is the classic mapping idea (called "homomorphic") which you have already seen in `List.map` and `Option.map`. It is especially useful in conjunction with `curry`, as in

  ```
  Reduction.rmap (curry lam x) (...)
  ```

  You may also find a use for `flip`.
- The composition arrow is mean to be used as an infix operator. In your solution file, copy these definitions:

  ```
  val >==> = Reduction.>==>
  infix 1 >==>
  ```
- The most beautiful code emerges if you define functions `beta`, `eta`, `nu`, `mu`, and `xi`, then compose them:

  ```
  val strategy = xi >==> mu >==> beta >==> ...  (* don't use this order *)
  ```

  But there's a problem here: the `nu`, `mu`, and `xi` rules all need the capability of doing general reduction on a subterm, which means they have to be mutually recursive with the reducer. Mutually recursion can be handled in several ways, but the easiest is to define the individual rule functions *inside* the reducer, in a `let` binding. This easy way does, however, duplicate code. If you want to avoid the duplication, you can do something like this:

[39] This operator is an example of "Kleisli composition," which is an advanced form of function composition.

```
fun xi_maker reducer term = (case term of ...) : term Reduction.result

fun reduceZZZ m =
  let val this = reduceZZZ
      val xi = xi_maker this
      val mu = mu_maker this
      ...
      val reduce = xi >==> mu >==> beta >==> ...    (* don't use this order *)
  in  reduce m
  end
```

Overall, I think the higher-order option is worth the extra effort needed to understand the `reducer` type and its composition: when you split each rule into its own function, it's much, much easier to get them all right. And it's easy to reuse the same functions in multiple reduction strategies.

Debugging support

As shipped, the lambda-calculus interpreter reduces each term repeatedly, until it reaches a normal form. But when you are debugging reduction strategies, you may find it helpful to see the intermediate terms. The `compile105-lambda` script should produce an executable program `./lamstep`, which will show the results of every reduction step. You can compare this interpreter with the reference version, called `lamstep-nr`.

Even more debugging support

If the `./lamstep` interpreter doesn't provide enough information (or provides too much), 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 *)
```

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 `solution.sml` file.

Extra Credit. *Normalization.* Write a higher-order function that takes as argument a reduction 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. (If you choose the higher-order option for implementing reduction strategies, this extra credit is easy. Otherwise, not so much.)

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 problems, including possibly the `binary-sym` extra credit

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

`README`	Collaborators, extra credit, and so on
`solution.sml`	Your module implementing terms, substitution, and reduction
`subst-tests.sml`	Test cases for substitution

As soon as you have the files listed above, and all the code compiles, 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 names and contracts for helper functions.**
- Don't forget a semicolon after each definition.
- Don't forget the question mark in the name of `even?`.
- When using a fixed-point combinator to define a function, don't forget to use `noreduce` in the definition form.
- **Don't use the list representation or primitives from Wikipedia.** We will test your code using the representation and primitives from Coding in Lambda Calculus[40], which you will also find in the file `predefined.lam`[41].
- **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-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-nr
  ```

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 (the Xi rule).
- 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. If M_1 doesn't step, you must try to reduce M_2.
- If you are using the first-order implementation option, **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.

[40] ../handouts/lambda-coding.pdf
[41] https://www.cs.tufts.edu/comp/105/homework/predefined.lam

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[42], 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:

	Exemplary	**Satisfactory**	**Must Improve**
Documentation	• The contract[43] 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[44] that explains what λ-calculus 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[45] omits some parameters. • A function's documentation mentions every parameter, but does not specify a contract[46]. • 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[47] 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.