Midterm Part I: Lambda Calculus

Programming Languages (Fall 2024)

November 3, 2024

This part of the midterm is designed to give you an insight into lambda calculus and convince you it is possible to build any program out of only lambdas. As you work through each step, we will build up our own lambda calculus interpreter. Because of lisp's greedy evaluation and treatment of closures/lambdas we cannot just use the native lisp evaluation engine for lambda calculus (we could in a language like Haskell or Closure). Consider the following lisp program:

```
(setq lexical-binding t)

(setq K
  (lambda (x)
    (lambda (y) x)
  )
)
(setq I
  (lambda (x) x)
)
(setq KI
  (lambda (x)
    (lambda (y) y)
  )
)

(equal KI (funcall K I)) ; Returns nil
```

The reason this last line returns nil is because `KI`'s inner function is still a lambda while `(funcall K I)`'s inner lambda has already been closed (It was evaluated before it was passed in as a parameter to K). This greatly complicates things. As a result we will instead build our own representation for lambda functions within lisp and write our own interpreter for these functions. Hopefully you can see how the foundations we lay could be used to build a lisp interpreter within lambda calculus.

Deliverables: Submit one file "~/CS59/l-calc.lisp" on the pond which contains all the necessary variables and functions:

> **Lisp Functions:** `curry-lambda`, `conv-de-bruijn`, `re-associate`, `b-reduce`, `to-church`, `p-ch`
>
> **λ-Calc Variables:** `l-Not`, `l-Or`, `l-And`, `l-Eq-b`, `l-plus`, `l-mult`, `l-pow`, `l-car`, `l-cdr`, `l-null`, `l-sai`, `l-pred`, `l-zerop`, `l-oddp`, `l-leq`, `l-eq-ch`, `factorial`, `fibonacci`, `l-foldl`, `l-sum-odd`, `l-range`

Important: All code should work within the emacs-lisp environment running on *thepond*. (ssh into *thepond* and run "emacs", open your file and test it!)

Terms and Conditions This midterm is open-notes, open-shell, open-Internet (DO make a note of ALL the resources you used in your submission). You are allowed to discuss tools and techniques with your classmates (again, please make a note of your discussions). Do not disclose actual solutions or their parts.

Step 1: Converting Lisp to a Canonical Syntax

We want to be able to compare things like $\lambda x.\lambda y.x$ with $\lambda y.\lambda xy$ without having to deal with some of the awkward substitutions. In order to do this we will convert all of the lambdas to some canonical syntax. We've already been exposed to one: De Bruijn's notation!

Step 1.1: Currying functions

Lisp allows lambdas to contain multiple parameters. This is will make De Bruijn's notation needlessly complicated, so we want to convert all these multiple argument lambdas into single argument lambdas. Write a function `curry-lambda` which takes one parameter, a lisp lambda expression, and curries it into a new lisp lambda expression, where each lambda takes at most a single variable.

```
(curry-lambda '(lambda () x))          ; Returns (lambda nil x)
(curry-lambda '(lambda (x y) x))       ; Returns (lambda (x) (lambda (y) x))
(curry-lambda '(lambda (x y) (x y x))) ; Returns (lambda (x) (lambda (y) (x y x)))
(curry-lambda '(lambda (x) (x (lambda (u v) u))))
                                  ; Returns (lambda (x) (x (lambda (u) (lambda (v) u))))
```

Step 1.2: De Bruijn Notation

The next step is to replace all the variables with their De Bruijn numbers. Write a function `conv-de-bruijn` which takes in an expression and removes all the variables from the lambdas. Each symbol is then replaced with it's De Bruijn number. This process is tricky and it is strongly encouraged you write helper functions to keep track of your depth or a binding list.

```
(conv-de-bruijn '(lambda (x) (lambda (y) y))) ; Returns (lambda (lambda 1))
(conv-de-bruijn '(lambda (x) (lambda (y) x))) ; Returns (lambda (lambda 2))
(conv-de-bruijn '(lambda (x) (lambda () (x (lambda (x) x)))))
                                          ; Returns (lambda (lambda (2 (lambda 1))))
```

Step 1.3: Re-associate and Re-label

We still aren't completely done. We currently have a bunch of lists which are bulky and in-elegant. They are also naturally associate the wrong way: the list `(1 2 3)` is currently represented in lisp as a cons block containing 1 with a cdr of the cons block containing 2, with a cdr of a cons block containing 3, with a cdr of nil. In other words this list is really `[1 . [2 . [3 . nil]]]`. But with lambda calculus we want the association to go the other direction, because this *should* represent 1 applied to 2, and the resulting function then applied to 3. Also, everything is a function or an application of a function. So we can represent a function with the cons block `[L . <body>]` and a function application with the cons block `[<function> . <application>]`. Write a functions `re-associate` which does this transformation. This means:

```
(re-associate '(lambda (lambda 1)))                    ; Returns (L L . 1)
(re-associate '(lambda (1 1)))                         ; Returns (L 1 . 1)
(re-associate '(lambda (lambda (lambda (3 1 (2 1)))))) ; Returns (L L L (3 . 1) 2 . 1)
```

Now we can write a function which converts a lisp lambda expression into our canonical form.

```
(defun to-l-exp (exp)
  (re-associate (conv-de-bruijn (curry-lambda exp)))
)
(to-l-exp '(lambda (x y) x))                ; (L L . 2)
(to-l-exp '(lambda (x y) y))                ; (L L . 1)
(to-l-exp '(lambda (a b c d) (a c (b c d)))) ; (L L L L (4 . 2) (3 . 2) . 1)
(to-l-exp '(lambda (x y z) (x z (y z))))    ; (L L L (3 . 1) 2 . 1)
(to-l-exp '(lambda (z) ((lambda (y) (y (lambda (x) x))) (lambda (x) (z x)))))
                                            ; (L (L 1 L . 1) L 2 . 1)
```

Step 2: Implementing Function Application

Now that we can convert a lisp lambda into a canonical form, we need to be able to evaluate these expressions. However evaluation is simply just repeated β-reductions, so this is our first task. Write a function `b-reduce` which takes two parameters, both of which are lambda expressions in our canonical form, and then apply the second argument to the first argument. You may assume the first argument is a lambda expression. This is to say `(b-reduce [L . <body>] <arg>)` should return `<body>` with every instance of a variable associated by the lambda replaced by `<arg>`. See the following examples to understand how the β-reduction works:

```
(b-reduce (cons 'L (cons 'L 1)) (cons 'L 1))
    ; (L L . 1) applied to (L . 1) returns:       (L . 1)

(b-reduce (cons 'L (cons 'L 2)) (cons 'L 1))
    ; (L L . 2) applied to (L . 1) returns:       (L L . 1)

(b-reduce (cons 'L (cons 'L 4)) (cons 'L 1))
    ; (L L . 4) applied to (L . 1) returns:       (L . 3)

(b-reduce (cons 'L (cons 'L 2)) (cons 'L 4))
    ; (L L . 2) applied to (L . 4) returns:       (L L . 5)

(b-reduce (cons 'L (cons 4 1)) (cons 1 2))
    ; (L 4 . 1) applied to (2 . 4) returns:       (3 1 . 2)

(b-reduce (cons 'L (cons 1 2)) (cons 'L 4))
    ; (L 1 . 2) applied to (L . 4) returns:       ((L . 4) . 1)

(b-reduce (cons 'L (cons 1 (cons 'L (cons 7 2)))) (cons 'L (cons 1 3)))
    ; (L 1 L 7 . 2) applied to (L 1 . 3) returns: ((L 1 . 3) L 6 L 1 . 4)

(b-reduce (cons 'L (cons 'L (cons 'L (cons 'L (cons 4 5))))) (cons 'L 2))
    ; (L L L L 4 . 5) applied to (L . 2) returns: (L L L (L . 5) . 4)
```

Writing helper functions will be very helpful for this. You will notice this is not as simple as implementing a typical `apply` function, because it is not enough to just replace all instances of a variable with the argument. Because we are representing everything in De Bruijn notation, we have to figure out which variables are "free" (i.e. not bound by lambdas within the expression) and figure out how much they much be incremented or decremented by.

Step 3: The Evaluation Engine

We can now write an evaluation engine. Due to the nature of recursive structures in lambda calculus, we require lazy evaluation so that the Y-combinator doesn't cause infinite recursion, but to check equality we also require full simplification. As a result we implement two different types of evaluation: `eval-L` provides a lazy evaluation of lambda expressions while `simplify` will greedily evaluate everything until there is no more β-reductions to apply. Note that this solution is not optimized at all (but it will always terminate provided you do not pass in a bottom). If you want you can optimize it further by checking for certain structures and not expanding them if necessary. One method which will work to save time but will eat up a ton of stack space is to pass in a list of previously encountered expressions to `simplify-step` and never try to reduce them unless necessary. (I tried this first and it allows you to do things a lot quicker (5! takes less than a second) but will prevent you from doing basically anything with two Y-combinators because you overflow the stack.

```
(defun lambdap (l-exp)      ; checks if the l-exp is a lambda
  (and
    (consp l-exp)           ; the l-exp must be a cons cell
    (eq 'L (car l-exp))     ; and its car must be 'L'
  )
)

(defun simplify-step (l-exp full-eval)
  (if (consp l-exp)         ; check if the l-exp is function / application
    (let (                  ; simplify the function
        (car-exp (simplify-step (car l-exp) nil))
        (cdr-exp (cdr l-exp))
      )
      (if (lambdap car-exp) ; check if the function being applied is a lambda expression
        (simplify-step      ; if so b-reduce and then simplify again
          (b-reduce car-exp cdr-exp)
          full-eval
        )
        (cons               ; otherwise
          (simplify-step car-exp full-eval)
          (if full-eval     ; if we are full-simplifying
            (simplify-step  ; lazily evaluate the cdr, and then simplify it
              (simplify-step cdr-exp nil)
              t
            )
            cdr-exp         ; otherwise no need to worry about it
          )
        )
      )
    )
    l-exp                   ; if it is not a function / application no need to do anything
  )
)

defun eval-L (l-exp)        ; lazy evaluator
  (simplify-step l-exp nil)
)

(defun simplify (l-exp)     ; fully beta-reduce everything
  (simplify-step l-exp t)
)

(defun s-eq (l-exp1 l-exp2) ; check structural equality after simplifying
  (equal
    (simplify l-exp1)
    (simplify l-exp2)
  )
)
```

We can then create a few helper functions just to make directly creating De Bruijn easier. (You don't have to use these of course, you can always just use `to-l-exp` to convert lisp lambdas to this form if you want!)

```
(defun Lx (f)           ; Helper function to make a lambda expression
  (cons 'L f)
)
(defun app (&rest all) ; Helper function to apply one l-exp to one or more other l-exps
  (cl-reduce #'cons all)
)
```

And now we can verify the engine works:

```
(defvar I
  (Lx 1)
)
(defvar K
  (Lx (Lx 2))
)
(defvar KI
  (Lx (Lx 1))
)
(s-eq (app K I) KI) ; Returns true!
```

Step 4: Basic IO - Booleans, Church Numerals, and Lists

To make IO a bit easier for our self, we should make expression which make it easier to encode certain concepts into our lambda expression notation.

```
(defvar l-True K)
(defvar l-False KI)

(defun p-bool (l-exp)
  (let ((e-exp (simplify l-exp)))
  (if (s-eq e-exp l-True)
    'True
    (if (s-eq e-exp l-False)
      'False
         'Not-a-Bool
       )
  ))
)
```

Step 4.1: Boolean functions

Write variables `l-Not`, `l-And`, `l-Or`, `l-Eq-b` which express the corresponding boolean lambda functions.

```
(p-bool (app l-Not l-True))          ; False
(p-bool (app l-Not l-False))         ; True

(p-bool (app l-Or l-False l-False)) ; False
(p-bool (app l-Or l-False l-True))  ; True
(p-bool (app l-Or l-True l-False))  ; True
(p-bool (app l-Or l-True l-True))   ; True
```

```
(p-bool (app l-And l-False l-False))  ; False
(p-bool (app l-And l-False l-True))   ; False
(p-bool (app l-And l-True l-False))   ; False
(p-bool (app l-And l-True l-True))    ; True

(p-bool (app l-Eq-b l-False l-False)) ; True
(p-bool (app l-Eq-b l-False l-True))  ; False
(p-bool (app l-Eq-b l-True l-False))  ; False
(p-bool (app l-Eq-b l-True l-True))   ; True
```

Step 4.2: Church Numerals IO

Recall in class, we were able to encode numbers in lambda expression as two parameter functions, which apply the first argument to the second argument that number of times. These are called "Church Numerals" and provides one common formalism for the natural numbers within lambda calculus. Write a lisp function, `to-church` which takes a non-negative integer and constructs the lambda expression for the corresponding Church numeral.

```
(defvar Ch-0 (to-church 0))
(defvar Ch-1 (to-church 1))
(defvar Ch-2 (to-church 2))
(defvar Ch-3 (to-church 3))
(defvar Ch-4 (to-church 4))
(defvar Ch-5 (to-church 5))
(defvar Ch-6 (to-church 6))
(defvar Ch-7 (to-church 7))
(defvar Ch-8 (to-church 8))
(defvar Ch-9 (to-church 9))

Ch-0 ; returns (L L . 1)
Ch-1 ; returns (L L 2 . 1)
Ch-4 ; returns (L L 2 2 2 2 . 1)
```

Now write a lisp function `p-ch` which takes a lambda expression and if that expression is a church numeral, returns the integer it represents. Otherwise it returns `Not-a-Church-Num` (similarly to how `p-bool` could return `Not-a-Bool`)

```
(p-ch Ch-0)             ; returns 0
(p-ch Ch-3)             ; returns 3
(p-ch (to-church 194)) ; returns 194
(p-ch K)                ; returns Not-a-Church-Num;
```

Step 4.3: Basic Church Arithmetic

Write variables `l-plus`, `l-mult`, `l-exp` for the lambda expression for plus, multiplications, and exponentiation on the church numerals. (You can return any value for 0^0, though if you want extra points return something that will print `Not-a-Church-Num`)

```
(p-ch (app l-plus Ch-0 Ch-0)) ; returns 0
(p-ch (app l-plus Ch-3 Ch-4)) ; returns 7
(p-ch (app l-plus Ch-7 Ch-8)) ; returns 15

(p-ch (app l-mult Ch-1 Ch-3)) ; returns 3
(p-ch (app l-mult Ch-8 Ch-0)) ; returns 0
(p-ch (app l-mult Ch-5 Ch-9)) ; returns 45

(p-ch (app l-pow Ch-0 Ch-1)) ; returns 0
(p-ch (app l-pow Ch-7 Ch-0)) ; returns 1
(p-ch (app l-pow Ch-4 Ch-1)) ; returns 4
(p-ch (app l-pow Ch-1 Ch-8)) ; returns 1
(p-ch (app l-pow Ch-3 Ch-5)) ; returns 243
```

Step 4.4: Lists

We are getting closer to creating all of lisp within lambda calculus. The next major hurdle for us is lists. To do this, we need to make a cons cell. The are many possible options for how to do this, but to keep consistency between everyone, let's choose the following definition:

```
(defvar l-cons
  (Lx (Lx (Lx (app 1 3 2))))
)
```

Write two lambda expressions `l-car` and `l-cdr` which extract the first and second elements from a cons block constructed using the above definition for cons.

```
(p-ch (app l-car (app l-cons Ch-5 Ch-8))) ; returns 5
(p-ch (app l-cdr (app l-cons Ch-5 Ch-8))) ; returns 8
```

Now we need something to represent the end of the list. We could take inspiration from lisp and define `l-False` to be nil, however this ends up being somewhat inconvenient, so let's use the following definition instead:

```
(defvar l-nil
  (Lx l-True)
)
```

Write a variable `l-null` which represents a lambda expression which will return false when given a cons block constructed with `l-cons` and return true if given `l-nil` (you can leave it's behavior undefined if it is given any other expression).

```
(defun make-l (&rest l-exps)
  (cl-reduce
    (lambda (l-exp l-list)
      (app l-cons l-exp l-list)
    )
    l-exps
    :initial-value l-nil
    :from-end t
  )
)

(defvar test-list (make-l Ch-0 Ch-1 Ch-5))

(p-ch (app l-car test-list))                                        ; returns 0
(p-ch (app l-car (app l-cdr test-list)))                            ; returns 1
(p-ch (app l-car (app l-cdr (app l-cdr test-list))))                ; returns 5
(p-bool (app l-null test-list))                                     ; returns False
(p-bool (app l-null (app l-cdr (app l-cdr test-list))))             ; returns False
(p-bool (app l-null (app l-cdr (app l-cdr (app l-cdr test-list))))) ; returns True
```

Step 4.5: Predecessor Function

While the church numerals may seem very convenient for things such as adding and multiplying, subtraction seems like an initial issue. However this becomes easier now that we are able to make lists (or, more specifically, make a pair of items). Define a lambda expression, `l-sai`, which is the "shift and increment function" : it takes one argument, a pair (a, b) and returns a new pair $(b, b+1)$.

```
(p-ch (app l-car (app l-sai (app l-cons Ch-4 Ch-5)))) ; returns 5
(p-ch (app l-cdr (app l-sai (app l-cons Ch-4 Ch-5)))) ; returns 6
(p-ch (app l-car (app l-sai (app l-cons Ch-0 Ch-0)))) ; returns 0
(p-ch (app l-cdr (app l-sai (app l-cons Ch-3 Ch-9)))) ; returns 10
```

We can use this operator to create a natural definition for the predecessor function. Create a lambda expression, `l-pred`, which takes a single natural number n and returns $n-1$ if n is greater than 0 and returns 0 if $n = 0$.

```
(p-ch (app l-pred Ch-0)) ; returns 0
(p-ch (app l-pred Ch-1)) ; returns 0
(p-ch (app l-pred Ch-9)) ; returns 8
```

Step 4.6: Other Basic Operations

We're getting close to being able to write some interesting functions using these building blocks, but we still have a few more building blocks left to build. First and foremost is an "if" statement. Definite a lambda expression `l-if` which takes a boolean argument and then two other untyped arguments. The expression returns the first untyped argument if the boolean argument is true, and the other untyped argument otherwise.

```
(p-ch (app l-if l-True Ch-3 Ch-8))  ; returns 3
(p-ch (app l-if l-False Ch-3 Ch-8)) ; returns 8
```

Now we would like to be able to compare church numerals with each other. Write lambda expressions `l-zero`, `l-oddp`, `l-leq`, `l-eq-ch` to test if a num is zero, odd, less than or equal to another number, and equal to another, respectively.

```
(p-bool (app l-zerop Ch-0))      ; returns true
(p-bool (app l-zerop Ch-1))      ; returns false
(p-bool (app l-zerop Ch-7))      ; returns false

(p-bool (app l-oddp Ch-0))       ; returns false
(p-bool (app l-oddp Ch-3))       ; returns true
(p-bool (app l-oddp Ch-8))       ; returns false

(p-bool (app l-leq Ch-4 Ch-4))   ; returns true
(p-bool (app l-leq Ch-0 Ch-6))   ; returns true
(p-bool (app l-leq Ch-8 Ch-7))   ; returns false

(p-bool (app l-eq-ch Ch-6 Ch-6)) ; returns true
(p-bool (app l-eq-ch Ch-1 Ch-0)) ; returns false
(p-bool (app l-eq-ch Ch-3 Ch-9)) ; returns false
```

Step 4.7 Integers (Extra Credit)

As you can see, we can get really far just using the church numerals, but sometimes we want integers. There are two common approaches both of which represent the integer as a cons cell. The most programmatic would be to have one element of the cons cell be a bool representing the sign while the other element is the church numeral for the absolute value. The most mathematically elegant approach is to have the both elements of the cons block be a church numeral and the value of (a, b) is $a - b$.

Step 5: Recursion

We saw in class how fixed-point operators can be used to create recursion without the need for naming a function. For systems (like ours) which use lazy evaluation, the `Y`-combinator tends to be the simplest (In systems with greedy evaluation, you can still use the same tricks but the `Z`-combinator is generally needed; which is the same as wrapping the inside of `Y` with an extra lambda). Both of these functions have the nice property that `Y g = g (Y g)` (which, incidentally, has the same property as `fix` but doesn't require name spaces to work). Recall:

```
(defvar Y
  (Lx
    (app
      (Lx (app 2 (app 1 1)))
      (Lx (app 2 (app 1 1)))
    )
  )
)
```

Use the `Y` combinator two make two functions `factorial` and `fibonacci`, which compute the factorial and Fibonacci function of a number, respectively.

```
(p-ch (app factorial Ch-0)) ; returns 1
(p-ch (app factorial Ch-2)) ; retirns 2
(p-ch (app factorial Ch-5)) ; returns 120 (after a long pause)

(p-ch (app fibonacci Ch-0)) ; returns 0
(p-ch (app fibonacci Ch-2)) ; returns 1
(p-ch (app fibonacci Ch-5)) ; returns 5
(p-ch (app fibonacci Ch-9)) ; returns 34 (after a long pause)
```

Write a lambda expression `l-foldl` which performs a left-fold down a list made using `l-cons`. The first parameter is the function being folded, the second the initial value, and the third the list.

```
(p-ch (app l-foldl l-plus Ch-0 (make-l)))                ; returns 0
(p-ch (app l-foldl l-plus Ch-0 (make-l Ch-3 Ch-2 Ch-9))) ; returns 14
(p-ch (app l-foldl l-mult Ch-1 (make-l Ch-6 Ch-8 Ch-5))) ; returns 240
(p-ch (app l-foldl l-pow Ch-2 (make-l Ch-7)))            ; returns 128
```

Write a lambda expression `sum-odd` which adds all the odd numbers in a list (just like your homework 2!)

```
(p-ch (app l-sum-odd (make-l)))                          ; returns 0
(p-ch (app l-sum-odd (make-l Ch-2)))                     ; returns 0
(p-ch (app l-sum-odd (make-l Ch-3 Ch-9)))                ; returns 12
(p-ch (app l-sum-odd (make-l Ch-1 Ch-5 Ch-8 Ch-2 Ch-3))) ; returns 9
```

Write a lambda expression `l-range` which takes two arguments and generates a list contain the numbers in between the two (including both, and if the second argument is less than the first then the list is empty)

```
(p-ch (app l-car (app l-range Ch-3 Ch-8)))               ; returns 3
(p-ch (app l-sum-odd (app l-range Ch-1 Ch-9)))           ; returns 25 (after a pause)
(p-bool (app l-null (app l-range Ch-4 Ch-3)))            ; returns True
(p-bool (app l-null (app l-range Ch-7 Ch-7)))            ; returns False
(p-bool (app l-null (app l-cdr (app l-range Ch-2 Ch-2)))) ; returns True
```

Note that our evaluation engine is naturally lazy, so you can do all the same fun Haskell tricks for making infinite lists. You can use the Y combinator to make up for the lack of name spaces. For extra credit make infinite lists for the naturals, square numbers, Fibonacci sequence and other things you can think of!