Pure Lambda Calculus in Scheme

A set of Racket macros to transform syntax into pure lambda calculus before evaluation

Eric Breyer

March 2022

Contents

0 Purpose

This document is meant as a companion to the Racket file containing the comprehensive list of macro definitions for this "meta-language." It is not comprehensive, but details some of the background more complicated parts of the project. Please feel free to download the file and play around with the definitions and pure expansion of various programs.

1 The Premise

In learning about lambda calculus and playing around with Scheme/Racket, I started to notice the parallels between the Scheme programming language and lambda calculus. Scheme at the base level with concepts of conditionals, cons lists, recursion, and let expressions can all be built with 3 fundamental concepts:

1. Variables: x
2. Lambda expressions: **lambda** (args) (body)
3. Function application: (func args)

This means we should be able to write Turing-complete Scheme code using only the keyword **lambda**, arbitrary variable names, and a few parenthesis.

It would not be feasible to write all these lambdas from scratch, so we will utilize Scheme's macro feature to change more readable code into pure lambdas before evaluation. The only evaluated computations will only be done with the pure lambda calculus terms and operations.

2 Tools

Lambda Calculus The whole point of this project is to use lambda calculus to make functioning programs. I have already written a sort of write up as I learned about lambda calculus. It details many concepts such as Booleans, numerals, arithmetic, control flow, lists, and recursion. I will be using it as a guide for the lambda calculus parts of this project.

Racket Although the original idea was and is to emulate Scheme with pure lambda calculus using Scheme macros, I will be using the Racket programming language for this project. Racket began as a superset of "base" Scheme, but as features were added it became more accurate to call it a different language. Racket is not Scheme, but draws heavily from it and still acts very much like a superset of it, except for a few edge cases.

Macros From the Racket docs: "A pattern-based macro replaces any code that matches a pattern to an expansion that uses parts of the original syntax that match parts of the pattern." We can define a new syntax rule, and the compiler will replace any code that matches our syntax with the new syntax. This means we can use macros to replace "simple" functions like **add** with their pure lambda calculus representations before runtime.

DrRacket The DrRacket IDE will also prove to be a very helpful tool, specifically its macro stepper. The macro stepper allows us to actually see what the compiler is replacing syntax with. This is a powerful debugging tool but also lets us see the pure lambda calculus that the macros will reduce the code to.

3 "Purifying" the Let Syntax

The Goal Create a macro to convert the more readable Scheme syntax for let expressions: `(let [(a b) (c d)] **body**)` into the pure lambda calculus form:

```
((lambda (a) ((lambda (c) **body**) d)) b)
```

The let macro We will first define a macro to match the let syntax and transform it into a (multi-argument) lambda

```
(define-syntax let
  (syntax-rules ()
    [(_ ((a b) ...) body ...) ((lambda (a ...) body ...) b ...)]))
```

This macro matches the typical let syntax of pairwise definitions and a body. It then transforms it into a lambda expression where the arguments are the variable names, the body is the let body, and the variable values are applied to it.

```
(let [(a 2)
      (b 4)]
  (+ a b))
```

expands to

```
((lambda (a b) (+ a b)) 2 4)
```

using this new syntax rule.

Currying This expanded definition does only use lambdas for the let, but it uses multi-argument functions, which are invalid in lambda calculus. We can define a curry macro to take a multi argument function/application and curry it into a sequence of single-argument functions.

```
(define-syntax curry
  (syntax-rules ()
    [(_ ((lambda (a) body ...) c)) ((lambda (a) body ...)c)]
    [(_ ((lambda (a b ...) body ...) c d ...))
       ((lambda (a) (curry ((lambda (b ...) body ...) d ...))) c)]))
```

Snippet 1: The curry macro

This macro uses a recursive-like definition to reduce a function with an arbitrary amount of elements into single-argument lambda expressions. Line 3 is like the recursive base case: if the outermost lambda already has only one argument, we are done and don't change the syntax. Lines 4 and 5 are the recursive case. They match a multi-argument lambda, then take the first argument and pull it out into a single argument lambda, also pulling out the corresponding application argument, and it will then curry the rest.

As an example:

```
(curry ((lambda (a b) (+ a b)) 2 4))
```

will match the first case and reduce to:

```
(lambda (a) (curry ((lambda (b) (+ a b)) 4))) 2)
```

which will match the second case and reduce to:

```
((lambda (a) ((lambda (b) (+ a b)) 4)) 2)
```

The "pure" let macro Per this definition of curry, all we need to do to output single-argument lambdas with let is curry the original converted syntax. Thus the final let macro is

```
1 (define-syntax let
2   (syntax-rules ()
3     [(_ ((a b) ...) body ...) (curry ((lambda (a ...) body ...) b ...))]))
```

Snippet 2: The let macro

Syntax expansion using the let macro

1. `(let [(a 2) (b 4)] (+ a b))`
2. `(curry ((lambda (a b) (+ a b)) 2 4))`
3. `(lambda (a) (curry ((lambda (b) (+ a b)) 4))) 2)`
4. `((lambda (a) ((lambda (b) (+ a b)) 4)) 2)`

4 Numbers to Church Numerals

The let macros were a good introduction to syntax rules and recursive macros, and much of the learnings and patterns can be reused in further definitions. Converting numbers (1, 2, 3) to church numerals is the key to making this project work.

First try We will use the idea of a recursive macro to add function applications one by one

```
1 (define-syntax toChurch
2   (syntax-rules ()
3     [(_ num) (toChurch num (lambda (f) (lambda (x) x)))]
4     [(_ 0 (_ (f) (_ (x) body))) (lambda (f) (lambda (x) body))]
5     [(_ num (_ (f) (_ (x) body)))
6        (toChurch (- num 1) (lambda (f) (lambda (x) (f body))))]))
```

This macro is fairly complicated so lets break it down.

Line 3: "Initial" case This line takes the "initial" syntax form and transforms it into the "recursive" form. The recursive form of the macro takes a church numeral and the number of applications left to add to that numeral. This line 3 kicks of this recursion by calling the macro with the original number to convert and a zero in lambda calculus.
The syntax `(toChurch 3)` will match the case in line 3 and be transformed into `(toChurch 3 (lambda (f) (lambda (x) x)))`

Line 4: Base case If there are no applications left to add to the church numeral, simply drop the recursive macro and reduce to the body.
The syntax `(toChurch 0 (lambda (f) (lambda (x) (f (f (f x))))))` will match the case in line 4 and be transformed into `(lambda (f) (lambda (x) (f (f (f x)))))`

Lines 5-6: Recursive case The recursive case adds one more function application around the body, and applies the toChurch macro again with the counter reduced by one.
The syntax `(toChurch 2 (lambda (f) (lambda (x) (f x))))` will match the case in line 5 and **should be** transformed into `(toChurch 1 (lambda (f) (lambda (x) (f (f x)))))`.

However, syntax-rules works only in syntax, so this syntax actually transforms into `(toChurch (- 2 1) (lambda (f) (lambda (x) (f (f x)))))`, the (- 2 1) does not evaluate inside this syntax transformation. This new syntax does not match any case of the toChurch macro, and thus breaks the recursion.

Second try We have to use some tricks to move expressions from data (so we can perform arithmetic) to syntax (so we can do the transformation). The correct macro is

```
1 (define-syntax (toChurch stx)
2   (syntax-case stx ()
3     [(_ num) #'(toChurch num (lambda (f) (lambda (x) x)))]
4     [(_ 0 (_ (f) (_ (x) body))) #'(lambda (f) (lambda (x) body))]
5     [(_ num (_ (f) (_ (x) body)))
6      (with-syntax ([newnum (datum->syntax #'num
7                                           (- (syntax->datum #'num) 1))])
8       #'(toChurch newnum (lambda (f) (lambda (x) (f body))))])))
```

Snippet 3: The toChurch macro

Lines 6 and 7 define a new piece of syntax, newnum, and assign is a manipulated value of num, accomplished by taking num in and out of being a syntax object. We can then use this newnum in the new syntax.

Syntax expansion using the toChurch macro

1. `(toChurch 3)` matching first case
2. `(toChurch 3 (lambda (f) (lambda (x) x)))` matching third case
3. `(toChurch 2 (lambda (f) (lambda (x) (f x))))` matching third case
4. `(toChurch 1 (lambda (f) (lambda (x) (f (f x)))))` matching third case
5. `(toChurch 0 (lambda (f) (lambda (x) (f (f (f x))))))` matching first case
6. `(lambda (f) (lambda (x) (f (f (f x)))))`

Smaller syntax (toChurch _) is slightly verbose, since we will likely be using numbers a lot, and we will need to convert them to lambda form every time. I will define a short alias for the toChurch macro: $.

```
1 (define-syntax $
2   (syntax-rules ()
3     [($ arg) (toChurch arg)]))
```

Snippet 4: The $ macro

Syntax expansion using the $ macro

1. `($ 7)`
2. `(toNum 7)`
3. `(lambda (f) (lambda (x) (f (f (f (f (f (f (f x))))))))))`

5 Simple Macros

The syntax rules for functions such as SUCC, AND, and PAIR are fairly simple. We simply need to swap out the "simple" syntax for the pure lambda definitions and apply the given arguments. These macros do need a few extra cases depending on how many arguments are passed in. This is to allow for partial application and passing functions around. I have defined a few examples bellow, the rest can be found in the full list of macros in the Racket file at my GitHub.

```
(define-syntax (SUCC stx)
  (syntax-case stx ()
    [(_ arg) #'((lambda (n) (lambda (f) (lambda (x) (f ((n f) x))))) arg)]
    [SUCC #'(lambda (n) (lambda (f) (lambda (x) (f ((n f) x)))))]))

(define-syntax (AND stx)
  (syntax-case stx ()
    [(_ arg1) #'((lambda (p) (lambda (q) ((p q) p)))arg1)]
    [(_ arg1 arg2) #'(((lambda (p) (lambda (q) ((p q) p)))arg1)arg2)]
    [AND #'(lambda (p) (lambda (q) ((p q) p)))]))

(define-syntax (PAIR stx)
  (syntax-case stx ()
    [(_ arg1) #'((lambda (x) (lambda (y) (lambda (f) ((f x) y))))arg1)]
    [(_ arg1 arg2)
       #'(((lambda (x) (lambda (y) (lambda (f) ((f x) y))))arg1)arg2)]
    [(_ arg1 arg2 func)
      #'((((lambda (x) (lambda (y) (lambda (f) ((f x) y))))arg1)arg2)func)]
    [PAIR #'(lambda (x) (lambda (y) (lambda (f) ((f x) y))))]))
```

Snippet 5: Simple macro examples

6 The List Macro

We need a better way to define lists than long sequences of cons. With this macro, a list can be defined as (LIST (listElement1 listElement2 listElement3 ...)) and be converted to a lambda calculus cons list. This macro works quite similarly to toNum.

```
1 (define-syntax LIST
2   (syntax-rules ()
3     [(_ (arg ... last)) (LIST (arg ...) (PAIR last NIL))]
4     [(_ () pairs) pairs]
5     [(_ (arg ... last) pairs) (LIST (arg ...) ((PAIR last) pairs))]))
```

Snippet 6: The LIST macro

Line 3: "Initial" case The syntax `(LIST (($ 3) ($ 2) ($ 1)))` will match the case in line 3 and be transformed into `(LIST (($ 3) ($ 2)) (PAIR ($ 1) NIL))`

Line 4: Base case If there are no arguments left to add to the church list, simply drop the recursive macro and reduce to the body.
The syntax `(LIST () (PAIR ($ 3) (PAIR ($ 2) (PAIR ($ 1) NIL))))` will match the case in line 4 and be transformed into `(PAIR ($ 3) (PAIR ($ 2) (PAIR ($ 1) NIL)))`

Lines 5-6: Recursive case The recursive case adds one more function application around the body, and applies the toChurch macro again with the counter reduced by one.
The syntax `(LIST (($ 3) ($ 2)) ((PAIR ($ 1) NIL))` will match the case in line 5 and be transformed into The syntax `(LIST (($ 3)) (PAIR ($ 2) (PAIR ($ 1) NIL)))`

Syntax expansion using the LIST macro

1. `(LIST (($ 3) ($ 2) ($ 1)))`
2. `(LIST (($ 3) ($ 2)) ((PAIR ($ 1) NIL))`
3. `(LIST (($ 3)) (PAIR ($ 2) (PAIR ($ 1) NIL)))`
4. `(LIST () (PAIR ($ 3) (PAIR ($ 2) (PAIR ($ 1) NIL))))`
5. `(PAIR ($ 3) (PAIR ($ 2) (PAIR ($ 1) NIL)))`

7 Recursion

To make a recursive function we need to use the Y combinator. I am using my "lazy" version of the Y combinator here.

```
(define-syntax M
  (syntax-rules ()
    ((_ func) ((lambda (f) (f f)) func))))

(define-syntax Y
  (syntax-rules ()
    [(_ func) ((lambda (f) (M (lambda (x) (f (lambda () (x x))))))) func)]))
```

Snippet 7: The Y combinator

8 Examples

These macros will transform the nicer syntax into pure lambda calculus syntax before evaluation. Racket will then evaluate and do calculations using the lambda form. Here are some examples of transforming from regular to lambda syntax. I will be using Rackets macro stepper for this.

8.1 Simple add

```
(let [(THREE ($ 3))
      (TEN ($ 10))]
  (ADD THREE TEN))
```

Is parsed into the code bellow by the macro definitions.

```
((lambda (THREE)
  ((lambda (TEN)
    (((lambda (m) (lambda (n)
      ((m (lambda (n) (lambda (f) (lambda (x) (f ((n f) x)))))) n)))
      THREE) TEN))
    (lambda (f) (lambda (x) (f (f (f (f (f (f (f (f (f (f x))))))))))))
    (lambda (f) (lambda (x) (f (f (f x)))))))
```

- Lines 1, 2, 6, 7 are the let expression
- Lines 3, 4 are the ADD expression
- Line 4 contains the expansion of SUCC in ADD
- Line 5 is the application to ADD

8.2 List of Fibonacci Numbers

The extra lambdas in the definitions are so only one side of the conditional is evaluated.

```
(let [
  (fib
   (Y
    (lambda (f)
     (lambda (x)
      ((((ZERO? x) (lambda () ($ 0)))
        (((ZERO? (PRED x)) (lambda () ($ 1)))
          (lambda () ((ADD ((f) (PRED x))) ((f) (PRED (PRED x))))))))))))
  (fibList
   (Y
    (lambda (f)
     (lambda (x)
      ((((ZERO? x) (lambda () ((PAIR (fib ($ 0))) NIL)))
       (lambda () ((PAIR (fib x)) ((f) (PRED x))))))))))

      ]
  (fibList ($ 16)))
```

Reduces into the following using the macro stepper. (The :n after some lambdas and variables is just the macro stepper differentiating between seperate terms with the same name)

The output of this program is `'(987 610 377 233 144 89 55 34 21 13 8 5 3 2 1 1 0)`

Again, all computation was done using pure lambda calculus syntax.

```
((lambda (fib)
  ((lambda (fibList) (fibList (lambda (f) (lambda (x) (f (f (f (f (f (f (f (f (f
      (f (f (f (f (f x))))))))))))))))))
   ((lambda:22 (f:22) ((lambda:23 (f:23) (f:23 f:23)) (lambda:22 (x:22) (f:22 (
       lambda:22 () (x:22 x:22))))))
    (lambda (f)
     (lambda (x)
      (((((lambda:24 (n:24)
       ((n:24 (lambda:24 (x:24) (lambda:25 (a:25) (lambda:25 (b:25) b:25)))) (
           lambda:26 (a:26) (lambda:26 (b:26) a:26))))
         x)
        (lambda ()
         (((lambda:27 (x:27) (lambda:27 (y:27) (lambda:27 (f:27) ((f:27 x:27) y
             :27))))
          (fib (lambda:28 (f:29) (lambda:28 (x:29) x:29))))
         (lambda:30 (x:30) (lambda:31 (a:31) (lambda:31 (b:31) a:31))))))
       (lambda ()
        (((lambda:32 (x:32) (lambda:32 (y:32) (lambda:32 (f:32) ((f:32 x:32) y
            :32)))) (fib x))
         ((f)
          ((lambda:33 (n:33)
             ((lambda:34 (p:34) (p:34 (lambda:35 (a:35) (lambda:35 (b:35) a:35))
                 ))
```

```
                ((n:33
                  (lambda:36 (p:36)
                    (((lambda:37 (x:37) (lambda:37 (y:37) (lambda:37 (f:37) ((f:37
                        x:37) y:37))))
                      ((lambda:38 (p:38) (p:38 (lambda:39 (a:39) (lambda:39 (b:39)
                          b:39)))) p:36))
                     ((lambda:40 (n:40) (lambda:40 (f:40) (lambda:40 (x:40) (f:40
                         ((n:40 f:40) x:40)))))
                      ((lambda:41 (p:41) (p:41 (lambda:42 (a:42) (lambda:42 (b:42)
                          b:42)))) p:36)))))
                 (((lambda:43 (x:43) (lambda:43 (y:43) (lambda:43 (f:43) ((f:43 x
                     :43) y:43))))
                  (lambda:44 (f:45) (lambda:44 (x:45) x:45)))
                  (lambda:46 (f:47) (lambda:46 (x:47) x:47))))))
               x))))))))))
((lambda:48 (f:48) ((lambda:49 (f:49) (f:49 f:49)) (lambda:48 (x:48) (f:48 (lambda
    :48 () (x:48 x:48))))))
 (lambda (f)
   (lambda (x)
     (((((lambda:50 (n:50)
          ((n:50 (lambda:50 (x:50) (lambda:51 (a:51) (lambda:51 (b:51) b:51)))) (
              lambda:52 (a:52) (lambda:52 (b:52) a:52))))
        x)
       (lambda () (lambda:53 (f:54) (lambda:53 (x:54) x:54))))
      ((((lambda:55 (n:55)
          ((n:55 (lambda:55 (x:55) (lambda:56 (a:56) (lambda:56 (b:56) b:56)))) (
              lambda:57 (a:57) (lambda:57 (b:57) a:57))))
         ((lambda:58 (n:58)
            ((lambda:59 (p:59) (p:59 (lambda:60 (a:60) (lambda:60 (b:60) a:60))))
             ((n:58
               (lambda:61 (p:61)
                 (((lambda:62 (x:62) (lambda:62 (y:62) (lambda:62 (f:62) ((f:62 x
                     :62) y:62))))
                   ((lambda:63 (p:63) (p:63 (lambda:64 (a:64) (lambda:64 (b:64) b
                       :64)))) p:61))
                  ((lambda:65 (n:65) (lambda:65 (f:65) (lambda:65 (x:65) (f:65 ((n
                      :65 f:65) x:65)))))
                   ((lambda:66 (p:66) (p:66 (lambda:67 (a:67) (lambda:67 (b:67) b
                       :67)))) p:61)))))
              (((lambda:68 (x:68) (lambda:68 (y:68) (lambda:68 (f:68) ((f:68 x:68)
                  y:68))))
               (lambda:69 (f:70) (lambda:69 (x:70) x:70)))
               (lambda:71 (f:72) (lambda:71 (x:72) x:72))))))
          x))
        (lambda () (lambda:73 (f:74) (lambda:73 (x:74) (f:74 x:74)))))
       (lambda ()
         (((lambda:76 (m:76)
             (lambda:76 (n:76) ((m:76 (lambda:77 (n:77) (lambda:77 (f:77) (lambda
                 :77 (x:77) (f:77 ((n:77 f:77) x:77)))))) n:76)))
           ((f)
            ((lambda:78 (n:78)
               ((lambda:79 (p:79) (p:79 (lambda:80 (a:80) (lambda:80 (b:80) a:80)))
```

```
        )
      ((n:78
        (lambda:81 (p:81)
          (((lambda:82 (x:82) (lambda:82 (y:82) (lambda:82 (f:82) ((f:82 x
              :82) y:82))))
            ((lambda:83 (p:83) (p:83 (lambda:84 (a:84) (lambda:84 (b:84) b
                :84)))) p:81))
           ((lambda:85 (n:85) (lambda:85 (f:85) (lambda:85 (x:85) (f:85 ((
              n:85 f:85) x:85)))))
            ((lambda:86 (p:86) (p:86 (lambda:87 (a:87) (lambda:87 (b:87) b
                :87)))) p:81)))))
       (((lambda:88 (x:88) (lambda:88 (y:88) (lambda:88 (f:88) ((f:88 x
           :88) y:88))))
         (lambda:89 (f:90) (lambda:89 (x:90) x:90)))
        (lambda:91 (f:92) (lambda:91 (x:92) x:92))))))
   x)))
((f)
 ((lambda:93 (n:93)
    ((lambda:94 (p:94) (p:94 (lambda:95 (a:95) (lambda:95 (b:95) a:95))))
     ((n:93
       (lambda:96 (p:96)
         (((lambda:97 (x:97) (lambda:97 (y:97) (lambda:97 (f:97) ((f:97 x
             :97) y:97))))
           ((lambda:98 (p:98) (p:98 (lambda:99 (a:99) (lambda:99 (b:99) b
               :99)))) p:96))
          ((lambda:100 (n:100) (lambda:100 (f:100) (lambda:100 (x:100) (f
              :100 ((n:100 f:100) x:100)))))
           ((lambda:101 (p:101) (p:101 (lambda:102 (a:102) (lambda:102 (b
               :102) b:102)))) p:96)))))
      (((lambda:103 (x:103) (lambda:103 (y:103) (lambda:103 (f:103) ((f
          :103 x:103) y:103))))
        (lambda:104 (f:105) (lambda:104 (x:105) x:105)))
       (lambda:106 (f:107) (lambda:106 (x:107) x:107))))))
  ((lambda:108 (n:108)
     ((lambda:109 (p:109) (p:109 (lambda:110 (a:110) (lambda:110 (b:110)
         a:110))))
      ((n:108
        (lambda:111 (p:111)
          (((lambda:112 (x:112) (lambda:112 (y:112) (lambda:112 (f:112) ((
              f:112 x:112) y:112))))
            ((lambda:113 (p:113) (p:113 (lambda:114 (a:114) (lambda:114 (b
                :114) b:114)))) p:111))
           ((lambda:115 (n:115) (lambda:115 (f:115) (lambda:115 (x:115) (f
              :115 ((n:115 f:115) x:115)))))
            ((lambda:116 (p:116) (p:116 (lambda:117 (a:117) (lambda:117 (b
                :117) b:117)))) p:111)))))
       (((lambda:118 (x:118) (lambda:118 (y:118) (lambda:118 (f:118) ((f
           :118 x:118) y:118))))
         (lambda:119 (f:120) (lambda:119 (x:120) x:120)))
        (lambda:121 (f:122) (lambda:121 (x:122) x:122))))))
   x)))))))))))
```

9 Impurities

Unfortunately, there is no way to output to the console using pure lambda expressions. Because of this, I had to define a series of functions that convert from the church definitions to definitions that scheme can output readably to the console. This means our programs can't actually be made out of pure lambdas, but we can restrict these functions so only be used for printing to the console. Therefore, all computation will still only be performed by lambda expressions, and the extraneous functions only serve the purpose of human readable output, not calculation.