# MAPREDUCE, CHURCH NUMERALS 10

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

July 7, 2010

## 1 Generic Operators: Dyadic Operations

Our shape example is easier than the arithmetic example in the book because our operations only require one operand, not two. For arithmetic operations like +, it's not good enough to connect the operation with a type; the two operands might have two different types.

What should you do if you have to add a rational number to a complex number? There is no perfect solution to this problem.

For the particular case of arithmetic, we're lucky in that the different types form a sequence of larger and larger sets. Every integer is a rational number; every rational is a real; every real is a complex. So we can deal with type mismatch by raising the less-complicated operand to the type of the other one. To add a rational number to a complex number, raise the rational number to complex and then you're left with the problem of adding two complex numbers. So we only need $n$ addition algorithms, not $n^2$algorithms, where $n$ is the number of types. Do we need $n^2$ raising algorithms? No, because we don't have to know directly how to raise a rational number to complex. We can raise the rational number to the next higher type (real), and then raise that real number to complex. So if we want to add $1$, $3$, and $2 + 5i$ the answer comes out $2.3333 + 5i$. As this example shows, nonchalant raising can lose information. It would be better, perhaps, if we could get the answer $\frac{7}{3} + 5i$ instead of the decimal approximation.

Numbers are a rat's nest full of traps for the unwary. You will live longer if you only write programs about integers.

# 2 MapReduce

## 2.1 Background

In the past, functional programming, and higher-order functions in particular, have been considered esoteric and unimportant by most programmers. But the advent of highly parallel computation is changing that, because functional programming has the very useful property that the different pieces of a program don't interfere with each other, so it doesn't matter in what order they are invoked. Later this semester, when we have more sophisticated functional mechanisms to work with, we'll be examining one famous example of functional programming at work: the `MapReduce` programming paradigm developed by Google that uses higher-order functions to allow a programmer to process large amount of data in parallel on many computers.

Much of the computing done at Google consists of relatively simple algorithms applied to massive amounts of data, such as the entire World Wide Web. It's routine for them to use clusters consisting of many thousands of processors, all running the same program, with a distributed filesystem that gives each processor local access to part of the data.

In 2003 some very clever people at Google noticed that the majority of these computations could be viewed as a `map` of some function over the data followed by an `accumulate` (they use the name `reduce`, which is a synonym for this function) to collect the results. Although each program was conceptually simple, a lot of programmer effort was required to manage the parallelism; every programmer had to worry about things like how to recover from a processor failure (virtually certain to happen when a large computation uses thousands of machines) during the computation. They wrote a library procedure named `MapReduce` that basically takes two functions as arguments, a one-argument function for the `map` part and a two-argument function for the `accumulate` part. (The actual implementation is more complicated, but this is the essence of it.) Thus, only the implementors of `MapReduce` itself had to worry about the parallelism, and application programmers just have to write the two function arguments. `MapReduce` is a little more complicated than just

```
(define (mapreduce mapper reducer base-case data) ; Nope.
   (accumulate reducer base-case (map mapper data)))
```

because of the parallelism. The input data comes in pieces; several computers run the `map` part in parallel, and each of them produces some output. These intermediate results are rearranged into groups, and each computer does a `reduce` of part of the data. The final result isn't one big list, but separate output files for each reducing process.

To make this a little more specific, today we'll see a toy version of the algorithm that just handles small data lists in one processor.

Data pass through the program in the form of *key-value pairs:*

```
(define make-kv-pair cons)
(define kv-key car)
(define kv-value cdr)
```

A list of key-value pairs is called an *association list* or *a-list* for short. We'll see a-lists in many contexts other than `MapReduce`. Conceptually, the input to `MapReduce` is an a-list, although in practice there are several a-lists, each on a different processor.

Any computation in `MapReduce` involves two function arguments: the *mapper* and the *reducer*. (Note: The Google `MapReduce` paper in the course reader says "the `map` function" to mean the function that the user writes, the one that's applied to each datum; this usage is confusing since everyone else uses "`map`" to mean

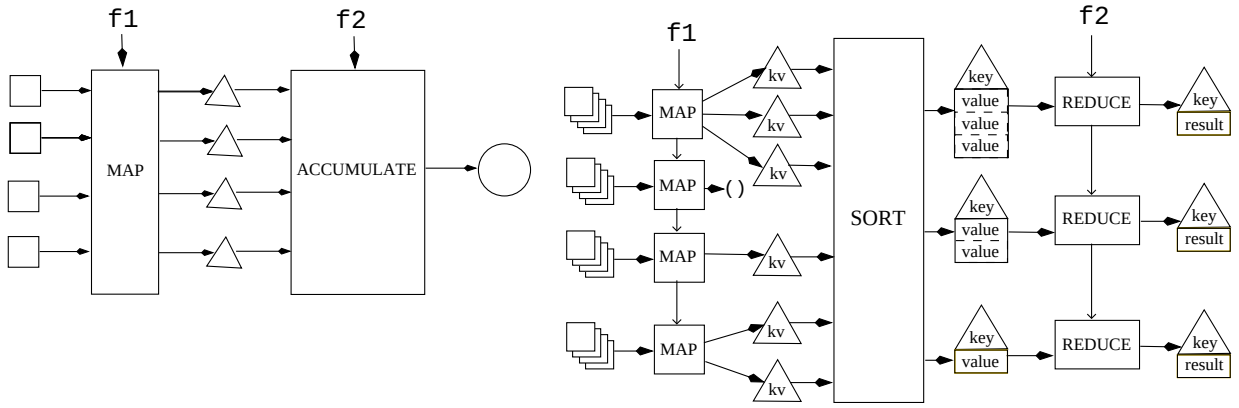
the higher-order function that controls the invocation of the user's function, so we're calling the latter the *mapper*:

```
(map mapper data)
```

Similarly, we'll use `reduce` to refer to the higher-order function, and `reducer` to mean the user's accumulation function.

## 2.2 Implementation

`(accumulate f2 base (map f1 data))` `(mapreduce f1 f2 base dataname)`




The argument to the mapper is always one kv-pair. Keys are typically used to keep track of where the data came from. For example, if the input consists of a bunch of Web pages, the keys might be their URLs. Another example we'll be using is Project Gutenberg, an online collection of public-domain books; there the key would be the name of a book (more precisely, the filename of the file containing that book). In most uses of a-lists, there will only be one kv-pair with a given key, but that's not true here; for example, each line of text in a book or Web page might be a datum, and every line in the input will have the same key.

The value returned by the mapper must be *a list of* kv-pairs. The reason it's a list instead of a single kv-pair, as you might expect, is twofold. First, a single input may be split into smaller pieces; for example, a line of text might be mapped into a separate kv-pair for each word in the line. Second, the mapper might return an empty list, if this particular kv-pair shouldn't contribute to the result at all; thus, the mapper might also be viewed as a filterer. The mapper is not required to use the same key in its output kv-pairs that it gets in its input kv-pair.

Since `map` handles each datum independently of all the others, the fact that many `map`s are running in parallel doesn't affect the result; we can model the entire process with a single `map` invocation. That's not the case with the `reduce` part, because the data are being combined, so it matters which data end up on which machine. This is where the keys are most important. Between the mapping and the reduction is an intermediate step in which the kv-pairs are sorted based on the keys, and all pairs with the same key are reduced together. Therefore, the reducer doesn't need to look at keys at all; its two arguments are a value and the result of the partial accumulation of values already done. In many cases, just as in the accumulations we've seen earlier, the reducer will be a simple associative and commutative operation such as `+`.

The overall result is an a-list, in which each key occurs only once, and the value paired with that key is the result of the `reduce` invocation that handled that key. The keys are guaranteed to be in order. (This is the result of the 61A version of `MapReduce`; the real Google software has a more complicated interface

because each computer in the cluster collects its own `reduce` results, and there are many options for how the reduction tasks are distributed among the processors. You'll learn more details in later courses.) So in today's single-processor simulation, instead of talking about `reduce` we'll use a higher order function called `groupreduce` that takes a *list of a-lists* as argument, with each sublist having kv-pairs with the same key, does a separate reduction for each sublist, and returns an a-list of the results. So a complete `MapReduce` operation works roughly like this:

```
(define (mapreduce mapper reducer base-case data) ; handwavy approximation
  (groupreduce reducer base-case
               (sort-into-buckets (map mapper data))))
(define (groupreduce reducer base-case buckets)
  (map (lambda (subset) (make-kv-pair
                         (kv-key (car subset))
                         (reduce reducer base-case (map kv-value subset))))
       buckets))
```

As a first example, we'll take some grades from various exams and add up the grades for each student. This example doesn't require `map`. Here's the raw data:

```
(define mt1 '((cs61a-xc . 27) (cs61a-ya . 40) (cs61a-xw . 35) (cs61a-xd . 38)
              (cs61a-yb . 29) (cs61a-xf . 32)))
(define mt2 '((cs61a-yc . 32) (cs61a-xc . 25) (cs61a-xb . 40) (cs61a-xw . 27)
              (cs61a-yb . 30) (cs61a-ya . 40)))
(define mt3 '((cs61a-xb . 32) (cs61a-xk . 34) (cs61a-yb . 30) (cs61a-ya . 40)
              (cs61a-xc . 28) (cs61a-xf . 33)))
```

Each midterm in this toy problem corresponds to the output of a parallel `map` operation in a real problem.

First we combine these into one list, and use that as input to the `sortintobuckets` procedure:

```
> (sort-into-buckets (append mt1 mt2 mt3))
(((cs61a-xb . 40) (cs61a-xb . 32))
 ((cs61a-xc . 27) (cs61a-xc . 25) (cs61a-xc . 28))
 ((cs61a-xd . 38))
 ((cs61a-xf . 32) (cs61a-xf . 33))
 ((cs61a-xk . 34))
 ((cs61a-xw . 35) (cs61a-xw . 27))
 ((cs61a-ya . 40) (cs61a-ya . 40) (cs61a-ya . 40))
 ((cs61a-yb . 29) (cs61a-yb . 30) (cs61a-yb . 30))
 ((cs61a-yc . 32)) )
```

In the real parallel context, instead of the `append`, each `map` process would sort its own results into the right buckets, so that too would happen in parallel.

Now we can use `groupreduce` to add up the scores in each bucket separately:

```
> (groupreduce + 0 (sort-into-buckets (append mt1 mt2 mt3)))
((cs61a-xb . 72) (cs61a-xc . 80) (cs61a-xd . 38) (cs61a-xf . 65)
(cs61a-xk . 34) (cs61a-xw . 62) (cs61a-ya . 120) (cs61a-yb . 89)
(cs61a-yc . 32))
```

Note that the returned list has the keys in sorted order. This is a consequence of the sorting done by `sort-into-buckets`, and also, in the real parallel `mapreduce`, a consequence of the order in which keys are assigned to processors (the "partitioning function" discussed in the `MapReduce` paper).

Similarly, we could ask *how many* midterms each student took:

```
> (groupreduce (lambda (new old) (+ 1 old)) 0
               (sort-into-buckets (append mt1 mt2 mt3)))
((cs61a-xb . 2) (cs61a-xc . 3) (cs61a-xd . 1) (cs61a-xf . 2)
 (cs61a-xk . 1) (cs61a-xw . 2) (cs61a-ya . 3) (cs61a-yb . 3)
 (cs61a-yc . 1))
```

We could combine these in the obvious way to get the average score per student, for exams actually taken.

### 2.3 Word Frequency Counting

A common problem is to look for commonly used words in a document. For starters, we'll count word frequencies in a single sentence. The first step is to turn the sentence into key-value pairs in which the key is the word and the value is always 1:

```
> (map (lambda (wd) (list (make-kv-pair wd 1))) '(cry baby cry))
((cry . 1) (baby . 1) (cry . 1))
```

If we group these by key and add the values, we'll get the number of times each word appears.

```
(define (wordcounts1 sent)
  (groupreduce + 0 (sort-into-buckets (map (lambda (wd) (make-kv-pair wd 1))
                                           sent))))
> (wordcounts1 '(cry baby cry)) ((baby . 1) (cry . 2))
```

Now to try the same task with (simulated) files. When we use the real `mapreduce`, it'll give us file data in the form of a key-value pair whose key is the name of the file and whose value is a line from the file, in the form of a sentence. For now, we're going to simulate a file as a list whose car is the "filename" and whose cdr is a list of sentences, representing the lines of the file. In other words, a file is a list whose first element is the filename and whose remaining elements are the lines.

```
(define filename car)
(define lines cdr)
```

Here's some data for us to play with:

```
(define file1 '((please please me) (i saw her standing there) (misery)
                (anna go to him) (chains) (boys) (ask me why)
                (please please me) (love me do) (ps i love you)
                (baby its you) (do you want to know a secret)))
(define file2 '((with the beatles) (it wont be long) (all ive got to do)
                (all my loving) (dont bother me) (little child)
                (till there was you) (roll over beethoven) (hold me tight)
                (you really got a hold on me) (i wanna be your man)
                (not a second time)))
(define file3 '((a hard days night) (a hard days night)
                (i should have known better) (if i fell)
                (im happy just to dance with you) (and i love her)
                (tell me why) (cant buy me love) (any time at all)
                (ill cry instead) (things we said today) (when i get home)
                (you cant do that) (ill be back)))
```

We start with a little procedure to turn a "file" into an a-list in the form `mapreduce` will give us:

```
(define (file->linelist file)
  (map (lambda (line) (make-kv-pair (filename file) line))
    (lines file)))
> (file->linelist file1)
(((please please me) i saw her standing there)
 ((please please me) misery)
 ((please please me) anna go to him)
 ((please please me) chains)
 ((please please me) boys)
 ((please please me) ask me why)
 ((please please me) please please me)
 ((please please me) love me do)
 ((please please me) ps i love you)
 ((please please me) baby its you)
 ((please please me) do you want to know a secret))
```

Note that `((please please me) misery)` is how Scheme prints the kv-pair:

`((please please me) . (misery)).`

Now we modify our `wordcounts1` procedure to accept such kv-pairs:

```
(define (wordcounts files)
  (groupreduce + 0 (sort-into-buckets
                    (flatmap (lambda (kv-pair)
                               (map (lambda (wd) (make-kv-pair wd 1))
                                    (kv-value kv-pair)))
                             files))))
> (wordcounts (append (file->linelist file1)
                      (file->linelist file2)
                      (file->linelist file3)))
((a . 4) (all . 3) (and . 1) (anna . 1) (any . 1) (ask . 1) (at . 1) (baby . 1)
(back . 1) (be . 3) (beethoven . 1) (better . 1) (bother . 1) (boys . 1)
(buy . 1) (cant . 2) (chains . 1) (child . 1) (cry . 1) (dance . 1) (days . 1)
(do . 4) (dont . 1) (fell . 1) (get . 1) (go . 1) (got . 2) (happy . 1)
(hard . 1) (have . 1) (her . 2) (him . 1) (hold . 2) (home . 1) (i . 7) (if . 1)
(ill . 2) (im . 1) (instead . 1) (it . 1) (its . 1) (ive . 1) (just . 1)
(know . 1) (known . 1) (little . 1) (long . 1) (love . 4) (loving . 1) (man . 1)
(me . 8) (misery . 1) (my . 1) (night . 1) (not . 1) (on . 1) (over . 1)
(please . 2) (ps . 1) (really . 1) (roll . 1) (said . 1) (saw . 1) (second . 1)
(secret . 1) (should . 1) (standing . 1) (tell . 1) (that . 1) (there . 2)
(things . 1) (tight . 1) (till . 1) (time . 2) (to . 4) (today . 1) (wanna . 1)
(want . 1) (was . 1) (we . 1) (when . 1) (why . 2) (with . 1) (wont . 1)
(you . 7) (your . 1))
```

(If you count yourself to check, remember that words in the album titles don't count! They're keys, not values.)

Note the call to `flatmap` above. In a real `mapreduce`, each file would be mapped on a different processor, and the results would be distributed to `reduce` processes in parallel. Here, the `map` over files gives us a list of a-lists, one for each file, and we have to append them to form a single a-list. `Flatmap` flattens (appends) the results from calling `map`.

We can postprocess the groupreduce output to get an overall reduction to a single value:

```
(define (mostfreq files)
  (accumulate (lambda (new old)
                (cond ((> (kv-value new) (kv-value (car old)))
                       (list new))
                      ((= (kv-value new) (kv-value (car old)))
                       (cons new old)) ; In case of tie, remember both.
                      (else old)))
              (list (make-kv-pair 'foo 0)) ; Starting value.
              (groupreduce + 0 (sort-into-buckets
                                (flatmap (lambda (kv-pair)
                                           (map (lambda (wd)
                                                  (make-kv-pair wd 1))
                                                (kv-value kv-pair)))
                                         files)))))
> (mostfreq (append (file->linelist file1)
                    (file->linelist file2)
                    (file->linelist file3)))
((me . 8))
```

(Second place is "you" and "I" with 7 appearances each, which would have made a two-element a-list as the result.) If we had a truly enormous word list, we'd put it into a distributed file and use another `mapreduce` to find the most frequent words of subsets of the list, and then find the most frequent word of those most frequent words.

### 2.4 Searching for a Pattern

Another task is to search through files for lines matching a pattern. A *pattern* is a sentence in which the word `*` matches any set of zero or more words:

```
> (match? '(* i * her *) '(i saw her standing there))
#t
> (match? '(* i * her *) '(and i love her))
#t
> (match? '(* i * her *) '(ps i love you))
#f
```

Here's how we look for lines in files that match a pattern:

```
(define (grep pattern files)
  (groupreduce cons '()
               (sort-into-buckets
                 (flatmap (lambda (kv-pair)
                            (if (match? pattern (kv-value kv-pair))
                                (list kv-pair)
                                '()))
                          files))))
> (grep '(* i * her *) (append (file->linelist file1)
                               (file->linelist file2)
                               (file->linelist file3)))
(((a hard days night) (and i love her))
 ((please please me) (i saw her standing there)))
```

### 2.5 Summary

The general pattern here is:

```
(groupreduce reducer base-case (sort-into-buckets (map-or-flatmap mapper data)))
```

This corresponds to

```
(mapreduce mapper reducer base-case data)
```

in the truly parallel `mapreduce` exploration we'll be doing later.

# 3 Church Numerals

This is something I think is incredibly, incredibly cool. A few weeks ago, I talked to you about how `lambda` can be used to implement everything in Scheme. When I said that, I don't know if you believed me. But I am going to do a demo of a few things. Before I begin, I want to warn you, nothing in this section will show up in any exams in this course. I do however, think this is something very very cool.

Let's think about numbers in a whole new way. Lets think about the number 1 as defined as something like `(1+ 0)`. Therefore, `(1+ (1+ 0))` is `2`. We can generalize this to just calling any function, and define `(f x) = 1`, and `(f (f x)) = 2`.

Lets start!

```
(define one (lambda (f) (lambda (x) (f x))))
 define two (lambda (f) (lambda (x) (f (f x)))))
(define three (lambda (f) (lambda (x) (f (f (f x))))))
```

Or, equivalently,

```
(define one (lambda (f) f))
```

So, a number is a *function of a function* (not surprising, since functions are all we have). The argument `x` to the returned function is, of course, also a function, but it'll usually be a function that represents a number, whereas `f` might be a function *whose domain* is numbers, such as `1+`. This business of keeping track of the *abstract* type of various things all of which are, concretely, just functions, is what makes this a hard thing to understand.

Now let's define arithmetic:

```
(define (plus a b) (lambda (f) (lambda (x) ((a f) ((b f) x)))))
(define (times a b) (lambda (f) (a (b f))))
(define (expt a b) (b a))
```

Just to test our code, we can invoke these on f being 1+ and x being 0:

```
(define (try num)                  ; convert Church to
  ((num (lambda (x) (+ x 1))) 0)) ; ordinary number
```

Before we get to subtraction, let's do conditionals and pairs first. We'll see that we'll use them to implement subtraction.

Let's define `true` and `false`.

```
(define true (lambda (a b) a))
(define false (lambda (a b) b))
```

Now that we have those, let's define `if`:

```
(define (if pred t f) (pred t f))
```

Why should this work? Substitute TRUE for PRED and you get this:

```
((lambda (a b) a) t f)
```

Now let's write a predicate:

```
(define (zero? num)
  (lambda (a b)
    ((num (lambda (x) b))
     a)))
```

What does this mean? A predicate takes the form

```
(lambda (a b) [expression that returns A to mean true and B to mean false])
```

The function `(LAMBDA (X) B)` ignores its argument `X`, and always returns `B`. Composing this function with itself, any number of times, still gives a function that ignores its argument and returns `B`. So `(NUM (LAMBDA (X) B))`, which composes `(LAMBDA (X) B)` with itself `NUM` times, is a function that returns `B` *unless* `NUM` is zero. Recall that zero is defined this way:

```
(define zero (lambda (f) (lambda (x) x)))
```

So if `num` is zero, its argument (`f` in the expression above, which is `(LAMBDA (X) B)` in our case) is ignored, and `NUM` returns the identity function.

So, what are we doing with the result of `(NUM (LAMBDA (X) B))`? We're invoking it with the argument `A`! So if `NUM` isn't ZERO, the result will be `B`, and if `num` is `zero`, the result will be `A`. This is just what we needed to test whether `num` is `zero`!

We'll invent more predicates later, but this will do for now.

Here's an equivalent to TRY for debugging conditionals:

```
(define (trypred pred) ; convert Church to
   (pred #t #f))       ; ordinary predicate
```

For example:

```
> (trypred (zero? three)) #F
```

For the next step we must invent pairs (as a way to provide two arguments to a one-argument function):

```
(define (kons a b) (lambda (msg) (msg a b)))
(define (kar pair) (pair (lambda (a b) a)))
(define (kdr pair) (pair (lambda (a b) b)))
```

This is like the redefinition of pairs shown in lecture, except that instead of the symbols `car` and `cdr`, we use functions as the messages, namely the function that returns its first argument for `car`, and the one that returns its second argument for `cdr`.

You'll notice that this implementation of pairs is exactly like the implementation of conditionals above! Pairs and conditionals are two abstract data types implemented the same way, just like many such examples in the text and lectures.

Now we can use pairs to subtract 1 from a number. This is the hardest thing to invent, the core of the whole project:

```
(define (-1 num)
  (lambda (f)
    (lambda (x)
      (kdr ((num (lambda (pair) (kons f ((kar pair) (kdr pair)))))
      (kons (lambda (z) z) x))))))
```

What does this mean? Well, basically, since a number is a function, the only thing we can do to it is invoke it on some argument. So we're invoking NUM on a function whose domain and range are both pairs. And what the number does to that function is invoke it repeatedly on some argument. Here's a table showing the result:

| | car of pair | cdr of pair |
|---|---|---|
| initial argument | identity | `x` |
| after 1 invocation | `f` | `(id x) = x` |
| after 2 invocations | `f` | `(f x)` |
| after 3 invocations | `f` | `(f (f x))` |
| after 4 invocations | `f` | `(f (f (f x)))` |

...so after NUM invocations, we've composed NUM-1 invocations of F. That's exactly what `(-1 num)` means. The trick is that we call `(kar pair)` `num` times, not `num-1` times, but we nullify the first invocation because that first time, `(kar pair)` is the identity function. All the other times, `(kar pair)` is `f`.

Once we can subtract 1 from a number, we can subtract anything by repeatedly subtracting 1:

```
(define (minus x y) ((y -1+) x))
```

Beyond this point we need recursion, so we invent the Y combinator:

```
(define (Y f) (lambda (x) (f f x)))
(define (YY f) (lambda (x y) (f f x y)))
```

And now we can do just about anything straightforwardly:

```
(define =?
  (YY (lambda (f x y)
    (if (zero? x)
        (if (zero? y) true false)
        (if (zero? y) false
                      (f f (-1+ x) (-1+ y)))))))
(define fact
  (Y (lambda (f n)
             (if (zero? n)
                 one
                 (times n (f f (-1+ n)))))))
```

The Boolean functions (as regular procedures, not special forms):

```
(define (both p q)   ; and
  (p q false))
(define (either p q) ; or
  (p true q))
(define (knot p)
  (p false true))
```

Here we take advantage of the fact that minus returns zero if the result should be negative:

```
(define (lesseq a b) (zero? (minus a b)))
(define (greatereq a b) (lesseq b a))
```

```
(define (less a b) (both (lesseq a b) (knot (=? a b))))
(define (greater a b) (less b a))
```

Alternatively, we can use lesseq to create a non-recursive equal test:

```
(define (equal a b) (both (lesseq a b) (greatereq a b)))
(define (less a b) (both (lesseq a b) (knot (greatereq a b))))
```

Technically, we're cheating by using `define`, but if we really use just `lambda`, the resulting code is unreadable. Here's `(fact five)`:

```
> (try
   (((lambda (f) (lambda (x) (f f x)))            ; ((Y
     (lambda (f n)                                ; (lambda (f n)
       (((lambda (pred t f) (pred t f))           ; (if
   ((lambda (num)                                 ; (zero?
     (lambda (a b)
       ((num (lambda (x) b))
        a)))
    n)                                            ; n)
   (lambda () (lambda (f) f))                     ; one
   (lambda () ((lambda (a b)                      ; (times
           (lambda (f) (a (b f))))
         n                                        ; n
         (f f ((lambda (num)                      ; (f f (-1+
           (lambda (f)
             (lambda (x)
         ((lambda (pair) (pair (lambda (a b) b)))
          ((num (lambda (pair)
            ((lambda (a b)
               (lambda (msg) (msg a b)))
             f
             (((lambda (pair)
                 (pair (lambda (a b) a)))
               pair)
              ((lambda (pair)
                 (pair (lambda (a b) b)))
               pair)))))
           ((lambda (a b)
              (lambda (msg) (msg a b)))
            (lambda (z) z)
            x))))))
       n))))))))                                  ; n))))))
(lambda (f) (lambda (x) (f (f (f (f (f x)))))))) ; five
```