SICP 2.6: Church Numerals (2011) (jlongster.com)

59 points by poppingtonic on May 11, 2014 | hide | past | web | favorite | 18 comments

gshubert17 on May 11, 2014 [-]

Tom Stuart used various Church encodings in his presentation "Programming with Nothing" [1] [2]. He used nothing in the Ruby language but creating procs and calling procs. He demonstrated this by implementing the fizz-buzz program in nothing but procs.

[1] http://codon.com/programming-with-nothing

[2] http://computationbook.com

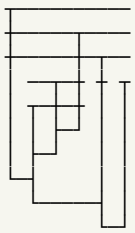
brianmwaters_hn on May 11, 2014 [-]

What's really interesting beyond just type coercion and "increment," is the way addition, multiplication, and exponentiation were implemented by Church and Rosser back in the 30's (see https://en.wikipedia.org/wiki/Lambda_calculus). Rather than using SICP-style recursion to count down to zero and repeatedly apply inc/add/mul, they took advantage of some of the special properties of Church numerals. A year later, and I still have not really wrapped my head around their solutions. Maybe you will fare better.

For a real challenge, try to implement the decrement function. Legend has it, Rosser thought of the solution while on the dentist's chair. (It is a bit heavier on code than the other Church numeral functions, but conceptually simple - and was a creative idea at the time, before functional programming had really come into being.)

tromp on May 12, 2014 [-]

The graphical notation for the decrement function



makes for an artistic picture...

lisper on May 12, 2014 [-]

Here's what the factorial function on Church numerals ends up looking like if you expand it out:

```
 ((λ (f) ((λ (g) (g g)) (λ (h) (λ (x) ((f (h h)) x)))))
(λ (f)
   (λ (n)
      (if ((λ (p a b) (p a b)) ((λ (n) (n (λ (x) (λ (x y) y)) (λ (x y) x))) n) t nil)
        ((λ (n) (λ (f x) (f (n f x)))) (λ (f x) x))
        ((λ (m n) (m ((λ (m n) (λ (f x) (m f (n f x)))) n) (λ (f x) x)))
         n (f ((λ (n) (λ (f x) (n (λ (g h) (h (g f))) (λ (u) x) (λ (u) u)))) n)))))))
```

Note the use of the Y combinator to implement recursion :-)

Here it is in action computing the factorial of 6:

```
 ? (((λ (f) ((λ (g) (g g)) (λ (h) (λ (x) ((f (h h)) x)))))
(λ (f)
   (λ (n)
      (if ((λ (p a b) (p a b)) ((λ (n) (n (λ (x) (λ (x y) y)) (λ (x y) x))) n) t nil)
        ((λ (n) (λ (f x) (f (n f x)))) (λ (f x) x))
        ((λ (m n) (m ((λ (m n) (λ (f x) (m f (n f x)))) n) (λ (f x) x)))
         n (f ((λ (n) (λ (f x) (n (λ (g h) (h (g f))) (λ (u) x) (λ (u) u)))) n)))))))
((λ (n) (λ (f x) (f (n f x))))
 ((λ (n) (λ (f x) (f (n f x))))
  ((λ (n) (λ (f x) (f (n f x))))
   ((λ (n) (λ (f x) (f (n f x))))
    ((λ (n) (λ (f x) (f (n f x))))
     ((λ (n) (λ (f x) (f (n f x))))
      (λ (f x) x)))))))
#<COMPILED-LEXICAL-CLOSURE #x30200105130F>
? (funcall * '1+ 0)
720
```

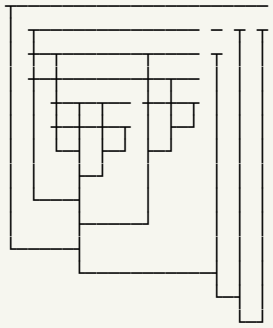
tromp on May 12, 2014 [-]

Church factorial can be implemented without recursion though.

Hint: consider a function that maps the pair <n,n!> to <(n+1),(n+1)!> ...

▲ tromp on May 12, 2014 [-]

Here is Church factorial in graphical notation:



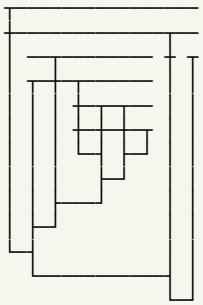
▲ lisper on May 12, 2014 [-]

That's pretty cool looking, but I have no idea what I'm looking at. Does this notation have a name? Is it documented somewhere? (Hm, I wonder how hard it would be to write a parser for it. :-)

▲ tromp on May 12, 2014 [-]

That's my graphical notation for lambda calculus, documented at http://www.cwi.nl/~tromp/cl/diagrams.html

Btw, here's an even shorter version of factorial:



In de-Bruijn notation it is \\2 (\\1 (2 (\\3 2 (2 1)))) (\2) (\1), which in conventional notation is \n\f.n (\x\m.m (x (\a\b.m a (a b)))) (\x.f) (\x.x)

▲ lisper on May 14, 2014 [-]

BTW, is there a description of how this works written up anywhere? I tried to reverse-engineer it and got lost in lambdas. (I did manage to translate it into Scheme and verify empirically that it works!)

▲ lisper on May 14, 2014 [-]

Wow! That is wicked-cool! Thanks!

▲ lisper on May 12, 2014 [-]

Whether the loop is recursive or iterative you still need a loop, so you still need a Y combinator.

▲ tromp on May 12, 2014 [-]

You don't need Y if you apply a Church numeral instead, which gives you a bounded iteration.

▲ lisper on May 13, 2014 [-]

Ah, good point!

▲ dizzystar on May 12, 2014 [-]

Oh man. I worked through SICP when I was just starting to program. Little did I know that no other book would be so full of gems and enlightenment like this[0]. Some of the stuff stopped me dead and caused me to daze off for several minutes.

What's amazing is that I feel like I didn't fully "get" a large portion of the book. I could work through it again in a few years and learn something from every page.

[0]Other books come close, but the impact of this book will stay with me forever.

bridger on May 12, 2014 [-]

When I was doing this exercise with a friend on Understudy, he found solutions to multiplication and exponentiation too. I'm still trying to wrap my head around the exponentiation. Passing a number to another number?!

```
(define (multiply number1 number2)
  (lambda (f)
    (number1 (lambda (x) ((number2 f) x)))))

(define (exponentiate base exponent)
  (exponent base))
```

rspeer on May 12, 2014 [-]

Let's use "twice" as the name of Church numeral 2, and "thrice" as the name of numeral 3.

If you look at the input and output of thrice, it looks like this:

```
(function that gets applied once) -> (function that gets applied 3 times)
```

What if we used the output as the input again?

```
(function that gets applied once) -> (function that gets applied 3 times) -> (function that gets applied 9 times)
```

We have a name for what we just did to the "thrice" function, and that name is "twice". So that's (twice thrice), and we can see it makes something happen 3^2 = 9 times.

iano on May 12, 2014 [-]

Along the same vane from the math world is how natural numbers (which can then be used to build the reals, etc) are represented using only empty sets. Pretty awesome.

See http://en.wikipedia.org/wiki/Set-theoretic_definition_of_nat...

raphaelj on May 11, 2014 [-]

This exercise was an enlightening when I read it for the first time.

I came just after 2.5 which was already an awesome magic trick.