The Lambda Calculus

notes by Don Blaheta

October 12, 2000

"A little bondage is always a good thing." —sk

We've seen that Scheme is, as languages go, pretty small. There are just a few keywords, and most of the utility of the language is inherent in its minimal, unornamented structure, unlike, say, "public static void main" Java. We can come up with a short list of fundamental Scheme concepts:

`if`	`cdr`	`false`
`lambda`	numbers	`(`
`cons`	arithmetic	`)`
`define`	symbols	
`car`	`true`	

Without motivating it too deeply, let's consider how we could reduce this list further, by implementing some of these features in terms of others.

Warming up: booleans and conditionals

In Scheme, what is the basic way of making conditional statements?

```
(if C T F)
```

There is a form that begins with the keyword `if`, and has three arguments, a test expression *C*, and things to do depending on the value of *C*. *C* is a boolean, and as such can have two possible values, true and false. How might we represent these? Well, these values are making a choice between two values, so why not pass in the values and let them choose?

$$\begin{aligned} \text{true} &\equiv (\lambda(T\ F)\ T) \\ \text{false} &\equiv (\lambda(T\ F)\ F) \end{aligned}$$

Thus true and false are functions with an arity[1] of 2[2] that select one or the other of their arguments. `if` then is simply

$$\texttt{(if } C\ T\ F\texttt{)} \equiv \texttt{(}C\ T\ F\texttt{)}$$

, which applies its first argument to the other two. Note that for the usual 'short-circuiting' behaviour of `if` to work (and it is necessary, in order to properly terminate recursion, avoid dividing by zero, etc.), evaluation must be *lazy*, that is, we don't actually evaluate an expression until we need its value, e.g. to perform arithmetic on it. Since we never need a value from the unused half of the conditional, we never try to evaluate it and the short-circuiting behaviour is preserved.

We've thus eliminated `if`, `true`, and `false` from our list. Thus emboldened, we move on to

Lists

What, really, is a Scheme list? It's a pair, with the first value of the pair representing the first item of the list, and the second half of the pair containing the rest of the list. To use standard LISPish terminology,

$$\begin{aligned}(\text{car } (\text{cons } A\ B)) &\equiv A\\ (\text{cdr } (\text{cons } A\ B)) &\equiv B\end{aligned}$$

So how can we represent a cons? If we think in an object-oriented way for a moment[5], we realise that a list, or a pair, is something that we want to feed two values into, and then be able to pass messages to in order to get those values back out. Well, we can feed two values in as arguments—

$$(\lambda(a\ b)$$

—and then return something that reads messages and uses them to act on the data—

$$(\lambda(m)(m\ a\ b)))\,.$$

[1]The "arity" of a function is simply a fancy word for the number of arguments it takes. Derived from 'bin*ary*', 'tern*ary*', '*n-ary*'.

[2]We can simplify further and make λ only create functions of one argument by *currying*[3] the arguments; for legibility and convenience we will not be doing this here.

[3]To 'curry' a function of n arguments, one makes a function of 1 argument that returns a function of $n-1$ arguments, repeating until $n=1$ at which point you perform the calculation in the original n-ary function. Named after Haskell Curry[4], who did much of the early work with the process, though it was actually invented by a guy named Schönfinkel; presumably "schönfinkeling" was considered too unwieldy a term.

[4]A local—his parents were the founders of Curry College of Boston.

[5]Just make sure to wash your hands afterwards.

Our messages m that we pass in obviously have to be functions, and since they're fed the two halves of the pair as arguments, they just pick one and return it:

$$\begin{aligned} \text{car} &\equiv (\lambda(a\ b)\ a) \\ \text{cdr} &\equiv (\lambda(a\ b)\ b) \end{aligned}$$

Just like true and false! This isn't *too* surprising, though, since the structure is fairly similar: like conditionals, which select between two expressions to evaluate, the list operators are also selecting between two values (the halves of the pair).

That removes `cons`, `car`, `cdr` from our list. Let's move on to something a little more challenging.

Something a little more challenging: Numbers

First, a simplifying assumption: we will only deal with the set of natural numbers. That is, all integers not less than zero, or to put it more usefully, a natural number is either zero or one plus a natural number.

What is a number? A number is a count of stuff. It's a number of things, it's a number of sheep, it's a number of whatever you want it to be. Such as... function applications? Consider

$$\begin{aligned} N &= 0 && \text{0 function applications} \\ &\mid 1 + N && \text{1 more function application than } N \end{aligned}$$

What function should we use? It doesn't really matter. Pick something relevant to the task at hand.

$$\begin{aligned} 0 &\equiv (\lambda(f)\ (\lambda(x)\ x)) \\ 1 &\equiv (\lambda(f)\ (\lambda(x)\ (f\ x))) \\ 2 &\equiv (\lambda(f)\ (\lambda(x)\ (f\ (f\ x)))) \\ N &\equiv (\lambda(f)\ (\lambda(x)\ (f^N\ x))) \end{aligned}$$

This numeric representation is known as a "Church numeral"[6]. To get a Scheme number out of a Church numeral, just use `add1` for f and `0` for x.

Of course, numbers aren't all that useful in and of themselves, until we can do things with them. Like the "succ" function, i.e. `add1`. It takes a Church numeral. What does it return? A Church numeral, which is starts with λf, then λx, and the actual result is f applied one more time:

$$\text{succ} \equiv (\lambda(n)\ (\lambda(f)\ (\lambda(x)\ (f\ ((n\ f)\ x)))))$$

[6]Not an ecclesiastical appelation, but an honorary one—they are named after their discoverer, Alonzo Church, who was a contemporary of Curry, Schönfinkel, Turing, and that whole crowd.

What about addition? Well, we can be a little creative here and use our Church numerals—they apply a function n times, and we have a "plus 1" function (succ), so we take one of the addends a, and use the other addend b to apply the succ function b times:

$$\text{add} \equiv (\lambda(a)\ (\lambda(b)\ ((a\ \text{succ})\ b)))$$

Similar reasoning can be used to get multiplication—$a \times b$ is just adding b, a times, to zero.

$$\text{mult} \equiv (\lambda(a)\ (\lambda(b)\ ((a\ (\text{add}\ b))\ \text{zero})))$$

But we ain't seen nothing yet. Increasing a number is easy ("easy"), because you can just wrap a few more function applications around it. But once you've done that, you have a closure, and since you can't see into a closure, how can you reduce the number of function applications? How in the world can we do the 'pred' function (subtract one)?

A brilliant intuition by Stephen Kleene[7] shows us that it is possible[8]. This fundamental insight goes as follows: ultimately, we want a mapping from numbers to their predecessors; we could view this as a list of pairs. Each pair contains a number and its predecessor (ignoring the base case of zero, for now)

$$\langle 0,0\rangle$$
$$\langle 1,0\rangle$$
$$\langle 2,1\rangle$$
$$\langle 3,2\rangle$$

Well, it's pretty easy to construct one pair if you already have the previous one, so what we need is something that will take the base case of $\langle 0,0\rangle$ and find the nth pair from there. As it happens, the Church numeral itself will do this! Once we have the nth pair, we merely look at the second half to find the value of $n-1$. Thus we have

$$\begin{aligned}\text{pred} \quad = \quad & (\lambda(n)\\ &\quad (cdr\\ &\qquad ((n\ (\lambda(p)\\ &\qquad\qquad (\text{cons}\ (\text{succ}\ (\text{car}\ p))\\ &\qquad\qquad\qquad (\text{car}\ p))))\\ &\qquad (\text{cons}\ 0\ 0))))\end{aligned}$$

And finally, to wrap up the numbers section, we need some numeric test in order to bottom out our recursion. How do we test if something is zero? Well the

[7]Another of those swinging theoreticians of the 1930s. This stuff is way cooler than regular expressions, though that was the field where Kleene's name became more famous.

[8]And indeed, put his advisor, Alonzo Church, back on his track of research in this area.

number 0 is represented by zero function applications, so we need to pass our number a base case that starts out true, and a function that returns false if ever applied. Simply,

$$\begin{aligned}\text{zero?} \quad = \quad & (\lambda(n) \\ & \quad ((n\ (\lambda(\text{dummy})\ \text{false})\ \text{true}))\end{aligned}$$

Good, that knocks numbers and arithmetic off our list; there's not much left:

`lambda`   `define`   symbols   (   )

The parentheses aren't going away, and symbols will be left as an exercise to the reader. That leaves `define` as the only thing not defined in terms of lambda; that is,

What's left? *Recursion.*

Our first instinct is simply to define factorial as

```
fact = (lambda (n)
         (if (zero? n)
             1
             (* n (fact (sub1 n)))))
```

, which (except for the symbols) we've now shown can be entirely reduced to lambdas. Problem is, it doesn't work: `fact` hasn't yet been bound when we use it in the last line. Well, we could do

```
fact = (lambda (n)
         (if (zero? n)
             1
             (* n ((lambda (n)
                     (if (zero? n)
                         1
                         (* n (fact (sub1 n)))))
                   (sub1 n)))))
```

ad nauseam, but of course that's just postponing the problem. However, there is a pattern here, and we know what to do with patterns: lambda-abstract them.

```
((lambda (mk-fact)


      ...
              )
 (lambda (fact)
   (lambda (n)
     (if (zero? n) 1 (* n (fact (sub1 n)))))))
```

So far, now, we've just assigned a name: we'll call those last three lines `mk-fact`. What goes in the ...? Let's first try something like

```
    (mk-fact 💣)
```

, which is to say, 💣 is something we don't want to touch, if we do it'll blow up. Well, if we can't touch the bomb that the symbol `fact` is bound to, we can't do the recursive case—but we can do the base case. So `(mk-fact 💣)` returns a function that correctly calculates the factorial of zero, with unspecified behaviour if you pass it other stuff. Not very useful, but now consider

```
    (mk-fact (mk-fact 💣))
```

First time through, `fact` is now bound to `(mk-fact 💣)`, so if it tries to call `fact` the recursion actually works. But not very well, because that second time through, touching `fact` hits the 💣. All the same, we now have something that correctly calculates the factorial of both zero and one. Ultimately, this isn't any different than the version where we kept rewriting the entire function, but at least it's more succinct and compact.

However. Consider what happens when we try

```
    (mk-fact mk-fact)
```

That is, the entire thing looks like this:

```
((lambda (mk-fact)
   (mk-fact mk-fact))
 (lambda (fact)
   (lambda (n)
     (if (zero? n) 1 (* n (fact (sub1 n)))))))
```

The `mk-fact` in line 1 feeds its value to the two `mk-fact`s in line 2; the second of these becomes the value of `fact` in the function of lines 3–5, and this function is itself bound to `mk-fact`. Nifty, we seem to have set up recursion. Of course, that last line won't work, since `fact` (whose value is `mk-fact`) is expecting a function, not a value like `(sub1 n)`, so let's feed it a function, like say `fact`. Meanwhile, we should rename `fact` since it is no longer a number-to-number function, but a function-to-function function that makes a factorial function; how about a name like "`mk-fact`"?

```
((lambda (mk-fact)
   (mk-fact mk-fact))
 (lambda (mk-fact)
   (lambda (n)
     (if (zero? n) 1 (* n ((mk-fact mk-fact) (sub1 n)))))))
```

Mind-boggling. Why/how does it work? By making sure that anytime we are about to "run out of function", we make another one. Earlier, we noted that if we're taking the factorial of zero, it doesn't matter what the value of mk-fact is—💣 or anything else. And if we follow the recursive case, we run `mk-fact` to generate a function that will do one iteration of `fact`. Inductively, it then works for all cases.

Those last two lines are almost, but not quite, the pretty factorial function we all know and love, but that `(mk-fact mk-fact)` is ugly and doesn't have much to do with the actual factorial. We can abstract that out... and what to name it? Why, `fact`, of course:

```
((lambda (mk-fact)
   (mk-fact mk-fact))
 (lambda (mk-fact)
   ((lambda (fact)
      (lambda (n)
        (if (zero? n)
            1
            (* n (fact (- n 1))))))
    (mk-fact mk-fact))))
```

That is, `(mk-fact mk-fact)` is bound to `fact`, giving us exactly the previous version.[9] But a further modification is desirable: lines 3–5 here are really the meat of the factorial calculation, while lines 1, 2, and 6 are all part of the instrumentation of making the recursion work, and furthermore, those lines could be reused for other functions we want to be recursive. Thus we have the function

```
(lambda (f)
  ((lambda (x) (x x))
   (lambda (x) (f (x x)))))
```

This function is known as Y, or the *Y-combinator*, and it enables one to perform recursion in the lambda calculus. It obeys the equality

$$(Y\ f) \equiv (f\ (Y\ f))$$

, that is, it is a fixed point.

[9]Note that this crucially only works under lazy evaluation—otherwise the `(mk-fact mk-fact)` would get evaluated once, get its value bound to `fact`, and that'd be the end of it, but we need `fact` to generate a new function each time through. In fact, the lambda calculus *always* requires lazy evaluation; there is a parallel theory known as the lambda-*value* calculus in which eager evaluation is used. It is left as an exercise to the reader how to modify the example so that it would work in an eager regime.

A job well done: Epilogue

We have now reduced our language to the following three-case grammar:

$$\begin{array}{lcl} \Lambda & :== & \langle\text{var}\rangle \\ & | & (\Lambda\ \Lambda) \\ & | & (\lambda(\text{var})\ \Lambda) \end{array}$$

The language Λ has, simply, variables, function application, and abstraction; nothing else. It is known as the *lambda calculus*, and its development in the 1930s made theoreticians deliriously happy, because A) it strongly lends itself to inductive proofs, and B) the proofs need only deal with three cases!

Back in the 1930s, a lot of people were asking the question, "what can we compute?" This was not a trivial question, as it had important bearing on whether one language (or method of computation) was 'more powerful' than another (is C more powerful than Scheme? and so on). Turing tried and succeeded in reducing a huge variety of computation to a few simple operations involving a tape with a read/write head; meanwhile Church, Curry, et al were attacking the same problem from a completely different direction, and reduced a huge variety of computation to the lambda calculus. Happily, it was proven that Turing's tape machine and Church's lambda calculus are isomorphic—an algorithm in one can be reduced to an algorithm for the other—so that theoreticians can prove things under one system or the other, as convenient. Obviously, though, the ones that prove using the lambda calculus are much cooler.