Volume Number: 7
Issue Number: 5
Column Tag: Lisp Listener

Lambda Calculus

By André van Meulebrouck, Chatsworth, CA (vanmeule@roadrunner.com)

“A Calculus for the Algebraic-like Manipulation of Computer Code”, or “Why Oh Why Oh Y?”

“People who like this sort of thing will find that this is the sort of thing they like.” *Abe Lincoln*

[André van Meulebrouck is a consultant who makes his base camp in Southern California. When not cloistered in front of a keyboard, you’ll most likely find him donning a Zippy T-shirt in the great out-of-doors, or pursuing various athletic endeavors. (And listening to Pat Metheny music, no doubt.)]

Note to the Überprogrammer and disclaimer: I will be endeavoring to write so simply that everyone should be able to get something out of this article. Such tutorialness should not be taken as an insult to anyone’s intelligence.

Introduction

Wouldn’t it be nice to be able to manipulate computer code in the same sort of clean mathematical way that one does algebra? Usually computer code is too ad hoc to allow much of that, but what if we had a super clean model that we could work with? Even if such a model wasn’t practical, it would certainly still be interesting, and what could be found out from it might be mappable back into computer languages which don’t have as neat a set of “algebraic-like” properties.

Most people have heard of the Turing machine, as it’s now tantamount to being the measuring stick by which computability is judged. If something is computable on a Turing machine, it is considered computable; if something isn’t computable on a Turing machine it’s considered uncomputable.

There is however, another model, equally as powerful, that is quite different. It has no concept of Turing machine style “state”, and it has some very nice algebraic-like properties. It’s called the λ-calculus.

This article will introduce λ-calculus and combinators, and show some similarity between λ-calculus and the programming language LISP. An attempt will then be made to use λ-calculus as a programming language in its own right to code a popular function in an effort to show the clean, algebraic-like properties of λ-calculus and to consider (in a small way) if λ-calculus might serve as a “light unto the footsteps” to guide the way to computing without gratuitous complexity.

Intro to λ-calculus

λ-calculus is a calculus which expresses “computation” via anonymous functions. λ-calculus preceded LISP and LISP drew from it to some extent. λ is the anonymous function in λ-calculus, and lambda is the anonymous function in LISP.

LISP 101

What follows will be a crash course in LISP. *(This will be just enough to "make you dangerous".)*

Let's start with a some arbitrary function and convert it from how you'd see it in a math book or C program to how it would look in LISP. Let's pick the sine function as it would look if applied to the numerical argument 3: sin(3). Let's tuck sin into the list of arguments and use spaces instead of commas if there is more than one argument: (sin 3). This form of prefix notation is the chosen syntax of LISP. Let's stick to this syntax for everything, lest the syntax get so unwieldy that parsing become a science in and of itself. So, 2 + 2 will be (+ 2 2). Note that with this style of syntax, it is easy to represent + with an arbitrary numbers of arguments: (+ 1 2 3 4 5).

Further, let's say that every top level position of a list gets one round of evaluation. Specifically, the function position gets evaluated, then each argument position gets evaluated, then, the evaluated function gets applied to the evaluated arguments. The "one round of evaluation" can be used to invoke a function call from any position:

```
>>>
(+ (+ 2 3)
   (+ 3 5))
13
```

or even

```
>>>
(+ (+ (+ 1 2)
      (* 3 4))
   (* 5 2))
25
```

because a list in any position can have a function position and argument positions of its own, and so on, to arbitrary depth.

Note: In the Scheme dialect of LISP, the function position is treated precisely the same as the argument positions. Let me motivate an example of that by first showing the if function (which in LISP is a true function). if has the form: (if <condition> <then> <else>). If <condition> is true, <then> is evaluated and returned, otherwise <else> is evaluated and returned. zero? is a function (called a predicate) that is used to determine whether a number is 0 or not. Consider this example:

```
>>> ((if (zero? 0) + -) 3 4)
7
```

The next thing we need is a way to abstract out common process patterns into descriptions. This is done via lambda, the anonymous function. For instance, (lambda (x) (+ x 1)) is a function that takes in an evaluated argument, binds it with x, and then computes the body of the lambda form with the understanding that any occurrence of parameter x in the body will refer to the value of x bound by the lambda form. In this case, the returned result will be the argument plus one, and the argument will not be side effected. To invoke an anonymous function, we simply invoke it like any other function. We invoked sine like this: (sin 3). Invoking (lambda (x) (+ x 1)) on the argument 3 would look like this:

```
>>> ((lambda (x) (+ x 1)) 3)
4
```

To define something we do this: (define foo 3). Thereafter, typing foo at the MacScheme prompt, we find it evaluates to 3.

```
>>> (define foo 3)
foo
>>> foo
3
```

Here's an example of naming an anonymous function:

```
>>>
(define plus-one
  (lambda (x)
     (+ x 1)))
plus-one
>>> (plus-one 3)
4
```

Currying

λ-calculus is very economical. It has only a few syntactic constructs yet it can still express everything that is computable [Peyton Jones, 1987]. Anything it lacks can be bootstrapped.

Since I described LISP previously, I can now use LISP to better describe what λ-calculus is: for us herein, λ-calculus will essentially amount to restricting ourselves to using nothing but LISP's lambda forms.

That's almost a good definition, however, due to the extreme thriftiness of λ-calculus, we must impose an even more draconian restriction: every LISP lambda form must have one and only one argument! (Note that while this improves our description of λ-calculus, it's still only a very nutshell definition.)

How then does λ-calculus express a lambda form that has 2 or more arguments? For instance: (lambda (x y) ...)? This is no problem, because any function of n parameters can be converted to a function of n-1 parameters, and so forth (recursively) until all n parameters are "abstracted out". For instance,

(lambda (x y) ...) would become (lambda (x) (lambda (y) ...)), and (lambda (x y z) ...) would become (lambda (x) (lambda (y) (lambda (z) ...))), etc..

This technique is called "currying" after Haskell Curry, but was originally introduced by Schönfinkel [Révész, 1988].

Free Variables and Higher-Order Functions

Notice that currying relies on two issues that are quite interesting; free variables, and the ability to return a function as a result.

Free variables are variables which are not bound. This is relative to the context you're looking at things from. For instance x is bound in (lambda (x) (lambda (y) x)), but is free in: (lambda (y) x). Consider it a question of ownership: if a lambda form references a parameter it does not own, it is said to be referencing a free variable; if a lambda form references a parameter it does own, it is said to be referencing a bound variable.

Higher-order functions are functions which can return functions as values or accept them as arguments.

Returning a function as a result is something not widely available in conventional languages, and even in those that allow it, it's usually not provided for as powerfully as in LISP and functional languages. When a function gets defined (evaluated) it must do a closure, which is to say that the environment in force at the time (the parent's environment, or lexically apparent scope) gets encapsulated with the function. Implementationally, this could be the saving of a pointer to the parent's environment with the function so that if/when the function gets invoked (applied), it will know what environment to look to in order to resolve any free variables. A function could be passed around quite a bit, and could be held onto for quite some time before it actually gets invoked. And who knows what environment will be in force when it does get invoked? A function can be sure to be invoked in the right environment by taking along a recollection of it's defining environment wherever it goes.

Intro to Combinators

(Prerequisites: A grade of C or better in; Intro to λ-calculus, LISP 101, Currying, and Free Variables and Higher-Order Functions.)

At this point, I want to complicate things just a bit by introducing a narrower subset of λ-calculus: combinators.

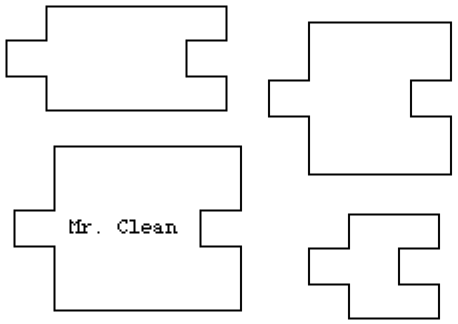
To convert a λ-calculus expression into a combinator, we get rid of the free variables in it. A λ-calculus expression which has no free variables is a combinator. A combinator can have free variables within its subexpressions, but ultimately all free variables in all its subexpressions must be resolved within the context of the combinator, without requiring any external context.

Why the preoccupation with ridding free variables? On a grand scale, it has to do with modularity. The idea of modularity is to break up a problem into black boxes, the combining of which can be done with little care as to the contents (as long as the contract for what the black box does is well understood). Taking this idea to extremes, what would the ultimate in modularity be? How about small black boxes that have no “wires” hanging out of them? I claim that every free variable is precisely one such “wire” that we need to expend mental energy on keeping track of to make sure they get “hooked up” right and that other black boxes that also hook up to them don’t do anything nasty to them that would cause problems for other black boxes.

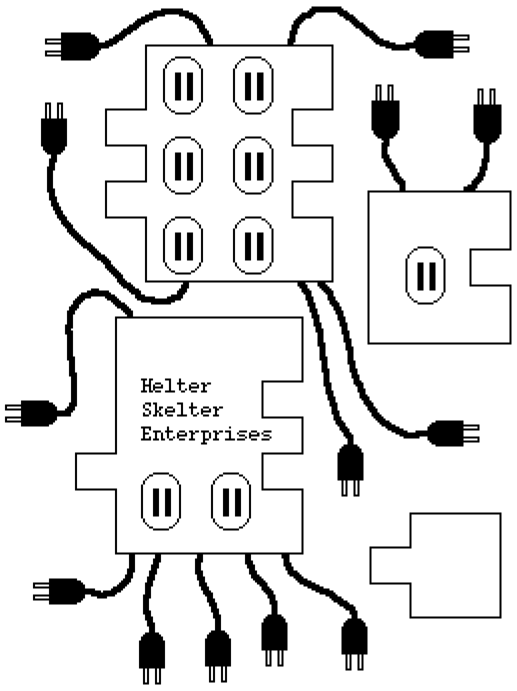
The next concern of modularity is glue. Once you’ve pulverized something into pieces, how do you glue the pieces together? The methods a computer language gives you to break down a problem into blocks, and the quality of glue you’re given to combine those blocks, makes a big difference in how good the language is.

With combinators, the blocks are small and smooth (i.e., no wires hanging out), and the glue is very simple: each block can combine with one (and only one) block at a time.

Perhaps some graphical propaganda with subliminal messages is in order here. Would you rather hook up blocks like these:



or like these?



Programming with Combinators

λ-calculus interpreters are not widely available, and investigating λ-calculus is rather difficult in many of the more conventional languages available today. However, LISP is close enough to allow a fairly immediate translation. And Scheme (an elegantly simple dialect of LISP) is even closer.

Let's take a look at what a popular function would look like in λ-calculus by approximating it as closely as possible in Scheme. Our chosen function will be the factorial function, which is a canonical example in LISP circles.

I think you'll find λ-calculus to be an incredibly simple calculus, yet despite its simplicity, you'll probably find you're outrunning your "headlights of conception" very quickly (and often with incredibly short expressions)! This game might seem similar, in fact, to programming a Turing machine inasmuch as both are a sort of "minimalist" game; and with such severe restrictions on how expression can take place, it requires a "black magic" style of cleverness that many might find intriguing and gratifying.

(Note: Examples cited in the text of the article will be taken from a MacScheme transcript window. What the user types will be represented in boldface, and MacScheme responses will be in italics. The MacScheme interpreter prompt is >>> after which the reader should type in the form of the example, then press the enter key to compile and evaluate it. All the code necessary to run the λ-calculus version of factorial is contained at the end of the article, and the article refers to it heavily. By typing in all the code at the end of the article, and putting it into a file, one can then open that file and select "Eval Window" off the "Command" menu, then quickly select the transcript window (to watch the action). All necessary functions will then get defined, and a test case will be run giving a result of 120.)

Here's what our chosen function looks like in Scheme.

```
MacScheme™ Top Level
>>>
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (- n 1))))))
fact
>>> (fact 5)
120
```

I want to trivially rewrite fact to use a repertoire of primitives closer to what's available in primitive recursion ("primitive recursion" here meaning the topic of that name in theoretical Computer Science [Wood, 1987]). The appeal of primitive recursion in this exercise is that it is involved with a similar style of bootstrapping as is done in λ-calculus: more sophisticated functions are derived using an extremely simple set of primitive functions.

```
>>>
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (pred n))))))
fact
```

pred is the predecessor function defined as follows (bear with me through the trivial--things will get hairy fast enough).

```
>>>
(define pred
  (lambda (n)
    (- n 1)))
pred
>>> (pred 5)
4
>>> (fact 5)
120
```

With no further delays, it's time to derive the combinator version of factorial. Let's start by making some combinators we know we'll need.

Booleans and Conditionals

combinator-true and combinator-false are nothing more than projection functions [Wood, 1987], which consume two arguments. combinator-true "projects" (returns) its first argument, and combinator-false returns its second argument.

```
>>> ((combinator-true 1) 2)
1
>>> ((combinator-false 1) 2)
2
```

With only (anonymous) functions to work with, it's no wonder the constants true and false are functions (i.e. get used to it). But why these particular functions? Was this an arbitrary choice, or is there something that forced our hand here? Stay tuned!

One of the most basic functions we'll need (and the one which combinator-true and combinator-false were designed for) is the conditional: if. It would be most straightforward if we could define our combinator-if as follows.

```
(define combinator-if
  (lambda (condition)
    (lambda (then)
      (lambda (else)
        ((condition then) else)))))
```

What we want is for the evaluation of argument condition to produce a function which must then consume the other arguments (then and else) and return whichever is appropriate. While this version of combinator-if will work in λ-calculus, it won't work in Scheme. The reason: if is inherently normal order in its evaluation (call by name or call by need), but Scheme evaluation is applicative order (call by value). Applicative order means the evaluated function will get applied to the evaluated arguments, whereas in normal order, arguments get consumed unevaluated. I'm perhaps giving very oversimplified "nutshell" style definitions here, but the issues of evaluation are beyond the scope of this article and you needn't worry too much about them, except to understand that you don't want to evaluate both branches of an if (indeed doing so can cause infinite recursion).

(While on the topic of evaluation, it might be worth noting that one of the practical payoffs of λ-calculus is the Church-Rosser theorem [Révész, 1988] which says that if a lambda expression has a normal form (roughly translated: a terminating answer), it can be found regardless of what evaluation order is used. This has implications for multiprocessor/parallel architectures.)

To get around Scheme's applicative order evaluation of combinator-if, we can simply "thunkify" all arguments to it, and then later "force" the result.

To thunkify something means to wrap a lambda of no arguments around it. The net effect is to delay evaluation of the thing which gets wrapped up, because when the lambda of no arguments gets evaluated, it will do a closure. Even though the lambda form has no arguments of its own, a closure will still be done! And that's important, because there might be free variables inside the body (i.e. the thing which got wrapped up might have free variables in it). The resultant wrapped up object is called a "thunk".

(It might be worth noting that in λ-calculus the contents of a lambda form can get evaluated, to some extent, prior to the lambda form being invoked! This is called partial evaluation, and has some advantages (It's essentially like being able to run code, to some extent, before you have any data/arguments to run it with!). LISP however, is not allowed to evaluate the contents of a lambda form until the lambda form gets applied.)

Since the evaluation of the wrapper used up the round of evaluation that the contents would normally have gotten, another round of evaluation is required in order force the contents inside the wrapper to get evaluated. This is easily done by forcing the thunk, which is to say applying the thunk to no arguments!

Here are two test cases to verify combinator-if.

```
>>>
(((combinator-if combinator-false)
  (lambda () combinator-true))
 (lambda () combinator-false))
#<PROCEDURE combinator-false>
>>>
(((combinator-if combinator-true)
  (lambda () combinator-true))
 (lambda () combinator-false))
#<PROCEDURE combinator-true>
```

(Note: I've violated one of our draconian policies here. Each lambda form was supposed to have one argument only (no less, no more). A thunk violates this. However, thunks could be made to toe the line. As an exercise to the reader, explain how to make thunks of one argument, and how to force them.)

Church Numerals

λ-calculus is so thrifty it doesn't have numbers. Unless of course, we extend pure λ-calculus. However, even if we did, we'd still want to see if we could implement them with combinators for the sheer fun of it; more importantly, since numbers aren't part of pure λ-calculus, we don't know for sure if they will have the same properties that pure λ-calculus has unless we implement them in pure λ-calculus (if we can, then we'll know that we can safely extend pure λ-calculus to include non-negative integers) [Révész, 1988].

There are different ways of bootstrapping numbers. Herein we will implement them by function applications. A number will be a function that consumes two arguments (which can be thought of as being a function and an object) and will then apply the function to the object n times (to represent n). Numbers derived in this manner are called Church numerals owing to the fact that they were invented by the inventor of λ-calculus: Alonzo Church.

combinator-zero takes in a function and object and just returns the object after not having applied the function to it (i.e. after having applied the function to it zero times). Note that combinator-zero is the same as combinator-false.

combinator-zero? is the predicate used to determine whether an argument is combinator-zero or not.

Before explaining combinator-zero?, I want to define some terms to make the ensuing discussion less cumbersome. Let’s call a projection function that returns it’s last argument an n-consumer where n is a positive integer representing how many arguments the function wants to consume. So, identity is a 1-consumer, combinator-false and combinator-zero are 2-consumers, and project-3rd-of-3 is a 3-consumer.

An observation: an n-consumer applied to anything will yield an n-1-consumer if n is greater than 1. For example, a 3-consumer applied to anything will return a 2-consumer.

combinator-zero? applies its argument (a Church numeral) to a 3-consumer, and then applies the result of that to combinator-true.

In the case where combinator-zero?’s argument is combinator-zero we have the case wherein a 2-consumer gets applied to something. The result will be identity. Since identity simply bounces back whatever argument it gets handed, when it gets applied to combinator-true it will return combinator-true.

In the case where combinator-zero?’s argument is other than combinator-zero, we wind up with m nestings of 3-consumers (where m is the value of the Church numeral that combinator-zero? was called with), and all those nested 3-consumers get applied to combinator-true. The result will be a 2-consumer (which is to say combinator-false) because what is returned from the innermost application becomes the next anything for the next outermost application (and so on)!.

Notice that the second test (below) did not return #<PROCEDURE combinator-false>. However, in the third test, (which called the same result with arguments to see which one it would return) we see that the result behaves like combinator-false. Explanation: This anomaly is due to LISP’s inability to determine function equality. Functions are considered equal in LISP if and only if the pointers to them are the same. In this case, we have two functions that are semantically the same, but are syntactically different.

```
>>> (combinator-zero? combinator-zero)
#<PROCEDURE combinator-true>
>>> (combinator-zero? combinator-one)
#<PROCEDURE>
>>> (((combinator-zero? combinator-one)
       ‘i-am-true)
     ‘i-am-false)
i-am-false
```

dechurchify-numeral can be applied to Church numerals to convert them to “regular” numbers. To convert a Church numeral to a regular number you “unravel” it. A Church numeral is nothing more than a conglomerate of functions waiting to be invoked on some object. So, by calling a Church numeral with 1+ as the function and 0 as the object, we can essentially get a Church numeral to sum itself up!

make-church-numeral is the companion to dechurchify-numeral: it converts “regular” numbers to Church numerals. (make-church-numeral is not itself a combinator, but it doesn’t need to be--it’s merely a user convenience function so we can see the results of Church numeral computations in human readable form.)

```
>>> (dechurchify-numeral
     (make-church-numeral 50))
50
```

Arithmetic Functions

combinator-succ is the successor function (addition by 1 in the set of nonnegative integers). combinator-succ works by consuming a Church numeral (let’s call it n), then returning another numeral. The numeral it returns will, like all Church numerals, want to consume a function and an object--let’s call the function f. Inside the numeral it returns will be f applied to the unraveling of n. In other words, it will unravel n and then apply f to it once more.

All counting numbers can be now be bootstrapped via combinator-zero and combinator-succ. For instance, if we apply combinator-succ to combinator-zero, that should give us combinator-one, to which we could then apply combinator-succ again to get combinator-two, etc.. (This is precisely how make-church-numeral makes Church numerals.)

combinator-* does nothing more than compose it’s arguments. To compose functions you apply the functions, one after another, to an argument. That is what Church numerals themselves are: composed functions! combinator-* wants to consume two Church numerals (let’s call them m and n). After consuming them, it returns a numeral. Let’s call the function the returned numeral will want to consume: f. Inside the returned numeral, f will be handed to n, which will result in a new function being made. The new function will be a function that will do f n times. It will then hand this new function to m, meaning the new function will be done m times. So, the function that does f n times will be run m times! (Essentially, this amounts to composing composed functions!)

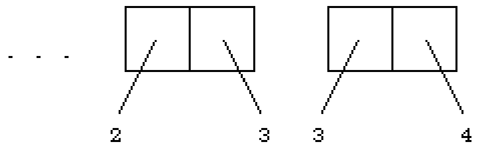
```
>>>
(dechurchify-numeral
 ((combinator-* (make-church-numeral 45))
  (make-church-numeral 3)))
135
```

combinator-pred is subtraction by 1 in the set of nonnegative integers. This one is tricky--Church himself had trouble with it, and it was ultimately Kleene who came up with it [Révész, 1988]. However, what they were doing was tantamount to programming in LISP before LISP had been invented! In some ways, combinators are like a form of Assembly language, and therefore if you try to understand them at too low a level without any high level concepts to guide you, it’s easy to get so lost that you can’t see the forest for the trees. Therefore I want to explain this function in terms of LISP concepts: car, cdr, and cons.

First, we need to come up with an algorithm that doesn’t violate the rules of the game. Basically, we know how to count up: combinator-succ. But how do we translate that into counting down? One thing we could do is to start from 0 and count up to the number we want the predecessor of, then return the next-to-the-last number we generated while counting up.

Note that n - 1 and n can be thought of as a tuple: (n - 1, n), which we can think of as being (m, m + 1) wherein m + 1 = n.

We have tuples in LISP: they are called cons cells (in Scheme, they are called pairs), and they look like this: (3 . 4), which is a dotted pair (discussing the difference between dotted pairs versus “regular” pairs is beyond my game plan). Using cons (the LISP constructor function, which is used to build tuples and lists), we can make the aforementioned pair like this: (cons 3 4). And, we can access the different parts of it using car and cdr: (car (cons 3 4)) => 3, and (cdr (cons 3 4)) => 4.



So, our job is clear: make tuples until we reach n, then return the car of the tuple that contains: n-1 and n.

Building one tuple is a start, but how do we build n of them? In LISP we could recurse from 0 to n, checking to see when we’ve reached n (building, of course, a tuple during each recursive call). While recursion doesn’t come for free in λ-calculus (as you will see) we can get it. That leaves us with having to come up with a predicate to test for Church numeral equality. Even if we could coin the needed predicate, it still somehow seems like using a bulldozer to dig a hole for a petunia, doesn’t it? Could we could skip recursion and the equality predicate? Perhaps at first thought that might seem like a tall order, but it turns out we can appeal to the definition of Church numerals to get the n iterations!

This is the same trick used by dechurchify-numeral. Recalling that a Church numeral is really nothing more than a conglomerate of functions waiting to be applied to an object, and since we can pass in the function to be applied and the object we want it to be applied to, we can use this to our advantage just as dechurchify-numeral does: instead of passing in 1+, what’s to stop us from passing in some other function, like say a tuple maker? And instead of passing in 0, we could pass in an initial tuple of (0, 0) as the object. The final result will be a tuple, which we can then take the car of.

(Note that if the initial tuple is (“error”, 0) we can trap out attempts to call the predecessor function on 0.)

```
>>> (dechurchify-numeral
     (combinator-pred
       (make-church-numeral 3)))
2
```

Note that combinator-car and combinator-cdr take in an object that wants a selector as an argument. For the selector argument, combinator-car will pass to the object a projection function that will return its 1st of 2 arguments, and combinator-cdr will pass it a projection function that will return its 2nd of 2 arguments.

Recursion

At this point we now have the following, and a test case reveals it works:

```
>>>
(define fact
  (lambda (n)
    (((combinator-if (combinator-zero? n))
      (lambda ()
        combinator-one))
     (lambda ()
       ((combinator-* n)
        (fact (combinator-pred n)))))))
fact
>>> (dechurchify-numeral
     (fact (make-church-numeral 5)))
120
```

However, we’re still not there yet because we rely on recursion through the use of a named symbol, and that’s: fact. It’s okay to use named symbols as a shorthand (in the sense of a macro expansion) for code, but not if we rely on them for computational expressiveness. To illustrate the problem, consider the following.

(Note: let is a LISP special form that creates local bindings. Below, fact-wrong will be bound to the lambda form (lambda (n) (if ...)) within the scope of the let, which ends with the right paren that closes the let.)

```
>>>
(define local-fact-wrong
  (lambda (n)
    (let ((fact-wrong
           (lambda (n)
             (if (zero? n)
                 1
                 (* n (fact-wrong (- n 1)))))))
      (fact-wrong n))))
local-fact-wrong
>>> (local-fact-wrong 5)

ERROR:  Undefined global variable
fact-wrong

Entering debugger.  Enter ? for help.
debug:>
```

Obviously the semantics for define are quite different from the semantics for let. What happened above actually makes perfect sense. It’s called lexical scoping: The reference to the free variable fact-wrong inside the lambda form that was being bound to fact-wrong must refer to a variable named fact-wrong in a parent environment. And indeed, at the time the lambda form was closed and bound to fact-wrong, there was no fact-wrong in any parent environment.

In less formal terms: We were defining something, then in the midst of defining it, we made a reference to the very thing we were in the midst of defining! It makes sense that you have to completely finish defining something before you can make references to it!

We can fix this problem by first creating the variable that we want to use recursively, then evaluating the lambda form (to achieve closure) so that the free reference will refer to the variable we just created. Then, we bash the binding of the variable we created to get it to point to the closed lambda form [Gabriel, 1988], [Rees et al., 1986]. This requires assignment, which is done by the destructive operator set! (pronounced: “set bang”).

(Note: begin simply allows sequencing. It signifies that the forms inside it’s scope should be evaluated one after another. It is generally used in places where normally only one form is allowed, such as a branch of an if test. Below, the use of begin is superfluous, but serves to highlight the fact that sequencing is taking place.)

```
>>>
(define local-fact-right
  (lambda (n)
    (let ((fact-right ()))
      (begin
       (set! fact-right
             (lambda (n)
               (if (zero? n)
                   1
                   (* n
                      (fact-right (- n 1))))))
       (fact-right n)))))
local-fact-right
>>> (local-fact-right 5)
120
```

As you can see, that makes us win. There is a facility for defining local functions in Scheme already, that does something similar to the above: letrec.

```
>>>
(define local-fact
  (lambda (n)
    (letrec
      ((fact-right
        (lambda (n)
          (if (zero? n)
              1
              (* n
                 (fact-right (- n 1)))))))
       (fact-right n)))))
local-fact
>>> (local-fact 5)
120
```

Obviously define has letrec semantics rather than let semantics. Another way of looking at this is that lexical scoping makes sense until we look at the global level, or top level (the level you’re at when you type forms at the MacScheme prompt). Funny things happen at the global level. Since it’s the last environment and is the parent of everything, we can refer to variables that don’t exist yet. If you think of scoping as a search, then you’re searching the lexical environments from the innermost to the outermost to find any particular variable. If none is found, then you search the global environment. So, even if define had let semantics, unless the style of lexical search employed was semantically different from the style of search employed by the metacircular interpreter in [Abelson et al., 1985], recursive definitions would still work at the top level!

The fact that we can refer to global variables that don’t exist yet does make for one very important convenience: We can define functions in any order we want. For instance, if foo calls fido, we can define foo first and fido later, or vice versa. It just simply doesn’t matter.

(Exercise for the reader: What happens if two symbols; fluff and fido, are “bound” by a letrec? Does fluff know about fido? Does fido know about fluff? How does Scheme define the semantics of letrec? See [Rees et al., 1986]. By the way, fluff and fido are two really “with it” trendoids; fido is an AI dog, and fluff is an object oriented cat. Holy buzzwords, Batman! (I just couldn’t resist the chance to be a joker.))

I’ve diverted a bit in order to drive home the point that recursion relies on a sort of tricky way of resolving free variables, and I think you can now imagine why such a trick won’t work in λ-calculus.

The Y Combinator

So, how do we get recursion in λ-calculus? Well, one thing we do know how to do is: We know how to get rid of free variables. Think about it: Why is x free in (lambda (y) x)? Because it doesn’t look like this: (lambda (x) (lambda (y) x)). Well, if we want it to look like that, let’s just make it be that! After all, we’re not changing the value of the expression, we’re only changing the way the free variable will derive its meaning: We’re promising to pass in the value rather than rely on the rules of lexical scoping to ascribe the right value from the static (lexically apparent) context. This process of factoring out free variables is called abstraction. A form is not a combinator until we resolve all the free variable references in it to make them all bound within the context of the proposed combinator. So, let’s convert fact into a combinator.

```
(lambda (fact)
  (lambda (n)
    (((combinator-if (combinator-zero? n))
      (lambda () combinator-one))
     (lambda () ((combinator-* n)
                 (fact (combinator-pred n)))))))
```

Now we’ve got a combinator, but we’re not off scot-free yet. We need to know what should be bound to fact. In other words, what do we apply (lambda (fact) ...) to so that fact gets bound to the right thing?

What do we want do we want it to be bound to? Well, fact should be bound to the code of the fact function. (Of course! It’s recursive!) In other words, we want to apply (lambda (fact) ...) to a copy of fact’s code. But we’d have to do something like that for every level the recursive version of fact would have recursed.

Essentially, what we want is a way to get infinite recursion, and then we want to knit into each recursive level a combinator version of the function we’re trying to simulate recursion on. (The infinite recursion will get stopped at the appropriate time since the function definition includes a stop condition.) To get infinite recursion is trivial.

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

Then all we need to do is “knit in” a version of the fact combinator at each level of recursion.

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

And again, we need to factor out the free variable, or said another way, abstract it out.

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

This special combinator has a name: It’s called the Y combinator. (Kids, do try this at home!)

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

This version of Y works in λ-calculus where evaluation is normal order, but will hang (when invoked) if evaluation is applicative order.

For our purposes, we need the applicative order Y combinator, which is the following fix to the (normal order) Y combinator. (Exercise for the Überprogrammer: Derive the below tweak to Y and convince yourself that you understand why it’s necessary for applicative order.)

```
>>>
(define applicative-order-y
  (lambda (f)
    ((lambda (x)
       (f (lambda (arg) ((x x) arg))))
     (lambda (x)
       (f (lambda (arg) ((x x) arg)))))))
applicative-order-y
```

Putting it all together, we get the chosen function we wanted.

```
>>>
(define combinator-fact
  (applicative-order-y
   (lambda (fact)
     (lambda (n)
       (((combinator-if (combinator-zero? n))
         (lambda () combinator-one))
        (lambda ()
          ((combinator-* n)
           (fact (combinator-pred n)))))))))
combinator-fact
```

And it works.

```
>>>
(dechurchify-numeral
 (combinator-fact (make-church-numeral 5)))
120
```

Aftermath

Now that we’ve gotten this far, let’s compare our model with LISP (LISP here meaning Scheme and/or Common LISP). For instance, does our if accept the same things that LISP’s if accepts? No. Ours is much stricter. It accepts only canonical true or canonical false, while the empty list is taken for false in Common LISP [Steele, 1990]. Scheme is closer to our model (see [Rees et al., 1986]) but does itself have some deviations.

Exercise for the reader: Decide for yourself if our if is correct (and thereby LISP should be made to tow the line, or at least, we should program in LISP as if it were as strict as our model), or, that LISP’s permissiveness is a justifiable programmer convenience which causes no problems. In determining this, you might want to tweak our model to mimic the behavior of LISP (different tweaks would be needed for Scheme versus Common LISP). Then, run various tests and see if the tweaks cause problems, confusion, slow code execution, etc.. (Perhaps there’s no right or wrong answer here.)

Note: the empty list and the predicate to check for it are () and null? respectively (in Scheme) and nil and null in Common LISP. I haven’t shown their implementation in terms of combinators, so the above exercise would require some digging, especially since most books gloss over their implementation.

Exercise for the Überprogrammer: One might think, on first seeing curried functions, that currying is an expressive weakness inflicted on λ-calculus due to its thriftiness. While fully uncurrying all the code needed for combinator-fact, think about whether there are or might be cases wherein curried functions are more expressively powerful than their uncurried counterparts. Also, see if all curried functions can be uncurried in a purely mechanical fashion, or if some require a bit of tweaking!

Yet another exercise: In a sense, Church numerals are a rather odd way of thinking about numbers, because Church numerals are “verb” oriented (based on functions calls waiting to be set into motion), whereas perhaps normally we tend to think of numbers as being “noun” oriented (based on static data-like objects). We could devise numerals based on the latter approach. For instance, the empty list could be 0, 1 could be a list containing 1 element, and n could be a list containing n elements, etc.. Implement such numerals and note how it changes the arithmetic routines. For instance; succ will turn out to be cons, pred will be cdr, plus will be append, and delistify-numeral will be length, etc.. (Note how much simpler pred is!) Compare and contrast the Church numeral approach with the list numeral approach. For instance, will recursion be necessary to express multiplication? Are Church numerals more expressively powerful?

A question for the Überprogrammer: I’ve shown two ways of looking at recursion. One involves state (the bashing of a bound variable) and the other: the Y combinator. What I didn’t mention was that (lambda (x) (x x)) applied to itself, is an example of self replicating code. While some folks don’t like state, others don’t like the idea of self replicating code. If we decide we don’t like either (for whatever reasons), what does that do to recursion? With what other model can we understand or justify recursion?

(Warning: λ-calculus can be hazardous to your financial health.)

Acknowledgments

“Thanks” to technical reviewer Henry G. Baker, Ph.D. for helpful suggestions.

“Thanks” to John Koerber for his efforts at making this article safe for nonLISPers by not letting me get away with things a LISP reviewer would have let me get away with!

Bugs/infelicities mea culpa.

Bibliography and References

[Abelson et al., 1985] Harold Abelson and Gerald Jay Sussman with Julie Sussman. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, Massachusetts, USA, 1985.

[Gabriel, 1988] Richard P. Gabriel. The Why of Y. LISP Pointers, vol. 2, no. 2 October-November-December, 1988.

[Peyton Jones, 1987] Simon L. Peyton Jones. The Implementation of Functional Programming Languages. Prentice-Hall International, 1987.

[Rees et al., 1986] Jonathan Rees and William Clinger (editors). Revised3 Report on the Algorithmic Language Scheme; AI Memo 848a. MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, USA, September 1986.

[Révész, 1988] György E. Révész. Lambda-Calculus, Combinators, and Functional Programming. Cambridge University Press, Cambridge, England, 1988.

[Steele, 1990] Guy L. Steele Jr.. Common LISP the language 2nd edition. Digital Press, USA, 1990.

[Wood, 1987] Derick Wood. Theory of Computation. Harper and Row, Publishers, Inc., 1987.

. . .

A note on references: I’m being very informal in the citing of references, thinking it more appropriate for a non-academic article, and because the citing of multitudinous references could become very burdensome.

All the λ-calculus versions of the combinators presented herein (from which the Scheme versions were derived) are from [Révész, 1988], except in the case of the applicative order Y combinator which abounds in LISP literature.

[Peyton Jones, 1987] presents introductory λ-calculus, as well as denotational semantics and other topics.

[Abelson et al., 1985] includes Church numerals, the applicative order Y combinator, and similar, and implements them in Scheme, however these are more of a sideline issue alluded to in exercises.

[Rees et al., 1986] is the definition of the Scheme language. Copies of the Revisedn Report can be obtained from the

MIT AI Lab for a small fee.

[Gabriel, 1988] presents a tutorial on Y in Common LISP.

MacScheme™ is put out by Lightship Software, P.O. Box 1636, Beaverton, OR 97075 USA. Phone: (503) 643-6909.

There are numerous other sources of information on this topic that I could list, however I feel these references most closely represent the materials I actually drew from in this article.

```
; File:  fact.y.

; Here's the chosen function.

(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (- n 1))))))
;
(fact 5)
;
(define identity
  (lambda (x) x))
;
(define project-1st-of-2
  (lambda (x)
    (lambda (y)
      x)))
;
(define project-2nd-of-2
  (lambda (x)
    identity))
;
(define project-3rd-of-3
  (lambda (x)
    (lambda (y)
      identity)))
;
;
; true and false could be defined this way:
;
(define combinator-true
  project-1st-of-2)
;
(define combinator-false
  project-2nd-of-2)
;
; but we'll do them as follows.
;
(define combinator-true
 (lambda (x)
 (lambda (y)
 x)))

;
(define combinator-false
 (lambda (x)
 (lambda (y)
 y)))

;
(define combinator-cons
  (lambda (x)
    (lambda (y)
      (lambda (selector)
        ((selector x) y)))))
;
(define combinator-car
  (lambda (object)
    (object project-1st-of-2)))
;
(define combinator-cdr
  (lambda (object)
    (object project-2nd-of-2)))
;
(define force-a-thunk
  (lambda (thunk)
    (thunk)))
;
(define combinator-if
  (lambda (condition)
    (lambda (then)
      (lambda (else)
        (force-a-thunk ((condition then) else))))))
;
(define combinator-zero
  project-2nd-of-2)
;
(define combinator-zero?
  (lambda (n)
    ((n project-3rd-of-3) combinator-true)))
;
(define combinator-succ
  (lambda (n)
    (lambda (f)
      (lambda (x)
        (f ((n f) x))))))
;
(define dechurchify-numeral
  (lambda (numeral)
    ((numeral 1+) 0)))
;
(define make-church-numeral
  (lambda (n)
    (if (zero? n)
        combinator-zero
        (combinator-succ
         (make-church-numeral (- n 1))))))
;
(define combinator-*
  (lambda (m)
    (lambda (n)
      (lambda (f)
        (m (n f))))))
;
(define combinator-pred
  (lambda (n)
    (combinator-car
     ((n (lambda (tuple)
           ((combinator-cons
             (combinator-cdr tuple))
            (combinator-succ (combinator-cdr tuple)))))
      ((combinator-cons "combinator-pred called on 0")
       combinator-zero)))))
;
(define combinator-applicative-order-y
  (lambda (f)
    ((lambda (x) (f (lambda (arg) ((x x) arg))))
     (lambda (x) (f (lambda (arg) ((x x) arg)))))))
;
(define combinator-one
  (make-church-numeral 1))
;
(define combinator-fact
  (combinator-applicative-order-y
   (lambda (fact)
     (lambda (n)
       (((combinator-if (combinator-zero? n))
         (lambda () combinator-one))
        (lambda () ((combinator-* n)
                    (fact (combinator-pred n)))))))))
;
(dechurchify-numeral
 (combinator-fact (make-church-numeral 5)))
;
; Done.
```