PL: Lecture #13

Tuesday, October 22nd

Lambda Calculus — Schlac

> 👉 ***PLAI §22*** *(we do much more)*

We know that many constructs that are usually thought of as primitives are not really needed — we can implement them ourselves given enough tools. The question is how far can we go?

The answer: as far as we want. For example:

```
(define foo((lambda(f)((lambda(x)(x x))(lambda(x)(f(x x)))))(lambda(
f)(lambda(x)(((x(lambda(x)(lambda(x y)y))(lambda(x y)x))(x(lambda(x)
(lambda(x y)y))(lambda(x y)x)))(((x(lambda (p)(lambda(s)(s(p(lambda(x
y)y))(lambda(f x)(f((p(lambda(x y)y))f x))))))(lambda(s) (s(lambda(f
x)x)(lambda(f x)x))))(lambda(x y)x))(lambda(x)(lambda(x y)y))(lambda
(x y)x)))(lambda(f x)(f x))(((f((x(lambda(p)(lambda(s)(s(p(lambda(x y
)y))(lambda(f x)(f((p(lambda(x y)y))f x))))))(lambda(y s)(s(lambda(f
x)x)(lambda(f x)x))))(lambda(x y)x)))(lambda(n)(lambda(f x)(f(n f x)
)))(f((((x(lambda(p)(lambda(s)(s(p (lambda(x y)y))(lambda(f x)(f((p(
lambda(x y)y))f x))))))(lambda(s)(s(lambda(f x) x)(lambda(f x)x))))(
lambda(x y)x))(lambda(p)(lambda(s)(s(p(lambda(x y)y))(lambda(f x)(f(
(p(lambda(x y)y))f x))))))(lambda(s)(s(lambda(f x)x)(lambda(f x)x)))
)(lambda(x y)x)))))))))
```

We begin with a very minimal language, which is based on the Lambda Calculus. In this language we get a very minimal set of constructs and values.

In DrRacket, this we will use the Schlac language level (stands for "~~Scheme~~Racket as Lambda Calculus"). This language has a Racket-like syntax, but don't be confused — it is *very* different from Racket. The only constructs that are available in this language are: lambda expressions of at least one argument, function application (again, at least one argument), and simple definition forms which are similar to the ones in the "Broken define" language — definitions are used as shorthand, and cannot be used for recursive function definition. They're also only allowed at the toplevel — no local helpers, and a definition is not an expression that can appear anywhere. The BNF is therefore:

```
<SCHLAC>       ::= <SCHLAC-TOP> ...

<SCHLAC-TOP>   ::= <SCHLAC-EXPR>
                 | (define <id> <SCHLAC-EXPR>)

<SCHLAC-EXPR> ::= <id>
                 | (lambda (<id> <id> ...) <SCHLAC-EXPR>)
                 | (<SCHLAC-EXPR> <SCHLAC-EXPR> <SCHLAC-EXPR> ...)
```

Since this language has no primitive values (other than functions), Racket numbers and booleans are also considered identifiers, and have no built-in value that come with the language. In addition, all functions and function calls are curried, so

```
(lambda (x y z) (z y x))
```

is actually shorthand for

```
(lambda (x) (lambda (y) (lambda (z) ((z y) x))))
```

The rules for evaluation are simple, there is one very important rule for evaluation which is called “beta reduction”:

```
((lambda (x) E1) E2) --> E1[E2/x]
```

where substitution in this context requires being careful so you won’t capture names. This requires you to be able to do another kind of transformation which is called “alpha conversion”, which basically says that you can rename identifiers as long as you keep the same binding structure (eg, a valid renaming does not change the de-Bruijn form of the expression). There is one more rule that can be used, *eta conversion* which says that `(lambda (x) (f x))` is the same as `f` (we used this rule above when deriving the Y combinator).

One last difference between Schlac and Racket is that Schlac is a *lazy* language. This will be important since we do not have any built-in special forms like `if`.

Here is a Schlac definition for the identity function:

```
(define identity (lambda (x) x))
```

and there is not much that we can do with this now:

```
> identity
#<procedure:identity>
> (identity identity)
#<procedure:identity>
> (identity identity identity)
#<procedure:identity>
```

(In the last expression, note that `(id id id)` is shorthand for `((id id) id)`, and since `(id id)` is the identity, applying that on `id` returns it again.)

> *Something to think about: are we losing anything because we have no no-argument functions?*

Church Numerals

So far, it seems like it is impossible to do anything useful in this language, since all we have are functions and applications. We know how to write the identity function, but what about other values? For example, can you write code that evaluates to zero?

> *What’s zero? I only know how to write functions!*
>
> *(Turing Machine / Assembly programmer: “What’s a function? — I only know how to write 0s and 1s!”)*

The first thing we therefore need is to be able to *encode* numbers as functions. For zero, we will use a function of two arguments that simply returns its second value:

```
(define 0 (lambda (f) (lambda (x) x)))
```

or, more concisely

```
(define 0 (lambda (f x) x))
```

This is the first step in an encoding that is known as *Church Numerals*: an encoding of natural numbers as functions. The number zero is encoded as a function that takes in a function and a second value, and applies the function zero times on the argument (which is really what the above definition is doing). Following this view, the number one is going to be a function of two arguments, that applies the first on the second one time:

```
(define 1 (lambda (f x) (f x)))
```

and note that `1` is just like the identity function (as long as you give it a function as its first input, but this is always the case in Schlac). The next number on the list is two — which applies the first argument on the second one twice:

```
(define 2 (lambda (f x) (f (f x))))
```

We can go on doing this, but what we really want is a way to perform arbitrary arithmetic. The first requirement for that is an `add1` function that increments its input (an encoded natural number) by one. To do this, we write a function that expects an encoded number:

```
(define add1 (lambda (n) ...))
```

and this function is expected to return an encoded number, which is always a function of `f` and `x`:

```
(define add1 (lambda (n) (lambda (f x) ...)))
```

Now, in the body, we need to apply `f` on `x` n+1 times — but remember that `n` is a function that will do `n` applications of its first argument on its second:

```
(define add1 (lambda (n) (lambda (f x) ... (n f x) ...)))
```

and all we have left to do now is to apply `f` one more time, yielding this definition for `add1`:

```
(define add1 (lambda (n) (lambda (f x) (f (n f x)))))
```

Using this, we can define a few useful numbers:

```
(define 1 (add1 0))
(define 2 (add1 1))
(define 3 (add1 2))
(define 4 (add1 3))
(define 5 (add1 4))
```

This is all nice theoretically, but how can we make sure that it is correct? Well, Schlac has a few additional built-in functions that translate Church numerals into Racket numbers. To try our definitions we use the `->nat` (read: to natural number):

```
(->nat 0)
(->nat 5)
(->nat (add1 (add1 5)))
```

You can now verify that the identity function is really the same as the number 1:

```
(->nat identity)
```

We can even write a test case, since Schlac contains the `test` special form, but we have to be careful in that — first of all, we cannot test whether functions are equal (why?) so we must use `->nat`, but

```
(test (->nat (add1 (add1 5))) => 7)
```

will not work since `7` is undefined. To overcome this, Schlac has a `back-door` for primitive Racket values — just use a quote:

```
(test (->nat (add1 (add1 5))) => '7)
```

We can now define natural number addition — one simple idea is to get two encoded numbers `m` and `n`, then start with `x`, apply `f` on it `n` times by using it as a function, then apply `f` `m` more times on the result in the same way:

```
(define + (lambda (m n) (lambda (f x) (m f (n f x)))))
```

or equivalently:

```
(define + (lambda (m n f x) (m f (n f x))))
```

Another idea is to use `add1` and increment `n` by `m` using `add1`:

```
(define + (lambda (m n) (m add1 n)))
(->nat (+ 4 5))
```

We can also define multiplication of `m` and `n` quite easily — begin with addition — `(lambda (x) (+ n x))` is a function that expects an `x` and returns `(+ x n)` — it's an increment-by-n function. But since all functions and applications are curried, this is actually the same as `(lambda (x) ((+ n) x))` which is the same as `(+ n)`. Now, what we want to do is repeat this operation `m` times over zero, which will add `n` to zero `m` times, resulting in `m * n`. The definition is therefore:

```
(define * (lambda (m n) (m (+ n) 0)))
(->nat (* 4 5))
(->nat (+ 4 (* (+ 2 5) 5)))
```

An alternative approach is to consider

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

for some encoded number `n` and a function `f` — this function is like `f^n` (f composed n times with itself). But remember that this is shorthand for

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

and we know that `(lambda (x) (foo x))` is just like `foo` (if it is a function), so this is equivalent to just

```
(n f)
```

So `(n f)` is `f^n`, and in the same way `(m g)` is `g^m` — if we use `(n f)` for `g`, we get `(m (n f))` which is n self-compositions of `f`, self-composed m times. In other words, `(m (n f))` is a function that is like `m*n` applications of `f`, so we can define multiplication as:

```
(define * (lambda (m n) (lambda (f) (m (n f)))))
```

which is the same as

```
(define * (lambda (m n f) (m (n f))))
```

The same principle can be used to define exponentiation (but now we have to be careful with the order since exponentiation is not commutative):

```
(define ^ (lambda (m n) (n (* m) 1)))
(->nat (^ 3 4))
```

And there is a similar alternative here too —

- a Church numeral `m` is the m-self-composition function,
- and `(1 m)` is just like `m^1` which is the same as `m` (`1=identity`)
- and `(2 m)` is just like `m^2` — it takes a function `f`, self composes it `m` times, and self composes the result `m` times — for a total of `f^(m*m)`
- and `(3 m)` is similarly `f^(m*m*m)`
- so `(n m)` is `f^(m^n)` (note that the first `^` is self-compositions, and the second one is a mathematical exponent)
- so `(n m)` is a function that returns `m^n` self-compositions of an input function, Which means that `(n m)` is the Church numeral for `m^n`, so we get:

  (define ^ (lambda (m n) (n m)))

which basically says that any number encoding `n` is also the `?^n` operation.

All of this is was not too complicated — but all so far all we did is write functions that increment their inputs in various ways. What about `sub1`? For that, we need to do some more work — we will need to encode booleans.

More Encodings

Our choice of encoding numbers makes sense — the idea is that the main feature of a natural number is repeating something a number of times. For booleans, the main property we're looking for is choosing between two values. So we can encode true and false by functions of two arguments that return either the first or the second argument:

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

Note that this encoding of `#f` is really the same as the encoding of `0`, so we have to know what type to expect an use the proper operations (this is similar to C, where everything is just integers). Now that we have these two, we can define `if`:

```
(define if (lambda (c t e) (c t e)))
```

it expects a boolean which is a function of two arguments, and passes it the two expressions. The `#t` boolean will simply return the first, and the `#f` boolean will return the second. Strictly speaking, we don't really need this definition, since instead of writing `(if c t e)`, we can simply write `(c t e)`. In any case, we need the language to be lazy for this to work. To demonstrate this, we'll intentionally use the quote back-door to use a non-functional value, using this will normally result in an error:

```
(+ '1 '2)
```

But testing our `if` definition, things work just fine:

```
(if #t (+ 4 5) (+ 1 2))
```

and we see that DrRacket leaves the second addition expression in red, which indicates that it was not executed. We can also make sure that even when it is defined as a function, it is still working fine because the language is lazy:

```
(if #f ((lambda (x) (x x)) (lambda (x) (x x))) 3)
```

What about `and` and `or`? Simple, `or` takes two arguments, and returns either true or false if one of the inputs is true:

```
(define or (lambda (a b) (if a #t (if b #t #f))))
```

but `(if b #t #f)` is really the same as just `b` because it must be a boolean (we cannot use more than one "truty" or "falsy" values):

```
(define or (lambda (a b) (if a #t b)))
```

also, if `a` is true, we want to return `#t`, but that is exactly the value of `a`, so:

```
(define or (lambda (a b) (if a a b)))
```

and finally, we can get rid of the `if` (which is actually breaking the `if` abstraction, if we encode booleans in some other way):

```
(define or (lambda (a b) (a a b)))
```

Similarly, you can convince yourself that the definition of `and` is:

```
(define and (lambda (a b) (a b a)))
```

Schlac has to-Racket conversion functions for booleans too:

```
(->bool (or #f #f))
(->bool (or #f #t))
(->bool (or #t #f))
(->bool (or #t #t))
```

and

```
(->bool (and #f #f))
(->bool (and #f #t))
(->bool (and #t #f))
(->bool (and #t #t))
```

A `not` function is quite simple — one alternative is to choose from true and false in the usual way:

```
(define not (lambda (a) (a #f #t)))
```

and another is to return a function that switches the inputs to an input boolean:

```
(define not (lambda (a) (lambda (x y) (a y x))))
```

which is the same as

```
(define not (lambda (a x y) (a y x)))
```

We can now put numbers and booleans together: we define a `zero?` function.

```
(define zero? (lambda (n) (n (lambda (x) #f) #t)))
(test (->bool (and (zero? 0) (not (zero? 3)))) => '#t)
```

(Good question: is this fast?)

(Note that it is better to test that the value is explicitly `#t`, if we just use `(test (->bool ...))` then the test will work even if the expression in question evaluated to some bogus value.)

The idea is simple — if `n` is the encoding of zero, it will return it's second argument which is `#t`:

```
(zero? 0) --> ((lambda (f n) n) (lambda (x) #f) #t) -> #t
```

if `n` is an encoding of a bigger number, then it is a self-composition, and the function that we give it is one that always returns `#f`, no matter how many times it is self-composed. Try `2` for example:

```
(zero? 2) --> ((lambda (f n) (f (f n))) (lambda (x) #f) #t)
          --> ((lambda (x) #f) ((lambda (x) #f) #t))
          --> #f
```

Now, how about an encoding for compound values? A minimal approach is what we use in Racket — a way to generate pairs (`cons`), and encode lists as chains of pairs with a special value at the end (`null`). There is a natural encoding for pairs that we have previously seen — a pair is a function that expects a selector, and will apply that on the two values:

```
(define cons (lambda (x y) (lambda (s) (s x y))))
```

Or, equivalently:

```
(define cons (lambda (x y s) (s x y)))
```

To extract the two values from a pair, we need to pass a selector that consumes two values and returns one of them. In our framework, this is exactly what the two boolean values do, so we get:

```
(define car (lambda (x) (x #t)))
(define cdr (lambda (x) (x #f)))

(->nat (+ (car (cons 2 3)) (cdr (cons 2 3))))
```

We can even do this:

```
(define 1st car)
(define 2nd (lambda (l) (car (cdr l))))
(define 3rd (lambda (l) (car (cdr (cdr l)))))
(define 4th (lambda (l) (car (cdr (cdr (cdr l))))))
(define 5th (lambda (l) (car (cdr (cdr (cdr (cdr l)))))))
```

or write a `list-ref` function:

```
(define list-ref (lambda (l n) (car (n cdr l))))
```

Note that we don't need a recursive function for this: our encoding of natural numbers makes it easy to "iterate N times". What we get with this encoding is essentially free natural-number recursion.

We now need a special `null` value to mark list ends. This value should have the same number of arguments as a `cons` value (one: a selector/boolean function), and it should be possible to distinguish it from other values. We choose

```
(define null (lambda (s) #t))
```

Testing the list encoding:

```
(define l123 (cons 1 (cons 2 (cons 3 null))))
(->nat (2nd l123))
```

And as with natural numbers and booleans, Schlac has built-in facility to convert encoded lists to Racket values, except that this requires specifying the type of values in a list so it's a higher-order function:

```
((->listof ->nat) l123)
```

which ("as usual") can be written as

```
(->listof ->nat l123)
```

We can even do this:

```
(->listof (->listof ->nat) (cons l123 (cons l123 null)))
```

Defining `null?` is now relatively easy (and it's actually already used by the above `->listof` conversion). The following definition

```
(define null? (lambda (x) (x (lambda (x y) #f))))
```

works because if `x` is null, then it simply ignores its argument and returns `#t`, and if it's a pair, then it uses the input selector, which always returns `#f` in its turn. Using some arbitrary `A` and `B`:

```
(null? (cons A B))
  --> ((lambda (x) (x (lambda (x y) #f))) (lambda (s) (s A B)))
  --> ((lambda (s) (s A B)) (lambda (x y) #f))
  --> ((lambda (x y) #f) A B)
  --> #f
(null? null)
  --> ((lambda (x) (x (lambda (x y) #f))) (lambda (s) #t))
  --> ((lambda (s) #t) (lambda (x y) #f))
  --> #t
```

We can use the Y combinator to create recursive functions — we can even use the rewrite rules facility that Schlac contains (the same one that we have previously seen):

```
(define Y
  (lambda (f)
    ((lambda (x) (x x)) (lambda (x) (f (x x))))))
(rewrite (define/rec f E) => (define f (Y (lambda (f) E))))
```

and using it:

```
(define/rec length
  (lambda (l)
    (if (null? l)
      0
      (add1 (length (cdr l))))))
(->nat (length l123))
```

And to complete this, um, journey — we're still missing subtraction. There are many ways to solve the problem of subtraction, and for a challenge try to come up with a solution yourself. One of the clearer solutions uses a simple idea — begin with a pair of two zeroes `<0,0>`, and repeat this transformation `n` times: `<a,b>` -> `<b,b+1>`. After `n` steps, we will have `<n-1,n>` — so we get:

```
(define inccons (lambda (p) (cons (cdr p) (add1 (cdr p)))))
(define sub1 (lambda (n) (car (n inccons (cons 0 0)))))
(->nat (sub1 5))
```

And from this the road is short to general subtraction, `m-n` is simply `n` applications of `sub1` on `m`:

```
(define - (lambda (m n) (n sub1 m)))
(test (->nat (- 3 2)) => '1)
(test (->nat (- (* 4 (* 5 5)) 5)) => '95)
```

We now have a normal-looking language, and we're ready to do anything we want. Here are two popular examples:

```
(define/rec fact
  (lambda (x)
    (if (zero? x) 1 (* x (fact (sub1 x))))))
(test (->nat (fact 5)) => '120)

(define/rec fib
  (lambda (x)
    (if (or (zero? x) (zero? (sub1 x)))
      1
      (+ (fib (- x 1)) (fib (- x 2))))))
(test (->nat (fib (* 5 2))) => '89)
```

To get generalized arithmetic capability, Schlac has yet another built-in facility for translating Racket natural numbers into Church numerals:

```
(->nat (fib (nat-> '10)))
```

… and to get to that frightening expression in the beginning, all you need to do is replace all definitions in the `fib` definition over and over again until you're left with nothing but lambda expressions and applications, then reformat the result into some cute shape. For extra fun, you can look for immediate applications of lambda expressions and reduce them manually.

All of this is in the following code:

```
;; Making Schlac into a practical language (not an interpreter)

#lang pl schlac
```

```
(define identity (lambda (x) x))

;; Natural numbers

(define 0 (lambda (f x) x))

(define add1 (lambda (n) (lambda (f x) (f (n f x)))))
;; same as:
;;   (define add1 (lambda (n) (lambda (f x) (n f (f x)))))

(define 1 (add1 0))
(define 2 (add1 1))
(define 3 (add1 2))
(define 4 (add1 3))
(define 5 (add1 4))
(test (->nat (add1 (add1 5))) => '7)

(define + (lambda (m n) (m add1 n)))
(test (->nat (+ 4 5)) => '9)

;; (define * (lambda (m n) (m (+ n) 0)))
(define * (lambda (m n f) (m (n f))))
(test (->nat (* 4 5)) => '20)
(test (->nat (+ 4 (* (+ 2 5) 5))) => '39)

;; (define ^ (lambda (m n) (n (* m) 1)))
(define ^ (lambda (m n) (n m)))
(test (->nat (^ 3 4)) => '81)

;; Booleans

(define #t (lambda (x y) x))
(define #f (lambda (x y) y))

(define if (lambda (c t e) (c t e))) ; not really needed

(test (->nat (if #t 1 2)) => '1)
(test (->nat (if #t (+ 4 5) (+ '1 '2))) => '9)

(define and (lambda (a b) (a b a)))
(define or  (lambda (a b) (a a b)))
;; (define not (lambda (a) (a #f #t)))
(define not (lambda (a x y) (a y x)))

(test (->bool (and #f #f)) => '#f)
(test (->bool (and #t #f)) => '#f)
(test (->bool (and #f #t)) => '#f)
(test (->bool (and #t #t)) => '#t)
(test (->bool (or  #f #f)) => '#f)
(test (->bool (or  #t #f)) => '#t)
(test (->bool (or  #f #t)) => '#t)
(test (->bool (or  #t #t)) => '#t)
(test (->bool (not #f)) => '#t)
(test (->bool (not #t)) => '#f)

(define zero? (lambda (n) (n (lambda (x) #f) #t)))
(test (->bool (and (zero? 0) (not (zero? 3)))) => '#t)
```

```
;; Lists

(define cons (lambda (x y s) (s x y)))

(define car (lambda (x) (x #t)))
(define cdr (lambda (x) (x #f)))

(test (->nat (+ (car (cons 2 3)) (cdr (cons 2 3)))) => '5)

(define 1st car)
(define 2nd (lambda (l) (car (cdr l))))
(define 3rd (lambda (l) (car (cdr (cdr l)))))
(define 4th (lambda (l) (car (cdr (cdr (cdr l))))))
(define 5th (lambda (l) (car (cdr (cdr (cdr (cdr l)))))))

(define null (lambda (s) #t))
(define null? (lambda (x) (x (lambda (x y) #f))))

(define l123 (cons 1 (cons 2 (cons 3 null))))
;; Note that `->listof' is a H.O. converter
(test ((->listof ->nat) l123) => '(1 2 3))
(test (->listof ->nat l123) => '(1 2 3))  ; same as the above
(test (->listof (->listof ->nat) (cons l123 (cons l123 null)))
      => '((1 2 3) (1 2 3)))

;; Subtraction is tricky

(define inccons (lambda (p) (cons (cdr p) (add1 (cdr p)))))
(define sub1 (lambda (n) (car (n inccons (cons 0 0)))))

(test (->nat (sub1 5)) => '4)

(define - (lambda (a b) (b sub1 a)))

(test (->nat (- 3 2)) => '1)
(test (->nat (- (* 4 (* 5 5)) 5)) => '95)
(test (->nat (- 2 4)) => '0) ; this is "natural subtraction"

;; Recursive functions

(define Y
  (lambda (f)
    ((lambda (x) (x x)) (lambda (x) (f (x x))))))
(rewrite (define/rec f E) => (define f (Y (lambda (f) E))))

(define/rec length
  (lambda (l)
    (if (null? l)
      0
      (add1 (length (cdr l))))))
(test (->nat (length l123)) => '3)

(define/rec fact
  (lambda (x)
    (if (zero? x) 1 (* x (fact (sub1 x))))))
(test (->nat (fact 5)) => '120)
```

```
(define/rec fib
  (lambda (x)
    (if (or (zero? x) (zero? (sub1 x)))
      1
      (+ (fib (sub1 x)) (fib (sub1 (sub1 x)))))))
(test (->nat (fib (* 5 2))) => '89)

#|
;; Fully-expanded Fibonacci
(define fib
  ((lambda (f)
     ((lambda (x) (x x)) (lambda (x) (f (x x)))))
   (lambda (f)
     (lambda (x)
       ((lambda (c t e) (c t e))
        ((lambda (a b) (a a b))
         ((lambda (n)
            (n (lambda (x) (lambda (x y) y)) (lambda (x y) x)))
          x)
         ((lambda (n)
            (n (lambda (x) (lambda (x y) y)) (lambda (x y) x)))
          ((lambda (n)
             ((lambda (x) (x (lambda (x y) x)))
              (n (lambda (p)
                   ((lambda (x y s) (s x y))
                    ((lambda (x) (x (lambda (x y) y))) p)
                    ((lambda (n) (lambda (f x) (f (n f x))))
                     ((lambda (x) (x (lambda (x y) y))) p))))
                 ((lambda (x y s) (s x y))
                  (lambda (f x) x)
                  (lambda (f x) x)))))
           x)))
        ((lambda (n) (lambda (f x) (f (n f x)))) (lambda (f x) x))
        ((lambda (x y)
           (x (lambda (n) (lambda (f x) (f (n f x)))) y))
         (f ((lambda (n)
               ((lambda (x) (x (lambda (x y) x)))
                (n (lambda (p)
                     ((lambda (x y s) (s x y))
                      ((lambda (x) (x (lambda (x y) y))) p)
                      ((lambda (n) (lambda (f x) (f (n f x))))
                       ((lambda (x) (x (lambda (x y) y))) p))))
                   ((lambda (x y s) (s x y))
                    (lambda (f x) x)
                    (lambda (f x) x)))))
             x))
          (f ((lambda (n)
                ((lambda (x) (x (lambda (x y) x)))
                 (n (lambda (p)
                      ((lambda (x y s) (s x y))
                       ((lambda (x) (x (lambda (x y) y))) p)
                       ((lambda (n) (lambda (f x) (f (n f x))))
                        ((lambda (x) (x (lambda (x y) y))) p))))
                    ((lambda (x y s) (s x y))
                     (lambda (f x) x)
                     (lambda (f x) x)))))
              ((lambda (n)
                 ((lambda (x) (x (lambda (x y) x)))
```

```
                (n (lambda (p)
                     ((lambda (x y s) (s x y))
                      ((lambda (x) (x (lambda (x y) y))) p)
                      ((lambda (n) (lambda (f x) (f (n f x))))
                       ((lambda (x) (x (lambda (x y) y))) p))))
                   ((lambda (x y s) (s x y))
                    (lambda (f x) x)
                    (lambda (f x) x)))))
             x)))))))))

;; The same after reducing all immediate function applications
(define fib
  ((lambda (f)
     ((lambda (x) (x x)) (lambda (x) (f (x x)))))
   (lambda (f)
     (lambda (x)
       (((x (lambda (x) (lambda (x y) y)) (lambda (x y) x))
         (x (lambda (x) (lambda (x y) y)) (lambda (x y) x))
         (((x (lambda (p)
                (lambda (s)
                  (s (p (lambda (x y) y))
                     (lambda (f x)
                       (f ((p (lambda (x y) y)) f x))))))
              (lambda (s)
                (s (lambda (f x) x) (lambda (f x) x))))
           (lambda (x y) x))
          (lambda (x) (lambda (x y) y))
          (lambda (x y) x)))
        (lambda (f x) (f x))
        ((f ((x (lambda (p)
                  (lambda (s)
                    (s (p (lambda (x y) y))
                       (lambda (f x)
                         (f ((p (lambda (x y) y)) f x))))))
                (lambda (y s)
                  (s (lambda (f x) x) (lambda (f x) x))))
            (lambda (x y) x)))
         (lambda (n) (lambda (f x) (f (n f x))))
         (f ((((x (lambda (p)
                    (lambda (s)
                      (s (p (lambda (x y) y))
                         (lambda (f x)
                           (f ((p (lambda (x y) y)) f x))))))
                  (lambda (s)
                    (s (lambda (f x) x) (lambda (f x) x))))
              (lambda (x y) x))
             (lambda (p)
               (lambda (s)
                 (s (p (lambda (x y) y))
                    (lambda (f x)
                      (f ((p (lambda (x y) y)) f x))))))
             (lambda (s)
               (s (lambda (f x) x) (lambda (f x) x))))
            (lambda (x y) x)))))))))

;; Cute reformatting of the above:
(define fib((lambda(f)((lambda(x)(x x))(lambda(x)(f(x x)))))(lambda(
f)(lambda(x)(((x(lambda(x)(lambda(x y)y))(lambda(x y)x))(x(lambda(x)
```

```
(lambda(x y)y))(lambda(x y) x))(((x(lambda(p)(lambda(s)(s(p(lambda(x
y)y))(lambda(f x)(f((p(lambda(x y)y))f x))))))(lambda(s) (s(lambda(f
x)x)(lambda(f x)x))))(lambda(x y)x))(lambda(x)(lambda(x y)y))(lambda
(x y)x)))(lambda(f x)(f x))((f((x(lambda(p)(lambda(s)(s(p(lambda(x y
)y))(lambda(f x)(f((p(lambda(x y)y))f x))))))(lambda(y s)(s(lambda(f
x)x)(lambda(f x)x))))(lambda(x y)x)))(lambda(n)(lambda(f x)(f(n f x)
)))(f((((x(lambda(p)(lambda(s)(s(p (lambda(x y)y))(lambda(f x)(f((p(
lambda(x y) y))f x))))))(lambda(s)(s(lambda(f x)x)(lambda(f x)x))))(
;;                       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
;;                       `---------------(cons 0 0)---------------'
lambda(x y)x))(lambda(p)(lambda(s)(s(p(lambda(x y)y))(lambda(f x)(f(
(p(lambda(x y)y))f x))))))(lambda(s)(s(lambda(f x)x)(lambda(f x)x)))
)(lambda(x y)x)))))))))

;; And for extra fun:

         (λ(f)(λ
         (x)(((x(λ(
         x)(λ(x y)y)
         )(λ(x y)x))(
         x(λ(x)(λ(x y)
             y))(λ(x y
               )x))(((
                x(λ(p)(
                 λ(s)(s
                 (p (λ(
                 x y)y))
                  (λ(f x
                  )(f((p(
                   λ(x y)
                   y))f x
                   ))))))(
                   λ(s)(s(
                  λ(f x)x)
                   (λ(f x)x)
                  )))(λ(x y)
                  x))(λ(x)(λ(
                  x y)y)) (λ(
                 x y) x)))(λ(
                 f x)(f x))((f
                ((x(λ(p )(λ (s
                )(s(p(  λ(x y)
               y))(λ (  f x)(f(
               (p (λ(    x y)y)
              )f x)))    )))(λ(
              y s)(s     (λ (f x
              )x)(λ(      f x)x)
             )))(λ(       x y)x))
             )(λ(n)        (λ (f
            x)(f (n        f x)))
            )(f(((         (x(λ(p)
           (λ(s)(s          (p( λ(
           x y )y           ))(λ(f
          x) (f((           p(λ(x y
          )y)) f             x)))))
         )(λ(s)(             s(λ(f x
         )x)(λ(               f x)x)
         ))) (λ               (x y)x
```

```
       ))(λ(p                )(λ(s)(
       s(p(λ(                 x y)y)
      )(λ (f                  x)(f((
      p(λ (x                  y)y)) f
     x))))))                   (λ(s)(
     s(λ (f                    x)x)(λ
    (f x)x)                     )))(λ(
    x y)x)                      ))))))

|#
```

Alternative Church Encoding

Finally, note that this is just one way to encode things — other encodings are possible. One alternative encoding is in the following code — it uses a list of `N` falses as the encoding for `N`. This encoding makes it easier to `add1` (just `cons` another `#f`), and to `sub1` (simply `cdr`). The tradeoff is that some arithmetics operations becomes more complicated, for example, the definition of `+` requires the fixpoint combinator. (As expected, some people want to see what can we do with a language without recursion, so they don't like jumping to Y too fast.)

```
;; An alternative "Church" encoding: use lists to encode numbers

#lang pl schlac

(define identity (lambda (x) x))

;; Booleans (same as before)

(define #t (lambda (x y) x))
(define #f (lambda (x y) y))

(define if (lambda (c t e) (c t e))) ; not really needed

(test (->bool (if #t #f #t))
      => '#f)
(test (->bool (if #f ((lambda (x) (x x)) (lambda (x) (x x))) #t))
      => '#t)

(define and (lambda (a b) (a b a)))
(define or  (lambda (a b) (a a b)))
(define not (lambda (a x y) (a y x)))

(test (->bool (and #f #f)) => '#f)
(test (->bool (and #t #f)) => '#f)
(test (->bool (and #f #t)) => '#f)
(test (->bool (and #t #t)) => '#t)
(test (->bool (or  #f #f)) => '#f)
(test (->bool (or  #t #f)) => '#t)
(test (->bool (or  #f #t)) => '#t)
(test (->bool (or  #t #t)) => '#t)
(test (->bool (not #f)) => '#t)
(test (->bool (not #t)) => '#f)

;; Lists (same as before)

(define cons (lambda (x y s) (s x y)))
```

```
(define car (lambda (x) (x #t)))
(define cdr (lambda (x) (x #f)))

(define 1st car)
(define 2nd (lambda (l) (car (cdr l))))
(define 3rd (lambda (l) (car (cdr (cdr l)))))
(define 4th (lambda (l) (car (cdr (cdr (cdr l))))))
(define 5th (lambda (l) (car (cdr (cdr (cdr (cdr l)))))))

(define null (lambda (s) #t))
(define null? (lambda (x) (x (lambda (x y) #f))))

;; Natural numbers (alternate encoding)

(define 0 identity)

(define add1 (lambda (n) (cons #f n)))

(define zero? car) ; tricky

(define sub1 cdr) ; this becomes very simple

;; Note that we could have used something more straightforward:
;; (define 0 null)
;; (define add1 (lambda (n) (cons #t n))) ; cons anything
;; (define zero? null?)
;; (define sub1 (lambda (l) (if (zero? l) l (cdr l))))

(define 1 (add1 0))
(define 2 (add1 1))
(define 3 (add1 2))
(define 4 (add1 3))
(define 5 (add1 4))

(test (->nat* (add1 (add1 5))) => '7)
(test (->nat* (sub1 (sub1 (add1 (add1 5))))) => '5)
(test (->bool (and (zero? 0) (not (zero? 3)))) => '#t)
(test (->bool (zero? (sub1 (sub1 (sub1 3))))) => '#t)

;; list-of-numbers tests
(define l123 (cons 1 (cons 2 (cons 3 null))))
(test (->listof ->nat* l123) => '(1 2 3))
(test (->listof (->listof ->nat*) (cons l123 (cons l123 null)))
      => '((1 2 3) (1 2 3)))

;; Recursive functions

(define Y
  (lambda (f)
    ((lambda (x) (x x)) (lambda (x) (f (x x))))))
(rewrite (define/rec f E) => (define f (Y (lambda (f) E))))

;; note that this example is doing something silly now
(define/rec length
  (lambda (l)
    (if (null? l)
      0
      (add1 (length (cdr l))))))
```

```
(test (->nat* (length l123)) => '3)

;; addition becomes hard since it requires a recursive definition
;; (define/rec +
;;   (lambda (m n) (if (zero? n) m (+ (add1 m) (sub1 n)))))
;; (test (->nat* (+ 4 5)) => '9)

;; faster alternative:
(define/rec +
  (lambda (m n)
    (if (zero? m) n
        (if (zero? n) m
            (add1 (add1 (+ (sub1 m) (sub1 n))))))))
(test (->nat* (+ 4 5)) => '9)

;; subtraction is similar to addition
;; (define/rec -
;;   (lambda (m n) (if (zero? n) m (- (sub1 m) (sub1 n)))))
;; (test (->nat* (- (+ 4 5) 4)) => '5)
;; but this is not "natural subtraction": doesn't work when n>m,
;; because (sub1 0) does not return 0.

;; a solution is like alternative form of +:
(define/rec -
  (lambda (m n)
    (if (zero? m) 0
        (if (zero? n) m
            (- (sub1 m) (sub1 n))))))
(test (->nat* (- (+ 4 5) 4)) => '5)
(test (->nat* (- 2 5)) => '0)
;; alternatively, could change sub1 above:
;; (define sub1 (lambda (n) (if (zero? n) n (cdr n))))

;; we can do multiplication in a similar way
(define/rec *
  (lambda (m n)
    (if (zero? m) 0
        (+ n (* (sub1 m) n)))))
(test (->nat* (* 4 5)) => '20)
(test (->nat* (+ 4 (* (+ 2 5) 5))) => '39)

;; and the rest of the examples

(define/rec fact
  (lambda (x)
    (if (zero? x) 1 (* x (fact (sub1 x))))))
(test (->nat* (fact 5)) => '120)

(define/rec fib
  (lambda (x)
    (if (or (zero? x) (zero? (sub1 x)))
      1
      (+ (fib (sub1 x)) (fib (sub1 (sub1 x)))))))
(test (->nat* (fib (* 5 2))) => '89)

#|
;; Fully-expanded Fibonacci (note: much shorter than the previous
;; encoding, but see how Y appears twice -- two "((lambda" pairs)
```

```
(define fib((lambda(f)((lambda(x)(x x))(lambda(x)(f(x x)))))(lambda(
f)(lambda(x)(((((x(lambda(x y)x))(x(lambda(x y)x)))((x(lambda(x y)y)
)(lambda(x y)x)))(lambda(s)(s(lambda(x y)y)(lambda(x)x))))(((lambda
(f)((lambda(x)(x x))(lambda(x)(f(x x))))) (lambda(f)(lambda(m n)((m(
lambda(x y)x))n (((n(lambda(x y)x)) m)(lambda(s)((s (lambda(x y)y))(
lambda(s)((s (lambda(x y)y))((f(m(lambda(x y)y)))(n(lambda(x y)y))))
))))))))(f(x(lambda(x y)y))))(f((x(lambda(x y)y))(lambda(x y)y))))))
)))
|#
```

Implementing `define-type` & `cases` in Schlac

(The following explanation originates from Chris Okasaki.)

Another interesting way to implement lists follows the pattern matching approach, where both pairs and the null value are represented by a function that serves as a kind of a `match` dispatcher. This function takes in two inputs — if it is the representation of null then it will return the first input, and if it is a pair, then it will apply the second input on the two parts of the pair. This is implemented as follows (with type comments to make it clear):

```
;; null : List
(define null
  (lambda (n p)
    n))

;; cons : A List -> List
(define cons
  (lambda (x y)
    (lambda (n p)
      (p x y))))
```

This might seem awkward, but it follows the intended use of pairs and null as a match-like construct. Here is an example, with the equivalent Racket code on the side:

```
;; Sums up a list of numbers
(define/rec (sum l)
  (l                    ; (match l
   0                    ;   ['() 0]
   (lambda (x xs)       ;   [(cons x xs)
     (+ x (sum xs)))))  ;    (+ x (sum xs))])
```

In fact, it's easy to implement our selectors and predicate using this:

```
(define null? (lambda (l) (l #t (lambda (x xs) #f))))
(define car   (lambda (l) (l #f (lambda (x xs) x))))
(define cdr   (lambda (l) (l #f (lambda (x xs) xs))))
;; in the above `#f' is really any value, since it
;; should be an error alternatively:
(define car (lambda (l)
              (l ((lambda (x) (x x)) (lambda (x) (x x))) ; "error"
                 (lambda (x y) x))))
```

The same approach can be used to define any kind of new data type in a way that looks like our own `define-type` definitions. For example, consider a much-simplified definition of the AE type we've seen early in the semester, and a matching `eval` definition as an example for using `cases`:

```
(define-type AE
  [Num Number]
  [Add AE AE])
```

```
(: eval : AE -> Number)
(define (eval expr)
  (cases expr
    [(Num n)   n]
    [(Add l r) (+ (eval l) (eval r))]))
```

We can follow the above approach now to write Schlac code that more than being equivalent, is also very similar in nature. Note that the type definition is replaced by two definitions for the two constructors:

```
(define Num (lambda (n)   (lambda (num add) (num n  ))))
(define Add (lambda (l r) (lambda (num add) (add l r))))
(define/rec eval
  (lambda (expr) ; `expr` is always a (lambda (num add) ...), and it
                 ; expects a unary `num` argument and a binary `add`
    (expr (lambda (n)   n)
          (lambda (l r) (+ (eval l) (eval r))))))
(test (->nat (eval (Add (Num 1) (Num 2)))) => '3)
```

We can even take this further: the translations from `define-type` and `cases` are mechanical enough that we could implement them almost exactly via rewrites (there are a subtle change in that we're list field names rather than types):

```
(rewrite (define-type -ignored-Type- [Variant arg ...] ...)
      => (define Variant
           (lambda (arg ...)
             (lambda (Variant ...) (Variant arg ...))))
         ...)
(rewrite (cases value [(-ignored-Variant- arg ...) result] ...)
      => (value (lambda (arg ...) result) ...))
```

And using that, an evluator is simple:

```
(define-type AE [Num n] [Add l r] [Sub l r] [Mul l r])
(define/rec eval
  (lambda (expr)
    (cases expr
      [(Num n)   n]
      [(Add l r) (+ (eval l) (eval r))]
      [(Sub l r) (- (eval l) (eval r))]
      [(Mul l r) (* (eval l) (eval r))])))
(test (->nat (eval (Mul (Add (Num 1) (Num 2))
                        (Sub (Num 4) (Num 2)))))
      => '6)
```