PL: Lecture #18

Tuesday, November 5th

Lazy Evaluation: Using a Lazy Racket

> 👉 ***PLAI §7*** *(done with Haskell)*

For this part, we will use a new language, Lazy Racket.

```
#lang pl lazy
```

As the name suggests, this is a version of the normal (untyped) Racket language that is lazy.

First of all, let's verify that this is indeed a lazy language:

```
> (define (foo x) 3)
> (foo (+ 1 "2"))
3
```

That went without a problem — the argument expression was indeed not evaluated. In this language, you can treat all expressions as future `promises` to evaluate. There are certain points where such promises are actually `forced`, all of these stem from some need to print a resulting value, in our case, it's the REPL that prints such values:

```
> (+ 1 "2")
+: expects type <number> as 2nd argument,
given: "2"; other arguments were: 1
```

The expression by itself only generates a promise, but when we want to print it, this promise is forced to evaluate — this forces the addition, which forces its arguments (plain values rather than computation promises), and at this stage we get an error. (If we never want to see any results, then the language will never do anything at all.) So a promise is forced either when a value printout is needed, or if it is needed to recursively compute a value to print:

```
> (* 1 (+ 2 "3"))
+: expects type <number> as 2nd argument,
given: "3"; other arguments were: 2
```

Note that the error was raised by the internal expression: the outer expression uses `*`, and `+` requires actual values not promises.

Another example, which is now obvious, is that we can now define an `if` function:

```
> (define (my-if x y z) (if x y z))
> (my-if (< 1 2) 3 (+ 4 "5"))
3
```

Actually, in this language `if`, `and`, and `or` are all function values instead of special forms:

```
> (list if and or)
(#<procedure:if> #<procedure:and> #<procedure:or>)
> ((third (list if and or)) #t (+ 1 "two"))
#t
```

(By now, you should know that these have no value in Racket — using them like this in plain will lead to syntax errors.) There are some primitives that do not force their arguments. Constructors fall in this

category, for example `cons` and `list`:

```
> (define (fib n) (if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2)))))
> (define a (list (+ 1 2) (+ 3 "4") (fib 30) (* 5 6)))
```

Nothing — the definition simply worked, but that's expected, since nothing is printed. If we try to inspect this value, we can get some of its parts, provided we do not force the bogus one:

```
> (first a)
3
> (fourth a)
30
> (third a)
196418
> (second a)
+: contract violation, expected: number?, given: "4" ...
```

The same holds for cons:

```
> (second (cons 1 (cons 2 (first null))))
2
```

Now if this is the case, then how about this:

```
> (define ones (cons 1 ones))
```

Everything is fine, as expected — but what is the value of `ones` now? Clearly, it is a list that has 1 as its first element:

```
> (first ones)
1
```

But what do we have in the tail of this list? We have `ones` which we already know is a list that has 1 in its first place — so following Racket's usual rules, it means that the second element of `ones` is, again, 1. If we continue this, we can see that `ones` is, in fact, an *infinite* list of 1s:

```
> (second ones)
1
> (fifth ones)
1
```

In this sense, the way `define` behaves is that it defines a true equation: if ones is defined as (cons 1 ones), then the real value does satisfy

```
(equal? ones (cons 1 ones))
```

which means that the value is the fixpoint of the defined expression.

We can use `append` in a similar way:

```
> (define foo (append (list 1 2 3) foo))
> (fourth foo)
1
```

This looks like it has some common theme with the discussion of implementing recursive environments — it actually demonstrates that in this language, `letrec` can be used for *simple* values too. First of all, a side note — here an expression that indicated a bug in our substituting evaluator:

```
> (let ([x (list y)])
    (let ([y 1])
      x))
reference to undefined identifier: y
```

When our evaluator returned `1` for this, we noticed that this was a bug: it does not obey the lexical scoping rules. As seen above, Lazy Racket is correctly using lexical scope. Now we can go back to the use of `letrec` — what do we get by this definition:

```
> (define twos (let ([xs (cons 2 xs)]) xs))
```

we get an error about `xs` being undefined.

`xs` is unbound because of the usual scope that `let` uses. How can we make this work? — We simply use `letrec`:

```
> (define twos (letrec ([xs (cons 2 xs)]) xs))
> (first twos)
2
```

As expected, if we try to print an infinite list will cause an infinite loop, which DrRacket catches and prints in that weird way:

```
> twos
#0=(2 . #0#)
```

How would we inspect an infinite list? We write a function that returns part of it:

```
> (define (take n l)
    (if (or (<= n 0) (null? l))
      null
      (cons (first l) (take (sub1 n) (rest l)))))
> (take 10 twos)
(2 2 2 2 2 2 2 2 2 2)
> (define foo (append (list 1 2 3) foo))
> (take 10 foo)
(1 2 3 1 2 3 1 2 3 1)
```

Dealing with infinite lists can lead to lots of interesting things, for example:

```
> (define fibs (cons 1 (cons 1 (map + fibs (rest fibs)))))
> (take 10 fibs)
(1 1 2 3 5 8 13 21 34 55)
```

To see how it works, see what you know about `fibs[n]` which will be our notation for the nth element of `fibs` (starting from `1`):

```
fibs[1] = 1  because of the first `cons'
fibs[2] = 1  because of the second `cons'
```

and for all `n>2`:

```
fibs[n] = (map + fibs (rest fibs))[n-2]
        = fibs[n-2] + (rest fibs)[n-2]
        = fibs[n-2] + fibs[n-2+1]
        = fibs[n-2] + fibs[n-1]
```

so it follows the exact definition of Fibonacci numbers.

Note that the list examples demonstrate that laziness applies to nested values (actually, nested computations) too: a value that is not needed is not computed, even if it is *contained* in a value that is needed. For example, in:

```
(define x (/ 1 0))
(if (list (+ 1 x)) 1 2)
```

the `if` needs to know only whether its first argument (note: it *is* an *argument*, since this `if` is a function) is `#f` or not. Once it is determined that it is a pair (a `cons` cell), there is no need to actually look at the

values inside the pair, and therefore `(+ 1 x)` (and more specifically, `x`) is never evaluated and we see no error.

Lazy Evaluation: Some Issues

There are a few issues that we need to be aware of when we're dealing with a lazy language. First of all, remember that our previous attempt at lazy evaluation has made

```
{with {x y}
  {with {y 1}
    x}}
```

evaluate to 1, which does not follow the rules of lexical scope. This is *not* a problem with lazy evaluation, but rather a problem with our naive implementation. We will shortly see a way to resolve this problem. In the meanwhile, remember that when we try the same in Lazy Racket we do get the expected error:

```
> (let ([x y])
    (let ([y 1])
      x))
reference to undefined identifier: y
```

A second issue is a subtle point that you might have noticed when we played with Lazy Racket: for some of the list values we have see a " `.` " printed. This is part of the usual way Racket displays an *improper list* — any list that does not terminate with a null value. For example, in plain Racket:

```
> (cons 1 2)
(1 . 2)
> (cons 1 (cons 2 (cons 3 4)))
(1 2 3 . 4)
```

In the dialect that we're using in this course, this is not possible. The secret is that the `cons` that we use first checks that its second argument is a proper list, and it will raise an error if not. So how come Lazy Racket's `cons` is not doing the same thing? The problem is that to know that something is a proper list, we will have to force it, which will make it not behave like a constructor.

> *As a side note, we can achieve some of this protection if we don't insist on immediately checking the second argument completely, and instead we do the check when needed — lazily:*
>
> ```
> (define (safe-cons x l)
>   (cons x (if (pair? l) l (error "poof"))))
> ```

Finally, there are two consequences of using a lazy language that make it more difficult to debug (or at lease take some time to get used to). First of all, control tends to flow in surprising ways. For example, enter the following into DrRacket, and run it in the normal language level for the course:

```
                  (define (foo3 x)
                    (/ x "1"))
(define (foo2 x)
  (foo3 x))
                  (define (foo1 x)
                    (list (foo2 x)))
(define (foo0 x)
  (first (foo1 x)))

          (+ 1 (foo0 3))
```

In the normal language level, we get an error, and red arrows that show us how where in the computation the error was raised. The arrows are all expected, except that `foo2` is not in the path — why is that? Remember the discussion about tail-calls and how they are important in Racket since they are the only tool

to generate loops? This is what we're seeing here: `foo2` calls `foo3` in a tail position, so there is no need to keep the `foo2` context anymore — it is simply replaced by `foo3`. (Incidentally, there is also no arrow that goes through `foo1`: Racket does some smart inlining, and it figures out that `foo0+foo1` are simply returning the same value, so it skips `foo1`.)

Now switch to Lazy Racket and re-run — you'll see no arrows at all. What's the problem? The call of `foo0` creates a promise that is forced in the top-level expression, that simply returns the `first` of the `list` that `foo1` created — and all of that can be done without forcing the `foo2` call. Going this way, the computation is finally running into an error *after* the calls to `foo0`, `foo1`, and `foo2` are done — so we get the seemingly out-of-context error.

To follow what's happening here, we need to follow how promise are forced: when we have code like

```
> (define (foo x) (/ x 0))
> (foo 9)
```

then the `foo` call is a *strict point*, since we need an actual value to display on the REPL. Since it's in a strict position, we do the call, but when we're in the function there is no need to compute the division result — so it is returned as a lazy promise value back to the toplevel. It is only then that we continue the process of getting an actual value, which leads to trying to compute the division and get the error.

Finally, there are also potential problems when you're not careful about memory use. A common technique when using a lazy language is to generate an infinite list and pull out its Nth element. For example, to compute the Nth Fibonacci number, we've seen how we can do this:

```
(define fibs (cons 1 (cons 1 (map + fibs (rest fibs)))))
(define (fib n) (list-ref fibs n))
```

and we can also do this (reminder: `letrec` is the same as an internal definition):

```
(define (fib n)
  (letrec ([fibs (cons 1 (cons 1 (map + fibs (rest fibs))))])
    (list-ref fibs n))) ; tail-call => no need to keep `fibs`
```

but the problem here is that when `list-ref` is making its way down the list, it might still hold a reference to `fibs`, which means that as the list is forced, all intermediate values are held in memory. In the first of these two, this is guaranteed to happen since we have a binding that points at the head of the `fibs` list. With the second form things can be confusing: it might be that our language implementation is smart enough to see that `fibs` is not really needed anymore and release the offending reference. If it isn't, then we'd have to do something like

```
(define (fib n)
  (list-ref
   (letrec ([fibs (cons 1 (cons 1 (map + fibs (rest fibs))))])
     fibs)
   n))
```

to eliminate it. But even if the implementation does know that there is no need for that reference, there are other tricky situations that are hard to avoid.

Side note: Racket didn't use to do this optimization, but now it does, and the lazy language helped in clarifying more cases where references should be released. To see that, consider these two variants:

```
(define (nat1 n)
  (define nats (cons 1 (map add1 nats)))
  (if (number? (list-ref nats n))
    "a number"
    "not a number"))

;; we want to provide some information: show the first element
(define (nat2 n)
  (define nats (cons 1 (map add1 nats)))
```

```
  (if (number? (list-ref nats n))
    "a number"
    (error 'nat "the list starting with ~s is broken"
           (first nats))))
```

If we try to use them with a big input:

```
(nat1 300000) ; or with nat2
```

then `nat1` would work fine, whereas `nat2` will likely run into DrRacket's memory limit and the computation will be terminated. The problem is that `nat2` uses the `nats` value *after* the `list-ref` call, which will make a reference to the head of the list, preventing it from being garbage-collected while `list-ref` is `cdr`-ing down the list and making more cons cells materialize.

It's still possible to show the extra information though — just save it:

```
;; we want to provide some information: show the first element
(define (nat3 n)
  (define nats (cons 1 (map add1 nats)))
  (define fst (first nats))
  (if (number? (list-ref nats n))
    "a number"
    (error 'nat "the list starting with ~s is broken" fst)))
```

It looks like it's spending a redundant runtime cycle in the extra computation, but it's a lazy language so this is not a problem.

Lazy Evaluation: Shell Examples

There is a very simple and elegant principle in shell programming — we get a single data type, a character stream, and many small functions, each doing a single simple job. With these small building blocks, we can construct more sequences that achieve more complex tasks, for example — a sorted frequency table of lines in a file:

```
sort foo | uniq -c | sort -nr
```

This is very much like a programming language — we get small blocks, and build stuff out of them. Of course there are swiss army knives like awk that try to do a whole bunch of stuff, (the same attitude that brought Perl to the world…) and even these respect the "stream" data type. For example, a simple `{ print $1 }` statement will work over all lines, one by one, making it a program over an infinite input stream, which is what happens in reality in something like:

```
cat /dev/console | awk ...
```

But there is something else in shell programming that makes so effective: it is implementing a sort of a lazy evaluation. For example, compare this:

```
cat foo | awk '{ print $1+$2; }' | uniq
```

to:

```
cat foo | awk '{ print $1+$2; }' | uniq | head -5
```

Each element in the pipe is doing its own small job, and it is always doing just enough to feed its output. Each basic block is designed to work even on infinite inputs! (Even sort works on unlimited inputs…) (Soon we will see a stronger connection with lazy evaluation.)

> *Side note: (Alan Perlis) "It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures"… But the uniformity comes at a price: the biggest problem shells have is in their lack of a recursive structure, contaminating the world with way too many hacked up solutions. More than that, it is extremely inefficient and usually leads to data being re-parsed over and over and over — each small Unix command needs to*

> *always output stuff that is human readable, but the next command in the pipe will need to re-parse that, eg, rereading decimal numbers. If you look at pipelines as composing functions, then a pipe of numeric commands translates to something like:*
>
> `itoa(baz(atoi(itoa(bar(atoi(itoa(foo(atoi(inp)))))))))`
>
> *and it is impossible to get rid of the redundant* `atoi(itoa(...))`*s.*

Lazy Evaluation: Programming Examples

We already know that when we use lazy evaluation, we are guaranteed to have more robust programs. For example, a function like:

```
(define (my-if x y z)
  (if x y z))
```

is completely useless in Racket because all functions are eager, but in a lazy language, it would behave exactly like the real if. Note that we still need *some* primitive conditional, but this primitive can be a function (and it is, in Lazy Racket).

But we get more than that. If we have a lazy language, then *computations* are pushed around as if they were values (computations, because these are expressions that are yet to be evaluated). In fact, there is no distinction between computations and values, it just happens that some values contain “computational promises”, things that will do something in the future.

To see how this happens, we write a simple program to compute the (infinite) list of prime numbers using the sieve of Eratosthenes. To do this, we begin by defining the list of all natural numbers:

```
(define nats (cons 1 (map add1 nats)))
```

And now define a `sift` function: it receives an integer `n` and an infinite list of integers `l`, and returns a list without the numbers that can be divided by `n`. This is simple to write using `filter`:

```
(define (sift n l)
  (filter (lambda (x) (not (divides? n x))) l))
```

and it requires a definition for `divides?` — we use Racket’s `modulo` for this:

```
(define (divides? n m)
  (zero? (modulo m n)))
```

Now, a `sieve` is a function that consumes a list that begins with a prime number, and returns the prime numbers from this list. To do this, it returns a list that has the same first number, and for its tail it sifts out numbers that are divisible by the first from the original list’s tail, and calls itself recursively on the result:

```
(define (sieve l)
  (cons (first l) (sieve (sift (first l) (rest l)))))
```

Finally, the list of prime numbers is the result of applying `sieve` on the list of numbers from `2`. The whole program is now:

```
#lang pl lazy

(define nats (cons 1 (map add1 nats)))

(define (divides? n m)
  (zero? (modulo m n)))

(define (sift n l)
  (filter (lambda (x) (not (divides? n x))) l))
```

```
(define (sieve l)
  (cons (first l) (sieve (sift (first l) (rest l)))))

(define primes (sieve (rest nats)))
```

To see how this runs, we trace `modulo` to see which tests are being used. The effect of this is that each time `divides?` is actually required to return a value, we will see a line with its inputs, and its output. This output looks quite tricky — things are computed only on a "need to know" basis, meaning that debugging lazy programs can be difficult, since things happen when they are needed which takes time to get used to. However, note that the program actually performs the same tests that you'd do using any eager-language implementation of the sieve of Eratosthenes, and the advantage is that we don't need to decide in advance how many values we want to compute — all values will be computed when you want to see the corresponding result. Implementing *this* behavior in an eager language is more difficult than a simple program, yet we don't need such complex code when we use lazy evaluation.

Note that if we trace `divides?` we see results that are some promise struct — these are unevaluated expressions, and they point at the fact that when `divides?` is used, it doesn't really force its arguments — this happens later when these results are forced.

The analogy with shell programming using pipes should be clear now — for example, we have seen this:

```
cat foo | awk '{ print $1+$2; }' | uniq | head -5
```

The last `head -5` means that no computation is done on parts of the original file that are not needed. It is similar to a `(take 5 l)` expression in Lazy Racket.

Side Note: Similarity to Generators and Channels

Using infinite lists is similar to using channels — a tool for synchronizing threads and (see a **Rob Pike's talk**), and generators (as they exist in Python). Here are examples of both, note how similar they both are, and how similar they are to the above definition of `primes`. (But note that there is an important difference, can you see it? It has to be with whether a stream is reusable or not.)

First, the threads & channels version:

```
#lang racket

(define-syntax-rule (bg expr ...) (thread (lambda () expr ...)))

(define nats
  (let ([out (make-channel)])
    (define (loop i) (channel-put out i) (loop (add1 i)))
    (bg (loop 1))
    out))

(define (divides? n m)
  (zero? (modulo m n)))

(define (filter pred c)
  (define out (make-channel))
  (define (loop)
    (let ([x (channel-get c)])
      (when (pred x) (channel-put out x))
      (loop)))
  (bg (loop))
  out)

(define (sift n c)
  (filter (lambda (x) (not (divides? n x))) c))
```

```
(define (sieve c)
  (define out (make-channel))
  (define (loop c)
    (define first (channel-get c))
    (channel-put out first)
    (loop (sift first c)))
  (bg (loop c))
  out)

(define primes
  (begin (channel-get nats) (sieve nats)))

(define (take n c)
  (if (zero? n) null (cons (channel-get c) (take (sub1 n) c))))

(take 10 primes)
```

And here is the generator version:

```
#lang racket

(require racket/generator)

(define nats
  (generator ()
    (define (loop i)
      (yield i)
      (loop (add1 i)))
    (loop 1)))

(define (divides? n m)
  (zero? (modulo m n)))

(define (filter pred g)
  (generator ()
    (define (loop)
      (let ([x (g)])
        (when (pred x) (yield x))
        (loop)))
    (loop)))

(define (sift n g)
  (filter (lambda (x) (not (divides? n x))) g))

(define (sieve g)
  (define (loop g)
    (define first (g))
    (yield first)
    (loop (sift first g)))
  (generator () (loop g)))

(define primes
  (begin (nats) (sieve nats)))

(define (take n g)
  (if (zero? n) null (cons (g) (take (sub1 n) g))))

(take 10 primes)
```

Call by Need vs Call by Name

Finally, note that on requiring different parts of the `primes`, the same calls are not repeated. This indicates that our language implements “call by need” rather than “call by name”: once an expression is forced, its value is remembered, so subsequent usages of this value do not require further computations.

Using “call by name” means that we actually use expressions which can lead to confusing code. An old programming language that used this is Algol. A confusing example that demonstrates this evaluation strategy is:

```
#lang algol60
begin
  integer procedure SIGMA(x, i, n);
    value n;
    integer x, i, n;
  begin
    integer sum;
    sum := 0;
    for i := 1 step 1 until n do
      sum := sum + x;
    SIGMA := sum;
  end;
  integer q;
  printnln(SIGMA(q*2-1, q, 7));
end
```

`x` and `i` are arguments that are passed by name, which means that they can use the same memory location. This is called *aliasing*, a problem that happens when pointers are involved (eg, pointers in C and `reference` arguments in C++). The code, BTW, is called “Jensen’s device”.

Example of Feature Embedding

Another interesting behavior that we can now observe, is that the TOY evaluation rule for `with`:

```
eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
```

is specifying an eager evaluator *only if* the language that this rule is written in is itself eager. Indeed, if we run the TOY interpreter in Lazy Racket (or other interpreters we have implemented), we can verify that running:

```
(run "{bind {{x {/ 1 0}}} 1}")
```

is perfectly fine — the call to Racket’s division is done in the evaluation of the TOY division expression, but since Lazy Racket is lazy, then if this value is never used then we never get to do this division! On the other hand, if we evaluate

```
(run "{bind {{x {/ 1 0}}} {+ x 1}}")
```

we do get an error when DrRacket tries to display the result, which forces strictness. Note how the arrows in DrRacket that show where the computation is are quite confusing: the computation seem to go directly to the point of the arithmetic operations (`arith-op`) since the rest of the evaluation that the evaluator performed was already done, and succeeded. The actual failure happens when we try to force the resulting promise which contains only the strict points in our code.