PL: Lecture #22

Tuesday, November 19th

Lazy Constructions in an Eager Language

> 👉 ***PLAI §37*** *(has some examples)*

This is not really lazy evaluation, but it gets close, and provides the core useful property of easy-to-use infinite lists.

```
(define-syntax-rule (cons-stream x y)
  (cons x (lambda () y)))
(define stream? pair?)
(define null-stream null)
(define null-stream? null?)
;; note that there are not proper lists in racket,
;; so we use car and cdr here
(define stream-first car)
(define (stream-rest s) ((cdr s)))
```

Using it:

```
(define ones (cons-stream 1 ones))
(define (stream-map f s)
  (if (null-stream? s)
    null-stream
    (cons-stream (f (stream-first s))
                 (stream-map f (stream-rest s)))))
(define (stream-map2 f s1 s2)
  (if (null-stream? s1)
    null-stream
    (cons-stream (f (stream-first s1) (stream-first s2))
                 (stream-map2 f (stream-rest s1)
                                (stream-rest s2)))))
(define ints (cons-stream 0 (stream-map2 + ones ints)))
```

Actually, all Scheme implementations come with a generalized tool for (local) laziness: a `delay` form that delays computation of its body expression, and a `force` function that forces such promises. Here is a naive implementation of this:

```
(define-type promise
  [make-promise (-> Any)])

(define-syntax-rule (delay expr)
  (make-promise (lambda () expr)))

(define (force p)
  (cases p [(make-promise thunk) (thunk)]))
```

Proper definitions of `delay`/`force` cache the result — and practical ones can get pretty complex, for example, in order to allow tail calls via promises.

Recursive Macros

Syntax transformations can be recursive. For example, we have seen how `let*` can be implemented by a transformation that uses two rules, one of which expands to another use of `let*`:

```
(define-syntax let*
  (syntax-rules ()
    [(let* () body ...)
     (let () body ...)]
    [(let* ((x v) (xs vs) ...) body ...)
     (let ((x v)) (let* ((xs vs) ...) body ...))]))
```

When Racket expands a `let*` expression, the result may contain a new `let*` which needs extending as well. An important implication of this is that recursive macros are fine, as long as the recursive case is using a *smaller* expression. This is just like any form of recursion (or loop), where you need to be looping over a `well-founded` set of values — where each iteration uses a new value that is closer to some base case.

For example, consider the following macro:

```
(define-syntax-rule (while condition body ...)
  (when condition
    body ...
    (while condition body ...)))
```

It seems like this is a good implementation of a `while` loop — after all, if you were to implement it as a function using thunks, you'd write very similar code:

```
(define (while condition-thunk body-thunk)
  (when (condition-thunk)
    (body-thunk)
    (while condition-thunk body-thunk)))
```

But if you look at the nested `while` form in the transformation rule, you'll see that it is exactly the same as the input form. This means that this macro can never be completely expanded — it specifies infinite code! In practice, this makes the (Racket) compiler loop forever, consuming more and more memory. This is unlike, for example, the recursive `let*` rule which uses one less binding-value pair than specified as its input.

The reason that the function version of `while` is fine is that it iterates using the *same* code, and the condition thunk will depend on some state that converges to a base case (usually the body thunk will perform some side-effects that makes the loop converge). But in the macro case there is *no* evaluation happening, if the transformed syntax contains the same input pattern, we end up having a macro that expands infinitely.

The correct solution for a `while` macro is therefore to use plain recursion using a local recursive function:

```
(define-syntax-rule (while condition body ...)
  (letrec ([loop (lambda ()
                   (when condition
                     body ...
                     (loop)))])
    (loop)))
```

A popular way to deal with macros like this that revolve around a specific control flow is to separate them into a function that uses thunks, and a macro that does nothing except wrap input expressions as thunks. In this case, we get this solution:

```
(define (while/proc condition-thunk body-thunk)
  (when (condition-thunk)
```

```
    (body-thunk)
    (while/proc condition-thunk body-thunk)))

(define-syntax-rule (while condition body ...)
  (while/proc (lambda () condition)
              (lambda () body ...)))
```

Another example: a simple loop

Here is an implementation of a macro that does a simple arithmetic loop:

```
(define-syntax for
  (syntax-rules (= to do)
    [(for x = m to n do body ...)
     (letrec ([loop (lambda (x)
                      (when (<= x n)
                        body ...
                        (loop (+ x 1))))])
       (loop m))]))
```

(Note that this is not complete code: it suffers from the usual problem of multiple evaluations of the `n` expression. We'll deal with it soon.)

This macro combines both control flow and lexical scope. Control flow is specified by the loop (done, as usual in Racket, as a tail-recursive function) — for example, it determines how code is iterated, and it also determines what the `for` form will evaluate to (it evaluates to whatever `when` evaluates to, the void value in this case). Scope is also specified here, by translating the code to a function — this code makes `x` have a scope that covers the body so this is valid:

```
(for i = 1 to 3 do (printf "i = ~s\n" i))
```

but it also makes the boundary expression `n` be in this scope, making this:

```
(for i = 1 to (if (even? i) 10 20) do (printf "i = ~s\n" i))
```

valid. In addition, while evaluating the condition on each iteration might be desirable, in most cases it's not — consider this example:

```
(for i = 1 to (read) do (printf "i = ~s\n" i))
```

This is easily solved by using a `let` to make the expression evaluate just once:

```
(define-syntax for
  (syntax-rules (= to do)
    [(for x = m to n do body ...)
     (let ([m* m]  ; execution order
           [n* n])
       (letrec ([loop (lambda (x)
                        (when (<= x n*)
                          body ...
                          (loop (+ x 1))))])
         (loop m*)))]))
```

which makes the previous use result in a "`reference to undefined identifier: i`" error.

Furthermore, the fact that we have a hygienic macro system means that it is perfectly fine to use nested `for` expressions:

```
(for a = 1 to 9 do
  (for b = 1 to 9 do (printf "~s,~s " a b))
  (newline))
```

The transformation is, therefore, completely specifying the semantics of this new form.

Extending this syntax is easy using multiple transformation rules — for example, say that we want to extend it to have a `step` optional keyword. The standard idiom is to have the step-less pattern translated into one that uses `step 1`:

```
(for x = m to n do body ...)
--> (for x = m to n step 1 do body ...)
```

Usually, you should remember that `syntax-rules` tries the patterns one by one until a match is found, but in this case there is no problems because the keywords make the choice unambiguous:

```
(define-syntax for
  (syntax-rules (= to do step)
    [(for x = m to n do body ...)
     (for x = m to n step 1 do body ...)]
    [(for x = m to n step d do body ...)
     (let ([m* m]
           [n* n]
           [d* d])
       (letrec ([loop (lambda (x)
                        (when (<= x n*)
                          body ...
                          (loop (+ x d*))))])
         (loop m*)))]))

(for i = 1 to 10 do (printf "i = ~s\n" i))
(for i = 1 to 10 step 2 do (printf "i = ~s\n" i))
```

We can even extend it to do a different kind of iteration, for example, iterate over list:

```
(define-syntax for
  (syntax-rules (= to do step in)
    [(for x = m to n do body ...)
     (for x = m to n step 1 do body ...)]
    [(for x = m to n step d do body ...)
     (let ([m* m]
           [n* n]
           [d* d])
       (letrec ([loop (lambda (x)
                        (when (<= x n*)
                          body ...
                          (loop (+ x d*))))])
         (loop m*)))]
    ;; list
    [(for x in l do body ...)
     (for-each (lambda (x) body ...) l)]))

(for i in (list 1 2 3 4) do (printf "i = ~s\n" i))

(for i in (list 1 2 3 4) do
  (for i = 0 to i do (printf "i = ~s  " i))
  (newline))
```

Yet Another: List Comprehension

At this point it's clear that macros are a powerful language feature that makes it relatively easy to implement new features, making it a language that is easy to use as a tool for quick experimentation with new language features. As an example of a practical feature rather than a toy, let's see how we can implement **Python's list comprehenions**. These are expressions that conveniently combine `map`, `filter`, and nested uses of both.

First, a simple implementation that uses only the `map` feature:

```
(define-syntax list-of
  (syntax-rules (for in)
    [(list-of EXPR for ID in LIST)
     (map (lambda (ID) EXPR)
          LIST)]))

(list-of (* x x) for x in (range 10))
```

It is a good exercise to see how everything that we've seen above plays a role here. For example, how we get the `ID` to be bound in `EXPR`.

Next, add a condition expression with an `if` keyword, and implemented using a `filter`:

```
(define-syntax list-of
  (syntax-rules (for in if)
    [(list-of EXPR for ID in LIST if COND)
     (map (lambda (ID) EXPR)
          (filter (lambda (ID) COND) LIST))]
    [(list-of EXPR for ID in LIST)
     (list-of EXPR for ID in LIST if #t)]))

(list-of (* x x) for x in (range 10) if (odd? x))
```

Again, go over it and see how the binding structure makes the identifier available in both expressions. Note that since we're just playing around we're not paying too much attention to performance etc. (For example, if we cared, we could have implemented the `if`-less case by not using `filter` at all, or we could implement a `filter` that accepts `#t` as a predicate and in that case just returns the list, or even implementing it as a macro that identifies a `(lambda (_) #t)` pattern and expands to just the list (a bad idea in general).)

The last step: Python's comprehension accepts multiple `for-in`s for nested loops, possibly with `if` filters at each level:

```
(define-syntax list-of
  (syntax-rules (for in if)
    [(list-of EXPR for ID in LIST if COND)
     (map (lambda (ID) EXPR)
          (filter (lambda (ID) COND) LIST))]
    [(list-of EXPR for ID in LIST)
     (list-of EXPR for ID in LIST if #t)]
    [(list-of EXPR for ID in LIST for MORE ...)
     (list-of EXPR for ID in LIST if #t for MORE ...)]
    [(list-of EXPR for ID in LIST if COND for MORE ...)
     (apply append (map (lambda (ID) (list-of EXPR for MORE ...))
                        (filter (lambda (ID) COND) LIST)))]))
```

A collection of examples that I found in the Python docs and elsewhere, demonstrating all of these:

```
;; [x**2 for x in range(10)]
(list-of (* x x) for x in (range 10))
```

```
;; [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
(list-of (list x y) for x in '(1 2 3) for y in '(3 1 4)
                    if (not (= x y)))

(define (round-n x n) ; python-like round to n digits
  (define 10^n (expt 10 n))
  (/ (round (* x 10^n)) 10^n))
;; [str(round(pi, i)) for i in range(1, 6)]
(list-of (number->string (round-n pi i)) for i in (range 1 6))

(define matrix
  '((1 2 3 4)
    (5 6 7 8)
    (9 10 11 12)))
;; [[row[i] for row in matrix] for i in range(4)]
(list-of (list-of (list-ref row i) for row in matrix)
         for i in (range 4))

(define text '(("bar" "foo" "fooba")
               ("Rome" "Madrid" "Houston")
               ("aa" "bb" "cc" "dd")))
;; [y for x in text if len(x)>3 for y in x]
(list-of y for x in text if (> (length x) 3) for y in x)
;; [y for x in text for y in x if len(y)>4]
(list-of y for x in text for y in x if (> (string-length y) 4))
;; [y.upper() for x in text if len(x) == 3
;;             for y in x if y.startswith('f')]
(list-of (string-upcase y) for x in text if (= (length x) 3)
                           for y in x if (regexp-match? #rx"^f" y))
```

Problems of `syntax-rules` Macros

As we've seen, using `syntax-rules` solves many of the problems of macros, but it comes with a high price tag: the macros are "just" rewrite rules. As rewrite rules they're pretty sophisticated, but it still loses a huge advantage of what we had with `define-macro` — the macro code is no longer Racket code but a simple language of rewrite rules.

There are two big problems with this which we will look into now. (DrRacket's macro stepper tool can be very useful in clarifying these examples.) The first problem is that in some cases we want to perform computations at the macro level — for example, consider a `repeat` macro that needs to expand like this:

```
(repeat 1 E)  -->  (begin E)
(repeat 2 E)  -->  (begin E E)
(repeat 3 E)  -->  (begin E E E)
...
```

With a `syntax-rules` macro we can match over specific integers, but we just cannot do this with *any* integer. Note that this specific case can be done better via a function — better by not replicating the expression:

```
(define (repeat/proc n thunk)
  (when (> n 0) (thunk) (repeat/proc (sub1 n) thunk)))
(define-syntax-rule (repeat N E)
  (repeat/proc N (lambda () E)))
```

or even better, assuming the above `for` is already implemented:

```
(define-syntax-rule (repeat N E)
  (for i = 1 to N do E))
```

But still, we want to have the ability to do such computation. A similar, and perhaps better example, is better error reporting. For example, the above `for` implementation blindly expands its input, so:

```
> (for 1 = 1 to 3 do (printf "i = ~s\n" i))
lambda: not an identifier in: 1
```

we get a bad error message in terms of `lambda`, which is breaking abstraction (it comes from the expansion of `for`, which is an implementation detail), and worse — it is an error about something that the user didn't write.

Yet another aspect of this problem is that sometimes we need to get creative solutions where it would be very simple to write the corresponding Racket code. For example, consider the problem of writing a `rev-app` macro — (rev-app F E …) should evaluate to a function similar to (F E …), except that we want the evaluation to go from right to left instead of the usual left-to-right that Racket does. This code is obviously very broken:

```
(define-syntax-rule (rev-app F E ...)
  (let (reverse ([x E] ...))
    (F x ...)))
```

because it *generates* a malformed `let` form — there is no way for the macro expander to somehow know that the `reverse` should happen at the transformation level. In this case, we can actually solve this using a helper macro to do the reversing:

```
(define-syntax-rule (rev-app F E ...)
  (rev-app-helper F (E ...) ()))
(define-syntax rev-app-helper
  (syntax-rules ()
    ;; this rule does the reversing, collecting the reversed
    ;; sequence in the last part
    [(rev-app-helper F (E0 E ...) (E* ...))
     (rev-app-helper F (E ...) (E0 E* ...))]
    ;; and this rule fires up when we're done with the reversal
    [(rev-app-helper F () (E ...))
     (let ([x E] ...)
       (F x ...))]))
```

There are still problems with this — it complains about `x ...` because there is a single `x` there rather than a sequence of them; and even if it did somehow work, we also need the `x`s in that last line in the original order rather than the reversed one. So the solution is complicated by collecting new `x`s while reversing — and since we need them in both orders, we're going to collect both orders:

```
(define-syntax-rule (rev-app F E ...)
  (rev-app-helper F (E ...) () () ()))
(define-syntax rev-app-helper
  (syntax-rules ()
    ;; this rule does the reversing, collecting the reversed
    ;; sequence in the last part -- also make up new identifiers
    ;; and collect them in *both* directions (`X' is the straight
    ;; sequence of identifiers, `X*' is the reversed one, and `E*'
    ;; is the reversed expression sequence); note that each
    ;; iteration introduces a new identifier called `t'
    [(rev-app-helper F (E0 E ...) (X ...  ) (  X* ...) (   E* ...))
     (rev-app-helper F (   E ...) (X ... t) (t X* ...) (E0 E* ...))]
    ;; and this rule fires up when we're done with the reversal and
    ;; the generation
    [(rev-app-helper F () (x ...) (x* ...) (E* ...))
```

```
      (let ([x* E*] ...)
        (F x ...))]))

;; see that it works
(define (show x) (printf ">>> ~s\n" x) x)
(rev-app list (show 1) (show 2) (show 3))
```

So, this worked, but in this case the simplicity of the `syntax-rules` rewrite language worked against us, and made a very inconvenient solution. This could have been much easier if we could just write a "meta-level" reverse, and a use of `map` to generate the names.

… And all of that was just the first problem. The second one is even harder: `syntax-rules` is *designed* to avoid all name captures, but what if we *want* to break hygiene? There are some cases where you want a macro that "injects" a user-visible identifier into its result. The most common (and therefore the classic) example of this is an anaphoric `if` macro, that binds `it` to the result of the test (which can be any value, not just a boolean):

```
;; find the element of `l' that is immediately following `x'
;; (assumes that if `x' is found, it is not the last one)
(define (after x l)
  (let ([m (member x l)])
    (if m
      (second m)
      (error 'after "~s not found in ~s" x l))))
```

which we want to turn to:

```
;; find the element of `l' that is immediately following `x'
;; (assumes that if `x' is found, it is not the last one)
(define (after x l)
  (if (member x l)
    (second it)
    (error 'after "~s not found in ~s" x l)))
```

The obvious definition of `if-it' doesn't work:

```
(define-syntax-rule (if-it E1 E2 E3)
  (let ([it E1]) (if it E2 E3)))
```

The reason it doesn't work should be obvious now — it is *designed* to avoid the `it` that the macro introduced from interfering with the `it` that the user code uses.

Next, we'll see Racket's "low level" macro system, which can later be used to solve these problems.

Racket's "Low-Level" Macros

As we've previously seen, `syntax-rules` *creates* transformation functions — but there are other more direct ways to write these functions. These involve writing a function directly rather than creating one with `syntax-rules` — and because this is a more low-level approach than using `syntax-rules` to generate a transformer, it is called a "low level macro system". All Scheme implementations have such low-level systems, and these systems vary from one to the other. They all involve some particular type that is used as "syntax" — this type is always related to S-expressions, but it cannot be the simple `define-macro` tool that we've seen earlier if we want to avoid the problems of capturing identifiers.

Historical note: For a very long time the Scheme standard had avoided a concrete specification of this low-level system, leaving `syntax-rules` as the only way to write portable Scheme code. This had lead some people to explore more thoroughly the things that can be done with just `syntax-rules` rewrites, even beyond the examples we've seen. As it turns out, there's a lot that can be done with it — in fact, it is possible to write rewrite rules that **implement a lambda calculus**, making it possible to write things that look kind of like "real code". This is, however, awkward to say the least, and redundant with a macro

system that can use the full language for arbitrary computations. It has also became less popular recently, since R6RS dictates something that is known as a “syntax-case macro system” (not really a good name, since `syntax-case` is just a tool in this system).

Racket uses an extended version of this `syntax-case` system, which is what we will discuss now. In the Racket macro system, “syntax” is a new type, not just S-expressions as is the case with `define-macro`. The way to think about this type is as a wrapper around S-expressions, where the S-expression is the “raw” symbolic form of the syntax, and a bunch of “stuff” is added. Two important bits of this “stuff” are the source location information for the syntax, and its lexical scope. The source location is what you’d expect: the file name for the syntax (if it was read from a file), its position in the file, and its line and column numbers; this information is mostly useful for reporting errors. The lexical scope information is used in a somewhat peculiar way: there is no direct way to access it, since usually you don’t need to do so — instead, for the rare cases where you do need to manipulate it, you *copy* the lexical scope of an existing syntax to create another. This allows the macro interface to be usable without specification of a specific representation for the scope.

The main piece of functionality in this system is `syntax-case` (which has lead to its common name) — a form that is used to deconstruct the input via pattern-matching similar to `syntax-rules`. In fact, the syntax of `syntax-case` looks very similar to the syntax of `syntax-rules` — there are zero or more parenthesized keywords, and then clauses that begin with the same kind of patterns to match the syntax against. The first obvious difference is that the syntax to be matched is given explicitly:

```
(syntax-case <value-to-match> (<keywords>)
  [<pattern> <result>]
  ...)
```

A macro is written as a plain function, usually used as the value in a `define-syntax` form (but it could also be used in plain helper functions). For example, here’s how the `orelse` macro is written using this:

```
(define-syntax orelse
  (lambda (stx)
    (syntax-case stx ()
      [(orelse x y) ???])))
```

Racket’s `define-syntax` can also use the same syntactic sugar for functions as `define`:

```
(define-syntax (orelse stx)
  (syntax-case stx ()
    [(orelse x y) ???]))
```

The second significant change from `syntax-rules` is that the right-hand-side expressions in the branches are not patterns. Instead, they’re plain Racket expressions. In this example (as in most uses of `syntax-case`) the result of the `syntax-case` form is going to be the result of the macro, and therefore it should return a piece of syntax. So far, the only piece of syntax that we see in this code is just the input `stx` — and returning that will get the macro expander in an infinite loop (because we’re essentially making a transformer for `orelse` expressions that expands the syntax to itself).

To return a *new* piece of syntax, we need a way to write new syntax values. The common way to do this is using a new special form: `syntax`. This form is similar to `quote` — except that instead of an S-expression, it results in a syntax. For example, in this code:

```
(define-syntax (orelse stx)
  (printf "Expanding ~s\n" stx)
  (syntax-case stx ()
    [(orelse x y) (syntax (printf "Running an orelse\n"))]))
```

the first printout happens during macro expansion, and the second is part of the generated code. Like `quote`, `syntax` has a convenient abbreviation — “`#'`”:

```
(define-syntax (orelse stx)
  (printf "Expanding ~s\n" stx)
```

```
    (syntax-case stx ()
      [(orelse x y) #'(printf "Running an orelse\n")]))
```

Now the question is how we can get the actual macro working. The thing is that `syntax` is not completely quoting its contents as a syntax — there could be some meta identifiers that are bound as “pattern variables” in the `syntax-case` pattern that was matched for the current clause — in this case, we have `x` and `y` as such pattern variables. (Actually, `orelse` is a pattern variable too, but this doesn’t matter for our purpose.) Using these inside a `syntax` will have them replaced by the syntax that they matched against. The complete `orelse` definition is therefore very easy:

```
(define-syntax (orelse stx)
  (syntax-case stx ()
    [(orelse <expr1> <expr2>)
     #'(let ((temp <expr1>))
         (if temp temp <expr2>))]))
```

The same treatment of `...` holds here too — in the matching pattern they specify 0 or more occurrences of the preceding pattern, and in the output template they mean that the matching sequence is “spliced” in. Note that `syntax-rules` is now easy to define as a macro that expands to a function that uses `syntax-case` to do the actual rewrite work:

```
(define-syntax (syntax-rules stx)
  (syntax-case stx ()
    [(syntax-rules (keyword ...)
       [pattern template]
       ...)
     #'(lambda (stx)
         (syntax-case stx (keyword ...)
           [pattern #'template]
           ...))]))
```