PL: Lecture #14

Tuesday, October 22nd

Recursive Environments

> 👉 ***PLAI §11.5***

What we really need for recursion, is a special kind of an environment, one that can refer to itself. So instead of doing (note: `call`s removed for readability):

```
{with {fact {fun {n}
               {if {zero? n} 1 {* n {fact {- n 1}}}}}}
  {fact 5}}
```

which does not work for the usual reasons, we want to use some

```
{rec {fact {fun {n}
             {if {zero? n} 1 {* n {fact {- n 1}}}}}}
  {fact 5}}
```

that will do the necessary magic.

One way to achieve this is using the Y combinator as we have seen — a kind of a “constructor” for recursive functions. We can do that in a similar way to the `rewrite` rule that we have seen in Schlac — translate the above expression to:

```
{with {fact {make-rec
              {fun {fact}
                {fun {n}
                  {if {zero? n} 1 {* n {fact {- n 1}}}}}}}}
  {fact 5}}
```

or even:

```
{with {fact {{fun {f} {{fun {x} {f {x x}}} {fun {x} {f {x x}}}}}
             {fun {fact}
               {fun {n}
                 {if {zero? n} 1 {* n {fact {- n 1}}}}}}}}
  {fact 5}}
```

Now, we will see how it can be used in *our* code to implement a recursive environment.

If we look at what `with` does in

```
{with {fact {fun {n}
               {if {zero? n} 1 {* n {call fact {- n 1}}}}}}
  {call fact 5}}
```

then we can say that to evaluate this expression, we evaluate the body expression in an extended environment that contains `fact`, even if a bogus one that is good for `0` only — the new environment is created with something like this:

```
extend("fact", make-fact-closure(), env)
```

so we can take this whole thing as an operation over `env`

```
add-fact(env) := extend("fact", make-fact-closure(), env)
```

This gives us the first-level fact. But `fact` itself is still undefined in `env`, so it cannot call itself. We can try this:

```
add-fact(add-fact(env))
```

but that still doesn't work, and it will never work no matter how far we go:

```
add-fact(add-fact(add-fact(add-fact(add-fact(...env...)))))
```

What we really want is infinity: a place where add-fact works and the result is the same as what we've started with — we want to create a "magical" environment that makes this possible:

```
let magic-env = ???
such that:
   add-fact(magic-env) = magic-env
```

which basically gives us the illusion of being at the infinity point. This magic-env thing is exactly the *fixed-point* of the `add-fact` operation. We can use:

```
magic-env = rec(add-fact)
```

and following the main property of the Y combinator, we know that:

```
magic-env = rec(add-fact)           ; def. of magic-env
          = add-fact(rec(add-fact)) ; Y(f) = f(Y(f))
          = add-fact(magic-env)     ; def. of magic-env
```

What does all this mean? It means that if we have a fixed-point operator at the level of the implementation of our environments, then we can use it to implement a recursive binder. In our case, this means that a fixpoint in Racket can be used to implement a recursive language. But we have that — Racket does have recursive functions, so we should be able to use that to implement our recursive binder.

There are two ways that make it possible to write recursive functions in Racket. One is to define a function, and use its name to do a recursive call — using the Racket formal rules, we can see that we said that we mark that we now *know* that a variable is bound to a value. This is essentially a side-effect — we modify what we know, which corresponds to modifying the global environment. The second way is a new form: `letrec`. This form is similar to `let`, except that the scope that is established includes the named expressions — it is exactly what we want `rec` to do. A third way is using recursive local definitions, but that is equivalent to using `letrec`, more on this soon.

Recursion: Racket's `letrec`

So we want to add recursion to our language, practically. We already know that Racket makes it possible to write recursive functions, which is possible because of the way it implements its "global environment": our evaluator can only *extend* an environment, while Racket *modifies* its global environment. This means that whenever a function is defined in the global environment, the resulting closure will have it as its environment "pointer", but the global environment was not extended — it stays the same, and was just modified with one additional binding.

But Racket has another, a bit more organized way of using recursion: there is a special local-binding construct that is similar to `let`, but allows a function to refer to itself. It is called `letrec`:

```
(letrec ([fact (lambda (n)
                 (if (zero? n)
                   1
                   (* n (fact (- n 1)))))])
  (fact 5))
```

Some people may remember that there was a third way for creating recursive functions: using local definition in function bodies. For example, we have seen things like:

```
(define (length list)
  (define (helper list len)
    (if (null? list)
      len
      (helper (rest list) (+ len 1))))
  (helper list 0))
```

This looks like the same kind of environment magic that happens with a global `define` — but actually, Racket defines the meaning of internal definitions using `letrec` — so the above code is exactly the same as:

```
(define (length list)
  (letrec ([helper (lambda (list len)
                     (if (null? list)
                       len
                       (helper (rest list) (+ len 1))))])
    (helper list 0)))
```

The scoping rules for a `letrec` is that the scope of the bound name covers both the body *and* the named expression. Furthermore, multiple names can be bound to multiple expressions, and the scope of each name covers all named expression as well as the body. This makes it easy to define mutually recursive functions, such as:

```
(letrec ([even? (lambda (n) (if (zero? n) #t (odd?  (- n 1))))]
         [odd?  (lambda (n) (if (zero? n) #f (even? (- n 1))))])
  (even? 99))
```

But it is not a required functionality — it could be done with a single recursive binding that contains several functions:

```
(letrec ([even+odd
          (list (lambda (n)
                  (if (zero? n) #t ((second even+odd) (- n 1))))
                (lambda (n)
                  (if (zero? n) #f ((first  even+odd) (- n 1)))))])
  ((first even+odd) 99))
```

This is basically the same problem we face if we want to use the Y combinator for mutually recursive bindings. The above solution is inconvenient, but it can be improved using more `let`s to have easier name access. For example:

```
(letrec ([even+odd
          (list (lambda (n)
                  (let ([even? (first  even+odd)]
                        [odd?  (second even+odd)])
                    (if (zero? n) #t (odd? (- n 1)))))
                (lambda (n)
                  (let ([even? (first  even+odd)]
                        [odd?  (second even+odd)])
                    (if (zero? n) #f (even? (- n 1))))))])
  (let ([even? (first  even+odd)]
        [odd?  (second even+odd)])
    (even? 99)))
```

Implementing Recursion using `letrec`

We will see how to add a similar construct to our language — for simplicity, we will add a `rec` form that handles a single binding:

```
{rec {fact {fun {n}
             {if {= 0 n}
               1
               {* n {fact {- n 1}}}}}}
  {fact 5}}
```

Using this, things can get a little tricky. What should we get if we do:

```
{rec {x x} x}
```

? Currently, it seems like there is no point in using any expression except for a *function* expression in a `rec` expression, so we will handle only these cases.

(BTW, under what circumstances would non-function values be useful in a letrec?)

One way to achieve this is to use the same trick that we have recently seen: instead of re-implementing language features, we can use existing features in our own language, which hopefully has the right functionality in a form that can be re-used to in our evaluator.

Previously, we have seen a way to implement environments using Racket closures:

```
;; Define a type for functional environments
(define-type ENV = Symbol -> VAL)

(: EmptyEnv : -> ENV)
(define (EmptyEnv)
  (lambda (id) (error 'lookup "no binding for ~s" id)))

(: lookup : Symbol ENV -> VAL)
(define (lookup name env)
  (env name))

(: Extend : Symbol VAL ENV -> ENV)
(define (Extend id val rest-env)
  (lambda (name)
    (if (eq? name id)
      val
      (rest-env name))))
```

We can use this implementation, and create circular environments using Racket's `letrec`. The code for handling a `with` expressions is:

```
[(With bound-id named-expr bound-body)
 (eval bound-body
       (Extend bound-id (eval named-expr env) env))]
```

It looks like we should be able to handle `rec` in a similar way (the AST constructor name is `WRec` ("with-rec") so it doesn't collide with TR's `Rec` constructor for recursive types):

```
[(WRec bound-id named-expr bound-body)
 (eval bound-body
       (Extend bound-id (eval named-expr env) env))]
```

but this won't work because the named expression is evaluated prematurely, in the previous environment. Instead, we will move everything that needs to be done, including evaluation, to a separate `extend-rec` function:

```
[(WRec bound-id named-expr bound-body)
 (eval bound-body
       (extend-rec bound-id named-expr env))]
```

Now, the `extend-rec` function needs to provide the new, "magically circular" environment. Following what we know about the arguments to `extend-rec`, and the fact that it returns a new environment (= a lookup function), we can sketch a rough definition:

```
(: extend-rec : Symbol FLANG ENV -> ENV) ; FLANG, not VAL!
;; extend an environment with a new binding that is the result of
;; evaluating an expression in the same environment as the extended
;; result
(define (extend-rec id expr rest-env)
  (lambda (name)
    (if (eq? name id)
      ... something that uses expr to get a value ...
      (rest-env name))))
```

What should the missing expression be? It can simply evaluate the object given itself:

```
(define (extend-rec id expr rest-env)
  (lambda (name)
    (if (eq? name id)
      (eval expr ...this environment...)
      (rest-env name))))
```

But how do we get *this environment*, before it is defined? Well, the environment is itself a Racket *function*, so we can use Racket's `letrec` to make the function refer to itself recursively:

```
(define (extend-rec id expr rest-env)
  (letrec ([rec-env (lambda (name)
                      (if (eq? name id)
                        (eval expr rec-env)
                        (rest-env name)))])
    rec-env))
```

It's a little more convenient to use an internal definition, and add a type for clarity:

```
(define (extend-rec id expr rest-env)
  (: rec-env : Symbol -> VAL)
  (define (rec-env name)
    (if (eq? name id)
      (eval expr rec-env)
      (rest-env name)))
  rec-env)
```

This works, but there are several problems:

1. First, we no longer do a simple lookup in the new environment. Instead, we evaluate the expression on *every* such lookup. This seems like a technical point, because we do not have side-effects in our language (also because we said that we want to handle only function expressions). Still, it wastes space since each evaluation will allocate a new closure.
2. Second, a related problem — what happens if we try to run this:

```
{rec {x x} x}
```

? Well, we do that stuff to extend the current environment, then evaluate the body in the new environment, this body is a single variable reference:

```
(eval (Id 'x) the-new-env)
```

so we look up the value:

```
(lookup 'x the-new-env)
```

which is:

```
(the-new-env 'x)
```

which goes into the function which implements this environment, there we see that `name` is the same as `name1`, so we return:

```
(eval expr rec-env)
```

but the `expr` here is the original named-expression which is itself `(Id 'x)`, and we're in an infinite loop.

We can try to get over these problems using another binding. Racket allows several bindings in a single `letrec` expression or multiple internal function definitions, so we change `extend-rec` to use the newly-created environment:

```
(define (extend-rec id expr rest-env)
  (: rec-env : Symbol -> VAL)
  (define (rec-env name)
    (if (eq? name id)
      val
      (rest-env name)))
  (: val : VAL)
  (define val (eval expr rec-env))
  rec-env)
```

This runs into an interesting type error, which complains about possibly getting some `Undefined` value. It does work if we switch to the untyped language for now (using `#lang pl untyped`) — and it seems to run fine too. But it raises more questions, beginning with: what is the meaning of:

```
(letrec ([x ...]
         [y ...x...])
  ...)
```

or equivalently, an internal block of

```
(define x ...)
(define y ...x...)
```

? Well, DrRacket seems to do the "right thing" in this case, but what about:

```
(letrec ([y ...x...]
         [x ...])
  ...)
```

? As a hint, see what happens when we now try to evaluate the problematic

```
{rec {x x} x}
```

expression, and compare that with the result that you'd get from Racket. This also clarifies the type error that we received.

It should be clear now why we want to restrict usage to just binding recursive functions. There are no problems with such definitions because when we evaluate a `fun` expression, there is no evaluation of the

body, which is the only place where there are potential references to the same function that is defined — a function's body is *delayed*, and executed only when the function is applied later.

But the biggest question that is still open: we just implemented a circular environment using Racket's own circular environment implementation, and that does not explain how they are actually implemented. The cycle of pointers that we've implemented depends on the cycle of pointers that Racket uses, and that is a black box we want to open up.

For reference, the complete code is below.

```
#lang pl

#|
The grammar:
  <FLANG> ::= <num>
            | { + <FLANG> <FLANG> }
            | { - <FLANG> <FLANG> }
            | { * <FLANG> <FLANG> }
            | { / <FLANG> <FLANG> }
            | { with { <id> <FLANG> } <FLANG> }
            | { rec { <id> <FLANG> } <FLANG> }
            | <id>
            | { fun { <id> } <FLANG> }
            | { call <FLANG> <FLANG> }

Evaluation rules:
  eval(N,env)                = N
  eval({+ E1 E2},env)        = eval(E1,env) + eval(E2,env)
  eval({- E1 E2},env)        = eval(E1,env) - eval(E2,env)
  eval({* E1 E2},env)        = eval(E1,env) * eval(E2,env)
  eval({/ E1 E2},env)        = eval(E1,env) / eval(E2,env)
  eval(x,env)                = lookup(x,env)
  eval({with {x E1} E2},env) = eval(E2,extend(x,eval(E1,env),env))
  eval({rec {x E1} E2},env)  = ???
  eval({fun {x} E},env)      = <{fun {x} E}, env>
  eval({call E1 E2},env1)    = eval(B,extend(x,eval(E2,env1),env2))
                               if eval(E1,env1) = <{fun {x} B}, env2>
                             = error!  otherwise
|#

(define-type FLANG
  [Num  Number]
  [Add  FLANG FLANG]
  [Sub  FLANG FLANG]
  [Mul  FLANG FLANG]
  [Div  FLANG FLANG]
  [Id   Symbol]
  [With Symbol FLANG FLANG]
  [WRec Symbol FLANG FLANG]
  [Fun  Symbol FLANG]
  [Call FLANG FLANG])

(: parse-sexpr : Sexpr -> FLANG)
;; parses s-expressions into FLANGs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
```

```
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        (With name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(cons 'rec more)
     (match sexpr
       [(list 'rec (list (symbol: name) named) body)
        (WRec name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `rec' syntax in ~s" sexpr)])]
    [(cons 'fun more)
     (match sexpr
       [(list 'fun (list (symbol: name)) body)
        (Fun name (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list 'call fun arg)
                       (Call (parse-sexpr fun) (parse-sexpr arg))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(: parse : String -> FLANG)
;; parses a string containing a FLANG expression to a FLANG AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))

;; Types for environments, values, and a lookup function

(define-type VAL
  [NumV Number]
  [FunV Symbol FLANG ENV])

;; Define a type for functional environments
(define-type ENV = Symbol -> VAL)

(: EmptyEnv : -> ENV)
(define (EmptyEnv)
  (lambda (id) (error 'lookup "no binding for ~s" id)))

(: lookup : Symbol ENV -> VAL)
;; lookup a symbol in an environment, return its value or throw an
;; error if it isn't bound
(define (lookup name env)
  (env name))

(: Extend : Symbol VAL ENV -> ENV)
;; extend a given environment cache with a new binding
(define (Extend id val rest-env)
  (lambda (name)
    (if (eq? name id)
      val
      (rest-env name))))

(: extend-rec : Symbol FLANG ENV -> ENV)
;; extend an environment with a new binding that is the result of
;; evaluating an expression in the same environment as the extended
;; result
```

```
(define (extend-rec id expr rest-env)
  (: rec-env : Symbol -> VAL)
  (define (rec-env name)
    (if (eq? name id)
      val
      (rest-env name)))
  (: val : VAL)
  (define val (eval expr rec-env))
  rec-env)

(: NumV->number : VAL -> Number)
;; convert a FLANG runtime numeric value to a Racket one
(define (NumV->number val)
  (cases val
    [(NumV n) n]
    [else (error 'arith-op "expected a number, got: ~s" val)]))

(: arith-op : (Number Number -> Number) VAL VAL -> VAL)
;; gets a Racket numeric binary operator, and uses it within a NumV
;; wrapper
(define (arith-op op val1 val2)
  (NumV (op (NumV->number val1) (NumV->number val2))))

(: eval : FLANG ENV -> VAL)
;; evaluates FLANG expressions by reducing them to values
(define (eval expr env)
  (cases expr
    [(Num n) (NumV n)]
    [(Add l r) (arith-op + (eval l env) (eval r env))]
    [(Sub l r) (arith-op - (eval l env) (eval r env))]
    [(Mul l r) (arith-op * (eval l env) (eval r env))]
    [(Div l r) (arith-op / (eval l env) (eval r env))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (Extend bound-id (eval named-expr env) env))]
    [(WRec bound-id named-expr bound-body)
     (eval bound-body
           (extend-rec bound-id named-expr env))]
    [(Id name) (lookup name env)]
    [(Fun bound-id bound-body)
     (FunV bound-id bound-body env)]
    [(Call fun-expr arg-expr)
     (define fval (eval fun-expr env))
     (cases fval
       [(FunV bound-id bound-body f-env)
        (eval bound-body
              (Extend bound-id (eval arg-expr env) f-env))]
       [else (error 'eval "`call' expects a function, got: ~s"
                    fval)])]))

(: run : String -> Number)
;; evaluate a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str) (EmptyEnv))])
    (cases result
      [(NumV n) n]
      [else (error 'run "evaluation returned a non-number: ~s"
                   result)])))
```

```
;; tests
(test (run "{call {fun {x} {+ x 1}} 4}")
      => 5)
(test (run "{with {add3 {fun {x} {+ x 3}}}
              {call add3 1}}")
      => 4)
(test (run "{with {add3 {fun {x} {+ x 3}}}
              {with {add1 {fun {x} {+ x 1}}}
                {with {x 3}
                  {call add1 {call add3 x}}}}}")
      => 7)
(test (run "{with {identity {fun {x} x}}
              {with {foo {fun {x} {+ x 1}}}
                {call {call identity foo} 123}}}")
      => 124)
(test (run "{with {x 3}
              {with {f {fun {y} {+ x y}}}
                {with {x 5}
                  {call f 4}}}}")
      => 7)
(test (run "{call {with {x 3}
                    {fun {y} {+ x y}}}
                  4}")
      => 7)
(test (run "{with {f {with {x 3} {fun {y} {+ x y}}}}
              {with {x 100}
                {call f 4}}}")
      => 7)
(test (run "{call {call {fun {x} {call x 1}}
                        {fun {x} {fun {y} {+ x y}}}}
                  123}")
      => 124)
```