PL: Lecture #16

Tuesday, October 29th

Variable Mutation

> ☞ ***PLAI §12*** *and* ☞ ***PLAI §13*** *(different: adds boxes to the language)*
> ☞ ***PLAI §14*** *(that's what we do)*

The code that we now have implements recursion by *changing* bindings, and to make that possible we made environments hold boxes for all bindings, therefore bindings are *all* mutable now. We can use this to add more functionality to our evaluator, by allowing changing any variable — we can add a `set!` form:

```
{set! <id> <FLANG>}
```

to the evaluator that will modify the value of a variable. To implement this functionality, all we need to do is to use `lookup` to retrieve some box, then evaluate the expression and put the result in that box. The actual implementation is left as a homework exercise.

One thing that should be considered here is — all of the expressions in our language evaluate to some value, the question is what should be the value of a `set!` expression? There are three obvious choices:

1. return some bogus value,
2. return the value that was assigned,
3. return the value that was previously in the box.

Each one of these has its own advantage — for example, C uses the second option to `chain` assignments (eg, `x = y = 0`) and to allow side effects where an expression is expected (eg, `while (x = x-1) ...`).

The third one is useful in cases where you might use the old value that is overwritten — for example, if C had this behavior, you could `pop` a value from a linked list using something like:

```
first(stack = rest(stack));
```

because the argument to `first` will be the old value of `stack`, before it changed to be its `rest`. You could also swap two variables in a single expression: `x = y = x`.

(Note that the expression `x = x + 1` has the meaning of C's `++x` when option (2) is used, and `x++` when option (3) is used.)

Racket chooses the first option, and we will do the same in our language. The advantage here is that you get no discounts, therefore you must be explicit about what values you want to return in situations where there is no obvious choice. This leads to more robust programs since you do not get other programmers that will rely on a feature of your code that you did not plan on.

In any case, the modification that introduces mutation is small, but it has a tremendous effect on our language: it was true for Racket, and it is true for FLANG. We have seen how mutation affects the language subset that we use, and in the extension of our FLANG the effect is even stronger: since *any* variable can change (no need for explicit `box`es). In other words, a binding is not always the same — in can change as a result of a `set!` expression. Of course, we could extend our language with boxes (using Racket boxes to implement FLANG boxes), but that will be a little more verbose.

> *Note that Racket does have a `set!` form, and in addition, fields in structs can be made modifiable. However, we do not use any of these. At least not for now.*

State and Environments

A quick example of how mutation can be used:

```
(define counter
  (let ([counter (box 0)])
    (lambda ()
      (set-box! counter (+ 1 (unbox counter)))
      (unbox counter))))
```

and compare that to:

```
(define (make-counter)
  (let ([counter (box 0)])
    (lambda ()
      (set-box! counter (+ 1 (unbox counter)))
      (unbox counter))))
```

It is a good idea if you follow the exact evaluation of

```
(define foo (make-counter))
(define bar (make-counter))
```

and see how both bindings have separate environment so each one gets its own private state. The equivalent code in the homework interpreter extended with `set!` doesn't need boxes:

```
{with {make-counter
        {fun {}
          {with {counter 0}
            {fun {}
              {set! counter {+ counter 1}}
              counter}}}}
  {with {foo {call make-counter}}
    {with {bar {call make-counter}}
      ...}}}
```

To see multiple values from a single expression you can extend the language with a `list` binding. As a temporary hack, we can use dummy function inputs to cover for our lack of nullary functions, and use `with` (with dummy bound ids) to sequence multiple expressions:

```
{with {make-counter
        {fun {init}
          {with {counter init}
            {fun {_}
              {with {_ {set! counter {+ counter 1}}}
                counter}}}}}
  {with {foo {call make-counter 0}}
    {with {bar {call make-counter 1}}
      {+ {+ {call foo 0} {+ {* 10 {call foo 0}}
                            {* 100 {call foo 0}}}}
         {* 10000 {+ {call bar 0} {+ {* 10 {call bar 0}}
                                     {* 100 {call bar 0}}}}}}}}}
```

Note that we cannot describe this behavior with substitution rules! We now use the environments to make it possible to change bindings — so finally an environment is actually an environment rather than a substitution cache.

When you look at the above, note that we still use lexical scope — in fact, the local binding is actually a private state that nobody can access. For example, if we write this:

```
(define counter
  (let ([counter (box 0)])
    (lambda ()
      (set-box! counter (+ 1 (unbox counter)))
      (if (zero? (modulo (unbox counter) 4)) 'tock 'tick))))
```

then the resulting function that us bound to `counter` keeps a local integer state which no other code can access — you cannot modify it, reset it, or even know if it is really an integer that is used in there.

Implementing Objects with State

We have already seen how several pieces of information can be encapsulate in a Racket closure that keeps them all; now we can do a little more — we can actually have mutable state, which leads to a natural way to implement objects. For example:

```
(define (make-point x y)
  (let ([xb (box x)]
        [yb (box y)])
    (lambda (msg)
      (match msg
        ['getx (unbox xb)]
        ['gety (unbox yb)]
        ['incx (set-box! xb (add1 (unbox xb)))]))))
```

implements a constructor for `point` objects which keep two values and can move one of them. Note that the messages act as a form of methods, and that the values themselves are hidden and are accessible only through the interface that these messages make. For example, if these points correspond to some graphic object on the screen, we can easily incorporate a necessary screen update:

```
(define (make-point x y)
  (let ([xb (box x)]
        [yb (box y)])
    (lambda (msg)
      (match msg
        ['getx (unbox xb)]
        ['gety (unbox yb)]
        ['incx (set-box! xb (add1 (unbox xb)))
               (update-screen)]))))
```

and be sure that this is always done when the value changes — since there is no way to change the value except through this interface.

A more complete example would define functions that actually send these messages — here is a better implementation of a point object and the corresponding accessors and mutators:

```
(define (make-point x y)
  (let ([xb (box x)]
        [yb (box y)])
    (lambda (msg)
      (match msg
        ['getx (unbox xb)]
        ['gety (unbox yb)]
        [(list 'setx newx)
         (set-box! xb newx)
         (update-screen)]
        [(list 'sety newy)
         (set-box! yb newy)
         (update-screen)]))))
```

```
(define (point-x p) (p 'getx))
(define (point-y p) (p 'gety))
(define (set-point-x! p x) (p (list 'setx x)))
(define (set-point-y! p y) (p (list 'sety y)))
```

And a quick imitation of inheritance can be achieved using delegation to an instance of the super-class:

```
(define (make-colored-point x y color)
  (let ([p (make-point x y)])
    (lambda (msg)
      (match msg
        ['getcolor color]
        [else (p msg)]))))
```

You can see how all of these could come from some preprocessing of a more normal-looking class definition form, like:

```
(defclass point (x y)
  (public (getx) x)
  (public (gety) y)
  (public (setx new) (set! x newx))
  (public (setx new) (set! x newx)))

(defclass colored-point point (c)
  (public (getcolor) c))
```

The Toy Language

> *Not in PLAI*

A quick note: from now on, we will work with a variation of our language — it will change the syntax to look a little more like Racket, and we will use Racket values for values in our language and Racket functions for built-ins in our language.

Main highlights:

- There can be multiple bindings in function arguments and local `bind` forms — the names are required to be distinct.
- There are now a few keywords like `bind` that are parsed in a special way. Other forms are taken as function application, which means that there are no special parse rules (and AST entries) for arithmetic functions. They're now bindings in the global environment, and treated in the same way as all bindings. For example, `*` is an expression that evaluates to the *primitive* multiplication function, and `{bind {{+ *}} {+ 2 3}}` evaluates to `6`.
- Since function applications are now the same for primitive functions and user-bound functions, there is no need for a `call` keyword. Note that the function call part of the parser must be last, since it should apply only if the input is not some other known form.
- Note the use of `make-untyped-list-function`: it's a library function (included in the course language) that can convert a few known Racket functions to a function that consumes a list of *any* Racket values, and returns the result of applying the given Racket function on these values. For example:

  ```
  (define add (make-untyped-list-function +))
  (add (list 1 2 3 4))
  ```

  evaluates to `10`.

- Another important aspect of this is its type — the type of `add` in the previous example is `(List -> Any)`, so the resulting function can consume *any* input values. If it gets a bad value, it will throw an appropriate error. This is a hack: it basically means that the resulting `add` function has a very generic type (requiring just a list), so errors can be thrown at run-time. However, in this case, a better solution is not going to make these run-time errors go away because the language that we're implementing is not statically typed.
- The benefit of this is that we can avoid the hassle of more verbose code by letting these functions dynamically check the input values, so we can use a single `RktV` variant in `VAL` which wraps any Racket value. (Otherwise we'd need different wrappers for different types, and implement these dynamic checks.)

The following is the complete implementation.

```
#lang pl

;;; ----------------------------------------------------------------
;;; Syntax

#| The BNF:
   <TOY> ::= <num>
           | <id>
           | { bind {{ <id> <TOY> } ... } <TOY> }
           | { fun { <id> ... } <TOY> }
           | { if <TOY> <TOY> <TOY> }
           | { <TOY> <TOY> ... }
|#

;; A matching abstract syntax tree datatype:
(define-type TOY
  [Num  Number]
  [Id   Symbol]
  [Bind (Listof Symbol) (Listof TOY) TOY]
  [Fun  (Listof Symbol) TOY]
  [Call TOY (Listof TOY)]
  [If   TOY TOY TOY])

(: unique-list? : (Listof Any) -> Boolean)
;; Tests whether a list is unique, guards Bind and Fun values.
(define (unique-list? xs)
  (or (null? xs)
      (and (not (member (first xs) (rest xs)))
           (unique-list? (rest xs)))))

(: parse-sexpr : Sexpr -> TOY)
;; parses s-expressions into TOYs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'bind more)
     (match sexpr
       [(list 'bind (list (list (symbol: names) (sexpr: nameds))
                          ...)
          body)
        (if (unique-list? names)
          (Bind names (map parse-sexpr nameds) (parse-sexpr body))
          (error 'parse-sexpr "duplicate `bind' names: ~s" names))]
       [else (error 'parse-sexpr "bad `bind' syntax in ~s" sexpr)])]
    [(cons 'fun more)
     (match sexpr
       [(list 'fun (list (symbol: names) ...) body)
        (if (unique-list? names)
          (Fun names (parse-sexpr body))
          (error 'parse-sexpr "duplicate `fun' names: ~s" names))]
       [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(cons 'if more)
     (match sexpr
       [(list 'if cond then else)
        (If (parse-sexpr cond)
            (parse-sexpr then)
            (parse-sexpr else))]
       [else (error 'parse-sexpr "bad `if' syntax in ~s" sexpr)])]
    [(list fun args ...) ; other lists are applications
     (Call (parse-sexpr fun)
           (map parse-sexpr args))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(: parse : String -> TOY)
;; Parses a string containing an TOY expression to a TOY AST.
(define (parse str)
  (parse-sexpr (string->sexpr str)))

;;; ------------------------------------------------------------------
;;; Values and environments

(define-type ENV
  [EmptyEnv]
  [FrameEnv FRAME ENV])

;; a frame is an association list of names and values.
(define-type FRAME = (Listof (List Symbol VAL)))

(define-type VAL
  [RktV  Any]
  [FunV  (Listof Symbol) TOY ENV]
  [PrimV ((Listof VAL) -> VAL)])

(: extend : (Listof Symbol) (Listof VAL) ENV -> ENV)
;; extends an environment with a new frame.
(define (extend names values env)
  (if (= (length names) (length values))
    (FrameEnv (map (lambda ([name : Symbol] [val : VAL])
                     (list name val))
                   names values)
              env)
    (error 'extend "arity mismatch for names: ~s" names)))

(: lookup : Symbol ENV -> VAL)
;; lookup a symbol in an environment, frame by frame,
;; return its value or throw an error if it isn't bound
(define (lookup name env)
  (cases env
    [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
    [(FrameEnv frame rest)
     (let ([cell (assq name frame)])
       (if cell
          (second cell)
          (lookup name rest))]]))

(: unwrap-rktv : VAL -> Any)
;; helper for `racket-func->prim-val': unwrap a RktV wrapper in
;; preparation to be sent to the primitive function
(define (unwrap-rktv x)
  (cases x
    [(RktV v) v]
    [else (error 'racket-func "bad input: ~s" x)]))

(: racket-func->prim-val : Function -> VAL)
;; converts a racket function to a primitive evaluator function
;; which is a PrimV holding a ((Listof VAL) -> VAL) function.
;; (the resulting function will use the list function as is,
;; and it is the list function's responsibility to throw an error
;; if it's given a bad number of arguments or bad input types.)
(define (racket-func->prim-val racket-func)
  (define list-func (make-untyped-list-function racket-func))
  (PrimV (lambda (args)
           (RktV (list-func (map unwrap-rktv args))))))

;; The global environment has a few primitives:
(: global-environment : ENV)
(define global-environment
  (FrameEnv (list (list '+ (racket-func->prim-val +))
                  (list '- (racket-func->prim-val -))
                  (list '* (racket-func->prim-val *))
                  (list '/ (racket-func->prim-val /))
                  (list '< (racket-func->prim-val <))
                  (list '> (racket-func->prim-val >))
                  (list '= (racket-func->prim-val =))
                  ;; values
                  (list 'true  (RktV #t))
                  (list 'false (RktV #f)))
            (EmptyEnv)))

;;; ------------------------------------------------------------------
;;; Evaluation

(: eval : TOY ENV -> VAL)
;; evaluates TOY expressions
(define (eval expr env)
  ;; convenient helper
  (: eval* : TOY -> VAL)
  (define (eval* expr) (eval expr env))
  (cases expr
    [(Num n)   (RktV n)]
    [(Id name) (lookup name env)]
    [(Bind names exprs bound-body)
     (eval bound-body (extend names (map eval* exprs) env))]
    [(Fun names bound-body)
     (FunV names bound-body env)]
    [(Call fun-expr arg-exprs)
     (define fval (eval* fun-expr))
     (define arg-vals (map eval* arg-exprs))
     (cases fval
       [(PrimV proc) (proc arg-vals)]
       [(FunV names body fun-env)
        (eval body (extend names arg-vals fun-env))]
       [else (error 'eval "function call with a non-function: ~s"
                    fval)])]
    [(If cond-expr then-expr else-expr)
     (eval* (if (cases (eval* cond-expr)
                  [(RktV v) v] ; Racket value => use as boolean
                  [else #t])   ; other values are always true
              then-expr
              else-expr))]))

(: run : String -> Any)
;; evaluate a TOY program contained in a string
(define (run str)
  (let ([result (eval (parse str) global-environment)])
    (cases result
      [(RktV v) v]
      [else (error 'run "evaluation returned a bad value: ~s"
                   result)])))

;;; ----------------------------------------------------------------
;;; Tests

(test (run "{{fun {x} {+ x 1}} 4}")
      => 5)
(test (run "{bind {{add3 {fun {x} {+ x 3}}}} {add3 1}}")
      => 4)
(test (run "{bind {{add3 {fun {x} {+ x 3}}}
                   {add1 {fun {x} {+ x 1}}}}
              {bind {{x 3}} {add1 {add3 x}}}}")
      => 7)
(test (run "{bind {{identity {fun {x} x}}
                   {foo {fun {x} {+ x 1}}}}
              {{identity foo} 123}}")
      => 124)
(test (run "{bind {{x 3}}
              {bind {{f {fun {y} {+ x y}}}}
                {bind {{x 5}}
                  {f 4}}}}")
      => 7)
(test (run "{{{fun {x} {x 1}}
              {fun {x} {fun {y} {+ x y}}}}
             123}")
      => 124)

;; More tests for complete coverage
(test (run "{bind x 5 x}")      =error> "bad `bind' syntax")
(test (run "{fun x x}")         =error> "bad `fun' syntax")
(test (run "{if x}")            =error> "bad `if' syntax")
(test (run "{}")                =error> "bad syntax")
(test (run "{bind {{x 5} {x 5}} x}") =error> "duplicate*bind*names")
(test (run "{fun {x x} x}")     =error> "duplicate*fun*names")
(test (run "{+ x 1}")           =error> "no binding for")
(test (run "{+ 1 {fun {x} x}}") =error> "bad input")
(test (run "{+ 1 {fun {x} x}}") =error> "bad input")
(test (run "{1 2}")             =error> "with a non-function")
(test (run "{{fun {x} x}}")     =error> "arity mismatch")
(test (run "{if {< 4 5} 6 7}")  => 6)
```

```
(test (run "{if {< 5 4} 6 7}")  => 7)
(test (run "{if + 6 7}")        => 6)
(test (run "{fun {x} x}")       =error> "returned a bad value")

;;; ----------------------------------------------------------------
```