Implementing Lexical Scope using Racket Closures and Environments

[[[PLAI Chapter 11 (without the last part about recursion)]]]

An alternative representation for an environment.

We've already seen how first-class functions can be used to implement "objects" that contain some information. We can use the same idea to represent an environment. The basic intuition is -- an environment is a
mapping (a function) between an identifier and some value. For example, we can represent the environment that maps 'a to 1 and 'b to 2 (using just numbers for simplicity) using this function:

```
(: my-map : Symbol -> Number)
(define (my-map id)
  (cond [(eq? 'a id) 1]
        [(eq? 'b id) 2]
        [else (error ...)]))
```

An empty mapping that is implemented in this way has the same type:

```
(: empty-mapping : Symbol -> Number)
(define (empty-mapping id)
  (error ...))
```

We can use this idea to implement our environments: we only need to define three things -- `EmptyEnv', `Extend', and `lookup'. If we manage
to keep the contract to these functions intact, we will be able to simply plug it into the same evaluator code with no other changes. It will also be more convenient to define `ENV' as the appropriate function
type for use in the VAL type definition instead of using the actual type:

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

Now we get to `EmptyEnv' -- this is expected to be a function that expects no arguments and creates an empty environment, one that behaves like the `empty-mapping' function defined above. We could define it like this (changing the `empty-mapping' type to return a VAL):

```
(define (EmptyEnv) empty-mapping)
```

but we can skip the need for an extra definition and simply return an empty mapping function:

```
(: EmptyEnv : -> ENV)
(define (EmptyEnv)
  (lambda (id) (error ...)))
```

(The un-Rackety name is to avoid replacing previous code that used the `EmptyEnv' name for the constructor that was created by the type definition.)

The next thing we tackle is `lookup'. The previous definition that was used is:

```
(: lookup : Symbol ENV -> VAL)
(define (lookup name env)
  (cases env
    [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
    [(Extend id val rest-env)
     (if (eq? id name) val (lookup name rest-env))]))
```

How should it be modified now? Easy -- an environment is a mapping: a Racket function that will do the searching job itself. We don't need to
modify the contract since we're still using `ENV', except a different implementation for it. The new definition is:

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

Note that `lookup' does almost nothing -- it simply delegates the real work to the `env' argument. This is a good hint for the error message that empty mappings should throw --

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

Finally, `Extend' -- this was previously created by the variant case of the ENV type definition:

```
[Extend (id Symbol) (v VAL) (rest-env ENV)]
```

keeping the same type that is implied by this variant means that the new
`Extend' should look like this:

```
(: Extend : Symbol VAL ENV -> ENV)
(define (Extend id v rest-env)
  ...)
```

The question is -- how do we extend a given environment? Well, first, we know that the result should be mapping -- a `symbol -> VAL' function that expects an identifier to look for:

```
(: Extend : Symbol VAL ENV -> ENV)
(define (Extend id v rest-env)
  (lambda (name)
    ...))
```

Next, we know that in the generated mapping, if we look for `id' then the result should be `v':

```
(: Extend : Symbol VAL ENV -> ENV)
(define (Extend id v rest-env)
  (lambda (name)
```

```
      (if (eq? name id)
        v
        ...)))
```

If the `name' that we're looking for is not the same as `id', then we need to search through the previous environment, eg: (lookup name rest).
But we know what `lookup' does -- it simply delegates back to the mapping function (which is our `rest' argument), so we can take a direct
route:

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

(Note that the last line is simply `(lookup name rest-env)', but we know
that we have a functional implementation.)

To see how all this works, try out extending an empty environment a few times and examine the result. For example, the environment that we began with:

```
  (define (my-map id)
    (cond [(eq? 'a id) 1]
          [(eq? 'b id) 2]
          [else (error ...)]))
```

behaves in the same way (if the type of values is numbers) as

```
  (Extend 'a 1 (Extend 'b 2 (EmptyEnv)))
```

The new code is now the same, except for the environment code:

```
  ----------------------------------------------------------------------
-
  #lang pl

  #|
  The grammar:
    <FLANG> ::= <num>
              | { + <FLANG> <FLANG> }
              | { - <FLANG> <FLANG> }
              | { * <FLANG> <FLANG> }
              | { / <FLANG> <FLANG> }
              | { with { <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({fun {x} E},env)      = <{fun {x} E}, env>
  eval({call E1 E2},env1)
           = eval(Ef,extend(x,eval(E2,env1),env2))
                              if eval(E1,env1) = <{fun {x} Ef}, 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]
  [Fun  Symbol FLANG]
  [Call FLANG FLANG])

(: parse-sexpr : Sexpr -> FLANG)
;; to convert 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 '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)))

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

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

(: 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->number : VAL -> Number)
  (define (NumV->number v)
    (cases v
      [(NumV n) n]
      [else (error 'arith-op "expects a number, got: ~s" v)]))
  (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))]
    [(Id name) (lookup name env)]
    [(Fun bound-id bound-body)
     (FunV bound-id bound-body env)]
    [(Call fun-expr arg-expr)
     (let ([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 "{call {call {fun {x} {call x 1}}
                        {fun {x} {fun {y} {+ x y}}}}
                    123}")
        => 124)
  ----------------------------------------------------------------------
-

=====================================================================
=
>>> More Closures (on both levels)

Racket closures (=functions) can be used in other places too, and as we
have seen, they can do more than encapsulate various values -- they can
also hold the behavior that is expected of these values.

To demonstrate this we will deal with closures in our language.  We
currently use a variant that holds the three pieces of relevant
information:

    [FunV (name Symbol) (body FLANG) (env ENV)]

We can replace this by a functional object, which will hold the three
values.  First, change the VAL type to hold functions for FunV values:

  (define-type VAL
    [NumV Number]
    [FunV (? -> ?)])
```

And note that the function should somehow encapsulate the same information that was there previously, the question is *how* this information is going to be done, and this will determine the actual type. This information plays a role in two places in our evaluator -- generating a closure in the `Fun' case, and using it in the `Call' case:

```
      [(Fun bound-id bound-body)
       (FunV bound-id bound-body env)]
      [(Call fun-expr arg-expr)
       (let ([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)]))]
```

we can simply fold the marked functionality bit of `Call' into a Racket function that will be stored in a `FunV' object -- this piece of functionality takes an argument value, extends the closure's environment
with its value and the function's name, and continues to evaluate the function body. Folding all of this into a function gives us:

```
  (lambda (arg-val)
    (eval bound-body (Extend bound-id arg-val env)))
```

where the values of `bound-body', `bound-id', and `val' are known at the
time that the FunV is *constructed*. Doing this gives us the following code for the two cases:

```
      [(Fun bound-id bound-body)
       (FunV (lambda (arg-val)
               (eval bound-body (Extend bound-id arg-val env))))]
      [(Call fun-expr arg-expr)
       (let ([fval (eval fun-expr env)])
         (cases fval
           [(FunV proc) (proc (eval arg-expr env))]
           [else (error 'eval "`call' expects a function, got: ~s"
                        fval)]))]
```

And now the type of the function is clear:

```
  (define-type VAL
    [NumV Number]
    [FunV (VAL -> VAL)])
```

And again, the rest of the code is unmodified:

```
  ----------------------------------------------------------------------
-
  #lang pl
```

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

(: parse-sexpr : Sexpr -> FLANG)
;; to convert 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 '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 (VAL -> VAL)])

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

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

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

```
      (rest-env name))))

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

(: 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->number : VAL -> Number)
  (define (NumV->number v)
    (cases v
      [(NumV n) n]
      [else (error 'arith-op "expects a number, got: ~s" v)]))
  (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))]
    [(Id name) (lookup name env)]
    [(Fun bound-id bound-body)
     (FunV (lambda (arg-val)
             (eval bound-body (Extend bound-id arg-val env))))]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr env)])
       (cases fval
         [(FunV proc) (proc (eval arg-expr 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 "{call {call {fun {x} {call x 1}}
                        {fun {x} {fun {y} {+ x y}}}}
                    123}")
        => 124)
  ----------------------------------------------------------------------
-

=====================================================================
=
>>> Types of Evaluators
```