PL: Lecture #11

Tuesday, October 15th

Lexical Scope using Racket Closures

> 👉 ***PLAI §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 Symbol VAL 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 val 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 val rest-env)
  (lambda (name)
    ...))
```

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

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

If the `name` that we're looking for is not the same as `id`, then we need to search through the previous environment:

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

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 instead:

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

```
  (lambda (name)
    (if (eq? name id)
      val
      (rest-env name)))) ; same as (lookup name rest-env)
```

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(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]
  [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 '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)
;; 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))))

(: 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))

(: 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))]
    [(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)
```

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 Symbol FLANG 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)
 (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)])]
```

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)
 (define 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)
;; 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 '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)
;; 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))))

(: 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))

(: 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))]
```

```
    [(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)
     (define 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 "{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)
```

Types of Evaluators

What we did just now is implement lexical environments and closures in the language we implement using lexical environments and closures in our own language (Racket)!

This is another example of embedding a feature of the host language in the implemented language, an issue that we have already discussed.

There are many examples of this, even when the two languages involved are different. For example, if we have this bit in the C implementation of Racket:

```
// Disclaimer: not real Racket code
Racket_Object *eval_and(int argc, Racket_Object *argv[]) {
  Racket_Object *tmp;
  if ( argc != 2 )
    signal_racket_error("bad number of arguments");
  else if ( racket_eval(argv[0]) != racket_false
            &&
            (tmp = racket_eval(argv[1])) != racket_false )
    return tmp;
  else
    return racket_false;
}
```

then the special semantics of evaluating a Racket `and` form is being inherited from C's special treatment of `&&`. You can see this by the fact that if there is a bug in the C compiler, then it will propagate to the resulting Racket implementation too. A different solution is to not use `&&` at all:

```
// Disclaimer: not real Racket code
Racket_Object *eval_and(int argc, Racket_Object *argv[]) {
  Racket_Object *tmp;
  if ( argc != 2 )
    signal_racket_error("bad number of arguments");
  else if ( racket_eval(argv[0]) != racket_false )
    return racket_eval(argv[1]);
  else
    return racket_false;
}
```

and we can say that this is even better since it evaluates the second expression in tail position. But in this case we don't really get that benefit, since C itself is not doing tail-call optimization as a standard feature (though some compilers do so under some circumstances).

We have seen a few different implementations of evaluators that are quite different in flavor. They suggest the following taxonomy.

- A ***syntactic evaluator*** is one that uses its own language to represent expressions and semantic runtime values of the evaluated language, implementing all the corresponding behavior explicitly.
- A ***meta evaluator*** is an evaluator that uses language features of its own language to directly implement behavior of the evaluated language.

While our substitution-based FLANG evaluator was close to being a syntactic evaluator, we haven't written any purely syntactic evaluators so far: we still relied on things like Racket arithmetics etc. The most recent evaluator that we have studied, is even more of a *meta* evaluator than the preceding ones: it doesn't even implement closures and lexical scope, and instead, it uses the fact that Racket itself has them.

With a good match between the evaluated language and the implementation language, writing a meta evaluator can be very easy. With a bad match, though, it can be very hard. With a syntactic evaluator, implementing each semantic feature will be somewhat hard, but in return you don't have to worry as much about how well the implementation and the evaluated languages match up. In particular, if there is a particularly strong mismatch between the implementation and the evaluated language, it may take less effort to write a syntactic evaluator than a meta evaluator.

As an exercise, we can build upon our latest evaluator to remove the encapsulation of the evaluator's response in the VAL type. The resulting evaluator is shown below. This is a true meta evaluator: it uses

Racket closures to implement FLANG closures, Racket function application for FLANG function application, Racket numbers for FLANG numbers, and Racket arithmetic for FLANG arithmetic. In fact, ignoring some small syntactic differences between Racket and FLANG, this latest evaluator can be classified as something more specific than a meta evaluator:

- A ***meta-circular evaluator*** is a meta evaluator in which the implementation and the evaluated languages are the same.

  This is essentially the concept of a “universal” evaluator, as in a “universal turing machine”.

(Put differently, the trivial nature of the evaluator clues us in to the deep connection between the two languages, whatever their syntactic differences may be.)

Feature Embedding

We saw that the difference between lazy evaluation and eager evaluation is in the evaluation rules for `with` forms, function applications, etc:

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

is eager, and

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

is lazy. But is the first rule *really* eager? The fact is that the only thing that makes it eager is the fact that our understanding of the mathematical notation is eager — if we were to take math as lazy, then the description of the rule becomes a description of lazy evaluation.

Another way to look at this is — take the piece of code that implements this evaluation:

```
(: eval : FLANG -> Number)
;; evaluates FLANG expressions by reducing them to numbers
(define (eval expr)
  (cases expr
    ...
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (Num (eval named-expr))))]
    ...))
```

and the same question applies: is this really implementing eager evaluation? We know that this is indeed eager — we can simply try it and check that it is, but it is only eager because we are using an eager language for the implementation! If our own language was lazy, then the evaluator’s implementation would run lazily, which means that the above applications of the `eval` and the `subst` functions would also be lazy, making our evaluator lazy as well.

This is a general phenomena where some of the semantic features of the language we use (math in the formal description, Racket in our code) gets *embedded* into the language we implement.

Here’s another example — consider the code that implements arithmetics:

```
(: eval : FLANG -> Number)
;; evaluates FLANG expressions by reducing them to numbers
(define (eval expr)
  (cases expr
    [(Num n) n]
    [(Add l r) (+ (eval l) (eval r))]
    ...))
```

what if it was written like this:

```
FLANG eval(FLANG expr) {
  if (is_Num(expr))
    return num_of_Num(expr);
  else if (is_Add(expr))
    return eval(lhs_of_Add(expr)) + eval(rhs_of_Add(expr));
  else if ...
  ...
}
```

Would it still implement unlimited integers and exact fractions? That depends on the language that was used to implement it: the above syntax suggests C, C++, Java, or some other relative, which usually come with limited integers and no exact fractions. But this really depends on the language — even our own code has unlimited integers and exact rationals only because Racket has them. If we were using a language that didn't have such features (there are such Scheme implementations), then our implemented language would absorb these (lack of) features too, and its own numbers would be limited in just the same way. (And this includes the syntax for numbers, which we embedded intentionally, like the syntax for identifiers).

The bottom line is that we should be aware of such issues, and be very careful when we talk about semantics. Even the language that we use to communicate (semi-formal logic) can mean different things.

Aside: read "Reflections on Trusting Trust" by Ken Thompson (You can skip to the "Stage II" part to get to the interesting stuff.)

(And when you're done, look for "XcodeGhost" to see a relevant example, and don't miss the leaked document on the wikipedia page…)

Here is yet another variation of our evaluator that is even closer to a meta-circular evaluator. It uses Racket values directly to implement values, so arithmetic operations become straightforward. Note especially how the case for function application is similar to arithmetics: a FLANG function application translates to a Racket function application. In both cases (applications and arithmetics) we don't even check the objects since they are simple Racket objects — if our language happens to have some meaning for arithmetics with functions, or for applying numbers, then we will inherit the same semantics in our language. This means that we now specify less behavior and fall back more often on what Racket does.

We use Racket values with this type definition:

```
(define-type VAL = (U Number (VAL -> VAL)))
```

And the evaluation function can now be:

```
(: eval : FLANG ENV -> VAL)
;; evaluates FLANG expressions by reducing them to values
(define (eval expr env)
  (cases expr
    [(Num n) n]                 ;*** return the actual number
    [(Add l r) (+ (eval l env) (eval r env))]
    [(Sub l r) (- (eval l env) (eval r env))]
    [(Mul l r) (* (eval l env) (eval r env))]
    [(Div l r) (/ (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)
     (lambda ([arg-val : VAL]) ;*** return the racket function
       ;; note that this requires input type specifications since
       ;; typed racket can't guess the right one
       (eval bound-body (Extend bound-id arg-val env)))]
    [(Call fun-expr arg-expr)
     ((eval fun-expr env)       ;*** trivial like the arithmetics!
      (eval arg-expr env))]))
```

Note how the arithmetics implementation is simple — it's a direct translation of the FLANG syntax to Racket operations, and since we don't check the inputs to the Racket operations, we let Racket throw type errors for us. Note also how function application is just like the arithmetic operations: a FLANG application is directly translated to a Racket application.

However, this does not work quite as simply in Typed Racket. The whole point of typechecking is that we never run into type errors — so we cannot throw back on Racket errors since code that might produce them is forbidden! A way around this is to perform explicit checks that guarantee that Racket cannot run into type errors. We do this with the following two helpers that are defined inside `eval`:

```
  (: evalN : FLANG -> Number)
  (define (evalN e)
    (let ([n (eval e env)])
      (if (number? n)
        n
        (error 'eval "got a non-number: ~s" n))))
  (: evalF : FLANG -> (VAL -> VAL))
  (define (evalF e)
    (let ([f (eval e env)])
      (if (function? f)
        f
        (error 'eval "got a non-function: ~s" f))))
```

Note that Typed Racket is "smart enough" to figure out that in `evalF` the result of the recursive evaluation has to be either `Number` or `(VAL -> VAL)`; and since the `if` throws out on numbers, we're left with `(VAL -> VAL)` functions, not just any function.

```
#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)
;; 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 '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

;; Values are plain Racket values, no new VAL wrapper;
;; (but note that this is a recursive definition)
(define-type VAL = (U Number (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)
;; 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))))

(: 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))

(: eval : FLANG ENV -> VAL)
;; evaluates FLANG expressions by reducing them to values
(define (eval expr env)
  (: evalN : FLANG -> Number)
  (define (evalN e)
    (let ([n (eval e env)])
      (if (number? n)
        n
        (error 'eval "got a non-number: ~s" n))))
  (: evalF : FLANG -> (VAL -> VAL))
  (define (evalF e)
    (let ([f (eval e env)])
      (if (function? f)
        f
        (error 'eval "got a non-function: ~s" f))))
  (cases expr
    [(Num n) n]
    [(Add l r) (+ (evalN l) (evalN r))]
    [(Sub l r) (- (evalN l) (evalN r))]
    [(Mul l r) (* (evalN l) (evalN r))]
    [(Div l r) (/ (evalN l) (evalN r))]
```

```
    [(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)
     (lambda ([arg-val : VAL])
       (eval bound-body (Extend bound-id arg-val env)))]
    [(Call fun-expr arg-expr)
     ((evalF fun-expr) (eval arg-expr env))]))

(: run : String -> VAL) ; no need to convert VALs to numbers
;; evaluate a FLANG program contained in a string
(define (run str)
  (eval (parse str) (EmptyEnv)))

;; 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)
```

Recursion, Recursion, Recursion

> 👉 ***PLAI §9***

There is one major feature that is still missing from our language: we have no way to perform recursion (therefore no kind of loops). So far, we could only use recursion when we had *names*. In FLANG, the only way we can have names is through `with` which not good enough for recursion.

To discuss the issue of recursion, we switch to a “broken” version of (untyped) Racket — one where a `define` has a different scoping rules: the scope of the defined name does *not* cover the defined expression. Specifically, in this language, this doesn’t work:

```
#lang pl broken
(define (fact n)
  (if (zero? n) 1 (* n (fact (- n 1)))))
(fact 5)
```

In our language, this translation would also not work (assuming we have `if` etc):

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

And similarly, in plain Racket this won’t work if `let` is the only tool you use to create bindings:

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

In the broken-scope language, the `define` form is more similar to a mathematical definition. For example, when we write:

```
(define (F x) x)
(define (G y) (F y))
(G F)
```

it is actually shorthand for

```
(define F (lambda (x) x))
(define G (lambda (y) (F y)))
(G F)
```

we can then replace defined names with their definitions:

```
(define F (lambda (x) x))
(define G (lambda (y) (F y)))
((lambda (y) (F y)) (lambda (x) x))
```

and this can go on, until we get to the actual code that we wrote:

```
((lambda (y) ((lambda (x) x) y)) (lambda (x) x))
```

This means that the above `fact` definition is similar to writing:

```
fact := (lambda (n)
          (if (zero? n) 1 (* n (fact (- n 1)))))
(fact 5)
```

which is not a well-formed definition — it is *meaningless* (this is a formal use of the word “meaningless”). What we’d really want, is to take the *equation* (using `=` instead of `:=`)

```
fact = (lambda (n)
         (if (zero? n) 1 (* n (fact (- n 1)))))
```

and find a solution which will be a value for `fact` that makes this true.

If you look at the Racket evaluation rules handout on the web page, you will see that this problem is related to the way that we introduced the Racket `define`: there is a hand-wavy explanation that talks about *knowing* things.

The big question is: can we define recursive functions without Racket’s magical `define` form?

> *Note: This question is a little different than the question of implementing recursion in our language — in the Racket case we have no control over the implementation of the language. As it will eventually turn out, implementing recursion in our own language will be quite easy when we use mutation in a specific way. So the question that we're now facing can be phrased as either "can we get recursion in Racket without Racket's magical definition forms?" or "can we get recursion in our interpreter without mutation?".*