# PL: Lecture #8

***Tuesday, October 1st***

## The FLANG Language

Now for the implementation — we call this new language FLANG.

First, the BNF:

```
<FLANG> ::= <num>
          | { + <FLANG> <FLANG> }
          | { - <FLANG> <FLANG> }
          | { * <FLANG> <FLANG> }
          | { / <FLANG> <FLANG> }
          | { with { <id> <FLANG> } <FLANG> }
          | <id>
          | { fun { <id> } <FLANG> }
          | { call <FLANG> <FLANG> }
```

And the matching type definition:

```
(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] ; No named-expression
  [Call FLANG FLANG])
```

The parser for this grammar is, as usual, straightforward:

```
(: 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)]))
```

We also need to patch up the substitution function to deal with these things. The scoping rule for the new function form is, unsurprisingly, similar to the rule of `with`, except that there is no extra expression now, and the scoping rule for `call` is the same as for the arithmetic operators:

```
N[v/x]                = N

{+ E1 E2}[v/x]        = {+ E1[v/x] E2[v/x]}

{- E1 E2}[v/x]        = {- E1[v/x] E2[v/x]}

{* E1 E2}[v/x]        = {* E1[v/x] E2[v/x]}

{/ E1 E2}[v/x]        = {/ E1[v/x] E2[v/x]}

y[v/x]                = y
x[v/x]                = v

{with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]}
{with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}

{call E1 E2}[v/x]     = {call E1[v/x] E2[v/x]}

{fun {y} E}[v/x]      = {fun {y} E[v/x]}
{fun {x} E}[v/x]      = {fun {x} E}
```

And the matching code:

```
(: subst : FLANG Symbol FLANG -> FLANG)
;; substitutes the second argument with the third argument in the
;; first argument, as per the rules of substitution; the resulting
;; expression contains no free instances of the second argument
(define (subst expr from to)
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
    [(Sub l r) (Sub (subst l from to) (subst r from to))]
    [(Mul l r) (Mul (subst l from to) (subst r from to))]
    [(Div l r) (Div (subst l from to) (subst r from to))]
    [(Id name) (if (eq? name from) to expr)]
    [(With bound-id named-expr bound-body)
     (With bound-id
           (subst named-expr from to)
           (if (eq? bound-id from)
             bound-body
             (subst bound-body from to)))]
    [(Call l r) (Call (subst l from to) (subst r from to))]
    [(Fun bound-id bound-body)
     (if (eq? bound-id from)
       expr
       (Fun bound-id (subst bound-body from to)))]))
```

---

Now, before we start working on an evaluator, we need to decide on what exactly do we use to represent values of this language. Before we had functions, we had only number values and we used Racket numbers to represent them. Now we have two kinds of values — numbers and functions. It seems easy

enough to continue using Racket numbers to represent numbers, but what about functions? What should be the result of evaluating

```
{fun {x} {+ x 1}}
```

? Well, this is the new toy we have: it should be a function value, which is something that can be used just like numbers, but instead of arithmetic operations, we can `call` these things. What we need is a way to avoid evaluating the body expression of the function — *delay* it — and instead use some value that will contain this delayed expression in a way that can be used later.

To accommodate this, we will change our implementation strategy a little: we will use our syntax objects for numbers (`(Num n)` instead of just `n`), which will be a little inconvenient when we do the arithmetic operations, but it will simplify life by making it possible to evaluate functions in a similar way: simply return their own syntax object as their values. The syntax object has what we need: the body expression that needs to be evaluated later when the function is called, and it also has the identifier name that should be replaced with the actual input to the function call. This means that evaluating:

```
(Add (Num 1) (Num 2))
```

now yields

```
(Num 3)
```

and a number `(Num 5)` evaluates to `(Num 5)`.

In a similar way, `(Fun 'x (Num 2))` evaluates to `(Fun 'x (Num 2))`.

Why would this work? Well, because `call` will be very similar to `with` — the only difference is that its arguments are ordered a little differently, being retrieved from the function that is applied and the argument.

The formal evaluation rules are therefore treating functions like numbers, and use the syntax object to represent both values:

```
eval(N)         = N

eval({+ E1 E2}) = eval(E1) + eval(E2)

eval({- E1 E2}) = eval(E1) - eval(E2)

eval({* E1 E2}) = eval(E1) * eval(E2)

eval({/ E1 E2}) = eval(E1) / eval(E2)

eval(id)        = error!

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

eval(FUN)       = FUN ; assuming FUN is a function expression

eval({call E1 E2})
                = eval(B[eval(E2)/x])   if eval(E1) = {fun {x} B}
                = error!                otherwise
```

Note that the last rule could be written using a translation to a `with` expression:

```
eval({call E1 E2})
                = eval({with {x E2} B}) if eval(E1) = {fun {x} B}
                = error!                otherwise
```

And alternatively, we could specify `with` using `call` and `fun`:

```
eval({with {x E1} E2}) = eval({call {fun {x} E2} E1})
```

There is a small problem in these rules which is intuitively seen by the fact that the evaluation rule for a `call` is expected to be very similar to the one for arithmetic operations. We now have two kinds of values, so we need to check the arithmetic operation's arguments too:

```
eval({+ E1 E2}) = N1 + N2
                  if eval(E1), eval(E2) evaluate to numbers N1, N2
                  otherwise error!
...
```

The corresponding code is:

```
(: eval : FLANG -> FLANG)                        ;*** note return type
;; evaluates FLANG expressions by reducing them to *expressions* but
;; only expressions that stand for values: only `Fun`s and `Num`s
(define (eval expr)
  (cases expr
    [(Num n) expr]                               ;*** change here
    [(Add l r) (arith-op + (eval l) (eval r))]
    [(Sub l r) (arith-op - (eval l) (eval r))]
    [(Mul l r) (arith-op * (eval l) (eval r))]
    [(Div l r) (arith-op / (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (eval named-expr)))]           ;*** no `(Num ...)'
    [(Id name) (error 'eval "free identifier: ~s" name)]
    [(Fun bound-id bound-body) expr]             ;*** similar to `Num'
    [(Call (Fun bound-id bound-body) arg-expr) ;*** nested pattern
     (eval (subst bound-body                     ;*** just like `with'
                  bound-id
                  (eval arg-expr)))]
    [(Call something arg-expr)
     (error 'eval "`call' expects a function, got: ~s" something)]))
```

Note that the `Call` case is doing the same thing we do in the `With` case. In fact, we could have just *generated* a `With` expression and evaluate that instead:

```
    ...
    [(Call (Fun bound-id bound-body) arg-expr)
     (eval (With bound-id arg-expr bound-body))]
    ...
```

The `arith-op` function is in charge of checking that the input values are numbers (represented as FLANG numbers), translating them to plain numbers, performing the Racket operation, then re-wrapping the result in a `Num`. Note how its type indicates that it is a higher-order function.

```
(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)
;; gets a Racket numeric binary operator, and uses it within a FLANG
;; `Num' wrapper (note the H.O. type, and note the hack of the `val`
;; name which is actually an AST that represents a runtime value)
(define (arith-op op val1 val2)
  (Num (op (Num->number val1) (Num->number val2))))
```

It uses the following function to convert FLANG numbers to Racket numbers. (Note that `else` is almost always a bad idea since it can prevent the compiler from showing you places to edit code — but this case is an exception since we never want to deal with anything other than `Num`s.) The reason that this function is relatively trivial is that we chose the easy way and represented numbers using Racket numbers, but we could have used strings or anything else.

```
(: Num->number : FLANG -> Number)
;; convert a FLANG number to a Racket one
(define (Num->number e)
  (cases e
    [(Num n) n]
    [else (error 'arith-op "expected a number, got: ~s" e)]))
```

We can also make things a little easier to use if we make `run` convert the result to a number:

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

Adding few simple tests we get:

```
;; The Flang interpreter

#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:

  subst:
    N[v/x]                = N
    {+ E1 E2}[v/x]        = {+ E1[v/x] E2[v/x]}
    {- E1 E2}[v/x]        = {- E1[v/x] E2[v/x]}
    {* E1 E2}[v/x]        = {* E1[v/x] E2[v/x]}
    {/ E1 E2}[v/x]        = {/ E1[v/x] E2[v/x]}
    y[v/x]                = y
    x[v/x]                = v
    {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]} ; if y =/= x
    {with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
    {call E1 E2}[v/x]     = {call E1[v/x] E2[v/x]}
    {fun {y} E}[v/x]      = {fun {y} E[v/x]}           ; if y =/= x
    {fun {x} E}[v/x]      = {fun {x} E}

  eval:
    eval(N)            = N
    eval({+ E1 E2})    = eval(E1) + eval(E2)  \ if both E1 and E2
    eval({- E1 E2})    = eval(E1) - eval(E2)   \ evaluate to numbers
    eval({* E1 E2})    = eval(E1) * eval(E2)   / otherwise error!
    eval({/ E1 E2})    = eval(E1) / eval(E2)  /
```

```
    eval(id)           = error!
    eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
    eval(FUN)          = FUN ; assuming FUN is a function expression
    eval({call E1 E2}) = eval(B[eval(E2)/x])
                           if eval(E1)={fun {x} B}, otherwise error!
|#

(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)))

(: subst : FLANG Symbol FLANG -> FLANG)
;; substitutes the second argument with the third argument in the
;; first argument, as per the rules of substitution; the resulting
;; expression contains no free instances of the second argument
(define (subst expr from to)
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
    [(Sub l r) (Sub (subst l from to) (subst r from to))]
    [(Mul l r) (Mul (subst l from to) (subst r from to))]
    [(Div l r) (Div (subst l from to) (subst r from to))]
```

```
    [(Id name) (if (eq? name from) to expr)]
    [(With bound-id named-expr bound-body)
     (With bound-id
           (subst named-expr from to)
           (if (eq? bound-id from)
             bound-body
             (subst bound-body from to)))]
    [(Call l r) (Call (subst l from to) (subst r from to))]
    [(Fun bound-id bound-body)
     (if (eq? bound-id from)
       expr
       (Fun bound-id (subst bound-body from to)))]))

(: Num->number : FLANG -> Number)
;; convert a FLANG number to a Racket one
(define (Num->number e)
  (cases e
    [(Num n) n]
    [else (error 'arith-op "expected a number, got: ~s" e)]))

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

(: eval : FLANG -> FLANG)
;; evaluates FLANG expressions by reducing them to *expressions* but
;; only expressions that stand for values: only `Fun`s and `Num`s
(define (eval expr)
  (cases expr
    [(Num n) expr]
    [(Add l r) (arith-op + (eval l) (eval r))]
    [(Sub l r) (arith-op - (eval l) (eval r))]
    [(Mul l r) (arith-op * (eval l) (eval r))]
    [(Div l r) (arith-op / (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (eval named-expr)))]
    [(Id name) (error 'eval "free identifier: ~s" name)]
    [(Fun bound-id bound-body) expr]
    [(Call (Fun bound-id bound-body) arg-expr)
     (eval (subst bound-body
                  bound-id
                  (eval arg-expr)))]
    [(Call something arg-expr)
     (error 'eval "`call' expects a function, got: ~s" something)]))

(: run : String -> Number)
;; evaluate a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str))])
    (cases result
      [(Num 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)
```

---

There is still a problem with this version. First a question — if `call` is similar to arithmetic operations (and to `with` in what it actually does), then how come the code is different enough that it doesn't even need an auxiliary function?

Second question: what *should* happen if we evaluate these code snippets:

```
(run "{with {add {fun {x}
                   {fun {y}
                     {+ x y}}}}
       {call {call add 8} 9}}")
(run "{with {identity {fun {x} x}}
        {with {foo {fun {x} {+ x 1}}}
          {call {call identity foo} 123}}}")
(run "{call {call {fun {x} {call x 1}}
                  {fun {x} {fun {y} {+ x y}}}}
            123}")
```

Third question, what *will* happen if we do the above?

What we're missing is an evaluation of the function expression, in case it's not a literal `fun` form. The following fixes this:

```
(: eval : FLANG -> FLANG)
;; evaluates FLANG expressions by reducing them to *expressions* but
;; only expressions that stand for values: only `Fun`s and `Num`s
(define (eval expr)
  (cases expr
    [(Num n) expr]
    [(Add l r) (arith-op + (eval l) (eval r))]
    [(Sub l r) (arith-op - (eval l) (eval r))]
    [(Mul l r) (arith-op * (eval l) (eval r))]
    [(Div l r) (arith-op / (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (eval named-expr)))]
    [(Id name) (error 'eval "free identifier: ~s" name)]
    [(Fun bound-id bound-body) expr]
    [(Call fun-expr arg-expr)
     (define fval (eval fun-expr))       ;*** need to evaluate this!
     (cases fval
       [(Fun bound-id bound-body)
        (eval (subst bound-body bound-id (eval arg-expr)))]
       [else (error 'eval "`call' expects a function, got: ~s"
                    fval)])]))
```

The complete code is:

```
;; The Flang interpreter

#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:

  subst:
    N[v/x]                = N
    {+ E1 E2}[v/x]        = {+ E1[v/x] E2[v/x]}
    {- E1 E2}[v/x]        = {- E1[v/x] E2[v/x]}
    {* E1 E2}[v/x]        = {* E1[v/x] E2[v/x]}
    {/ E1 E2}[v/x]        = {/ E1[v/x] E2[v/x]}
    y[v/x]                = y
    x[v/x]                = v
    {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]} ; if y =/= x
    {with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
    {call E1 E2}[v/x]     = {call E1[v/x] E2[v/x]}
    {fun {y} E}[v/x]      = {fun {y} E[v/x]}           ; if y =/= x
    {fun {x} E}[v/x]      = {fun {x} E}

  eval:
    eval(N)            = N
    eval({+ E1 E2})    = eval(E1) + eval(E2)  \ if both E1 and E2
    eval({- E1 E2})    = eval(E1) - eval(E2)   \ evaluate to numbers
    eval({* E1 E2})    = eval(E1) * eval(E2)   / otherwise error!
    eval({/ E1 E2})    = eval(E1) / eval(E2)  /
    eval(id)           = error!
    eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
    eval(FUN)          = FUN ; assuming FUN is a function expression
    eval({call E1 E2}) = eval(B[eval(E2)/x])
                           if eval(E1)={fun {x} B}, otherwise error!
|#

(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)))

(: subst : FLANG Symbol FLANG -> FLANG)
;; substitutes the second argument with the third argument in the
;; first argument, as per the rules of substitution; the resulting
;; expression contains no free instances of the second argument
(define (subst expr from to)
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
    [(Sub l r) (Sub (subst l from to) (subst r from to))]
    [(Mul l r) (Mul (subst l from to) (subst r from to))]
    [(Div l r) (Div (subst l from to) (subst r from to))]
    [(Id name) (if (eq? name from) to expr)]
    [(With bound-id named-expr bound-body)
     (With bound-id
           (subst named-expr from to)
           (if (eq? bound-id from)
             bound-body
             (subst bound-body from to)))]
    [(Call l r) (Call (subst l from to) (subst r from to))]
    [(Fun bound-id bound-body)
     (if (eq? bound-id from)
       expr
       (Fun bound-id (subst bound-body from to)))]))

(: Num->number : FLANG -> Number)
;; convert a FLANG number to a Racket one
(define (Num->number e)
  (cases e
    [(Num n) n]
    [else (error 'arith-op "expected a number, got: ~s" e)]))
```

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

(: eval : FLANG -> FLANG)
;; evaluates FLANG expressions by reducing them to *expressions* but
;; only expressions that stand for values: only `Fun`s and `Num`s
(define (eval expr)
  (cases expr
    [(Num n) expr]
    [(Add l r) (arith-op + (eval l) (eval r))]
    [(Sub l r) (arith-op - (eval l) (eval r))]
    [(Mul l r) (arith-op * (eval l) (eval r))]
    [(Div l r) (arith-op / (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (eval named-expr)))]
    [(Id name) (error 'eval "free identifier: ~s" name)]
    [(Fun bound-id bound-body) expr]
    [(Call fun-expr arg-expr)
     (define fval (eval fun-expr))
     (cases fval
       [(Fun bound-id bound-body)
        (eval (subst bound-body bound-id (eval arg-expr)))]
       [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))])
    (cases result
      [(Num 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 {add {fun {x}
                         {fun {y}
                           {+ x y}}}}
              {call {call add 8} 9}}")
      => 17)
(test (run "{with {identity {fun {x} x}}
              {with {foo {fun {x} {+ x 1}}}
```

```
                  {call {call identity foo} 123}}}")
      => 124)
(test (run "{call {call {fun {x} {call x 1}}
                        {fun {x} {fun {y} {+ x y}}}}
                  123}")
      => 124)
```

# Introducing Racket's `lambda`

> *Quick laundry list of things to go over:*
>
> - *`fun` & `lambda`*
> - *difference between lambda and simple values*
> - *not being able to do recursive functions with `let`*
> - *let* as a derived form*
> - *let with lambda in Racket –> can be a derived form*
> - *how `if` can be used to implement `and` and `or` as derived forms*
> - *Newtonian syntax vs. a lambda expression.*

Almost all modern languages have this capability. For example, this:

```
(define (f g) (g 2 3))
(f +) ==> 5
(f *) ==> 6
(f (lambda (x y) (+ (square x) (square y)))) ==> 13
```

Can be written in JavaScript like this:

```
function f(g) { return g(2,3); }
function square(x) { return x*x; }
console.log(f(function (x,y) { return square(x) + square(y); }));
```

or in ES6 JavaScript:

```
let f = (g) => g(2,3);
let square = (x) => x*x;
console.log(f((x,y) => square(x) + square(y)));
```

In Perl:

```
sub f { my ($g) = @_; return $g->(2,3); }
sub square { my ($x) = @_; return $x * $x; }
print f(sub { my ($x, $y) = @_; return square($x) + square($y); });
```

In Ruby:

```
def f(g) g.call(2,3) end
def square(x) x*x end
puts f(lambda{|x,y| square(x) + square(y)})
```

etc. Even **Java has lambda expressions**, and "recently" **C++ added them too**.

# Using Functions as Objects

A very important aspect of Racket — using "higher order" functions — functions that get and return functions. Here is a very simple example:

```
(define (f x) (lambda () x))
(define a (f 2))
(a)  -->  2
(define b (f 3))
(b)  -->  3
```

Note: what we get is actually an object that remembers (by the substitution we're doing) a number. How about:

```
(define aa (f a))
(aa)    -->  #<procedure> (this is a)
((aa))  -->  2
```

Take this idea to the next level:

```
(define (kons x y)
  (lambda (b)
    (if b x y)))
(define (kar p) (p #t))
(define (kdr p) (p #f))
(define a (kons 1 2))
(define b (kons 3 4))
(list (kar a) (kdr a))
(list (kar b) (kdr b))
```

Or, with types:

```
(: kons : (All (A B) A B -> (Boolean -> (U A B))))
(define (kons x y)
  (lambda (b)
    (if b x y)))
(: kar : (All (T) (Boolean -> T) -> T))
(define (kar p) (p #t))
(: kdr : (All (T) (Boolean -> T) -> T))
(define (kdr p) (p #f))
(define a (kons 1 2))
(define b (kons 3 4))
(list (kar a) (kdr a))
(list (kar b) (kdr b))
```

Even more — why should the internal function expect a boolean and choose what to return? We can simply expect a function that will take the two values and return one:

```
(define (kons x y) (lambda (s) (s x y)))
(define (kar p) (p (lambda (x y) x)))
(define (kdr p) (p (lambda (x y) y)))
(define a (kons 1 2))
(define b (kons 3 4))
(list (kar a) (kdr a))
(list (kar b) (kdr b))
```

And a typed version, using our own constructor to make it a little less painful:

```
(define-type (Kons A B) = ((A B -> (U A B)) -> (U A B)))

(: kons : (All (A B) A B -> (Kons A B)))
(define (kons x y) (lambda (s) (s x y)))
(: kar : (All (A B) (Kons A B) -> (U A B)))
(define (kar p) (p (lambda (x y) x)))
(: kdr : (All (A B) (Kons A B) -> (U A B)))
```

```
(define (kdr p) (p (lambda (x y) y)))
(define a (kons 1 2))
(define b (kons 3 4))
(list (kar a) (kdr a))
(list (kar b) (kdr b))
```

Note that the `Kons` type definition is the same as:

```
(define-type Kons = (All (A B) (A B -> (U A B)) -> (U A B)))
```

so `All` is to polymorphic type definitions what `lambda` is for function definitions.

Finally, in JavaScript:

```
function kons(x,y) { return function(s) { return s(x, y); } }
function kar(p) { return p(function(x,y){ return x; }); }
function kdr(p) { return p(function(x,y){ return y; }); }
a = kons(1,2);
b = kons(3,4);
console.log('a = <' + kar(a) + ',' + kdr(a) + '>' );
console.log('b = <' + kar(b) + ',' + kdr(b) + '>' );
```

or with ES6 *arrow functions*, the function definitions become:

```
const kons = (x,y) => s => s(x,y);
const kar  = p => p((x,y) => x);
const kdr  = p => p((x,y) => y);
```

and using Typescript to add types:

```
type Kons<A,B> = (s: (x:A, y:B) => A|B) => A | B
const kons = <A,B>(x:A,y:B) => (s: ((x:A, y:B) => A|B)) => s(x,y);
const kar  = <A,B>(p: Kons<A,B>) => p((x,y) => x);
const kdr  = <A,B>(p: Kons<A,B>) => p((x,y) => y);
```

# Using `define-type` for new “type aliases”

As seen in these examples, there is another way to use `define-type`, using a `=` to create a new type name “alias” for an *existing* type. For example:

```
(define-type Strings = (Listof String))
```

These uses of `define-type` do not define any new kind of type, they are essentially a convenience tool for making code shorter and more readable.

```
(define-type NumericFunction = Number -> Number)

(: square : NumericFunction)
(define (square n) (* n n))
```

Note in particular that this can also be used to define “alias type constructors” too: somewhat similar to creating new “type functions”. For example:

```
(define-type (BinaryFun In Out) = In In -> Out)

(: diagonal : (BinaryFun Natural Number))
(define (diagonal width height)
  (sqrt (+ (* width width) (* height height))))
```

This is something that we will only need in a few rare cases.

# Currying

A *curried* function is a function that, instead of accepting two (or more) arguments, accepts only one and returns a function that accepts the rest. For example:

```
(: plus : Number -> (Number -> Number))
(define (plus x)
  (lambda (y)
    (+ x y)))
```

It's easy to write functions for translating between normal and curried versions.

```
(define (currify f)
  (lambda (x)
    (lambda (y)
      (f x y))))
```

Typed version of that, with examples:

```
(: currify : (All (A B C) (A B -> C) -> (A -> (B -> C))))
;; convert a double-argument function to a curried one
(define (currify f)
  (lambda (x) (lambda (y) (f x y))))

(: add : Number Number -> Number)
(define (add x y) (+ x y))

(: plus : Number -> (Number -> Number))
(define plus (currify add))

(test ((plus 1) 2) => 3)
(test (((currify add) 1) 2) => 3)
(test (map (plus 1) '(1 2 3)) => '(2 3 4))
(test (map ((currify add) 1) '(1 2 3)) => '(2 3 4))
(test (map ((currify +) 1) '(1 2 3)) => '(2 3 4))
```

Usages — common with H.O. functions like map, where we want to *fix* one argument.

When dealing with such higher-order code, the types are very helpful, since every arrow corresponds to a function:

```
(: currify : (All (A B C) (A B -> C) -> (A -> (B -> C))))
```

It is common to make the `->` function type associate to the right, so you can find this type written as:

```
currify : (A B -> C) -> (A -> B -> C)
```

or even as

```
currify : (A B -> C) -> A -> B -> C
```

but that can be a little confusing…

# Using Higher-Order & Anonymous Functions

Say that we have a function for estimating derivatives of a function at a specific point:

```
(define dx 0.01)

(: deriv : (Number -> Number) Number -> Number)
;; compute the derivative of `f' at the given point `x'
(define (deriv f x)
  (/ (- (f (+ x dx)) (f x)) dx))

(: integrate : (Number -> Number) Number -> Number)
;; compute an integral of `f' at the given point `x'
(define (integrate f x)
  (: loop : Number Number -> Number)
  (define (loop y acc)
    (if (> y x)
      (* acc dx)
      (loop (+ y dx) (+ acc (f y)))))
  (loop 0 0))
```

And say that we want to try out various functions given some `plot` function that draws graphs of numeric functions, for example:

```
(plot sin)
```

The problem is that `plot` expects a single `(Number -> Number)` function — if we want to try it with a derivative, we can do this:

```
(: sin-deriv : Number -> Number)
;; the derivative of sin
(define sin-deriv (lambda (x) (deriv sin x)))
(plot sin-deriv)
```

But this will get very tedious very fast — it is much simpler to use an anonymous function:

```
(plot (lambda (x) (deriv sin x)))
```

we can even verify that our derivative is correct by comparing a known function to its derivative

```
(plot (lambda (x) (- (deriv sin x) (cos x))))
```

But it's still not completely natural to do these things — you need to explicitly combine functions, which is not too convenient. Instead of doing this, we can write H.O. functions that will work with functional inputs and outputs. For example, we can write a function to subtract functions:

```
(: fsub : (Number -> Number) (Number -> Number)
          -> (Number -> Number))
;; subtracts two numeric 1-argument functions
(define (fsub f g)
  (lambda (x) (- (f x) (g x))))
```

and the same for the derivative:

```
(: fderiv : (Number -> Number) -> (Number -> Number))
;; compute the derivative function of `f'
(define (fderiv f)
  (lambda (x) (deriv f x)))
```

Now we can try the same in a much easier way:

```
(plot (fsub (fderiv sin) cos))
```

More than that — our `fderiv` could be created from `deriv` automatically:

```
(: currify : (All (A B C) (A B -> C) -> (A -> B -> C)))
;; convert a double-argument function to a curried one
(define (currify f)
  (lambda (x) (lambda (y) (f x y))))

(: fderiv : (Number -> Number) -> (Number -> Number))
;; compute the derivative function of `f'
(define fderiv (currify deriv))
```

Same principle with `fsub`: we can write a function that converts a binary arithmetical function into a function that operates on unary numeric function. But to make things more readable we can define new types for unary and binary numeric functions:

```
(define-type UnaryFun  = (Number -> Number))
(define-type BinaryFun = (Number Number -> Number))

(: binop->fbinop : BinaryFun -> (UnaryFun UnaryFun -> UnaryFun))
;; turns an arithmetic binary operator to a function operator
(define (binop->fbinop op)
  (lambda (f g)
    (lambda (x) (op (f x) (g x)))))

(: fsub : UnaryFun UnaryFun -> UnaryFun)
;; functional pointwise subtraction
(define fsub (binop->fbinop -))
```

We can do this with anything — developing a rich library of functions and functionals (functions over functions) is extremely easy… Here's a pretty extensive yet very short library of functions:

```
#lang pl untyped

(define (currify f)
  (lambda (x) (lambda (y) (f x y))))
(define (binop->fbinop op)
  (lambda (f g)
    (lambda (x) (op (f x) (g x)))))
(define (compose f g)
  (lambda (x) (f (g x))))

(define dx 0.01)
(define (deriv f x)
  (/ (- (f (+ x dx)) (f x)) dx))
(define (integrate f x)
  (define over? (if (< x 0) < >))
  (define step  (if (< x 0) - +))
  (define add   (if (< x 0) - +))
  (define (loop y acc)
    (if (over? y x)
      (* acc dx)
      (loop (step y dx) (add acc (f y)))))
  (loop 0 0))

(define fadd (binop->fbinop +))
(define fsub (binop->fbinop -))
(define fmul (binop->fbinop *))
(define fdiv (binop->fbinop /))
(define fderiv     (currify deriv))
```

```
(define fintegrate (currify integrate))
;; ...
```

This is written in the “untyped dialect” of the class language, but it should be easy now to add the types.

Examples:

```
;; want to verify that `integrate' is the opposite of `deriv':
;;   take a function, subtract it from its derivative's integral
(plot (fsub sin (fintegrate (fderiv sin))))

;; want to magnify the errors? -- here's how you magnify:
(plot (compose ((currify *) 5) sin))

;; so:
(plot (compose ((currify *) 20)
               (fsub sin (fintegrate (fderiv sin)))))
```

## Side-note: “Point-Free” combinators

> *Forming functions without using `lambda` (or an implicit `lambda` using a `define` syntactic sugar) is called point-free style. It’s especially popular in Haskell, where it is easier to form functions this way because of implicit currying and a large number of higher level function combinators. If used too much, it can easily lead to obfuscated code.*

## This is not Runtime Code Generation

All of this is similar to run-time code generation, but not really. The only thing that `fderiv` does is take a function and store it somewhere in the returned function, then when that function receives a number, it uses the stored function and send it to deriv with the number. We could simply write deriv as what `fderiv` is — which is the *real* derivative function:

```
(define (deriv f)
  (lambda (x)
    (/ (- (f (+ x dx)) (f x)) dx)))
```

but again, this is not faster or slower than the plain `deriv`. However, there are some situations where we can do some of the computation on the first-stage argument, saving work from the second stage. Here is a cooked-to-exaggeration example — we want a function that receives two inputs `x`, `y` and returns `fib(x)*y`, but we must use a stupid `fib`:

```
(define (fib n)
  (if (<= n 1)
    n
    (+ (fib (- n 1)) (fib (- n 2)))))
```

The function we want is:

```
(define (bogus x y)
  (* (fib x) y))
```

If we currify it as usual (or just use `currify`), we get:

```
(define (bogus x)
  (lambda (y)
    (* (fib x) y)))
```

And try this several times:

```
(define bogus36 (bogus 36))
(map bogus36 '(1 2 3 4 5))
```

But in the definition of `bogus`, notice that `(fib x)` does not depend on `y` — so we can rewrite it a little differently:

```
(define (bogus x)
  (let ([fibx (fib x)])
    (lambda (y)
      (* fibx y))))
```

and trying the above again is much faster now:

```
(define bogus36 (bogus 36))
(map bogus36 '(1 2 3 4 5))
```

This is therefore not doing any kind of runtime code generation, but it *enables* doing similar optimizations in our code. A proper RTCG facility would recompile the curried function for a given first input, and (hopefully) automatically achieve the optimization that we did in a manual way.