Functions & First Class Function Values

[[[PLAI Chapter 4]]]

Allowing functions in our language

Now that we have a form for **local bindings**, which forced us to deal with proper substitutions and everything that is related, we can get to functions. The concept of a function is itself very close to substitution, and to our `**with**' form. For example, consider the expression:

```
{with {x 5}
  {* x x}}
```

Here, the "**{* x x}**" body is itself parameterized over some value for `**x**'. If we take this expression and take out the "**5**", we're left with something that has all of the necessary ingredients of a function - a bunch of code that is parameterized over some input identifier:

```
{with {x}
  {* x x}}
```

A new form - for (anonymous) functions

We only need to replace `**with**' and use another (proper) name that indicates that it's a function:

```
{fun {x}
  {* x x}}
```

Now we have **a new form** in our language, one that should have a function as its meaning.

A form for calling a function

As we have seen in the case of `**with**' expressions, we also need a new form to **use** (apply) these functions. We will use `**call**' for this, so that

```
{call {fun {x} {* x x}}
      5}
```

will be the same as the original `**with**' expression that we started with -- the `**fun**' expression is like the `**with**' expression with no value, and applying it on `**5**' is providing that value back:

```
{with {x 5}
  {* x x}}
```

So far, of course, this does not help much -- all we get is a way to use local bindings that is more verbose from what we started with. What we're really missing is a way to *__name__* these functions. If we get the right evaluation rules, we can evaluate a `**fun**' expression to

some value -- which will allow us to bind it to a variable using `**with**'. Something like this:

```
{with {sqr {fun {x} {* x x}}}
  {+ {call sqr 5}
     {call sqr 6}}}
```

In this expression, we say that `**x**' is the **formal parameter** (or **argument**), and the `**5**' and `**6**' are **actual parameters** (sometimes abbreviated as **formals** and **actuals**).

Implementing First Class Function Values

[[[PLAI Chapter 6 (uses some stuff from ch. 5, which we do later)]]]

We have a simple plan, but it is directly related to how functions are going to be used in our language. We know that **{call {fun {x} E1} E2}** is equivalent to a `**with**' expression, but the **new thing here** is that we do allow writing just the **{fun ...}** expression by itself, and therefore we need to have some meaning for it.

Semantics of 'fun' expressions:
The meaning (or the value) of this expression should roughly be -- "an expression that **needs a value** to be plugged in for `**x**'". In other words, our language will have these new kinds of values that contain an expression to be evaluated later on.

There are three basic approaches that classify programming languages in relation to how the deal with functions:

1. **First order:** functions are not real values. They cannot be used or returned as values by other functions. This means that they cannot be stored in data structures. This is what most "conventional" languages used to have in the past.

   An example of such a language is the Beginner Student language that is used in HtDP, where the language is intentionally **first-order** to help students write correct code (at the early stages where using a function as a value is usually an error). It's hard to find practical modern languages that fall in this category.

2. **Higher order:** functions can receive and return other functions as values. This is what you get with C.

3. **First class:** functions are values with all the rights of other values. In particular, they can be supplied to other functions, returned from functions, stored in data structures, and new functions can be created at run-time. (And most modern languages have first class functions.)

First Class Functions (cont.)

The last category is the most interesting one. Here is an analogy that may give you some intuition:
Back in the old days, complex expressions were not first-class in that they could not be freely composed. This is still the case in machine-code: as we've seen earlier, to compute an expression such as

```
(-b + sqrt(b^2 - 4*a*c)) / 2a
```

You have to do something like this:

```
x = b * b
y = 4 * a
y = y * c
x = x - y
x = sqrt(x)
y = -b
x = y + x
y = 2 * a
s = x / y
```

In other words, every intermediate value needs to have its own name. But with proper ("high-level") programming languages (at least most of them...) you can just write the original expression, with no names for these values.

With first-class functions something similar happens -- it is possible to have complex expressions that consume and return functions, and they **do not need to be named.**

What we get with our `**fun**' expression (if we can make it work) is exactly this: **it generates a function, and you can choose to either bind it to a name, or not. The important thing is that the value exists independently of a name.**

This has a major effect on the "personality" of a programming language as we will see. In fact, just adding this feature will make our language much more advanced than languages with just higher-order or first-order functions.

Quick Example: the following is working JavaScript code that uses first class functions.

```
function foo(x) {
  function bar(y) { return x + y; }
  return bar;
}
function main() {
  var f = foo(1);
  var g = foo(10);
  alert(">> "+ f(2) + ", " + g(2));
}
```

Note that the above definition of `**foo**' does ***not*** use an *anonymous* "**lambda expression**" -- in Racket terms, it's translated to

```
(define (foo x)
  (define (bar y) (+ x y))
  bar)
```

The returned function is not anonymous, but it's not really named either: the `**bar**' name is bound only inside the body of `**foo**', and outside of it that name **no longer exists** since it's not its scope.

GCC includes extensions that allow internal function definitions, but it still does not have first class functions -- trying to do the above is broken:

```
#include <stdio.h>
typedef int(*int2int)(int);
int2int foo(int x) {
  int bar(int y) { return x + y; }
  return bar;
}
int main() {
  int2int f = foo(1);
  int2int g = foo(10);
  printf(">> %d, %d\n", f(2), g(2));
}
```

The FLANG Language

We will now extend our language to support first-class functions as well as everything we already supported (i.e., arithmetic expressions and with expressions). Thus, we change the language's name from **WAE** to **FLANG**.
Now for the implementation -- we call this new language FLANG.

BNF for 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]
  [Call FLANG FLANG])
```

A parser for FLANG

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

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

Evaluating FLANG expressions

Substitution rules for fun and call expressions

We also need to patch up the **substitution function** to deal with these things.

1. The scoping rule for the new '**fun**' form should consider an expression like **{fun {x} {* x x}}**. Consider a substation where we wish to substitute instances of **x** with a value **v** in this expression – we definitely DON'T want to touch the inner instances of x in the body **{* x x}** as these should be bound to the formal parameter of the function. Thus, unsurprisingly, the scoping rule here is **similar to the rule of `with'** (except that there is no extra expression now).
2. 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)))]))
```

Evaluating FLANG expressions – the 'eval' function

Question: what should be the type of the returned value of 'eval'?
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 incorporated functions:** We only had number values and we used Racket numbers to represent them.

- **Now that we incorporated functions:** We have **two** kinds of values –
  1. Numbers, and
  2. 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 (upon demand). 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.

Our solution: 'eval' will return FLANG
To accommodate such a delay effect, we need some way to hold the following information about a function:
- **The body expression** that needs to be evaluated *later* when the function is called.
- **The identifier's name (Formal parameters)** that should be replaced with the **actual input** (actual parameters) to the function call.

We observe that our abstract syntax object (i.e., a **FLANG** type object) contains exactly that. Hence, to represent a function, we can simply leave the **FLANG** describing it, as is. That is, **to evaluate a (Fun …) FLANG AST, simply return its own syntax object** as its value.

To be consistent with our treatment of *number* values, 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.

Examples:
The above 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.

Evaluation rules

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

The close relation between `call' expressions and `with' expression

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

```
eval({call E1 E2})
                = eval({with {x E2} Ef}) if eval(E1) = {fun {x} Ef}
                = 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: we now have two kinds of values, so we need to check the arithmetic operation's arguments too:

```
eval({+ E1 E2}) = eval(E1) + eval(E2)
                    if eval(E1) & eval(E2) are numbers
                    otherwise error!
...
```

The corresponding code is:

```
(: eval : FLANG -> FLANG)                          ; <- note return type
;; evaluates FLANG expressions by reducing them to *expressions*
(define (eval expr)
  (cases expr
    [(Num n) expr]                                 ; <- change here
    [(Add l r) (arith-op + (eval l) (eval r))]     ; <- change here
    [(Sub l r) (arith-op - (eval l) (eval r))]     ; <- change here
    [(Mul l r) (arith-op * (eval l) (eval r))]     ; <- change here
    [(Div l r) (arith-op / (eval l) (eval r))]     ; <- change here
    [(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)]))
```

The `arith-op' function:
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 H.O type)
(define (arith-op op expr1 expr2)
  (: Num->number : FLANG -> Number)
  (define (Num->number e)
    (cases e
      [(Num n) n]
      [else (error 'arith-op "expects a number, got: ~s" e)]))
  (Num (op (Num->number expr1) (Num->number expr2))))
```

The `run' function:

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(Ef[eval(E2)/x]) if eval(E1)={fun {x}Ef}
                        = 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)))

(: 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)))]))

(: 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 expr1 expr2)
  (: Num->number : FLANG -> Number)
  (define (Num->number e)
    (cases e
      [(Num n) n]
      [else (error 'arith-op "expects a number, got: ~s" e)]))
  (Num (op (Num->number expr1) (Num->number expr2))))

(: eval : FLANG -> FLANG)
;; evaluates FLANG expressions by reducing them to *expressions*
(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) ; <- 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)]))



(: 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)
```

A problem:

There is still a problem with this version.

1. 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?

2. Second question: what *should* happen if we evaluate these:

```
(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}")
```

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

The following simple fix takes care of this:

```
(: eval : FLANG -> FLANG)
;; evaluates FLANG expressions by reducing them to *expressions*
(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)
     (let ([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:

```
---<<<FLANG>>>-------------------------------------------------------


;; 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(Ef[eval(E2)/x]) if eval(E1)={fun {x}
Ef}
                       = 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)))

(: 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)))]))
```

```
(: 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 expr1 expr2)
  (: Num->number : FLANG -> Number)
  (define (Num->number e)
    (cases e
      [(Num n) n]
      [else (error 'arith-op "expects a number, got: ~s" e)]))
  (Num (op (Num->number expr1) (Num->number expr2))))

(: eval : FLANG -> FLANG)
;; evaluates FLANG expressions by reducing them to *expressions*
(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)
     (let ([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 {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)
```
