Programming Languages

Oran Danon

Contents

1 First Lecture 1

2 Second Lecture 1
- 2.1 List operations – Predicates . . . 2
- 2.2 Tail Recursion . . . 2
- 2.3 side-effect . . . 3

3 BNF & Grammers 4

4 Fourth Lecture – Simple Parsing 4

5 Fifth Lecture – Evaluation 5

6 Sixth Lecture – Binding & Substitution 6

7 Eighth lecture - First class functions 6
- 7.1 The Equivalence Between With & Call . . . 6

8 The complete code – Eighth lecture 7

9 Ninth lecture 10
- 9.1 Substitution Cache – Dynamic Programming . . . 10
  - 9.1.1 Formal Rules . . . 10
  - 9.1.2 Examples . . . 11
- 9.2 Static Programming - environment model . . . 12
  - 9.2.1 Formal Rules . . . 12
  - 9.2.2 Enviorment model – The complete code . . . 14

1 First Lecture

2 Second Lecture

Definition 2.1 – Syntactic sugar. A way which is more convenient to preform an action which already implemented in the language.

Put another words, a feature which does not extends the functionality of the language but provide a cleaner interface.
An example to syntactic sugar is $x++$.

- Anything except from $false \equiv \#f$ is true.

- Racket is a functional language, this means that everything has a value.
- The last bullet explains why no return statement should be define at racket.
- When using **cond** one must insure there is a case for 'else', usually we will do so by the keyword else but one can replace this keyword with any true expression.
- The difference between **match** and **cases** is that **match** is used for type who were defined in the language, and **cases** is used for types that we defined.

Definition 2.2 – List. A list defined as follows:

1. An empty list (i.e. 'null', 'empty' or '()).
2. A pair such that the second element is a list.

Please notice that list is a recursive structure, thus function that operates on lists should be recursive functions. We use **cons** in order to create a pair, and (**list-ref** `lst k`) – in order to return the kth element in the list `lst`, note that the indexes in the list are starting to enumerate from zero.

2.1 List operations – Predicates

- **null?** – true only for the empty list.
- **pair?** – true for any **cons** cell.
- **list?** – check whether the following argument is a list as defined above.

2.2 Tail Recursion

Definition 2.3 – Tail Recursion. A recursive function is tail recursive if the final result of the recursive call is the final result of the function itself.

Note: If the result of the recursive call must be further processed (say, by adding 1 to it, or consing another element onto the beginning of it), it is not tail recursive.

Assume we want to calculate $S(t) := \sum_{i=1}^{t} i$ using recursion, the following is an trivial way for doing so:

```
public static int sum(int t)
{
    if (t==1)
    {
        return t;
    }
    else
    {
        return t + sum(t-1);
    }
}
```

However the recursion tree would look as follows:

```
    sum(5)
   /    \
5  +  sum(4)
        /   \
5  + (4  +  sum(3))
             /   \
5 + (4 + (3 + sum(2)))
                 /    \
5 + (4 + (3 + (2 + sum(1))))
                    /   \
5 + (4 + (3 +   ( 2  +  1) ) )
```

And this does not settle with our definition for tail recursion, as the we should end all the recursive calls in order to compute the all sum. We shall use the following tail-recursive function:

```java
    public static tail_recursive_sum(int t, int sum = 0) {
    if (t==0) {
        return sum;
    } else {
        return tail_recursive_sum(t-1, sum+x);
    }
}
```

Which result the following recursive calls:

```
tailrecsum(5, 0) =>
tailrecsum(4, 5) =>
tailrecsum(3, 9) =>
tailrecsum(2, 12)=>
tailrecsum(1, 14)=>
tailrecsum(0, 15)=>
15
```

One can verify that this is indeed a tail recursive function as the recursive call calls only for the function itself, and no other external calculation are done. Another example for tail recursive is the following function which computes $n!$.

```java
    public static int factorial(int n, int result) {
        if (n == 1)
            return result;
        else
            return factorial(n-1, n*result);
}
```

2.3 side-effect

Definition 2.4 – side-effect. An operation has side effect if it modifies some state of a variable outside its local environment, put another words, there is an observable effect besides returning a value (the main effect) to the invoker of the operation.

Consider the following example:

```java
public class SideEffect
{
   int global_variable = 0;
   int square(int x)
   {
     global_variable = 1;
     return x*x;
   }
   public static void main(String[] args)
   {
       int res;
       global_variable = 0;
       res = square(5);       // This cause a side effect on global_variable
       res += global_variable;
       System.out.println("5*5=" + res);
   }
}
```

If one would take a look on the main function only he should except that the result would be $5^2 = 25$, but since square has a side effect which effects the global variable, the result in 26.

3 BNF & Grammers

Every language has a syntax, which on can view as the grammar of the language. We will define the grammar of the language using BNF (Backus-Naur Form).

Definition 3.1 – Ambiguity. When an expression can be derived by different derivation trees, the grammar is called ambiguous.

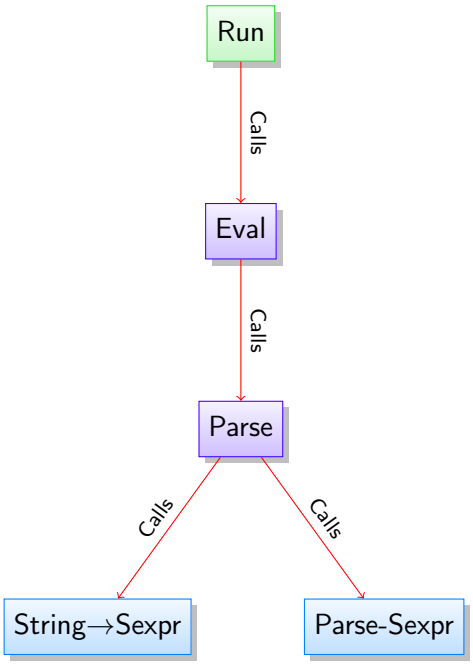
4 Fourth Lecture – Simple Parsing

Definition 4.1 – Parser. Given a concrete syntax we would like to convert it into abstract syntax tree.

This may look a little weird, but think that you have many synthetics sugar, you want to convert them to the same abstract tree.

Info:

1. The type sexpr defined recursively as follows:
   - Num, Symbol, are sexpr.
   - Any list of Sexpr is an Sexpr.
2. We know that there is a function which convert a string to string-expression (which is either a list, symbols, or numbers) which is called `string->sexpr`. Basically, it converts each `{}` to `()` and splits between `{}`s, finally it categorize each element to either list, symbol, or number. The advantage is followed from the abstraction of this data structure i.e. each element is an expression in contrast to a string.
3. Then our goal is to parse the string-expressions into an abstract syntax tree (here-forth AST). In order to do this properly we do so in several steps.
   a) We convert the entire string into a list of string-expressions using `string->sexpr`.
   b) We create a function that convert a single string-expression into an AST. Usually this function is called `parse-sexpr`.
   c) We invoke `parse-sexpr` on each string-expression , as an convention this will be done by `parse` function.
4. `eval` will evaluate an abstract syntax tree .
5. `run` will run all the entire project by calling `(eval (parse project_string))`. Please note that `eval` now operates on the entire abstract syntax tree that represent the whole program.



5 Fifth Lecture – Evaluation

As we mention above evaluation deals with the meaning of the abstract syntax tree .

Definition 5.1 – Compositionality. The meaning of a compound expression is a function of the meanings of its parts.

An example is as follows

```
<NUM> ::= <digit> | <NUM> <digit>
eval(<NUM> <digit>) = eval(<NUM>) * 10 + eval(<digit>)
```

6 Sixth Lecture – Binding & Substitution

We would like to get rid of repeated expressions. This come from several reasons:

1. **Redundant Computation** – We would like not to compute the same expression twice.
2. **Simplicity**.
3. **DRY** – Don't repeat yourself principle.

Definition 6.1 – Binding Instance. A binding instance is an identifier for an expression or variable.

Definition 6.2 – Scope. The region in which instances of an identifier refers to the binding instance.

Definition 6.3 – Bound Instance. An instance of an identifier is bound if it is contained within the scope of a binding instance of its name.

Definition 6.4 – Free Instance. An identifier that is not contained in the scope of any binding instance of its name is said to be free.

Definition 6.5 – `e[v/i]`. Substitute the identifier 'i' with the expression 'v' in the expression 'e'. Put another words, replace all instances of 'i' that are free in 'e' with the expression 'v'.

7 Eighth lecture - First class functions

7.1 The Equivalence Between With & Call

The following are equivalent:

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

8 The complete code – Eighth lecture

```
-------------------------------------<<<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)
----------------------------------------------------------------------
```

9 Ninth lecture

9.1 Substitution Cache – Dynamic Programming

9.1.1 Formal Rules

```
---------------------------------------------------------------------
Lookup rules:
  lookup(x,empty-subst)      = error!
  lookup(x,extend(x,E,sc))   = E
  lookup(x,extend(y,E,sc))   = lookup(x,sc)  if `x' is not `y'

Evaluation rules:
    eval(N,sc)                 = N
    eval({+ E1 E2},sc)         = eval(E1,sc) + eval(E2,sc)
    eval({- E1 E2},sc)         = eval(E1,sc) - eval(E2,sc)
    eval({* E1 E2},sc)         = eval(E1,sc) * eval(E2,sc)
    eval({/ E1 E2},sc)         = eval(E1,sc) / eval(E2,sc)
    eval(x,sc)                 = lookup(x,sc)
    eval({with {x E1} E2},sc)  = eval(E2,extend(x,eval(E1,sc),sc))
    eval({fun {x} E},sc)       = {fun {x} E}
    eval({call E1 E2},sc)
         = eval(Ef,extend(x,eval(E2,sc),sc))
                            if eval(E1,sc) = {fun {x} Ef}
         = error!           otherwise

---------------------------------------------------------------------
```

9.1.2 Examples

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

  (define x 123)

  (define (getx) x) ;; return the last binding to x

  (define (bar1 x) (getx)) ;; here the last binding is the formal param
  (define (bar2 y) (getx)) ;; here the last binding is the global var

  (test (getx) => 123)
  (test (let ([x 456]) (getx)) => 456)
  (test (getx) => 123)
  (test (bar1 999) => 999)
  (test (bar2 999) => 123)

  (define (foo x)
     (define (helper) (+ x 1))
      helper) ;; foo returns a function that returns x + 1 according to the last
       ↪  binding of x, the formal variable does not influence anything.
  (test ((foo 0)) => 124) ;; here the last binding to x is 123.

  ;; and *much* worse:
  (define (add x y) (+ x y))
  (test (let ([+ *]) (add 6 7)) => 42)
;; Here we change the binding of + to be * and the result would be 6*7 as the
↪  programmer excepts for 6+7.
  ----------------------------------------------------------------------
```

9.2 Static Programming - environment model

9.2.1 Formal Rules

```
  ----------------------------------------------------------------------
  eval(N,sc)                 = N
  eval({+ E1 E2},sc)         = eval(E1,sc) + eval(E2,sc)
  eval({- E1 E2},sc)         = eval(E1,sc) - eval(E2,sc)
  eval({* E1 E2},sc)         = eval(E1,sc) * eval(E2,sc)
  eval({/ E1 E2},sc)         = eval(E1,sc) / eval(E2,sc)
  eval(x,sc)                 = lookup(x,sc)
  eval({with {x E1} E2},sc) = eval(E2,extend(x,eval(E1,sc),sc))
  eval({fun {x} E},sc)       = <{fun {x} E}, sc>
  eval({call E1 E2},sc1)
           = eval(Ef,extend(x,eval(E2,sc1),sc2))
                              if eval(E1,sc1) = <{fun {x} Ef}, sc2>
           = error!           otherwise


;;The Old rules -for reference
#|eval({fun {x} E},sc)       = {fun {x} E}
  eval({call E1 E2},sc)
           = eval(Ef,extend(x,eval(E2,sc),sc))
                              if eval(E1,sc) = {fun {x} Ef}
           = error!           otherwise
           |#

                 Final Version
  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},env)
           = eval(Ef,extend(x,eval(E2, env),fenv))
                              if eval(E1,env) = <{fun {x} Ef}, fenv>
           = error!           otherwise
```

9.2.2 Enviorment model – The complete code

```
-----------------------<<<FLANG-ENV>>>-----------------------------------
;; The Flang interpreter, using environments
#lang pl
#|
<FLANG> ::= <num>
            | { + <FLANG> <FLANG> }
            | { - <FLANG> <FLANG> }
            | { * <FLANG> <FLANG> }
            | { / <FLANG> <FLANG> }
            | { with { <id> <FLANG> } <FLANG> }
            | <id>
            | { fun { <id> } <FLANG> }
            | { call <FLANG> <FLANG> }

Evaluation rules:
  eval(N,env)                 = N
  eval({+ E1 E2},env)         = eval(E1,env) + eval(E2,env)
  eval({- E1 E2},env)         = eval(E1,env) - eval(E2,env)
  eval({* E1 E2},env)         = eval(E1,env) * eval(E2,env)
  eval({/ E1 E2},env)         = eval(E1,env) / eval(E2,env)
  eval(x,env)                 = lookup(x,env)
  eval({with {x E1} E2},env) = eval(E2,extend(x,eval(E1,env),env))
  eval({fun {x} E},env)       = <{fun {x} E}, env>
  eval({call E1 E2},env1)
           = eval(Ef,extend(x,eval(E2,env1),env2))
                              if eval(E1,env1) = <{fun {x} Ef}, env2>
           = error!           otherwise
|#

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

(: parse-sexpr : Sexpr -> FLANG)
;; to convert s-expressions into FLANGs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        (With name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(cons 'fun more)
     (match sexpr
       [(list 'fun (list (symbol: name)) body)
        (Fun name (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

```
(: parse : String -> FLANG)
;; parses a string containing a FLANG expression to a FLANG AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))

;; Types for environments, values, and a lookup function

(define-type ENV
  [EmptyEnv]
  [Extend Symbol VAL ENV])

(define-type VAL
  [NumV Number]
  [FunV Symbol FLANG ENV])

(: 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))]))

(: arith-op : (Number Number -> Number) VAL VAL -> VAL)
;; gets a Racket numeric binary operator, and uses it within a NumV
;; wrapper
(define (arith-op op val1 val2)
  (: NumV->number : VAL -> Number)
  (define (NumV->number v)
    (cases v
      [(NumV n) n]
      [else (error 'arith-op "expects a number, got: ~s" v)]))
  (NumV (op (NumV->number val1) (NumV->number val2))))

(: eval : FLANG ENV -> VAL)
;; evaluates FLANG expressions by reducing them to values
(define (eval expr env)
  (cases expr
    [(Num n) (NumV n)]
    [(Add l r) (arith-op + (eval l env) (eval r env))]
    [(Sub l r) (arith-op - (eval l env) (eval r env))]
    [(Mul l r) (arith-op * (eval l env) (eval r env))]
    [(Div l r) (arith-op / (eval l env) (eval r env))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (Extend bound-id (eval named-expr env) env))]
    [(Id name) (lookup name env)]
    [(Fun bound-id bound-body)
     (FunV bound-id bound-body env)]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr env)])
       (cases fval
         [(FunV bound-id bound-body f-env)
          (eval bound-body
                (Extend bound-id (eval arg-expr env) f-env))]
         [else (error 'eval "`call' expects a function, got: ~s"
                      fval)]))]))

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

```
  ;; tests
  (test (run "{call {fun {x} {+ x 1}} 4}")
        => 5)
  (test (run "{with {add3 {fun {x} {+ x 3}}}
                 {call add3 1}}")
        => 4)
  (test (run "{with {add3 {fun {x} {+ x 3}}}
                 {with {add1 {fun {x} {+ x 1}}}
                   {with {x 3}
                     {call add1 {call add3 x}}}}}")
        => 7)
  (test (run "{with {identity {fun {x} x}}
                 {with {foo {fun {x} {+ x 1}}}
                   {call {call identity foo} 123}}}")
        => 124)
  (test (run "{with {x 3}
                 {with {f {fun {y} {+ x y}}}
                   {with {x 5}
                     {call f 4}}}}")
        => 7) ;; the example we considered for subst-caches
  (test (run "{call {with {x 3}
                       {fun {y} {+ x y}}}
                     4}")
        => 7)
  (test (run "{call {call {fun {x} {call x 1}}
                          {fun {x} {fun {y} {+ x y}}}}
                     123}")
        => 124)
----------------------------------------------------------------------
```