
Lecture #1 (plus practice session)

Administrative stuff.

Lecturer: Eran Omri
Email address: omrier@ariel.ac.il
Room: 11.2.11
Office hours: TBD

References:
Main:
Eli Barzilay's course -- http://pl.barzilay.org/
Introduction to Programming Languages (CS4400/CS5400)
Based on -- PLAI book --
http://pl.barzilay.org/plai.pdf

Also good to know:
Mira Blalaban's course -- http://www.cs.bgu.ac.il/~ppl122/Main
And her book -- http://www.cs.bgu.ac.il/~mira/ppl-book.pdf

Language Installation Instructions:

1. Install Racket (and DrRacket)
2. Run DrRacket
3. Choose File > Install .plt file
4. In the "web" tab, enter:
http://pl.barzilay.org/pl.plt

Course overview

- General plan for how the course will go.
  - We will be designing a programming language
  - We will start with a very simple language – for performing simple arithmetic operations.
  - With time, we will expand the language to enable programmers to perform more sophisticated tasks.
- The goal of the above process is to take the point of view of programming languages designers (while keeping in mind that of programmers).
  - Through that we hope to understand some of the issues that come up when designing programming languages.
  - Should we all be designers? No, but you would like to understand what lies behind many decisions (don't want to feel like taking your car to the mechanic and be overcharged)
- Why should we care about programming languages? (Any examples of big projects *without* a little language?)

PLAI Chapter 1

What are the essentials of a language?

- **Syntax** -- the formal rules of how to construct programs (phrases).
- **Semantics** – The true meaning of things.

An important difference between syntax and semantics:

A good way to think about this is the difference between the string "42" stored in a file somewhere

(two ASCII values), and the number 42 stored in memory (in some representation). You could also continue with the above example: there is nothing wrong with **"robbery"** – it's just a word, but **robbery** is something you'll go to jail for.

* How important is each of these?

<u>Syntax:</u> Assume that an array called a was initialized to have 15 cells.

- Compare:

```
a[25]+5                    (Java:)

(+ (vector-ref a 25) 5)    (Racket:)

a[25]+5                (JavaScript:)

a[25]+5                   (Python:)

$a[25]+5                  (Perl:)

a[25]+5                  (C:)

a[25]+5                 (ML:)
```

<u>Syntax:</u>

- Compare:

```
a[25]+5                    (Java: exception)

(+ (vector-ref a 25) 5)    (Racket: exception)

a[25]+5                (JavaScript: NaN or undefined)

a[25]+5                   (Python: exception)
```

```
$a[25]+5                    (Perl: 5)

a[25]+5                    (C: <<<BOOM>>>)

a[25]+5                   (ML: not an array ref at all)
    -> syntax is mostly in the cosmetics
department.
    -> semantics is the real thing.
```

How should we talk about semantics?

- A few well-known formalisms for semantics.
- We will use programs to explain semantics: the best explanation *is* a program.
- Ignore possible philosophical issues with circularity (but be aware of them). (Actually, they are solved: Scheme has a formal explanation that can be taken as a translation from Scheme to logic, which means that things that we write can be translated to logic.)
- We will use Racket for many reasons (syntax, functional, practical, simple, formal, statically typed, environment).

The evaluation function that Racket uses is actually - a function that takes a **piece of syntax** and returns (or executes) **its semantics**.

Introduction to Racket

* General layout of the parts of Racket:

  - The Racket language is (mostly) in the Scheme family, or more generally in the Lisp family;

  - Racket: the core language implementation (language and runtime), written mostly in C;

  - The actual language(s) that are available in Racket have lots of additional parts that are implemented in Racket itself;

  - GRacket: a portable Racket GUI extension, written in Racket too;

  - DrRacket: a GRacket application (also written in Racket);

  - Our language(s)...

* **Documentation**: the Racket documentation is your friend (But beware that some things are provided in different forms from different places)

Side-note: (programming languages are alive and changing)

E.W. DIJKSTRA

"Goto Statement Considered Harmful."

This paper tries to convince us that **the well-known goto statement should be eliminated from our programming languages**…

…I doubt that any real-world **program will ever be written in such a style**… Publishing this **would waste valuable paper**… **30 years from now, the goto will still be alive and well** and used as widely as it is today.

Quick Introduction to Racket

Racket syntax... Similar to other Sexpr-based languages.

Reminder: the parentheses can be compared to C/etc function cal parens - they always mean that some function is applied:
Reminder: In C - foo(int x);

This is the reason why (+ (1) (2)) won't work: if you use C syntax that would be "+(1(), 2())" but "1" isn't a function so "1()" is an error.

`define' expressions: are used for creating new **bindings**.

Note: do not try to use them to change values. For example, you should not try to write something like (define x (+ x 1)) in an attempt to mimic `x = x+1'. It will not work.

End of class #1

Introduction to Typed Racket

The plan:

- Racket Crash Course
- Typed Racket and PL Racket
- Differences with the text
- Some PL Racket Examples

The Racket "Ecosystem"

- The Racket language is (mostly) in the Scheme family, or more generally in the Lisp family;
- Racket: the core language implementation;
- DrRacket: a Racket application, and IDE
- Our language(s)...
- Documentation: the Racket documentation is your friend (But beware that some things are provided in different forms from different places). `http://docs.racket-lang.org` is a good starting point

Getting started

- Find a machine with DrRacket installed (e.g. the linux lab).
- Follow the instructions at `http://www.cs.unb.ca/~bremner/teaching/cs3613/racket-setup` to customize DrRacket
- First part is based on `http://www.cs.unb.ca/~bremner/teaching/cs3613/racket/quick-ref.rkt`

Starting files

- Typed Racket files start like this:

```
#lang typed/racket
;; Program goes here.
```

but we will use a variant of the Typed Racket language, which has a few additional constructs:

```
#lang pl
;; Program goes here.
```

Racket Expressions

Racket is an expression based language. We can program by interactively evaluating expressions.

3

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Built-in atomic data

;; Booleans

true
false
#t ; another name for true, and the way
   ; it prints
#f ; ditto for false

; (:print-type #f)
```

4

```
;; Numbers

1
0.5

1/2 ; this is a literal fraction, not a
    ; division operation

1+2i ; complex number

; (:print-type 1/2)
```

5

```
;; Strings

"apple"
"banana cream pie"
```

6

```
;; Symbols

'apple
'banana-cream-pie
'a->b
'#%$^@*&?!
;(:print-type 'apple)
```

7

```
;; Characters

#\a
#\b
#\A
#\space  ; same as #\  (with a space after \)
(string #\a #\b)
```

Prefix Expressions

Racket is a member of the lisp (scheme) family and uses prefix notation (and many parentheses).

8

```
;; Procedure application
;;  (<expr> <expr>*)

(not true)                    ; => #f

(+ 1 2)                       ; => 3
(< 2 1)                       ; => #f
(= 1 1)                       ; => #t

(string-append "a" "b")       ; => "ab"
(string-ref "apple" 0)        ; => #\a
```

9

```
(eq? 'apple 'apple)          ; Object identity
(eq? 'apple 'orange)         ; => #f
(eq? "apple" "apple")        ; => depends

(equal? "apple" "apple")     ; => Content equality
(string=? "apple" "apple")   ; => ... for strings

(null? null)                 ; => #t

(number? null)               ; => #f
(number? 12)                 ; => #t
```

Comments


```
;; This is a comment that continues to
;; the end of the line.
; One semi-colon is enough.

;; A common convention is to use two
;; semi-colons for multiple lines of
;; comments, and a single semi-colon when
;; adding a comment on the same line as
;; code.

#| This is a block comment, which starts
   with '#|' and ends with a '|#'.
|#

#;(comment out a single form)
```

Conditionals

11

```
;;   (cond
;;     [<expr> <expr>]*)
;;   (cond
;;     [<expr> <expr>]*
;;     [else <expr>])

(cond                          ;
  [(< 2 1) 17]                 ;
  [(> 2 1) 18])                ; => 18

;; second expression not evaluated
(cond
  [true 8]                     ;
  [false (* 'a 'b)])           ; => 8
```

```
;; any number of cond-lines allowed
(cond
  [(< 3 1) 0]                ;
  [(< 3 2) 1]                ;
  [(< 3 3) 2]                ;
  [(< 3 4) 3]
  [(< 3 5) 4])               ; => 3

;; else allowed as last case
(cond
  [(eq? 'a 'b) 0]            ;
  [(eq? 'a 'c) 1]            ;
  [else 2])                  ; => 2

(cond
  [(< 3 1) 1]
  [(< 3 2) 2])               ; => prints nothing

(void)                       ; => prints nothing
```

Racket Lists

13

```
;; Building lists
null                          ; => '()

(list 1 2 3)                   ; => '(1 2 3)
(cons 0 (list 1 2 3))          ; => '(0 1 2 3)

(cons 1 null)                 ; => '(1)
(cons 1 '())                   ; => '(1)
(cons 'a (cons 2 null))       ; => '(a 2)

(list 1 2 3 null)             ; => '(1 2 3 ())
```

```
;; Functions on lists

(append (list 1 2) null) ; => '(1 2)
(append (list 1 2)
        (list 3 4))      ; => '(1 2 3 4)
(append (list 1 2)
        (list 'a 'b)
        (list true))     ; => '(1 2 a b #t)

(first (list 1 2 3))     ; => 1
(rest (list 1 2 3))      ; => '(2 3)
(first (rest (list 1 2))) ; => 2

(list-ref '(1 2 3) 2)    ; => 3
```

Defining Constants and Procedures/Functions

15

```
(define PI 3.14)

(define (double x)
        (list x x))

(define (Not a)
  (cond
    [a #f]
    [else #t]))

(: length : (Listof Any) -> Natural)
(define (length l)
 (cond
   [(null? l) 0]
   [else (add1 (length (rest l)))]))
```

Racket and Typed Racket

- So far almost everything we saw is (un-typed) Racket.
  Typed racket adds *type annotations* and a *type checker*.
- Type annotations and *type inference* reduce the amount declarations needed.
  Everything we saw so far is also validly typed.

Types of Typing

- Who has used a (statically) typed language?
- Who has used a typed language that's not Java?
- Who has used a dynamically typed language?

Why types?

- Types help structure programs.
- Types provide enforced and mandatory documentation.
- Types help catch errors.

Why Typed Racket?

- Racket it is an excellent language for experimenting with programming languages.
- Types are an important programming language feature; Typed Racket will help us understand them.
- Data-first design. The structure of your program is derived from the structure of your data.
  Types make this pervasive – we have to think about our data before our code.
- A language for describing data; Having such a language means that we get to be more precise and more expressive talking about code.

Definitions with type annotations

16

```
(define PI 3.14159)
(* PI 10)                    ; => 31.4159

;; (: <id> <type>)
(: PI2 Real)
(define PI2 (* PI PI))

(: circle-area : Number -> Number )
(define (circle-area r)
  (* PI r r))
(circle-area 10)             ; => 314.159
```

17

```
(: f : Number -> Number)
(define (f x)
   (* x (+ x 1)))

;; Less commonly in this course:
(define: (f2 [x : Number]) : Number
    (* x (+ x 1)))
```

Defining datatypes


```
;;  (define-type <id>
;;    [<id> <type>*]*)

(define-type Animal
  [Snake   Symbol Number Symbol]
  [Tiger   Symbol Number])

(Snake 'Slimey 10 'rats)
(Tiger 'Tony 12)
```

19

```
#;(Snake 10 'Slimey 5)
; => compile error: 10 is not a symbol

(Animal? (Snake 'Slimey 10 'rats)) ; => #t
(Animal? (Tiger 'Tony 12)) ; => #t
(Animal? 10)               ; => #f
```

20

```
;; A type can have any number of variants:
(define-type Shape
  [Square  Number] ; Side length
  [Circle  Number] ; Radius
  [Triangle Number Number]) ; height width

(Shape? (Triangle 10 12)) ; => #t
```

Datatype case dispatch

21

```
;;   (cases  <expr>
;;     [(<id> <id>*) <expr>]*)
;;   (cases <expr>
;;     [<id> (<id>*) <expr>]*
;;     [else <expr>])
```

```
(cases (Snake 'Slimey 10 'rats)
  [(Snake n w f) n]
  [(Tiger n sc) n])
```

22

```
(: animal-name : Animal -> Symbol)
(define (animal-name a)
  (cases a
    [(Snake n w f) n]
    [(Tiger n sc) n]))

(animal-name (Snake 'Slimey 10 'rats))
; => 'Slimey

(animal-name (Tiger 'Tony 12)) ; => 'Tony

#;(animal-name 10)          ; => error: Type error
```

23

```
(: animal-weight : Animal -> (U Number #f))
(define (animal-weight a)
  (cases a
    [(Snake n w f) w]
    [else #f]))

(animal-weight (Snake 'Slimey 10 'rats))
(animal-weight (Tiger 'Tony 12))
```

Short Circuit And/Or


```
;;   (and <expr>*)
;;   (or <expr>*)

(and true true)              ; => #t
(and true false)             ; => #f
(and (< 2 1) true)           ; => #f
(and (< 2 1) (+ 'a 'b))      ; => #f short circuit
```

25

```
(or false true)                ; #t
(or false false)               ; #f

(and true true true true) ; => #t
(or false false false)    ; => #f

(and true 1)                   ; => 1

(or true (/ 1 0))
```

Local binding


```
;;  (let ([<id> <expr>]*) <expr>)

(let ([x 10]
      [y 11])
  (+ x y))

(let ([x 0])
  (let ([x 10]
        [y (+ x 1)])
    (+ x y)))

(let ([x 0])
  (let* ([x 10]
         [y (+ x 1)])
    (+ x y)))
```

First-class functions


```
;; Anonymous function:
;;  (lambda (<id>*) <expr>)
;;  (lambda: ( (<id> : <type>)* ) <expr>)

(lambda: [(x : Number)] (+ x 1))
;; => #<procedure>

;; Note the type annotation is not needed here

((lambda (x) (+ x 1)) 10) ; => 11
```

28

```
(define add-one
  (lambda: [(x : Number)]
    (+ x 1)))
(add-one 10)                    ; => 11
```

29

```
;; Similarly note here the inner lambda does
   not need annotation
(: make-adder : Number -> (Number -> Number))
(define (make-adder n)
  (lambda (m)
    (+ m n)))
(make-adder 8)                  ; => #<procedure>
(define add-five (make-adder 5))
(add-five 12)                   ; => 17
((make-adder 5) 12)             ; => 17
```

30

```
(map (lambda: [(x : Number)]
        (* x x))
     '(1 2 3))                ; => (list 1 4 9)

(andmap (lambda: [(x : Number)] (< x 10))
        '(1 2 3))             ; => #t
(andmap (lambda: [(x : Number)]
                  (< x 10))
        '(1 20 3))            ; => #f
```

31

```
;; The apply function may be useful eventually:

(: f :  Number Number Number -> Number)
(define (f a b c) (+ a (- b c)))
(define l '(1 2 3))

#;(f l)                    ; => error: f expects
   3 arguments
(apply f l)                ; => 0

;; apply is most useful with functions that
   accept any
;;  number of arguments:

(apply + '(1 2 3 4 5))     ; => 15
```

Examples


```
(: is-odd? : Number -> Boolean)
(define (is-odd? x)
  (if (zero? x)
      false
      (is-even? (- x 1))))

(: is-even? : Number -> Boolean)
(define (is-even? x)
  (if (zero? x)
      true
      (is-odd? (- x 1))))
(is-odd? 12)                    ; => #f
```

33

```
(: digit-num : Number -> (U Number String))
(define (digit-num n)
  (cond [(<= n 9)     1]
        [(<= n 99)    2]
        [(<= n 999)   3]
        [(<= n 9999)  4]
        [else         "a lot"]))
```

34

```
(: fact : Number -> Number)
(define (fact n)
  (if (zero? n)
    1
    (* n (fact (- n 1)))))
```

```
(: helper : Number Number -> Number)
(define (helper n acc)
  (if (zero? n)
    acc
    (helper (- n 1) (* acc n))))

(: fact : Number -> Number)
(define (fact n)
  (helper n 1))
```

```
(: fact : Number -> Number)
(define (fact n)
  (: helper : Number Number -> Number)
  (define (helper n acc)
    (if (zero? n)
      acc
      (helper (- n 1) (* acc n))))
  (helper n 1))
```

37

```
(: every? : (All (A) (A -> Boolean)
                     (Listof A) -> Boolean))
;; Returns true if all pass pred.
(define (every? pred lst)
  (or (null? lst)
      (and (pred (first lst))
           (every? pred (rest lst)))))
```

A parser for arithmetic


```
(: parse-sexpr : Sexpr -> AE)
;; to convert s-expressions into AEs
(define (parse-sexpr sexpr)
   (match sexpr
    [(number: n) (Num n)]
    [(list '+ left right)
     (Add (parse-sexpr left) (parse-sexpr
        right))]
    [(list '- left right)
     (Sub (parse-sexpr left) (parse-sexpr
        right))]
    [else (error 'parse-sexpr
                 "bad syntax in ~s" sexpr)]))
```

Differences with the text

- PL Racket uses types, not predicates, in define-type.


```
(define-type AE
  [Num Number]
  [Add AE AE])

;;  versus

; (define-type AE
;   [Num (n number?)]
;   [Add (l AE?) (r AE?)])
```

Fancier type examples

Typed Racket has unions, and gathers type information via predicates.

40

```
(: foo : (U String Number) -> Number)
(define (foo x)
  (if (string? x)
    (string-length x)
    ;; at this point it knows that `x' is not a
       string, therefore it
    ;; must be a number
    (+ 1 x)))
```

Statically typed languages are usually limited to "disjoint unions". For example, in Haskell you'd write:

41

```
data StrNum = Str String | Num Int

foo::StrNum -> Int
foo (Str string) = length string
foo (Num num) = num

-- And use it with an explicit constructor:

a = foo (Str "bar")
b = foo (Num 3)
```

Unions and Subtypes

- Typed Racket has a concept of subtypes In fact, the fact that it has (arbitrary) unions means that it must have subtypes too, since a type is always a subtype of a union that contains this type.
- Another result of this feature is that there is an 'Any' type that is the union of all other types.
- Consider the type of 'error': it's a function that returns a type of 'Nothing' – a type that is the same as an **empty** union: (U). This means that an 'error' expression can be used anywhere you want because it is a subtype of anything at all.

or Else what?

- An ‘else’ clause in a ‘cond’ expression is almost always needed, for example:

42

```
(: digit-num : Number -> (U Number String))
(define (digit-num n)
  (cond [(<= n 9)     1]
        [(<= n 99)    2]
        [(<= n 999)   3]
        [(<= n 9999)  4]
        [(>  n 9999)  "a lot"]))
```

- if you think that the type checker should know what this is doing, then how about replacing the last test with


```
(> (* n 10) (/ (* (- 10000 1) 20) 2))

;;  or

(>= n 10000)
```

Polymorphic trouble

It is difficult to do the right inference when polymorphic functions are passed around to higher-order functions. For example:

44
```
(: call : (All (A B) (A -> B) A -> B))
(define (call f x)
  (f x))
(call rest (list 4))
```

In such cases, we can use 'inst' to "instantiate" a function with a polymorphic type to a given type – in this case, we can use it to make it treat 'rest' as a function that is specific for numeric lists:

45
```
(call (inst rest Number) (list 4))
```

In other rare cases, Typed Racket will infer a type that is not suitable for us – there is another 'ann' form that allows us to specify a certain type. Using this in the 'call' example is more verbose:

46

```
(call (ann rest : ((Listof Number)
                   -> (Listof Number))) (list
                   4))
```

However, these are going to be rare and will be mentioned explicitly whenever they're needed.

Lecture #2 (plus practice session)

There are two **Boolean values** built-in in Racket:

`#t' (true) and `#f' (false). They can be used in `if' statements, for example:

```
(if (< 2 3) 10 20)  -->  10
```

because (< 2 3) evaluates to `#t'.

As a matter of fact, ***any*** value except for `#f' is considered to be **true**, so:

```
(if 0 1 2)  -->  1

(if "false" 1 2)  -->  1

(if "" 1 2)  -->  1

(if null 1 2)  -->  1  ; null is also a built-in value

(if #f 1 2)  -->  2       ; the only false value
```

Note: Racket is a **functional language** -- so **everything** has a value.

This makes the expression

```
(if test consequent)
```

have no meaning when "test" evaluates to `#f'. This is unlike Pascal/C, where statements **do something** (side effect) like printing or an assignment -- here an if-statement with no

alternate part will just "do nothing" if the test is false...

Racket, however, must return some value -- it could decide on simply returning `#f' (or some unspecified value) as the value of

```
(if #f something)
```

as some implementations do, but Racket just declares it a syntax error.
(As we will see in the future, Racket has a more convenient `when' with a clearer intention.)

Well, *almost* everything has a value...

There are certain things that are part of Racket's syntax -- for example `if' and `define' are **special forms**, they do not have a value! More about this shortly.

(Bottom line: much more things do have a value, compared with other languages.)

`cond' is used for a sequence of `if...else if...else if...else'. The problem is that **nested `if's are inconvenient**. For example,

```
(define (digit-num n)
  (if (<= n 9)
    1
    (if (<= n 99)
      2
      (if (<= n 999)
        3
        (if (<= n 9999)
          4
          "a lot")))))
```

In C/Java/Whatever, you'd write:

```
function digit_num(n) {
  if (n <= 9)          return 1;
  else if (n <= 99)    return 2;
  else if (n <= 999)   return 3;
  else if (n <= 9999)  return 4;
  else return "a lot";
}
```

- (Side question: why isn't there a `return' statement in Racket?)

Bad indentation in Racket – DON'T DO

Trying to force Racket code to look similar:

```
(define (digit-num n)
  (if (<= n 9)
    1
  (if (<= n 99)
    2
  (if (<= n 999)
    3
  (if (<= n 9999)
    4
    "a lot")))))
```

is more than just bad taste -- the indentation rules are there for a reason, the main one is that you can see the structure of your program at a quick glance, and this is no longer true in the above code. (Such code will be penalized!)

So, instead of this, we can use Racket's `cond' statement, like this:

```
(define (digit-num n)
  (cond [(<= n 9)    1]
        [(<= n 99)   2]
        [(<= n 999)  3]
        [(<= n 9999) 4]
        [else        "a lot"]))
```

Note that `else' is a keyword that is used by the `cond' form - you should always use an `else' clause (for similar reasons as an `if', and we will need it when we use a typed language).

[Square brackets] are read by DrRacket like round parens, it will only make sure that the paren pairs match. We use this to make code more readable -- specifically, there is a major difference between the above use of "[]" from the conventional use of "()". Can you see what it is?

The general structure of a `cond':

```
(cond [test-1 expr-1]
      [test-2 expr-2]
      ...
      [test-n expr-n]
      [else   else-expr])
```

Example for using an if-statement, and a recursive function (not tail recursive):

```
(define (fact n)
  (if (zero? n)
    1
    (* n (fact (- n 1)))))
```

Use this to show the different tools, esp:

- special objects that *cannot* be used
- syntax-checker
- stepper

An example of converting it to tail recursive form:

```
(define (helper n acc)
  (if (zero? n)
    acc
    (helper (- n 1) (* acc n))))

(define (fact n)
  (helper n 1))
```

Lists & Recursion

Lists are a **fundamental** Racket data type.

A list is defined as either:

1 the **empty list** (`null', `empty', or '()),

2 a **pair** (`cons' cell) of anything and a **list**.

As simple as this may be, it gives us precise *formal* rules to prove that something is a list.

(Question: Why is there a "the" in the first rule?)

Examples:

```
null
(cons 1 null)
(cons 1 (cons 2 (cons 3 null)))
(list 1 2 3) ; a more convenient function to get the above
```

List operations -- predicates:

```
null? ; true only for the empty list
pair? ; true for any cons cell
list? ; this can be defined using the above
```

We can derive `list?' from the above rules:

```
(define (list? x)
  (if (null? x)
    #t
    (and (pair? x) (list? (rest x)))))
```

Or better yet…

```
(define (list? x)
  (or (null? x)
      (and (pair? x) (list? (rest x)))))
```

But why can't we define `list?' more **simply** as

```
(define (list? x)
  (or (null? x) (pair? x)))
```

The difference between the above definition and the proper one can be observed in the full Racket language, not in the student languages (where, there are no pairs with non-list values in their tails).

List operations -- destructors for pairs (cons cells):

```
first
rest
```

Traditionally called `car', `cdr'.

Also, any `c<x>r' combination for <x> that is made of up to four `a's
and/or `d's -- we will probably not use much more than `cadr', `caddr'
etc.

Example for recursive function involving lists:

```
(define (list-length list)
  (if (null? list)
    0
    (+ 1 (list-length (rest list)))))
```

Use different tools, esp:
* syntax-checker
* stepper

Question: How come we could use `list' as an argument? -- use the syntax checker

---

```
(define (list-length-helper list len)
  (if (null? list)
    len
    (list-length-helper (rest list) (+ len 1))))

(define (list-length list)
  (list-length-helper list 0))
```

Main idea: lists are a **recursive structure**, so **functions** that operate on lists **should be recursive** functions that follow the recursive definition of lists.

Another example for list function -- summing a list of numbers

```
(define (sum-list l)
  (if (null? l)
    0
    (+ (first l) (sum-list (rest l)))))
```

Also show how to implement `rcons', using this guideline.

More examples:

Define `reverse' -- solve the problem using `rcons'.

`rcons' can be generalized into something very useful: `append'.

* How would we use `append' instead of `rcons'?

* How much time will this take? Does it matter if we use `append' or
  `rcons'?

Redefine `reverse' using tail recursion.

* Is the result more complex? (Yes, but not too bad because it collects the elements in reverse.)

BNF, Grammars, the Simple AE Language

- **Context free grammars using BNF**

Getting back to the theme of the course: we want to investigate programming languages, and we want to do that *using* a programming language.

The first thing when we design a language is to specify the language. For this we use **BNF** (**B**ackus-**N**aur **F**orm). For example, here is the definition of a simple arithmetic language:

```
<AE> ::= <num>
       | <AE> + <AE>
       | <AE> - <AE>
```

Explain the different parts. Specifically, this is a mixture of low-level (concrete) syntax definition with parsing.

We use this to **derive expressions** in some language. **We start with** <AE>, which should be one of these:
* a number <num>
* <AE>, the text "+", and another <AE>
* the same but with "-"

It should be clear that the "+" and the "-" are things we expect to find in the input -- because they are not wrapped in <>s. Each of them is a **terminal**.
<num> is also a **terminal**: when we reach it in the derivation, **we're done**. (Small letters)
<AE> is a **non-terminal**: when we reach it, we have to continue with one of the options.

Example:

```
<AE> ::= <num>              ; (1)
       | <AE> + <AE>        ; (2)
       | <AE> - <AE>        ; (3)
```

We can derive the expression "1-2+3" in the following manner (proving "1-2+3" to be a valid <AE> expression):

```
<AE>                      ;          ==>
  <AE> + <AE>             ; (2)      ==>
  <AE> + <num>            ; (1)      ==>
  <AE> - <AE> + <num>     ; (3)      ==>
  <AE> - <AE> + 3         ; (num)    ==>
  <num> - <AE> + 3        ; (1)      ==>
  <num> - <num> + 3       ; (1)      ==>
  1 - <num> + 3           ; (num)    ==>
  1 - 2 + 3               ; (num)
```

- **Using Racket to implement a language.**

We could specify what <num> is (turning it into a <NUM> non-terminal):

```
<NUM> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
        | <NUM> <NUM>
```

But we don't -- why?
Because in Racket **we have numbers as primitives** and we want to use Racket to implement our languages. This makes life a lot easier, and we get free stuff like floats, rationals, etc.

Back to our example:

```
<AE>                      ;          ==>
<AE> + <AE>               ; (2)      ==>
<AE> + <num>              ; (1)      ==>
<AE> - <AE> + <num>       ; (3)      ==>
<AE> - <AE> + 3           ; (num)    ==>
<num> - <AE> + 3          ; (1)      ==>
<num> - <num> + 3         ; (1)      ==>
1 - <num> + 3             ; (num)    ==>
1 - 2 + 3                 ; (num)
```

This would be one way of doing this. Instead, we can **visualize the derivation using a tree**, with the rules used at every node. (Leave this on -- later show that this removes some confusion but not all.)

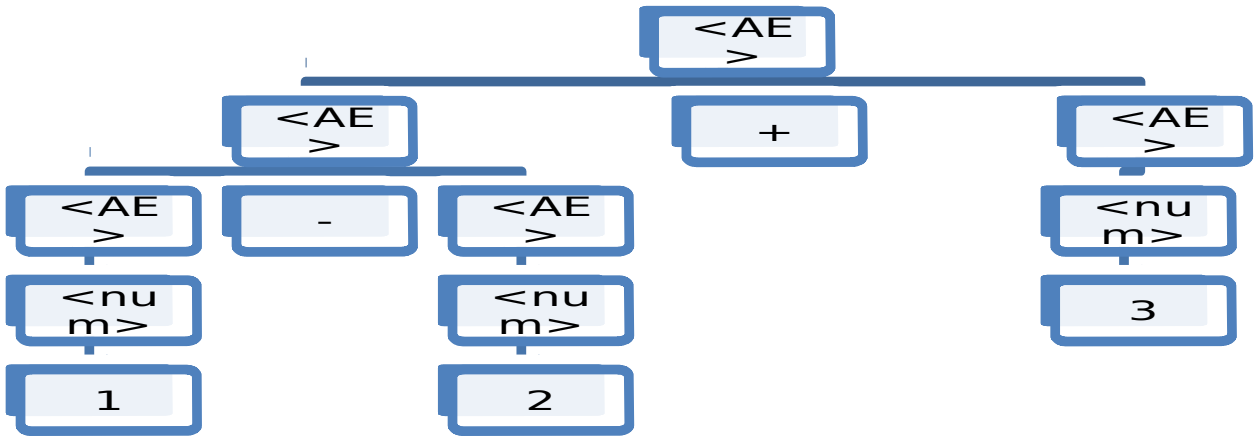

Figure 1 (Leaves are read left to right)

- **Ambiguity**

These specifications suffer from being ambiguous: **an expression can be derived in multiple ways (i.e., have multiple derivation trees).** Even the little syntax for a number is ambiguous -- a number like "123" can be derived in two ways that result in trees that look different. This **ambiguity** is not a "real" problem now, but it will become one very soon. We want to get rid of this ambiguity, so that there is a **single** (= **deterministic**) way to derive all expressions.

There is a standard way to resolve that -- we add another **non-terminal** to the definition, and make it so that each rule can continue to exactly one of its alternatives.

For example: this is what we can do with numbers:

```
<NUM>   ::= <DIGIT> | <DIGIT> <NUM>

<DIGIT> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Similar solutions can be applied to the **<AE>** BNF -- we either restrict the way derivations can happen or we come up

with new non-terminals to force a deterministic derivation trees.

- **Restricting derivations**

As an example of restricting derivations, we look at the current (ambiguous) grammar:

```
<AE> ::= <num>
       | <AE> + <AE>
       | <AE> - <AE>
```

Example 1 – **Left association:**
Instead of allowing an <AE> on both sides of the operation, we force the left one to be a number:

```
<AE> ::= <num>
       | <num> + <AE>
       | <num> - <AE>
```

Now there is a **single** way to derive any expression, and it is always associating operations to the right: an expression like "1+2+3" can only be derived as "1+(2+3)".

Example 2 – **Right association:**
To change this to left-association, we would use this:

```
<AE> ::= <num>
       | <AE> + <num>
       | <AE> - <num>
```

Example 3 – **Semantically required precedence:**
But what if we want to force precedence? Say that our AE syntax has addition and multiplication (now (1+2)*3 is no longer semantically the same as 1+(2*3)):

```
<AE> ::= <num>
       | <AE> + <AE>
       | <AE> * <AE>
```

We can do that same thing as above and add new non-terminals -- say one for "factors":

```
<AE>  ::= <num>
        | <AE> + <AE>
        | <FAC>

<FAC> ::= <num>
        | <FAC> * <FAC>
```

Now we must parse any AE expression as additions of multiplications (or numbers). First, note that if <AE> goes to <fac> and that goes to <num>, then there is no need for an <AE> to go to a <num>, so this is the same syntax:

```
<AE>  ::= <AE> + <AE>
        | <FAC>

<FAC> ::= <num>
        | <FAC> * <FAC>
```

To further loose ambiguity we do the following

```
<AE>  ::= <FAC> + <AE>
        | <FAC>

<FAC> ::= <num>
        | <num> * <FAC>
```

Multiplying additions:
If we want to still be able to **multiply additions**, we can force them to appear in parentheses:

```
<AE>  ::= <AE> + <AE>
        | <FAC>

<FAC> ::= <num>
        | <FAC> * <FAC>
        | ( <AE> )
```

Loosing ambiguity:
Note that AE is still ambiguous about additions (e.g., 1+2+3). This can be fixed by forcing the left hand side of an addition to be a factor:

```
<AE>  ::= <FAC> + <AE>
        | <FAC>
```

```
<FAC> ::= <num>
        | <FAC> * <FAC>
        | ( <AE> )
```

We still have an ambiguity for multiplications (e.g., 1*2*3), so we do the same thing and add another non-terminal for "atoms":

```
<AE>   ::= <FAC> + <AE>
         | <FAC>

<FAC>  ::= <ATOM> * <FAC>
         | <ATOM>

<ATOM> ::= <num>
         | ( <AE> )
```

- **Loosing Ambiguity – the simpler solution**

You can try to derive several expressions to be convinced that derivation is always deterministic now.

But as you can see, this is exactly the cosmetics that we want to avoid -- it will lead us to things that might be interesting, but unrelated to the principles behind programming languages. It will also become **much** worse when we have a real language rather such a tiny one.

Is there a good solution? -- It is right in front of us: do what Racket does -- always use fully parenthesized expressions:

```
<AE> ::= <num>
       | ( <AE> + <AE> )
       | ( <AE> - <AE> )
```

To prevent confusing Racket code with code in our language(s), we also change the parentheses to curly ones:

```
<AE> ::= <num>
       | { <AE> + <AE> }
       | { <AE> - <AE> }
```

- **Using Prefix notation:**

But in Racket ***everything*** has a value -- including those `+'s and `-'s, which makes this extremely convenient with future operations that might have either more or less arguments than 2 as well as treating these arithmetic operators as plain functions. In our toy language we will not do this initially (that is, `+' and `-' are **second order operators**: they cannot be used as values). However, since we will get to it later, we'll adopt the Racket solution and use a fully-parenthesized prefix notation:

```
<AE> ::= <num>
       | { + <AE> <AE> }
       | { - <AE> <AE> }
```

(Remember that in a sense, Racket code is written in a form of already-parsed syntax...)

Our goal here: Implementing a parser in pl for simple arithmetic expression language defined by the following grammar:

```
<AE> ::= <num>
       | { + <AE> <AE> }
       | { - <AE> <AE> }
```

(Remember that in a sense, Racket code is written in a form of already-parsed syntax...)

• Simple Parsing -- Implementing a "parser"

Regardless of what the **syntax actually looks like**, we want to parse it as soon as possible -- converting the **concrete syntax** to an **abstract syntax** tree.

No matter how we write our syntax:
- **3+4 (infix),**
- **3 4 + (postfix),**
- **+(3,4) (prefix with operands in parens),**
- **(+ 3 4) (parenthesized prefix),**

In all of the above, we have the same **abstract syntax** – an operator "+" with two operands "3" and "4" (semantically, we mean the same thing -- adding the number 3 and the number 4). The essence of this (i.e., the **abstract syntax**) is basically a **tree structure** with an addition operation as the root and two leaves holding the two numerals.

With the right data definition, we can describe this in **Racket** -- as the expression (Add (Num 3) (Num 4)) where `Add' and `Num' are **constructors** of a tree type for syntax, or in a **C-like** language, it could be something like Add(Num(3),Num(4)).

Similarly, the expression

```
(3-4)+7
```

will be described in Racket as the expression:

```
(Add (Sub (Num 3) (Num 4)) (Num 7))
```

Important note: "expression" was used in two *different* ways in the above -- each way corresponds to a different language.

Defining a new data type for the task:

To define the data type and the necessary constructors we will use this:

```
(define-type AE
    [Num Number]
    [Add AE AE]
    [Sub AE AE])
```

- Note -- **Racket** follows the tradition of **Lisp** which **makes syntax issues almost negligible** -- the language we use is almost as if we are using the parse tree directly. Actually, it is a very simple syntax for parse trees, one that makes parsing extremely easy.

[This has an interesting historical reason... Some Lisp history -- M-expressions vs. S-expressions, and the fact that we write code that is isomorphic to an AST. Later we will see some of the advantages that we get by doing this. See also "The Evolution of Lisp", section 3.5.1 (especially the last sentence).]

To make things at a very simple level, we will use the above fact through a double-level approach:

- **Read into a list form** – we first "parse" our language into an intermediate representation – a Racket list -- this is mostly done by a modified version of Racket's `read' that uses curly braces "{}"s instead of round parens "()"s.
  [This part will be given to us for free – we will be using a function that reads a string and outputs an Sexpr (a Racket number or list of elements)]
- **Parse into an AST data type** – then, we write our own `parse' function that will parse the resulting list into an instance of the AE type -- an **abstract syntax tree (AST)**.

The latter is achieved by the following simple recursive function:

```
 (: parse-sexpr : Sexpr -> AE)
  ;; to convert s-expressions into AEs
  (define (parse-sexpr sexpr)
    (cond [(number? sexpr) (Num sexpr)]
          [(and (list? sexpr) (= 3 (length sexpr)))
           (let ([make-node
                  (cond [(equal? '+ (first sexpr)) Add]
                        [(equal? '- (first sexpr)) Sub]
                        [else (error 'parse-sexpr "don't
know about ~s"
                                     (first sexpr))])])
             (make-node (parse-sexpr (second sexpr))
                        (parse-sexpr (third sexpr))))]
          [else (error 'parse-sexpr "bad syntax in ~s"
sexpr)]))
```

Equivalently, we can write (using the special form `**match**'):

```
  (: parse-sexpr : Sexpr -> AE)
  ;; to convert s-expressions into AEs
  (define (parse-sexpr sexpr)
    (cond [(number? sexpr) (Num sexpr)]
          [(and (list? sexpr) (= 3 (length sexpr)))
           (let ([make-node
                  (match (first sexpr)
                    ['+ Add]
                    ['- Sub]
                    [else (error 'parse-sexpr "don't know
about ~s"
                                 (first sexpr))])
                  #| the above is the same as:
                  (cond [(equal? '+ (first sexpr)) Add]
                        [(equal? '- (first sexpr)) Sub]
                        [else (error 'parse-sexpr "don't
know about ~s"
                                     (first sexpr))])
                  |#])
             (make-node (parse-sexpr (second sexpr))
                        (parse-sexpr (third sexpr))))]
          [else (error 'parse-sexpr "bad syntax in ~s"
sexpr)]))
```

This function is pretty simple, but as our languages grow, they will become more verbose and more difficult to write. So, instead, we use `match' in a more profound way - as what it does is to match a value and bind new identifiers to different parts (try it with "Check Syntax"). Re-writing the above code using `match':

```
(: parse-sexpr : Sexpr -> AE)
;; to convert s-expressions into AEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n) (Num n)]
    [(list '+ left right)
     (Add (parse-sexpr left) (parse-sexpr right))]
    [(list '- left right)
     (Sub (parse-sexpr left) (parse-sexpr right))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

To make things less confusing, we will combine this with the function that parses a string into a sexpr so we can use strings to represent our programs:

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

▪ The `match' Form

The syntax for `match' is

```
(match value
  [pattern result-expr]
  ...)
```

The value is matched against each pattern, possibly binding names in the process, and if a pattern matches it evaluates the result expression.

The simplest form of a pattern is simply an identifier -- it always matches and binds that identifier to the value:

```
(match (list 1 2 3)
  [x x]) ; evaluates to the list
```

Another simple pattern is **a quoted symbol**, which matches that symbol.
For example:

```
(match var
   ['x "yes"]
   [else "no"])
```

will evaluate to "yes" if `var' is the symbol `x', and to "no" otherwise. Note that `else' is not a keyword here -- it happens to be a pattern that always succeeds, so it behaves like an else clause except that it binds `else' to the unmatched-so-far value.

Many patterns **look like** function application -- but don't confuse them with applications. A `(list x y z)' pattern matches a list of exactly three items and binds the three identifiers; or if the "arguments" are themselves patterns, `match' will descend into the values and match them too. This means that the patterns can be nested:

```
(match (list 1 2 3)
  [(list x y z) (+ x y z)]) ; evaluates to 6
(match '((1) (2) 3)
  [(list (list x) (list y) z) (+ x y z)]) ; evaluates to 6
```

There is also a `cons' pattern that matches a non-empty list and then matches the first part against the head for the list and the second part against the tail of the list.

In a `list' pattern, you can use `...' to specify that the previous pattern is repeated zero or more times, and bound names get bound to the list of respective matching. One simple consequent is that the `(list hd tl ...)' pattern is exactly the same as `(cons hd tl)', but being able to repeat an arbitrary pattern is very useful:

```
(match '((1 2) (3 4) (5 6) (7 8))
  [(list (list x y) ...) (append x y)])
; evaluates to (1 3 5 7 2 4 6 8)
```

A few more useful patterns:

```
id              -- matches anything, binds `id' to it
```

```
  _                 -- matches anything, but does not bind
  (number: n)       -- matches any number and binds it to `n'
  (symbol: s)       -- same for symbols
  (string: s)       -- strings
  (sexpr: s)        -- S-expressions (needed sometimes for
Typed Racket)
  (and pat1 pat2) -- matches both patterns
  (or pat1 pat2)  -- matches either pattern (careful with
bindings)
```

The patterns are tried one by one *in-order*, and if no pattern matches the value, an error is raised.

Note that `...' in a `list' pattern can follow *any* pattern, including all of the above, and including nested list patterns.

Here are a few examples -- you can try them out with "**#lang pl untyped**" (for our purposes pl -typed- will suffice) at the top of the definitions window. This:

```
  (match x
    [(list (symbol: syms) ...) syms])
```

matches `x' against a pattern that accepts only a list of symbols, and binds `syms' to those symbols. And here's an example that matches a list of any number of lists, where each of the sub-lists begins with a symbol and then has any number of numbers. Note how the `n' and `s' bindings get values for a list of all symbols and a list of lists of the numbers:

```
  > (define (foo x)
      (match x
        [(list (list (symbol: s) (number: n) ...) ...)
         (list 'symbols: s 'numbers: n)]))
  > (foo (list (list 'x 1 2 3) (list 'y 4 5)))
  '(symbols: (x y) numbers: ((1 2 3) (4 5)))
```

Here is a quick example for how `or' is used with two literal alternatives, how `and' is used to name a specific piece of data, and how `or' is used with a binding:

```
  > (define (foo x)
      (match x
```

```
      [(list (or 1 2 3)) 'single]
      [(list (and x (list 1 _)) 2) x]
      [(or (list 1 x) (list 2 x)) x]))
> (foo (list 3))
'single
> (foo (list (list 1 99) 2))
'(1 99)
> (foo (list 1 10))
10
> (foo (list 2 10))
10
```

Semantics (= Evaluation)

[[[PLAI Chapter 2]]]

Back to BNF -- now, **meaning**.

An important feature of these BNF specifications: we can use the derivations to specify *meaning* (and meaning in our context is "running" a program (or "interpreting", "compiling", but we will use "evaluating")). For example:

```
  <AE> ::= <num>          ; <AE> evaluates to the number
         | <AE1> + <AE2> ; <AE> evaluates to the sum of
evaluating
                          ;      <AE1> and <AE2>
         | <AE1> - <AE2> ; ... the subtraction of <AE2> from
<AE1>
                                  (... roughly!)
```

To do this a little more formally:

a. **eval(<num>) = <num>** ; <-- special rule: moves syntax into a value
b. **eval(<AE1> + <AE2>) = eval(<AE1>) + eval(<AE2>)**
c. **eval(<AE1> - <AE2>) = eval(<AE1>) - eval(<AE2>)**

Note the completely different roles of the two "+"s and "-"s. In fact, it might have been more correct to write:

a. **eval("<num>") = <num>**
b. **eval("<AE1> + <AE2>") = eval("<AE1>") + eval("<AE2>")**
c. **eval("<AE1> - <AE2>") = eval("<AE1>") - eval("<AE2>")**

or even using a marker to denote meta-holes in these strings:

```
  a. eval("$<num>") = <num>
  b. eval("$<AE1> + $<AE2>") = eval("$<AE1>") +
eval("$<AE2>")
  c. eval("$<AE1> - $<AE2>") = eval("$<AE1>") -
eval("$<AE2>")
```

but we will avoid pretending that we're doing that kind of string manipulation. (For example, it will require specifying what does it mean to return <num> for "$<num>" (involves `string->number'), and the fragments on the right

side mean that we need to specify these as substring operations.)

Note that there's a similar kind of informality in our BNF specifications, where we assume that "<foo>" refers to some terminal or non-terminal. In texts, where more formality is required (for example, in RFC specifications), each literal part of the BNF is usually marked with double quotes, so we'd get

```
<AE> ::= <num> | <AE1> "+" <AE2> | <AE1> "-" <AE2>
```

An alternative popular notation for eval(X) is [[X]]:

```
a. [[<num>]] = <num>
b. [[<AE1> + <AE2>]] = [[<AE1>]] + [[<AE2>]]
c. [[<AE1> - <AE2>]] = [[<AE1>]] - [[<AE2>]]
```

Is there a problem with this definition? **Ambiguity:**

```
eval(1 - 2 + 3) = ?
```

Depending on the way the expression is parsed, we get either 2 or -4:

```
eval(1 - 2 + 3) = eval(1 - 2) + eval(3)          [b]
                = eval(1) - eval(2) + eval(3)    [c]
                = 1 - 2 + 3                      [a,a,a]
                = 2

eval(1 - 2 + 3) = eval(1) - eval(2 + 3)          [c]
                = eval(1) - (eval(2) + eval(3))  [a]
                = 1 - (2 + 3)                    [a,a,a]
                = -4
```

Again, be very aware of confusing subtleties which are extremely important: We need parens around a sub-expression only on one side, why?
-- When we write:

```
eval(1 - 2 + 3) = ... = 1 - 2 + 3
```

we have two expressions, but one stands for an *input syntax*, and one stands for a `real' mathematical expression.

In a case of a computer implementation, the syntax on the left is (as always) an AE syntax, and the `real' expression on the right is an expression in whatever language we use to implement our AE language.

Like we said earlier, ambiguity is **not a real problem** until the actual parse tree matters. With `**eval**' it **definitely matters**, so we must **not** make it possible to derive any syntax in multiple ways or our evaluation will be non-deterministic.

Compositionality – an important feature of a syntax
(Compositionality: The meaning of a complex expression is determined by its structure and the meanings of its constituents.**)**

Quick exercise:

We can define a meaning for <digit>s and then <num>s in a similar way:

```
<NUM> ::= <digit> | <digit> <NUM>

eval(0) = 0
eval(1) = 1
eval(2) = 2
...
eval(9) = 9

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

Is this exactly what we want? -- Depends on what we actually want...

* First, there's a bug in this code -- having a BNF derivation like

```
<NUM> ::= <digit> | <digit> <NUM>
```

is unambiguous, but makes it **hard** to parse a number. We get:

```
eval(123) = 10*eval(1) + eval(23)
          = 10*1 + 10*eval(2) + eval(3)
```

```
                = 10*1 + 10*2 + 3
                = 33
```

Changing the order of the last rule works much better:

```
<NUM> ::= <digit> | <NUM> <digit>
```

and then:

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

* As a concrete example see how you would make it work with "107", which demonstrates how the compositionality is important.

* Example for free stuff that looks trivial: if we were to define the meaning of numbers this way, would it always work? Think of an average language that does not give you **bignums**, making the above rules fail when the numbers are too big. In Racket, we happen to be using an integer representation for the syntax of integers, and both are unlimited. But what if we wanted to write a Racket compiler in C or a C compiler in Racket? What about a C compiler in C, where the compiler runs on a 64 bit machine, and the result needs to run on a 32 bit machine?

Side comment on compositionality

The example of

```
<NUM> ::= <digit> | <NUM> <digit>
```

being a language that is easier to write an evaluator for leads us to an important concept -- **compositionality.** This definition is easier to write an evaluator for, since the **resulting language is compositional:** the meaning of an expression -- for example `123' -- is composed out of the meaning of its two parts, which in this BNF are `12' and `3'.

Specifically, the evaluation of `<NUM> <digit>' is 10 * the evaluation of the first, plus the evaluation of the second. In the `<digit> <NUM>' case this is more difficult -- the meaning of such a number **depends not only on the *meaning* of the two parts**, but also on the `<NUM>' *syntax*:

eval(<digit> <NUM>) = eval(<digit>) * 10^length(<NUM>) + eval(<NUM>)

This case can be **tolerable**, since the meaning of the expression is still made out of its parts -- but **imperative programming** (when you use side effects) is much more problematic since it is not compositional (at least not in the obvious sense). This is compared to functional programming, where the meaning of an expression is a combination of the meanings of its sub-expressions. For example, every sub-expression in a functional program has some known meaning, and these all make up the meaning of the expression that contains them -- but in an imperative program we can have a part of the code be `x++' - and that doesn't have a meaning by itself, at least not one that contributes to the meaning of the whole program in a direct way.

(Actually, we can have a well-defined meaning for such an expression: the meaning is going from a world where `x' is a container of some value N, to a world where the same container has a different value N+1. You can probably see now how this can make things more complicated. On an intuitive level -- **if we look at a random part of a functional program we can tell its meaning**, so building up the meaning of the whole code is easy, but **in an imperative program, the meaning of a random part is pretty much useless.)**

Implementing an Evaluator

Now continue to implement the semantics of our syntax -- we express that through an `**eval**' function that evaluates an expression.

We use a basic programming principle -- **splitting the code into two layers**, one for parsing the input, and one for doing the evaluation. Doing this avoids the mess we'd get into otherwise, for example:

```
(define (eval sexpr)
  (match sexpr
    [(number: n) n]
    [(list '+ left right) (+ (eval left) (eval right))]
    [(list '- left right) (- (eval left) (eval right))]
    [else (error 'eval "bad syntax in ~s" sexpr)]))
```

This is messy because it **combines two very different things** - **syntax** and **semantics** -- into a single lump of code. For this particular kind of evaluator it looks simple enough, but this is only because it's simple enough that all we do is replace constructors by arithmetic operations. Later on things will get more complex, and bundling the evaluator with the parser will be more problematic. (Note: The fact, that we can replace constructors with the run-time operators, means that we have a very simple, calculator-like language, and that we can, in face, "compile" all programs down to a number.)

If we split the code, we can easily **include decisions** like making

```
{+ 1 {- 3 "a"}}
```

syntactically invalid. (Which is not, BTW, what Racket does...) (Also, this is like the distinction between XML syntax and well-formed XML syntax.)

An additional advantage is that by using two separate components, **it is simple to replace each one**, making it possible to change the input syntax, and the semantics independently -- we only need to keep the same interface data (the AST) and things will work fine.

Our `**parse**' function converts an input syntax to an abstract syntax tree (AST). It is abstract exactly because it is

independent of any actual concrete syntax that you type in, print out etc.

Implementing The AE Language

Back to our `eval' -- this will be its (obvious) type:

```
(: eval : AE -> Number)
;; consumes an AE and computes the corresponding number
```

which leads to some obvious test cases (using the new `test' form that the `#lang pl' language provides):

```
(test (eval (parse "3"))              => 3)
(test (eval (parse "{+ 3 4}"))        => 7)
(test (eval (parse "{+ {- 3 4} 7}")) => 6)
```

Note that we're testing *only* at the interface level -- only running whole functions. For example, you could think about a test like:

```
(test (parse "{+ {- 3 4} 7}")
      => (Add (Sub (Num 3) (Num 4)) (Num 7)))
```

but the details of parsing and of the constructor names are things that nobody outside of our evaluator cares about -- so we're not testing them. In fact, we shouldn't even mention `**parse**' in these tests, since it is not part of the public interface of our users; they only care about using it as a compiler-like black box. (This is sometimes called "integration tests".) We'll address this shortly.

Like everything else, the structure of the recursive `eval' code follows the recursive structure of its input. In HtDP terms, our template is:

```
(: eval : AE -> Number)
(define (eval expr)
  (cases expr
    [(Num n)    ... n ...]
    [(Add l r) ... (eval l) ... (eval r) ...]
    [(Sub l r) ... (eval l) ... (eval r) ...]))
```

In this case, filling in the gaps is very simple

```
(: eval : AE -> Number)
(define (eval expr)
  (cases expr
    [(Num n)   n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]))
```

We now further combine `**eval**' and `**parse**' into a single `**run**' function that evaluates an AE string.

```
(: run : String -> Number)
;; evaluate an AE program contained in a string
(define (run str)
  (eval (parse str)))
```

This function becomes the single public entry point into our code, and the only thing that should be used in tests that verify our interface:

```
(test (run "3")              => 3)
(test (run "{+ 3 4}")        => 7)
(test (run "{+ {- 3 4} 7}") => 6)
```

The resulting *full* code is:

<<<AE>>>

```
#lang pl

#| BNF for the AE language:
   <AE> ::= <num>
          | { + <AE> <AE> }
          | { - <AE> <AE> }
          | { * <AE> <AE> }
          | { / <AE> <AE> }
|#

;; AE abstract syntax trees
(define-type AE
  [Num Number]
  [Add AE AE]
  [Sub AE AE]
  [Mul AE AE]
  [Div AE AE])
```

```
(: parse-sexpr : Sexpr -> AE)
;; to convert s-expressions into AEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n) (Num n)]
    [(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))]
    [else
     (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

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

(: eval : AE -> Number)
;; consumes an AE and computes the corresponding number
(define (eval expr)
  (cases expr
    [(Num n)   n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]
    [(Mul l r) (* (eval l) (eval r))]
    [(Div l r) (/ (eval l) (eval r))]))

(: run : String -> Number)
;; evaluate an AE program contained in a string
(define (run str)
  (eval (parse str)))


;; tests
(test (run "3") => 3)
(test (run "{+ 3 4}") => 7)
(test (run "{+ {- 3 4} 7}") => 6)
```

(Note that the tests are done with a `**test**' form, which we mentioned above.)

For anyone who thinks that Racket is a bad choice, this is a good point to think how much code would be needed in some other language to do the same as above.

Bindings & Substitution

We now get to an important concept: **substitution.**

Even in our simple language, we encounter **repeated expressions**. For example, if we want to compute the square of some expression:

```
{* {+ 4 2} {+ 4 2}}
```

- Why would we want to get rid of the repeated sub-expression?
  - **Redundant computation** -- In this example, we want to avoid computing the same sub-expression a second time.
  - **More complicated computation** -- computation can be simpler without the repetition. Compare the above with:

    ```
    x = {+ 4 2},
      {* x x}
    ```

  - **Duplicating information is always a bad thing** -- Among other bad consequences, it can even **lead to bugs** that could not happen if we wouldn't duplicate code. A toy example is "fixing" one of the numbers in one expression and forgetting to fix the corresponding one:

    ```
    {* {+ 4 2} {+ 4 1}}
    ```

    Real world examples involve much more code, which make such bugs very difficult to find, but they still follow the same principle.

  - **More expressive power** -- We don't just say that we want to multiply two expressions that both *happen to be* `{+ 4 2}`, we say that we multiply the `{+ 4 2}` expression by ***itself***. It allows us to express identity of two values as well as using two values that happen to be the same.

Identifiers

The normal way to avoid redundancy is to introduce an identifier. Even when we speak, we might say: "***let x be 4 plus 2, multiply x by x***".

(These are often called "variables", but we will try to avoid this name: what if the identifier does not change (vary)?)

To get this, we introduce a new form into our language:

```
{with {x {+ 4 2}}
  {* x x}}
```

We expect to be able to reduce this to:

```
{* 6 6}
```

by substituting '`x`' by `6` in the body sub-expression of '`with`'.

A little more complicated example:

```
{with {x {+ 4 2}}
  {with {y {* x x}}
    {+ y y}}}

[add]  = {with {x 6} {with {y {* x x}} {+ y y}}}
[subst]= {with {y {* 6 6}} {+ y y}}
[mul]  = {with {y 36} {+ y y}}
[subst]= {+ 36 36}
[add]  = 72
```

Adding Bindings to AE: The WAE Language

[[[PLAI Chapter 3]]]

To add this to our language, we start with the BNF. We now call our language '**WAE**' (With+AE):

```
<WAE> ::= <num>
        | { + <WAE> <WAE> }
        | { - <WAE> <WAE> }
        | { * <WAE> <WAE> }
        | { / <WAE> <WAE> }
        | { with { <id> <WAE> } <WAE> }
        | <id>
```

Note that we had to introduce **two new rules**:

- **One for introducing an identifier, and**
- **One for using it.**

This is common in many language specifications, for example '**define-type**' introduces a new type, and it comes with '**cases**' that allows us to destruct its instances.

For <id> we need to use some form of identifiers, the natural choice in Racket is to use symbols. We can therefore write the corresponding type definition:

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

The parser is easily extended to produce these syntax objects:

```
(: parse-sexpr : Sexpr -> WAE)
;; to convert s-expressions into WAEs

(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)     (Num n)]
    [(symbol: name) (Id name)]
    [(list 'with (list (symbol: name) named) body)
     (With name (parse-sexpr named) (parse-sexpr body))]
    [(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))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

But note that this parser is inconvenient -- if any of these expressions:

```
{* 1 2 3}
{foo 5 6}
{with x 5 {* x 8}}
{with {5 x} {* x 8}}
```

would result in a **"bad syntax"** error, which is not very informative. To make things better, we can add another case for `with' expressions that are malformed, and give a more specific message in that case:

```
(: parse-sexpr : Sexpr -> WAE)
;; to convert s-expressions into WAEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)     (Num n)]
    [(symbol: name) (Id name)]
    [(list 'with (list (symbol: name) named) body)
     (With name (parse-sexpr named) (parse-sexpr body))]
    [(cons 'with more)
     (error 'parse-sexpr "bad `with' 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))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

Finally, to group all of the parsing code that deals with 'with' expressions (both valid and invalid ones), we can use a single case for both of them:

```
(: parse-sexpr : Sexpr -> WAE)
;; to convert s-expressions into WAEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)     (Num n)]
    [(symbol: name)  (Id name)]
    [(cons 'with more)
     ;; go in here for all sexpr that begin with a 'with
     (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)])]
    [(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))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

And now we're done with the syntactic part of the '**with** ' extension.

Quick note – the difference between 'With' and 'with':

- We would indent 'With' like a normal function in code like this

```
(With 'x
      (Num 2)
      (Add (Id 'x) (Num 4)))
```

instead of an indentation that looks like a 'let'

```
(With 'x (Num 2)
  (Add (Id 'x) (Num 4)))
```

The reason for this is that the second indentation looks more a binding construct (eg, how a 'let' is indented), but '**With**' is *__not__* a binding form -- it's a plain function because it's at the Racket level. You should therefore keep in mind the huge difference between that '**With**' and the '**with**' that appears in WAE programs:

```
{with {x 2}
  {+ x 4}}
```

Another way to look at it: imagine that we intend for the language to be used by Spanish speakers. In this case we would translate "**with**":

```
{con {x 2}
  {+ x 4}}
```

but we will not do the same for '**With**'.)

Implementing Evaluation of 'with' (evaluation with substitutions)

To make this work, we will need to do some **substitutions**.

We basically want to say that to evaluate:

```
{with {id WAE1} WAE2}
```

we need to evaluate **WAE2** with id substituted by **WAE1**. Formally:

```
eval( {with {id WAE1} WAE2} ) = eval( subst(WAE2,id,eval(WAE1)) )
```

There is a more common syntax for substitution (quick: what do I mean by this "**syntax**"?):

```
eval( {with {id WAE1} WAE2} ) = eval( WAE2[eval(WAE1)/id] )
```

(Side note: this syntax originates with logicians who used '[x/v]e', and later there was a convention that mimicked the more natural order of arguments to a function with 'e[x->v]', and eventually both of these got combined into 'e[v/x]' which is a little confusing in that the left-to-right order of the arguments is not the same as for the 'subst' function.)

Substitutions:

Now all we need is an exact definition of substitution. (Note that **substitution is not the same as evaluation**, only part of the evaluation process. In the previous examples, when we evaluated the expression we did substitutions as well as the usual arithmetic operations that were already part of the AE evaluator. In this last definition there is still a missing evaluation step, see if you can find it.)

Defining substitutions:

[***substitution***, take 1] **e[v/i]**
To substitute an identifier '**i**' in an expression '**e**' with an expression '**v**', **replace all identifiers** in '**e**' that have the **same name** '**i**' **by the expression** '**v**'.

This seems to work with simple expressions, for example:

```
{with {x 5} {+ x x}}   -->  {+ 5 5}
{with {x 5} {+ 10 4}}  -->  {+ 10 4}
```

Problem: we crash with an invalid syntax if we try:

```
{with {x 5} {+ x {with {x 3} 10}}}  -->  {+ 5 {with {5 3} 10}} ???
```

-- we got to an invalid expression.

To fix this, we need to distinguish **"normal"** occurrences of identifiers, and ones that are used as **new bindings**. We need a few new terms for this:

1. **Binding Instance**: a binding instance of an identifier is one that is used to name it in a new binding. In our **<WAE>** syntax, binding instances are only the **<id>** position of the '**with**' form.

2. **Scope**: the scope of a binding instance is the region of program text in which instances of the identifier refer to the value bound in the binding instance. (Note that this definition actually relies on a definition of substitution, because that is what is used to specify how identifiers refer to values.)

3. **Bound Instance (or Bound Occurrence)**: an instance of an identifier is bound if it is contained within the scope of a binding instance of its name.

4. **Free Instance (or Free Occurrence)**: An identifier that is not contained in the scope of any binding instance of its name is said to be free.

Using this we can say that the problem with the previous definition of substitution is that **it failed to distinguish between bound instances** (which should be substituted) from **binding instances** (which should not).

So, we try to fix this:

[***substitution***, take 2] **e[v/i]**
To substitute an identifier '**i**' in an expression '**e**' with an expression '**v**', replace all instances of '**i**' that are **not themselves binding instances** with the expression '**v**'.

First of all, check the previous examples:

```
{with {x 5} {+ x x}}   -->  {+ 5 5}
{with {x 5} {+ 10 4}}  -->  {+ 10 4}
```

still work, and

```
{with {x 5} {+ x {with {x 3} 10}}}  -->  {+ 5 {with {x 3} 10}}
                                    -->  {+ 5 10}
```

also works. However, if we try this:

```
{with {x 5}
  {+ x {with {x 3}
         x}}}
```

we get:

```
-->  {+ 5 {with {x 3} 5}}
-->  {+ 5 5}
-->  10
```

but we want that to be **8**: the inner '**x**' should be bound by the closest

'**with**' that binds it.

The problem is that the new definition of substitution that we have respects binding instances, but it **fails to deal with their scope**. In the above example, we want the inner '**with**' to *__shadow__* the outer '**with**'s binding for '**x**'.

[***substitution***, take 3] **e[v/i]**
To substitute an identifier '**i**' in an expression '**e**' with an expression '**v**', replace all instances of '**i**' that are **not themselves binding instances**, and that are **not in any nested scope**, with the expression '**v**'.

This avoids bad substitution above, but it is now doing things too carefully:

```
{with {x 5} {+ x {with {y 3} x}}}
```

becomes

```
-->  {+ 5 {with {y 3} x}}
-->  {+ 5 x}
```

which is an **error** because '**x**' is unbound (and there is no reasonable rule that we can specify to evaluate it).

The problem is that our substitution halts at every new scope, in this case, it stopped at the new '**y**' scope, but it shouldn't have because it uses a different name.

Revise again:

[***substitution***, **take 4**] **e[v/i]**
To substitute an identifier '**i**' in an expression '**e**' with an expression '**v**', replace all instances of '**i**' that **are not themselves binding instances**, and that **are not in any nested scope of** '**i**', with the expression '**v**'.

Finally, this is a good definition. It is just a little too mechanical. Notice that we actually refer to all instances of '**i**' that are not in a scope of a binding instance of '**i**', which simply means all *__free occurrences__* of '**i**' -- free in '**e**' (why? -- remember the definition of "free"?):

[***substitution***, **take 4a**] **e[v/i]**
To substitute an identifier '**i**' in an expression '**e**' with an expression '**v**', replace all instances of '**i**' that **are free in** '**e**' with the expression '**v**'.

Based on this we can finally write the <u>code for it</u>:

```
(: subst : WAE Symbol WAE -> WAE)
;; 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) ; returns expr[to/from]
  (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)
     (if (eq? bound-id from)
       expr           ; <-- don't go in!
       (With bound-id
             named-expr
             (subst bound-body from to)))]))
```

... and this is just the same as writing a formal "paper version" of the substitution rule.

... but we still have bugs! ……

```
(: subst : WAE Symbol WAE -> WAE)
;; 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) ; returns expr[to/from]
  (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)
     (if (eq? bound-id from)
       expr            ; <-- don't go in!
       (With bound-id
             named-expr
             (subst bound-body from to)))]))
```

... and this is just the same as writing a formal "paper version" of the substitution rule.

... but we still have bugs!

Before we find the bugs, let us see the bigger context…

When and How substitution is used in the evaluation process.

Modifying our evaluator:

We will need rules to deal with the new syntax pieces -- `**with**' expressions and identifiers.

- Evaluating `**with**' expressions - for an expression of the following form:

  ```
  {with {x E1} E2}
  ```

  We need to:
  1. ***evaluate*** `**E1**' to get a **value** `**V1**',
  2. we then substitute the identifier `**x**' with the expression `**V1**' in `**E2**', and
  3. finally, we evaluate this resulting new expression.

  In other words, we have the following evaluation rule:

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

  So we know what to do with `**with**' expressions.

- Evaluating **identifiers** -

  The main feature of `**subst**', as said in the purpose statement, is that **it leaves no free instances of the substituted variable**

around. This means that if the initial expression is valid (did not contain any free variables), then when we go from

```
{with {x E1} E2}
```

to

```
E2[E1/x]
```

Then, the result is an expression that has ***no* free instances of** `x'. So we don't need to handle identifiers in the evaluator -- substitutions make them all go away.

We can now extend the formal definition of **AE** to that of **WAE**:

```
eval(...) = ... same as the AE rules ...
eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
eval(id) = error!
```

Important to note: (*** what should we feed to `subst')
If you're paying close attention, you might catch a potential problem in this definition. We're substituting `eval(E1)' for `x' in `E2'

- `subst' **requires a WAE expression**. However,
- **what we are really getting** is `eval(E1)' which **is a number**.

(Look at the type of the `eval' definition we had for **AE**, then look at the above definition of `subst'.) This seems like being overly pedantic, but it will require some resolution when we get to the code.

The above rules are easily coded as follows:

```
(: eval : WAE -> Number)
;; evaluates WAE expressions by reducing them to numbers
(define (eval expr)
  (cases expr
    [(Num n) n]                                  ;; same as before
    [(Add l r) (+ (eval l) (eval r))]            ;; same as before
    [(Sub l r) (- (eval l) (eval r))]            ;; same as before
    [(Mul l r) (* (eval l) (eval r))]            ;; same as before
    [(Div l r) (/ (eval l) (eval r))]            ;; same as before
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (Num (eval named-expr))))] ; <-*** (see note)
    [(Id name) (error 'eval "free identifier: ~s" name)]))
```

Note the `Num' expression in the marked line: evaluating the named expression gives us back a number -- we need to convert this number into an abstract syntax (**WAE**) to be able to use it with `subst'.

The solution is to use the constructor `**Num**' to convert the resulting number into a numeral (the syntax of a number). It's not an elegant solution, but it will do for now.

- -

<u>A few test cases</u>:
We use a new `**test**' special form which is part of the course plugin. The way to use ` **test**' is with two expressions and an `=>' arrow -- DrRacket evaluates both, and nothing will happen if the results are equal. If the results are different, you will get a warning line, but evaluation will continue so you can try additional tests. You can also use an `=error>' arrow to test an error message -- use it with some text from the expected error, `?' stands for any single character, and `*' is a sequence of zero or more characters.
(When you use `test' in your homework, the handin server will abort when tests fail.) We expect these tests to succeed (make sure that you understand *why* they should succeed).

```
;; tests
(test (run "5") => 5)
(test (run "{+ 5 5}") => 10)
(test (run "{with {x {+ 5 5}} {+ x x}}") => 20)
(test (run "{with {x 5} {+ x x}}") => 10)
(test (run "{with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}") => 14)
(test (run "{with {x 5} {with {y {- x 3}} {+ y y}}}") => 4)
(test (run "{with {x 5} {+ x {with {x 3} 10}}}") => 15)
(test (run "{with {x 5} {+ x {with {x 3} x}}}") => 8)
(test (run "{with {x 5} {+ x {with {y 3} x}}}") => 10)
(test (run "{with {x 5} {with {y x} y}}") => 5)
(test (run "{with {x 5} {with {x x} x}}") => 5)
(test (run "{with {x 1} y}") =error> "free identifier")
```

Putting this all together, we get the following code; trying to run this code will raise an unexpected error...

```
#lang pl

#| BNF for the WAE language:
     <WAE> ::= <num>
             | { + <WAE> <WAE> }
             | { - <WAE> <WAE> }
             | { * <WAE> <WAE> }
             | { / <WAE> <WAE> }
             | { with { <id> <WAE> } <WAE> }
             | <id>
|#

;; WAE abstract syntax trees
(define-type WAE
  [Num  Number]
  [Add  WAE WAE]
  [Sub  WAE WAE]
  [Mul  WAE WAE]
  [Div  WAE WAE]
  [Id   Symbol]
  [With Symbol WAE WAE])

(: parse-sexpr : Sexpr -> WAE)
;; to convert s-expressions into WAEs
(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)])]
    [(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))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

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

(: subst : WAE Symbol WAE -> WAE)
;; 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)
       (if (eq? bound-id from)
         expr
         (With bound-id
               named-expr
               (subst bound-body from to)))]))

  (: eval : WAE -> Number)
  ;; evaluates WAE expressions by reducing them to numbers
  (define (eval expr)
    (cases expr
      [(Num n) n]
      [(Add l r) (+ (eval l) (eval r))]
      [(Sub l r) (- (eval l) (eval r))]
      [(Mul l r) (* (eval l) (eval r))]
      [(Div l r) (/ (eval l) (eval r))]
      [(With bound-id named-expr bound-body)
       (eval (subst bound-body
                    bound-id
                    (Num (eval named-expr))))]
      [(Id name) (error 'eval "free identifier: ~s" name)]))

  (: run : String -> Number)
  ;; evaluate a WAE program contained in a string
  (define (run str)
    (eval (parse str)))

  ;; tests
  (test (run "5") => 5)
  (test (run "{+ 5 5}") => 10)
  (test (run "{with {x {+ 5 5}} {+ x x}}") => 20)
  (test (run "{with {x 5} {+ x x}}") => 10)
  (test (run "{with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}") => 14)
;; in reality returns -- eval: free identifier: x
  (test (run "{with {x 5} {with {y {- x 3}} {+ y y}}}") => 4)
  (test (run "{with {x 5} {+ x {with {x 3} 10}}}") => 15)
  (test (run "{with {x 5} {+ x {with {x 3} x}}}") => 8)
  (test (run "{with {x 5} {+ x {with {y 3} x}}}") => 10)
  (test (run "{with {x 5} {with {y x} y}}") => 5)
  (test (run "{with {x 5} {with {x x} x}}") => 5)
  (test (run "{with {x 1} y}") =error> "free identifier")
  ----------------------------------------------------------------------
```

Oops, this program **still has problems** that were caught by the tests -- we encounter unexpected free identifier errors. What's the problem now? In expressions like:

```
  {with {x 5}
    {with {y x}
      y}}
```

The problem:

We forgot to substitute `x' in the expression that `y' is bound to. We need to perform the recursive substitution in two places:

1. **The "with"'s body expression** (which we already have), and
2. **The "with"'s named expression.**

The new **'subst'** code will be:

```
(: subst : WAE Symbol WAE -> WAE)
;; 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)
     (if (eq? bound-id from)
       expr
       (With bound-id
             (subst named-expr from to)        ; <-- new
             (subst bound-body from to)))]))
```

However, we *still* have a problem...

Look at the expression:

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

Halts with an error -- **eval: free identifier: x**
It should, however, evaluate to **5**. Carefully trying out our substitution code reveals **the problem**:
We do not go inside the inner `with'** when we substitute `5' for the outer `x'. This is because it has the **same name 'x' – but we ***do*** need to go into its named expression.
We need to substitute in the named expression even if the identifier has the *same* name as the one we're substituting:

```
(: subst : WAE Symbol WAE -> WAE)
;; 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) ;; new - only ask on body
              bound-body
              (subst bound-body from to)))]))
```

The complete (and, finally, correct) version of the code is now:

```
---<<<WAE>>>-------------------------------------------------------

#lang pl

#| BNF for the WAE language:
     <WAE> ::= <num>
             | { + <WAE> <WAE> }
             | { - <WAE> <WAE> }
             | { * <WAE> <WAE> }
             | { / <WAE> <WAE> }
             | { with { <id> <WAE> } <WAE> }
             | <id>
|#

;; WAE abstract syntax trees
(define-type WAE
  [Num  Number]
  [Add  WAE WAE]
  [Sub  WAE WAE]
  [Mul  WAE WAE]
  [Div  WAE WAE]
  [Id   Symbol]
  [With Symbol WAE WAE])

(: parse-sexpr : Sexpr -> WAE)
;; to convert s-expressions into WAEs
(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)])]
    [(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))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

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

#| Formal specs for `subst':
   (`N' is a <num>, `E1', `E2' are <WAE>s, `x' is some <id>, `y' is a
   *different* <id>)
```

```
      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}
|#

(: subst : WAE Symbol WAE -> WAE)
;; 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)))]))

#| Formal specs for `eval':
     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 : WAE -> Number)
;; evaluates WAE expressions by reducing them to numbers
(define (eval expr)
  (cases expr
    [(Num n) n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]
    [(Mul l r) (* (eval l) (eval r))]
    [(Div l r) (/ (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (Num (eval named-expr))))]
    [(Id name) (error 'eval "free identifier: ~s" name)]))

(: run : String -> Number)
;; evaluate a WAE program contained in a string
```

```
(define (run str)
  (eval (parse str)))

;; tests
(test (run "5") => 5)
(test (run "{+ 5 5}") => 10)
(test (run "{with {x {+ 5 5}} {+ x x}}") => 20)
(test (run "{with {x 5} {+ x x}}") => 10)
(test (run "{with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}") => 14)
(test (run "{with {x 5} {with {y {- x 3}} {+ y y}}}") => 4)
(test (run "{with {x 5} {+ x {with {x 3} 10}}}") => 15)
(test (run "{with {x 5} {+ x {with {x 3} x}}}") => 8)
(test (run "{with {x 5} {+ x {with {y 3} x}}}") => 10)
(test (run "{with {x 5} {with {y x} y}}") => 5)
(test (run "{with {x 5} {with {x x} x}}") => 5)
(test (run "{with {x 1} y}") =error> "free identifier")
----------------------------------------------------------------------
```

Reminder:

* We started doing substitution, with a `let'-like form: `with'.

* Reasons for using bindings:
  - Avoid writing expressions twice.
    -> More expressive language (can express identity).
    -> Duplicating is bad! (=> DRY, Don't Repeat Yourself)
    --> Static redundancy.
  - Avoid redundant computations.
    --> Dynamic redundancy.

* BNF:

```
<WAE> ::= <num>
        | { + <WAE> <WAE> }
        | { - <WAE> <WAE> }
        | { * <WAE> <WAE> }
        | { / <WAE> <WAE> }
        | { with { <id> <WAE> } <WAE> }
        | <id>
```

  Note that we had to introduce two new rules: one for introducing an identifier, and one for using it.

* Type definition:

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

* Parser:

```
  (: parse-sexpr : Sexpr -> WAE)
  ;; to convert s-expressions into WAEs
  (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)])]
      [(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) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

* We need to define substitution.
  Terms:
  1. Binding Instance.
  2. Scope.
  3. Bound Instance.
  4. Free Instance.

* After lots of attempts:

  e[v/i] -- To substitute an identifier `i' in an expression `e' with an expression `v', replace all instances of `i' that are free in `e' with the expression `v'.

* Implemented the code, and again, needed to fix a few bugs:

```
  (: subst : WAE Symbol WAE -> WAE)
  ;; 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)))]))
```

  (Note that the bugs that we fixed clarify the exact way that our scopes work: in `{with {x 2} {with {x {+ x 2}} x}}', the scope of the first `x' is: ^^^^^^^)

* We then extended the AE evaluation rules:

```
    eval(...) = ... same as the AE rules ...
    eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
    eval(id) = error!
```

and noted the possible type problem.

* The above translated into a Racket definition for an `eval' function (with a hack to avoid the type issue):

```
(: eval : WAE -> Number)
;; evaluates WAE expressions by reducing them to numbers
(define (eval expr)
  (cases expr
    [(Num n) n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]
    [(Mul l r) (* (eval l) (eval r))]
    [(Div l r) (/ (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (Num (eval named-expr))))]
    [(Id name) (error 'eval "free identifier: ~s" name)]))
```

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)
```

Substitution Caches

[[[PLAI Chapter 5 (called "deferred substitutions" there)]]]

Evaluating using substitutions is very inefficient -- at each scope, we copy a piece of the program AST. This includes all function calls, which implies an impractical cost (function calls should be ***cheap***!).

To get over this, we want to use a cache of substitutions.

Basic idea: we begin evaluating with no cached substitutions, and collect them as we encounter bindings.

Implies another change for our evaluator: we don't really substitute identifiers until we get there, so when we reach an identifier it is no longer an error -- we must consult the substitution cache at that point.

Initial Implementation of Cache Functionality

First, we need a type for a substitution cache. For this we will use a list of lists of two elements each -- a name and its value ***FLANG***:

```
;; a type for substitution caches:
(define-type SubstCache = (Listof (List Symbol FLANG)))
```

We need to have an empty substitution cache, a way to extend it, and a way to look things up:

```
(: empty-subst : SubstCache)
(define empty-subst null)

(: extend : Symbol FLANG SubstCache -> SubstCache)
(define (extend id expr sc)
  (cons (list id expr) sc))

(: lookup : Symbol SubstCache -> FLANG)
(define (lookup name sc)
  (cond [(null? sc) (error 'lookup "no binding for ~s" name)]
        [(eq? name (first (first sc))) (second (first sc))]
        [else (lookup name (rest sc))]))
```

Actually, the reason to use such list of lists is that Racket has a built-in function called `**assq**' that will do this kind of search (`**assq**' is a search in an association list using `**eq?**' for the key comparison).

Example:

> (assq 3 (list (list 1 2) (list 3 4) (list 5 6)))

'(3 4)

> (assq 7 (list (list 1 2) (list 3 4) (list 5 6)))

#f

This is a version of `**lookup**' that uses `**assq**':

```
(: lookup : Symbol SubstCache -> FLANG)
(define (lookup name sc)
  (let ([cell (assq name sc)])
    (if cell
      (second cell)
      (error 'lookup "no binding for ~s" name))))
```

Formal Rules for Cached Substitutions

The formal evaluation rules are now different. Evaluation carries along a **"substitution cache"** that begins its life as ***empty***: so `**eval**' needs an **extra argument.**

Lookup rules:
We begin by writing the rules that deal with the cache, and use the above function names for simplicity -- the behavior of the three definitions can be summed up in a single rule for `lookup':

```
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:
Now, we can write the new rules for `**eval**'

```
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
```

1. Note that there is no mention of `**subst**' -- the whole point is that we don't really do substitution, but use the cache instead. The `**lookup**' rules and the places where `**extend**' is used -- replace `**subst**', and therefore specify our scoping rules.

2. Also note that the rule for `**call**' is still very similar to the rule for `**with**', but it looks like we have lost something -- the

interesting bit with substituting into `**fun**' expressions (i.e., whether we should or should not substitute).

Evaluating with Substitution Caches

Implementing the new `eval' is easy now -- it is extended in the same way that the formal `eval' rule is extended:

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

Again, note that we don't need `subst' anymore, but the rest of the code (the data type definition, parsing, and `arith-op') is exactly the same.

Finally, we need to make sure that `eval' is initially called with an ***empty cache***. This is easy to change in our main `run' entry point:

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

The full code (including the same tests, but not including formal rules for now) follows. Note that one test does not pass…

```
#lang pl

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

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

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

;; a type for substitution caches:
(define-type SubstCache = (Listof (List Symbol FLANG)))

(: empty-subst : SubstCache)
(define empty-subst null)

(: extend : Symbol FLANG SubstCache -> SubstCache)
(define (extend name val sc)
  (cons (list name val) sc))
```

```
(: lookup : Symbol SubstCache -> FLANG)
(define (lookup name sc)
  (let ([cell (assq name sc)])
    (if cell
      (second cell)
      (error 'lookup "no binding for ~s" name))))

(: 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 SubstCache -> FLANG)
;; evaluates FLANG expressions by reducing them to expressions
(define (eval expr sc)
  (cases expr
    [(Num n) expr]
    [(Add l r) (arith-op + (eval l sc) (eval r sc))]
    [(Sub l r) (arith-op - (eval l sc) (eval r sc))]
    [(Mul l r) (arith-op * (eval l sc) (eval r sc))]
    [(Div l r) (arith-op / (eval l sc) (eval r sc))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (extend bound-id (eval named-expr sc) sc))]
    [(Id name) (lookup name sc)]
    [(Fun bound-id bound-body) expr]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr sc)])
       (cases fval
         [(Fun bound-id bound-body)
          (eval bound-body
                (extend bound-id (eval arg-expr sc) sc))]
         [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) empty-subst)])
    (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 "{with {x 3}
                {with {f {fun {y} {+ x y}}}
                  {with {x 5}
                    {call f 4}}}}")
      => "???")
(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)
```

Dynamic and Lexical Scopes

This seems like it should work, and it even worked on a few examples, except for one which was hard to follow. Seems like **we have a bug**...

Now we get at a tricky issue that managed to be a problem for *__lots__* of language implementors, ***including the first version of Lisp***. Let's try to run the following expression -- try to figure out what it will evaluate to:

```
(run "{with {x 3}
        {with {f {fun {y} {+ x y}}}
          {with {x 5}
            {call f 4}}}}")
```

We expect it to return **7** (at least I do!), but instead it returns **9**... The question is -- ***should* it return 9**?

What we have arrived at is called "**dynamic scope**". Scope is determined by the dynamic run-time environment (which is represented by our substitution cache). This is *__almost always__* undesirable, as I hope to convince you.

Before we start, we define two options for a programming language:

- **Static Scope** (also called Lexical Scope): In a language with static scope, each identifier gets its value from the scope in which it was **defined** (not the one in which it is used).
- **Dynamic Scope:** In a language with dynamic scope, each identifier gets its value from the scope of its use (not its definition).

Racket uses lexical scope, our new evaluator uses dynamic, the old substitution-based evaluator was static etc.

Side-remark: Lisp began its life as a dynamically-scoped language. The artifacts of this were (sort-of) dismissed as an implementation bug. When Scheme was introduced, it was the first Lisp dialect that used strictly lexical scoping, and Racket is obviously doing the same. (Some Lisp implementations used dynamic scope for interpreted code and lexical scope for compiled code!) In fact, Emacs Lisp is the only *live* dialects of Lisp that is still dynamically scoped by default. Too see this, compare a version of the above code in Racket:

```
(let ((x 3))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 5))
      (f 4))))
```

and the Emacs Lisp version (which looks almost the same):

```
(let ((x 3))
  (let ((f (lambda (y) (+ x y))))
```

```
      (let ((x 5))
        (funcall f 4))))
```

which also happens when we use another function on the way:

```
  (defun blah (func val)
    (funcall func val))

  (let ((x 3))
    (let ((f (lambda (y) (+ x y))))
      (let ((x 5))
        (blah f 4))))
```

and note that renaming identifiers can lead to different code -- change that `val' to `x':

```
  (defun blah (func x)
    (funcall func x))

  (let ((x 3))
    (let ((f (lambda (y) (+ x y))))
      (let ((x 5))
        (blah f 4))))
```

and you get 8 because the argument name changed the `x' that the internal function sees!

Consider also this Emacs Lisp function:

```
  (defun return-x ()
    x)
```

which has no meaning by itself (`x' is unbound),

```
  (return-x)
```

but can be given a dynamic meaning using a `let':

```
  (let ((x 5)) (return-x))
```

or a function application:

```
  (defun foo (x)
    (return-x))

  (foo 5)
```

There is also a dynamically-scoped language in the course languages:

```
#lang pl dynamic

(define x 123)

(define (getx) x)

(define (bar1 x) (getx))
(define (bar2 y) (getx))

(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)
(test ((foo 0)) => 124)

;; and *much* worse:
(define (add x y) (+ x y))
(test (let ([+ *]) (add 6 7)) => 42)
```

Note how bad the last example gets: you basically cannot call any function and know in advance what it will do.

There are some cases where dynamic scope can be useful in that it allows you to "remotely" customize any piece of code. A good example of where this is taken to an extreme is Emacs: originally, it was based on an ancient Lisp dialect that was still dynamically scoped, but it retained this feature even when practically all Lisp dialects moved on to having lexical scope by default. The reason for this is that the danger of dynamic scope is also a way to make a very open system where **almost anything can be customized by changed it "remotely".** Here's a concrete example for a similar kind of dynamic scope usage that makes a very hackable and open system:

```
#lang pl dynamic

(define tax% 6.5)
(define (with-tax n)
  (+ n (* n (/ tax% 100))))

(with-tax 10) ; how much do we pay?
(let ([tax% 17.0]) (with-tax 10)) ; how much would we pay in Israel?

;; make that into a function
(define il-tax% 17.0)
(define (us-over-il-saving n)
  (- (let ([tax% il-tax%]) (with-tax n))
     (with-tax n)))
```

```
(us-over-il-saving 10)
;; can even control that: how much would we save if the tax in israel
;; went down one percent?
(let ([il-tax% (- il-tax% 1)]) (us-over-il-saving 10))
```

Obviously, this power to customize everything is also the main source of problems with getting no guarantees for code. A common way to get the best of both worlds is to have "controllable" dynamic scope. For example, Common Lisp also has lexical scope everywhere by default, but some variables can be declared as "special", which means that they are dynamically scoped. The main problem with that is that you can't tell when a variable is special by just looking at the code that uses it, so a more popular approach is the one that is used in Racket: all bindings are always lexically scoped, but there are "parameters" which are a kind of dynamically scoped value containers -- but they are bound to plain (lexically scoped) identifiers. Here's the same code as above, translated to Racket with parameters:

```
#lang racket

(define tax% (make-parameter 6.5))    ; create the dynamic container
(define (with-tax n)
  (+ n (* n (/ (tax%) 100))))         ; note how its value is accessed

(with-tax 10) ; how much do we pay?
(parameterize ([tax% 17.0]) (with-tax 10)) ; `parameterize', not `let'

;; make that into a function
(define il-tax% (make-parameter 17.0))
(define (us-over-il-saving n)
  (- (parameterize ([tax% (il-tax%)]) (with-tax n))
     (with-tax n)))

(us-over-il-saving 10)
(parameterize ([il-tax% (- (il-tax%) 1)]) (us-over-il-saving 10))
```

The main point here is that the points where a dynamically scoped value is used are under the programmer's control -- you cannot "customize" what `-' is doing, for example. This gives us back the guarantees that we like to have (= that code works), but of course these points are pre-determined, unlike an environment where everything can be customized including things that are unexpectedly useful.

(As a side-note, after many decades of debating this, Emacs has finally added lexical scope in its core language, but this is still determined by a flag -- a global `**lexical-binding**' variable.)

Dynamic versus Lexical Scope (offline discussion):

Back to the discussion of whether we should use dynamic or lexical scope:

- The most important fact is that we want to view programs as executed by the normal substituting evaluator. Our original motivation was to optimize evaluation only -- not to *change* the

semantics! It follows that we want the result of this optimization to behave in the same way. All we need is to evaluate:

```
(run "{with {x 3}
  {with {f {fun {y} {+ x y}}}
    {with {x 5}
      {call f 4}}}}")
```

In the original evaluator to get convinced that 7 should be the correct result (note also that the same code, when translated into Racket, evaluates to **7**).

(Yet, this is a very important optimization, which without it lots of programs become too slow to be feasible, so you might claim that you're fine with the modified semantics...)

- It does not allow using functions as objects, for example, we have seen that we have a functional representation for pairs:

```
(define (kons x y)
  (lambda (n)
    (match n
      ['first  x]
      ['second y]
      [else (error ...)])))

(define my-pair (kons 1 2))
```

If this is evaluated in a dynamically-scoped language, we do get a function as a result, but the values bound to x and y are now gone!
Using the substitution model we substituted these values in, but now they were only held in a cache which has no entries for them...

In the same way, currying would not work, our nice `deriv' function would not work etc etc etc.

- Makes reasoning impossible, because any piece of code behaves in a way that *cannot* be predicted until run-time. For example, if dynamic scoping was used in Racket, then you wouldn't be able to know what this function is doing:

```
(define (foo)
  x)
```

As it is, it will cause a run-time error, but if you call it like this:

```
(let ([x 1])
  (foo))
```

Then, it will return 1, and if you later do this:

```
(define (bar x)
  (foo))
```

```
(let ([x 1])
  (bar 2))
```

Then, you would get 2!

These problems can be demonstrated in Emacs Lisp too, but Racket goes one step further -- it uses the same rule for evaluating a function as well as its values (Lisp uses a different name-space for functions). Because of this, you cannot even rely on the following function:

```
(define (add x y)
  (+ x y))
```

to always add x and y! -- A similar example to the above:

```
(let ([+ -])
  (add 1 2))
```

would return -1!

Many so-called "scripting" languages begin their lives with dynamic scoping. The main reason, as we've seen, is that implementing it is extremely simple (no, *nobody* does substitution in the real world! (Well, *almost* nobody...)).

Another reason is that these problems make life impossible if you want to use functions as object like you do in Racket, so you notice them very fast -- but in a `normal' language without first-class functions, problems are not as obvious.

* For example, bash has `local' variables, but they have dynamic scope:

```
x="the global x"
print_x() { echo "The current value of x is \"$x\""; }
foo() { local x="x from foo"; print_x; }
print_x; foo; print_x
```

Perl began its life with dynamic scope for variables that are declared `local':

```
$x="the global x";
sub print_x { print "The current value of x is \"$x\"\n"; }
sub foo { local($x); $x="x from foo"; print_x; }
print_x; foo; print_x;
```

When faced with this problem, "the Perl way" was, obviously, not to remove or fix features, but to pile them up -- so local *still* behaves in this way, and now there is a `my' declaration which achieves proper lexical scope...

There are other examples of languages that changed, and languages that want to change (e.g, nobody likes dynamic scope in Emacs Lisp, but there's just too much code now).

* This is still a tricky issue, like any other issue with bindings. For example, googling got me quickly to this site:

  http://www.hetland.org/python/instant-python.php

  which is confused about what "dynamic scoping" is... It claims that Python uses dynamic scope (Search for "Python uses dynamic as opposed to lexical scoping"), yet python always used lexical scope rules, as can be seen by translating their code to Racket (ignore side-effects in this computation):

```
(define (orange-juice)
  (* x 2))
(define x 3)
(define y (orange-juice)) ; y is now 6
(define x 1)
(define y (orange-juice)) ; y is now 2
```

  or by trying this in Python:

```
def orange_juice():
  return x*2
def foo(x):
  return orange_juice()
foo(2)
```

  The real problem of python (pre 2.1, and pre 2.2 without the funny
    from __future__ import nested_scope
  line) is that it didn't create closures, which we will talk about shortly.

* Another example, which is an indicator of how easy it is to mess up your scope is the following Ruby bug -- running in `irb':

```
% irb
irb(main):001:0> x = 0
=> 0
irb(main):002:0> lambda{|x| x}.call(5)
=> 5
irb(main):003:0> x
=> 5
```

  (This is a bug due to weird scoping rules for variables, which was apparently fixed in newer versions of Ruby. See
    http://ceaude.twoticketsplease.de/articles/
      ruby-and-the-principle-of-unwelcome-surprise.html
  for details on this and on other surprises in Ruby...)

* Another thing to consider is the fact that compilation is something that you do based only on the lexical structure of programs, since compilers never actually run code. This means that dynamic scope makes compilation close to impossible.

* There are some advantages for dynamic scope too. Two notable ones are:

  - Dynamic scope makes it easy to have a "configuration variable"

easily change for the extend of a calling piece of code (this is used extensively in Emacs, for example). The thing is that usually we want to control which variables are "configurable" in this way, statically scoped languages like Racket often choose a separate facility for these. To rephrase the problem of dynamic scoping, it's that *all* variables are modifiable.

The same can be said about functions: it is sometimes desirable to change a function dynamically (for example, see "Aspect Oriented Programming"), but if there is no control and all functions can change, we get a world where no code can every be reliable.

- It makes recursion immediately available -- for example,

```
{with {f {fun {x} {call f x}}} {call f 0}}
```

is an infinite loop with a dynamically scoped language. But in a lexically scoped language we will need to do some more work to get recursion going.

Implementing Lexical Scope: Closures and Environments

So how do we fix this?

The root of the problem: the new evaluator does not behave in the same way as the substituting evaluator.

In the old evaluator: functions can behave as objects that **remember values**. For example, when we do this:

```
{with {x 1}
  {fun {y}
    {+ x y}}}
```

The result was a function value, which actually was the syntax object for this:

```
{fun {y} {+ 1 y}}
```

If we call this function from someplace else like:

```
{with {f {with {x 1} {fun {y} {+ x y}}}}
  {with {x 2}
    {call f 3}}}
```

It is clear what the result will be: ***f*** is bound to a function that adds **1** to its input, so in the above the later binding for `**x**' has no effect at all.

With the caching evaluator: the value of

```
{with {x 1}
  {fun {y}
    {+ x y}}}
```

is simply:

```
{fun {y} {+ x y}}
```

There is no place where we save the 1 -- *that's* the root of our problem. (That's also what makes people suspect that using `lambda' in Racket and any other functional language involves some inefficient code-recompiling magic.) In fact, we can verify that -- by inspecting the returned value, and seeing that it does contain a free identifier.

Solution: We already know that we need an object that contains the body and the argument list (like the function syntax object).
Since we don't do substitutions – we need in addition to "**remember**" that we still need to substitute **x** by **1**.

This means that the pieces of information we need to know are:

- **formal argument(s):** {**y**}
- **body:** {**+ x y**}
- **pending substitutions:** [**1/x**]

That last piece has the missing **1**.
The resulting object is called a `**closure**' because it closes the function body over the substitutions that are still pending (its **environment**).

A first change: The info that we need to hold – **closures** (functions) need all three fields above (unlike the `**Fun**' case for the syntax object).

A second place that needs changing is the **when functions are called**. When we're done evaluating the `**call**' arguments (the function value and the argument value) but before we apply the function we have two ***values*** -- there is no more use for the current substitution cache at this point: we have finished dealing with all substitutions that were necessary over the current expression -- we now continue with evaluating the body of the function, with the new substitutions for the formal arguments and actual values given. But the body itself is the same one we had before -- which is the previous body with its suspended substitutions that we *still* did not do.

Rewriting evaluation rules -- all are the same except for evaluating a `**fun**' form and a `**call**' form:

```
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
```

Environments

These substitution caches are a little more than "just a cache" now -- they actually hold an "**environment**" of substitutions in which expressions should be evaluated. So, we will switch to the common "**environment**" name now:

```
  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
```

In case you find this easier to follow, the "flat algorithm" for evaluating a `**call**' is:

1. f := evaluate E1 in env1
2. if f is not a <{fun ...},...> closure then error!
3. a := evaluate E2 in env1
4. new_env := extend env_of(f) by mapping arg_of(f) to a
5. evaluate (and return) body_of(f) in new_env

Note how the scoping rules that are implied by this definition match the scoping rules that were implied by the substitution-based rules. (It should be possible to prove that they are the same.)

The changes to the code are almost trivial, except that we need a way to represent **<{fun {x} Ef}, env>** pairs.

The closure/environment interpreter

A new type for values (also for functions):
The implication of this change is that we now cannot use the same type for function syntax and function values since function values have more than just syntax. There is a simple solution to this -- we never do any substitutions now, so we don't need to translate values into expressions -- we can come up with a **new type for values, separate from the type of abstract syntax trees.**

When we do this, we will also fix our hack of using **FLANG** as the type of values: this was merely a convenience since the AST type had cases for all kinds of values that we needed. (In fact, you should have noticed that Racket does this too: numbers, strings, booleans, etc are all used by both programs and syntax representation (s-expressions) -- but note that function values are ***not*** used in syntax.) We will now implement a separate `**VAL**' type for runtime values.

Implementing a new type for run-time values

ENV type:

First, we need now a type for such environments -- we can use `**Listof**' for this:

```
;; a type for environments:
(define-type ENV = (Listof (List Symbol VAL)))
```

However, we can just as well define a new type for environment values:

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

Re-implementing `**lookup**' is now simple:

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

... We don't need `**extend**' because **we get `Extend' from the type definition**, and we also get `**(EmptyEnv)**' instead of `**empty-subst**'.

VAL type:

We now use this with the new type for values -- two variants of these:

```
(define-type VAL
  [NumV Number]
  [FunV Symbol FLANG ENV]) ; arg-name, body, scope
```

The new implementation of `eval': using the new type and implementing **lexical scope:**

```
(: 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); We expect a closure
          (eval bound-body
                (Extend bound-id (eval arg-expr env) f-env))]
         [else (error 'eval "`call' expects a function, got: ~s"
                      fval)]))]))
```

We also need to update `**arith-op**' to use **VAL** objects. The full code follows -- it now passes all tests, including the example that we used to find the problem.

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

;; The Flang interpreter, using environments

#lang pl

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

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

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

(: parse-sexpr : Sexpr -> FLANG)
;; to convert s-expressions into FLANGs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        (With name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(cons 'fun more)
     (match sexpr
```

```
        [(list 'fun (list (symbol: name)) body)
         (Fun name (parse-sexpr body))]
        [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

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

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

(define-type 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)
```

Fixing an overlooked Bug

Incidentally, this version fixes a bug we had previously in the **substitution version** of FLANG:

```
(run "{with {f {fun {y} {+ x y}}}
        {with {x 7}
          {call f 1}}}")
```

This bug was due to our naive `subst', which doesn't avoid capturing renames. But note that since that version of the evaluator makes its way from the outside in, there is no difference in semantics for *valid* programs -- ones that don't have free identifiers.

(Reminder: This was ***not*** a dynamically scoped language, just a bug that happened when `**x**' wasn't substituted away before `**f**' was replaced with something that refers to `**x**'.)

Previously on PL…

Dynamic vs. static:

```
#lang pl dynamic



  (define x 123)

  (define (getx) x)

  (define (bar1 x) (getx))
  (define (bar2 y) (getx))

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

  (define (foo x) (define (helper) (+ x 1)) helper)
  (test ((foo 0)) => ?)


  ;; and much worse:
  (define (add x y) (+ x y))
  (test (let ([+ *]) (add 6 7)) => ?)

  (test ((let ([v 5])
           (let ([foo (lambda (y) (+ v y))])
             foo)) 3) => ?)


---<<<FLANG-ENV>>>-----------------------------------------------------

  ;; The Flang interpreter, using environments

  #lang pl

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

  Evaluation rules:
    eval(N,env)                = N
    eval({+ E1 E2},env)        = eval(E1,env) + eval(E2,env)
```

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

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

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

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

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

(define-type 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)
```

Fixing an overlooked Bug

Incidentally, this version fixes a bug we had previously in the **substitution version** of FLANG:

```
(run "{with {f {fun {y} {+ x y}}}
        {with {x 7}
          {call f 1}}}")
```

This bug was due to our naive `subst', which doesn't avoid capturing renames. But note that since that version of the evaluator makes its way from the outside in, there is no difference in semantics for *valid* programs -- ones that don't have free identifiers.

(Reminder: This was ***not*** a dynamically scoped language, just a bug that happened when `**x**' wasn't substituted away before `**f**' was replaced with something that refers to `**x**'.)

Lazy Evaluation: Using a Lazy Racket

[[[PLAI Chapter 7 (done with Haskell)]]]

Lazy evaluation is an evaluation strategy that holds the evaluation of an expression until its value is needed. In addition, it avoids repeated evaluation.

For this part, we will use a new language, Lazy Racket.

```
#lang pl lazy
```

As the name suggests, this is a version of the normal (untyped) Racket language that is lazy.

First of all, let's verify that this is indeed a lazy language:

```
> (define (foo x) 3)
> (foo (+ 1 "2"))
3
```

That went without a problem -- the argument expression was indeed not evaluated. In this language, you can treat all expressions as future `**promises**' to evaluate. There are certain points, where such promises are actually `**forced**', all of these stem from a need to print a resulting value:

```
> (+ 1 "2")
+: expects type <number> as 2nd argument, given: "2"; other
arguments were: 1
```

The expression by itself only generates a promise, but when we want to print it, this promise is forced to evaluate -- this forces the addition, which forces its arguments (plain values rather than computation promises), and at this stage we get an error. (If we never want to see any results, then the language will never do anything at all.) So a promise is forced either when a value printout is needed, or if it is needed to recursively compute a value to print:

```
> (* 1 (+ 2 "3"))
+: expects type <number> as 2nd argument, given: "3"; other
arguments were: 2
```

Note that the error was raised by the internal expression: the outer expression uses `*', and `+' requires actual values not promises.

Another example, which is now obvious, is that we can now define an `if' function:

```
> (define (my-if x y z) (if x y z))
> (my-if (< 1 2) 3 (+ 4 "5"))
3
```

Actually, in this language `if', `and', and `or' are all function values instead of special forms:

```
> (list if and or)
(#<procedure:if> #<procedure:and> #<procedure:or>)
```

(By now, you should know that these have no value in Racket - using them like this in plain will lead to syntax errors.) There are some primitives that do not force their arguments. Constructors fall in this category, for example `cons' and `list':

```
> (define a (list 1 2 (+ 3 "4") (* 5 6)))
```

Nothing -- the definition simply worked, but that's expected, since nothing is printed. If we try to inspect this value, we can get some of its parts, provided we do not force the bogus one:

```
> (first a)
1
> (second a)
2
> (fourth a)
30
> (third a)
+: expects type <number> as 2nd argument, given: "4"; other
arguments
were: 3
```

The same holds for cons:

```
> (second (cons 1 (cons 2 (first null))))
2
```

Now if this is the case, then how about this:

```
> (define ones (cons 1 ones))
```

Everything is fine, as expected -- but what is the value of `ones' now?
Clearly, it is a list that has 1 as its first element:

```
> (first ones)
1
```

But what do we have in the tail of this list? We have `ones' which we already know is a list that has 1 in its first place -- so following Racket's usual rules, it means that the second element of `ones' is, again, 1. If we continue this, we can see that `ones' is, in fact, an
infinite list of 1s:

```
> (second ones)
1
> (fifth ones)
1
```

In this sense, the way `define' behaves is that it defines a true equation: if ones is defined as (cons 1 ones), then the real value does satisfy

```
(equal? ones (cons 1 ones))
```

which means that the value is the fixpoint of the defined expression.

We can use `append' in a similar way:

```
> (define foo (append (list 1 2 3) foo))
> (fourth foo)
1
```

This looks like it has some common theme with the discussion of implementing recursive environments -- it actually demonstrates that in this language, `letrec' can be used for "simple" values too. First of all, a side note -- here an expression that indicated a bug in our substituting evaluator:

```
> (let ([x (list y)])
    (let ([y 1])
      x))
reference to undefined identifier: y
```

When our evaluator returned `1' for this, we noticed that this was a bug: it does not obey the lexical scoping rules. As seen above, Lazy Racket is correctly using lexical scope. Now we can go back to the use
of `letrec' -- what do we get by this definition:

```
> (define twos (let ([xs (cons 2 xs)]) xs))
```

we get an error about `xs' being undefined.

`xs' is unbound because of the usual scope that `let' uses. How can we make this work? -- We simply use `letrec':

```
> (define twos (letrec ([xs (cons 2 xs)]) xs))
> (first twos)
2
```

As expected, if we try to print an infinite list will cause an infinite
loop, which DrRacket catches and prints in that weird way:

```
> twos
#0=(2 . #0#)
```

How would we inspect an infinite list? We write a function that returns
part of it:

```
> (define (take n l)
    (if (or (<= n 0) (null? l))
      null
      (cons (first l) (take (sub1 n) (rest l)))))
> (take 10 twos)
(2 2 2 2 2 2 2 2 2 2)
> (define foo (append (list 1 2 3) foo))
> (take 10 foo)
(1 2 3 1 2 3 1 2 3 1)
```

Dealing with infinite lists can lead to lots of interesting things, for
example:

```
> (define fibs (cons 1 (cons 1 (map + fibs (rest fibs)))))
> (take 10 fibs)
(1 1 2 3 5 8 13 21 34 55)
```

Too see how it works, see what you know about `fibs[n]' which will be
our notation for the nth element of `fibs' (starting from 1):

```
fibs[1] = 1  because of the first `cons'
fibs[2] = 1  because of the second `cons'
```

and for all n>2:

```
fibs[n] = (map + fibs (rest fibs))[n-2]
        = fibs[n-2] + (rest fibs)[n-2]
        = fibs[n-2] + fibs[n-2+1]
        = fibs[n-2] + fibs[n-1]
```

so it follows the exact definition of Fibonacci numbers.

==========
>>> Lazy Evaluation: Some Issues

There are a few issues that we need to be aware of when we're dealing with a lazy language. First of all, remember that our previous attempt at lazy evaluation has made

```
{with {x y}
  {with {y 1}
    x}}
```

evaluate to 1, which does not follow the rules of lexical scope. This is *not* a problem with lazy evaluation, but rather a problem with our naive implementation. We will shortly see a way to resolve this problem. In the meanwhile, remember that when we try the same in Lazy Racket we do get the expected error:

```
> (let ([x y])
    (let ([y 1])
      x))
reference to undefined identifier: y
```

A second issue is a subtle point that you might have noticed when we played with Lazy Racket: for some of the list values we have see a "." printed. This is part of the usual way Racket displays an "improper list" -- any list that does not terminate with a null value. For example, in plain Racket:

```
> (cons 1 2)
(1 . 2)
> (cons 1 (cons 2 (cons 3 4)))
(1 2 3 . 4)
```

In the dialect that we're using in this course, this is not possible.
The secret is that the `cons' that we use first checks that its second argument is a pro per list, and it will raise an error if not. So how come Lazy Racket's `cons' is not doing the same thing? The problem is that to know that something is a proper list, we will have to force it, which will make it not behave like a constructor.

[As a side note, we can achieve some of this protection if we don't
insist on immediately checking the second argument completely, and
instead we do the check when needed -- lazily:

```
  (define (safe-cons x l)
    (cons x (if (list? l) l (error "poof"))))

]
```

Finally, there are two consequences of using a lazy language that make
it more difficult to debug (or at lease take some time to get used to).
First of all, control tends to flow in surprising ways. For example,
enter the following into DrRacket, and run it in the normal language
level for the course:

```
                    (define (foo3 x)
                      (/ x "1"))
  (define (foo2 x)
    (foo3 x))
                    (define (foo1 x)
                      (list (foo2 x)))
  (define (foo0 x)
    (first (foo1 x)))

           (+ 1 (foo0 3))
```

In the normal language level, we get an error, and red arrows that show
us how where in the computation the error was raised. The arrows are
all expected, except that `foo2' is not in the path -- why is that?
Remember the discussion about tail-calls and how they are important in
Racket since they are the only tool to generate loops? This is what
we're seeing here: `foo2' calls `foo3' in a tail position, so there is
no need to keep the `foo2' context anymore -- it is simply replaced by
`foo3'. Now switch to Lazy Racket and re-run -- you'll see a single
arrow, skipping over everything and going straight to the erroneous
expression. What's the problem? The call of `foo0' creates a promise
that is forced in the top-level expression, that simply returns the
`first' of the `list' that `foo1' created -- and all of that can be done
without forcing the `foo2' call. Going this way, the computation is

finally running into an error *after* the calls to `foo0', `foo1', and
`foo2' are done -- so we get the seemingly out-of-context error.

Finally, there are also potential problems when you're not careful about memory use. A common technique when using a lazy language is to
generate an infinite list and pull out its Nth element. For example, to
compute the Nth Fibonacci number, we've seen how we can do this:

```
  (define fibs (cons 1 (cons 1 (map + fibs (rest fibs)))))
  (define (fib n) (list-ref fibs n))
```

and we can also do this (reminder: `letrec' is the same as an internal
definition):

```
  (define (fib n)
    (letrec ([fibs (cons 1 (cons 1 (map + fibs (rest fibs))))])
      (list-ref fibs n)))
```

but the problem here is that when `list-ref' is doing its way down the
list, it might still hold a reference to `fibs', which means that as the
list is forced, all intermediate values are held in memory. In the
first of these two, this is guaranteed to happen since we have a binding
that points at the head of the `fibs' list. With the second form things
can be confusing: it might be the case that our language implementation
is smart enough to see that `fibs' is not really needed any more and
release the offending hold, but even if this is the case, tricky
situations are hard to avoid. (Side note: Racket didn't use to do this
optimization, but now it does.)

This is also tricky no matter how smart the compiler is. For example,
compare this:

```
  (define (foo)
    (define nats1 (cons 1 (map add1 nats1)))
    (define nats2 (cons 1 (map add1 nats2)))
    (or (equal? nats1 nats2)
        (error "the list that begins with ~s is bad" (first
nats1))))
```

to this:

```
  (define (foo)
    (define nats1 (cons 1 (map add1 nats1)))
    (define nats2 (cons 1 (map add1 nats2)))
    (let ([fst (first nats1)])
      (or (equal? nats1 nats2)
          (error "the list that begins with ~s is bad" fst))))
```

==========

======= Lazy Evaluation – Advantages

- It allows the language runtime to discard sub-expressions that are not directly linked to the final result of the expression – could be different in different executions.
- It reduces the time complexity of an algorithm by discarding the temporary computations and conditionals.
- It allows the programmer to access components of data structures out-of-order after initializing them, as long as they are free from any circular dependencies.
- It is best suited for loading data that will be infrequently accessed.

Lazy Evaluation – Drawbacks

- It forces the language runtime to hold the evaluation of sub-expressions until it is required in the final result by creating **thunks** (delayed objects).
- Sometimes it increases space complexity of an algorithm.
- It is very difficult to find its performance because it contains thunks of expressions before their execution.