CS 7400 November 16-18, 2009
Dr. Wand Readings:

Lecture 8: Specifying the Behavior of Programs

Key Concepts:

- Semantics
  - Denotational
  - Operational
    - Small-step (reduction) semantics
    - Big-step (evaluation) semantics
- Substitution
- Reductions
  - Context
  - Redex
  - Reductum

Our goal is to learn how to specify the behavior of programs in a programming language. This is called *semantics*.

There are two basic techniques for writing down the semantics of a programming language:

Denotational Semantics In this technique, each expression in a program is assigned a value in some mathematical domain. This is done by structural induction on the program. For a whole program, the value will correspond to the result of executing the program. Doing this properly can require some serious mathematics.

Operational Semantics In this technique, we consider axioms and rules of inference that will predict the results of executing the program. This technique has many similarities to logic. In general, it requires only simple mathematics. However, it requires some discipline to use appropriately.

There are additional tradeoffs between denotational and operational semantics, which are beyond the scope of this course.

We will concentrate on operational semantics. Operational semantics comes in two general flavors: *small-step* and *big-step*.

We will illustrate both these techniques on some small examples, and then use them to explore a somewhat larger example.

Our first example language is the language of arithmetic expressions:

Arithmetic Expressions

$$\begin{array}{l} n \in \text{Integers} \\ e ::= n \mid -(e_1, e_2) \end{array}$$

We will write a *small-step* or *reduction* semantics. The basic rule is to replace a difference expression with its value:

RED DIFF
$$-(n, m) \rightarrow p \quad \text{where } p = n - m$$

This is called a *reduction rule*, and an instance of its use is called a *reduction*. The left-hand side of an instance is called the *redex*, and the right-hand side is called the *reductum*.

But when and how can we apply this rule? As it stands, it can be applied only to very simple expressions, as in

$$-(44, 11) \rightarrow 33$$
$$-(33, 3) \rightarrow 30$$

But we can't paste these together to deduce that the value of $-(-(44, 11), 3)$ is 30: RED DIFF doesn't apply to $-(-(44, 11), 3)$.

To do this, we need some *congruence* or *context* rules that allow us to apply this rule to subexpressions of an expression:

$$\textsc{Cong Diff L}\ \frac{e_1 \to e_1'}{-(e_1, e_2) \to -(e_1', e_2)}$$

$$\textsc{Cong Diff R}\ \frac{e_2 \to e_2'}{-(n, e_2) \to -(n, e_2')}$$

These rules say that I can always take a reduction in the left-hand argument of a difference, and that I can take a reduction in the right-hand argument if the left-hand argument is an integer. Thus we can deduce:

$$\frac{-(44, 11) \to 33}{-(-(44, 11), 3) \to -(33, 3)}$$

or

$$\frac{\dfrac{-(44, 11) \to 33}{-(-(44, 11), 3) \to -(33, 3)}}{-(77, -(-(44, 11), 3)) \to -(77, -(33, 3))}$$

Taken together, these rules show us where we can take the reduction step in any arithmetic expression in our language.

But we still don't have enough rules to deduce that the value of $-(-(44,11),3)$ is 30. This requires multiple steps. So, we take the reflexive transitive closure of $\rightarrow$:

$$\frac{}{e \rightarrow^* e}\ \textsc{Red* ID}$$

$$\frac{e_1 \rightarrow e_2 \qquad e_2 \rightarrow^* e_3}{e_1 \rightarrow^* e_3}\ \textsc{Red* Step}$$

Now we can paste these derivations together to show that the value of $-(-(44,11),3)$ is 30:

$$\dfrac{\dfrac{-(44,11) \rightarrow 33}{-(-(44,11),3) \rightarrow -(33,3)} \qquad \dfrac{-(33,3) \rightarrow 30 \qquad 30 \rightarrow^* 30}{-(33,3) \rightarrow^* 30}}{-(-(44,11),3) \rightarrow^* 30}$$

In general, we'll skip the details of the congruence rules, and simply write:

$$\begin{aligned} &-(-(44,11),3) \\ &\rightarrow -(33,3) \\ &\rightarrow 30 \end{aligned}$$

Writing out the congruence rules can be a pain. An alternative, due to Felleisen, is to define *reduction contexts*. A *context* is an expression with a hole in it (denoted [ ]). Given a context E, we write $E[e]$ for the result of putting expression e in the hole of E.

We use a grammar to define the set of contexts in which reductions can take place:

[Note: these E's should really be K's, since they're continuations!]

Reduction Contexts

$$E ::= [\,] \mid -(E, e) \mid -(n, E)$$

RED DIFF
$E[-(n,m)] \rightarrow E[p]$ where $p = n - m$

Now the reduction process is split into three parts:

1. *Analyze* your term e into a reduction context E containing a redex l in the hole.
2. *Reduce* the redex l to the reductum r.
3. *Reconstruct* the next term e' by plugging r into the hole of E.

Let's see how that works for a slightly larger example: $-(-(44,11),-(20,1))$:

$e = E[l]$	E	l	r	$E[r]$
$-(-(44,11),-(20,1))$	$-([\,],-(20,1))$	$-(44,11)$	33	$-(33,-(20,1))$
$-(33,-(20,1))$	$-(33,[\,])$	$-(20,1)$	19	$-(33,19)$
$-(33,19)$	$[\,]$	$-(33,19)$	14	14

The grammar for reduction contexts

$$E ::= [\,] \mid -(E,e) \mid -(n,E)$$

makes it clear how to analyze a term in our language. First analyze the left-hand operand. If you find the redex there, you've successfully analyzed the term.

If the left-hand side is a number, then analyze the right-hand operand.

If both operands are numbers, then you can reduce the entire term using RED DIFF.

8.1 Implementation

Let's look at how this can be implemented. Here are some pieces of `core-lang/*.scm`. Note the use of the => idiom of Scheme.

```
;; step-once : closed-expression -> (maybe closed-expression)
;; reduces by one step, else returns #f
(define step-once
  (lambda (exp)
    (cases expression exp

      (const-exp (n) #f)

      (diff-exp (exp1 exp2)
        (cond
          ((step-once exp1)           ; try to step exp1
           => (lambda (new-exp1) (diff-exp new-exp1 exp2)))
          ((step-once exp2)           ; exp1 done, try exp2
           => (lambda (new-exp2) (diff-exp exp1 new-exp2)))
          (else                       ; args done, do the arithmetic
            (reduce-diff-redex exp1 exp2))))
      ...
      )))

;; reduce-diff-redex : exp * exp -> (maybe const-exp)
;; later on, we'll use the fact that this returns a const-exp and
;; not just an exp.
(define reduce-diff-redex
  (lambda (exp1 exp2)
    (if (and (const-exp? exp1) (const-exp? exp2))
      (const-exp (- (exp->const exp1) (exp->const exp2)))
      ;; can't reduce if operands are not constants
      #f)))
```

```
;; reduce-to-value : closed-expression -> closed-value
(define reduce-to-value
  (lambda (exp)
    (if *trace* (pretty-print exp))
    (cond
      ;; step-once either returns the next exp or #f
      ((step-once exp)
       => (lambda (next-exp) (reduce-to-value next-exp)))
      ;; or just ((step-once exp) => reduce-to-value) !!!
      (else exp))))
```

```
> (toggle-trace)
#t
> (run "-(-(44,11),3)")
#(struct:diff-exp
  #(struct:diff-exp
     #(struct:const-exp 44)
     #(struct:const-exp 11))
  #(struct:const-exp 3))
#(struct:diff-exp
  #(struct:const-exp 33)
  #(struct:const-exp 3))
#(struct:const-exp 30)
#(struct:const-exp 30)
> (run "-(77,-(-(44,11),3))")
#(struct:diff-exp
  #(struct:const-exp 77)
  #(struct:diff-exp
    #(struct:diff-exp
       #(struct:const-exp 44)
       #(struct:const-exp 11))
    #(struct:const-exp 3)))
#(struct:diff-exp
  #(struct:const-exp 77)
  #(struct:diff-exp
     #(struct:const-exp 33)
     #(struct:const-exp 3)))
#(struct:diff-exp
  #(struct:const-exp 77)
  #(struct:const-exp 30))
#(struct:const-exp 47)
#(struct:const-exp 47)
>
```

There's more to be done, of course, to have a complete implementation; we'll talk about that later.

8.2 The Pattern

To set up a reduction semantics, we proceed as follows:

1. We specify a set V of *values*. These are the terms in the language that signify completed computations.
2. We specify a set of *reduction contexts* E that identify the positions in a term where a reduction may be applied.
3. We specify a set of *reduction rules* (L, R) where L and R are (closed) terms of our language.
4. To evaluate a term, we analyze it into the form $E[L]$ and reduce it to $E[R]$. We continue until we reach a term that is a value.

For our language of arithmetic expressions we have:

Arithmetic Expressions

$$V ::= n$$
$$E ::= [\,] \mid -(E, e) \mid -(n, E)$$

REDDIFF
$$-(n, m) \rightarrow p \quad \text{where } p = n - m$$

8.3 Procedures

Let's add procedures to our core language. We'll add unary procedures only. The language will be:

Core Language: Expressions and Values

n	Integers
x	Variables
$e ::= n \mid -(e_1, e_2)$	Expressions
$\quad \mid x \mid (e_1\ e_2) \mid (\lambda x.e)$	
$v ::= n \mid (\lambda x.e)$	Values

An expression will be either an arithmetic expression, as before, or a variable or an application or an abstraction (a λ-expression).

We will follow the standard rule and regard any abstraction as a value, along with any number.

An expression is *closed* iff it has no free variables.

The basic rule for procedures will be β*-reduction*:[1]

$$((\lambda x.e)\ v) \rightarrow e[v/x]$$

where $e[v/x]$ means the expression e with v substituted for all free occurrences of x.

Let's do an example to see how this is going to work:

$$\begin{aligned}
&((\lambda f.(f\ 3))\ (\lambda x.-(x,1))) \\
&\rightarrow((\lambda x.-(x,1))\ 3) \\
&\rightarrow -(3,1) \\
&\rightarrow 2
\end{aligned}$$

[1] Technically, this is called β_v or β*-value* reduction, since the operand must be a value. In full β-reduction, the operand can be any expression.

Here's another. In each case, I've underlined the redex.

$$\underline{(((\lambda f.(\lambda x.(f\ (f\ x))))\ (\lambda n.-(n,1)))}\ -(33,11))$$

$$\rightarrow((\lambda x.((\lambda n.-(n,1))\ ((\lambda n.-(n,1))\ x)))\ \underline{-(33,11)})$$

$$\rightarrow((\lambda x.((\lambda n.-(n,1))\ ((\lambda n.-(n,1))\ x)))\ 22)$$

$$\rightarrow((\lambda n.-(n,1))\ \underline{((\lambda n.-(n,1))\ 22)})$$

$$\rightarrow((\lambda n.-(n,1))\ \underline{-(22,1)})$$

$$\rightarrow((\lambda n.-(n,1))\ 21)$$

$$\rightarrow\underline{-(21,1)}$$

$$\rightarrow 20$$

This sequence shows our intended order of reductions for an application:

1. Reduce the operator (to a λ-abstraction)
2. Then reduce the operand
3. Then apply beta-reduction

These rules are in addition to the rules for arithmetic expressions.
We can write this in the language of reduction contexts as:

Core Language: Reduction Contexts and Reductions

$$
\begin{array}{lll}
E ::= [\,] & & \text{Reduction Contexts} \\
\quad \mid\ -(E, e) \mid -(n, E) & & \text{CXT ARITH 1,2} \\
\quad \mid\ (E\ e) & & \text{CXT RATOR} \\
\quad \mid\ (v\ E) & & \text{CXT RAND}
\end{array}
$$

RED DIFF
$-(n, m) \rightarrow p \quad \text{where } p = n - m$

RED BETA
$((\lambda x.e)\ v) \rightarrow e[v/x]$

8.4 Substitution

So far we've been informal about substitution. We define substitution $e[e'/x]$, where e' is a *closed* expression, as follows:

Substitution

$$
\begin{array}{lll}
n[e'/x] & = n & \\
-(e_1, e_2)[e'/x] & = -(e_1[e'/x], e_2[e'/x]) & \\
x[e'/x] & = e' & \\
y[e'/x] & = y & y \neq x \\
(e_1\ e_2)[e'/x] & = (e_1[e'/x]\ e_2[e'/x]) & \\
(\lambda x.e)[e'/x] & = (\lambda x.e) & \\
(\lambda y.e)[e'/x] & = (\lambda y.e[e'/x]) & y \neq x
\end{array}
$$

Note how the next-to-last line respects the local binding of x.

This definition works only for e' closed. If e' were open (ie, it contained some free variables), it might contain a free occurrence of y, which then would be captured by the (λy.-). So if e' could be open, we would have to rename the bound variable y to some fresh variable z:

$$(\lambda y.e)[e'/x] = (\lambda z.(e[z/y])[e'/x])$$

Luckily, we won't have to worry about this case, because in our systems the substituend will always be closed.

8.5 Implementation

Now that we have our complete core language, we'll look at the implementation in more detail.

`lang.scm` deals with the language and substitution. It contains the usual parser stuff, plus substitution:

```
;;;;;;;;;;;;;;;; substitution ;;;;;;;;;;;;;;;;

;; subst-closed : expression * symbol * closed-expression
;;                  -> expression
;; This doesn't do renaming under a proc-exp, so it won't work if
;; new-exp isn't closed.

(define subst-closed
  (lambda (exp0 var new-exp)
    (let recur ((exp exp0))
      ;; recur : expression -> expression
      ;; (recur exp) = (subst-closed exp var new-exp)
      ;; (printf "exp = ~s~%" exp)
      (cases expression exp
        (const-exp (n) exp)
        (diff-exp (exp1 exp2) (diff-exp (recur exp1) (recur exp2)))

        (var-exp (var1) (if (eqv? var1 var) new-exp exp))
        (proc-exp (var1 body)
          (if (eqv? var1 var) exp
            (proc-exp var1 (recur body))))
        (app-exp (rator rand)
          (app-exp (recur rator) (recur rand)))))))
```

Note the use of the named-LET idiom of Scheme.

`smallstep.scm` contains the reduction engine.

```
(module smallstep (lib "eopl.ss" "eopl")

  (require "lang.scm")                  ; for AST defns.
  (require "reductions.scm")

  (provide reduce-program-to-value toggle-trace)

  ;; reduce-program-to-value : program -> value
  (define reduce-program-to-value
    (lambda (pgm)
      (cases program pgm
        (a-program (exp) (reduce-to-value exp)))))

  ;; reduce-to-value : closed-expression -> closed-value
  (define reduce-to-value
    (lambda (exp)
      (if *trace* (pretty-print exp))
      (cond
        ;; step-once either returns the next exp or #f
        ((step-once exp)
         => (lambda (next-exp) (reduce-to-value next-exp)))
        ;; or just ((step-once exp) => reduce-to-value) !!!
        (else exp))))

  (define *trace* #f)                   ; default tracing off

  (define toggle-trace
    (lambda ()
      (set! *trace* (not *trace*))
      *trace*))
```

```
;;;;;;;;;;;;;;;; the stepper ;;;;;;;;;;;;;;;;

;; step-once : closed-expression -> (maybe closed-expression)
;; reduces by one step, else returns #f
(define step-once
  (lambda (exp)
    (cases expression exp

      (const-exp (n) #f)

      (diff-exp (exp1 exp2)
        (cond
          ((step-once exp1)           ; try to step exp1
           => (lambda (new-exp1) (diff-exp new-exp1 exp2)))
          ((step-once exp2)           ; exp1 done, try exp2
           => (lambda (new-exp2) (diff-exp exp1 new-exp2)))
          (else                       ; args done, do the arithmetic
            (reduce-diff-redex exp1 exp2))))

      (var-exp (id)
        (eopl:error 'step-once "non-closed expression ~s" exp))

      (proc-exp (bvar body) #f)    ; lambda-abstractions are values

      (app-exp (rator rand)
        (cond
          ((step-once rator)          ; work on the rator
           => (lambda (new-rator) (app-exp new-rator rand)))
          ((step-once rand)
           => (lambda (new-rand) (app-exp rator new-rand)))
          (else
            (reduce-beta-redex rator rand)))))))

)
```

`reductions.scm` handles the reductions. This is separate because it will be shared by the bigstep evaluator.

```
(module reductions  (lib "eopl.ss" "eopl")
  (require "lang.scm")
  (provide reduce-diff-redex reduce-beta-redex)
  ;;;;;;;;;;;;;;;; reductions ;;;;;;;;;;;;;;;;

  ;; reduce-diff-redex : exp * exp -> (maybe const-exp)
  ;; later on, we'll use the fact that this returns a const-exp and
  ;; not just an exp.
  (define reduce-diff-redex
    (lambda (exp1 exp2)
      (if (and (const-exp? exp1) (const-exp? exp2))
        (const-exp (- (exp->const exp1) (exp->const exp2)))
        ;; can't reduce if operands are not constants
        #f)))

  (define const-exp?
    (lambda (exp)
      (cases expression exp
        (const-exp (n) #t)
        (else #f))))

  (define exp->const
    (lambda (exp)
      (cases expression exp
        (const-exp (n) n)
        (else (eopl:error 'exp->const "bad argument ~s" exp)))))

  ;; reduce-beta : rator rand -> (maybe closed-expression)
  (define reduce-beta-redex
    (lambda (rator rand-value)
      (cases expression rator
        (proc-exp (id body) (subst-closed body id rand-value))
        (else #f))))                    ; can't reduce if rator isn't
                                        ; a proc-exp
  )
```

`top.scm` is the toplevel module. It loads all the required pieces and runs the test suite.

```
(module top (lib "eopl.ss" "eopl")

  ;; top level module.  Loads all required pieces and runs the test
  ;; suite.

  (require "tests.scm")                 ; for test-list

  ;; choose one of the following evaluators:
  ;; all provide a function run : string -> expval

  (require "smallstep.scm")
  ;; (require "bigstep.scm")
  ;; more evaluators will go here

  ;;;;;;;;;;;;;;;; interface to test harness ;;;;;;;;;;;;;;;;

  ;; run-all : () -> unspecified
  as before ....

  ;; run-one : symbol -> expval
  as before ....

  (run-all)

)
```

```
(module tests mzscheme

  (require "lang.scm")                  ; for const-exp
  (require srfi/64)                     ; for testing framework

  (provide sloppy->expval run-tests test-list)

  ;; string * run-fn * (val * val -> bool) listof(tests) -> unspecified
  (define (run-tests group-name run-fn equal-test? test-list)
      (test-begin group-name)
      (for-each
       (lambda (test)
         (let ((test-name (car test))
               (test-pgm (cadr test))
               (expected-result (caddr test)))
           (if (eqv? expected-result 'error)
               (test-error test-name #t (run-fn test-pgm))
               (test-assert test-name
                 (equal-test? (run-fn test-pgm) expected-result)))))
       test-list)
      (test-end group-name))

  (define sloppy->expval const-exp)
```

```
;;;;;;;;;;;;;;;; tests ;;;;;;;;;;;;;;;;

(define test-list
  '(

    ;; simple arithmetic
    (positive-const "11" 11)
    (negative-const "-33" -33)
    (simple-arith-1 "-(44,33)" 11)

    ;; nested arithmetic
    (nested-arith-left "-(-(44,33),22)" -11)
    (nested-arith-right "-(55, -(22,11))" 44)

    ;; simple variables
    (test-var-1 "x" error)
    (test-var-2 "-(x,1)" error)
    (test-var-3 "-(1,x)" error)

    ;; simple unbound variables
    (test-unbound-var-1 "foo" error)
    (test-unbound-var-2 "-(x,foo)" error)

    (simple-app "(proc x x  11)" 11)
    (simple-curried-app "((proc x proc y -(x,y) 5) 6)" -1)
    (pass-a-procedure "(proc f (f 11) proc x x)" 11)
    (check-shadowing "(proc f (f 11) proc f f)" 11)
    ))
)
```

```
Welcome to DrScheme, version 4.2.1 [3m].
Language: Module; memory limit: 256 megabytes.
%%%% Starting test run-all  (Writing full log to "run-all.log")
# of expected passes      14
> (run "-(proc x x, 11)")
#(struct:diff-exp #(struct:proc-exp x #(struct:var-exp x)) #(struct:const-exp 11))
> (run "((proc f proc x (f (f x)) proc n -(n,1)) -(33,11))")
#(struct:const-exp 20)
> (toggle-trace)
#t
> (run "((proc f proc x (f (f x)) proc n -(n,1)) -(33,11))")
#(struct:app-exp
  #(struct:app-exp
    #(struct:proc-exp
      f
      #(struct:proc-exp
        x
        #(struct:app-exp
           #(struct:var-exp f)
           #(struct:app-exp #(struct:var-exp f) #(struct:var-exp x)))))
    #(struct:proc-exp n
       #(struct:diff-exp #(struct:var-exp n) #(struct:const-exp 1))))
  #(struct:diff-exp #(struct:const-exp 33) #(struct:const-exp 11)))

#(struct:app-exp
  #(struct:proc-exp
    x
    #(struct:app-exp
      #(struct:proc-exp n
         #(struct:diff-exp #(struct:var-exp n) #(struct:const-exp 1)))
      #(struct:app-exp
        #(struct:proc-exp n
           #(struct:diff-exp #(struct:var-exp n) #(struct:const-exp 1)))
        #(struct:var-exp x))))
  #(struct:diff-exp #(struct:const-exp 33) #(struct:const-exp 11)))
```

```
#(struct:app-exp
  #(struct:proc-exp
    x
    #(struct:app-exp
      #(struct:proc-exp n
         #(struct:diff-exp #(struct:var-exp n) #(struct:const-exp 1)))
      #(struct:app-exp
        #(struct:proc-exp n
           #(struct:diff-exp #(struct:var-exp n) #(struct:const-exp 1)))
        #(struct:var-exp x))))
  #(struct:const-exp 22))

#(struct:app-exp
  #(struct:proc-exp n
     #(struct:diff-exp #(struct:var-exp n) #(struct:const-exp 1)))
  #(struct:app-exp
    #(struct:proc-exp n
       #(struct:diff-exp #(struct:var-exp n) #(struct:const-exp 1)))
    #(struct:const-exp 22)))

#(struct:app-exp
  #(struct:proc-exp n
     #(struct:diff-exp #(struct:var-exp n) #(struct:const-exp 1)))
  #(struct:diff-exp #(struct:const-exp 22) #(struct:const-exp 1)))

#(struct:app-exp
  #(struct:proc-exp n
     #(struct:diff-exp #(struct:var-exp n) #(struct:const-exp 1)))
  #(struct:const-exp 21))

#(struct:diff-exp #(struct:const-exp 21) #(struct:const-exp 1))

#(struct:const-exp 20)

#(struct:const-exp 20)
>
```

8.6 Big-Step Semantics

Small-step semantics is a convenient and flexible way of specifying the behavior of programs. An alternative method, useful in many situations, is *big-step* or *evaluation* semantics.

Small-step semantics specifies the behavior of a term by simulating the steps of its execution. In big-step semantics, we instead define the relation that holds between a term and its final value.

Formally, instead of defining a relation $\rightarrow$ between closed terms and closed terms, we define a relation $\Downarrow$ between closed terms and values. Just as we did for small-step semantics, we do this by writing axioms and rules of inference that define a relation $\Downarrow$.

Let's see how this works out for our core language. Recall the syntax of the core language:

Core Language: Expression and Values

n	Integers
x	Variables
$e ::= n \mid -(e_1, e_2)$	Expressions
$\quad \mid x \mid (e_1\ e_2) \mid (\lambda x.e)$	
$v ::= n \mid (\lambda x.e)$	Values

We will have one evaluation rule for each kind of closed expression in our language:

Core Language: Evaluation Rules

EVAL CONST
$$n \Downarrow n$$

EVAL DIFF
$$\frac{e_1 \Downarrow n \qquad e_2 \Downarrow m \qquad p = n - m}{-(e_1, e_2) \Downarrow p}$$

EVAL ABS
$$(\lambda x.e) \Downarrow (\lambda x.e)$$

EVAL APP
$$\frac{e_1 \Downarrow (\lambda x.e') \qquad e_2 \Downarrow v \qquad e'[v/x] \Downarrow w}{(e_1\ e_2) \Downarrow w}$$

Let's try writing out an example in this system:

Small step:

$$\begin{aligned}&((\lambda f.(f\ 3))\ (\lambda x.-(x,1)))\\ \rightarrow&((\lambda x.-(x,1))\ 3)\\ \rightarrow&-(3,1)\\ \rightarrow&2\end{aligned}$$

Big step:

$$\dfrac{\dfrac{(\lambda f.(f\ 3)) \Downarrow (\lambda f.(f\ 3)) \quad (\lambda x.-(x,1)) \Downarrow (\lambda x.-(x,1)) \quad \dfrac{(\lambda x.-(x,1)) \Downarrow (\lambda x.-(x,1)) \quad 3 \Downarrow 3 \quad -(3,1) \Downarrow 2}{((\lambda x.-(x,1))\ 3) \Downarrow 2}}{((\lambda f.(f\ 3))\ (\lambda x.-(x,1))) \Downarrow 2}$$

This gets a little clearer if we label each assertion with its address in the tree:

$$\dfrac{\dfrac{0.1 : (\lambda f.(f\ 3)) \Downarrow (\lambda f.(f\ 3)) \quad 0.2 : (\lambda x.-(x,1)) \Downarrow (\lambda x.-(x,1)) \quad 0.3.1 : (\lambda x.-(x,1)) \Downarrow (\lambda x.-(x,1)) \quad 0.3.2 : 3 \Downarrow 3 \quad 0.3.3 : -(3,1) \Downarrow 2}{0.3 : ((\lambda x.-(x,1))\ 3) \Downarrow 2}}{0 : ((\lambda f.(f\ 3))\ (\lambda x.-(x,1))) \Downarrow 2}$$

This is incredibly painful to typeset! It gets a little easier (and narrower!) if we switch to a vertical arrangement:

$$\dfrac{\begin{array}{|l}\text{hypothesis 1}\\ \text{hypothesis 2}\\ \text{hypothesis 3}\end{array}}{\text{Conclusion}}$$

For example, EVAL APP becomes:

$$\dfrac{\begin{array}{|l}e_1 \Downarrow (\lambda x.e')\\ e_2 \Downarrow v\\ e'[v/x] \Downarrow w\end{array}}{(e_1\ e_2) \Downarrow w}$$

Here's our first example again, in horizontal and then in vertical arrangement:

$$\dfrac{\begin{array}{l}(\lambda f.(f\ 3)) \Downarrow (\lambda f.(f\ 3))\\ (\lambda x.-(x,1)) \Downarrow (\lambda x.-(x,1))\end{array} \qquad \dfrac{\begin{array}{l}(\lambda x.-(x,1)) \Downarrow (\lambda x.-(x,1))\\ 3 \Downarrow 3\\ -(3,1) \Downarrow 2\end{array}}{((\lambda x.-(x,1))\ 3) \Downarrow 2}}{((\lambda f.(f\ 3))\ (\lambda x.-(x,1))) \Downarrow 2}$$

$$\dfrac{\begin{array}{|l}(\lambda f.(f\ 3)) \Downarrow (\lambda f.(f\ 3))\\ \\ (\lambda x.-(x,1)) \Downarrow (\lambda x.-(x,1))\\ \\ \quad\begin{array}{|l}(\lambda x.-(x,1)) \Downarrow (\lambda x.-(x,1))\\ 3 \Downarrow 3\\ -(3,1) \Downarrow 2\end{array}\\ \hline ((\lambda x.-(x,1))\ 3) \Downarrow 2\end{array}}{((\lambda f.(f\ 3))\ (\lambda x.-(x,1))) \Downarrow 2}$$

Exercise: work out the other example $((((\lambda f.(\lambda x.(f\ (f\ x))))\ (\lambda n.-(n,1)))\ -(33,11)))$ in the bigstep semantics.

8.7 Big-Step vs. Small-Step Semantics

We can prove an equivalence between big- and small-step semantics. We won't work out the details, but we can give the outline.

Theorem: If e is a closed term and v is a value, then

$$e \Downarrow v \iff e \to^* v$$

Proof:

($\Rightarrow$): By induction on the size of the derivation of $e \Downarrow v$: For each axiom $e \Downarrow v$, it is also true that $e \to^* v$. Furthermore, for each rule

$$\frac{\begin{array}{c} e_1 \Downarrow v_1 \\ \cdots \\ e_n \Downarrow v_n \end{array}}{e \Downarrow v}$$

it is easy to check that if

$$\begin{array}{c} e_1 \to^* v_1 \\ \cdots \\ e_n \to^* v_n \end{array}$$

are all true, then so is $e \to^* v$.

($\Leftarrow$): This direction is a little harder. We need to show:

1. If $e \to e'$, then for any E, $E[e] \to E[e']$, and
2. For any reduction context E, if

   $$E[(e_1\ e_2)] \to^* E[v]$$

   then there exist x, e', v' such that

   $$\begin{array}{l} e_1 \to^* (\lambda x.e') \\ e_2 \to^* v', \quad \text{and} \\ e'[v'/x] \to^* v \end{array}$$

   With this information, we can reconstruct a derivation of the required tree.

8.8 Implementation

The main procedure will be `value-of-expression`, which takes an expression e and tries to find a value v such that $e \Downarrow v$. This is in `core-lang/bigstep.scm`:

```
(module bigstep (lib "eopl.ss" "eopl")

  (require "lang.scm")                  ; for scan&parse
  (require "reductions.scm")            ; for reduce-*-redex

  (provide run)

;;;;;;;;;;;;;;;; top level ;;;;;;;;;;;;;;;;

  (define run
    (lambda (string)
      (value-of-program (scan&parse string))))

  (define value-of-program
    (lambda (pgm)
      (cases program pgm
        (a-program (exp) (value-of-expression exp)))))
```

```
;; value-of-expression :: closed exp -> closed value
(define value-of-expression
  (lambda (exp)
    (cases expression exp
```

EVAL CONST

$$n \Downarrow n$$

```
      (const-exp (n) exp)
```

EVAL DIFF

$$\frac{e_1 \Downarrow n \qquad e_2 \Downarrow m \qquad p = n - m}{-(e_1, e_2) \Downarrow p}$$

```
      (diff-exp (exp1 exp2)
        (let ((val1 (value-of-expression exp1))
              (val2 (value-of-expression exp2)))
          (let ((reductum (reduce-diff-redex val1 val2)))
            (if reductum
              ;; reduce-diff-redex always returns a value.
              reductum
              ;; if reduce-diff-redex fails, it's an error
              (eopl:error 'bad-argument-to-diff "~s ~s" val1 val2)))))

      (var-exp (id)
        (eopl:error 'step-once "non-closed expression ~s" exp))
```

EVAL ABS

$$(\lambda x.e) \Downarrow (\lambda x.e)$$

```
      (proc-exp (bvar body) exp)
```

EVAL APP

$$\frac{e_1 \Downarrow (\lambda x.e') \qquad e_2 \Downarrow v \qquad e'[v/x] \Downarrow w}{(e_1\ e_2) \Downarrow w}$$

```
(app-exp (rator rand)
  (let ((val1 (value-of-expression rator))
        (val2 (value-of-expression rand)))
    (let ((reductum (reduce-beta-redex val1 val2)))
      (if reductum
        (value-of-expression reductum)
        (eopl:error 'bad-proc-to-app "~s" val1))))))))

)
```

We re-use the reductions from `reductions.scm` whenever possible.

We take advantage of tail recursion to eliminate the final hypothesis in EVAL APP.

Notice the resemblance to an ordinary EOPL-style interpreter.

***** BELOW HERE NOT REVISED *****

8.9 A Larger Language: `base-lang`

Now we are ready to consider a somewhat larger language. Our new language adds the following features:

1. more arithmetic operators
2. booleans and conditionals
3. multi-argument abstraction and application
4. `let` and `letrec`

```
(define the-grammar
  '((program (expression) a-program)

    ;; numeric computations, but with many primitives

    (expression (number) const-exp)
    (expression
      (primitive "(" (separated-list expression ",") ")")
      primapp-exp)

    (primitive ("+")     add-prim)
    (primitive ("-")     subtract-prim)
    (primitive ("*")     mult-prim)
    (primitive ("add1")  incr-prim)
    (primitive ("sub1")  decr-prim)
    (primitive ("zero?") zero-test-prim)

    ;; booleans: producers and consumers

    (expression ("true") true-exp)
    (expression ("false") false-exp)

    (expression
      ("if" expression "then" expression "else" expression)
      if-exp)
```

```
;; multi-argument procedures

(expression (identifier) var-exp)
(expression
  ("proc" "(" (separated-list identifier ",") ")" expression)
  proc-exp)
(expression
  ("(" expression (arbno expression) ")")
  app-exp)

;; let and letrec

(expression
  ("let" (arbno  identifier "=" expression) "in" expression)
  let-exp)

(expression
  ("letrec" (arbno identifier "=" expression) "in" expression)
  letrec-exp)

))
```

Base Language: Expression, Values, Reduction Contexts

n	Integers
x	Variables
p	Primitives
$e ::= n \mid p(e_1, \ldots, e_n)$	Expressions
$\mid$ `true` $\mid$ `false` $\mid$ `if` e_0 `then` e_1 `else` e_2	
$\mid x \mid (e_0\ e_1\ \ldots\ e_n) \mid$ `proc` $(x_1, \ldots, x_n)\, e$	
$\mid$ `let` $\{x = e\}^*$ `in` e	
$\mid$ `letrec` $\{x = e\}^*$ `in` e	
$v ::= n \mid$ `true` $\mid$ `false` $\mid$ `proc` $(x_1, \ldots, x_n)\, e$	Values
$E ::= [\,]$	Reduction Contexts
$\mid p(v, \ldots, v, E, e, \ldots, e)$	CXT PRIM
$\mid$ `if` E `then` e_1 `else` e_2	CXT IF
$\mid (E\ e_1\ \ldots\ e_n)$	CXT RATOR
$\mid (v\ v\ \ldots\ v\ E\ e\ \ldots\ e)$	CXT RANDS
$\mid$ `let` $x = v\ \ldots\ x = v\ x = E\ x = e\ \ldots\ x = e$ `in` e	CXT LET

As before, we need to specify the reduction contexts. The reduction contexts are:

1. The first non-value operand of an arithmetic operator
2. The test in a conditional
3. The operator in an application
4. The first non-value operand in an application
5. The first non-value right-hand side in a `let`

Base Language: Reduction Rules

RED PRIM
$$p(n_1, \ldots, n_k) \to m \qquad \text{if } p^{\text{Arith}}(n_1, \ldots, n_k) = m$$

RED BETA
$$(\texttt{proc}\ (x_1\ \ldots\ x_n)\ e\quad v_1\ \ldots\ v_n) \to e[v_1\ \ldots\ v_n / x_1\ \ldots\ x_n]$$

RED IF TRUE
$$\texttt{if true then}\ e_1\ \texttt{else}\ e_2 \to e_1$$

RED IF FALSE
$$\texttt{if false then}\ e_1\ \texttt{else}\ e_2 \to e_2$$

RED LET
$$\texttt{let}\ x_1 = v_1\ \ldots\ x_n = v_n\ \texttt{in}\ e \to e[v_1\ \ldots\ v_n / x_1\ \ldots\ x_n]$$

RED LETREC
$$\texttt{letrec}\ x_1 = e_1\ \ldots\ x_n = e_n\ \texttt{in}\ e_0 \to e_0[(\texttt{letrec}\ x_1 = e_1\ \ldots\ x_n = e_n\ \texttt{in}\ e_i)/x_i]_{i=1,\ldots,n}$$

Here $e[e_1\ \ldots\ e_n/x_1\ \ldots\ x_n]$ means the simultaneous substitution in e of e_1 through e_n for x_1 through x_n. We'll also write

$$e[e_1/x_1, \ldots, e_n/x_n]$$

or occasionally (as in Red Letrec),

$$e[e_i/x_i]_{i=1,\ldots,n}$$

for the same thing.

Let's look at these rules:

1. RED PRIM is the same as in the core language

2. RED BETA is a straightforward generalization of the corresponding rule in the core

3. RED IF TRUE and RED IF FALSE are clear. In combination with CXT IF, they say that to reduce an `if`, you reduce the test expression until it is either `true` or `false`, and reduce to either the `then` expression or the `else` expression.

4. RED LET: CXT LET says that the right-hand sides of the declarations in a `let` are evaluated from left to right until they are all values. Then RED LET says the values are substituted into the body.

We defer the discussion of RED LETREC until later.

8.10 Implementation

```
(module reductions  (lib "eopl.ss" "eopl")

  (require "drscheme-init.scm")
  (require "lang.scm")

  (provide reduce-delta-redex reduce-beta-redex reduce-if-redex
    reduce-let-redex reduce-letrec-redex)

  ;; reduce-delta-redex : prim * (list-of exp) -> closed value
  (define reduce-delta-redex
    (lambda (prim exps)
      (let ((args (map exp->const exps)))
        (cases primitive prim
          (add-prim ()
            (const-exp (+ (car args) (cadr args))))
          (subtract-prim ()
            (const-exp (- (car args) (cadr args))))
          (mult-prim  ()
            (const-exp (* (car args) (cadr args))))
          (incr-prim  ()
            (const-exp (+ (car args) 1)))
          (decr-prim  ()
            (const-exp (- (car args) 1)))
          (zero-test-prim ()
            (if (zero? (car args))
              (true-exp) (false-exp)))
          ))))

  (define exp->const ...)           ; as before
```

```
;; reduce-if-redex : value * exp * exp -> exp
(define reduce-if-redex
  (lambda (e0 e1 e2)
    (cases expression e0
      (true-exp () e1)
      (false-exp () e2)
      (else
        (eopl:error 'reduce-if-redex
          "non-boolean test: ~s" e0)))))

(define reduce-let-redex
  (lambda (vars rhs-values body)
    (subst-closed body vars rhs-values)))

(define reduce-letrec-redex
  (lambda (vars rhss body)
    (subst-closed
      body
      vars
      (map
        (lambda (rhs) (letrec-exp vars rhss rhs))
        rhss))))
)
```

```
(module smallstep (lib "eopl.ss" "eopl")

  (require "drscheme-init.scm")         ; for pretty-print
  (require "lang.scm")                  ; for scan&parse
  (require "reductions.scm")

  (provide run toggle-trace)
  ;; (provide (all-defined))               ; handy for testing

  ;;;;;;;;;;;;;;;; top level ;;;;;;;;;;;;;;;;

  ;; run : string -> expval

  (define run
    (lambda (string)
      (reduce-program-to-value
        (scan&parse string))))

  ;; reduce-program-to-value : program -> value
  (define reduce-program-to-value
    (lambda (pgm)
      (cases program pgm
        (a-program (exp) (reduce-to-value exp)))))

  ;; reduce-to-value : closed-expression -> closed-value
  (define reduce-to-value
    (lambda (exp)
      ...                               ; as before
      ))

  (define *trace* #f)                   ; default tracing off
  (define toggle-trace ...)
```

```
;;;;;;;;;;;;;;;; the stepper ;;;;;;;;;;;;;;;;

;; step-once : closed expression -> (maybe closed-expression)
;; reduces by one step, else returns #f

(define step-once
  (lambda (exp)
    (cases expression exp

      (const-exp (n) #f)
      (primapp-exp (prim rands)
        (cond
          ((step-once-list rands)
           => (lambda (new-rands) (primapp-exp prim new-rands)))
          (else (reduce-delta-redex prim rands))))

      (true-exp () #f)
      (false-exp () #f)
      (if-exp (e0 e1 e2)
        (cond
          ((step-once e0) => (lambda (new-e0) (if-exp new-e0 e1 e2)))
          (else (reduce-if-redex e0 e1 e2))))

      (var-exp (id)
        (eopl:error 'step-once "non-closed expression ~s" exp))
      (proc-exp (ids body) #f)
      (app-exp (rator rands)
        (cond
          ((step-once rator)
           => (lambda (new-rator) (app-exp new-rator rands)))
          ((step-once-list rands)
           => (lambda (new-rands) (app-exp rator new-rands)))
          (else
            (reduce-beta-redex rator rands))))
```

```
      (let-exp (ids rhss body)
        (cond
          ((step-once-list rhss)
           => (lambda (new-rhss) (let-exp ids new-rhss body)))
          (else
            (reduce-let-redex ids rhss body))))

      (letrec-exp (ids rhss body)
        (reduce-letrec-redex ids rhss body))

      )))

;; step-once-list: (list-of closed-exp)
;;                  -> (maybe (list-of closed-exp))
;; returns a list like the original, with leftmost redex reduced, or
;; #f if no redex is found.
(define step-once-list
  (lambda (exps)
    (cond
      ((null? exps) #f)           ; the empty list can't take a step
      ((step-once (car exps))
       => (lambda (new-exp) (cons new-exp (cdr exps))))
      ((step-once-list (cdr exps))
       => (lambda (new-exps) (cons (car exps) new-exps)))
      (else #f))))


)
```

Let's watch this work on a sizeable example.

```
>  (run "let x = 5
        in let x = 38
               f = proc (y,z) *(y, +(x,z))
               g = proc (u) +(u,x)
           in (f (g 3) 17)")
#(struct:let-exp (x)
  (#(struct:const-exp 5))
  #(struct:let-exp (x f g)
    (#(struct:const-exp 38)
     #(struct:proc-exp (y z)
       #(struct:primapp-exp #(struct:mult-prim)
         (#(struct:var-exp y)
          #(struct:primapp-exp #(struct:add-prim)
             (#(struct:var-exp x)
              #(struct:var-exp z))))))
     #(struct:proc-exp (u)
       #(struct:primapp-exp #(struct:add-prim)
          (#(struct:var-exp u)
           #(struct:var-exp x)))))
    #(struct:app-exp
      #(struct:var-exp f)
      (#(struct:app-exp
          #(struct:var-exp g)
          (#(struct:const-exp 3)))
       #(struct:const-exp 17)))))
```

```
#(struct:let-exp (x f g)
  (#(struct:const-exp 38)
   #(struct:proc-exp (y z)
     #(struct:primapp-exp #(struct:mult-prim)
       (#(struct:var-exp y)
        #(struct:primapp-exp #(struct:add-prim)
           (#(struct:const-exp 5)
            #(struct:var-exp z))))))
   #(struct:proc-exp (u)
     #(struct:primapp-exp #(struct:add-prim)
        (#(struct:var-exp u)
         #(struct:const-exp 5)))))
  #(struct:app-exp
    #(struct:var-exp f)
    (#(struct:app-exp
        #(struct:var-exp g)
        (#(struct:const-exp 3)))
     #(struct:const-exp 17))))

#(struct:app-exp
  #(struct:proc-exp (y z)
    #(struct:primapp-exp #(struct:mult-prim)
      (#(struct:var-exp y)
       #(struct:primapp-exp #(struct:add-prim)
          (#(struct:const-exp 5)
           #(struct:var-exp z))))))
  (#(struct:app-exp
     #(struct:proc-exp (u)
       #(struct:primapp-exp #(struct:add-prim)
          (#(struct:var-exp u)
           #(struct:const-exp 5))))
     (#(struct:const-exp 3)))
   #(struct:const-exp 17)))
```

```
#(struct:app-exp
  #(struct:proc-exp (y z)
    #(struct:primapp-exp #(struct:mult-prim)
      (#(struct:var-exp y)
       #(struct:primapp-exp #(struct:add-prim)
          (#(struct:const-exp 5)
           #(struct:var-exp z))))))
  (#(struct:primapp-exp #(struct:add-prim)
      (#(struct:const-exp 3)
       #(struct:const-exp 5)))
   #(struct:const-exp 17)))

#(struct:app-exp
  #(struct:proc-exp (y z)
    #(struct:primapp-exp
      #(struct:mult-prim)
      (#(struct:var-exp y)
       #(struct:primapp-exp
          #(struct:add-prim)
          (#(struct:const-exp 5)
           #(struct:var-exp z))))))
  (#(struct:const-exp 8)
   #(struct:const-exp 17)))

#(struct:primapp-exp #(struct:mult-prim)
  (#(struct:const-exp 8)
   #(struct:primapp-exp #(struct:add-prim)
      (#(struct:const-exp 5)
       #(struct:const-exp 17)))))

#(struct:primapp-exp #(struct:mult-prim)
   (#(struct:const-exp 8)
    #(struct:const-exp 22)))
#(struct:const-exp 176)                ; last trace output
#(struct:const-exp 176)                ; value of (run ...)
>
```

8.11 Big-Step Semantics

Base Language: Evaluation Rules

$$\textsc{Eval Const}\quad n \Downarrow n$$

$$\textsc{Eval Arith}\quad \frac{e_1 \Downarrow n_1 \quad \ldots \quad e_k \Downarrow n_k \quad p^{\text{Arith}}(n_1, \ldots, n_k) = m}{p(e_1, \ \ldots \ , e_k) \Downarrow m}$$

$$\textsc{Eval True}\quad \texttt{true} \Downarrow \texttt{true} \qquad \textsc{Eval False}\quad \texttt{false} \Downarrow \texttt{false}$$

$$\textsc{Eval If True}\quad \frac{e_0 \Downarrow \texttt{true} \quad e_1 \Downarrow v}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v} \qquad \textsc{Eval If False}\quad \frac{e_0 \Downarrow \texttt{false} \quad e_2 \Downarrow v}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v}$$

$$\textsc{Eval Proc}\quad \texttt{proc}\ (x_1, \ \ldots \ , x_n)\ e \Downarrow \texttt{proc}\ (x_1, \ \ldots \ , x_n)\ e$$

$$\textsc{Eval App}\quad \frac{e_0 \Downarrow \texttt{proc}\ (x_1, \ \ldots \ , x_n)\ e \quad e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n \quad e[v_i/x_i]_{i=1,\ldots,n} \Downarrow w}{(e_0\ e_1\ \ldots\ e_n) \Downarrow w}$$

$$\textsc{Eval Let}\quad \frac{e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n \quad e[v_i/x_i]_{i=1,\ldots,n} \Downarrow w}{\texttt{let}\ x_1 = e_1\ \ldots\ x_n = e_n\ \texttt{in}\ e \Downarrow w}$$

$$\textsc{Eval Letrec}\quad \frac{e_0[(\texttt{letrec}\ x_1 = e_1\ \ldots\ x_n = e_n\ \texttt{in}\ e_i)/x_i]_{i=1,\ldots,n} \Downarrow w}{\texttt{letrec}\ x_1 = e_1\ \ldots\ x_n = e_n\ \texttt{in}\ e_0 \Downarrow w}$$

```
;;; base-lang/bigstep.scm

  ;;;;;;;;;;;;;;;; top level ;;;;;;;;;;;;;;;;

  (define run
    (lambda (string)
      (value-of-program (scan&parse string))))

  ;;;;;;;;;;;;;;;; evaluator ;;;;;;;;;;;;;;;;

  (define value-of-program
    (lambda (pgm)
      (cases program pgm
        (a-program (exp) (value-of-expression exp)))))

  ;; value-of-expression :: closed exp -> closed value
  (define value-of-expression
    (lambda (exp)
      (cases expression exp
```

EVAL CONST

$n \Downarrow n$

```
        (const-exp (n) exp)
```

EVAL ARITH

$$\frac{e_1 \Downarrow n_1 \quad \ldots \quad e_k \Downarrow n_k \qquad p^{\texttt{Arith}}(n_1, \ldots, n_k) = m}{p(e_1, \ldots, e_k) \Downarrow m}$$

```
        (primapp-exp (prim rands)
          (let ((vals (value-of-expressions rands)))
            (value-of-expression (reduce-delta-redex prim vals))))
```

EVAL TRUE

$\texttt{true} \Downarrow \texttt{true}$

```
        (true-exp () exp)
```

EVAL FALSE

$\texttt{false} \Downarrow \texttt{false}$

```
(false-exp () exp)
```

EVAL IF TRUE

$$\frac{e_0 \Downarrow \texttt{true} \qquad e_1 \Downarrow v}{\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v}$$

EVAL IF FALSE

$$\frac{e_0 \Downarrow \texttt{false} \qquad e_2 \Downarrow v}{\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v}$$

```
(if-exp (e0 e1 e2)
  (let ((val0 (value-of-expression e0)))
    (value-of-expression (reduce-if-redex val0 e1 e2))))

(var-exp (id)
  (eopl:error 'step-once "non-closed expression ~s" exp))
```

EVAL PROC

$\texttt{proc}\,(x_1, \ldots, x_n)\, e \Downarrow \texttt{proc}\,(x_1, \ldots, x_n)\, e$

```
(proc-exp (bvar body) exp)   ; lambda-abstractions are values
```

EVAL APP

$$\frac{e_0 \Downarrow \texttt{proc}\,(x_1, \ldots, x_n)\, e \qquad e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n \qquad e[v_i/x_i]_{i=1,\ldots,n} \Downarrow w}{(e_0\, e_1\, \ldots\, e_n) \Downarrow w}$$

```
(app-exp (rator rands)
  (let ((proc (value-of-expression  rator))
        (args (value-of-expressions rands)))
    (value-of-expression (reduce-beta-redex proc args))))
```

EVAL LET

$$\frac{e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n \qquad e[v_i/x_i]_{i=1,\ldots,n} \Downarrow w}{\texttt{let } x_1 = e_1 \ldots x_n = e_n \texttt{ in } e \Downarrow w}$$

```
(let-exp (ids rhss body)
  (let ((vals (value-of-expressions rhss)))
    (value-of-expression (reduce-let-redex ids vals body))))
```

EVAL LETREC

$$\frac{e_0[(\texttt{letrec}\ x_1 = e_1\ \ldots\ x_n = e_n\ \texttt{in}\ e_i)/x_i]_{i=1,\ldots,n} \Downarrow w}{\texttt{letrec}\ x_1 = e_1\ \ldots\ x_n = e_n\ \texttt{in}\ e_0 \Downarrow w}$$

```
      (letrec-exp (ids rhss body)
        (value-of-expression (reduce-letrec-redex ids rhss body)))
      )))

(define value-of-expressions
  (lambda (exps)
    (map value-of-expression exps)))
```

Are we having fun yet?

Now we have a basic framework, and we can go on to consider the other basic structures of programming languages: environments, continuations, and stores.