CS 7400 September 23–30, 2009
Dr. Wand Readings: EOPL3, Secs. 3.1–3.5

Lecture 3: Expressions

Key Concepts:

syntax and semantics
expressed and denoted values
expressions
environment
specifying the behavior of expressions
`let` expressions
implementation using PLT Scheme and its module system
procedures
formal parameter, bound variable
actual parameters, operand, argument
closures
`letrec`
poor man's letrec
multiple arguments, multiple declarations

3.1 LET: a simple language

This is in `http://www.ccs.neu.edu/course/cs7400/interps/lecture03/let-lang`.
(All these interpreters are in
`http://www.ccs.neu.edu/course/cs7400/interps/lecture03/`)

3.1.1 Specification of Syntax

Syntax for the LET language

Program ::= *Expression*
`a-program (exp)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `-(`*Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

Expression ::= `zero? (`*Expression*`)`
`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var rhs-exp body)`

```
> (scan&parse "-(55, -(22,11))")
#(struct:a-program
  #(struct:diff-exp
    #(struct:const-exp 55)
    #(struct:diff-exp
       #(struct:const-exp 22) #(struct:const-exp 11))))
```

3.1.2 Specification of Values

An important part of the specification of any programming language is the set of values that the language manipulates. Each language has at least two such sets: the *expressed values* and the *denoted values*. The expressed values are the possible values of expressions, and the denoted values are the values bound to variables.

In Scheme, for example, there are many kinds of expressed values, such as numbers, pairs, characters, and strings. However every Scheme variable denotes a location that is modifiable using `set!`, so there is only one kind of denoted value: a location containing expressed values.

In our language, the expressed and denoted values will be the same:

$$\begin{aligned} \mathit{ExpVal} &= \mathit{Number} + \mathit{Bool} \\ \mathit{DenVal} &= \mathit{Number} + \mathit{Bool} \end{aligned}$$

We model this using `define-datatype`

```
(define-datatype expval expval?
  (num-val
    (value number?))
  (bool-val
    (boolean boolean?)))
```

We typically use extractors with `expval`:

```
(define expval->num
  (lambda (v)
    (cases expval v
      (num-val (num) num)
      (else (expval-extractor-error 'num v)))))

(define expval->bool
  (lambda (v)
    (cases expval v
      (bool-val (bool) bool)
      (else (expval-extractor-error 'bool v)))))
```

```
(define expval-extractor-error
  (lambda (variant value)
    (eopl:error 'expval-extractors "Looking for a ~s, found ~s"
      variant value)))
```

3.1.3 Environments

If we're going to evaluate expressions containing variables, we'll need to know the value associated with each variable. We'll do this by keeping those values in an *environment*.

An environment is a function whose domain is a finite set of variables and whose range is the denoted values.

- ρ ranges over environments.
- [] denotes the empty environment.
- $[x = v]\rho$ denotes `(extend-env x v ρ)`.
- $[x_1 = v_1, x_2 = v_2]\rho$ abbreviates $[x_1 = v_1]([x_2 = v_2]\rho)$, etc.
- $[x_1 = v_1, x_2 = v_2, \ldots]$ denotes the environment in which the value of x_1 is v_1, etc.

We'll use indentation to to improve readability, eg:

```
[x=3]
 [y=7]
  [u=5]ρ
```

to abbreviate

```
(extend-env 'x 3
  (extend-env 'y 7
    (extend-env 'u 5 ρ)))
```

3.1.4 Specifying the Behavior of Expressions

6 kinds of expressions in our language: one for each production with *Expression* as its left-hand side.

Interface for expressions will contain seven procedures: six constructors and one observer.

We use `expval` to denote the set of expressed values.

Interface for Expressions

constructors:

const-exp	: `int -> expression`
zero?-exp	: `expression -> expression`
if-exp	: `expression * expression * expression -> expression`
diff-exp	: `expression * expression -> expression`
var-exp	: `symbol -> expression`
let-exp	: `symbol * expression * expression -> expression`

observer:

value-of	: `expression * environment -> expval`

Now need to specify *behavior* of the functions in this interface.

Specification will consist of assertions of the form

$$(\texttt{value-of}\ \ e\ \ \rho) = v$$

meaning that the value of expression e in environment ρ should be v.

We write down rules of inference and equations, like those in chapter 1, that will enable us to derive such assertions.

```
(value-of (const-exp n) ρ) = n

(value-of (var-exp x) ρ) = (apply-env ρ x)

(value-of (diff-exp e1 e2) ρ) = (-
                                  (value-of e1 ρ)
                                  (value-of e2 ρ))
```

3.1.5 An Example

Let $\rho =$ `[x=10,v=5,i=1]`.

We write `<<`e`>>` to denote the AST for expression e.

Then

```
(value-of-expression <<-(-(x,3),-(v,i))>> ρ)

= (-
    (value-of-expression <<-(x,3)>> ρ)
    (value-of-expression <<-(v,i)>> ρ))

= (-
    (-
      (value-of-expression <<x>> ρ)
      (value-of-expression <<3>> ρ))
    (value-of-expression <<-(v,i)>> ρ))

= (-
    (-
      10
      (value-of-expression <<3>> ρ))
    (value-of-expression <<-(v,i)>> ρ))

= (-
    (-
      10
      3)
    (value-of-expression <<-(v,i)>> ρ))

= (-
    7
    (value-of-expression <<-(v,i)>> ρ))
```

```
= (-
    7
    (-
      (value-of-expression <<v>> ρ)
      (value-of-expression <<i>> ρ)))

= (-
    7
    (-
      5
      (value-of-expression <<i>> ρ)))

= (-
    7
    (-
      5
      1))

= (-
    7
    4)

= 3
```

3.1.6 Specifying the Behavior of Programs

In our language, a whole program is just an expression. In order to find the value of such an expression, we need to specify the values of the free variables in the program. So the value of a program is just the value of that expression in a suitable initial environment. We choose our initial environment to be `[i=1,v=5,x=10]`.

`(value-of-program` *e*`) = (value-of` *e* `[i=1,v=5,x=10])`

3.1.7 Specifying Conditionals

The next portion of the language introduces an interface for booleans in our language. The language has one constructor of booleans, `zero?`, and one observer of booleans, the `if` expression.

The value of a `zero?` expression is a true value if and only if the value of its operand is zero.

$$\frac{(\texttt{value-of}\ e_1\ \rho) = v}{\begin{array}{l}(\texttt{value-of}\ (\texttt{zero?-exp}\ e_1)\ \rho) \\ \quad = \begin{cases}(\texttt{bool-val}\ \texttt{\#t}) & \text{if } (\texttt{expval->num}\ v) = 0 \\ (\texttt{bool-val}\ \texttt{\#f}) & \text{if } (\texttt{expval->num}\ v) \neq 0\end{cases}\end{array}}$$

We use `bool-val` as a constructor to turn a boolean into an expressed value, and `expval->num` as an extractor to check to see whether an expressed value is an integer, and if so, to return the integer.

An `if` expression is an observer of boolean values.

$$\frac{(\texttt{value-of}\ e_1\ \rho) = v}{\begin{array}{l}(\texttt{value-of}\ (\texttt{if-exp}\ e_1\ e_2\ e_3)\ \rho) \\ \quad = \begin{cases}(\texttt{value-of}\ e_2\ \rho) & \text{if } (\texttt{expval->bool}\ v) = \texttt{\#t} \\ (\texttt{value-of}\ e_3\ \rho) & \text{if } (\texttt{expval->bool}\ v) = \texttt{\#f}\end{cases}\end{array}}$$

Rules of inference like this make the intended behavior of any individual expression easy to specify, but they are not very good for displaying a deduction.

So we recast our rules as equations.

Can then use substitution of equals for equals to display a calculation.

```
(value-of (if-exp e1 e2 e3) ρ)
= (if (expval->bool (value-of e1 ρ))
      (value-of e2 ρ)
      (value-of e3 ρ))
```

An Example Let $\rho =$ `[x=66,y=22]`.

```
(value-of
  <<if zero?(-(x,11)) then -(y,2) else -(y,4)>>
  ρ)

= (if (expval->bool (value-of <<zero?(-(x,11))>> ρ))
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (if (expval->bool (bool-val #f)
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (if (expval->bool (= 0 (value-of <<-(x,11)>> ρ)))
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (if #f
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (value-of <<-(y,4)>> ρ)

= 18
```

3.1.8 Adding `let`

Examples

```
let x = 5
in -(x,3)

let x = 5
in let y = 3
   in -(x,y)

let z = 5
in let x = 3
   in let y = -(x,1)     % here x = 3
      in let x = 4
         in -(z, -(x,y)) % here x = 4
```

The `let` variable is bound in the body.

Specification As a rule of inference:

$$\frac{(\texttt{value-of}\ \ e_1\ \ \rho) = v_1}{(\texttt{value-of}\ \ (\texttt{let-exp}\ \ var\ \ e_1\ \ e_2)\ \ \rho) = (\texttt{value-of}\ \ e_2\ \ [var = v_1]\rho)}$$

Or as an equation:

$$\begin{array}{l}(\texttt{value-of}\ \ (\texttt{let-exp}\ \ x\ \ e_1\ \ e_2)\ \ \rho)\\ =\ (\texttt{value-of}\ \ e_2\ \ [x=(\texttt{value-of}\ \ e_1\ \ \rho)]\rho)\end{array}$$

An example:

Let $\rho =$ `[z = 7]`. Then

```
(value-of-expression <<let x = 5
                       in let y = -(z,3)
                          in -(-(x,y),z)>> ρ)

= (value-of-expression <<let y = -(z,3)
                         in -(-(x,y),z)>> [x = 5]ρ)
```

(Let ρ_1 = [x=5]ρ)

```
= (value-of-expression
    <<-(-(x,y),z)>>
    [y = (value-of-expression <<-(z,3)>> ρ1)]ρ1)


= (value-of-expression
    <<-(-(x,y),z)>>
    [y = (- (value-of-expression <<z>> ρ1)
            3)]ρ1)

= (value-of-expression
    <<-(-(x,y),z)>>
    [y = (- 7 3)]ρ1)

= (value-of-expression
    <<-(-(x,y),z)>>
    [y = (- 7 3)]ρ1)

= (value-of-expression
    <<-(-(x,y),z)>>
    [y = 4]ρ1)
```

(Let ρ_2 = [y = 4]ρ_1 = [y=4,x=5,z=7])

```
= (value-of-expression <<-(-(x,y),z)>> ρ2)
```

```
= (- (- (value-of-expression <<x>> ρ2)
        (value-of-expression <<y>> ρ2))
     (value-of-expression <<z>> ρ2))


= (- (- 5
        4)
     7)

= (- 1 7)

= -6
```

OK, this is tedious. Computers are clearly better at this than we are. So let's implement the specification.

3.1.9 Implementing the Language

We implement the language as a set of PLT Scheme modules.

```
(module lang                    ; this is lang.scm
  (lib "eopl.ss" "eopl")        ; it is written in the eopl language level

  (provide (all-defined))       ; it exports too much stuff to list.

  ;;;;;;;;;;;;;;;; grammatical specification ;;;;;;;;;;;;;;;;

  (define the-lexical-spec
    '((whitespace (whitespace) skip)
      (comment ("%" (arbno (not #\newline))) skip)
      (identifier
       (letter (arbno (or letter digit "_" "-" "?")))
       symbol)
      (number (digit (arbno digit)) number)
      (number ("-" digit (arbno digit)) number)
      ))

  (define the-grammar
    '((program (expression) a-program)
      (expression (identifier) var-exp)
      (expression (number) const-exp)
      (expression ("-" "(" expression "," expression ")") diff-exp)
      (expression ("zero?" "(" expression ")") zero?-exp)
      (expression
        ("if" expression "then" expression "else" expression)
       if-exp)
      (expression
       ("let" identifier "=" expression "in" expression)
       let-exp)
      ))
```

```
;;;;;;;;;;;;;;;; sllgen boilerplate ;;;;;;;;;;;;;;;;

(sllgen:make-define-datatypes the-lexical-spec the-grammar)

(define show-the-datatypes
  (lambda () (sllgen:list-define-datatypes the-lexical-spec the-grammar)))

(define scan&parse
  (sllgen:make-string-parser the-lexical-spec the-grammar))

(define just-scan
  (sllgen:make-string-scanner the-lexical-spec the-grammar))

)
```

This generates the following datatypes:

```
(define-datatype program program?
  (a-program
    (e expression?)))

 (define-datatype expression expression?
   (const-exp
     (n number?))
   (diff-exp
     (exp1 expression?)
     (exp2 expression?))
   (zero?-exp
     (exp1 expression?))
   (if-exp
     (exp1 expression?)
     (exp2 expression?)
     (exp3 expression?))
   (var-exp
     (var symbol?))
   (let-exp
     (var symbol?)
     (rhs-exp expression?)
     (body expression?))))
```

The next module is for data structures: for expressed values, also environments. Later on, these will be mutually recursive, so they will need to go in the same module.

```
(module data-structures (lib "eopl.ss" "eopl")

  (provide (all-defined))               ; too many things to list

;;;;;;;;;;;;;;;; expressed values ;;;;;;;;;;;;;;;;

;;; an expressed value is either a number, a boolean or a procval.

  (define-datatype expval expval?
    (num-val
      (value number?))
    (bool-val
      (boolean boolean?)))

;;; extractors:

  (define expval->num
    (lambda (v)
      (cases expval v
        (num-val (num) num)
        (else (expval-extractor-error 'num v)))))

  (define expval->bool
    (lambda (v)
      (cases expval v
        (bool-val (bool) bool)
        (else (expval-extractor-error 'bool v)))))

  (define expval-extractor-error
    (lambda (variant value)
      (eopl:error 'expval-extractors "Looking for a ~s, found ~s"
        variant value)))
```

```
;;; sloppy->expval :

  (define sloppy->expval
    (lambda (sloppy-val)
      (cond
        ((number? sloppy-val) (num-val sloppy-val))
        ((boolean? sloppy-val) (bool-val sloppy-val))
        (else
          (eopl:error 'sloppy->expval
            "Can't convert sloppy value to expval: ~s"
            sloppy-val)))))

;;;;;;;;;;;;;;;; environment structures ;;;;;;;;;;;;;;;;

;; example of a data type built without define-datatype

  (define empty-env-record
    (lambda ()
      '()))

  (define extended-env-record
    (lambda (sym val old-env)
      (cons (list sym val) old-env)))

  (define empty-env-record? null?)

  (define environment?
    (lambda (x)
      (or (empty-env-record? x)
          (and (pair? x)
               (symbol? (car (car x)))
               (expval? (cadr (car x)))
               (environment? (cdr x)))))))
```

```
  (define extended-env-record->sym
    (lambda (r)
      (car (car r))))

  (define extended-env-record->val
    (lambda (r)
      (cadr (car r))))

  (define extended-env-record->old-env
    (lambda (r)
      (cdr r)))

)
```

We put the code for the environment interface in a separate module.

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

  (require "data-structures.scm")
  (provide init-env empty-env extend-env apply-env)

;;;;;;;;;;;;;;;; initial environment ;;;;;;;;;;;;;;;;

  ;; init-env : () -> environment

  ;; (init-env) builds an environment in which i is bound to the
  ;; expressed value 1, v is bound to the expressed value 5, and x is
  ;; bound to the expressed value 10.

  (define init-env
    (lambda ()
      (extend-env
       'i (num-val 1)
       (extend-env
        'v (num-val 5)
        (extend-env
         'x (num-val 10)
         (empty-env))))))
```

```
;;;;;;;;;;;;;;;; environment constructors and observers ;;;;;;;;;;;;;;;;

  (define empty-env
    (lambda ()
      (empty-env-record)))

  (define empty-env?
    (lambda (x)
      (empty-env-record? x)))

  (define extend-env
    (lambda (sym val saved-env)
      (extended-env-record sym val saved-env)))

  (define apply-env
    (lambda (env search-sym)
      (if (empty-env? env)
        (eopl:error 'apply-env "No binding for ~s" search-sym)
        (let ((sym (extended-env-record->sym env))
              (val (extended-env-record->val env))
              (saved-env (extended-env-record->saved-env env)))
          (if (eqv? search-sym sym)
            val
            (apply-env saved-env search-sym))))))

  )
```

Now we get to the interpreter proper:

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

  (require "lang.scm")
  (require "data-structures.scm")
  (require "environments.scm")

  (provide value-of-program value-of)

;;;;;;;;;;;;;;;; the interpreter ;;;;;;;;;;;;;;;;

  ;; value-of-program : program -> expval

  (define value-of-program
    (lambda (pgm)
      (cases program pgm
        (a-program (body)
          (value-of body (init-env))))))
```

```
;; value-of : expression * environment -> expval

(define value-of
  (lambda (exp env)
    (cases expression exp
```

(value-of (const-exp n) ρ) = n

```
      (const-exp (num) (num-val num))
```

(value-of (var-exp x) ρ) = (apply-env ρ x)

```
      (var-exp (var) (apply-env env var))
```

(value-of (diff-exp e_1 e_2) ρ) =
(- (value-of e_1 ρ) (value-of e_2 ρ))

```
      (diff-exp (exp1 exp2)
        (let ((val1
                (expval->num
                  (value-of exp1 env)))
              (val2
                (expval->num
                  (value-of exp2 env))))
          (num-val
            (- val1 val2))))
```

$$\frac{\text{(value-of } e_1\ \rho) = v}{\text{(value-of (zero?-exp } e_1)\ \rho) = \begin{cases}\text{(bool-val \#t)} & \text{if (expval->num } v) = 0\\ \text{(bool-val \#f)} & \text{if (expval->num } v) \neq 0\end{cases}}$$

```
      (zero?-exp (exp1)
        (let ((val1 (expval->num (value-of exp1 env))))
          (if (zero? val1)
            (bool-val #t)
            (bool-val #f))))
```

$$\frac{(\texttt{value-of}\ \ e_1\ \ \rho) = v}{\begin{array}{l}(\texttt{value-of}\ \ (\texttt{if-exp}\ \ e_1\ \ e_2\ \ e_3)\ \ \rho) \\ \quad = \begin{cases}(\texttt{value-of}\ \ e_2\ \ \rho) & \text{if } (\texttt{expval->bool}\ \ v) = \texttt{\#t} \\ (\texttt{value-of}\ \ e_3\ \ \rho) & \text{if } (\texttt{expval->bool}\ \ v) = \texttt{\#f}\end{cases}\end{array}}$$

```
(if-exp (exp1 exp2 exp3)
  (if (expval->bool (value-of exp1 env))
    (value-of exp2 env)
    (value-of exp3 env)))
```

$$\frac{(\texttt{value-of}\ \ e_1\ \ \rho) = v_1}{(\texttt{value-of}\ \ (\texttt{let-exp}\ \ \mathit{var}\ \ e_1\ \ e_2)\ \ \rho) = (\texttt{value-of}\ \ e_2\ \ [\mathit{var} = v_1]\rho)}$$

```
(let-exp (var rhs-exp body)
  (let ((rhs-val (value-of rhs-exp env)))
    (value-of body
      (extend-env var rhs-val env))))

)))

)
```

3.1.10 Testing

A good test plan is important! One of the tenets of agile programming is test-driven programming:

- Build in support for unit tests from the beginning. We'll use the SRFI 64 framework for that. (It is still very simple, and more flexible than the `test-engine` (`check-expect`) framework.
- Design the tests before you design the program.

We start with a test suite:

```
(module tests mzscheme     ; could have put eopl language here

  (provide test-list)
  ;;;;;;;;;;;;;;;; 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" 10)
      (test-var-2 "-(x,1)" 9)
      (test-var-3 "-(1,x)" -9)

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

      ;; simple conditionals
```

```
    (if-true "if zero?(0) then 3 else 4" 3)
    (if-false "if zero?(1) then 3 else 4" 4)

    ;; test dynamic typechecking
    (no-bool-to-diff-1 "-(zero?(0),1)" error)
    (no-bool-to-diff-2 "-(1,zero?(0))" error)
    (no-int-to-if "if 1 then 2 else 3" error)

    ;; make sure that the test and both arms get evaluated
    ;; properly.
    (if-eval-test-true "if zero?(-(11,11)) then 3 else 4" 3)
    (if-eval-test-false "if zero?(-(11, 12)) then 3 else 4" 4)

    ;; and make sure the other arm doesn't get evaluated.
    (if-eval-test-true-2 "if zero?(-(11, 11)) then 3 else foo" 3)
    (if-eval-test-false-2 "if zero?(-(11,12)) then foo else 4" 4)

    ;; simple let
    (simple-let-1 "let x = 3 in x" 3)

    ;; make sure the body and rhs get evaluated
    (eval-let-body "let x = 3 in -(x,1)" 2)
    (eval-let-rhs "let x = -(4,1) in -(x,1)" 2)

    ;; check nested let and shadowing
    (simple-nested-let "let x = 3 in let y = 4 in -(x,y)" -1)
    (check-shadowing-in-body "let x = 3 in let x = 4 in x" 4)
    (check-shadowing-in-rhs "let x = 3 in let x = -(x,1) in x" 2)

    ))
)
```

Now let's put it all together

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

  ;; top level module.  Loads all required pieces
  ;; say (run-all) to run the test suite.

  (require "data-structures.scm")  ; for sloppy->expval
  (require "lang.scm")             ; for scan&parse
  (require "interp.scm")           ; for value-of-program
  (require "tests.scm")            ; for test-list
  (require srfi/64)                ; for

;;; this is the top-level module, so it needn't provide anything.
;  (provide (all-defined-out))
;  (provide (all-from "interp.scm"))
;  (provide (all-from "lang.scm"))

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

  ;; run : string -> expval

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

```
;; run-all : () -> unspecified
;; runs all the tests in test-list using the srfi-64 test-error and
;; test-expect.

(define run-all
  (lambda ()
    (test-begin "run-all")
    (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 test-pgm))
             (test-assert test-name
                          (equal? (run test-pgm)
                                  (sloppy->expval expected-result))))))
     test-list)
    (test-end "run-all")))

;; run-one : symbol -> expval
;; (run-one sym) runs the test whose name is sym

(define run-one
  (lambda (test-name)
    (let ((the-test (assoc test-name test-list)))
      (cond
        ((assoc test-name test-list)
         => (lambda (test)
              (run (cadr test))))
        (else (eopl:error 'run-one "no such test: ~s" test-name))))))

(run-all)

)
```

We run the tests by executing `top.scm` in the module language, which puts us inside the module and evaluates `(run-all)`.

```
Welcome to DrScheme, version 4.2.1 [3m].
Language: Module; memory limit: 1024 megabytes.
%%%% Starting test run-all  (Writing full log to "run-all.log")
# of expected passes      25
>
```

3.2 PROC: A Language with Procedures

Now it's time to move on to a language with procedures.

We will follow the design of Scheme, and let procedures be expressed values in our language, so that

$$\begin{aligned} \mathit{ExpVal} &= \mathit{Number} + \mathit{Bool} + \mathit{Proc} \\ \mathit{DenVal} &= \mathit{Number} + \mathit{Bool} + \mathit{Proc} \end{aligned}$$

where *Proc* is a set of values representing procedures. We will think of *Proc* as an abstract data type. We consider its interface and specification below.

We will also need syntax for procedure creation and calling. This is given by the productions

$$\begin{aligned} \mathit{Expression} ::= {} & \texttt{proc (}\mathit{Identifier}\texttt{)}\ \ \mathit{Expression} \\ & \boxed{\texttt{proc-exp (var body)}} \\ \mathit{Expression} ::= {} & \texttt{(}\mathit{Expression}\ \ \mathit{Expression}\texttt{)} \\ & \boxed{\texttt{call-exp (exp1 exp2)}} \end{aligned}$$

Some vocabulary words:

In `(proc-exp` *var* *e*`)`, the identifier *var* is the *bound variable* or *formal parameter* and *e* is the *body*. In a procedure call `(call-exp` e_1 e_2`)`, the expression e_1 is the *operator* and e_2 is the *operand* or *actual parameter*. We use the word *argument* to refer to the value of an actual parameter.

3.2.1 The datatype *Proc*

Interface: the constructor `procedure`, which tells how to build a procedure value, and the observer `apply-procedure`, which tells how to apply a procedure value.

Our next task is to determine what information must be included in a value representing a procedure. To do this, we consider what happens when we write a `proc` expression in an arbitrary position in our program.

```
let x = 200
in let f = proc (z) -(z,x)              ;; result should be sub200
   in let x = 100
      in let g = proc (z) -(z,x)        ;; same exp, but result should
                                        ;; be sub100
         in -((f 1), (g 1))
```

Two procedures, created from identical expressions, must behave differently.

Conclusion: value of a `proc` expression must depend in some way on the environment in which the expression is evaluated.

Therefore the constructor `procedure` must take three arguments: the bound variable, the body, and the environment, and the specification for a `proc` expression should be

```
(value-of (proc-exp var e) ρ)
  = (proc-val (procedure var e ρ))
```

where `proc-val` is the constructor, like `bool-val` or `num-val`, that builds an expressed value from a *Proc* .

At a procedure call, we want to find the value of the operator and the operand. If the value of the operator is a `proc-val`, then we want to apply it to the value of the operand.

```
(value-of (call-exp e1 e2) ρ)
  = (let ((v1 (value-of e1 ρ))
          (v2 (value-of e2 ρ)))
      (let ((proc (expval->proc v1)))
        (apply-procedure proc v2)))
```

What happens when `apply-procedure` is invoked?

Lexical scope rule tells us that when a procedure is applied, its body is evaluated in an environment that binds the formal parameter of the procedure to the argument of the call.

Furthermore any other variables must have the same values they had at procedure-creation time.

Therefore these procedures should satisfy the condition

```
(apply-procedure (procedure var e ρ) v)
= (value-of e [var=v]ρ)
```

3.2.2 Calculating with this Specification

Let's do a couple of short examples to show how these things fit together.

Note that this is really a calculation using the *specification*, not the implementation. Let ρ_0 = [x=10]

```
(value-of <<let f = proc(z)-(z,x) in (f 21)>> ρ0)

= (value-of <<(f 21)>>
    [f = (value-of <<proc(z)-(z,x)>> ρ0)]ρ0)

= (value-of <<(f 21)>>
    [f = (procedure z <<-(z,x)>> ρ0)]ρ0)

= (apply-procedure
    (value-of <<f>>  [f = (procedure z <<-(z,x)>> ρ0)]ρ0)
    (value-of <<21>> [f = (procedure z <<-(z,x)>> ρ0)]ρ0))

= (apply-procedure
    (procedure z <<-(z,x)>> ρ0)
    21)

= (value-of <<-(z,x)>> [z=21]ρ0)

= (- (value-of <<z>> [z=21]ρ0)
     (value-of <<x>> [z=21]ρ0))

= (- 21 10)

= 11
```

Exercise: try the same thing for

```
let f = proc (x) proc(y) -(x,y)
in let g = (f 55)
   in (g 33)
```

Now let's try the same example, except let's name the procedure x:

```
(value-of <<let x = proc(z)-(z,x) in (x 21)>> ρ0)

= (value-of <<(x 21)>>
    [x = (value-of <<proc(z)-(z,x)>> ρ0)]ρ0)

= (value-of <<(x 21)>>
    [x = (procedure z <<-(z,x)>> ρ0)]ρ0)

= (apply-procedure
    (value-of <<x>>  [x = (procedure z <<-(z,x)>> ρ0)]ρ0)
    (value-of <<21>> [x = (procedure z <<-(z,x)>> ρ0)]ρ0))

= (apply-procedure
    (procedure z <<-(z,x)>> ρ0)
    21)

= (value-of <<-(z,x)>> [z=21]ρ0)

= (- (value-of <<z> [z=21]ρ0)
    (value-of <<x>> [z=21]ρ0))

= (- 21 10)      ; x is still bound to 10!

= 11
```

3.2.3 Representing Procedures

Must define `apply-procedure` and `procedure` so that

```
(apply-procedure (procedure var e ρ) v)
= (value-of e (extend-env var v ρ))
```

Usual representation is as a data structure:

```
proc? : Scheme value -> boolean
procedure : vble * expression * env -> proc
;; This must be in data-structures.scm, since procedures, expvals,
;; and environments are mutually recursive.
(define-datatype proc proc?
  (procedure
    (var  symbol?)
    (body expression?)
    (env  environment?)))

apply-procedure : proc * expval -> expval
;; This must be in interp.scm, since apply-procedure and
;; value-of are mutually recursive.
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body (extend-env var val saved-env))))))
```

These data structures are often called *closures*, because they are self-contained: they contain everything the procedure needs in order to be applied. We sometimes say the procedure is *closed over* or *closed in* its creation environment.

This representation is in `http://www.ccs.neu.edu/course/cs7400/interps/lecture03/proc-lang/ds`

Alternatively, we can represent a procedure by its action under `apply-procedure`:

proc? : Scheme-value -> boolean
```
(define proc?
  (lambda (v)
    (procedure? v)))
```

procedure : vble * expression * env -> proc
```
;; This must be in interp.scm, since procedure and value-of
;; are mutually recursive.
(define procedure
  (lambda (x e saved-env)
    (lambda (v)
      (value-of e (extend-env x v saved-env)))))
```

apply-procedure : procedure * expval -> expval
```
;; in interp.scm
(define apply-procedure
  (lambda (proc v)
    (proc v)))
```

The function `proc?`, as defined here, is somewhat inaccurate, since not every Scheme procedure is a possible procedure in our language. We need it only for defining the datatype `expval`.

This is is at `http://www.ccs.neu.edu/course/cs7400/interps/lecture03/proc-lang/proc-rep`.

In either case we need to add an alternative to the datatype `expval`

```
(define-datatype expval expval?
  (num-val
    (value number?))
  (bool-val
    (boolean boolean?))
  (proc-val
    (proc proc?)))
```

and we need to add two new clauses to `value-of`

```
        (proc-exp (var body)
          (proc-val (procedure var body env)))

        (call-exp (exp1 exp2)
          (let ((v1 (value-of exp1 env))
                (v2 (value-of exp2 env)))
            (let ((proc (expval->proc v1)))
              (apply-procedure proc v2))))
```

And we're done!

And of course we have to add some tests:

```
;;; in tests.scm:

      ;; simple applications
      (apply-proc-in-rator-pos "(proc(x) -(x,1)  30)" 29)
      (apply-simple-proc "let f = proc (x) -(x,1) in (f 30)" 29)
      (let-to-proc-1 "(proc(f)(f 30)  proc(x)-(x,1))" 29)


      (nested-procs "((proc (x) proc (y) -(x,y)  5) 6)" -1)
      (nested-procs2 "let f = proc(x) proc (y) -(x,y) in ((f -(10,5)) 6)"
        -1)

      (y-combinator-1 "
let fix =  proc (f)
           let d = proc (x) proc (z) ((f (x x)) z)
           in proc (n) ((f (d d)) n)
in let
    t4m = proc (f) proc(x) if zero?(x) then 0 else -((f -(x,1)),-4)
in let times4 = (fix t4m)
   in (times4 3)"
        12)
```

3.3 LETREC: A language with recursive procedures

We now define a new language LETREC, which adds recursion to our language. Since our language has only 1-argument procedures, we make our life simpler by having our letrec's declare only a single 1-argument procedure, for example

```
letrec double(x) = if zero?(x)
                   then 0
                   else -((double -(x,1)), -2)
in (double 6)
```

The left-hand side of a recursive declaration is the name of the recursive procedure and its bound variable. Procedure body is to the right of the =.

Expression ::= `letrec` *Identifier* `(`*Identifier*`)` `=` *Expression* `in` *Expression*
`letrec-exp (proc-name bound-var proc-body letrec-body)`

The value of a `letrec` expression is the value of the body in an environment that has the desired behavior:

`(value-of (letrec-exp` *pname var* e_1 e_2`)` ρ`)`
`= (value-of` e_2 `(extend-env-recursively` *pname var* e_1 ρ`))`

To implement this, just need to add one line to `value-of`.

We add a new procedure `extend-env-recursively` to the environment interface.

We specify the behavior of this environment as follows: Let ρ_1 be the environment produced by (`extend-env-recursively` *pname* var_1 e_1 ρ). Then what should (`apply-env` ρ_1 var_2) return?

1. If the variable var_2 is the same as *pname*, then (`apply-env` ρ_1 var_2 should produce a closure whose bound variable is var_1, whose body is e_1, and with an environment in which *p* is bound to this procedure. But we already have such an environment, namely ρ_1 itself! So

   (`apply-env` ρ_1 *pname*) = (`procedure` var_1 e_1 ρ_1)

2. If var_2 is not the same as *pname*, then

   (`apply-env` ρ_1 var_2) = (`apply-env` ρ var_2)

Let's do an example.

```
(value-of <<letrec double(x) = if zero?(x)
                        then 0
                        else -((double -(x,1)), -2)
            in (double 6)>> ρ0)

= (value-of <<(double 6)>>
    (extend-env-recursively double x <<if zero?(x) ...>> ρ0))
```

```
= (apply-procedure
    (value-of <<double>> (extend-env-recursively double x
                           <<if zero?(x) ...>> ρ0))
    (value-of <<6>> (extend-env-recursively double x
                      <<if zero?(x) ...>> ρ0)))

= (apply-procedure
    (procedure x <<if zero?(x) ...>>
      (extend-env-recursively double x <<if zero?(x) ...>> ρ0))
    6)

= (value-of <<if zero?(x) ...>> [x=6](extend-env-recursively double x
                                       <<if zero?(x) ...>> ρ0))

...

= (-
    (value-of <<(double -(x,1))>> [x=6](extend-env-recursively double x
                                         <<if zero?(x) ...>> ρ0))
    -2)

= (-
    (apply-procedure
      (value-of <<double>> [x=6](extend-env-recursively double x
                                   <<if zero?(x) ...>> ρ0))
      (value-of <<-(x,1)>> [x=6](extend-env-recursively double x
                                   <<if zero?(x) ...>> ρ0)))
    -2)

= (-
    (apply-procedure
      (procedure x <<if zero?(x) ...>>   ;; <-- it found the right procedure!
        (extend-env-recursively double x <<if zero?(x) ...>> ρ0))
      5)
    -2)

= ...
```

We can implement `extend-env-recursively` in any way that satisfies these requirements.

In an abstract-syntax representation, add a new variant, and then make `apply-env` do the right thing.

```
;; in data-structures.scm
(define-datatype environment environment?
  (empty-env)
  (extend-env
    (var symbol?)
    (val expval?)
    (saved-env environment?))
  (extend-env-recursively
    (proc-name symbol?)
    (bound-var symbol?)
    (proc-body expression?)
    (saved-env environment?)))

;; in environments.scm
(define apply-env
  (lambda (env search-sym)
    (cases environment env
      (empty-env ()
        (eopl:error 'apply-env
          "No binding for ~s" search-sym))
      (extend-env (var val saved-env)
        (if (eqv? search-sym var)
          val
          (apply-env saved-env search-sym)))
      (extend-env-recursively (proc-name bound-var proc-body saved-env)
        (if (eqv? search-sym proc-name)
           (proc-val (procedure bound-var proc-body env))
           (apply-env saved-env search-sym))))))
```

The `env` on the next-to-last line corresponds to ρ_1 in the discussion above.

3.4 Simulating `letrec`

We don't really need a `letrec` primitive: in our language, we can do this already: consider the following bit of code:

```
let fix =  proc (f)
            let d = proc (x) proc (z) (f (x x) z)
            in proc (n) (f (d d) n)
    t4m = proc (f) proc (x) if x then +(4,(f -(x,1))) else 0
in let times4 = (fix t4m)
   in (times4 3)
```

Then we have

```
(times4 3)
= ((fix t4m) 3)
= ((t4m (d d)) 3)       ; d = proc (x) proc (z) ((t4m (x x)) z),
= +(4, ((d d) 2))       ; so ((d x) z) = ((t4m (x x)) z)
= +(4, (t4m (d d) 2))   ; ahh- we had ((t4m (d d)) 3) before-- so we've recurred!
= +(4, +(4, ((d d) 1)))
etc.
= +(4, +(4, +(4, ((t4m (d d)) 0))))
= +(4, +(4, +(4, 0)))
= 12
```

See Chapter 9 of the Little Schemer for more discussion.

[Puzzle: What is the justification for using equational reasoning here?]

3.5 Multiple Arguments and Declarations

LETREC illustrated most of the important features of programming languages that we will study. But it's still somewhat primitive. To get a somewhat more realistic language, we need to add more primitives (possibly of several arguments), procedures of several arguments, and declarations of multiple variables.

We'd like to write things like

```
let z = 5
    x = 3
in let x = 4
       y = +(x,z)          % here x = 3
   in *(z, +(x,y))         % here x = 4
```

and

```
letrec
  even(x) = if zero?(x) then 1 else (odd -(x,1))
  odd(x)  = if zero?(x) then 0 else (even -(x,1))
  prod(x,y) = if zero?(x) then 0 else +(x, (prod -(x,1) y))
in (odd 13)
```

This requires almost no new ideas, but a little more engineering. We'll just go over the differences. (Sorry, this isn't in the repository as of right now.)

```
;; In lang.scm:

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

      ...

      ;; application of a primitive
      (expression
        (primitive "(" (separated-list expression ",") ")")
        primcall-exp)
```

```
;; allow many declarations in a let
(expression
  ("let" (arbno identifier "=" expression) "in" expression)
  let-exp)

;; allow many bound variables in a procedure
(expression
  ("proc" "(" (separated-list identifier ",") ")" expression)
  proc-exp)

;; allow many operands in a procedure call
(expression
  ("(" expression (arbno expression) ")")
  call-exp)

;; allow many declarations of n-ary procedures in a letrec
(expression
  ("letrec"
    (arbno identifier "(" (separated-list identifier ",") ")"
      "=" expression)
    "in" expression)
  letrec-exp)

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

))
```

3.5.1 Specification

This will be much like the specification of 1-argument procedures, only with more arguments:

```
(value-of (proc-exp (var_1 ... var_n) e) ρ)
  = (proc-val (procedure (var_1 ... var_n) e ρ))

(value-of (call-exp rator rands) ρ)
  = (let ((rator-val (value-of rator ρ))
          (rand-vals (values-of rands ρ)))
      (let ((proc (expval->proc rator-val)))
        (apply-procedure proc rand-vals)))

(values-of (e_1 ... e_n) ρ)
  = (list (value-of e_1 ρ) ... (value-of e_n ρ))

(apply-procedure (procedure  (var_1 ... var_n) e ρ)
   (v_1 ... v_n))
  = (value-of e [var_1=v_1, ... var_n=v_n]ρ)
```

Also need to modify `extend-env` so that it takes a list of variables and a list of values and zips them into an environment, as indicated in the last equation above.

In `interp.scm`:

```
;; value-of : expression * environment -> expval
(define value-of
  (lambda (exp env)
    (cases expression exp

      ;; new
      (primcall-exp (prim rands)
        (let ((args (values-of rands env)))
          (apply-primitive prim args)))

      ;; changed
      (let-exp (vars rhs-exps body)
        (let ((vals (values-of rhs-exps env)))
          (value-of body
            (extend-env* vars vals env))))

      ;; bvar -> bvars
      (proc-exp (bvars body) (procedure bvars body env))

      ;; rand -> rands
      (call-exp (rator rands)
        (let ((rator-val (value-of  rator env))
              (rand-vals (values-of rands env)))
          (let ((proc (expval->proc rator-val)))
            (apply-procedure proc rand-vals))))


      ;; proc-name -> proc-names, etc.
      (letrec-exp (proc-names bound-varss proc-bodies letrec-body)
        (value-of letrec-body
          (extend-env-recursively proc-names bound-varss proc-bodies env)))

      )))
```

```
  ;; values-of-expressions : (list-of expression) * environment
     -> (list-of expval)
  (define values-of
    (lambda (exps env)
      (map
       (lambda (exp) (value-of exp env))
       exps)))

  (define apply-procedure
    (lambda (proc args)
      (cases procval proc
        (closure (vars body env)
          (value-of body
            (extend-env vars args env))))))
```

We need to add `apply-primitive.scm` (which we `require` in `interp.scm`):

```
(module apply-primitive (lib "eopl.ss" "eopl")

  (require "lang.scm")

  (provide apply-primitive)

  (define apply-primitive
    (lambda (prim args)
      (cases primitive prim
        (subtract-prim ()
          (let ((val1 (expval->num (car args)))
                (val2 (expval->num (cadr args))))
            (num-val (- val1 val2))))
        ...
        )))

  )
```

```
;; in data-structures.scm

(define-datatype environment environment?
  (empty-env)
  (extend-env
    (vars (list-of symbol?))
    (vals (list-of expval?))
    (saved-env environment?))
  (extend-env-recursively
    (proc-names (list-of symbol?))
    (bound-varss (list-of (list-of symbol?)))
    (proc-bodies (list-of expression?))
    (saved-env environment?)))

(define-datatype proc proc?
  (procedure
    (vars (list-of symbol?))
    (body expression?)
    (saved-env environment?)))
```

```
;; in environments.scm

(define apply-env
  (lambda (env search-sym)
    (cases environment env
      (empty-env ()
        (eopl:error 'apply-env
          "No binding for ~s" search-sym))
      (extend-env (vars vals saved-env)
        (let ((maybe-pos (find-pos search-sym vars)))
          (if (number? maybe-pos)
            (list-ref vals maybe-pos)
            (apply-env saved-env search-sym))))
      (extend-env-recursively (proc-names bvarss proc-bodies saved-env)
        (let ((maybe-pos (find-pos search-sym proc-names)))
          (if (number? maybe-pos)
            (proc-val
              (procedure
                (list-ref bvarss maybe-pos)
                (list-ref proc-bodies maybe-pos)
                env))
           (apply-env saved-env search-sym))))))

  ;; find-pos : sym * (listof sym) -> num OR non-number
  ;; (find-pos sym syms) returns the position of sym in syms,
  ;; otherwise returns a non-number

  ;; implementation is left as an exercise (See EOPL3, ex. 1.22).
```