Interpreters

Front end: takes source language program, produces abstract syntax tree.
Interpreter: takes abstract syntax tree, converts it to an answer — possibly using external inputs.
Compilers

Compiler: takes abstract syntax tree, produces text in target language to be interpreted.
Analyzer: tries to gather information.
Translator: performs translation.
Languages and Values

defined (source) language: implemented by the interpreter
defining (host) language: in which the interpreter itself is written — here: Scheme

expressed values: possible values of expressions in a language
denoted values: values bound to variables

e.g. in Scheme:
lots of expressed values: numbers, pairs, …
denoted values: only locations with expressed values
First Language

program: expression

a-program (exp)

expression: number

lit-expr (datum)

: id

var-exp (id)

: primitive '(' expression/',' ')'

primapp-exp (prim rands)

primitive: '+' | '-' | '*' | 'add1' | 'sub1'
(define-datatype program program? (a-program (exp expression?)))

(define-datatype expression expression? (lit-exp (datum number?)) (var-exp (id symbol?)) (primapp-exp (prim primitive?) (rands (list-of expression?)))

the field names are not really relevant
AST for First Language (2)

(define-datatype primitive primitive?
  (add-prim)
  (subtract-prim)
  (mult-prim)
  (incr-prim)
  (decr-prim)
)
Interpreter (1)

(define eval-program
  (lambda (pgm)
    (cases program pgm
      (a-program (body)
        (eval-expression body (init-env)))
    )
  )
)

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

evaluation follows the datatype, i.e., the AST. environment simulates Roman numerals, arbitrary, initialization is not totally transparent.
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (lit-exp (datum) ; represents itself
        datum
      )
      (var-exp (id) ; needs environment
        (apply-env env id)
      )
      (primapp-exp (prim rands)
        (let
          ((args (eval-rands rands env)))
          (apply-primitive prim args)
        )
      )))
)
Interpreter (3)

(define eval-rands
  (lambda (rands env)
    (map
     (lambda (x) (eval-rand x env))
     rands)
  ) )

(define eval-rand
  (lambda (rand env)
    (eval-expression rand env)
  ) )

map does not imply an order of evaluation.
(define apply-primitive
  (lambda (prim args)
    (cases primitive primitive prim
      (add-prim ()
        (+ (car args) (cadr args)))
      (subtract-prim ..)
      (mult-prim ..)
      (incr-prim () (+ (car args) 1))
      (decr-prim ..)
    )
  )
)

BUG: grammar does not guarantee enough operands
Environment-Passing Interpreters

Programming Language Essentials
2nd edition
Chapter 3.2 The Front End
**Terminology**

- **program**: represented as a sequence of characters
- **scanning**: dividing into words, numbers, ...
- **token**: indivisible unit, represented as characters
- **parsing**: organizing the token sequence into hierarchical syntactic structures
- **lexical specification**: defines scanning
- **grammar**: defines parsing
- **scanner**: takes character sequence, produces token sequence
- **parser**: takes token sequence, produces AST
- **parser-generator**: takes lexical specification and grammar and produces scanner and parser
First Language

program:    expression

    a-program (exp)

expression: number

    lit-expr (datum)

    : id

    var-exp (id)

    : primitive ' ( expression/ , , ) '

    primapp-exp (prim rands)

primitive: '+' | '-' | '*' | 'add1' | 'sub1'
(define the-lexical-spec
  '((white-sp (whitespace) skip)
    (comment
      ("%" (arbno (not #\newline)))
      skip
    )
    (id
      (letter (arbno (or letter digit "?")))
      symbol
    )
    (number (digit (arbno digit)) number)
  )
)
(define the-grammar

'((program (expression) a-program)

(expression (number) lit-exp)
(expression (id) var-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)

)

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

(define run ; evaluates string
  (lambda (string)
    (eval-program (scan&parse string))
  )
)

(define read-eval-print ; evaluates stdin
  (sllgen:make-rep-loop "--> " eval-program
    (sllgen:make-stream-parser
     the-lexical-spec the-grammar
    )
  )
)
How to execute an interpreter (1)

$ scheme48 -I eopl.image
> ,load environment.scm
> ,load the-lexical-spec.scm
> ,load the-grammar.scm
> (sllgen:make-define-datatatypes
    the-lexical-spec the-grammar)
> ,load eval-program.scm
> ,load boilerplate.scm

environment and boilerplate can be reused. specifications and interpreter can be combined.
How to execute an interpreter (2)

> (scan&parse "add1(2)")
'(a-program
  (primapp-exp
   (incr-prim)
   ((lit-exp 2)))
)

> (run "add1(2)")
3

> (read-eval-print)
--> add1(2)
3

--> ,
1> ,pop
Extended Language

program: expression

expression: number
  : id
  : primitive '(
    expression /
    ','
  )'
  : 'if' expression 'then' expression
  : 'else' expression

if-exp (test-exp true-exp false-exp)

primitive: '+' | '-' | '*' | 'add1' | 'sub1'
Changes

scanner: remains unchanged.

parser: new rule for expression.

interpreter: new case for if-exp, non-zero means true.

hosting the interpreter in Scheme forces complete decisions on the meaning.
Parser

(define the-grammar
  '(((program (expression) a-program)
     ..
     (expression "if" expression
      "then" expression "else" expression)
     if-exp)
     ..
  ) )
Interpreter

(define eval-expression
  (lambda (exp env)
    (cases expression exp
      ..
      (if-exp (test-exp true-exp false-exp)
        (if (true-value?
          (eval-expression test-exp env))
          (eval-expression true-exp env)
          (eval-expression false-exp env)
        )
      )
    )
  )
)

(define true-value?
  (lambda (x) (not (zero? x)))
)
How to execute

$ scheme48 -I eopl.image
> ,load environment.scm
> ,load 3.3.scm
> ,load boilerplate.scm
> (run "if 1 then 2 else 3")
2
> (run "if -(3,+(1,2)) then 2 else 3")
3

$ cd 3; make 3.3
Extended Language

program: expression

expression: number
  : id
  : primitive (' expression/',' ')'
  : 'if' expression 'then' expression 'else' expression
  : 'let' (id '=' expression)*
    'in' expression

let-exp (ids rands body)

primitive: '+' | '-' | '*' | 'add1' | 'sub1'
Changes

scanner: remains unchanged.

parser: new rule for expression.

interpreter: new case for let-exp, binding will extend environment.

hosting the interpreter in Scheme forces complete decisions on the meaning:

```
let x = 5 y = 6 in + (x, y)
let x = 1 in
  let x = + (x, 2) in add1 (x)
```
(define the-grammar
  '(((program (expression) a-program)
     ..
     (expression
       ("let" (arbno id "=" expression)
          "in" expression)
       let-exp)
     ..
  )
)
)
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      ..
      (let-exp (ids rands body)
        (let ((args (eval-rands rands env)))
          (eval-expression body
            (extend-env ids args env)
          )
        )
      )
    )
  )
)

Scheme's let arranges for the lexical binding in the defined language
How to execute

```scheme
$ scheme48 -I eopl.image
> ,load environment.scm
> ,load 3.4.scm
> ,load boilerplate.scm
> (run "let x = 5 y = 6 in +(x,y)"")
11
> (run "let x = 1 in
    let x = +(x,2) in add1(x)"")
4

$ cd 3; make 3.4
```
Extended Language

program: expr

expr: number | id
    | primitive '(' expr/',' ')'|
    | 'if' expr 'then' expr 'else' expr|
    | 'let' (id '=' expr)* 'in' expr|
    | 'proc' '(' id/',' ')' expr
      proc-exp (ids body)
    | '(' expr expr* ')
      app-exp (rator rands)

primitive: '+' | '-' | '*' | 'add1' | 'sub1'
Changes

scanner: remains unchanged.

parser: new rules for expression.

interpreter: new cases for proc-exp and app-exp, bodies are executed in their lexical environment plus arguments bound to parameters.

Expressed Values = Denoted Values =

\[ \text{number} + \text{procval} \]

let \( f = \text{proc} (y, z) + (y, -(z,5)) \)
in \((f\ 2\ 28)\)
Parser

(define the-grammar
  '((program  (expression) a-program))
  ...
  (expression ("proc"
                "(" (separated-list id ",," ) ")"
                expression)         proc-exp)
  (expression ("(" expression
               (arbno expression) ")")
                app-exp)
  ...
)  )
)
Closure

let \( x = 5 \)

in let \( f = \text{proc} \ (y, z) \ + (y, -(z, x)) \)

\[ x = 28 \]

in \( (f 2 \ x) \)

\text{procval contains environment at time of creation:}

\((\text{closure ids body env})\)

\text{apply-procval extends environment with arguments:}

\((\text{apply-procval (closure ids body env) args})\)

must be

\((\text{eval-expression body}

\ (\text{extend-env ids args env})

)\)
Procedural Representation

(define closure
  (lambda (ids body env)
    (lambda (args)
      (eval-expression body
        (extend-env ids args env)
      )
    )
  )
)

(define apply-procval
  (lambda (proc args)
    (proc args)
  )
)
Datatype Representation

(define-datatype procval procval? 
  (closure 
    (ids (list-of symbol?)) 
    (body expression?) 
    (env environment?) 
  ) )

(define apply-procval
  (lambda (proc args)
    (cases procval proc
      (closure (ids body env)
        (eval-expression body 
          (extend-env ids args env)
        ) ) ) )
)
Interpreter (1)

(define eval-expression
  (lambda (exp env)
    (cases expression exp
      ...
      (proc-exp (ids body)
        (closure ids body env)
      )
      ...
    )
  )
)

procedure definition only creates the closure.
..

```scheme
(app-exp (rator rands)
  (let
    ((proc (eval-expression rator env))
     (args (eval-rands rands env))
    )
    (if (procval? proc)
     (apply-procval proc args)
     (eopl: error 'eval-expression
       "Not a procedure: ~s" proc
     ))
  ))
)
```

procedure application evaluates the operands and binds the arguments to the parameters before evaluating the body.
Terminology

`let f = proc (y, z) +(y, -(z,5))
in (f +(1,1) +(14,14))`

**parameters**, formal parameters, bound variables: defined with the procedure.

**operands**, actual parameters: specified in the procedure application.

**arguments**: values of the operands, bound to the parameters.
How to execute

$ scheme48 -I eopl.image
> ,load environment.scm
> ,load 3.5.scm
> ,load boilerplate.scm
> (run " let f = proc (y, z) +(y, -(z,5))
          in (f 2 28) ")
25

> (run " let x = 5
          in let f = proc (y, z) +(y, -(z,x))
            x = 28
                in (f 2 x) ")
25

$ cd 3; make 3.5
Example

```scheme
(let x = 5
    in let x = 38
        f = proc (y, z) *(y, +(x, z))
        g = proc (u) +(u, x)
        in (f (g 3) 17)

(let ((x 5))
    (let ((x 38)
            (f (lambda (y z) (* y (+ x z))))
            (g (lambda (u) (+ u x)))
              (f (g 3) 17)
        )); 176
```
Trace

(eval-expression

"let x = 5

in let x = 38

  f = proc (y, z) *(y, +(x,z))

  g = proc (u) +(u,x)

in (f (g 3) 17)"

[ initial-environment])
Trace

(eval-expression
  "
  let x = 38
  f = proc (y, z) *(y, +(x,z))
  g = proc (u) +(u,x)
  in (f (g 3) 17)"

[env1: x = 5
 initial-environment])
Trace

(eval-expression
  
(f (g 3) 17)

[env2: x = 38
  f = (closure (y z) "*(y, +(x,z))" env1)
  g = (closure (u) "+(u,x)" env1)

env1: x = 5
  initial-environment)]

Trace

(eval-expression
  (app-exp ("f" ("(g 3)" "17"))

[env2: x = 38
  f = (closure (y z) "*(y, +(x,z))" env1)
  g = (closure (u) "+(u,x)" env1)
  env1: x = 5
  initial-environment])}
Trace

(eval-expression
 (app-exp ("f" ("(g 3)" "17")))
 (let ((proc (eval-expression "f" env2))
      (args (eval-rands ("(g 3)" "17") env2)))

(env2: x = 38
  f = (closure (y z) "*(y, +(x,z))" env1)
  g = (closure (u) "+(u,x)" env1)
(env1: x = 5
  initial-environment)
Trace

(eval-expression
  (app-exp ("f" ("(g 3)" "17"))
    (let ((proc (eval-expression "f" env2))
       (args (eval-rands ("(g 3)" "17") env2))
      (eval-expression "(g 3)" env2))
    [env2: x = 38
     f = (closure (y z) "*(y, +(x,z))" env1)
     g = (closure (u) "+(u,x)" env1)
    env1: x = 5
     initial-environment])
(eval-expression

  (app-exp ("f" ("(g 3)" "17")))

  {let ((proc (eval-expression "f" env2))

    (args (eval-rands ("(g 3)" "17") env2))

    (eval-expression ("(g 3)" env2))

    (app-exp ("g" ("3")))

        [env2: x = 38

         f = (closure (y z) "*(y, +(x,z))" env1)

         g = (closure (u) "+(u,x)" env1)

         env1: x = 5

         initial-environment]})
Trace

(eval-expression
  (app-exp ("f" ("(g 3)" "17"))
    (let ((proc (eval-expression "f" env2))
        (args (eval-rands ("(g 3)" "17") env2)))
  (eval-expression "(g 3)" env2)
  (app-exp ("g" ("3"))
    (let ((proc (eval-expression "g" env2))
        (args (eval-rands ("3" ) env2)))

[env2: x = 38
  f = (closure (y z) "*(y, +(x,z))" env1)
  g = (closure (u) "+(u,x)" env1)
env1: x = 5
  initial-environment])
Trace

(eval-expression
  (app-exp ("f" ("(g 3)" "17")))
    (let ((proc (eval-expression "f" env2))
          (args (eval-rands ("(g 3)" "17") env2)))
      (eval-expression "(g 3)" env2)
    (app-exp ("g" ("3"))
      (let ((proc (closure (u) "+(u,x)" env1))
            (args (3)))
        (env2: x = 38
          f = (closure (y z) "*(y, +(x,z))" env1)
          g = (closure (u) "+(u,x)" env1)
          env1: x = 5
          initial-environment))
Trace

(eval-expression
  (app-exp ("f" ("(g 3)" "17")))
    (let ((proc (eval-expression "f" env2))
        (args (eval-rands ("(g 3)" "17") env2))
        (eval-expression "(g 3)" env2))
  (app-exp ("g" ("3"))
    (let ((proc (closure (u) "+(u,x)" env1))
        (args (3)))
        (apply-procval proc args))
  [env2: x = 38
    f = (closure (y z) "*(y, +(x,z))" env1)
    g = (closure (u) "+(u,x)" env1)
  env1: x = 5
  initial-environment])
Trace

```
(eval-expression
  (app-exp ("f" ("(g 3)" "17")))
  (let ((proc (eval-expression "f" env2))
        (args (eval-rands ("(g 3)" "17") env2))
        (eval-expression "(g 3)" env2)
        (app-exp ("g" ("3"))
                 (apply-procval (closure (u) "+(u,x)" env1) (3))))

[env2: x = 38
  f = (closure (y z) "*(y, +(x,z))" env1)
  g = (closure (u) "+(u,x)" env1)
  env1: x = 5
  initial-environment])
```
Trace

(eval-expression
  (app-exp ("f" ((g 3) "17"))
    (let ((proc (eval-expression "f" env2))
        (args (eval-rands ("(g 3)" "17") env2))
      (eval-expression "(g 3)" env2))
    (app-exp ("g" ("3"))
      (apply-procval (closure (u) "+(u,x)" env1) (3))
      (eval-expression "+(u,x)" [env3: u = 3
                             env1]))
    [env2: x = 38
     f = (closure (y z) "*(y, +(x,z))" env1)
     g = (closure (u) "+(u,x)" env1)
     env1: x = 5
     initial-environment])

[env2: x = 38
 f = (closure (y z) "*(y, +(x,z))" env1)
 g = (closure (u) "+(u,x)" env1)
 env1: x = 5
 initial-environment])
Trace

(eval-expression
  (app-exp ("f" (("g 3") "17"))
    (let ((proc (eval-expression "f" env2))
      (args (eval-rands (("g 3") "17") env2))
    (eval-expression "(g 3)" env2))
  (app-exp ("g" ("3"))
    (apply-procval (closure (u) "+(u,x)" env1) (3))
    (eval-expression "+(3,5)" [env3: u = 3
      env1])
    [env2: x = 38
      f = (closure (y z) "*(y, +(x,z))" env1)
      g = (closure (u) "+(u,x)" env1)
      env1: x = 5
      initial-environment])
)
(eval-expression
  (app-exp ("f" ("(g 3)" "17"))
    (let ((proc (closure (y z) "*(y, +(x,z))" env1))
      (args (8 17)))

[env2: x = 38
  f = (closure (y z) "*(y, +(x,z))" env1)
  g = (closure (u) "+(u,x)" env1)

env1: x = 5
  initial-environment})
Trace

(eval-expression
  (app-exp ("f" ((g 3) "17"))
    (let ((proc (closure (y z) "*(y, +(x,z))" env1))
      (args (8 17)))
      (apply-procval proc args))

  [env2: x = 38
    f = (closure (y z) "*(y, +(x,z))" env1)
    g = (closure (u) "+(u,x)" env1)
    env1: x = 5
    initial-environment])
Trace

(eval-expression
  (app-exp ("f" ("(g 3)" "17")))
  (let ((proc (closure (y z) "*(y, +(x,z))" env1))
        (args (8 17)))
      (apply-procval (closure (y z) "*(y, +(x,z))" env1) (8 17)))

[env2: x = 38
  f = (closure (y z) "*(y, +(x,z))" env1)
  g = (closure (u) "+(u,x)" env1)
  env1: x = 5
  initial-environment])
Trace

(eval-expression
  (app-exp ("f" ((g 3) "17"))
    (let ((proc (closure (y z) "*(y, +(x,z))" env1))
      (args (8 17)))
      (apply-procval
        (closure (y z) "*(y, +(x,z))" env1) (8 17))
        (eval-expression "*(y, +(x,z))" [env4: y = 8
          z = 17
          env1]))
          [env2: x = 38
            f = (closure (y z) "*(y, +(x,z))" env1)
            g = (closure (u) "+(u,x)" env1)
            env1: x = 5
            initial-environment])
Trace

\[
\text{(eval-expression}
\text{(app-exp ("f" ("(g 3)" "17")))}
\text{(let ((proc (closure (y z) "*(y, +(x,z))" env1))}
\text{(args (8 17)))}
\text{(apply-procval}
\text{(closure (y z) "*(y, +(x,z))" env1) (8 17))}
\text{(eval-expression "*(8, +(5,17))" [env4: y = 8}
\text{z = 17 env1])})
\text{[env2: x = 38}
\text{f = (closure (y z) "*(y, +(x,z))" env1)}
\text{g = (closure (u) "+(u,x)" env1)}
\text{env1: x = 5}
\text{initial-environment]})
\]
Simulating Recursion

let mkmult = proc (mk, x)
    if x
        then +(4, (mk mk -(x,1)))
    else 0
in let times4 = proc (x) (mkmult mkmult x)
    in (times4 3)

performs multiplication by repeated addition.

$ cd 3; make 3.5
let factorial =
   proc (factorial, n)
     if n
      then *(n, (factorial factorial -(n,1)))
     else 1
   in let factorial =
      proc (n) (factorial factorial n)
   in (factorial 5)

$ cd 3; make 3.5
Scheme Version

(let ((mkmult
        (lambda (mk x)
          (if (= x 0)
              0
              (+ 4 (mk mk (- x 1)))))
        )
      )
  (let ((times4
          (lambda (x) (mkmult mkmult x))
        )
      )
    (times4 3)
  )

Note that if 0 still acts as true (thanks, Ute!).
Extended Language

program: expr
expr: number | id
    | primitive '( expr/',' ',' )'
    | 'if' expr 'then' expr 'else' expr
    | 'let' (id '=' expr)* 'in' expr
    | 'proc' '( id/',' ',' )' expr
    | '( expr expr expr* ' )'
    | 'letrec' ( id '(' id/',' ')' '=' expr )* 'in' expr

letrec-exp (names idss bodies body)

primitive: '+' | '-' | '*' | 'add1' | 'sub1'
Usage

letrec
    fact(x) =
        if x then *(x, (fact sub1(x))) else 1
    in (fact 6)

letrec
    even(x) = if x then (odd sub1(x)) else 1
    odd(x) = if x then (even sub1(x)) else 0
    in (odd 13)
Changes

scanner: remains unchanged.

parser: new rule for expression.

interpreter: new case for letrec-exp, addition to environment interface.

recursion is restricted to procedures, unlike Scheme.
Parser

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

  (expression ("letrec"
    (arbno id
      "(" (separated-list id ",") ")"
      "=" expression)
    "in" expression) letrec-exp)
  ..
)
)
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (letrec-exp (names idss bodies body)
        (eval-expression body
          (extend-env-recursively
            names idss bodies env
          ) ) ) ) ..
Strategy

(apply-env
  (extend-env-recursively names idss bodies env) sym)

if sym is not in names:
  (apply-env env sym)
else
  (closure ids body (e-e-r ..))

where sym ids body is one of the procedures.

$ cd 3; make 3.6a procedural environment
$ cd 3; make 3.6b datatype-based environment
$ cd 3; make 3.6c prebuilt closures
(define extend-env-recursively
  (lambda (names idss bodies env)
    (letrec
      ((rec-env
         (lambda (sym)
           (let ((pos (list-find-position sym names)))
             (if (number? pos)
                 (closure
                   (list-ref idss pos)
                   (list-ref bodies pos)
                   rec-env
                 )
                 (apply-env env sym)
               ))))))
    rec-env
  ))
)
Datatype-based environment (1)

(define-datatype environment environment? (recursively-extended-env-record
  (names (list-of symbol?))
  (idss (list-of (list-of symbol?)))
  (bodies (list-of expression?))
  (env environment?))
)

(define extend-env-recursively
  (lambda (names idss bodies env)
    (recursively-extended-env-record
      names idss bodies env
    ))
)
(define apply-env
  (lambda (env sym)
    (cases environment env
         (recursively-extended-env-record
          (names idss bodies old-env)
          (let ((pos (list-find-position sym names)))
            (if (number? pos)
                (closure
                 (list-ref idss pos)
                 (list-ref bodies pos)
                 env
                )
                (apply-env old-env sym)))))..
Prebuilt closures (1)

(define-datatype environment environment? (empty-env-record) (extended-env-record
  (syms (list-of symbol?))
  (vals vector?) ; ribcage representation
  (env environment?)
))

(define extend-env
  (lambda (syms vals env)
    (extended-env-record
      syms (list->vector vals) env
    )
  ))


Prebuilt closures (2)

(define extend-env-recursively
  (lambda (names idss bodies old-env)
    (let*
      ((len (length names))
        (vec (make-vector len)) ; preallocate
        (env (extended-env-record names vec old-env))
      )
    (for-each ; populate vector with closures
      (lambda (pos ids body)
        (vector-set! vec pos (closure ids body env))
      )
      (list-from-0-to len) idss bodies
    )
    env
  )
)
Chapter 3.7 Variable Assignment
Extended Language

program: expr

expr: number | id
    | primitive '(% expr/','','')'
    | 'if' expr 'then' expr 'else' expr
    | 'let' (id '=' expr)* 'in' expr
    | 'proc' '(% id/','','')' expr
    | '(% expr expr* ')
    | 'letrec' (id '(% id/','','')' '=' expr )*
    | 'in' expr
    | 'set' id '=' expr

primitive: '+' | '-' | '*' | 'add1' | 'sub1'

varassign-exp (id exp)
let x = 0
in letrec
  even() = if x
  then let d = set x = sub1(x)
  in (odd)
  else 1
odd()  = if x
  then let d = set x = sub1(x)
  in (even)
  else 0
in let d = set x = 13 in (odd)

sequencing by let d = set .. in ..
Usage (2)

```ml
let g = let count = 0
     in proc ()
     let d = set count = add1(count)
     in count

in +((g), (g))

g has hidden state
```
Call by value

let \( x = 100 \)
in let \( p = \text{proc} \ (x) \)
    let \( d = \text{set} \ x = \text{add1}(x) \)
    in \( x \)
in \( +(p \ x), (p \ x) \)

each call copies the argument value,
i.e., the result is 202, not 203
Changes

scanner: remains unchanged.

parser: new rule for expression.

interpreter: new case for varassign-exp, datatype reference to values, addition to environment interface.

Denoted Value = reference(Expressed Value)
Expressed Value = number + procval
Terminology

binding
creates new association of name with value.

assignment
changes value part of existing binding.

assignment is about sharing values if binding is shared.

introducing assignment is likely to change representation of binding, i.e., environment.
(define the-grammar
  '((program (expression) a-program)
    ...
    (expression ("set" id ":=" expression) varassign-exp)
    ...
  ) )
Interpreter

(define eval-expression
    (lambda (exp env)
        (cases expression exp
            (var-exp (id) (apply-env env id))
            (varassign-exp (id exp)
                (begin
                    (setref!
                        (apply-env-ref env id)
                        (eval-expression exp env))
                1 ; need to return a value..)
        )
    )
)

Strategy

(apply-env env id)
has to return the value bound to \text{id}.

(apply-env-ref env id)
has to return a reference bound to \text{id} so that

(setref! a-reference a-value)
can perform the assignment.

\text{set} should maybe return the assigned value rather than 1.
reference (1)

for the ribcage implementation of environment a reference should be a position in a vector:
(define-datatype reference reference? (a-ref
    (position integer?)
    (vec vector?))
)

(define deref
  (lambda (ref)
    (cases reference ref
      (a-ref (pos vec) (vector-ref vec pos))
    )
  )
)

(define setref!
  (lambda (ref val)
    (cases reference reference ref
      (a-ref (pos vec) (vector-set! vec pos val))
    )
  )
)
(define apply-env ; returns value, as before
  (lambda (env sym)
    (deref (apply-env-ref env sym))
  ))

(define apply-env-ref ; returns reference
  (lambda (env sym)
    (cases environment env
      (extended-env-record (syms vals env)
        (let ((pos (list-find-position sym syms)))
          (if (number? pos)
            (a-ref pos vals)
            (apply-env-ref env sym)
          )
        )
      )
    )
  ))

How to execute

$ scheme48 -I eopl.image
> ,load ref-environment.scm
> ,load 3.7.scm
> ,load boilerplate.scm
> (run "
        let x = 100
        in let p = proc (x)
            let d = set x = add1(x)
            in x
            in +((p x), (p x))
"")
202

$ cd 3; make 3.7
Chapter 3.8 Parameter-Passing Variations
Extended Language

program: expr

expr: number | id
   | primitive '()' expr ',' expr
   | 'if' expr 'then' expr 'else' expr
   | 'let' (id '=' expr)* 'in' expr
   | 'proc' '()' id ',' expr
   | '(' expr expr* ')
   | 'letrec' (id '(' id ',' id ')=' expr)* 'in' expr
   | 'set' id '=' expr
   | 'begin' expr ( ';' expr )* 'end'

primitive: '+' | '-' | '*' | 'add1' | 'sub1'
Usage (1)

\[
\text{let } a = 3 \\
\quad p = \text{proc} \ (x) \ \text{set} \ x = 4 \\
\quad \text{in begin} \ (p \ a) \ ; \ a \ \text{end}
\]

\textit{call-by-value}: parameter bound to new location with operand value; \(a\) remains 3.

\textit{call-by-reference}: if operand is variable reference, parameter is bound to location with reference to variable’s location; \(a\) becomes 4.
Usage (2)

let a = 3
    b = 4

swap = proc (x, y)
    let temp = x
    in begin
        set x = y;
        set y = temp
    end

in begin
    (swap a b);
    -(a, b)
end
Changes

scanner: remains unchanged.

parser: new rule for expression. This is just for convenience.

interpreter: new case for begin-exp,
           reference now to direct and indirect targets,
           environment now with targets.

Denoted Value = reference(Expressed Value)
Expressed Value = number + procval
target

(define-datatype target target?
  (direct-target
    (expval expval?))
  (indirect-target
    (ref ref-to-direct-target?)))
)

(define expval?
  (lambda (x)
    (or (number? x) (procval? x)))
)
ref-to-direct-target?

(define ref-to-direct-target?
  (lambda (x)
    (and
      (reference? x)
      (cases reference x
        (a-ref (pos vec)
          (cases target (vector-ref vec pos)
            (direct-target (v) #t)
            (indirect-target (v) #f)
          )
        )
      )
    )
  )
)
```scheme
(define deref
  (lambda (ref)
    (cases target (primitive-deref ref)
      (direct-target (expval) expval)
      (indirect-target (ref1)
        (cases target (primitive-deref ref1)
          (direct-target (expval) expval)
          (indirect-target (p)
            (error 'deref "Illegal reference" ref1)
          )
        )
      )
    )
  )
)

(define primitive-deref ; get value out of vector
  (lambda (ref)
    (cases reference ref
      (a-ref (pos vec) (vector-ref vec pos))
    )
  )
)```
setref!

(define setref!
  (lambda (ref expval)
    (let
      ((ref (cases target (primitive-deref ref)
                  (direct-target (expval1) ref)
                  (indirect-target (ref1) ref1)
                  )))
      (primitive-setref! ref (direct-target expval)))
    )
  )
)

(define primitive-setref! ; set value in vector
  (lambda (ref val)
    (cases reference ref
      (a-ref (pos vec) (vector-set! vec pos val)))
  )
)
Initial environment

(define init-env
  (lambda ()
    (extend-env
      '(i v x)
      (map direct-target '(1 5 10))
      (empty-env)
    ) ) )

every value in the environment has to be coded as target.
environment changes

(define extend-env
  (lambda (syms vals env)
    (if ((list-of target?) vals)
      (extended-env-record syms
        (list->vector vals) env)
      (error 'extend-env)
    ))
)

(define extend-env-recursively
  (lambda (names idss bodies old-env)
    ..
    (vector-set! vec pos
      (direct-target (closure ids body env))
    ) ..
  ))

(define the-grammar
  '(((program (expression) a-program)
     ..
     (expression ("begin" expression
                 (arbno ";" expression) "end")
                   begin-exp)
     ..
   )
  )
)
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (begin-exp (expl exps)
        (let loop
          ((acc (eval-expression expl env))
            (exps exps)
          )
          (if (null? exps)
            acc
            (loop
              (eval-expression (car exps) env)
              (cdr exps)
            )
          )
        )
      )
    )
  )
)
Modify all (sub)expression evaluations:

.. (primapp-exp (prim rands)
    (let ((args (eval-primapp-exp-rands rands env)))
        (apply-primitive prim args)
    )
  )

(define eval-primapp-exp-rands
    (lambda (rands env)
        (map (lambda (x) (eval-expression x env))
            rands))
)
Interpreter (3)

```
.. (let-exp (ids rands body)
    (let ((args (eval-let-exp-rands rands env)))
      (eval-expression body(extend-env ids args env))
    ) ) ..

(define eval-let-exp-rands
  (lambda (rands env)
    (map
      (lambda (x) (eval-let-exp-rand x env))
      rands
    ) ) )

(define eval-let-exp-rand
  (lambda (rand env)
    (direct-target (eval-expression rand env))
  ) )
```
.. (app-exp (rator rands)
    (let ((proc (eval-expression rator env))
        (args (eval-rands rands env)) ..

(define eval-rand ; ensure 1-level indirect
    (lambda (rand env)
        (cases expression rand
            (var-exp (id)
                (indirect-target
                    (let ((ref (apply-env-ref env id)))
                        (cases target (primitive-deref ref)
                            (direct-target (expval) ref)
                            (indirect-target (ref1) ref1)
                        )
                    )
                )
            )
        )
    )))
(else
    (direct-target (eval-expression rand env))
  ))
)
Strategy

\((\text{proc} \ (t, \ u, \ v, \ w))\)

\((\text{proc} \ (a, \ b))\)

\((\text{proc} \ (x, \ y, \ z))\)

\(\text{set} \ y = 13\)

\(a \ b \ 6)\)

\(3 \ v)\)

\(5 \ 6 \ 7 \ 8)\)

\(t \ u \ v \ w\)
How to execute

```
$ scheme48 -I eopl.image
> ,load indir-environment.scm
> ,load 3.8.scm
> ,load call-by-ref.scm
> ,load boilerplate.scm
> (run "
    let a = 3
    p = proc (x) set x = 4
    in begin (p a); a end
"
)
4
```

```
$ cd 3; make 3.8a
```
let y = 0
    g = proc (x, y)
        begin
            set y = 2;
            x
        end
    in (g + (y, y) y)

*call-by-name*: parameter bound to thunk returning reference; result is 4.

Jensen’s device for accessing arrays in Algol.
let g = let count = 0
             in proc ()
                 begin
                     set count = add1(count);
                     count
                 end
             in (proc (x) +(x, x) (g))

*call-by-need, lazy evaluation*: parameter bound to thunk, replaced on first call; result is 3 by-name, 2 by-need.
Terminology

**thunk**: procedure without arguments encapsulating an operand.

**freezing, thawing**: creating the thunk, evaluating it.

**memoization**: the value of the thunk is cached and returned — works in the absence of side-effects.

**β-reduction** in Lambda Calculus, copy rule: replace procedure call by procedure body; in body replace each parameter by corresponding operand.
Changes

scanner: remains unchanged.

parser: remains unchanged.

interpreter: new thunk-target, created by eval-rand, evaluated — and possibly replaced — by deref.

Denoted Value = reference(Expressed Value)
Expressed Value = number + procval
(define-datatype target target? (direct-target (expval expval?)) (indirect-target (ref ref-to-direct-target?)) (thunk-target (exp expression?) (env environment?)))
ref-to-direct-target?

(define ref-to-direct-target?
  (lambda (x)
    (and
      (reference? x)
      (cases reference x
        (a-ref (pos vec)
          (cases target (vector-ref vec pos)
            (direct-target (v) #t)
            (indirect-target (v) #f)
            (thunk-target (exp env) #t)
          )
        )
      )
    )
  )
)
deref

(define deref
  (lambda (ref)
    (cases target (primitive-deref ref)
      (direct-target (expval) expval)
      (indirect-target (ref1)
        (cases target (primitive-deref ref1)
          (direct-target (expval) expval)
          (indirect-target (p)
            (error 'deref "Illegal reference" ref1))
          (thunk-target (exp env) (eval-thunk ref1))
        )
        (thunk-target (exp env) (eval-thunk ref))
      )
    )
  )
)
(define eval-thunk
  (lambda (ref)
    (cases target (primitive-deref ref)
      (thunk-target (exp env)
        (let ((val (eval-expression exp env)))
          (primitive-setref! ref (direct-target val))
            val
          ))
      (else (eopl:error 'eval-thunk "Impossible!")))
    ))

call-by-name results if the thunk is not overwritten by the computed value.
setref!

(define setref!
  (lambda (ref expval)
    (let
      ((ref (cases target (primitive-deref ref)
                (direct-target (expval1) ref)
                (indirect-target (ref1) ref1)
                (thunk-target (exp env) ref)
          )))
      (primitive-setref! ref (direct-target expval)))
    ))
  )
)
Interpreter — call-by-reference

(define eval-rand ; ensure 1-level indirect
  (lambda (rand env)
    (cases expression rand

      (var-exp (id)
        (indirect-target
          (let ((ref (apply-env-ref env id)))
            (cases target (primitive-deref ref)
              (direct-target (expval) ref)
              (indirect-target (ref1) ref1)))))

      (else
        (direct-target (eval-expression rand env))

      )))
)


Interpreter — call-by-need

(define eval-rand ; ensure 1-level indirect
  (lambda (rand env)
    (cases expression rand
      (lit-exp (datum) (direct-target datum)) ; optimize
      (proc-exp (ids body)
        (direct-target (closure ids body env)))
      (var-exp (id)
        (indirect-target
          (let ((ref (apply-env-ref env id)))
            (cases target (primitive-deref ref)
              (direct-target (expval) ref)
              (indirect-target (ref1) ref1)
              (thunk-target (exp env) ref))))))
    (else
      (thunk-target rand env))
  ) ) ) )

How to execute

$ scheme48 -I eopl.image
> ,load need-environment.scm
> ,load 3.8.scm
> ,load call-by-need.scm
> ,load boilerplate.scm
> (run "
    let y = 0
    g = proc (x, y)
    begin set y = 2; x end
    in (g +(y,y) y)
"
)
4
$ cd 3; make 3.8b
Extended Language (1)

program: stmt

stmt: id '=' expr

  | assign-statement (id exp)
  | print-statement (exp)
  | '{' stmt/';' '}'

  | if-statement (exp true-stmt false-stmt)
  | while-statement (exp stmt)
  | block-statement (ids body)
Extended Language (2)

expr: number | id
  | primitive '( expr/',' ')'
  | 'if' expr 'then' expr 'else' expr
  | 'let' (id '=' expr)* 'in' expr
  | 'proc' '(' id/',' ')' expr
  | '(' expr expr* ')
  | 'letrec' ( id '(' id/',' ')' '=' expr )* 'in' expr
  | 'set' id '=' expr
  | 'begin' expr ( ';' expr )* 'end'

primitive: '+' | '-' | '*' | 'add1' | 'sub1'
  | 'equal?' | 'zero?' | 'greater?' | 'less?'
Usage (1)

```plaintext
var x; { x = 3; print (x);
    var x; { x = 4; print (x) };
    print (x) };

assignment requires binding; therefore, a program usually needs `var`.

`var` provides block structure.
```
Usage (2)

```plaintext
var x, y, not; {
    x = 36; y = 54;
    not = proc (x) zero?(x);
    while (not equal?(x, y)) do
        if greater?(x, y)
            x = -(x,y)
            y = -(y,x);
        print (x)
}

statement vs. expression design is confusing.
```
Changes

scanner: remains unchanged.

parser: new rules.

interpreter: new cases for statements and primitives, can be combined with call-by-reference or call-by-need.

Denoted Value = \text{reference}(Expressed Value)
Expressed Value = number + procval
How to execute

$ scheme48 -I eopl.image

> ,load need-environment.scm ; or indir-
> ,load 3.9.scm
> ,load call-by-need.scm ; or -ref
> ,load statement.scm
> ,load boilerplate.scm
> (run"

    var x,y; { 
        x = 3; y = 4; print (+ (x,y))
    }
"

")

7

$ cd 3; make 3.9[ab]