CS 565 — August 25, 1996
Jens Palsberg — Readings: EOPL, Chap. 5

Lecture Note 4:

Interpreters

A Simple Language

⟨expression⟩	::=	⟨integer-const⟩	`lit-exp`
	::=	⟨identifier⟩	`var-exp`
	::=	(⟨expression⟩ $\{$⟨expression⟩$\}^*$)	`app-exp`

(Book uses a different concrete syntax)

```
Concrete Syntax:
(ghgh (x 137) (foo 12 2 46))

Abstract Syntax:
(app-exp
  (var-exp ghgh)
  ((app-exp
     (var-exp x)
     ((lit-exp 137)))
   (app-exp
     (var-exp foo)
     ((lit-exp 12)
      (lit-exp 2)
      (lit-exp 46)))))
```


The Simplest Interpreter

```
(define eval-exp
  (lambda (exp)
    (record-case exp
      (lit-exp (const) const)
      (var-exp (id) (apply-env init-env id))
      (app-exp (rator rands)
        (let ((proc (eval-exp rator))
              (args (eval-rands rands)))
          (apply-proc proc args)))
      (else (error 'eval-exp
              "Bad abstract syntax: ~s" exp)))))

(define eval-rands
  (lambda (rands)
    (map eval-exp rands)))

(define run-string
  (lambda (string)
    (eval-exp (parse-exp (scan-string string)))))
```


Representing Procedures

⟨procedure⟩	::=	⟨primop⟩	`(primitive-proc` ⟨primop⟩`)`
⟨primop⟩	::=	⟨addition⟩	`+-op`
	::=	⟨subtraction⟩	`--op`
	::=	⟨multiplication⟩	`*-op`
	::=	⟨increment⟩	`+1-op`
	::=	⟨decrement⟩	`-1-op`

```
(define-record primitive-proc (primop))

(define apply-proc
  (lambda (proc args)
    (record-case proc
      (primitive-proc (primop)
        (apply-primop primop args))
      (else
        (error 'apply-proc "Bad Procedure: ~s" proc)))))

(define apply-primop
  (lambda (primop args)
    (case primop
      ((+-op)  (+ (car args) (cadr args)))
      ((--op)  (- (car args) (cadr args)))
      ((*-op)  (* (car args) (cadr args)))
      ((+1-op) (+ (car args) 1))
      ((-1-op) (- (car args) 1))
      (else (error 'apply-primop
              "Unknown Primop: ~s" primop)))))
```


Building environments from finite functions

```
;;; the-empty-env : env
;;; extend-env    : names * vals * env -> env
;;; apply-env     : env * name -> val

(define the-empty-env the-empty-ff)
(define extend-env extend-ff*)
(define apply-env apply-ff)
```

Building the initial environment:

```
(define init-pairs
  '((+    +-op)
    (ghgh +-op)                          ; an oddball name
    (-    --op)
    (*    *-op)
    (add1 +1-op)
    (sub1 -1-op)))
```

```
(define build-init-env
  (lambda (pairs)
    (extend-env
      (map car pairs)
      (map make-primitive-proc (map cadr pairs))
      the-empty-env)))

(define init-env (build-init-env init-pairs))
```


Adding if

⟨expression⟩ ::= `if` ⟨expression⟩ `then` ⟨expression⟩ `else` ⟨expression⟩

```
;; abstract syntax:

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

;; concrete syntax:

if x then (+ x y) else if (+ x 1) then z else foo


;; add to interpreter:

       (if-exp (test-exp true-exp false-exp)
         (if (true-val? (eval-exp test-exp))
           (eval-exp true-exp)
           (eval-exp false-exp)))

;; auxiliary:

(define true-val?
  (lambda (val)
    (not (zero? val))))
```


Adding let

```
let x = 5
    y = 6
in  (+ x y)


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

Grammar:

⟨expression⟩ ::= `let` $\{\langle\text{declaration}\rangle\}^*$ `in` ⟨expression⟩
⟨declaration⟩ ::= ⟨identifier⟩ `=` ⟨expression⟩

Abstract syntax:

```
(define-record let-exp (decls body))
(define-record decl (var exp))
```


Interpreter for let

```
(define eval-exp
  (lambda (exp env)
    (record-case exp
      (lit-exp (constant) constant)
      (var-exp (id) (apply-env env id))
      (if-exp (test-exp true-exp false-exp)
        (if
          (true-val? (eval-exp test-exp env))
          (eval-exp true-exp env)
          (eval-exp false-exp env)))
      (let-exp (decls body)
        (let
          ((ids  (map decl->var decls))
           (exps (map decl->exp decls)))
          (let ((new-env
                  (extend-env
                    ids
                    (eval-rands exps env)
                    env)))
            (eval-exp body new-env))))
      (app-exp (rator rands)
        (let ((proc (eval-exp rator env))
              (args (eval-rands rands env)))
          (apply-proc proc args)))
      (else (error 'eval-exp "Bad abstract syntax: ~s" exp)))))
```

```
(define eval-rands
  (lambda (rands env)
    (map (lambda (exp) (eval-exp exp env))
         rands)))

(define run
  (lambda (string)
    (eval-exp (scan&parse string) init-env)))
```


Adding User-Defined Procedures

$\langle\text{exp}\rangle$ ::= `proc` ($\{\langle\text{identifier}\rangle\}^*$) $\langle\text{expression}\rangle$

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

(proc (u) (+ u 5)  3)

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

let f = proc (y) (y (y 5))
    g = proc (z) (+ z 8)
in  (f g)

let f = proc (y) proc (z) (y (y z))
    g = proc (z) (+ z 8)
in ((f g) 5)

let f = proc (y z) (y (y z))
    g = proc (z) (+ z 8)
in  (f g 5)
```


Representing User-Defined Procedures

$\langle\text{procedure}\rangle$ ::= `(closure` `(`$\{\langle\text{identifier}\rangle\}^*$`)` $\langle\text{expression}\rangle$ $\langle\text{environment}\rangle$`)`

```
(define-record closure (formals body env))

(define apply-proc
  (lambda (proc args)
    (record-case proc
      (primitive-proc (primop) (apply-primop primop args))
      (closure (formals body env)
        (eval-exp body (extend-env formals args env)))
      (else (error 'apply-proc
              "Bad Procedure: ~s" proc)))))

(define eval-exp
  (lambda (exp env)
    (record-case exp
      (proc-exp (formals body)
        (make-closure formals body env))
      ...
      )))
```



```
(define eval-exp
  (lambda (exp env)
    (record-case exp
      (lit-exp (constant) constant)
      (var-exp (id) (apply-env env id))
      (proc-exp (formals body)
        (make-closure formals body env))
      (if-exp (test-exp true-exp false-exp)
        (if (zero? (eval-exp test-exp env))
          (eval-exp false-exp env)
          (eval-exp true-exp env)))
      (let-exp (decls body)
        (let ((ids (map decl->var decls))
              (exps  (map decl->exp decls)))
          (let ((new-env
                  (extend-env ids (eval-rands exps env) env)))
            (eval-exp body new-env))))
      (app-exp (rator rands)
        (let ((proc (eval-exp rator env))
              (args (eval-rands rands env)))
          (apply-proc proc args)))
      (else (error 'eval-exp "Bad abstract syntax: ~s" exp)))))


(define eval-rands
  (lambda (rands env)
    (map (lambda (exp) (eval-exp exp env)) rands)))
```

```
(define apply-proc
  (lambda (proc args)
    (record-case proc
      (primitive-proc (primop) (apply-primop primop args))
      (closure (formals body env)
        (eval-exp body (extend-env formals args env)))
      (else (error 'apply-proc "Bad Procedure: ~s" proc)))))
```


Adding Assignment

Simulating cells:

```
(define make-cell
  (lambda (value)
    (cons '*cell value)))

(define deref-cell cdr)

(define set-cell! set-cdr!)                ; danger!
```

Grammar:

⟨expression⟩ ::= `set!` ⟨identifier⟩ ⟨expression⟩

Abstract Syntax

```
(define-record assign-exp (ident rhs-exp))
```


Environments Containing Cells

An environment will be a finite function containing cells.

```
(define eval-exp
  (lambda (exp env)
    (record-case exp
      ;; change:
      (var-exp (id) (deref-cell (apply-env env id)))
      ...
      (assign-exp (ident rhs-exp)
        (set-cell!
          (apply-env env ident)
          (eval-exp rhs-exp env)))
      ...)))
```

And need to change `extend-env` to put the values in cells:

```
(define extend-env
  (lambda (names values env)
    (extend-ff* names (map make-cell values) env)))
```


Adding `letrec`

⟨expression⟩ ::= `letrec` {⟨proc-declaration⟩}* `in` ⟨expression⟩
⟨proc-declaration⟩ ::= `(` ⟨identifier⟩ {⟨identifier⟩}* `)` `=` ⟨expression⟩

```
letrec
  (fact x) = if (zero? x) then 1 else (* x (fact (sub1 x)))
in (fact 6)
```

```
letrec
  (even x) = if (zero? x) then 1 else (odd (sub1 x))
  (odd x)  = if (zero? x) then 0 else (even (sub1 x))
in (odd 13)
```

Abstract syntax:

```
(define-record letrec-exp (rec-decls body))
(define-record rec-decl (var formals body))
```

Interpreter line:

```
(letrec-exp (rec-decls body)
  (eval-exp body (extend-rec rec-decls env)))
```

Two kinds of environment records. Ordinary environments contain cells. Recursive environments contain `rec-decls`. If you look up a `rec-decl`, it gets closed in the environment frame that contains it:

```
(define-record the-empty-env ())
(define-record extended-env (names vals old-env))
(define-record extended-rec-env (rec-decls old-env))

;; we'll use more familiar names for these:

(define extend-env make-extended-env)
(define extend-rec make-extended-rec-env)
```



```
(define apply-env
  (lambda (env var-sym)
    (record-case env
      (the-empty-env ()
        (error 'apply-env "unbound identifier ~s" var-sym))
      (extended-env (names values old-env)
        (if (memq var-sym names)
          (ribassoc var-sym names vals)
          (apply-env old-env var-sym)))
      (extended-rec-env (decls old-env)
        (let ((names (map rec-decl->var decls)))
          (if (memq var-sym names)
            (let ((value (ribassoc var-sym names decls)))
              (make-cell  ; apply-env must return a cell
                (make-closure
                  (rec-decl->formals value)
                  (rec-decl->body value)
                  env)))
            (apply-env old-env var-sym))))
      (else
        (error 'apply-env
          "unknown environment frame ~s"
          env)))))
```

Exercise: avoid doing the `(map rec-decl->var decls)` on every lookup doing the search explicitly in the existing structure.

Variable lookup is the most frequent interpreter operation in Scheme-like languages. What else could be done to speed up variable lookup?