

The Environment Interface

Exercise 2.4

$$\begin{aligned}(\text{empty-stack}) &= [\emptyset] \\ (\text{push } v \ [s]) &= [r], \\ &\quad \text{where } (\text{top } [r]) = v \\ (\text{pop } [s]) &= [r], \\ &\quad \text{where } (\text{push } (\text{top } [s]) \ [r]) = [s] \\ (\text{top } [s]) &= v \\ (\text{empty-stack? } [s]) &= \begin{cases} \#t & [s] = [\emptyset] \\ \#f & \text{otherwise} \end{cases}\end{aligned}$$

`empty-stack`, `push`, and `pop` are constructors and `top` and `empty-stack?` are observers.

Exercise 2.5

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

(define apply-env
  (lambda (env search-var)
    (if (null? env)
        (report-no-binding-found search-var)
        (let ((saved-var (caar env))
              (saved-val (cdar env))
              (saved-env (cdr env)))
          (if (eqv? search-var saved-var)
              saved-val
              (apply-env saved-env search-var))))))
```

```
(define extend-env
  (lambda (var val env)
    (cons (cons var val) env)))
```

Exercise 2.6

This representation is the same as in Figure 2.1 but without tagging the lists with a symbol.

$$\begin{aligned} Env &::= () \\ &::= (Symbol\ SchemeVal\ Env) \end{aligned}$$

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

```
(define apply-env
  (lambda (env search-var)
    (if (null? env)
        (report-no-binding-found search-var)
        (let ((saved-var (car env))
              (saved-val (cadr env))
              (saved-env (caddr env)))
          (if (eqv? search-var saved-var)
              saved-val
              (apply-env saved-env search-var)))))))
```

```
(define extend-env
  (lambda (var val env)
    (list var val env)))
```

This representation uses a list of two lists, one for variables and another for values, where the index of a variable in the variable list corresponds to the index of its related value in the value list.

$$\begin{aligned}
Env & ::= (Var - list \ Val - list) \\
Var - list & ::= () \mid (Symbol \ Var - list) \\
Val - list & ::= () \mid (SchemeVal \ Val - list)
\end{aligned}$$

```

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

(define apply-env
  (lambda (env search-var)
    (scan (car env) (cadr env) search-var)))

(define scan
  (lambda (vars vals search-var)
    (cond ((null? vars)
           (report-no-binding-found search-var))
          ((eqv? (car vars) search-var)
           (car vals))
          (else (scan (cdr vars) (cdr vals) search-var)))))

(define extend-env
  (lambda (var val env)
    (list (cons var (car env))
          (cons val (cadr env)))))

```

This representation is made up of the cons of two lists, the variable list and the value list. The cons makes a single list whose car is the list of variables and whose cdr is the list of values. The index of a variable in the variable list corresponds to the index of its related value in the value list.

It makes no difference that *Var - list* and *Val - list* are defined in terms of the implicit syntactic category *Listof*.

$$\begin{aligned}
Env &::= (Var - list . Val - list) \\
Var - list &::= Listof(Symbol) \\
Val - list &::= Listof(SchemeVal)
\end{aligned}$$

```

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

```

```

(define extend-env
  (lambda (var val env)
    (cons (cons var (car env))
          (cons val (cdr env)))))

```

```

(define apply-env
  (lambda (env search-var)
    (app-env (car env) (cdr env) search-var)))

```

```

(define app-env
  (lambda (vars vals search-var)
    (cond ((null? vars)
           (report-no-binding-found search-var))
          ((eqv? (car vars) search-var)
           (car vals))
          (else (app-env (cdr vars) (cdr vals) search-var)))))

```

Exercise 2.7

```

(define apply-env
  (lambda (env search-var)
    (app-env env search-var env)))

```

```

(define app-env
  (lambda (env search-var e)
    (cond ((eqv? (car env) 'empty-env)

```

```

        (report-no-binding-found search-var))
    ((eqv? (car env) 'extend-env)
     (let ((saved-var (cadr env))
           (saved-val (caddr env))
           (saved-env (cadddr env)))
       (if (eqv? search-var saved-var)
           saved-val
           (app-env saved-env search-var e))))
    (else (report-invalid-env e))))

```

Exercise 2.8

```

(define empty-env?
  (lambda (env)
    (null? env)))

```

Exercise 2.9

```

(define has-binding?
  (lambda (env s)
    (if (null? env)
        #f
        (let ((saved-var (caar env))
              (saved-env (cdr env)))
          (if (eqv? s saved-var)
              #t
              (has-binding? saved-env s))))))

```

Exercise 2.10

```

(define extend-env*
  (lambda (vars vals env)
    (if (null? vars)
        env
        (extend-env (car vars)
                     (car vals)
                     (extend-env* (cdr vars) (cdr vals) env))))

```

```

        (extend-env* (cdr vars)
                     (cdr vals)
                     env))))

```

Exercise 2.11

```

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

(define apply-env
  (lambda (env search-var)
    (if (null? env)
        (report-no-binding-found search-var)
        (let ((saved-vars (caar env))
              (saved-vals (cdar env))
              (saved-env (cdr env)))
          (let ((val (apply-env-in-rib saved-vars
                                       saved-vals
                                       search-var)))
            (if val
                val
                (apply-env saved-env search-var)))))))

(define apply-env-in-rib
  (lambda (vars vals search-var)
    (cond ((null? vars) #f)
          ((eqv? (car vars) search-var) (car vals))
          (else (apply-env-in-rib (cdr vars)
                                   (cdr vals)
                                   search-var)))))

(define extend-env
  (lambda (var val env)
    (cons (cons (list var) (list val))
          env)))

```

```

env)))

(define extend-env*
  (lambda (vars vals env)
    (cons (cons vars vals)
          env)))

```

Procedural Representation

Exercise 2.12

```

(define empty-stack
  (lambda ()
    (lambda (observer)
      (if (eqv? observer 'empty?)
          #t
          (report-stack-is-empty observer))))))

(define push
  (lambda (val stack)
    (lambda (observer)
      (cond ((eqv? observer 'pop) stack)
            ((eqv? observer 'top) val)
            (else #f))))) ; empty-stack?

(define pop
  (lambda (stack)
    (stack 'pop)))

(define top
  (lambda (stack)
    (stack 'top)))

(define empty-stack?

```

```

(lambda (stack)
  (stack 'empty?)))

(define report-stack-is-empty
  (lambda (observer)
    (eopl:error 'empty-stack "Called ~s on an empty stack"
      observer)))

```

Exercise 2.13

```

(define empty-env
  (lambda ()
    (list (lambda (search-var)
            (report-no-binding-found search-var))
          (lambda ()
            #t)))))

(define extend-env
  (lambda (saved-var saved-val saved-env)
    (list (lambda (search-var)
            (if (eqv? search-var saved-var)
                saved-val
                (apply-env saved-env search-var)))
          (lambda ()
            #f)))))

(define apply-env
  (lambda (env search-var)
    ((car env) search-var)))

(define empty-env?
  (lambda (env)
    ((cadr env)))))

```


Exercise 2.14

```
(define empty-env
  (lambda ()
    (list (lambda (search-var)
            (report-no-binding-found search-var))
          (lambda ()
            #t)
          (lambda (s)
            #f)))))

(define extend-env
  (lambda (saved-var saved-val saved-env)
    (list (lambda (search-var)
            (if (eqv? search-var saved-var)
                saved-val
                (apply-env saved-env search-var)))
          (lambda ()
            #f)
          (lambda (s)
            (if (eqv? s saved-var)
                #t
                (has-binding? saved-env s)))))))

(define has-binding?
  (lambda (env s)
    ((caddr env) s)))
```