

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

Problem Set 1B Solutions

B. Programming Problems - Due September 18

In the next two problems, you will extend the PostFix interpreter described in Section 1.2.2 of the Scheme supplement to implement the extensions described in the first two problems. You may want to read appendix A of the Scheme supplement as early as possible.

We especially urge people to work in groups on programming assignments. Remember that you must write up your own solutions and attribute any joint work. See the collaboration policy described in Handout 3 for more information.

First, you should play with the existing PostFix interpreter. Download `ps1/postfix.scm` from the course directory and start the interpreter by evaluating the following Scheme+ expressions:

```
(load "postfix.scm")  
(pf-den-repl)
```

This will start a read-eval-print loop. (There is nothing to turn in for this part—just familiarize yourself with the interpreter.)

Problem 1 (PostFix + Pairs + {dup} Lab)

Extend the interpreter to handle the `pair`, `left`, and `right` commands of Problem 1 as well as the `dup` command discussed in Section 3.8 of the text.

As mentioned in Handout 3 (General Information), your solutions for this and all programming problems should clearly indicate what modifications you have made and include an English explanation of what those modifications do. Demonstrate that your program correctly solves the problem by testing it on a sufficient test suite.

Solution: Add the following lines to the command datatype:

```
(define-datatype command  
  :  
  ($dup)    ; dup  
  ($pair)   ; pairs  
  ($left)  
  ($right)  
)
```

Add the following lines to the parser:

```

(define (pf-command sexp)
  (match sexp
    :
    ('dup ($dup))
    ('pair ($pair))
    ('left ($left))
    ('right ($right))
    (_ (error "unrecognized command")))
  ))

```

Extend denotable values with pairs:

```

(define-datatype den-val
  (int->den-val int)
  (xform->den-val (-> (stack) stack))
  (pair->den-val den-val den-val) ; new
)

```

Add the following cases to the evaluator:

```

(define (eval-command com)
  (match com
    :
    (($dup) (with-value
              (lambda (v)
                (o (push v) (push v))))))
    (($pair) (with-value
                (lambda (v1)
                  (with-value
                    (lambda (v2)
                      (push (pair->den-val v2 v1)))))))
    (($left) (with-pair
                (lambda (left right)
                  (push left))))
    (($right) (with-pair
                (lambda (left right)
                  (push right))))
  ))

```

The auxiliary procedure with-pair is similar to with-integer and with-transform:

```

(define (with-pair proc)
  (with-value
    (lambda (v)
      (match v
        ((pair->den-val left right) (proc left right))
        ((int->den-val _) (error "integer where pair expected"))
        ((xform->den-val _) (error "transform where pair expected"))
      ))))

```

You might also modify `with-integer` and `with-transform` to provide more meaningful error messages.

Modify the unparser to recursively unparse pairs:

```
(define (unparse-value value)
  (match value
    ((int->den-val i) (int->sexp i))
    ((xform->den-val s) 'executable)
    ((pair->den-val left right)
     (cons (unparse-value left)
           (unparse-value right))) ; new
  ))
```

These are our test cases:

```
;;; pairs
```

```
pf-den> (1 2 pair left)
1
```

```
pf-den> (1 2 pair right)
2
```

```
pf-den> (1 2 pair)
(1 . 2)
```

```
pf-den> (1 2 3 pair pair)
(1 2 . 3)
```

```
pf-den> ((dup left swap right pair) 1 2 pair swap exec)
(1 . 2)
```

```
pf-den> ((dup right swap left pair) 1 2 pair swap exec)
(2 . 1)
```

```
;;; dup
```

```
pf-den> (7 (dup mul) exec)
49
```

```
pf-den> ((2 mul) (dup mul) pair)
(executable . executable)
```

```
;;; non-terminating program
```

```
pf-den> ((dup exec) dup exec)
```

```
;Quit!
```

```

;;; error conditions

pf-den> (1 pair)
;Empty stack

pf-den> (1 left)
;Integer where pair expected

pf-den> (dup)
;Empty stack

pf-den> (1 2 3 pair add)
;Pair where integer expected

pf-den> (1 2 add left)
;Integer where pair expected

```

Problem 2 (PostText Lab)

Extend the PostFix interpreter to implement the extensions described in Problem 2.

- a. Implement a dictionary abstraction by writing Scheme+ procedures that have the following interface:
 - (dict-empty) creates an empty dictionary.
 - (dict-bind *name value dict*) returns a new dictionary that extends *dict* with a binding between *name* and *value*.
 - (dict-lookup *name dict*) returns the value associated with *name* in the given dictionary, *dict*.

Based on the suggestion in Exercise 3.44(b), you should represent dictionaries as one-argument procedures that map symbols to values.

- b. Define a new state datatype as follows:

```
(define-datatype state (make-state stack dict))
```

Modify the evaluation procedures in the interpreter so that they map states to states rather than mapping stacks to stacks.

- c. Extend the interpreter to handle the *I*, *def*, and *ref* commands described in Exercise 3.44. One of your test cases should be a non-terminating program.

Solution:

- a. Dictionary abstraction:

```
(define (dict-empty)
  (lambda (id)
    (error "unbound identifier" id)))
```

```

(define (dict-bind id val dict)
  (lambda (lookup-id)
    (if (eq? id lookup-id)
        val
        (dict-lookup lookup-id dict))))

(define (dict-lookup id dict)
  (dict id))

```

- b. Modify the evaluation procedures in the interpreter so that they map states to states rather than mapping stacks to stacks:

Change `eval-program` to create an initial state:

```

(define (eval-program pgm)
  (match pgm
    (($prog seq) (top ((eval-commands seq)
                      (make-state (empty-stack) (dict-empty))))))
  ))

```

All the clauses in `eval-command` use the supporting functions `push`, `with-value`, etc., so we need only modify those functions to be state transformers:

```

(define (push val)
  (lambda (state)
    (match state
      ((make-state stack dict)
       (make-state (cons val stack) dict))))))

(define (with-value proc)
  (lambda (state)
    (match state
      ((make-state stack dict)
       (match stack
         ((null) (error "Empty stack"))
         ((cons v s)
          ((proc v) (make-state s dict))))))))))

```

The other supporting functions are unchanged.

- c. Extend the interpreter to handle the `I`, `def`, and `ref` commands described in exercise 3.27.

Extend the command datatype by adding the following lines:

```

(define-datatype command
  ...
  ($id symbol)
  ($def)
  ($ref)
  )

```

Modify the parser to handle the new commands:

```
(define (pf-command sexp)
  (match sexp
    ...
    ( 'def          ($def)          )
    ( 'ref          ($ref)          )
    ( (symbol->sexp s) ($id s)      )
    ( _             (error "Unrecognized command") )
  ))
```

It's important to make sure that the `(symbol->sexp s)` clause goes last to avoid shadowing other commands.

Extend denotable values with names:

```
(define-datatype den-val
  (int->den-val int)
  (xform->den-val (-> (state) state))
  (name->den-val sym)
)
```

Add the appropriate clauses to the evaluator:

```
(define (eval-command com)
  (match com
    ...
    ( ($id sym)      (push (name->den-val sym)))
    ( ($def)        (with-value
                     (lambda (val)
                       (with-name
                        (lambda (name)
                          (bind-name name val))))))
    ( ($ref)        (with-name
                     (lambda (name)
                       (lookup-name name))))
  ))
```

The auxiliary procedure `with-name` is similar to `with-integer` and `with-transform`:

```
(define (with-name proc)
  (with-value
   (lambda (v)
     (match v
      ((name->den-val sym) (proc sym))
      ((int->den-val _) (error "Integer where name expected"))
      ((xform->den-val _) (error "Transform where name expected"))
      ))))
```

Also modify `with-integer` and `with-transform` to provide more meaningful error messages.

The supporting functions for binding names and looking up values are defined as follows:

```
(define (bind-name name val)
  (lambda (state)
    (match state
      ((make-state stack dict)
       (make-state stack (dict-bind name val dict))))))

(define (lookup-name name)
  (lambda (state)
    (match state
      ((make-state stack dict)
       (make-state (cons (dict-lookup name dict) stack) dict)))))
```

Lastly, change the unparser to handle names: (We also make sure that command sequence results are distinguishable from the name "executable".)

```
(define (unparse-value value)
  (match value
    ((int->den-val i) (int->sexp i))
    ((xform->den-val s) (vector 'executable)) ; prints #(executable)
    ((name->den-val sym) (symbol->sexp sym))
  ))
```

d. Test cases:

The examples from the book:

```
pf-den> (average (add 2 div) def 3 7 average ref exec)
5
```

```
pf-den> (a 3 def dbl (2 mul) def a ref dbl ref exec 4 dbl ref exec add)
14
```

```
pf-den> (a b def a ref 23 def b ref)
23
```

```
pf-den> (a b def a ref 23 def a ref ref)
23
```

A nonterminating program:

```
pf-den> ((a ref exec) a swap def a ref exec)
;Quit!
```

Some error conditions:

```
pf-den> (a 19 def a ref 23 def b ref)
;Integer where name expected
```

```
pf-den> (c 4 def d ref 1 add)
;unbound identifier d
```

```
pf-den> (1 a add)
;Name where integer expected
```

```
pf-den> (a def)
;Empty stack
```

```
pf-den> ((2 mul) double def)
;Transform where name expected
```