A PostFix Interpreter in Racket

PostFix

PostFix is a stack-based mini-language that will be our first foray into the study of metalanguages = programs that manipulate programs.

It’s not a real language, but a “toy” mini-language used for studying programming language semantics and implementation. It is inspired by real stack-based languages like PostScript, Forth, and HP calculators.

For the syntax and semantics of PostFix, see these notes: http://cs.wellesley.edu/~cs251/notes/dcpl-introduction.pdf

Here’s an example PostFix program

(postfix 2 2 nget 0 gt (sub) (swap 1 nget mul) add) sel exec)

PostFix Syntax

A PostFix command C is one of:

- An integer
- One of pop, swap, nget, sel, exec, add, mul, sub, div, rem, ; arithops
  lt, eq, gt; relops
- An executable sequence of the form (C1 ... Cn)

PostFix command semantics (except exec)

<table>
<thead>
<tr>
<th>Stack Before</th>
<th>Command</th>
<th>Stack After</th>
</tr>
</thead>
<tbody>
<tr>
<td>(...)</td>
<td>integer N</td>
<td>(N ...)</td>
</tr>
<tr>
<td>(v1 ...)</td>
<td>pop</td>
<td>(...)</td>
</tr>
<tr>
<td>(v1 v2 ...)</td>
<td>swap</td>
<td>(v2 v1 ...)</td>
</tr>
<tr>
<td>(v1 v2 ...)</td>
<td>sub</td>
<td>(N ...) where N is v2 – v1 (other arithops similar)</td>
</tr>
<tr>
<td>(v1 v2 ...)</td>
<td>lt</td>
<td>(N ...) where N is 1 if v2 &lt; v1 and N is 0 otherwise (other relops similar)</td>
</tr>
<tr>
<td>(i v1 ... vk)</td>
<td>nget</td>
<td>(vi v1 ... vk) if 1 ≤ i ≤ k and vi is an integer</td>
</tr>
<tr>
<td>(velse vthen vtest ...)</td>
<td>sel</td>
<td>(vthen ...) if vtest ≠ 0 (velse ...) if vtest = 0</td>
</tr>
</tbody>
</table>
PostFix command semantics: exec

<table>
<thead>
<tr>
<th>Stack Before</th>
<th>Commands Before</th>
<th>Commands After</th>
</tr>
</thead>
<tbody>
<tr>
<td>((C1 ... Cn) rest-of-stack)</td>
<td>(exec rest-of-cmds)</td>
<td>(C1 ... Cn rest-of-cmds)</td>
</tr>
</tbody>
</table>

PostFix Syntax Abstractions in Racket

```scheme
(define (postfix-program? sexp)
  (and (list? sexp)
       (> (length sexp) 2)
       (eq? (first sexp) 'postfix)
       (integer? (second sexp))
       (postfix-command-sequence? (rest (rest sexp))))

(define (postfix-command-sequence? sexp)
  (and (list? sexp)
       (forall? postfix-command? sexp)))

(define (postfix-command? sexp)
  (or (integer? sexp)
      (memq sexp '(pop swap nget sel exec add mul sub div rem ; arithops lt eq gt)) ; relops
      (postfix-command-sequence? sexp))))

(define (postfix-numargs pgm) (second pgm))
(define (postfix-commands pgm) (rest (rest pgm)))
```

Testing membership with `memq/member`

```scheme
> (member 'c '(a b c d e))
'(c d e) ; returns sublist beginning with found item

> (member 'x '(a b c d e))
#f ; returns #f if item not found

> (define L '(a b))

> (memq L (list '(c d) L '(e f)))
'((a b) (e f))

> (memq L (list '(c d) '(a b) '(e f)))
#f ; not found because new list '(a b) not eq? to L

;; member is to memq what equal? is to eq?
> (member L (list '(c d) '(a b) '(e f)))
'((a b) (e f))
```

Your turn: PostFix program example

Consider this postfix program:

```
'(postfix 2
  2 nget 0 gt (swap 1 nget mul add) sel exec)
```

What is the result of running it on these arguments?

- `'(3 5)`
- `'(3 -5)`
Multiple versions of the PostFix interpreter

We will study three different approaches to implementing a PostFix interpreter.

1. **postfix-config-tail**: uses tail recursion to perform iterative execution of PostFix state configurations = (2-element list) of commands and stack.

2. **postfix-config-iterate**: uses tail recursion to perform iterative execution of PostFix state configurations.

3. **postfix-transform**: uses foldl on command sequence to transform initial stack to final stack. Treats exec as a stack transformer.

There are two flavors of each of these three interpreters:

- **simple**: limited error handling, straightforward arithops/relops, no tracing
- **Fancy**:
  - appropriate handling of all error cases;
  - the ability to trace step-by-step execution;
  - a general, extensible way to handle arithops and relops

```rkt
(define (postfix-run pgm args)
  (postfix-exec-config-tail (postfix-commands pgm) args))
```

```rkt
(define (postfix-exec-config-tail cmds stk)
  (cond ((null? cmds) (first stk)) ; Return top of stack at end
          ((eq? (first cmds) 'exec)
           (flesh-this-out)
           ; Continue iteration with next configuration
           (postfix-exec-config-tail (append (first stk) (rest cmds))
                                      (rest stk)))
          ;; Execute a non-exec command on a stack to yield a new stack.
          ;; So each command can be viewed as a "stack transformer"
          (else (postfix-exec-command cmd stk)
                (postfix-exec-config-tail (rest cmds)
                                          (postfix-exec-command (first stk) stk))))
)
```

```rkt
(define (postfix-exec-command cmd stk)
  (cond ((integer? cmd) 'flesh-this-out)
        ((eq? cmd 'pop) 'flesh-this-out)
        ((eq? cmd 'swap) 'flesh-this-out)
        ((eq? cmd 'sub) 'flesh-this-out)
        ; other arithops similar
        ((eq? cmd 'lt) 'flesh-this-out)
        ; other relops similar
        ((eq? cmd 'sel) 'flesh-this-out)
        ((postfix-command-sequence? cmd) 'flesh-this-out)
        (else (error "unrecognized command" cmd)))))
```

---

**postfix-exec-command Skeleton**

```rkt
(define (postfix-exec-command cmd stk)
  (cond ((integer? cmd) 'flesh-this-out)
        ((eq? cmd 'pop) 'flesh-this-out)
        ((eq? cmd 'swap) 'flesh-this-out)
        ((eq? cmd 'sub) 'flesh-this-out)
        ; other arithops similar
        ((eq? cmd 'lt) 'flesh-this-out)
        ; other relops similar
        ((eq? cmd 'sel) 'flesh-this-out)
        ((postfix-command-sequence? cmd) 'flesh-this-out)
        (else (error "unrecognized command" cmd)))))
```

---

**postfix-exec-config-tail Fleshed Out**

```rkt
(define (postfix-exec-config-tail cmds stk)
  (cond ((null? cmds) (first stk)) ; Return top of stack at end
          ((eq? (first cmds) 'exec)
           (flesh-this-out)
           ; Continue iteration with next configuration
           (postfix-exec-config-tail (append (first stk) (rest cmds))
                                      (rest stk)))
          ;; Execute a non-exec command on a stack to yield a new stack.
          ;; So each command can be viewed as a "stack transformer"
          (else (postfix-exec-command cmd stk)
                (postfix-exec-config-tail (rest cmds)
                                          (postfix-exec-command (first stk) stk))))
)
```
postfix-exec-command Fleshed Out

;; Initially simplify things by ignoring errors
(define (postfix-exec-command cmd stk)
  (cond ((integer? cmd) (cons cmd stk))
        ((eq? cmd 'pop) (rest stk))
        ((eq? cmd 'swap)
            (cons (second stk) (cons (first stk) (rest (rest stk))))))
        (cond ((eq? cmd 'sub)
            (cons (- (second stk) (first stk)) (rest (rest stk))))
              ((eq? cmd 'lt)
                (cons (if (< (second stk) (first stk)) 1 0) (rest (rest stk))))
              (else error "unrecognized command" cmd))))

Side Effects and Sequencing: printf and begin

> (begin (printf "~a + ~a is ~a\n" 1 2 (+ 1 2))
  (printf "~a * ~a is ~a\n" 3 4 (* 3 4)))
1 + 2 is 3
3 * 4 is 12

(define (print-and-return val)
  (begin (printf "~a\n" val) val))

> (* (print-and-return 3)
  (print-and-return (+ (print-and-return 4)
                     (print-and-return 5))))
3 ; printed
4 ; printed
5 ; printed
9 ; printed
27 ; returned

postfix-exec-config-tail with tracing

;; Set this to #t to turn on printing of intermediate stacks;
;; #f to turn it off
(define display-steps? #t)

(define (postfix-exec-config-tail cmds stk)
  (begin (if display-steps? ; Only print intermediate stack
           (begin e1 e2 ...

               desugars to (let ((id1 e1)) ; id1 is fresh
                             (begin e2 ...)))
             'do-nothing)))
postfix-run

;; Run a postfix program on initial stack from args
;; Simplify things by not checking for errors.
(define (postfix-run pgm args)
  (let ((final-stk (postfix-exec-commands (postfix-commands pgm) args)))
    (first final-stk)))

> (postfix-run '(postfix 2 7 4 pop swap sub) '(5 8))
after executing 7, stack is (7 5 8)
after executing 4, stack is (4 7 5 8)
after executing pop, stack is (7 5 8)
after executing swap, stack is (5 7 8)
after executing sub, stack is (2 8)
2

postfix-run with errors

;; Run a postfix program on initial stack from args
;; This version checks for errors
(define (postfix-run pgm args)
  (cond ((not (postfix-program? pgm))
            (error "Invalid PostFix program" pgm))
        ((not (postfix-arguments? args))
            (error "Invalid PostFix arguments")
        ((not (= (postfix-numargs pgm) (length args)))
            (error "expected number of arguments does not match actual number of arguments"
                (list (postfix-numargs pgm) (length args))))
        (else (let ((final-stack
                        (postfix-exec-commands (postfix-commands pgm) args)))
                   (cond ((null? final-stack)
                             (error "Stack empty at end of program")
                         ((not (integer? (first final-stack)))
                          (error "Top of final stack not an integer")
                          (else (first final-stack))))))))

postfix-exec-command with errors

(define (postfix-exec-command cmd stk)
  (cond
    ((integer? cmd) (cons cmd stk))
    ((eq? cmd 'pop)
      (if (< (length stk) 1)
          (error "postfix pop requires stack with at least one value"
              (list cmd stk))
          (rest stk)))
    (else (cond
               ((eq? cmd 'swap)
                (if (< (length stk) 2)
                    (error "postfix swap requires stack with at least two values"
                    (list cmd stk))
                    (cons (second stk) (cons (first stk) (rest (rest stk)))))
                (else (error "Unknown PostFix command" cmd))))))

Better handling of arithops (relops similar)

(define (postfix-exec-command cmd stk)
  (cond ...
    ((postfix-arithop? cmd)
      (cons ((postfix-arithop->racket-binop cmd)
             (second stk)
             (first stk))
             (rest (rest stk))))
    ...
    (else (error "Unknown PostFix command" cmd)))))

(define postfix-arithops
  (list (list 'add '+) (list 'mul '*) (list 'sub '-')
        (list 'div quotient) (list 'rem remainder)))

(define (postfix-arithop cmd)
  (assoc cmd postfix-arithops))

(define (postfix-arithop->racket-binop arithop)
  (second (assoc postfix-arithops)))
postfix-config-iterate-simple.rkt

(define (postfix-run pgm args)
  (postfix-exec-config-iterate (postfix-commands pgm) args))

(define (postfix-exec-config-iterate cmds stk)
  (iterate-apply postfix-exec-config-one-step
   (λ (cmds stk) (null? cmds))
   (λ (cmds stk) (first stk))
   (list cmds stk)))

(define (postfix-exec-config-one-step cmds stk)
  (if (eq? (first cmds) 'exec)
      (list (append (first stk) (rest cmds))
            (rest stk)))
      (list (rest cmds)
            (postfix-exec-command (first cmds) stk)))))

postfix-transform-simple.rkt

(define (postfix-run pgm args)
  (let {
        [final-stk
         (postfix-exec-commands (postfix-commands pgm) args)]
      (first final-stk))
    ;; Execute command list on initial stack
    ;; and return final stack
    (define (postfix-exec-commands cmds init-stk)
      (foldl postfix-exec-command init-stk cmds))

    (define (postfix-exec-command cmd stk)
      (cond ...
        ((eq? cmd 'exec)
         (postfix-exec-commands (first stk) (rest stk)))
        (else (error "unrecognized command" cmd))))

    > (postfix-exec-commands '(pop swap sub) '(4 7 5 8))
    '(2 8)

postfix-exec-commands with tracing

;; Execute command list on initial stack
;; and return final stack
;; Print each command and stack resulting from executing it
(define (postfix-exec-commands cmds init-stk)
  (foldl (λ (cmd stk)
           (let ((new-stk (postfix-exec-command cmd stk)))
             (begin (printf "after executing ~a, stack is ~a\n" cmd new-stk)
                    init-stk
                    cmds)))
         init-stk
         cmds))

> (postfix-exec-commands '(pop swap sub) '(4 7 5 8))
after executing pop, stack is (7 5 8)
after executing swap, stack is (5 7 8)
after executing sub, stack is (2 8)
'(2 8)