A PostFix Interpreter in Racket

CS251 Programming Languages
Spring 2017, Lyn Turbak
Department of Computer Science
Wellesley College

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 \((C_1 \ldots C_n)\)

PostFix Syntax

- `number of expected arguments`
- `commands`

A PostFix program

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

Here’s an example PostFix program

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

PostFix command semantics (except exec)

<table>
<thead>
<tr>
<th>Stack Before</th>
<th>Command</th>
<th>Stack After</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\ldots))</td>
<td>integer (N)</td>
<td>((N \ldots))</td>
</tr>
<tr>
<td>((v_1 \ldots))</td>
<td>pop</td>
<td>((\ldots))</td>
</tr>
<tr>
<td>((v_1 v_2 \ldots))</td>
<td>swap</td>
<td>((v_2 v_1 \ldots))</td>
</tr>
<tr>
<td>((v_1 v_2 \ldots))</td>
<td>sub</td>
<td>((N \ldots)) where (N) is (v_2 - v_1)</td>
</tr>
<tr>
<td>((v_1 v_2 \ldots))</td>
<td>lt</td>
<td>((N \ldots)) where (N) is 1 if (v_2 &lt; v_1) (N) is 0 otherwise</td>
</tr>
<tr>
<td>((i v_1 \ldots v_k))</td>
<td>nget</td>
<td>((v_i v_1 \ldots v_k)) if (1 \leq i \leq k) (v_i) is an integer</td>
</tr>
<tr>
<td>((\text{velse } v\text{then } v\text{test } \ldots))</td>
<td>sel</td>
<td>((v\text{then } \ldots)) if (v\text{test} \neq 0) ((\text{velse } \ldots)) if (v\text{test} = 0)</td>
</tr>
</tbody>
</table>

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 command semantics: exec

<table>
<thead>
<tr>
<th>Stack Before</th>
<th>Commands Before</th>
<th>Commands After</th>
</tr>
</thead>
<tbody>
<tr>
<td>((C_1 \ldots C_n) \ldots)</td>
<td>(\text{exec} \ldots)</td>
<td>((C_1 \ldots C_n) \ldots)</td>
</tr>
</tbody>
</table>

PostFix Syntax Abstractions in Racket

\[
\begin{align*}
&\text{(define (postfix-program? sexp)} \\
&\quad \text{(and (list? sexp)} \\
&\quad \quad \text{(>= (length sexp) 2)} \\
&\quad \quad \text{(eq? (first sexp) 'postfix)} \\
&\quad \quad \text{(integer? (second sexp)} \\
&\quad \quad \text{(postfix-command-sequence? (rest (rest sexp)))))})
\end{align*}
\]

\[
\begin{align*}
&\text{(define (postfix-command-sequence? sexp)} \\
&\quad \text{(and (list? sexp)} \\
&\quad \quad \text{(forall? postfix-command? sexp))})
\end{align*}
\]

\[
\begin{align*}
&\text{(define (postfix-command? sexp)} \\
&\quad \text{(or (integer? sexp)} \\
&\quad \quad \text{(memq sexp '(pop swap nget sel exec add mul sub div rem ; arithops lt eq gt))} \\
&\quad \quad \text{(relops (postfix-command-sequence? sexp)))))
\end{align*}
\]

\[
\begin{align*}
&\text{(define (postfix-numargs pgm) (second pgm))} \\
&\text{(define (postfix-commands pgm) (rest (rest pgm))))}
\end{align*}
\]

Testing membership with member

\[
\begin{align*}
&> (\text{member 'c '(a b c d e))} \\
&\quad \text{'(c d e)} \\
&> (\text{member 'x '(a b c d e))} \\
&\quad \#f
\end{align*}
\]

postfix-config-tail-starter.rkt

;; Run the given PostFix program on argument values, 
;; which form the initial stack 
(define (postfix-run pgm args)
  (postfix-exec-config-tail (postfix-commands pgm) args))

;; Use tail recursion to loop over a configuration state consisting 
;; of (1) list of commands and (2) list of stack values 
(define (postfix-exec-config-tail cmds stk)
  (cond ((null? cmds) 'flesh-this-out)
        (eq? (first cmds) 'exec)
        (postfix-exec-command (first cmds) stk)
        (rest cmds))
  
  (cond ((null? cmds) 'flesh-this-out)
        (postfix-exec-command-sequence? (rest (rest sexp)))))

;; Execute a non-exec command on a stack to yield a new stack. 
;; So each command can be viewed as a "stack transformer" 
(define (postfix-exec-command cmd stk) ...)

postfix-exec-command Skeleton

;; Initially simplify things by ignoring errors
(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)))))

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
begin is just sugar!

(begin e) desugars to e

(begin e1 e2 ...) desugars to (let ((id1 e1)) ; id1 is fresh 
(begin e2 ...))

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 (if display-steps? is #t
                     (printf "Commands: ~a\n Stack: ~a\n" cmds stk)
                     'do-nothing)
           (cond ...)))

postfix-exec-commands

;; Execute command list on initial stack
;; and return final stack
(define (postfix-exec-commands cmds init-stk)
  (foldl postfix-exec-command init-stk cmds))

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

Handling exec in postfix-exec-command

(define (postfix-exec-command cmd stk)
  (cond ...)
            (postfix-exec-commands (first stk) (rest stk)))
      (else (error "unrecognized command" cmd))))
**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
" cmd new-stk)
        init-stk
        cmds))
    new-stk)
  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)

**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" pgm))
  ((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)))))))

**Better handling of arithops**

(define (postfix-exec-command cmd stk)
  (cond _
    ((postfix-arithop? cmd)
      (cons ((postfix-arithop->racket-binop cmd)
        (second stk)
        (first stk))
      (rest (rest stk)))))

(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-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\}\}
   \{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\}\}\)\}\}
   \{(postfix-arithop? cmd)
   \{cond \{(< (stack-size stk) 2)
     \{error "postfix arithop requires two arguments" \{list cmd stk\}\}
   \{or \{(not (integer? (first stk))\)
    \{(not (integer? (second stk))\)
   \{error "postfix arithop requires two integers" \{list cmd stk\}\}
   \{else (\{(postfix-arithop->racket-binop cmd)
    \{second stk\} \{first stk\}
   \{rest \{rest stk\}\}\)\}
   \{other cases omitted\}
   \{else (error "Unknown PostFix command" cmd)\}\}))

II-21