A PostFix Interpreter in Racket

CS251 Programming Languages
Spring 2016, Lyn Turbak
Department of Computer Science
Wellesley College

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.

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 (mul) (swap 1 nget mul add) sel exec)

PostFix Syntax

number of expected arguments

(commands

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

(executable sequence command)

(executable sequence command)

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 Syntax Abstractions in Racket

(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 member

> (member 'c '(a b c d e))
'(c d e)

> (member 'x '(a b c d e))
#f

---

PostFix command semantics

<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</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</td>
</tr>
<tr>
<td>(v1 v1 ... v2)</td>
<td>nget</td>
<td>(v1 v1 ... v2) if 1 ≤ i ≤ k</td>
</tr>
<tr>
<td>(else vthen vtest ...)</td>
<td>sel</td>
<td>(vthen ...) if vtest + 0 = 0 (velse ...) if vtest = 0</td>
</tr>
<tr>
<td>(...)</td>
<td>(C1 ... Cn)</td>
<td>((C1 ... Cn) ...)</td>
</tr>
</tbody>
</table>

---

postfix-exec-command Skeleton

;; 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) (rest (rest stk))))
        ((eq? cmd 'sub) (cons (- (second stk) (first stk)) (rest (rest stk))))
        ; other arithops similar
        ((eq? cmd 'lt) (cons (if (< (second stk) (first stk)) 1 0) (rest (rest stk))))
        ; other relops similar
        ((eq? cmd 'nget) (cons (list-ref stk (first stk)) (rest stk)))
        ((eq? cmd 'sel) (cons (if (= (third stk) 0) (first stk) (second stk)) (rest (rest stk))))
        ; other arithops similar
        ((eq? cmd 'lt) (cons (if (< (second stk) (first stk)) 1 0) (rest (rest stk))))
        ; other relops similar
        ((eq? cmd 'nget) (cons (list-ref stk (first stk)) (rest stk)))
        ((eq? cmd 'sel) (cons (if (= (third stk) 0) (first stk) (second stk)) (rest (rest stk))))
        ; other arithops similar
        ((eq? cmd 'lt) (cons (if (< (second stk) (first stk)) 1 0) (rest (rest stk))))
        ; other relops similar
        ((eq? cmd 'nget) (cons (list-ref stk (first stk)) (rest stk)))
        ((eq? cmd 'sel) (cons (if (= (third stk) 0) (first stk) (second stk)) (rest (rest stk))))
        ; other arithops similar
        ((eq? cmd 'lt) (cons (if (< (second stk) (first stk)) 1 0) (rest (rest stk))))
        ; other relops similar
        ((eq? cmd 'nget) (cons (list-ref stk (first stk)) (rest stk)))
        ((eq? cmd 'sel) (cons (if (= (third stk) 0) (first stk) (second stk)) (rest (rest stk))))
        ;; See slide 11-10
        (else (error "unrecognized command" cmd)))

---

postfix-exec-command Flesched 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) (rest (rest stk))))
        ((eq? cmd 'sub) (cons (- (second stk) (first stk)) (rest (rest stk))))
        ; other arithops similar
        ((eq? cmd 'lt) (cons (if (< (second stk) (first stk)) 1 0) (rest (rest stk))))
        ; other relops similar
        ((eq? cmd 'nget) (cons (list-ref stk (first stk)) (rest stk)))
        ((eq? cmd 'sel) (cons (if (= (third stk) 0) (first stk) (second stk)) (rest (rest stk))))
        ; other arithops similar
        ((eq? cmd 'lt) (cons (if (< (second stk) (first stk)) 1 0) (rest (rest stk))))
        ; other relops similar
        ((eq? cmd 'nget) (cons (list-ref stk (first stk)) (rest stk)))
        ((eq? cmd 'sel) (cons (if (= (third stk) 0) (first stk) (second stk)) (rest (rest stk))))
        ; other arithops similar
        ((eq? cmd 'lt) (cons (if (< (second stk) (first stk)) 1 0) (rest (rest stk))))
        ; other relops similar
        ((eq? cmd 'nget) (cons (list-ref stk (first stk)) (rest stk)))
        ((eq? cmd 'sel) (cons (if (= (third stk) 0) (first stk) (second stk)) (rest (rest stk))))
        ;; See slide 11-10
        (else (error "unrecognized command" cmd))))
**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)))
```

---

**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-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)

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-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))
          (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))
    ((eqq? cmd 'pop)
      (if (< (length stk) 1)
          (error "postfix pop requires stack with at least one value"
                (list cmd stk))
          (rest stk)))
    ((eqq? 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 (apply (postfix-arithop->racket-binop cmd)
                        (second stk) (first stk))
                   (rest (rest stk))))
    (else (error "Unknown PostFix command" cmd))))

;; Other cases omitted
(else (error "Unknown PostFix command" cmd)))