

## A PostFix Interpreter in Racket



**CS251 Programming Languages**  
Spring 2017, Lyn Turbak

Department of Computer Science  
Wellesley College

## 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:

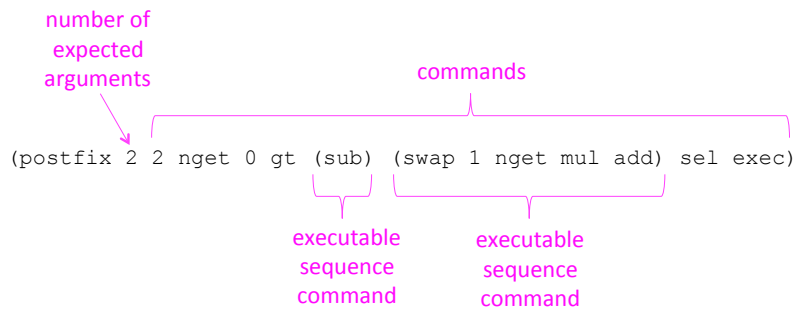
<http://cs.wellesley.edu/~cs251/notes/dcpl-introduction.pdf>

Here's an example PostFix program

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

11-2

## 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)`

11-3

## PostFix command semantics (except exec)

Stack Before	Command	Stack After
(...)	integer $N$	( $N$ ...)
( $v1$ ...)	pop	(...)
( $v1$ $v2$ ...)	swap	( $v2$ $v1$ ...)
( $v1$ $v2$ ...)	sub (other arithops similar)	( $N$ ...) where $N$ is $v2 - v1$
( $v1$ $v2$ ...)	lt (other relops similar)	( $N$ ...) where $N$ is 1 if $v2 < v1$ and $N$ is 0 otherwise
( $i$ $v1$ ... $vk$ )	nget	( $vi$ $v1$ ... $vk$ ) if $1 \leq i \leq k$ and $vi$ is an integer
( $velse$ $vthen$ $vtest$ ...)	sel	( $vthen$ ...) if $vtest \neq 0$ ( $velse$ ...) if $vtest = 0$

11-4

## PostFix command semantics: exec

Stack Before	Commands Before	Commands After
((C1 ... Cn) ...)	(exec ...)	((C1 ... Cn) ...)

11-5

## 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)))
```

11-6

## Testing membership with member

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

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

11-7

## 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 ; Return top of stack at end
        (eq? (first cmds) 'exec)
        ; Continue iteration with next configuration
        'flesh-this-out)
        ; Continue iteration with next configuration
        (else (postfix-exec-config-tail
                (rest cmds)
                (postfix-exec-command (first cmds) stk)))))

;; 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) ...)
```

11-8

## 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))))
```

11-9

## postfix-exec-config-tail Fleshed Out

```
;; 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) (first stk)) ; Return top of stack at end
        ((eq? (first cmds) 'exec)
         ; Continue iteration with next configuration
         (postfix-exec-config-tail (append (first stk) (rest cmds))
                                   (rest stk)))
        ; Continue iteration with next configuration
        (else (postfix-exec-config-tail
                (rest cmds)
                (postfix-exec-command (first cmds) stk)))))
```

11-10

## 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)))))
        ((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 (rest stk)))))
        ((postfix-command-sequence? cmd) (cons cmd stk))
        (else (error "unrecognized command" cmd))))
```

11-11

## 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
```

11-12

## begin is just sugar!

```
(begin e) desugars to e

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

11-13

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

11-14

## 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)
```

11-15

## 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))))
```

11-16

## 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 (lambda (cmd stk)
          (let ((new-stk (postfix-exec-command cmd stk)))
            (begin (printf "after executing ~a, stack is ~a\n"
                          cmd new-stk)
                   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)
```

11-17

## 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
```

11-18

## 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 arithop)))
```

11-19

## 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)))))))))
```

11-20

## 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 (cons ((postfix-arithop->racket-binop cmd)
                         (second stk) (first stk))
                         (rest (rest stk))))))
          ;; Other cases omitted
          (else (error "Unknown PostFix command" cmd))))))
```