CS301 Compiler Design and Implementation **Handout # 10**
Prof. Lyn Turbak **September 17, 2003**
Wellesley College ***Revised September 27, 2003***

PostFix: A Simple Stack Language

Several exercises and examples in this course will involve the PostFix mini-language. PostFix is a simple stack-based language inspired by the PostScript graphics language, the Forth programming language, Hewlett Packard calculators, and stack-based bytecode interpreters. Here we give a brief introduction to PostFix.

1 Syntax

1.1 Commands

The basic syntactic unit of a PostFix program is the *command*. Commands have the following form:

- Any integer numeral. E.g., `17`, `0`, `-3`.
- Any string literal (possibly containing the usual escape characters). E.g., `"CS301"`, `"Golf? No sir -- prefer prison flog!"`, `"You say \"Goodbye\"\n\tand I say \"Hello\"."`
- One of the following special command tokens: `add`, `div`, `eq`, `exec`, `ge`, `get`, `gt`, `le`, `lt`, `mul`, `ne`, `pop`, `pri`, `prs`, `put`, `rem`, `sel`, `sub`, and `swap`.
- An *executable sequence* — a single command represented as a parenthesized sequence of subcommands separated by whitespace.[1] E.g., `(7 add 3 swap)` and `(2 (5 mul) exec add)`.

Since executable sequences contain other commands (including other executable sequences), they can be arbitrarily nested. An executable sequence counts as a single command despite its hierarchical structure.

1.2 Programs

A PostFix *program* is a parenthesized sequence of consisting of (1) the token `postfix` followed by (2) an integer indicating the number of program parameters followed by (3) any number of PostFix commands. For example, here are some sample PostFix programs:

[1] Whitespace is any contiguous sequence of space, tab, and newline characters.

```
(postfix 1 0 get mul)

(postfix 2 add 2 div)

(postfix 4 0 get 1 get mul 4 get mul swap 3 get mul add add)

(postfix 1 (1 get 0 eq
            ("\n" prs)
            ("\n  n=" prs 1 get pri "; ans=" prs 0 get pri
             1 get mul 1 get 1 sub 1 put 2 get exec)
            sel exec)
           swap
           1 2 get exec)

(postfix 1 ((2 get swap exec) (3 mul swap exec) swap)
           (5 sub) swap exec exec)
```

1.3 Abstract vs. Concrete Syntax

The abstract syntax of PostFix programs and commands is captured in the following OCaml datatype definitions:

```
type pgm = (* PostFix programs *)
    Pgm of int * com list

and com = (* PostFix commands *)
    Int of int     (* push integer numeral *)
  | Str of string  (* push string literal *)
  | Seq of com list (* executable sequence *)
  | Pop  (* pop top value from stack *)
  | Swap (* swap top two values of stack *)
  | Sel  (* choose one of two values from stack *)
  | Get  (* push value at given stack index *)
  | Put  (* store top of stack at given stack index *)
  | Prs  (* print string *)
  | Pri  (* print integer *)
  | Exec (* execute sequence at top of stack *)
  | Add | Sub | Mul | Div | Rem (* arithmetic ops *)
  | LT | LE | EQ | NE | GE | GT (* relational ops *)
```

The concrete syntax of PostFix programs and commands has been designed so that they can be easily parsed and unparsed via s-expressions (see Handout #13). Converting between the concrete and abstract syntax of PostFix is left as an exercise.

2 Semantics

The meaning of a PostFix program is determined by executing its command sequence in left to right order. Each command manipulates an implicit stack of values that initially contains the integer arguments of the program (where the first argument is at the bottom of the stack and the

last arguments is at the top). A value on the stack is either (1) an integer (2) a string or (3) an executable sequence. The result of a program is the integer value at the top of the stack after its command sequence has been completely executed. A program signals an error (designated *error*) if (1) the final stack is empty, (2) the value at the top of the final stack is not an integer, or (3) an inappropriate stack of values is encountered when one of its commands is executed.

The behavior of PostFix commands is summarized in Fig. 1. Each command is specified in terms of how it manipulates the implicit stack.

The step-by-step execution of a PostFix program can be understood in terms of the operation of an **abstract machine**, where the **state** of the abstract machine is characterized by two components: (1) a sequence of commands to be executed and (2) a stack of values that initially contains the arguments to the program (in reverse order – i.e., the first argument value is at the bottom of the stack). On each execution step, the first command in the command sequence component is removed from the command sequence and executed; the result is an updated command sequence and stack. Like any iteration (remember CS111 and CS230?), this iterative execution process can be presented as a table in which each row contains the state variables of the iteration (in this case, the state of the abstract machine = commands and stack) and the rules of the iteration (in this case, the execution rules) are used to determine the next row of the table from the previous row. We shall call such a table the **trace** of an execution.

For example, consider a PostFix program that takes three arguments (call them a, b, and c) and calculates $2 \cdot a - b \cdot c$:

```
(postfix 3 mul swap 2 mul swap sub)
```

Note how swaps are used to reorder arguments on the stack. Here is a trace of this program run on the argument list `[3;4;5]` :

Commands	Stack

`mul`
`swap`
`2`
`mul`
`swap`
`sub`
	`5`
	`4`
	`3`

`swap`
`2`
`mul`
`swap`
`sub`
	`20`
	`3`

`2`
`mul`
`swap`
`sub`
	`3`
	`20`

`mul`
`swap`
`sub`
	`2`
	`3`
	`20`

`swap` `sub`	`6` `20`
`sub`	`20` `6`
	`-14`

- An integer numeral n : Push n onto the stack.
- A string literal s : Push s onto the stack.
- `sub` : Call the top stack value v_0 and the next-to-top stack value v_1. Pop these two values off the stack and push the result of $v_1 - v_0$ onto the stack. If there are fewer than two values on the stack or the top two values aren't both numerals, signal an error. The other binary arithmetic operators — `add` (addition), `mul` (multiplication), `div` (integer division[a]) and `rem` (remainder of integer division) — behave similarly. Both `div` and `rem` signal an error if v_0 is zero.
- `lt` : Call the top stack value v_0 and the next-to-top stack value v_1. Pop these two values off the stack. If $v_1 < v_0$, then push a `1` (a true value) on the stack, otherwise push a `0` (false). The other binary comparison operators — `le` (less than or equals), `eq` (equals), `ne` (not equals), `gt` (greater than), and `ge` (greater than or equals) — behave similarly. If there are fewer than two values on the stack or the top two values aren't both numerals, signal an error.
- `pop` : Pop the top element off the stack and discard it. Signal an error if the stack is empty.
- `swap` : Swap the top two elements of the stack. Signal an error if the stack has fewer than two values.
- `sel` : Call the top three stack values (from top down) v_0, v_1, and v_2. Pop these three values off the stack. If v_2 is the numeral `0`, push v_0 onto the stack; if v_2 is a non-zero numeral, push v_1 onto the stack. Signal an error if the stack does not contain three values, or if v_2 is not a numeral.
- `get` : Call the top stack value v_{index} and the remaining stack values (from top down) v_0, v_1, ... v_n. Pop v_{index} off the stack. If v_{index} is a numeral i such that $0 \leq i \leq n$, push v_i onto the stack. Signal an error if the stack does not contain at least one value, if v_{index} is not a numeral, or if i is not in the range $[0, n]$.
- `put` : Call the top stack value v_{index}, the next-to-top stack value v_{val}, the remaining stack values (from top down) v_0, v_1, ... v_n. Pop v_{index} and v_{val} off the stack. If v_{index} is a numeral i such that $0 \leq i \leq n$, replace the slot holding v_i on the stack by v_{val}. Signal an error if the stack does not contain at least two values, if v_{index} is not a numeral, or if i is not in the range $[0, n]$.
- `prs` : Call the top stack value v_s. Pop v_s off the stack. If v_s is the string s, display s in the terminal window. Signal an error if the stack does not contain at least one value or if v_s is not a string.
- `pri` : Call the top stack value v_i. Pop v_i off the stack. If v_i is the numeral i, display i in the terminal window. Signal an error if the stack does not contain at least one value or if v_i is not a numeral.
- `(`C_1 ... C_n`)` : Push the *executable sequence* `(`C_1 ... C_n`)` as a single value onto the stack. Executable sequences are used in conjunction with `exec`.
- `exec` : Pop the executable sequence from the top of the stack, and prepend its component commands onto the sequence of currently executing commands. Signal an error if the stack is empty or the top stack value isn't an executable sequence.

[a]The integer division of n and d returns the integer quotient q such that $n = qd + r$, where r (the remainder) is such that $0 \leq r < |d|$ if $n \geq 0$ and $-|d| < r \leq 0$ if $n < 0$.

Figure 1: Semantics of PostFix commands.

Often we do not care about the step-by-step details of a PostFix program execution, but only the final result. We use the notation $P \xrightarrow{args} v$ to mean that executing the PostFix program P on the arguments *args* returns the value v. The notation $P \xrightarrow{args} error$ means that executing the PostFix program P on the arguments signals an error. Errors are caused by inappropriate stack values or an insufficient number of stack values. In practice, it is desirable for an implementation to indicate the type of error. We will use comments (delimited by squiggly braces) to explain errors and other situations.

To illustrate the meanings of various commands, we show the results of some simple program executions. For example, numerals are pushed onto the stack, while `pop` and `swap` are the usual stack operations:

`(postfix 0  1 2 3)` $\xrightarrow{[]}$ `3` {*Only top stack value returned.*}
`(postfix 0  1 2 3 pop)` $\xrightarrow{[]}$ `2`
`(postfix 0  1 2 swap 3 pop)` $\xrightarrow{[]}$ `1`
`(postfix 1 swap)` $\xrightarrow{[5]}$ *error* {*Not enough values to swap.*}
`(postfix 1 pop)` $\xrightarrow{[17]}$ *error* {*Final stack empty.*}

Numerical operations are expressed in postfix notation, in which each operator comes after the commands that compute its operands. `add`, `sub`, `mul`, `div`, and `rem` are binary integer operators, while `lt`, `le`, `eq`, `ne`, `ge`, and `gt` are binary integer predicates returning either `1` (true) or `0` (false).

`(postfix 0 3 4 sub)` $\xrightarrow{[]}$ `-1`
`(postfix 0 3 4 add 5 mul 6 sub 7 div)` $\xrightarrow{[]}$ `4`
`(postfix 0 3 4 5 6 7 add mul sub swap div)` $\xrightarrow{[]}$ `-20`
`(postfix 0 1 20 300 4000 swap pop add)` $\xrightarrow{[]}$ `4020`
`(postfix 0 17 4 rem)` $\xrightarrow{[]}$ `1`
`(postfix 0 3 4 lt)` $\xrightarrow{[]}$ `1`
`(postfix 0 3 4 gt)` $\xrightarrow{[]}$ `0`
`(postfix 0 3 4 lt 10 add )` $\xrightarrow{[]}$ `11`
`(postfix 1 4 mul add)` $\xrightarrow{[3]}$ *error* `&` {*Not enough numbers to add.*}
`(postfix 2 4 sub div)` $\xrightarrow{[3;4]}$ *error* `&` {*Divide by 0.*}

Program arguments are pushed onto the stack before the execution of the program commands. It is an error if the actual number of arguments does not match the number specified in the program. Program arguments must be integers; they cannot be strings or executable sequences.

`(postfix 1 2 sub)` $\xrightarrow{[5]}$ `3`
`(postfix 2 sub)` $\xrightarrow{[5;4]}$ `1`
`(postfix 2)` $\xrightarrow{[5;4]}$ `4`
`(postfix 2 sub)` $\xrightarrow{[5]}$ *error*
`(postfix 2 sub)` $\xrightarrow{[5;4;3]}$ *error*

In all the examples seen so far, each stack value is used at most once. Sometimes it is necessary to use a value two or more times. The `get` command is necessary in these situations; it puts at the top of the stack a copy of a stack value at a specified index. The index is 0-based, from the top of the stack down, not counting the index value itself. It is an error to use an index that is out of bounds. Note that index of a given value increases every time a new value is pushed on the stack. In addition to copying a value so that it can be used more than once, `get` is also helpful for accessing values that are not near the top of the stack.

```
(postfix 2 0 get) [5;6]⟶ 6
(postfix 2 1 get) [5;6]⟶ 5
(postfix 2 2 get) [5;6]⟶ error {Too large an index}
(postfix 2 -1 get) [5;6]⟶ error {Too small an index}
(postfix 1 0 get mul) [5]⟶ 25 {A squaring program}
(postfix 4 {Given args a, b, c, x, calculates ax^2 + bx + c.}
           0 get {get x}
           1 get {get x again, at new index}
           mul {push x^2 on stack}
           4 get mul {multiply x^2 by a}
           swap {swap ax^2 with x}
           3 get mul {multiply x by b}
           add {ax^2 + bx}
           add {ax^2 + bx + c}
           ) [3;4;5;10]⟶ 345
```

Whereas `get` copies a value from a specified index on the stack, the `put` command stores a new value into a specified index on the stack. It can be used to model the mutable variables and cells of other languages in PostFix.

```
(postfix 2 {Given args a and b, calculate 2a - b}
           1 get {Push a}
           2 mul {Replace a by 2a on top of stack}
           1 put {Replace a by 2a on bottom of stack}
           sub {Calculate 2a - b}
           ) [5;6]⟶ 4
```

Executable sequences are compound commands like `(2 mul)` that are pushed onto the stack as a single value. They can be executed later by the `exec` command. Executable sequences act like subroutines in other languages. Execution of an executable sequence is similar to a subroutine call, except that transmission of arguments and results is accomplished via the stack.

```
(postfix 1 (2 mul) exec) [7]⟶ 14  {(2 mul) is a doubling subroutine.}
(postfix 0 (0 swap sub) 7 swap exec) []⟶ -7
(postfix 0 (7 swap exec) (0 swap sub) swap exec) []⟶ -7
(postfix 0 (2 mul)) []⟶ error {Top of stack not integer}
(postfix 0 3 (2 mul) gt) []⟶ error {Executable sequence where number expected.}
(postfix 0 3 exec) []⟶ error {Number where executable sequence expected.}
(postfix 1 ((2 get swap exec) (3 mul swap exec) swap)
           (5 sub) swap exec exec) [4]⟶ 7
```

The last example illustrates that evaluations involving executable sequences can be rather contorted. Fig. 2 shows a trace for this example. Convince yourself that for an argument n, the program in the last example calculates $3n - 5$ (though not in a straightforward way!).

Commands	Stack
((2 get swap exec) (3 mul swap exec) swap) (5 sub) swap exec exec	4
(5 sub) swap exec exec	((2 get swap exec) (3 mul swap exec) swap) 4
swap exec exec	(5 sub) ((2 get swap exec) (3 mul swap exec) swap) 4
exec exec	((2 get swap exec) (3 mul swap exec) swap) (5 sub) 4
(2 get swap exec) (3 mul swap exec) swap exec	(5 sub) 4
(3 mul swap exec) swap exec	(2 get swap exec) (5 sub) 4
swap exec	(3 mul swap exec) (2 get swap exec) (5 sub) 4
exec	(2 get swap exec) (3 mul swap exec) (5 sub) 4
2 get swap exec	(3 mul swap exec) (5 sub) 4
get swap exec	2 (3 mul swap exec) (5 sub) 4
swap exec	4 (3 mul swap exec) (5 sub) 4
exec	(3 mul swap exec) 4 (5 sub) 4
3 mul swap exec	4 (5 sub) 4
mul swap exec	3 4 (5 sub) 4
swap exec	12 (5 sub) 4
exec	(5 sub) 12 4
5 sub	12 4
sub	5 12 4
	7 4

Figure 2: Step-by-step execution of a contorted `exec` example.

The `sel` command selects between two values. It can be used in conjunction with `exec` to conditionally execute one of two executable sequences.

`(postfix 1 2 3 sel)` $\xrightarrow{\texttt{[1]}}$ `2`

`(postfix 1 2 3 sel)` $\xrightarrow{\texttt{[0]}}$ `3`

`(postfix 1 2 3 sel)` $\xrightarrow{\texttt{[17]}}$ `2` {*Any non-zero value is "true"*}

`(postfix 4 lt (add) (mul) sel exec)` $\xrightarrow{\texttt{[3;4;5;6]}}$ `7`

`(postfix 4 lt (add) (mul) sel exec)` $\xrightarrow{\texttt{[3;4;6;5]}}$ `12`

PostFix's combination of `exec`, `sel`, and `get/put` is very powerful, since it permits the expression of arbitrary iterations and recursions. For example, here is an iterative factorial program written in PostFix:

```
(postfix 1 ({factorial loop code}
            1 get 0 eq {is n = 0?}
            () {if yes, we're done; ans is on top of stack}
            (1 get mul          {if no, update state vars: ans ← n*ans}
             1 get 1 sub 1 put {and n ← n-1;}
             2 get exec)        {then execute loop again}
            sel exec)
           swap {swap n with factorial loop code}
           1 {push initial answer = 1}
           2 get exec {execute loop}
           )
```

To see how this works, consider the following execution trace for calculating the factorial of 3. To simplify the trace, many steps have been omitted, and the executable sequence annotated {*factorial loop code*} has been abbreviated as *FactIter* A key invariant maintained during the loop is that the bottom three stack values are, from top to bottom, (1) the current answer (i.e., the running product of numbers n processed so far); (2) the current value of n; and (3) the executable sequence *FactIter*.

Commands	**Stack**
FactIter `swap 1 2 get exec`	`3`
`swap 1 2 get exec`	*FactIter* `3`
`1 2 get exec`	`3` *FactIter*
`2 get exec`	`1 3` *FactIter*
`get exec`	`2 1 3` *FactIter*
`exec`	*FactIter* `1 3` *FactIter*
`1 get 0 eq () (1 get mul 1 get 1 sub 1 put 2 get exec) sel exec)`	`1 3` *FactIter*
... many steps omitted ...	
`1 get 0 eq () (1 get mul 1 get 1 sub 1 put 2 get exec) sel exec)`	`3 2` *FactIter*
... many steps omitted ...	
`1 get 0 eq () (1 get mul 1 get 1 sub 1 put 2 get exec) sel exec)`	`6 1` *FactIter*
... many steps omitted ...	
`1 get 0 eq () (1 get mul 1 get 1 sub 1 put 2 get exec) sel exec)`	`6 0` *FactIter*
... many steps omitted ...	
	`6 0` *FactIter*

It is even possible to calculate factorial recursively in PostFix, as is demonstrated by the following program:

```
(postfix 1 ({factorial recursion code}
            1 get 0 eq {is n = 0?}
            (pop pop 1) {if yes, return 1}
            (1 get 1 sub {push n-1}
             1 get 0 get exec {call fact recursively}
             swap pop         {delete fact code}
             mul              {multiply on way out}
            )
            sel exec)
           0 get exec {call fact initially}
           )
```

To see how it works, consider an execution trace on the argument 3 (Fig. 3). We use the name *FactRec* to abbreviate the executable sequence annotated {*factorial recursion code*}, and we omit many intermediate steps. On the way into the recursion, each integer from the intial argument value n down to 0 is pushed onto the stack along with a copy of the *FactRec* sequence. Additionally, the "pending code" `swap pop mul` is added to the command sequence. This pending code, which multiplies the numbers on the stack,k is performed on the way out of the recursion.

It turns out that PostFix is as powerful as any programming language can be. It is **Turing complete**, which means that it can express any computable function. Any Turing complete language is not only powerful, but also dangerous, in the sense that it is possible to write "infinite loops" that never return. Here is a simple infinite loop program:

`(postfix 0 (0 get exec) 0 get exec)` $\xrightarrow{[]}$ {*Never returns from infinite loop!*}

To see why this is a infinite loop, consider its execution trace:

Commands	**Stack**
`(0 get exec) 0 get exec`	
`0 get exec`	`(0 get exec)`
`get exec`	`0 (0 get exec)`
`exec`	`(0 get exec) (0 get exec)`
`0 get exec`	`(0 get exec)`
Hey! We've been here before, and will be here again infinitely many times!	

The only commands we haven't seen in action yet are those involving strings. This aren't strictly necessary, but are nice to have for debugging purposes. For example:

```
(postfix 2
  "\nAdding " prs {Display "Adding " after a newline}
  1 get pri       {Display 1st arg}
  " and " prs     {Display " and "}
  0 get pri       {Display 2nd arg}
  "\n" prs        {Display newline}
  add             {Return sum of args}
  ) [3;7] 10 {Returns 10, after displaying "Adding 3 and 7" in the console}
```

Commands	**Stack**
FactRec `0 get exec`	`3`
`0 get exec`	*FactRec* `3`

`get exec`	`0`
	FactRec
	`3`

`exec`	*FactRec* *FactRec* `3`
... many steps omitted ...	

`1 get 0 eq (pop pop 1)`
`(1 get 1 sub 1 get 0 get exec swap pop mul)`
`sel exec`
	FactRec
	`3`

... many steps omitted ...	

`1 get 0 eq (pop pop 1)`
`(1 get 1 sub 1 get 0 get exec swap pop mul)`
`sel exec`
`swap pop mul`
	FactRec
	`2`
	FactRec
	`3`

... many steps omitted ...	

`1 get 0 eq (pop pop 1)`
`(1 get 1 sub 1 get 0 get exec swap pop mul)`
`sel exec`
`swap pop mul`
`swap pop mul`
	FactRec
	`1`
	FactRec
	`2`
	FactRec
	`3`

... many steps omitted ...	

`1 get 0 eq (pop pop 1)`
`(1 get 1 sub 1 get 0 get exec swap pop mul)`
`sel exec`
`swap pop mul`
`swap pop mul`
`swap pop mul`
	FactRec
	`0`
	FactRec
	`1`
	FactRec
	`2`
	FactRec
	`3`

... many steps omitted ...	

`pop pop 1`
`swap pop mul`
`swap pop mul`
`swap pop mul`
	FactRec
	`0`
	FactRec
	`1`
	FactRec
	`2`
	FactRec
	`3`

... many steps omitted ...	

`swap pop mul`
`swap pop mul`
`swap pop mul`
	`1`
	FactRec
	`1`
	FactRec
	`2`
	FactRec
	`3`

... many steps omitted ...	

`swap pop mul`
`swap pop mul`
	`1`
	FactRec
	`2`
	FactRec
	`3`

... many steps omitted ...	

`swap pop mul`	`2`
	FactRec
	`3`

... many steps omitted ...	
	`6`

Figure 3: Trace of the recursive factorial program on the input 3.

As a more compelling example of using strings in a program, consider annotating the iterative factorial program with code that prints the values of n and ans at the beginning of each iteration of the loop:

```
(postfix 1 (1 get 0 eq
             ("\n" prs)
             ("\n  n=" prs 1 get pri "; ans=" prs 0 get pri
              1 get mul 1 get 1 sub 1 put 2 get exec)
             sel exec)
            swap
            1 2 get exec)
```

For example, running the above program on 5 returns 120 after displaying the following text in the console window:

```
n=5; ans=1
n=4; ans=5
n=3; ans=20
n=2; ans=60
n=1; ans=120
```