

1.4 PostFix: A Simple Stack Language

We will introduce the tools for syntax, semantics, and pragmatics in the context of a mini-language called `POSTFIX`. `POSTFIX` is a simple stack-based language inspired by the `POSTSCRIPT` graphics language, the `FORTH` programming language, and Hewlett Packard calculators. Here we give an informal introduction to `POSTFIX` in order to build some intuitions about the language. In subsequent chapters, we will introduce tools that allow us to study `POSTFIX` in more depth.

1.4.1 Syntax

The basic syntactic unit of a `POSTFIX` program is the **command**. Commands are of the following form:

- Any integer numeral. E.g., 17, 0, -3.
- One of the following special command tokens: `add`, `div`, `eq`, `exec`, `gt`, `lt`, `mul`, `nget`, `pop`, `rem`, `sel`, `sub`, `swap`.
- An **executable sequence** — a single command that serves as a subroutine. It is written as a parenthesized list of subcommands separated by whitespace (any contiguous sequence of characters that leave no mark on the page, such as spaces, tabs, and newlines). E.g., `(7 add 3 swap)` or `(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.

A `POSTFIX` **program** is a parenthesized sequence consisting of (1) the token `postfix` followed by (2) a natural number (i.e., nonnegative integer) indicating the number of program parameters followed by (3) zero or more `POSTFIX` commands. Here are some sample `POSTFIX` programs:

```
(postfix 0 4 7 sub)
(postfix 2 add 2 div)
(postfix 4 4 nget 5 nget mul mul swap 4 nget mul add add)
(postfix 1 ((3 nget swap exec) (2 mul swap exec) swap)
(5 sub) swap exec exec)
```

In `POSTFIX`, as in all the languages we'll be studying, all parentheses are required and none are optional. Moving parentheses around changes the structure of the program and most likely changes its behavior. Thus, while the following

POSTFIX executable sequences use the same numerals and command tokens in the same order, they are distinguished by their parenthesization, which, as we shall see below, makes them behave differently.

```
((1) (2 3 4) swap exec)
((1 2) (3 4) swap exec)
((1 2) (3 4 swap) exec)
```

1.4.2 Semantics

The meaning of a POSTFIX program is determined by executing its commands 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 top of the stack and the last argument is at the bottom). A value on the stack is either (1) an integer numeral or (2) 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 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 Figure 1.1. Each command is specified in terms of how it manipulates the implicit stack. We use the notation $P \xrightarrow{args} v$ to mean that executing the POSTFIX program P on the integer argument sequence $args$ returns the value v . The notation $P \xrightarrow{args} \mathbf{error}$ means that executing the POSTFIX program P on the arguments $args$ 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 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)  $\Downarrow$  3 {Only the top stack value is returned.}
(postfix 0 1 2 3 pop)  $\Downarrow$  2
(postfix 0 1 2 swap 3 pop)  $\Downarrow$  1
(postfix 0 1 swap)  $\Downarrow$  error {Not enough values to swap.}
(postfix 0 1 pop pop)  $\Downarrow$  error {Empty stack on second pop.}
```

Program arguments are pushed onto the stack (from last to first) before the execution of the program commands.

N: Push the numeral N onto the stack.

sub: Call the top stack value v_1 and the next-to-top stack value v_2 . Pop these two values off the stack and push the result of $v_2 - v_1$ 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_1 is zero.

lt: Call the top stack value v_1 and the next-to-top stack value v_2 . Pop these two values off the stack. If $v_2 < v_1$, then push a 1 (a true value) on the stack, otherwise push a 0 (false). The other binary comparison operators — **eq** (equals) and **gt** (greater than) — 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_1 , v_2 , and v_3 . Pop these three values off the stack. If v_3 is the numeral 0, push v_1 onto the stack; if v_3 is a nonzero numeral, push v_2 onto the stack. Signal an error if the stack does not contain three values, or if v_3 is not a numeral.

nget: Call the top stack value v_{index} and the remaining stack values (from top down) v_1, v_2, \dots, v_n . Pop v_{index} off the stack. If v_{index} is a numeral i such that $1 \leq i \leq n$ and v_i is a numeral, 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, if i is not in the range $[1..n]$, or if v_i is not a numeral.

$(C_1 \dots C_n)$: Push the *executable sequence* $(C_1 \dots C_n)$ 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.

^aThe 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 1.1 English semantics of POSTFIX commands.

(postfix 2) $\xrightarrow{[3,4]}$ 3 {Initial stack has 3 on top with 4 below.}
 (postfix 2 swap) $\xrightarrow{[3,4]}$ 4
 (postfix 3 pop swap) $\xrightarrow{[3,4,5]}$ 5

It is an error if the actual number of arguments does not match the number of parameters specified in the program.

```
(postfix 2 swap)  $\xrightarrow{[3]}$  error {Wrong number of arguments.}
(postfix 1 pop)  $\xrightarrow{[4,5]}$  error {Wrong number of arguments.}
```

Note that program arguments must be integers — they cannot be executable sequences.

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

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

In all the above examples, each stack value is used at most once. Sometimes it is desirable to use a number two or more times or to access a number that is not near the top of the stack. The `nget` command is useful in these situations; it puts at the top of the stack a copy of a number located on the stack at a specified index. The index is 1-based, from the top of the stack down, not counting the index value itself.

```
(postfix 2 1 nget)  $\xrightarrow{[4,5]}$  4 {4 is at index 1, 5 at index 2.}
(postfix 2 2 nget)  $\xrightarrow{[4,5]}$  5
```

It is an error to use an index that is out of bounds or to access a nonnumeric stack value (i.e., an executable sequence) with `nget`.

```
(postfix 2 3 nget)  $\xrightarrow{[4,5]}$  error {Index 3 is too large.}
(postfix 2 0 nget)  $\xrightarrow{[4,5]}$  error {Index 0 is too small.}
(postfix 1 (2 mul) 1 nget)  $\xrightarrow{[3]}$  error
{Value at index 1 is not a number but an executable sequence.}
```

The `nget` command is particularly useful for numerical programs, where it is common to reference arbitrary parameter values and use them multiple times.

```
(postfix 1 1 nget mul)  $\xrightarrow{[5]}$  25 {A squaring program.}
(postfix 4 4 nget 5 nget mul mul swap 4 nget mul add add)  $\xrightarrow{[3,4,5,2]}$  25
{Given a, b, c, x, calculates  $ax^2 + bx + c$ .}
```

As illustrated in the last example, the index of a given value increases every time a new value is pushed onto the stack. The final stack in this example contains (from top down) 25 and 2, showing that the program may end with more than one value on the stack.

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)  $\xrightarrow{[7]}$  14 {(2 mul) is a doubling subroutine.}
(postfix 0 (0 swap sub) 7 swap exec)  $\xrightarrow{[]}$  -7
{(0 swap sub) is a negation subroutine.}
(postfix 0 (2 mul))  $\xrightarrow{[]}$  error {Final top of stack is not an integer.}
(postfix 0 3 (2 mul) gt)  $\xrightarrow{[]}$  error
{Executable sequence where number expected.}
(postfix 0 3 exec)  $\xrightarrow{[]}$  error {Number where executable sequence expected.}
(postfix 0 (7 swap exec) (0 swap sub) swap exec)  $\xrightarrow{[]}$  -7
(postfix 2 (mul sub) (1 nget mul) 4 nget swap exec swap exec)
 $\xrightarrow{[-10,2]}$  42 {Given a and b, calculates  $b - a \cdot b^2$ .}
```

The last two examples illustrate that evaluations involving executable sequences can be rather contorted.

The `sel` command selects between two values based on a test value, where zero is treated as false and any nonzero integer is treated as true. It can be used in conjunction with `exec` to conditionally execute one of two executable sequences.

```
(postfix 1 2 3 sel)  $\xrightarrow{[1]}$  2
(postfix 1 2 3 sel)  $\xrightarrow{[0]}$  3
(postfix 1 2 3 sel)  $\xrightarrow{[17]}$  2 {Any nonzero number is "true."}
(postfix 0 (2 mul) 3 4 sel)  $\xrightarrow{[]}$  error {Test not a number.}
(postfix 4 lt (add) (mul) sel exec)  $\xrightarrow{[3,4,5,6]}$  30
(postfix 4 lt (add) (mul) sel exec)  $\xrightarrow{[4,3,5,6]}$  11
(postfix 1 1 nget 0 lt (0 swap sub) () sel exec)  $\xrightarrow{[-7]}$  7
{An absolute value program.}
(postfix 1 1 nget 0 lt (0 swap sub) () sel exec)  $\xrightarrow{[6]}$  6
```

Exercise 1.1 Determine the value of the following POSTFIX programs on an empty stack.

- a. `(postfix 0 10 (swap 2 mul sub) 1 swap exec)`
- b. `(postfix 0 (5 (2 mul) exec) 3 swap)`
- c. `(postfix 0 (() exec) exec)`
- d. `(postfix 0 2 3 1 add mul sel)`
- e. `(postfix 0 2 3 1 (add) (mul) sel)`
- f. `(postfix 0 2 3 1 (add) (mul) sel exec)`
- g. `(postfix 0 0 (2 3 add) 4 sel exec)`
- h. `(postfix 0 1 (2 3 add) 4 sel exec)`
- i. `(postfix 0 (5 6 lt) (2 3 add) 4 sel exec)`
- j. `(postfix 0 (swap exec swap exec) (1 sub) swap (2 mul)
swap 3 swap exec)`

Exercise 1.2

- a. What function of its argument does the following POSTFIX program calculate?

```
(postfix 1 ((3 nget swap exec) (2 mul swap exec) swap)
(5 sub) swap exec exec)
```
- b. Write a simpler POSTFIX program that performs the same calculation.

Exercise 1.3 Recall that executable sequences are effectively subroutines that, when invoked (by the `exec` command), take their arguments from the top of the stack. Write executable sequences that compute the following logical operations. Recall that 0 stands for false and all other numerals are treated as true.

- a. *not*: return the logical negation of a single argument.
- b. *and*: given two numeric arguments, return 1 if their logical conjunction is true, and 0 otherwise.
- c. *short-circuit-and*: return 0 if the first argument is false; otherwise return the second argument.
- d. Demonstrate the difference between *and* and *short-circuit-and* by writing a POSTFIX program with zero arguments that has a different result if *and* is replaced by *short-circuit-and*.

Exercise 1.4

- a. Without `nget`, is it possible to write a POSTFIX program that squares its single argument? If so, write it; if not, explain.

- b. Is it possible to write a POSTFIX program that takes three integers and returns the smallest of the three? If so, write it; if not, explain.
- c. Is it possible to write a POSTFIX program that calculates the factorial of its single argument (assume it's nonnegative)? If so, write it; if not, explain.

1.4.3 The Pitfalls of Informal Descriptions

The “by-example” and English descriptions of POSTFIX given above are typical of the way that programming languages are described in manuals, textbooks, courses, and conversations. That is, a syntax for the language is presented, and the semantics of each of the language constructs is specified using English prose and examples. The utility of this method for specifying semantics is apparent from the fact that the vast majority of programmers learn to read and write programs via this approach.

But there are many situations in which informal descriptions of programming languages are inadequate. Suppose that we want to improve a program by transforming complex phrases into phrases that are simpler and more efficient. How can we be sure that the transformation process preserves the meaning of the program?

Or suppose that we want to prove that the language as a whole has a particular property. For instance, it turns out that every POSTFIX program is guaranteed to terminate (i.e., a POSTFIX program cannot enter an infinite loop). How would we go about proving this property based on the informal description? Natural language does not provide any rigorous framework for reasoning about programs or programming languages. Without the aid of some formal reasoning tools, we can only give hand-waving arguments that are not likely to be very convincing.

Or suppose that we wish to extend POSTFIX with features that make it easier to use. For example, it would be nice to name values, to collect values into arrays, to query the user for input, and to loop over sequences of values. With each new feature, the specification of the language becomes more complex, and it becomes more difficult to reason about the interaction between various features. We'd like techniques that help to highlight which features are orthogonal and which can interact in subtle ways.

Or suppose that a software vendor wants to develop POSTFIX into a product that runs on several different machines. The vendor wants any given POSTFIX program to have exactly the same behavior on all of the supported machines. But how do the development teams for the different machines guarantee that they're all implementing the “same” language? If there are any ambiguities in the POSTFIX specification that they're implementing, different development

teams might resolve the ambiguity in incompatible ways. What's needed in this case is an unambiguous specification of the language as well as a means of proving that an implementation meets that specification.

The problem with informal descriptions of a programming language is that they're neither concise nor precise enough for these kinds of situations. English is often verbose, and even relatively simple ideas can be unduly complicated to explain. Moreover, it's easy for the writer of an informal specification to underspecify a language by forgetting to cover all the special cases (e.g., error situations in `POSTFIX`). It isn't that covering all the special cases is impossible; it's just that the natural-language framework doesn't help much in pointing out what the special cases are.

It is possible to overspecify a language in English as well. Consider the `POSTFIX` programming model introduced above. The current state of a program is captured in two entities: the stack and the current command sequence. To programmers and implementers alike, this might imply that a language implementation *must* have explicit stack and command sequence elements in it. Although these would indeed appear in a straightforward implementation, they are not in any way *required*; there are alternative models and implementations for `POSTFIX` (e.g., see Exercise 3.12 on page 70). It would be desirable to have a more abstract definition of what constitutes a legal `POSTFIX` implementation so that a would-be implementer could be sure that an implementation was faithful to the language definition regardless of the representations and algorithms employed.

1.5 Overview of the Book

The remainder of Part I introduces a number of tools that address the inadequacies outlined above and that form an essential foundation for the study of programming language design. Chapter 2 presents **s-expression grammars**, a simple specification for syntax that we will use to describe the structure of all of the mini-languages we will explore. Then, using `POSTFIX` and a simple expression language as our objects of study, we introduce two approaches to formal semantics:

- An **operational semantics** (Chapter 3) explains the meaning of programming language constructs in terms of the step-by-step process of an abstract machine.
- A **denotational semantics** (Chapter 4) explains the meaning of programming language constructs in terms of the meaning of their subparts.