8 Stacks and Recursion

8.1 Introduction

Before moving on from discussing dispensers to discussing collections, we must discuss the strong connection between stacks and recursion. Recall that recursion involves operations that call themselves.

> **Recursive operation:** An operation that either calls itself directly, or calls other operations that call it.

Recursion and stacks are intimately related in the following ways:

- Every recursive operation (or group of mutually recursive operations) can be rewritten without recursion using a stack.
- Every algorithm that uses a stack can be rewritten without a stack using one or more recursive operations.

To establish the first point, note that computers do not support recursion at the machine level—most processors can move data around, do a few simple arithmetic and logical operations, and compare values and branch to different points in a program based on the result, but that is all. Yet many programming language support recursion. How is this possible? At runtime, compiled programs use a stack that stores data about the current state of execution of a sub-program, called an *activation record*. When a sub-program is called, a new activation record is pushed on the stack to hold the sub-program's arguments, local variables, return address, and other book-keeping information. The activation record stays on the stack until the sub-program returns, when it is popped off the stack. Because every call of a sub-program causes a new activation record to be pushed on the stack, this mechanism supports recursion: every recursive call of a sub-program has its own activation record, so the data associated with a particular sub-program call is not confused with that of other calls of the sub-program. Thus the recursive calls can be "unwound" onto the stack, and a non-recursive machine can implement recursion.

The second point is not quite so easy to establish, but the argument goes like this: when an algorithm would push data on a stack, a recursive operation can preserve the data that would go on the stack in local variables and then call itself to continue processing. The recursive call returns just when it is time to pop the data off the stack, so processing can continue with the data in the local variables just as it would if the data had been popped off the stack.

In some sense, then, recursion and stacks are equivalent. It turns out that some algorithms are easier to write with recursion, some are easier to write with stacks, and some are just as easy (or hard) one way as the other. But in any case, there is always a way to write algorithms either entirely recursively without any stacks, or without any recursion using stacks.

In the remainder of this chapter we will illustrate the theme of the equivalence of stacks and recursion by considering a few examples of algorithms that need either stacks or recursion, and we will look at how to implement these algorithms both ways.

8.2 Balanced Brackets

Because of its simplicity, we begin with an example that doesn't really need a stack or recursion to solve, but illustrates how both can be used with equal facility: determining whether a string of brackets is balanced or not. The strings of balanced brackets are defined as follows:

1. The empty string and the string "[]" are both strings of balanced brackets.
2. If A is a string of balanced brackets, then so it "["A "]".
3. If A and B are strings of balanced brackets, then so is AB.

So, for example, [[[][]][]] is a string of balanced brackets, but [[[]][]][]] is not.

The recursive algorithm in Figure 1, written in Ruby, checks whether a string of brackets is balanced.

```
def recursive_balanced?(string)
  return true if string.empty?
  source = StringEnumerator.new(string)
  check_balanced?(source) && source.empty?
end

def check_balanced?(source)
  return false unless source.current == '['
  loop do
    if source.next == '['
     return false if !check_balanced?(source)
    end
    return false unless source.current == ']'
    break if source.next != '['
  end
  true
end
```

Figure 1: Recursive Algorithm For Checking String of Balanced Brackets

The `recursive_balanced?()` operation has a string argument containing only brackets. It does some initial and final processing, but most of the work is done by the recursive helper function `check_balanced?()`. Note that a `StringEnumerator` is created from the string argument and passed to the `check_balanced?()` operation. We will discuss enumerators later, but for now it suffices to say that a `StringEnumerator` is a class that provides characters from a string one by one when asked for them. It also indicates when all the characters have been provided, signalling the end of the string.

The algorithm is based on the recursive definition of balanced brackets. If the string is empty, then it is a string of balanced brackets. This is checked right away by `recursive_balanced?()`. If the string is not empty, then `check_balanced?()` is called to check the string. It first checks the current character to see whether it is a left bracket, and returns false if it is not. It then considers the next character. If it is another left bracket, then there is a nested string of balanced brackets, which is checked by a recursive call. In any case, a check is then made for the right bracket matching the initial left bracket, which takes care of the other basis case in the recursive definition. The loop is present to handle the case of a sequence of balanced brackets, as allowed by the recursive definition. Finally, when `check_balanced?()` returns its result to `recursive_balanced?()`, the latter checks to make sure that the string has all been consumed, which guarantees that there are no stray brackets at the end of the string.

This same job could be done just as well with a non-recursive algorithm using a stack. In the code in Figure 2 below, again written in Ruby, a stack is used to hold left brackets as they are encountered. If a right bracket is found for every left bracket on the stack, then the string of brackets is balanced. Note that the stack must be checked to make sure it is not empty as we go along (which would mean too many right brackets), and that it is empty when the entire string is processed (which would mean too many left brackets).

```
def stack_balanced?(string)
  stack = LinkedStack.new
  string.chars do | ch |
    case
    when ch == '['
      stack.push(ch)
    when ch == ']'
      return false if stack.empty?
      stack.pop
    else
      return false
    end
  end
  stack.empty?
end
```

Figure 2: Non-Recursive Algorithm For Checking Strings of Balanced Brackets

In this case the recursive algorithm is about as complicated as the stack-based algorithm. In the examples below, we will see that sometimes the recursive algorithm is simpler, and sometimes the stack-based algorithm is simpler, depending on the problem.

8.3 Infix, Prefix, and Postfix Expressions

The arithmetic expressions we learned in grade school are infix expressions, but other kinds of expressions, called prefix or postfix expressions, might also be used.

Infix expression: An expression in which the operators appear between their operands.

Prefix expression: An expression in which the operators appear before their operands.

Postfix expression: An expression in which the operators appear after their operands.

In a prefix expression, the operands of an operator appear immediately to its right, while in a postfix expression, they appear immediately to its left. For example, the infix expression (4 + 5) * 9 can be rewritten in prefix form as * + 4 5 9 and in postfix form as 4 5 + 9 *. An advantage of pre- and postfix expressions over infix expressions is that the latter don't need parentheses.

Many students are confused by prefix and postfix expressions the first time they encounter them, so lets consider a few more examples. In the expressions in the table below, all numbers are one digit long and the operators are all binary. All the expressions on a row are equivalent.

Infix	Prefix	Postfix
(2 + 8) * (7 % 3)	* + 2 8 % 7 3	2 8 + 7 3 % *
((2 * 3) + 5) % 4	% + * 2 3 5 4	2 3 * 5 + 4 %
((2 * 5) % (6 / 4)) + (2 * 3)	+ % * 2 5 / 6 4 * 2 3	2 5 * 6 4 / % 2 3 * +
1 + (2 + (3 + 4))	+ 1 + 2 + 3 4	1 2 3 4 + + +
((1 + 2) + 3) + 4	+ + + 1 2 3 4	1 2 + 3 + 4 +

Note that all the expressions have the digits in the same order. This is necessary because order matters for the subtraction and division operators. Also notice that the order of the operators in a prefix expression is not necessarily the reverse of its order in a postfix expression; sometimes operators are in the opposite order in these expressions, but not always. The systematic relationship between the operators is that the main operator always appears within the fewest number of parentheses in the infix expression, is first in the prefix expression, and is last in the postfix expression. Finally, in every expression, the number of constant arguments (digits) is always one more than the number of operators.

Let's consider the problem of evaluating prefix and postfix expressions. It turns out that sometimes it is much easier to write a recursive evaluation algorithm, and sometimes it is much easier to write a stack-based algorithm. In particular,

- It is very easy to write a recursive prefix expression evaluation algorithm, but somewhat harder to write this algorithm with a stack.
- It is very easy to write a stack-based postfix expression evaluation algorithm, but very hard to write this algorithm recursively.

To establish these claims, we will consider a few of the algorithms. An algorithm in Ruby to evaluate prefix expressions recursively appears in Figure 3 below. The main operation `recursive_eval_prefix()` accepts a string as an argument. Its job is to create a `StringEnumeration` object to pass along to the recursive helper function, and to make sure that the string has all been read (if not, then there are extra characters at the end of the expression). The real work is done by the `eval_prefix()` operation, which is recursive.

It helps to consider the recursive definition of a prefix expression to understand this algorithm:

> A prefix expression is either a digit, or if *A* and *B* are prefix expressions and *op* is an operator, then an expression of the form *op A B*.

The `eval_prefix()` operation first checks to see whether the string is exhausted and throws an exception if it is (because the empty string is not a prefix expression). Otherwise, it fetches the current character and advances to the next character to prepare for later processing. If the current character is a digit, this is the basis case of the recursive definition of a prefix expression, so it simply returns the integer value of the digit. Otherwise, the current character is an operator. According to the recursive definition, the operator should be followed by two prefix expressions, so the algorithm applies this operator to the result of recursively evaluating the following left and right arguments. If these arguments are not there, or are ill-formed, then one of these recursive calls will throw an exception that is propagated to the caller. The `evaluate()` operation is a helper function that simply applies the operation indicated in its `op` argument to its `left_arg` and `right_arg` values.

```
def recursive_eval_prefix(string)
  source = StringEnumerator.new(string)
  result = eval_prefix(source)
  raise "Too many arguments" unless source.empty?
  result
end

def eval_prefix(source)
  raise "Missing argument" if source.empty?
  ch = source.current
  source.next
  if ch =~ /\d/
    return ch.to_i
  else
    left_arg = eval_prefix(source)
    right_arg = eval_prefix(source)
    return evaluate(ch,left_arg, right_arg)
  end
end
```

Figure 3: Recursive Algorithm to Evaluate Prefix Expressions

This recursive algorithm is extremely simple, yet is does a potentially very complicated job. In contrast, algorithms to evaluate prefix expressions using a stack are quite a bit more complicated. One such an algorithm is shown below in Figure 4. This algorithm has two stacks: one for (integer) left arguments, and one for (character) operators.

```
def stack_eval_prefix(string)
  raise "Bad characters" if string =~ INVALID_CHARACTERS
  raise "Missing expression" if string == nil || string.empty?
  op_stack = LinkedStack.new
  val_stack = LinkedStack.new
  string.chars do | ch |
    case
    when ch =~ OPERATORS
      op_stack.push(ch)
    when ch =~ /\d/
      right_arg = ch.to_i
      loop do
        break if op_stack.empty? || op_stack.top != 'v'
        op_stack.pop
        raise "Missing operator" if op_stack.empty?
        right_arg = evaluate(op_stack.pop,val_stack.pop,right_arg)
      end
      op_stack.push('v')
      val_stack.push(right_arg)
    end
  end
  raise "Missing argument" if op_stack.empty?
  op_stack.pop
  raise "Missing expression" if val_stack.empty?
  result = val_stack.pop
  raise "Too many arguments" unless val_stack.empty?
  raise "Missing argument" unless op_stack.empty?
  result
end
```

Figure 4: Stack-Based Algorithm to Evaluate Prefix Expressions

The strategy of this algorithm is to process each character from the string in turn, pushing operators on the operator stack as they are encountered and values on the value stack as necessary to preserve left arguments. The placement of values relative to arguments is noted in the operator stack with a special `v` marker. An operator is applied as soon as two arguments are available. Results are pushed on the value stack and marked in the operator stack. Once the string is exhausted, the value stack should hold a single (result) value and the operator stack should hold a single `v` marker—if not, then there are either too many or too few arguments.

Clearly, this stack-based evaluation algorithm is more complicated than the recursive evaluation algorithm. In contrast, a stack-based evaluation algorithm for postfix expressions is quite simple, while a recursive algorithm is quite complicated. To illustrate, consider the stack-based postfix expression evaluation algorithm in Figure 5 below.

```
def stack_eval_postfix(string)
  stack = LinkedStack.new
  string.chars do | ch |
    case
    when ch =~ /\d/
      stack.push(ch.to_i)
    when ch =~ /[+\-*\/%]/
      raise "Missing argument" if stack.empty?
      right_arg = stack.pop
      raise "Missing argument" if stack.empty?
      left_arg = stack.pop
      stack.push( evaluate(ch, left_arg, right_arg) )
    end
  end
  raise "Missing expresion" if stack.empty?
  raise "Too many arguments" unless stack.size == 1
  stack.pop
end
```

Figure 5: Stack-Based Algorithm to Evaluate Postfix Expressions

The strategy of this algorithm is quite simple: there is a single stack that holds arguments, and values are pushed on the stack whenever they are encountered in the input string. Whenever an operator is encountered, the top two values are popped of the stack, the operator is applied to them, and the result is pushed back on the stack. This continues until the string is exhausted, at which point the final value should be on the stack. If the stack becomes empty along the way, or there is more than one value on the stack when the input string is exhausted, then the input expression is not well-formed.

The recursive algorithm for evaluating postfix expressions is quite complicated. The strategy is to remember arguments in local variables, making recursive calls as necessary until an operator is encountered. We leave this algorithm as a challenging exercise.

The lesson of all these examples is that although it is always possible to write an algorithm using either recursion or stacks, in some cases a recursive algorithm is easier to develop, and in other cases a stack-based algorithm is easier. Each problem should be explored by sketching out both sorts of algorithms, and then choosing the one that appears easiest for detailed development.

8.4 Tail Recursive Algorithms

We have claimed that every recursive algorithms can be replaced with a non-recursive algorithm using a stack. This is true, but it overstates the case: sometimes a recursive algorithm can be replaced with a non-recursive algorithm that does not even need to use a stack. If a recursive algorithm is such that at most one recursive call is made as the final step in each execution of the algorithm's body, then the recursion can be replaced with a loop. No stack is needed because data for additional recursive calls is not needed—there are no additional recursive calls. A simple example is a recursive algorithm to search an array for a key, like the one in Figure 6.

```
def recursive_search(a, key)
  return false if a.size == 0
  return true if a[0] == key
  return recursive_search(a[1..-1],key)
end
```

Figure 6: A Recursive Factorial Algorithm

The recursion in this algorithm can be replaced with a simple loop as shown in Figure 7.

```
def search(a, key)
  a.each { | v | return true if v == key }
  return false
end
```

Figure 7: A Non-Recursive Factorial Algorithm

Algorithms that only call themselves at most once as the final step in every execution of their bodies, like the array search algorithm, are called *tail-recursive*.

> **Tail recursive algorithm:** A recursive algorithm that calls itself at most once as the last step in every execution of its body.

Recursion can always be removed from tail-recursive algorithms without using a stack.

8.5 Summary and Conclusion

Algorithms that use recursion can always be replaced by algorithms that use a stack, and vice versa, so stacks and recursion are in some sense equivalent. However, some algorithms are much easier to write using recursion, while others are easier to write using a stack. Which is which depends on the problem. Programmers should evaluate both alternatives when deciding how to solve individual problems.

8.6 Review Questions

1. Which of the algorithms for determining whether a string of brackets is balanced is easiest to for you to understand?
2. What characteristics do prefix, postfix, and infix expressions share?
3. Which is easier: evaluating a prefix expression with a stack or using recursion?
4. Which is easier: evaluating a postfix expression with a stack or using recursion?
5. Is the recursive algorithm to determine whether a string of brackets is balanced tail recursive? Explain why or why not.

8.7 Exercises

1. We can slightly change the definition of strings of balanced brackets to exclude the empty string. Restate the recursive definition and modify the algorithms to check strings of brackets to see whether they are balanced to incorporate this change.
2. Fill in the following table with equivalent expressions in each row.

Infix	Prefix	Postfix
(((2 * 3) - 4) * (8 / 3)) + 2		
	% + 8 * 2 6 - 8 4	
		8 2 - 3 * 4 5 + 8 % /

3. Write a recursive algorithm to evaluate postfix expressions as discussed in this chapter.
3. Write a recursive algorithm to evaluate infix expressions. Assume that operators have equal precedence and are left-associative so that, without parentheses, operations are evaluated from left to right. Parentheses alter the order of evaluation in the usual way.
4. Write a stack-based algorithm to evaluate infix expressions as defined in the last exercise.
5. Which of the algorithms for evaluating infix expressions is easier to develop?
6. Write a non-recursive algorithm that does not use a stack to determine whether a string of brackets is balanced. Hint: count brackets.

8.8 Review Question Answers

1. This answer depends on the individual, but most people probably find the stack-based algorithm a bit easier to understand because its strategy is so simple.
2. Prefix, postfix, and infix expressions list their arguments in the same order. The number of operators in each is always one less than the number of constant arguments. The main operator in each expression and sub-expression is easy to find: the main operator in an infix expression is the left-most operator inside the fewest number of parentheses; the main operator of a prefix expression is the first operator; the main operator of a postfix expression is the last operator.
3. Evaluating a prefix expression recursively is much easier than evaluating it with a stack.
4. Evaluating a postfix expression with a stack is much easier than evaluating it recursively.
5. The recursive algorithm to determine whether a string of brackets is balanced calls itself at most once on each activation, but the recursive call is not the last step in the execution of the body of the algorithm—there must be a check for the closing right bracket after the recursive call. Hence this operation is not tail recursive and it cannot be implemented without a stack. (There is a non-recursive algorithm to check for balanced brackets without using a stack, but it uses a completely different approach from the recursive algorithms—see exercise 7).