Basic Data Structures

Stacks

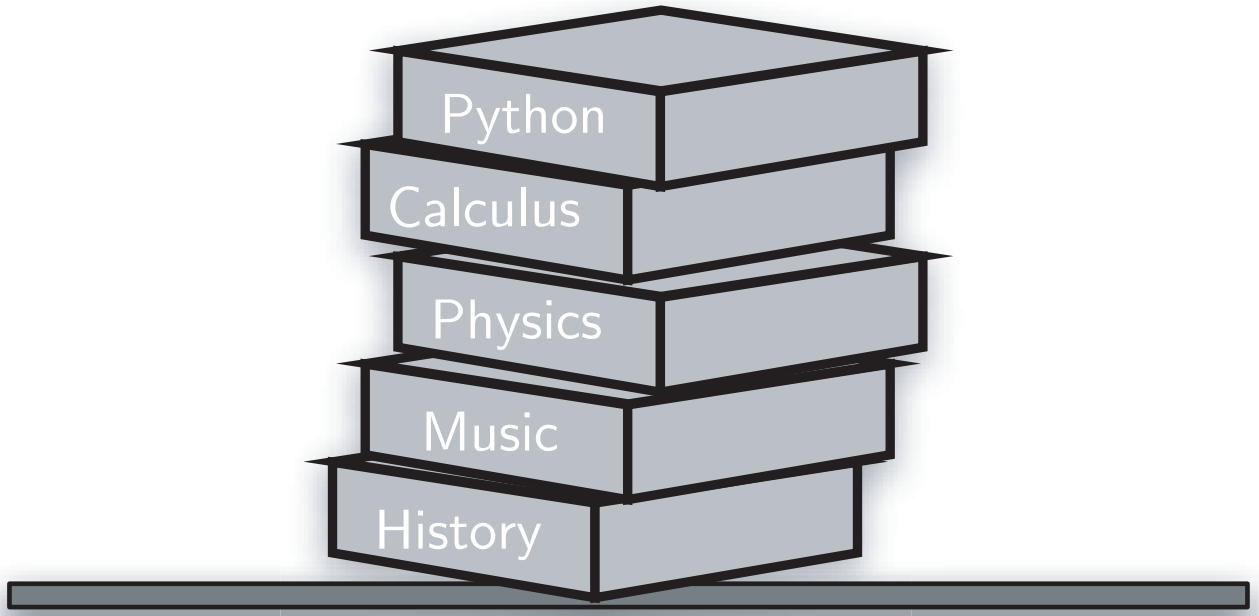
Brad Miller David Ranum

1/25/06

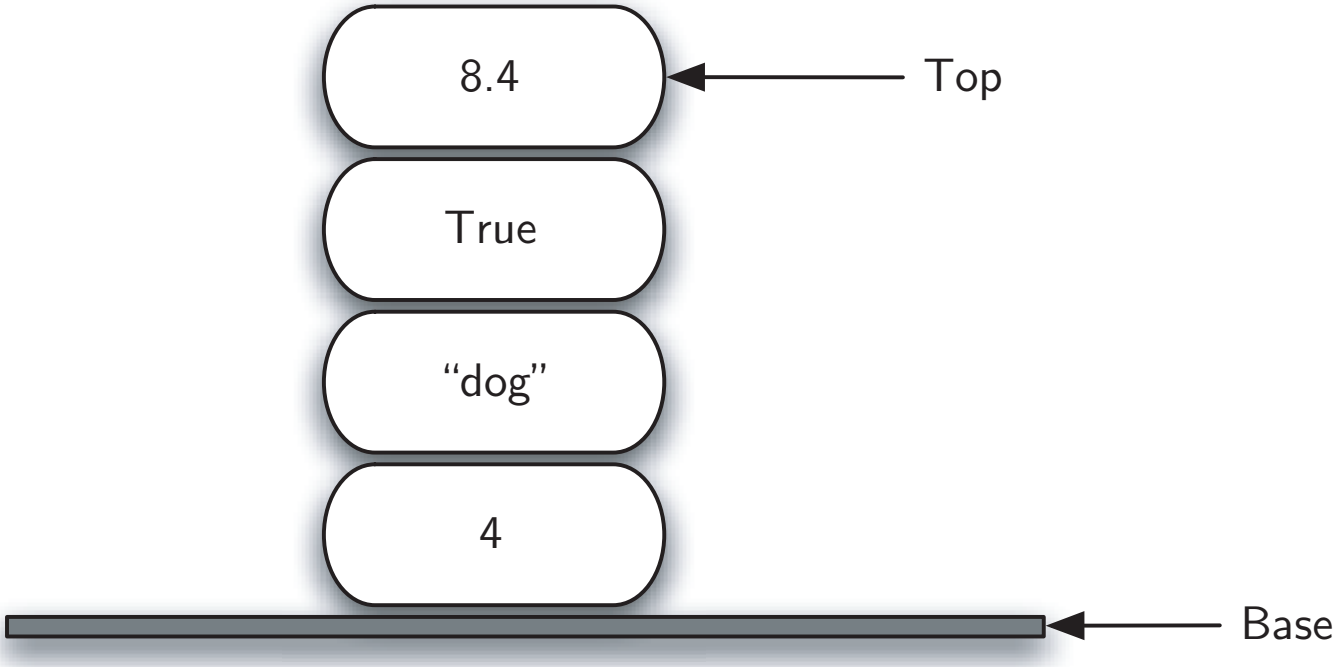
Outline

Outline

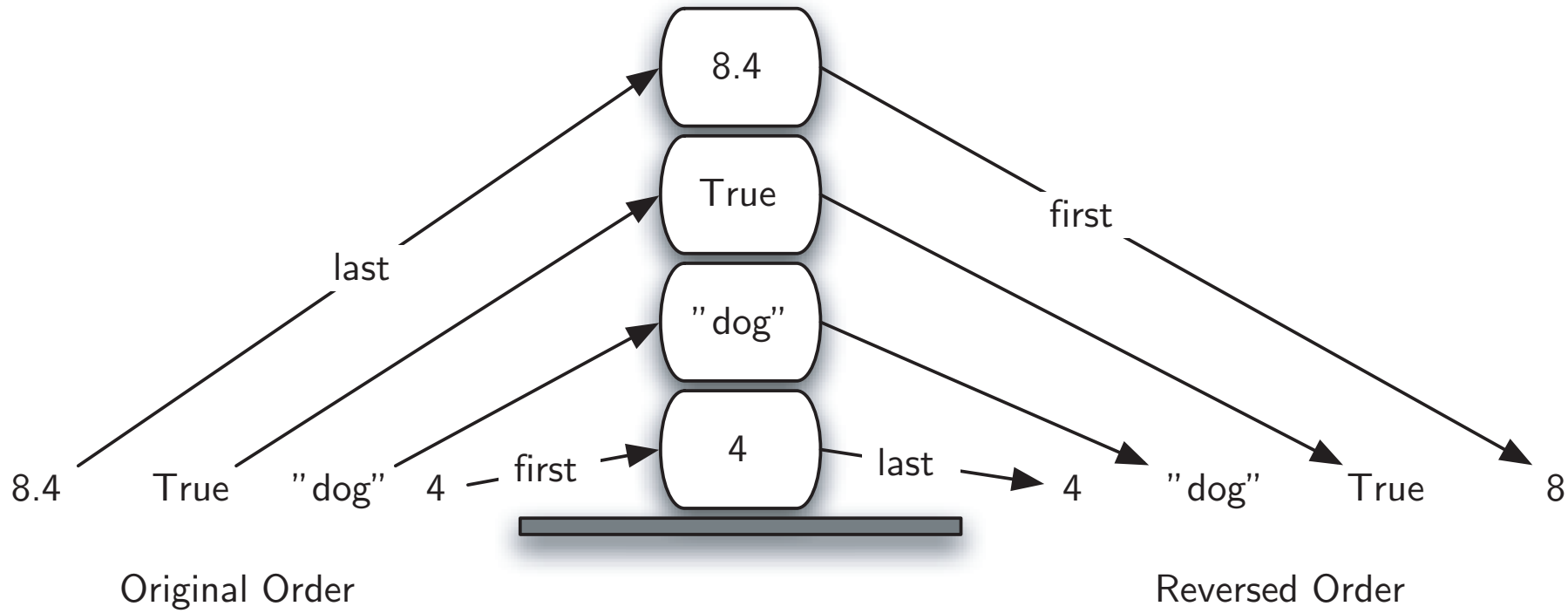
A Stack of Books



A Stack of Primitive Python Objects



The Reversal Property of Stacks



Outline

- `Stack()` creates a new stack that is empty. It needs no parameters and returns an empty stack.
- `push(item)` adds a new item to the top of the stack. It needs the item and returns nothing.
- `pop()` removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.
- `peek()` returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
- `isEmpty()` tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items on the stack. It needs no parameters and returns an integer.

Outline

Stack Implementation in Python I

```
1  class Stack:
2      def __init__(self):
3          self.items = []
4
5      def isEmpty(self):
6          return self.items == []
7
8      def push(self, item):
9          self.items.append(item)
10
11     def pop(self):
12         return self.items.pop()
13
14
```

Stack Implementation in Python II

```
15      def peek(self):
16          return self.items[len(self.items)-1]
17
18      def size(self):
19          return len(self.items)
```

Alternative ADT Stack Implementation in Python I

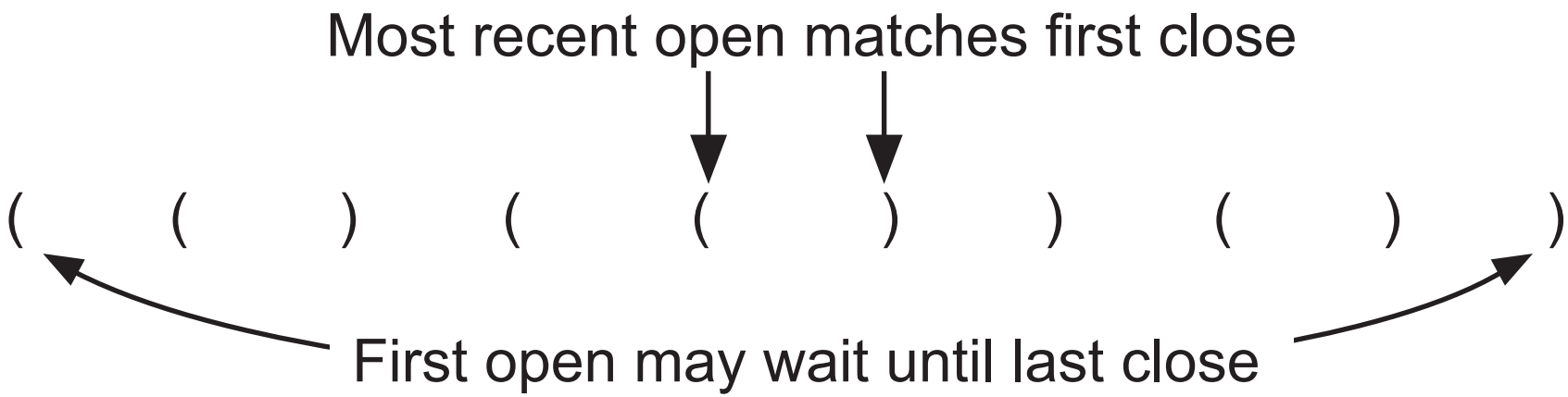
```
1  class Stack:
2      def __init__(self):
3          self.items = []
4
5      def isEmpty(self):
6          return self.items == []
7
8      def push(self, item):
9          self.items.insert(0,item)
10
11     def pop(self):
12         return self.items.pop(0)
13
14
```

Alternative ADT Stack Implementation in Python II

```
15        def peek(self):
16            return self.items[0]
17
18        def size(self):
19            return len(self.items)
```

Outline

Matching Parentheses



Simple Balanced Parentheses I

```
1  def parChecker(symbolString):
2      s = Stack()
3
4      balanced = True
5      index = 0
6
7      while index < len(symbolString) and balanced:
8          symbol = symbolString[index]
9          if symbol == "(":
10             s.push(symbol)
11         else:
12
13
14
```

Simple Balanced Parentheses II

```
15                if s.isEmpty():
16                    balanced = False
17                else:
18                    s.pop()
19
20            index = index + 1
21
22        if balanced and s.isEmpty():
23            return True
24        else:
25            return False
```

Outline

Balanced Symbols–A General Case I

```
1  def parChecker(symbolString):
2
3      s = Stack()
4
5      balanced = True
6      index = 0
7
8      while index < len(symbolString) and balanced:
9          symbol = symbolString[index]
10         if symbol in "([{":
11             s.push(symbol)
12         else:
13
14
```

Balanced Symbols–A General Case II

```
15
16                  if s.isEmpty():
17                      balanced = False
18                  else:
19                      top = s.pop()
20                      if not matches(top,symbol):
21                              balanced = False
22
23              index = index + 1
24
25          if balanced and s.isEmpty():
26              return True
27          else:
28              return False
29
```

Balanced Symbols–A General Case III

```
30  def matches(open,close):
31      opens = "([{"
32      closers = ")]}"
33
34      return opens.index(open) == closers.index(close)
```

Outline

Decimal-to-Binary Conversion

$233 / 2 = 116 \quad rem = 1$

$116 / 2 = 58 \quad rem = 0$

$58 / 2 = 29 \quad rem = 0$

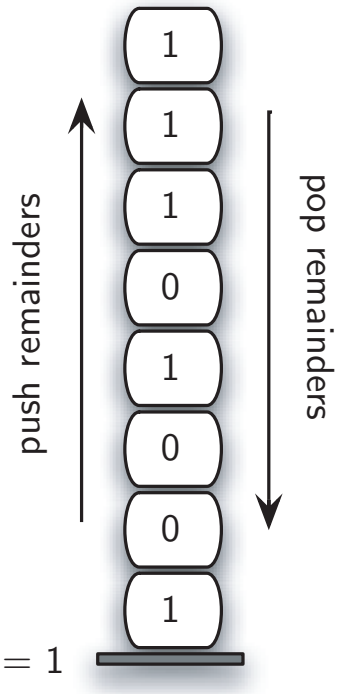
$29 / 2 = 14 \quad rem = 1$

$14 / 2 = 7 \quad rem = 0$

$7 / 2 = 3 \quad rem = 1$

$3 / 2 = 1 \quad rem = 1$

$1 / 2 = 0 \quad rem = 1$



Decimal to Binary Conversion

```
1  def divideBy2(decNumber):
2
3      remstack = Stack()
4
5      while decNumber > 0:
6          rem = decNumber % 2
7          remstack.push(rem)
8          decNumber = decNumber / 2
9
10     binString = ""
11     while not remstack.isEmpty():
12         binString = binString + str(remstack.pop())
13
14     return binString
```

Conversion to Any Base I

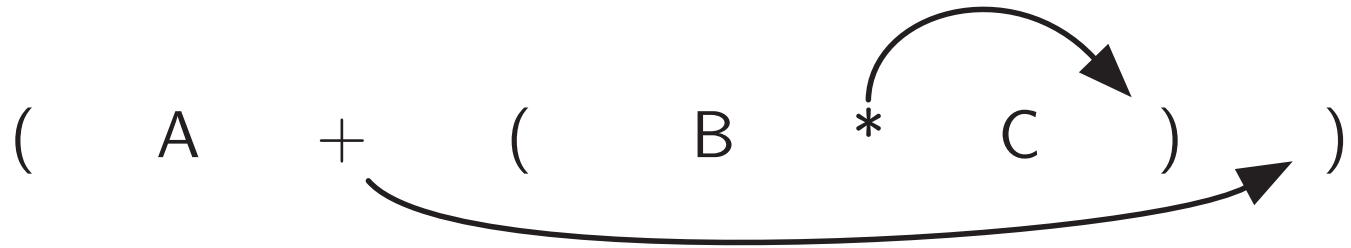
```
1  def baseConverter(decNumber,base):
2
3      digits = "0123456789ABCDEF"
4
5      remstack = Stack()
6
7      while decNumber > 0:
8          rem = decNumber % base
9          remstack.push(rem)
10         decNumber = decNumber / base
11
12
13
14
```

Conversion to Any Base II

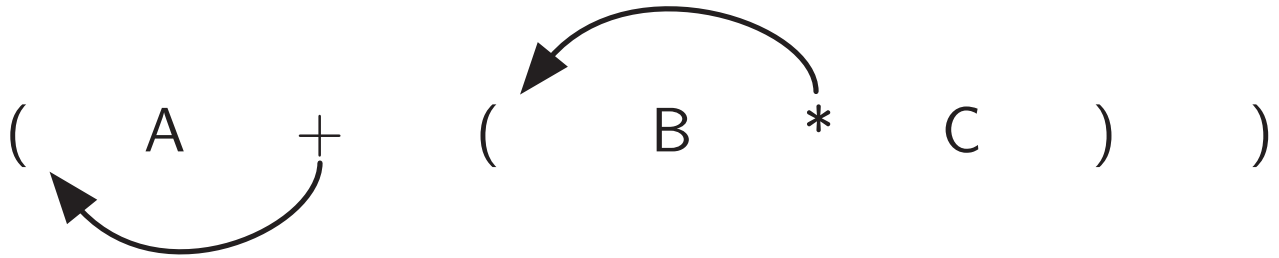
```
15      newString = ""
16      while not remstack.isEmpty():
17          newString = newString + digits[remstack.pop()]
18
19      return newString
```

Outline

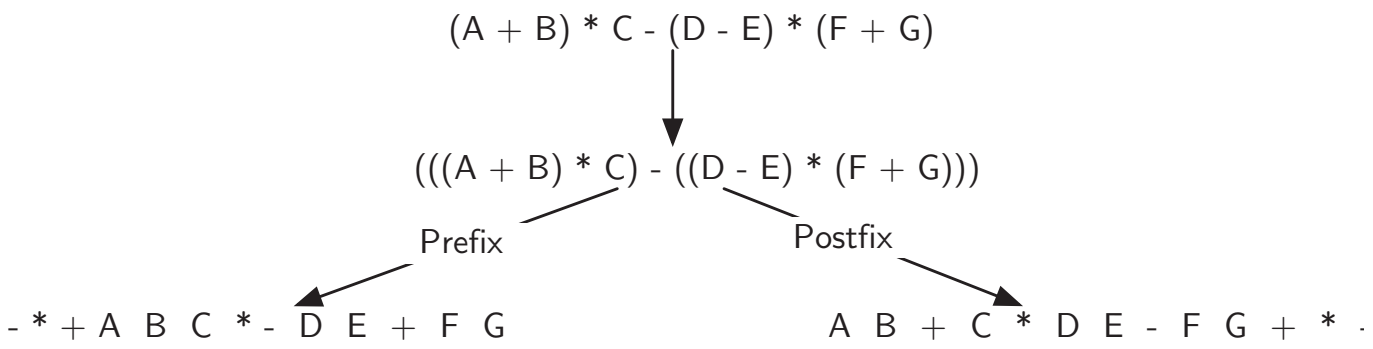
Moving Operators to the Right for Postfix Notation



Moving Operators to the Left for Prefix Notation



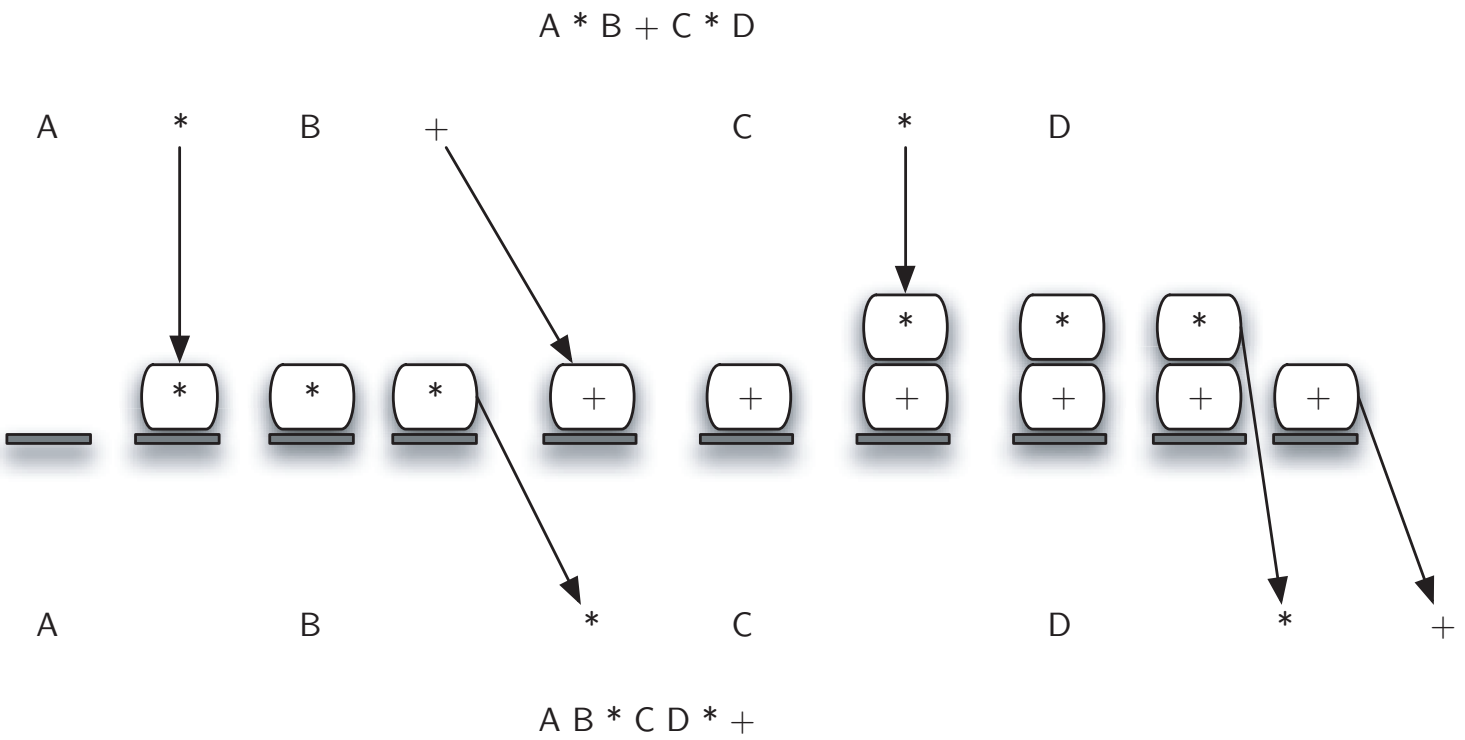
Converting a Complex Expression to Prefix and Postfix Notations



1. Create an empty stack called `opstack` for keeping operators. Create an empty list for output.
2. Convert the input infix string to a list by using the string method `split`.

3. Scan the token list from left to right.
   - If the token is an operand, append it to the end of the output list.
   - If the token is a left parenthesis, push it on the `opstack`.
   - If the token is a right parenthesis, pop the `opstack` until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
   - If the token is an operator, *, /, +, or -, push it on the `opstack`. However, first remove any operators already on the `opstack` that have higher or equal precedence and append them to the output list.
4. When the input expression has been completely processed, check the `opstack`. Any operators still on the stack can be removed and appended to the end of the output list.

Converting A * B + C * D to Postfix Notation



Converting Infix Expressions to Postfix Expressions I

```
1  import string
2  def infixToPostfix(infixexpr):
3
4      prec = {}
5      prec["*"] = 3
6      prec["/"] = 3
7      prec["+"] = 2
8      prec["-"] = 2
9      prec["("] = 1
10
11     opStack = Stack()
12     postfixList = []
13
14     tokenList = infixexpr.split()
```

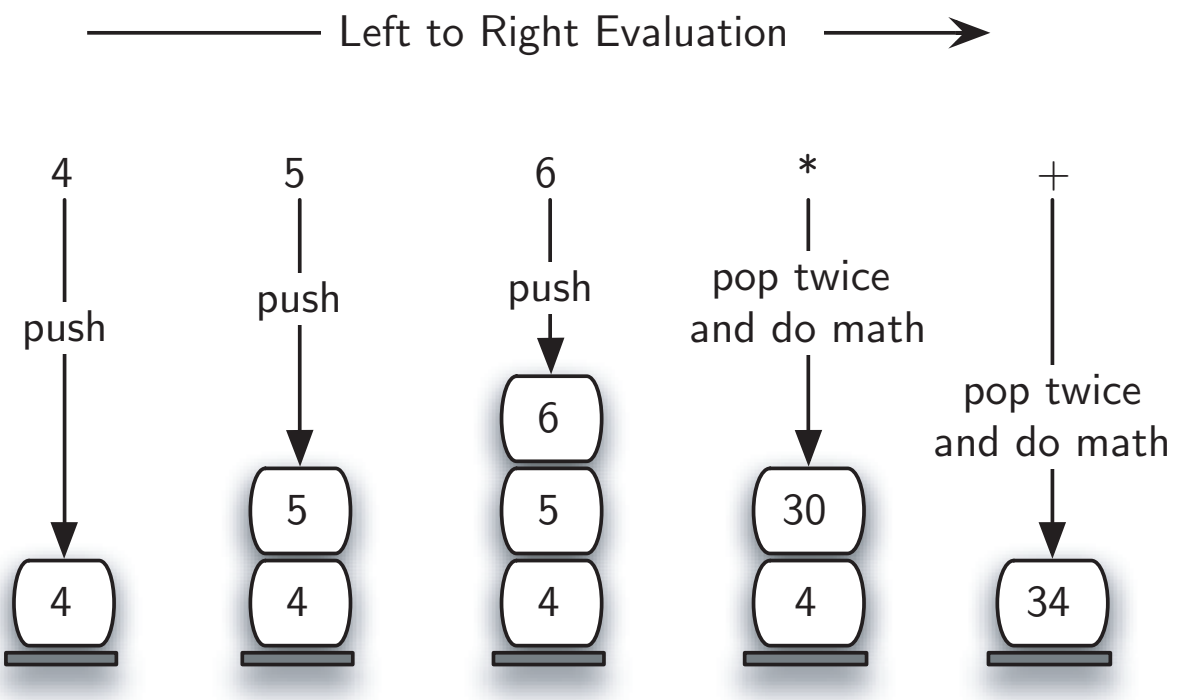
Converting Infix Expressions to Postfix Expressions II

```
15
16      for token in tokenList:
17          if token in string.uppercase:
18              postfixList.append(token)
19          elif token == '(':
20              opStack.push(token)
21          elif token == ')':
22              topToken = opStack.pop()
23              while topToken != '(':
24                  postfixList.append(topToken)
25                  topToken = opStack.pop()
26
27
28
29
```

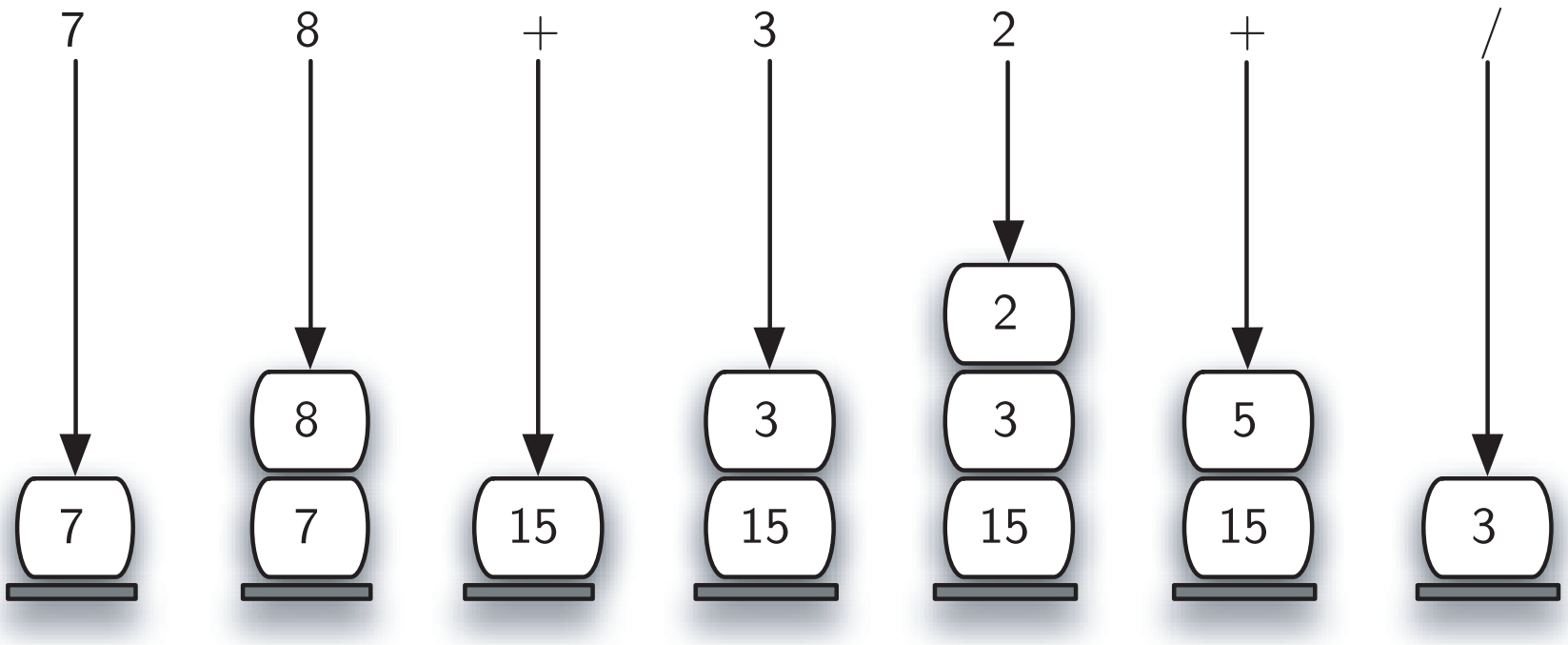
Converting Infix Expressions to Postfix Expressions III

```
30            else:
31                while (not opStack.isEmpty()) and \
32                   (prec[opStack.peek()] >= prec[token]):
33                      postfixList.append(opStack.pop())
34
35                opStack.push(token)
36
37        while not opStack.isEmpty():
38            postfixList.append(opStack.pop())
39
40        return string.join(postfixList)
```

Stack Contents During Evaluation



A More Complex Example of Evaluation



1. Create an empty stack called `operandStack`.
2. Convert the string to a list by using the string method `split`.
3. Scan the token list from left to right.
   - If the token is an operand, convert it from a string to an integer and push the value onto the `operandStack`.
   - If the token is an operator, *, /, +, or -, it will need two operands. Pop the `operandStack` twice. The first pop is the second operand and the second pop is the first operand. Perform the arithmetic operation. Push the result back on the `operandStack`.
4. When the input expression has been completely processed, the result is on the stack. Pop the `operandStack` and return the value.

Postfix Evaluation I

```
1  def postfixEval(postfixExpr):
2      operandStack = Stack()
3      tokenList = postfixExpr.split()
4
5      for token in tokenList:
6          if token in "0123456789":
7              operandStack.push(int(token))
8          else:
9              operand2 = operandStack.pop()
10             operand1 = operandStack.pop()
11             result = doMath(token,operand1,operand2)
12             operandStack.push(result)
13
14     return operandStack.pop()
```

Postfix Evaluation II

```
15
16  def doMath(op, op1, op2):
17      if op == "*":
18          return op1 * op2
19      else:
20          if op == "/":
21              return op1 / op2
22          else:
23              if op == "+":
24                  return op1 + op2
25              else:
26                  return op1 - op2
```