University of Pittsburgh

Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez’s and Dr. Farnan’s CS1501 slides.)

Announcements

- Upcoming Deadlines:
  - Homework 5: this Friday @ 11:59 pm
  - Lab 4: next Monday @ 11:59 pm
  - Programming Assignment 1: ~~Friday Oct. 7th~~ Monday Oct. 10th
- **Live Remote Support Session** for Assignment 1
  - Recording and slides on Canvas
- **Student Support Hours** of the teaching team are posted on the Syllabus page

Previous Lecture …

- ADT List
  - Refined Linked implementation with head and tail references
- ADT Stack
  - Array-based implementation
  - Linked implementation

Muddiest Points

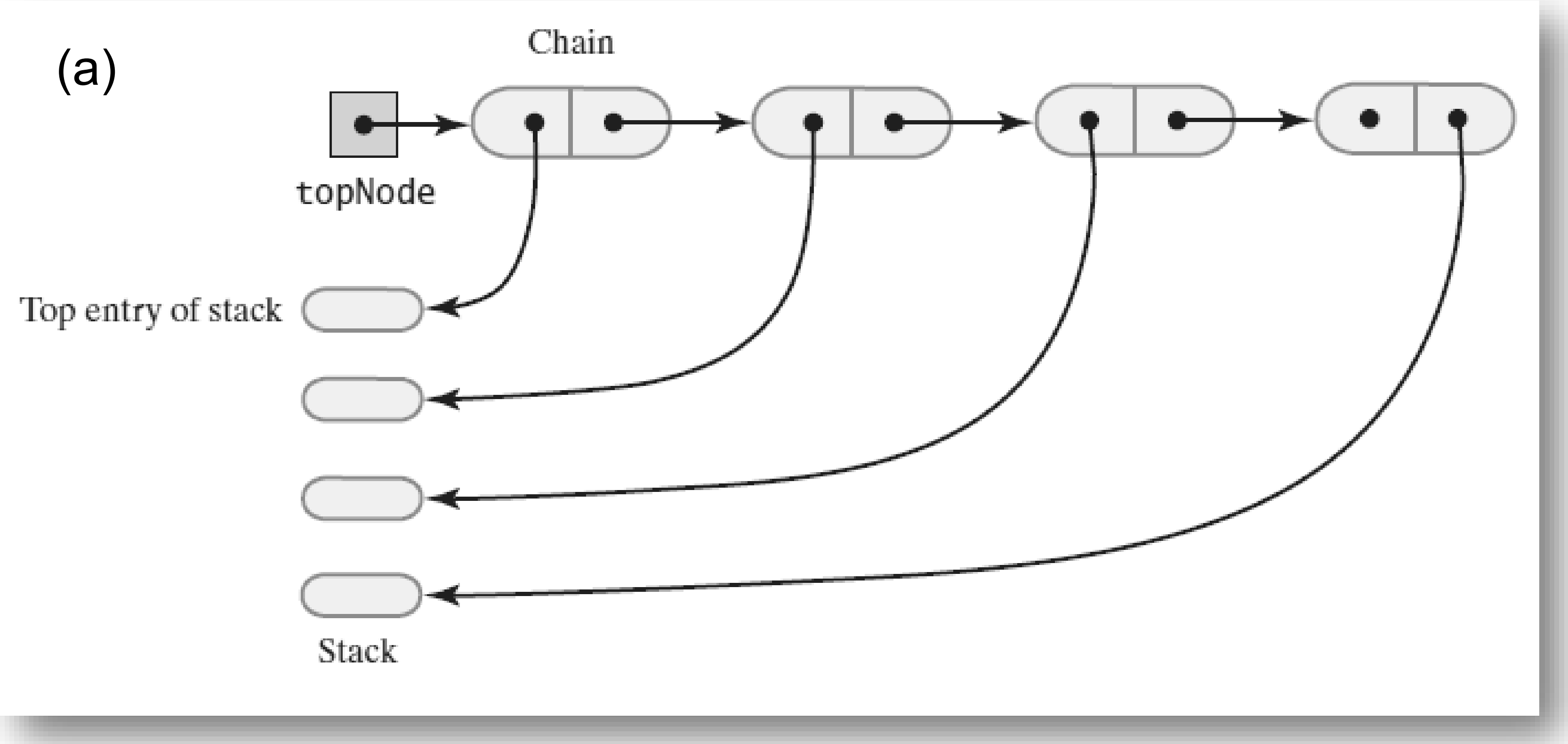
- **Q: Assignment 1. It's due in a week, worth 10% of our final grade, and there has been literally no guidance on how to start or successfully complete it. I have no experience with two-dimensional arrays, or arrays of objects, or implementing a new interface (especially one that isn't included in the textbook or hasn't been taught in lecture.) This seems like we're being asked to run while still learning to walk.**
- I will host a live remote support session this Friday @ 2:00 pm

Today …

- ADT Stack
  - Linked implementation
  - Implementation using ADT List
  - Application: Building a simple parser of Algebraic expressions

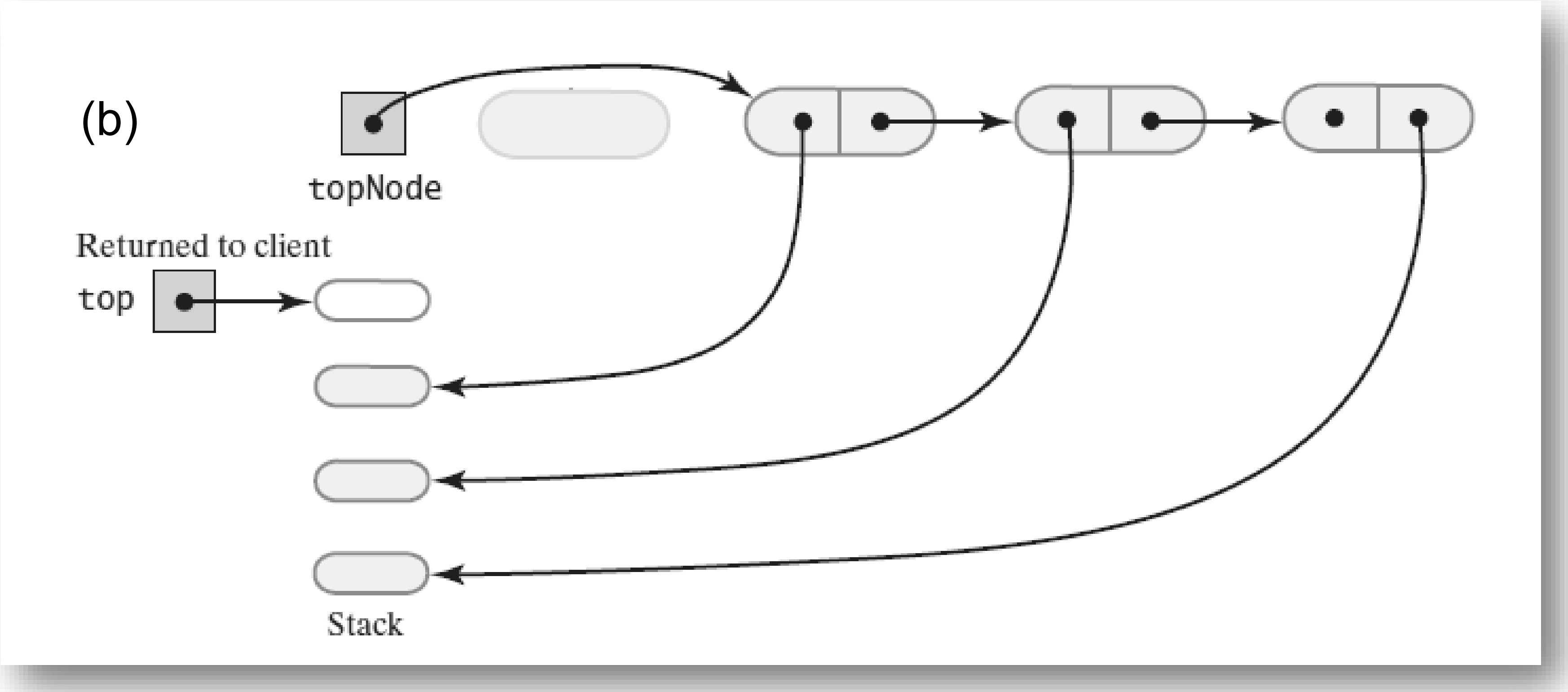
Linked Implementation

The stack before the
first node in the chain is deleted



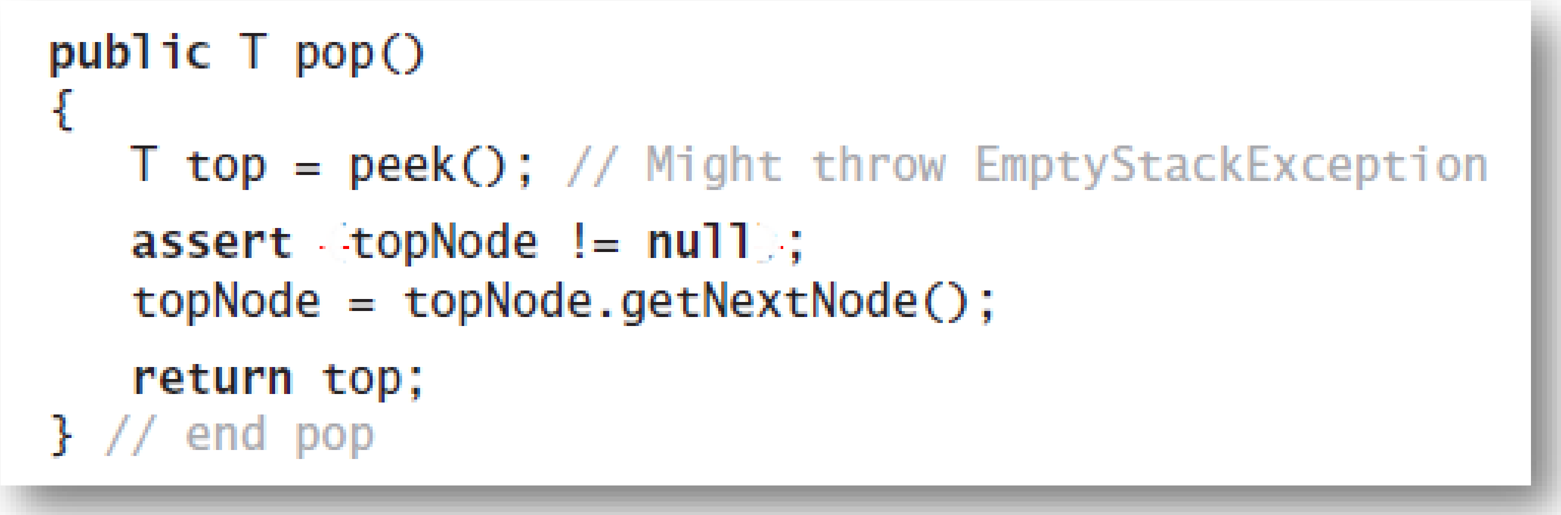
Linked Implementation

The stack after the
first node in the chain is deleted



Linked Implementation

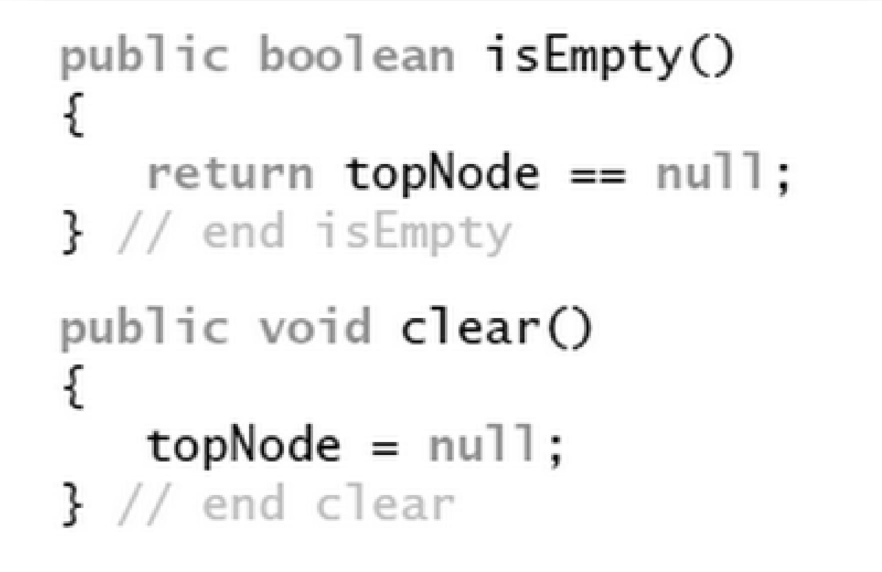
- Definition of **pop**



```
public T pop()
{
   T top = peek(); // Might throw EmptyStackException
   assert  topNode != null  ;
   topNode = topNode.getNextNode();
   return top;
} // end pop
```

Linked Implementation

- Definition of rest of class.



```
public boolean isEmpty()
{
   return topNode == null;
} // end isEmpty

public void clear()
{
   topNode = null;
} // end clear
```

ADT Stack Application

Let’s use the ADT Stack to design and implement a simple parser of Algebraic Expressions

Processing Algebraic Expressions

- Algebraic expressions can take different forms:
  - Infix: each binary operator appears between its operands *a* + *b*
  - Prefix: each binary operator appears before its operands + *a* *b*
  - Postfix*:* each binary operator appears after its operands *a* *b* +
- Prefix and Postfix forms are easy to evaluate
  - no parentheses needed
  - no need for operator precedence rules while evaluating the Postfix expression
- But we have to make sure first that the expressions is balanced
  - parentheses paired correctly

Our Plan

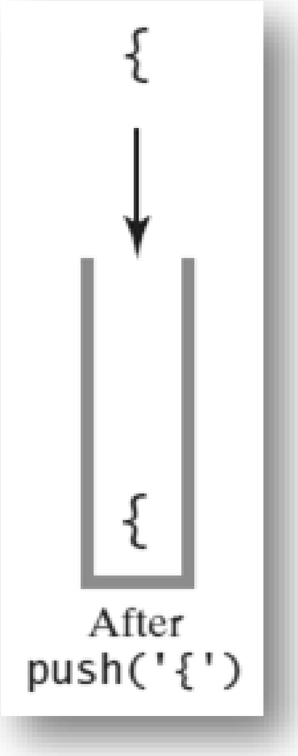
1. Check if input infix expression is balanced
2. Convert the expression from infix to postfix
3. Evaluate the postfix expression

Step 1: Balance Checking an Algebraic Expression

- Let’s use a stack!
- initialize an empty Stack
- for each character in the input infix expressions
  - if an open parenthesis
    - push to Stack
  - if a closing parenthesis
    - pop from stack and compare
    - if a matching pair, continue
    - else, report unbalanced and stop
  - if the stack is not empty
    - report unbalanced and stop
  - report balanced

Balance Checking an Algebraic Expression

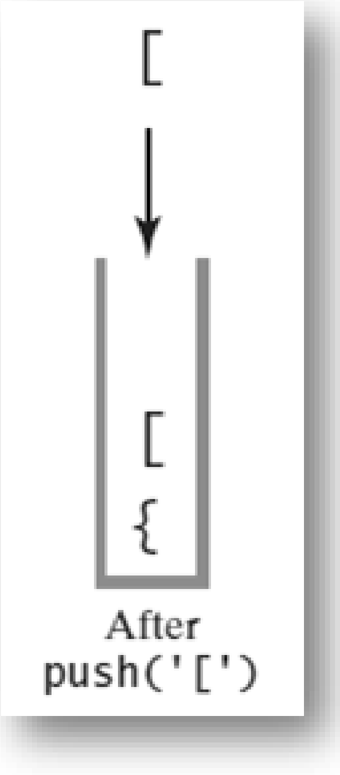
The contents of a stack during the scan of an expression that contains the balanced delimiters { [( )] }



{

Balance Checking an Algebraic Expression

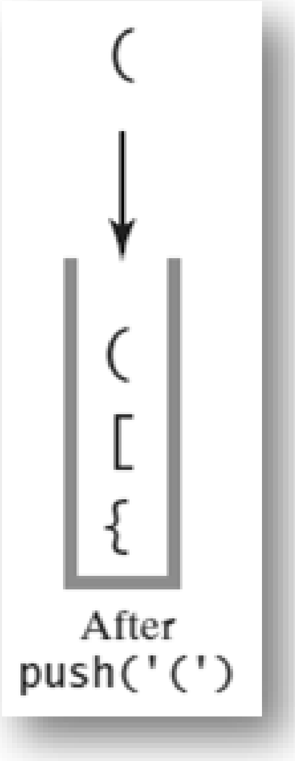
The contents of a stack during the scan of an expression that contains the balanced delimiters { [( )] }



[

Balance Checking an Algebraic Expression

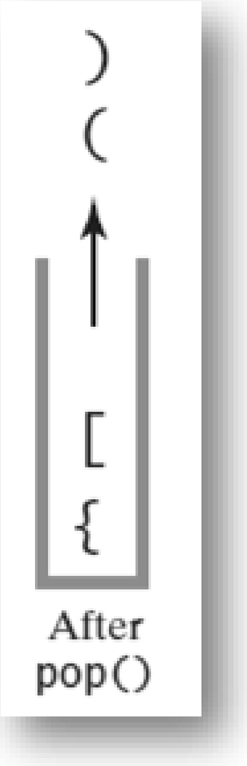
The contents of a stack during the scan of an expression that contains the balanced delimiters { [( )] }



(

Balance Checking an Algebraic Expression

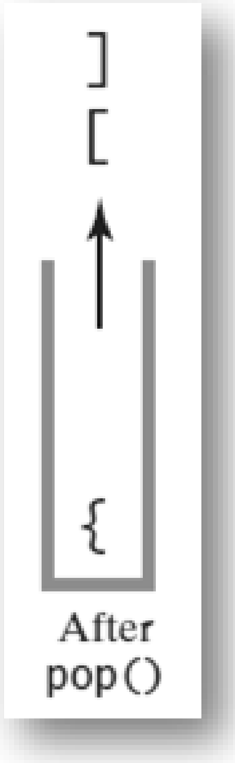
The contents of a stack during the scan of an expression that contains the balanced delimiters { [( )] }



)

Balance Checking an Algebraic Expression

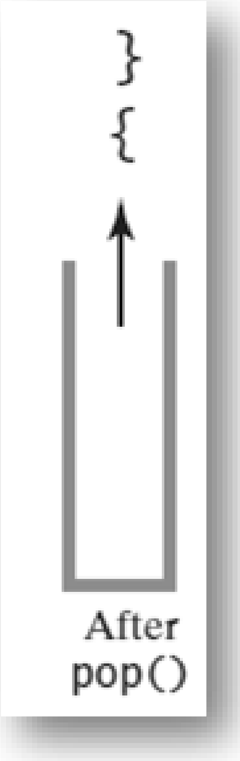
The contents of a stack during the scan of an expression that contains the balanced delimiters { [( )] }



]

Balance Checking an Algebraic Expression

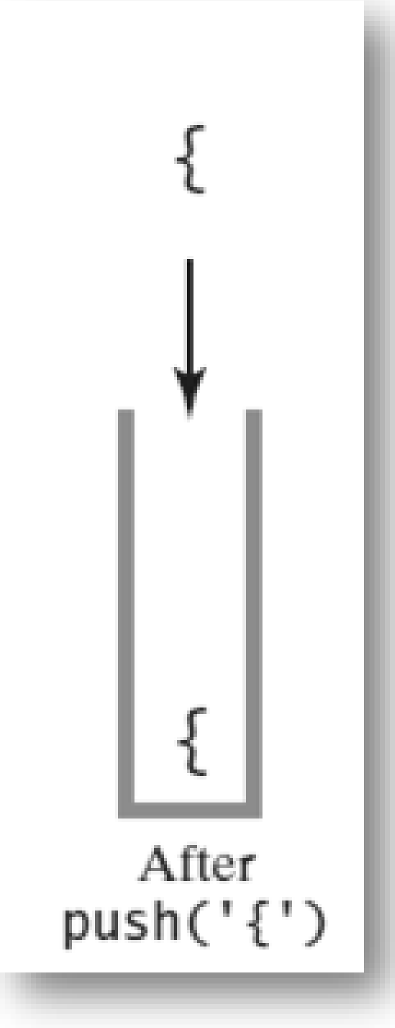
The contents of a stack during the scan of an expression that contains the balanced delimiters { [( )] }



}

Unbalanced Expression: Case 1

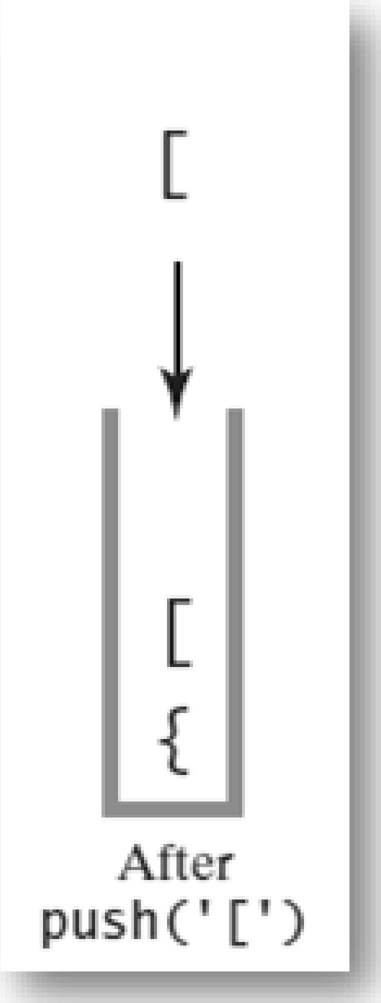
The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( ]) }



{

Unbalanced Expression: Case 1

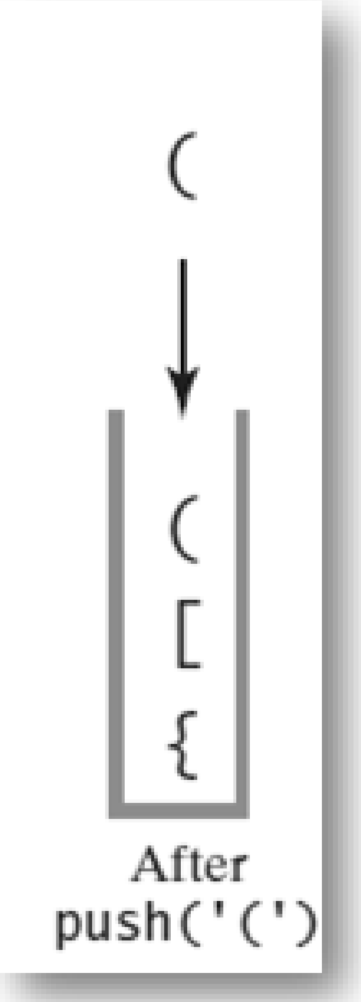
The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( ]) }



[

Unbalanced Expression: Case 1

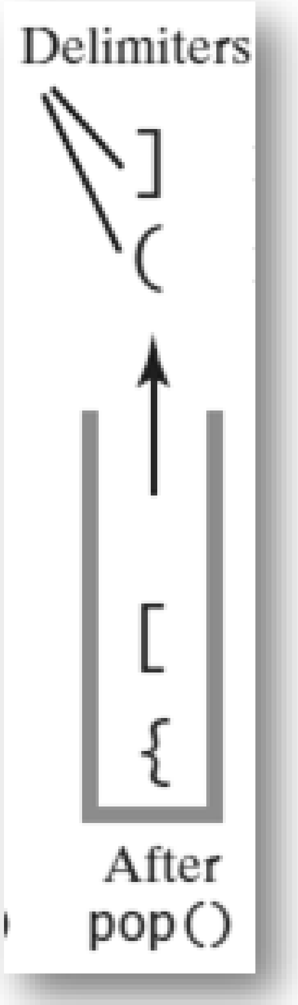
The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( ]) }



(

Unbalanced Expression: Case 1

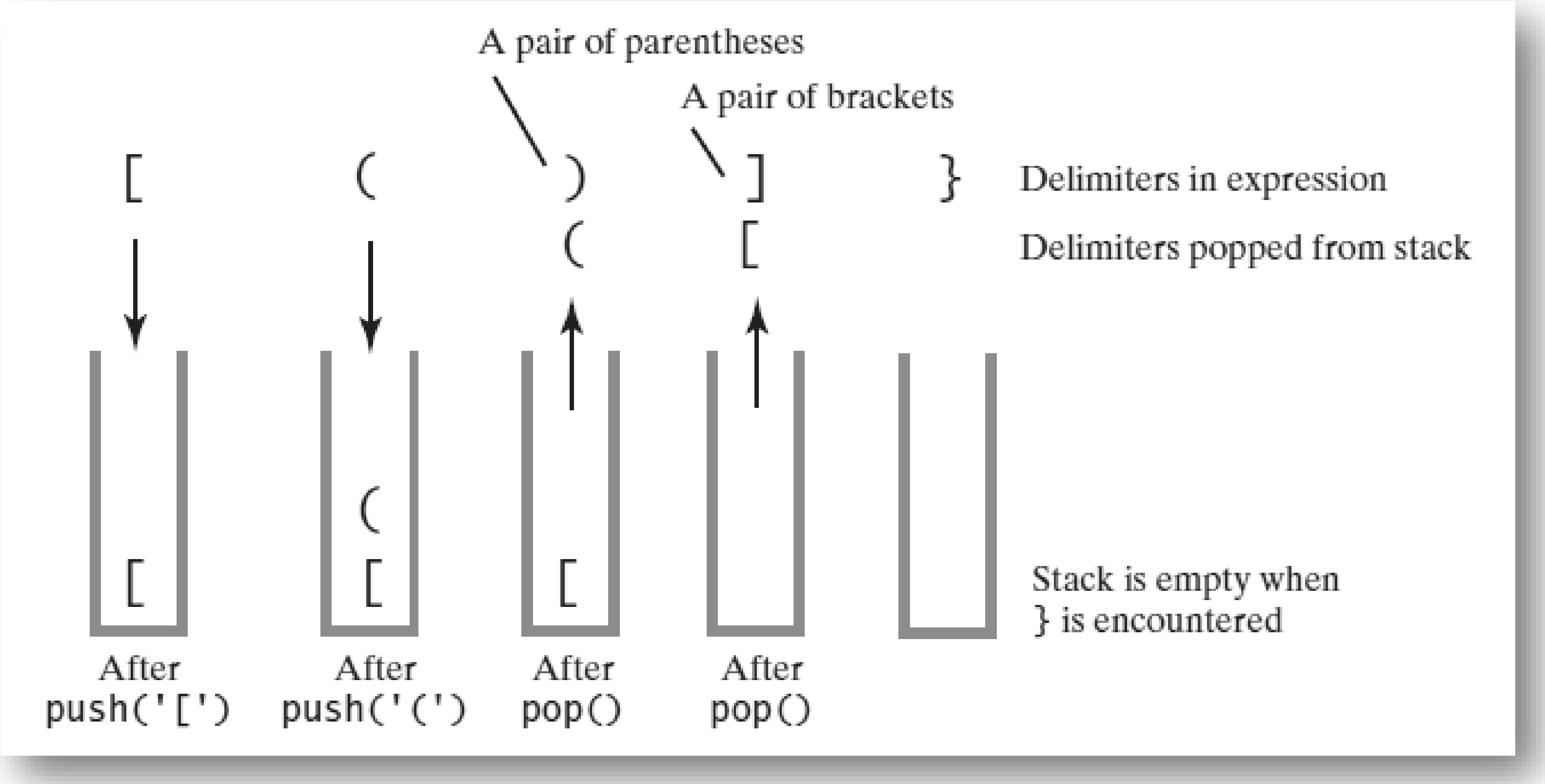
The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( ]) }



]

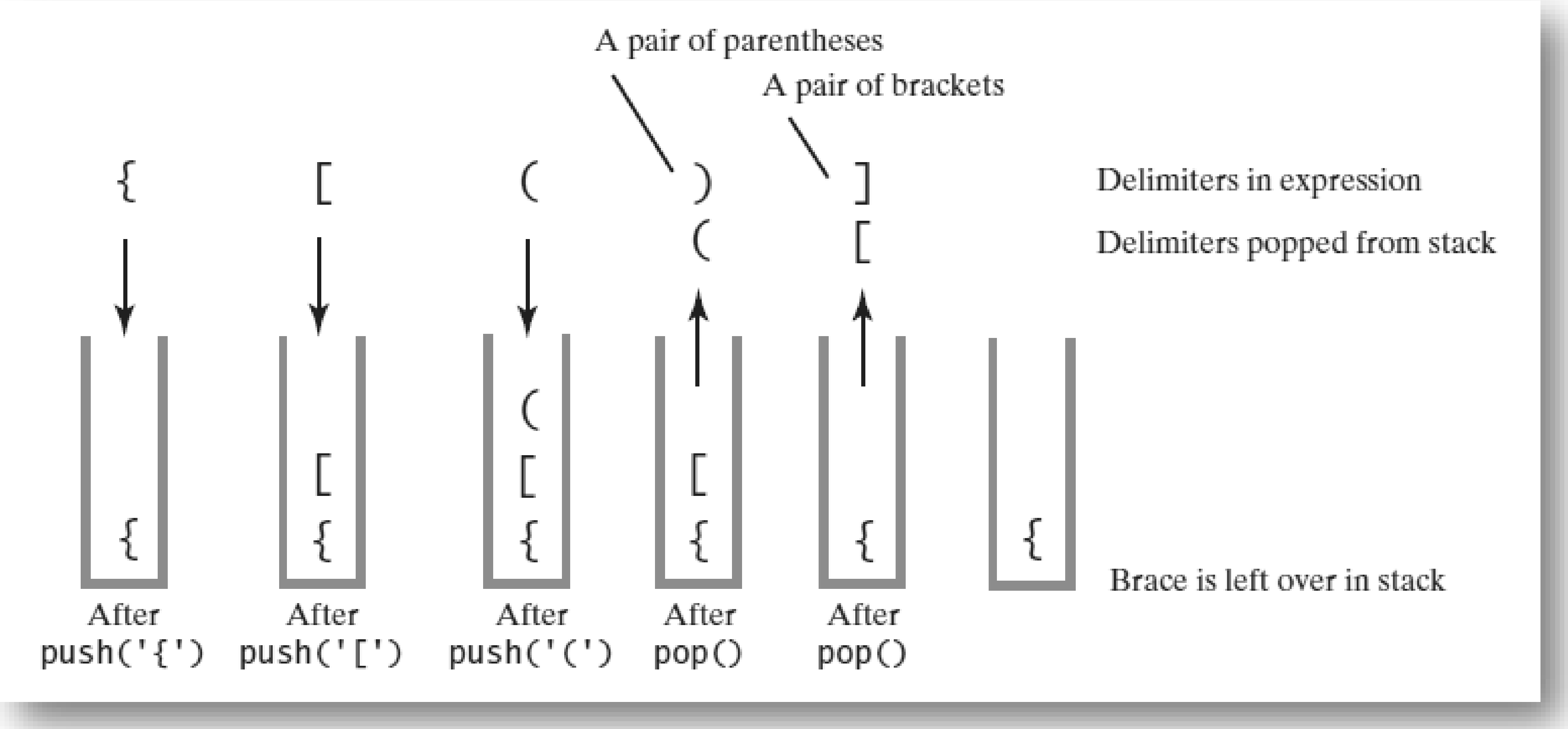
Unbalanced Expression: Case 2

The contents of a stack during the scan of an expression that contains the unbalanced delimiters [( )] }



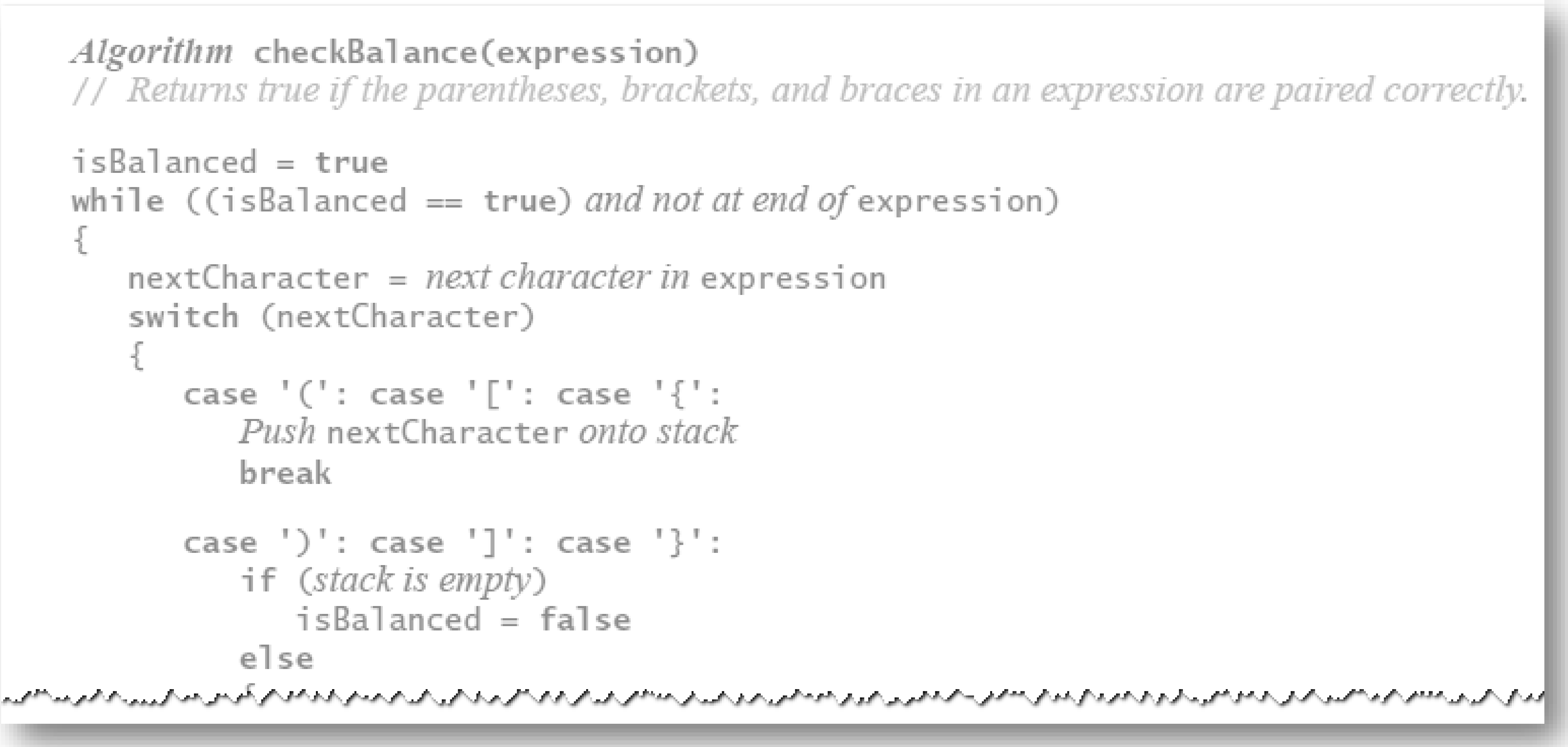
Unbalanced Expression: Case 3

The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( )]



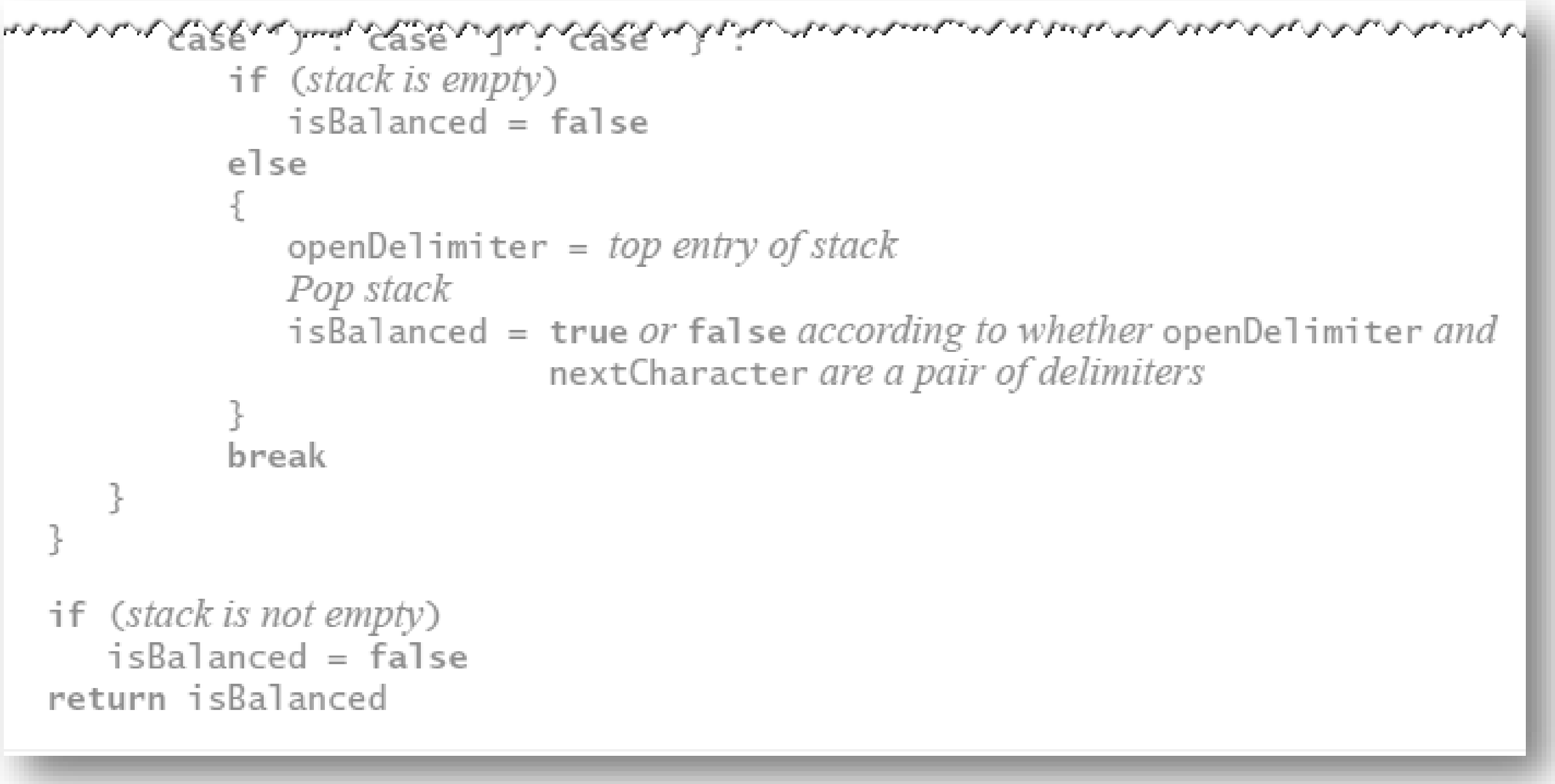
Processing Algebraic Expressions

Algorithm to check for balanced expression



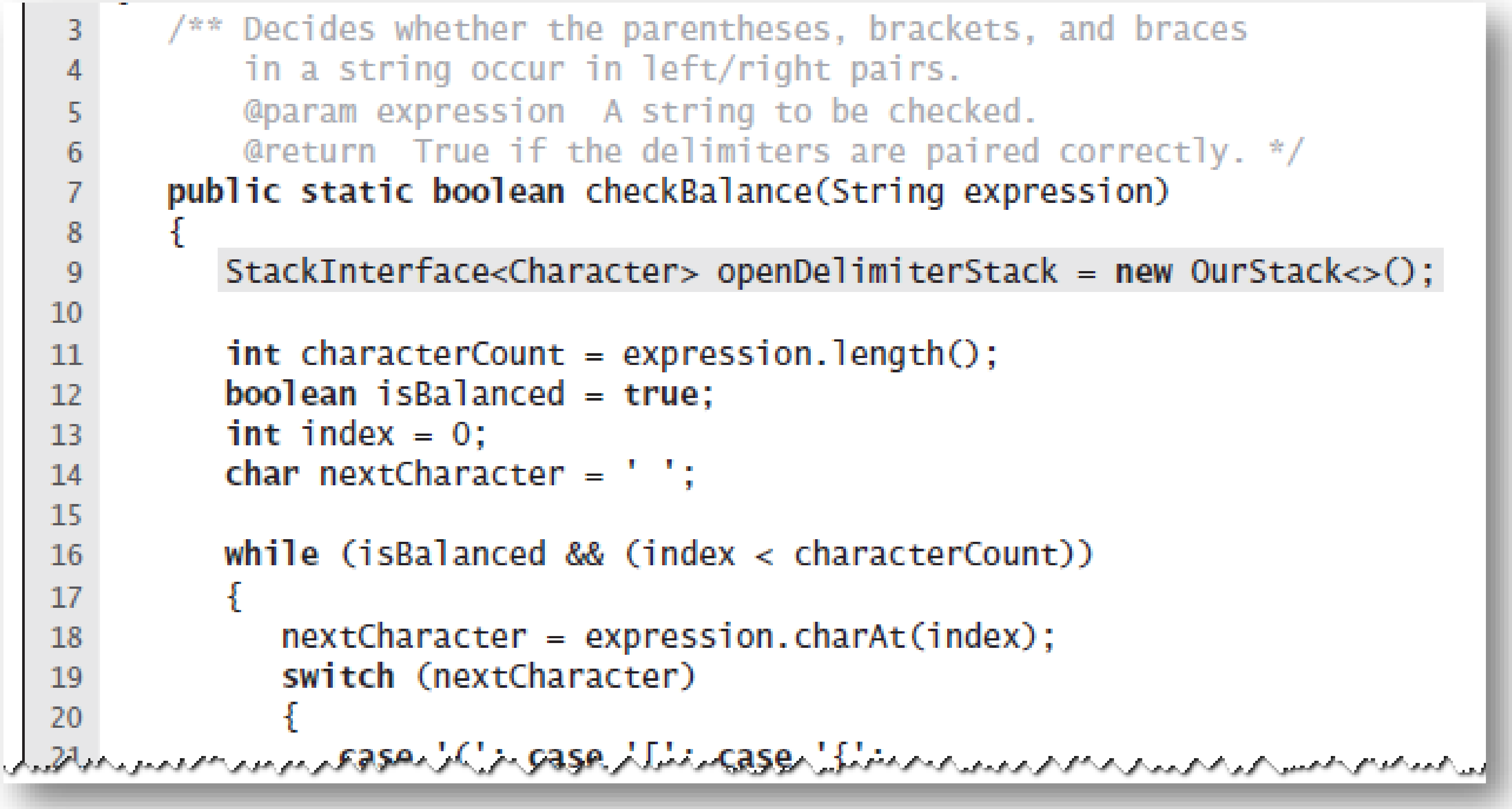
```
Algorithm checkBalance(expression)
// Returns true if the parentheses, brackets, and braces in an expression are paired correctly.

isBalanced = true
while ((isBalanced == true) and not at end of expression)
{
   nextCharacter = next character in expression
   switch (nextCharacter)
   {
      case '(': case '[': case '{':
         Push nextCharacter onto stack
         break

      case ')': case ']': case '}':
         if (stack is empty)
            isBalanced = false
         else
```

Processing Algebraic Expressions



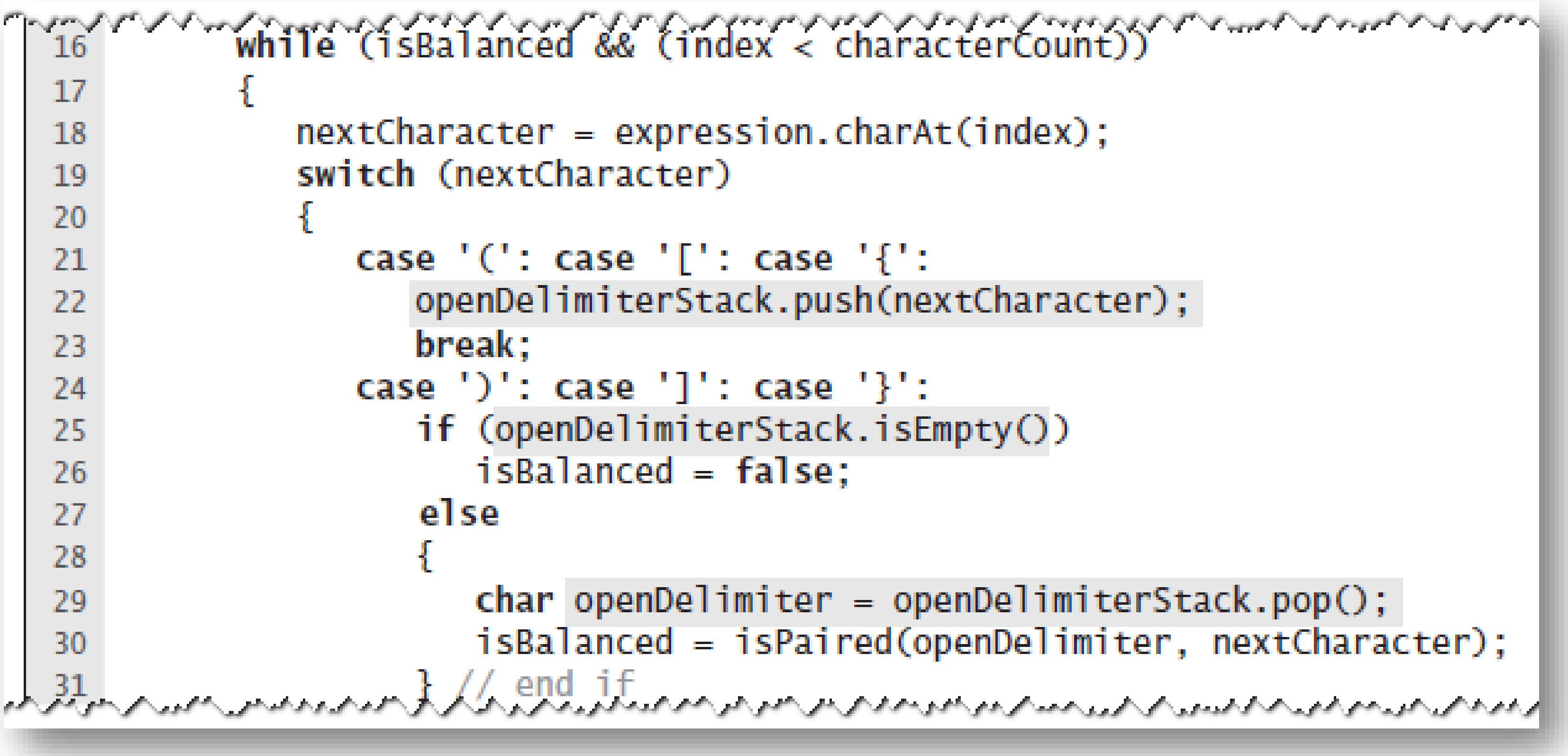
```
      case ')' : case ']' : case '}' :
         if (stack is empty)
            isBalanced = false
         else
         {
            openDelimiter = top entry of stack
            Pop stack
            isBalanced = true or false according to whether openDelimiter and
                             nextCharacter are a pair of delimiters
         }
         break
   }
}

if (stack is not empty)
   isBalanced = false
return isBalanced
```

Java Implementation



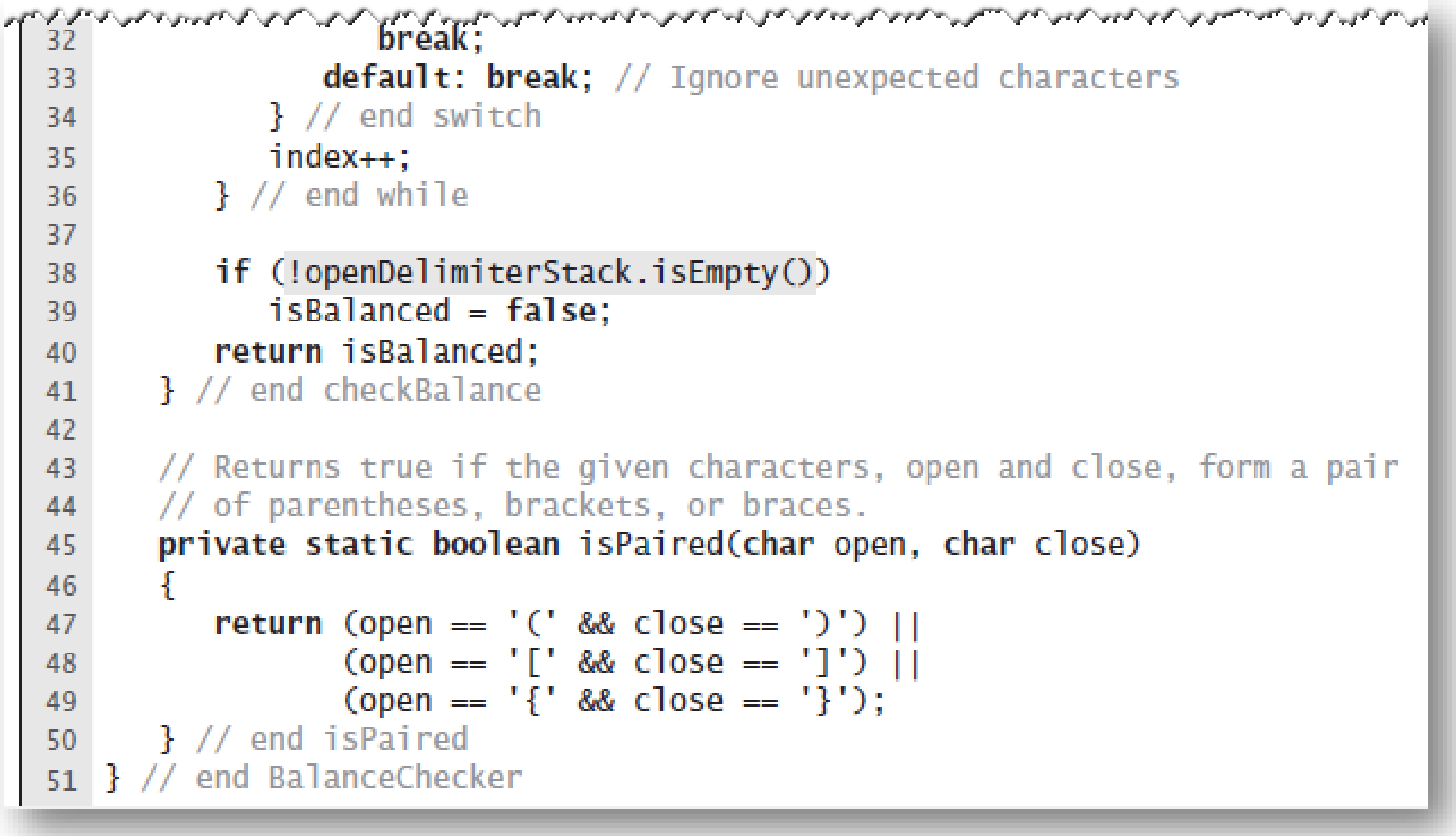
```java
/** Decides whether the parentheses, brackets, and braces
    in a string occur in left/right pairs.
    @param expression  A string to be checked.
    @return  True if the delimiters are paired correctly. */
public static boolean checkBalance(String expression)
{
   StackInterface<Character> openDelimiterStack = new OurStack<>();

   int characterCount = expression.length();
   boolean isBalanced = true;
   int index = 0;
   char nextCharacter = ' ';

   while (isBalanced && (index < characterCount))
   {
      nextCharacter = expression.charAt(index);
      switch (nextCharacter)
      {
         case '(': case '[': case '{':
```

Java Implementation



```
      while (isBalanced && (index < characterCount))
      {
         nextCharacter = expression.charAt(index);
         switch (nextCharacter)
         {
            case '(': case '[': case '{':
               openDelimiterStack.push(nextCharacter);
               break;
            case ')': case ']': case '}':
               if (openDelimiterStack.isEmpty())
                  isBalanced = false;
               else
               {
                  char openDelimiter = openDelimiterStack.pop();
                  isBalanced = isPaired(openDelimiter, nextCharacter);
               } // end if
```

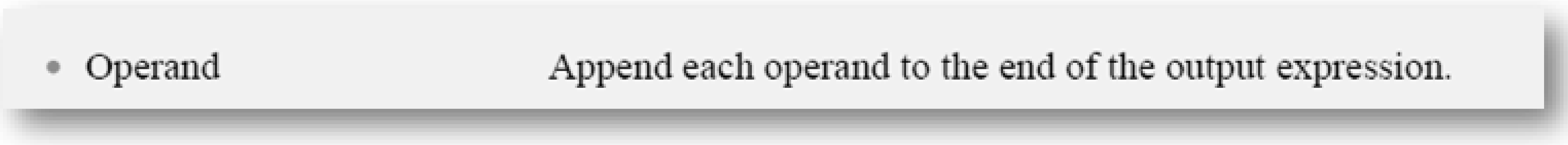
Java Implementation



```
               break;
            default: break; // Ignore unexpected characters
         } // end switch
         index++;
      } // end while

      if (!openDelimiterStack.isEmpty())
         isBalanced = false;
      return isBalanced;
   } // end checkBalance

   // Returns true if the given characters, open and close, form a pair
   // of parentheses, brackets, or braces.
   private static boolean isPaired(char open, char close)
   {
      return (open == '(' && close == ')') ||
             (open == '[' && close == ']') ||
             (open == '{' && close == '}');
   } // end isPaired
} // end BalanceChecker
```

Step 2: Infix-to-postfix Conversion Algorithm

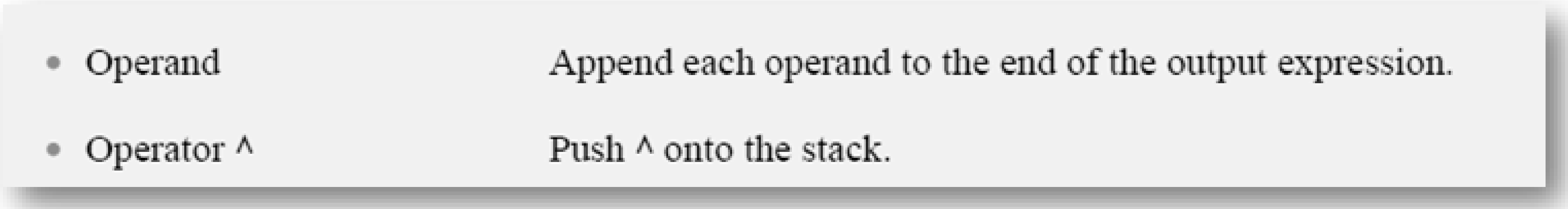
- for each character in the input expression



- Operand Append each operand to the end of the output expression.

Infix-to-postfix Conversion Algorithm

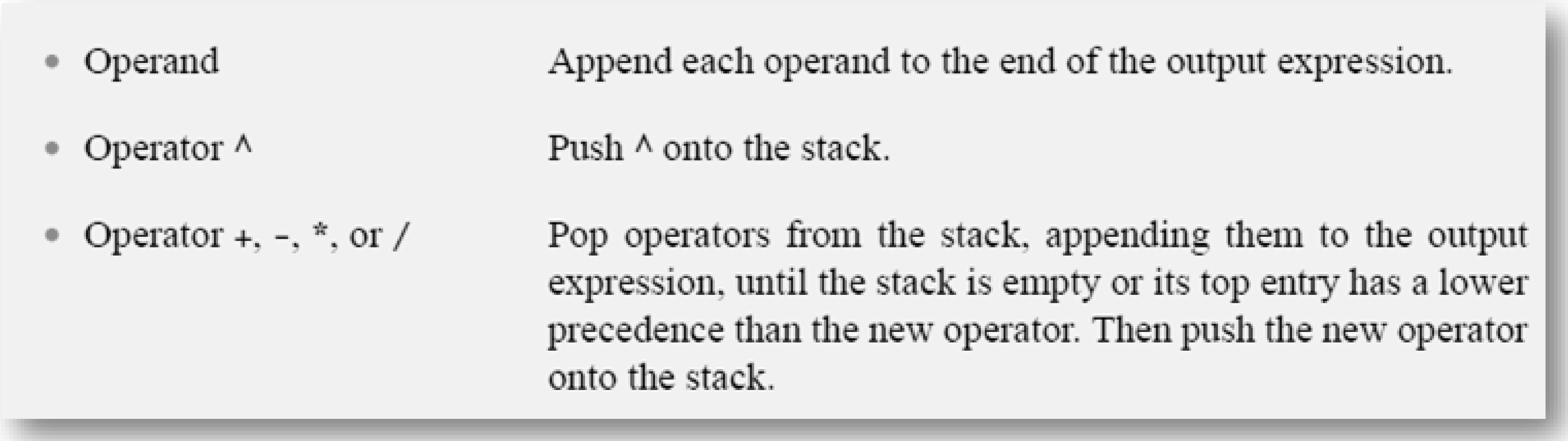
- for each character in the input expression



• Operand	Append each operand to the end of the output expression.
• Operator ^	Push ^ onto the stack.

Infix-to-postfix Conversion Algorithm

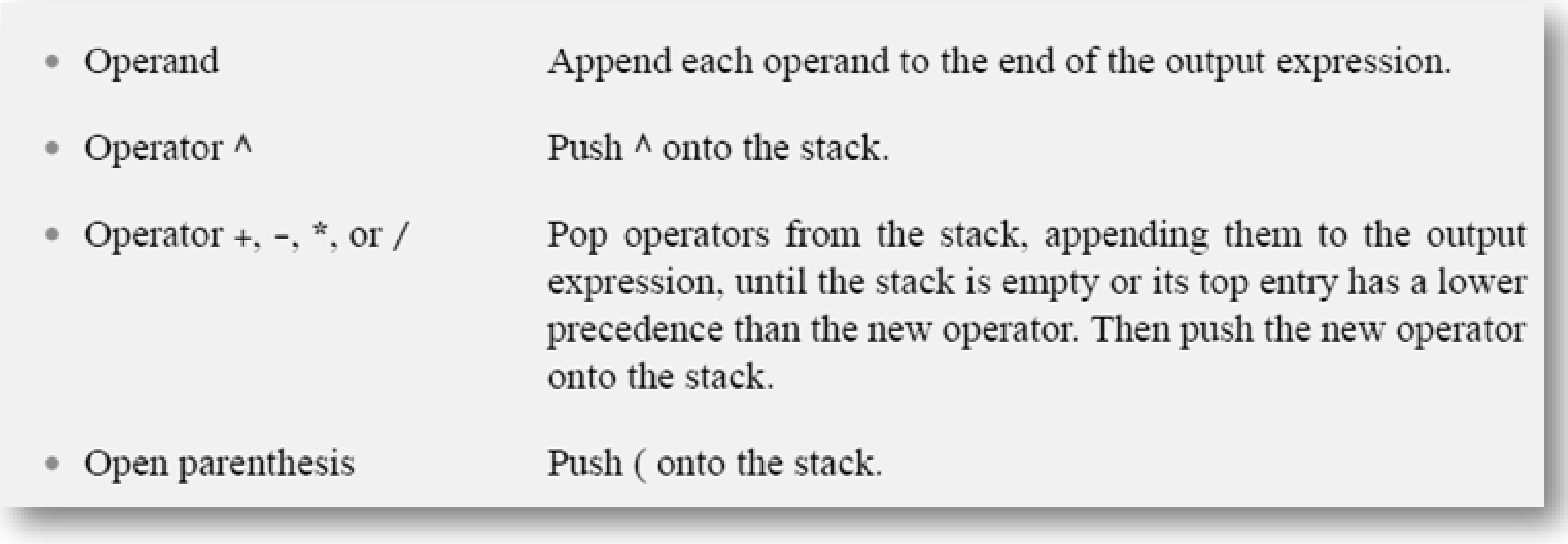
- for each character in the input expression



• Operand	Append each operand to the end of the output expression.
• Operator ^	Push ^ onto the stack.
• Operator +, -, *, or /	Pop operators from the stack, appending them to the output expression, until the stack is empty or its top entry has a lower precedence than the new operator. Then push the new operator onto the stack.

Infix-to-postfix Conversion Algorithm

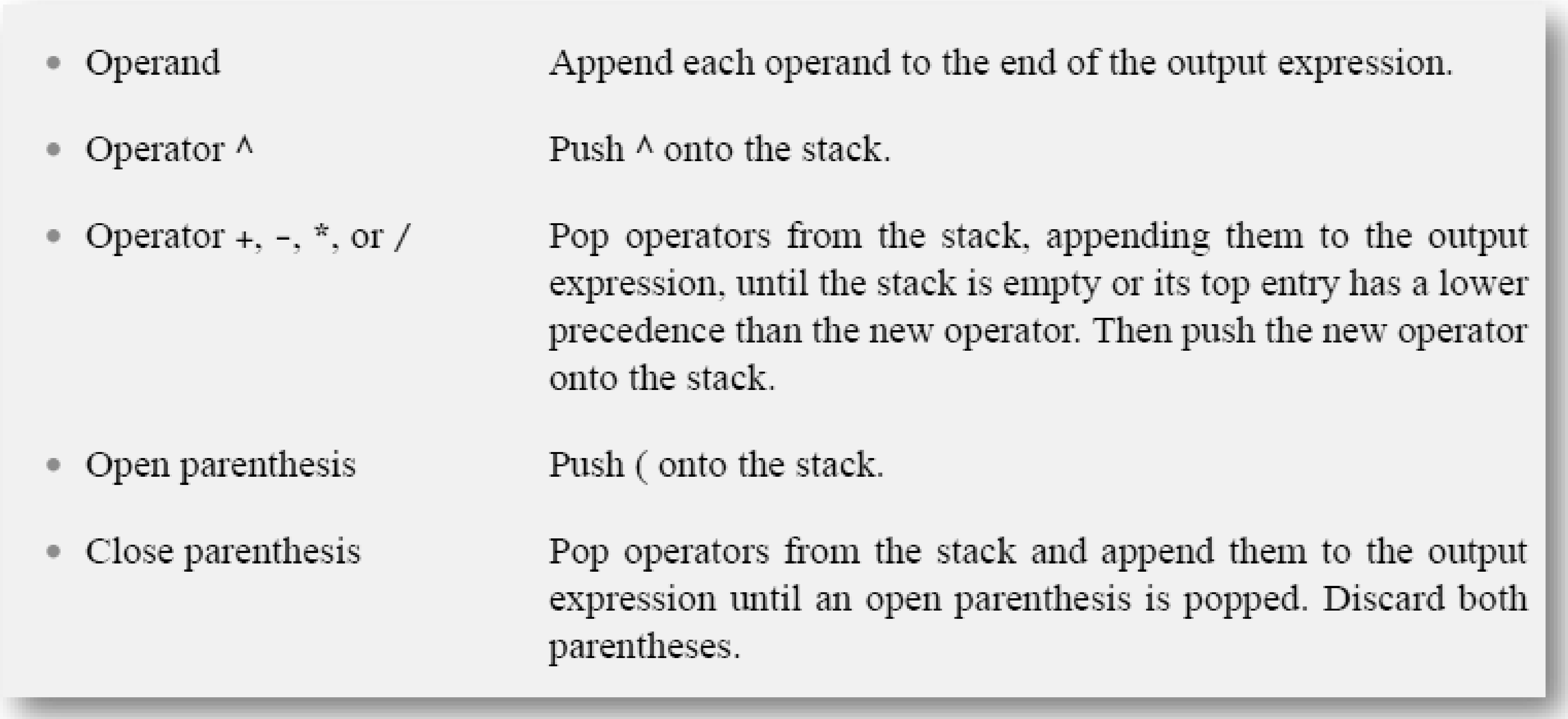
- for each character in the input expression



• Operand	Append each operand to the end of the output expression.
• Operator ^	Push ^ onto the stack.
• Operator +, -, *, or /	Pop operators from the stack, appending them to the output expression, until the stack is empty or its top entry has a lower precedence than the new operator. Then push the new operator onto the stack.
• Open parenthesis	Push (onto the stack.

Infix-to-postfix Conversion Algorithm

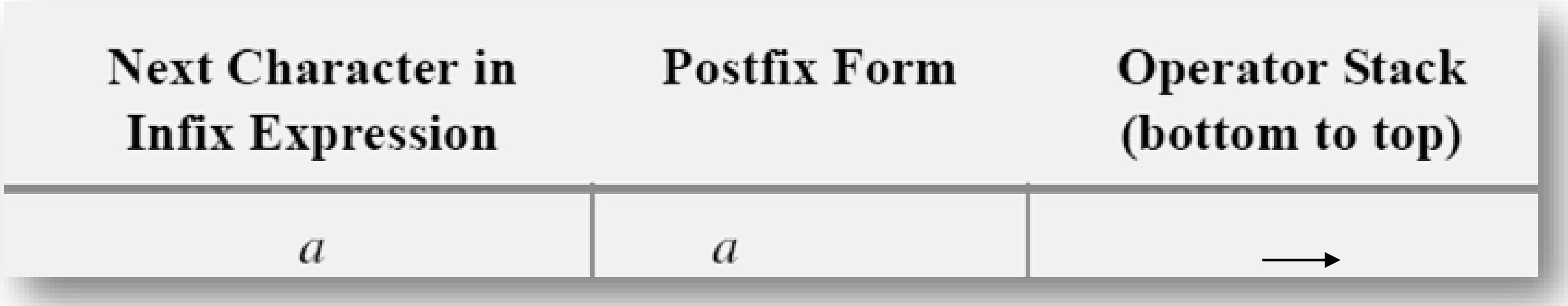
- for each character in the input expression



- Operand — Append each operand to the end of the output expression.
- Operator ^ — Push ^ onto the stack.
- Operator +, -, *, or / — Pop operators from the stack, appending them to the output expression, until the stack is empty or its top entry has a lower precedence than the new operator. Then push the new operator onto the stack.
- Open parenthesis — Push (onto the stack.
- Close parenthesis — Pop operators from the stack and append them to the output expression until an open parenthesis is popped. Discard both parentheses.

Infix to Postfix: Example 1

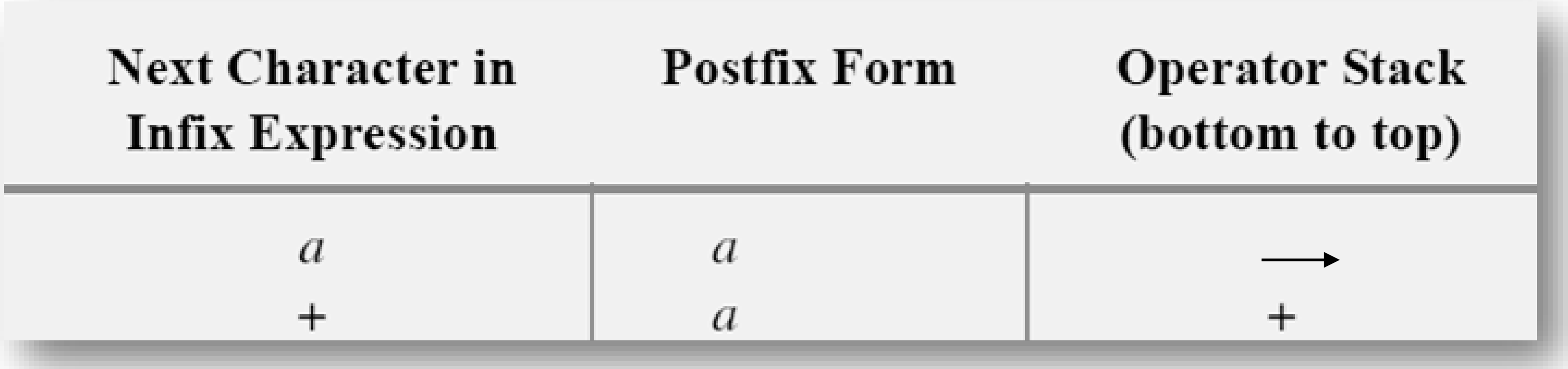
Converting the infix expression *a* + *b* * *c* to postfix form



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	→

Infix to Postfix: Example 1

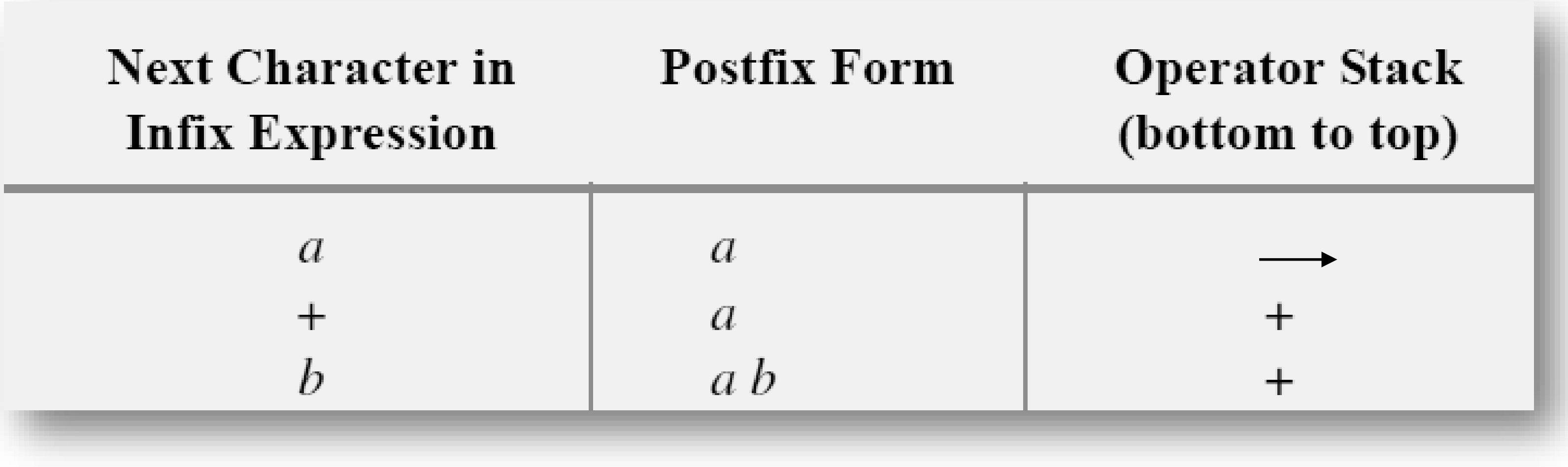
Converting the infix expression *a* + *b* * *c* to postfix form



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	→
+	*a*	+

Infix to Postfix: Example 1

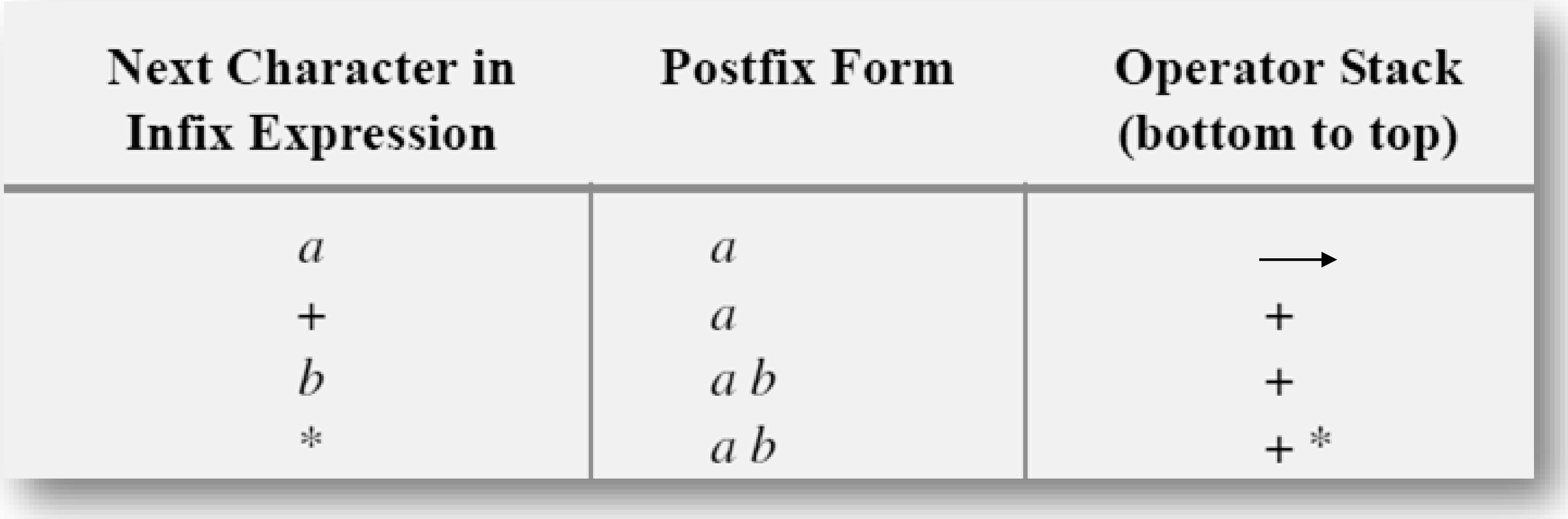
Converting the infix expression *a* + *b* * *c* to postfix form



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	→
+	*a*	+
b	*a b*	+

Infix to Postfix: Example 1

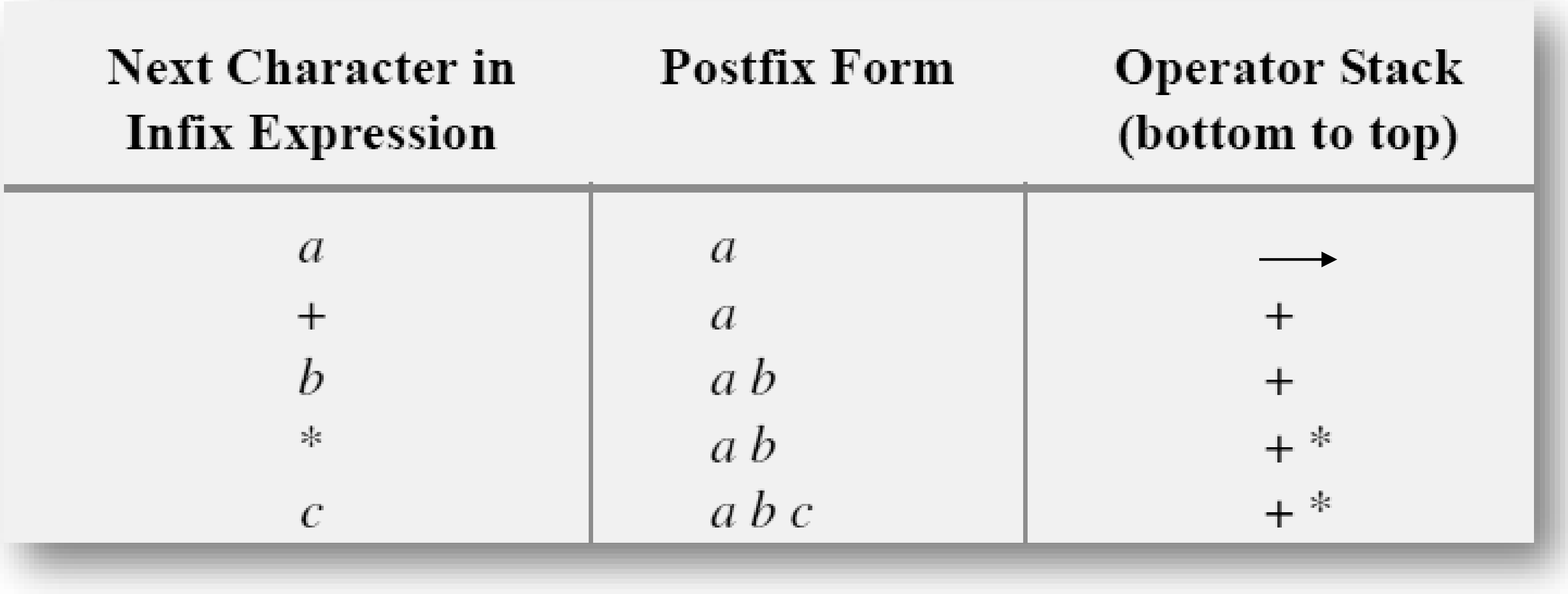
Converting the infix expression *a* + *b* * *c* to postfix form



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	→
+	*a*	+
b	*a b*	+
*	*a b*	+ *

Infix to Postfix: Example 1

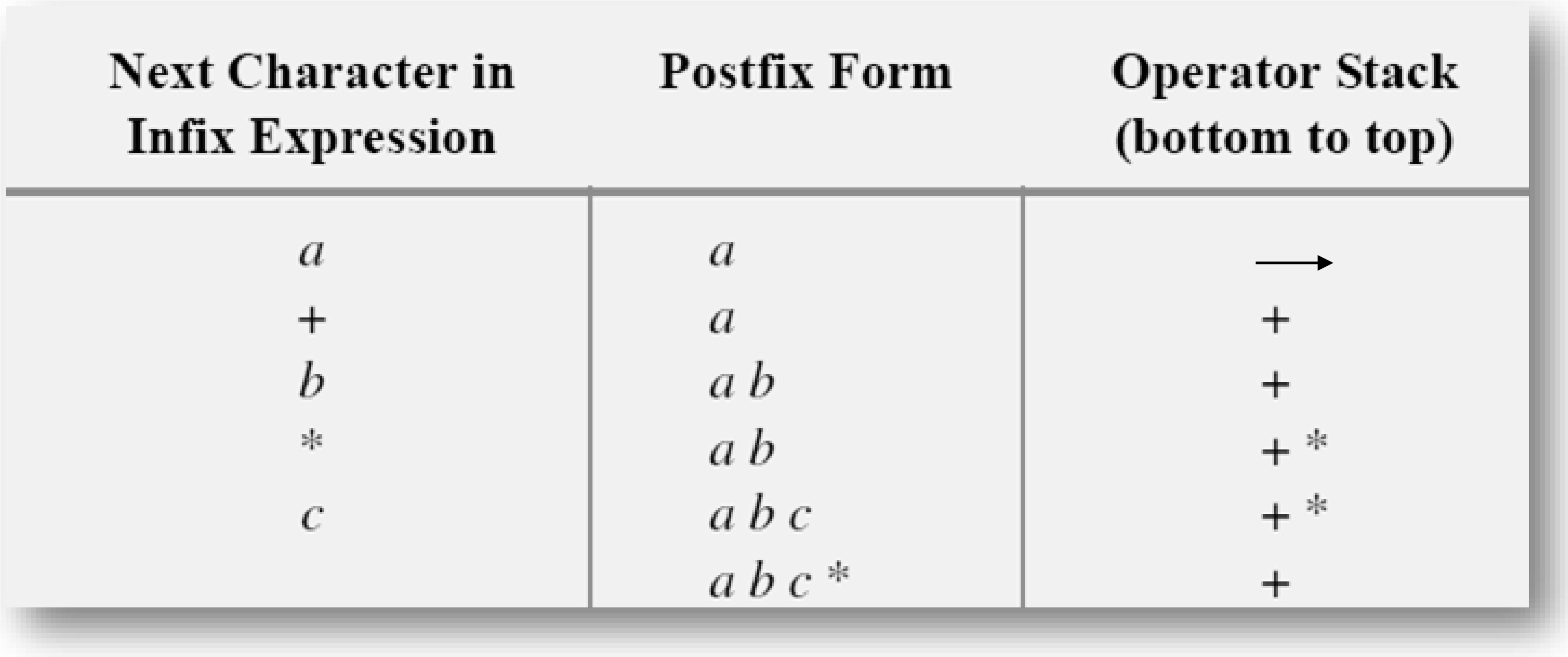
Converting the infix expression *a* + *b* * *c* to postfix form



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	a	⟶
$+$	a	$+$
b	$a\ b$	$+$
$*$	$a\ b$	$+\ *$
c	$a\ b\ c$	$+\ *$

Infix to Postfix: Example 1

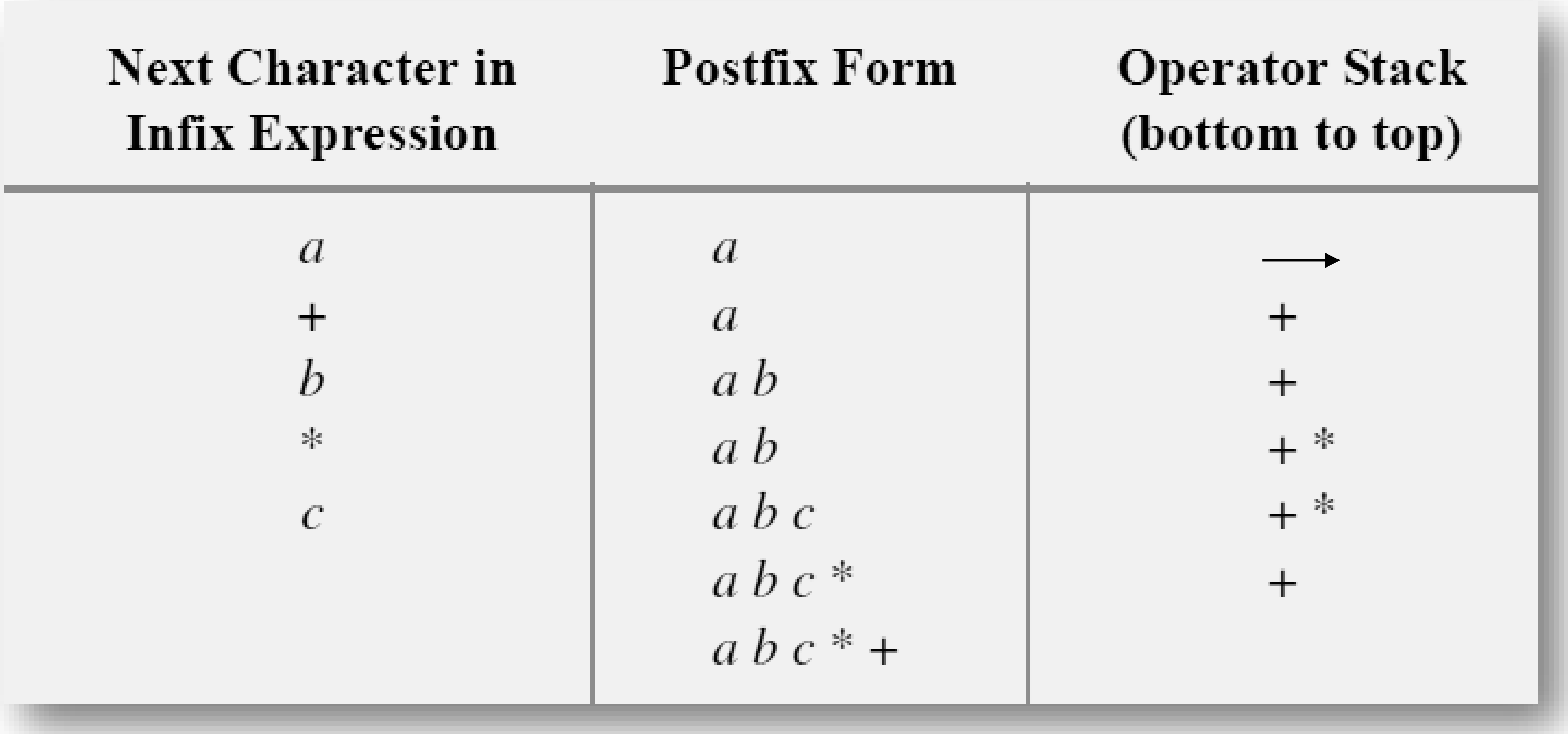
Converting the infix expression *a* + *b* * *c* to postfix form



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	→
+	*a*	+
b	*a b*	+
*	*a b*	+ *
c	*a b c*	+ *
	a b c *	+

Infix to Postfix: Example 1

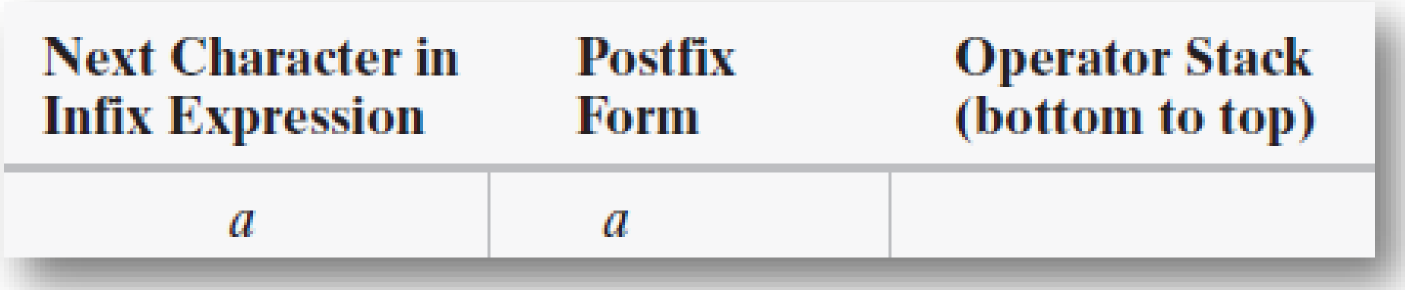
Converting the infix expression *a* + *b* * *c* to postfix form



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	→
+	*a*	+
b	*a b*	+
*	*a b*	+ *
c	*a b c*	+ *
	a b c *	+
	a b c * +	

Example with Successive Operators with Same Precedence

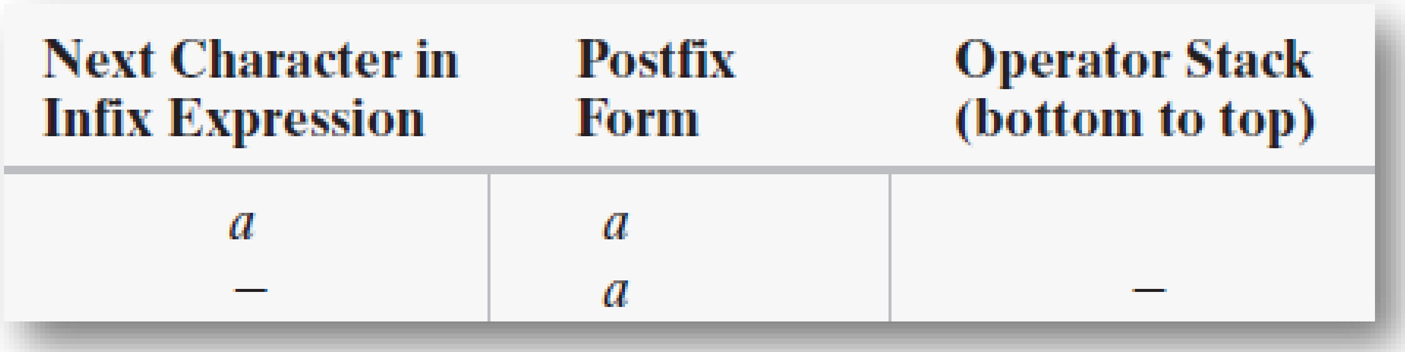
Converting an infix expression
to postfix form: *a - b + c*



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	

Example with Successive Operators with Same Precedence

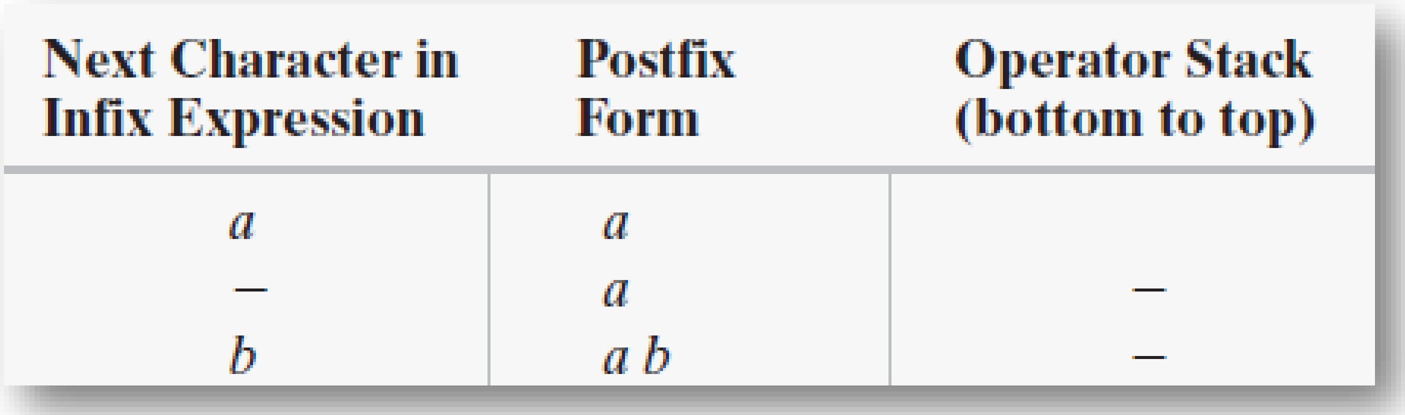
Converting an infix expression
to postfix form: *a* - *b* + *c*



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
–	*a*	–

Example with Successive Operators with Same Precedence

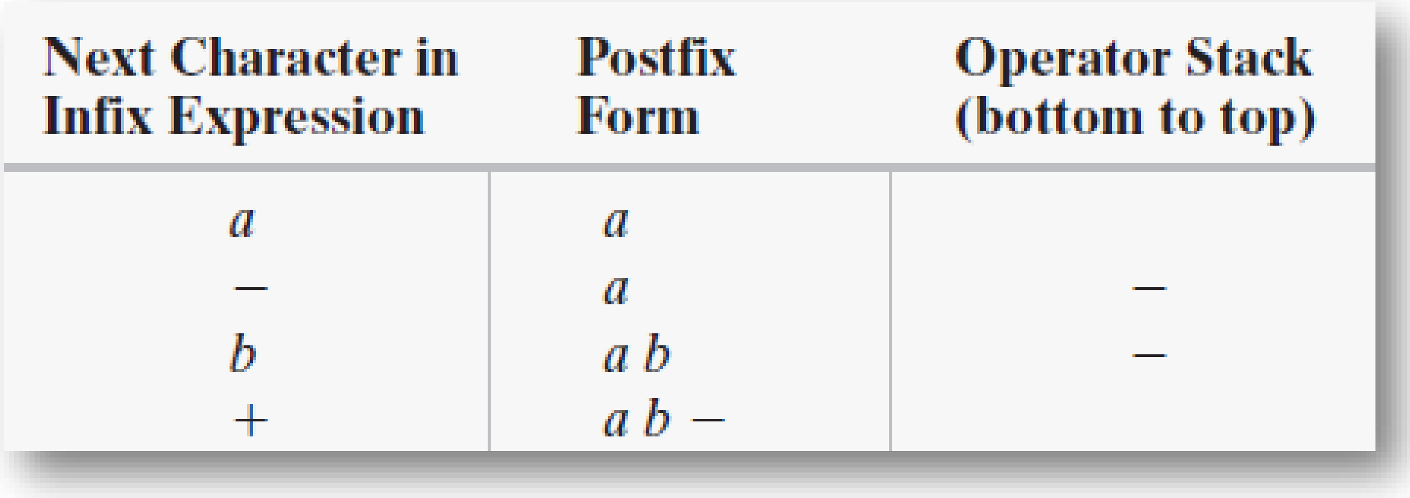
Converting an infix expression
to postfix form: *a* - *b* + *c*



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
–	*a*	–
b	*a b*	–

Example with Successive Operators with Same Precedence

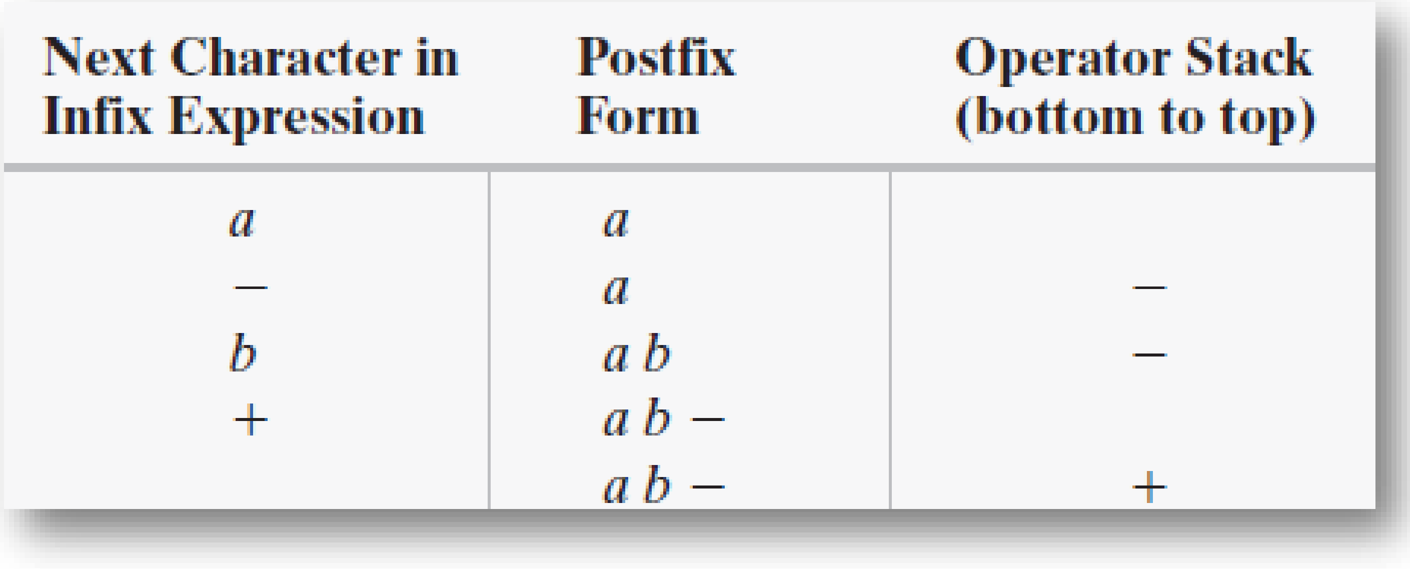
Converting an infix expression
to postfix form: *a* - *b* + *c*



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
−	*a*	−
b	*a b*	−
+	*a b* −	

Example with Successive Operators with Same Precedence

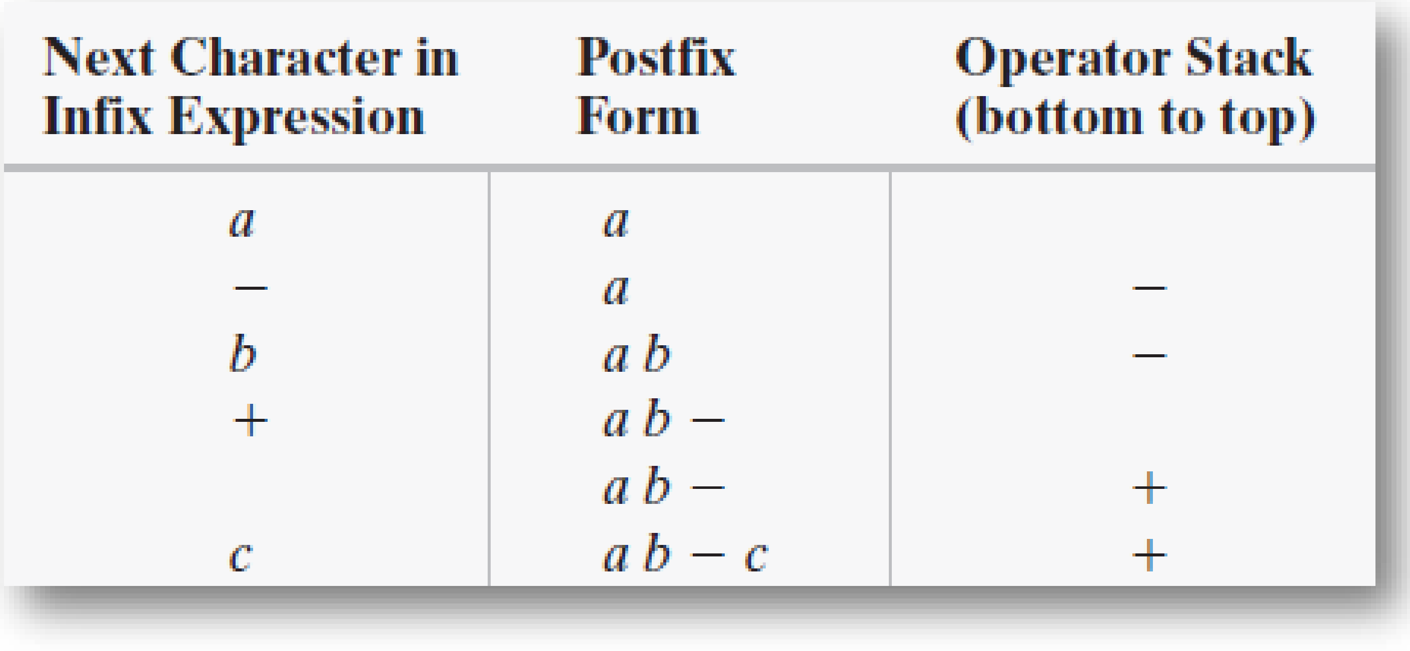
Converting an infix expression
to postfix form: *a* - *b* + *c*



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
–	*a*	–
b	*a b*	–
+	*a b* –	
	a b –	+

Example with Successive Operators with Same Precedence

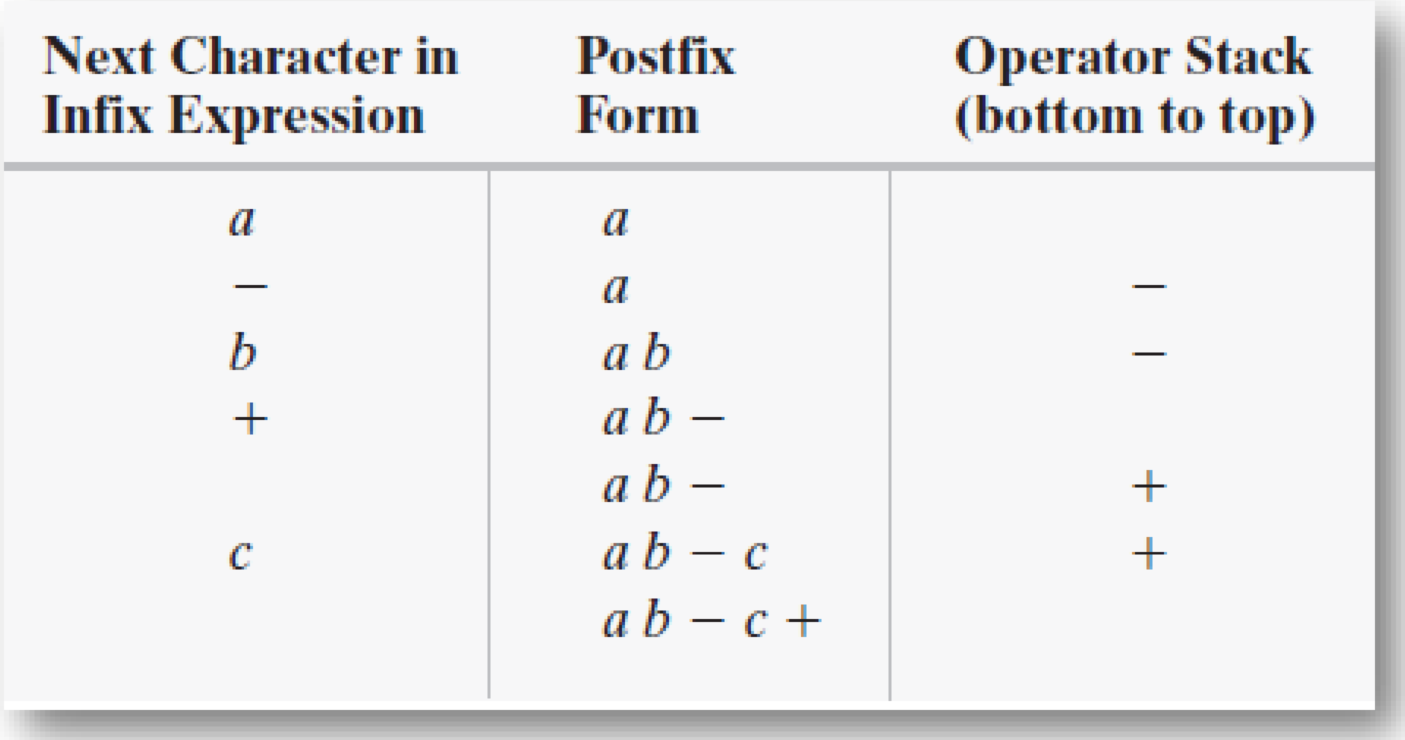
Converting an infix expression to postfix form: *a - b + c*



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
−	*a*	−
b	*a b*	−
+	*a b* −	
	a b −	+
c	*a b* − *c*	+

Example with Successive Operators with Same Precedence

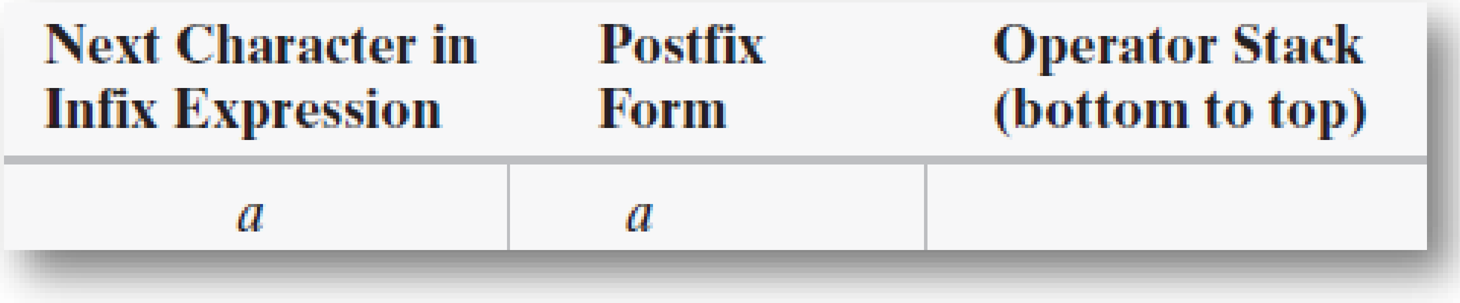
Converting an infix expression
to postfix form: *a - b + c*



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	a	
$-$	a	$-$
b	$a\ b$	$-$
$+$	$a\ b\ -$	
	$a\ b\ -$	$+$
c	$a\ b\ -\ c$	$+$
	$a\ b\ -\ c\ +$	

Another Example with Successive Operators with Same Precedence

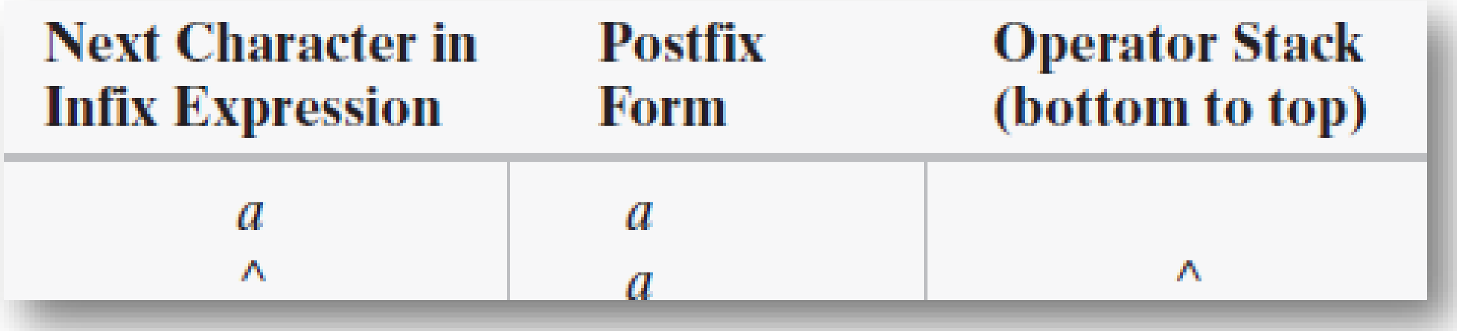
Converting an infix expression to postfix form: *a* ^ *b* ^ *c*



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	

Another Example with Successive Operators with Same Precedence

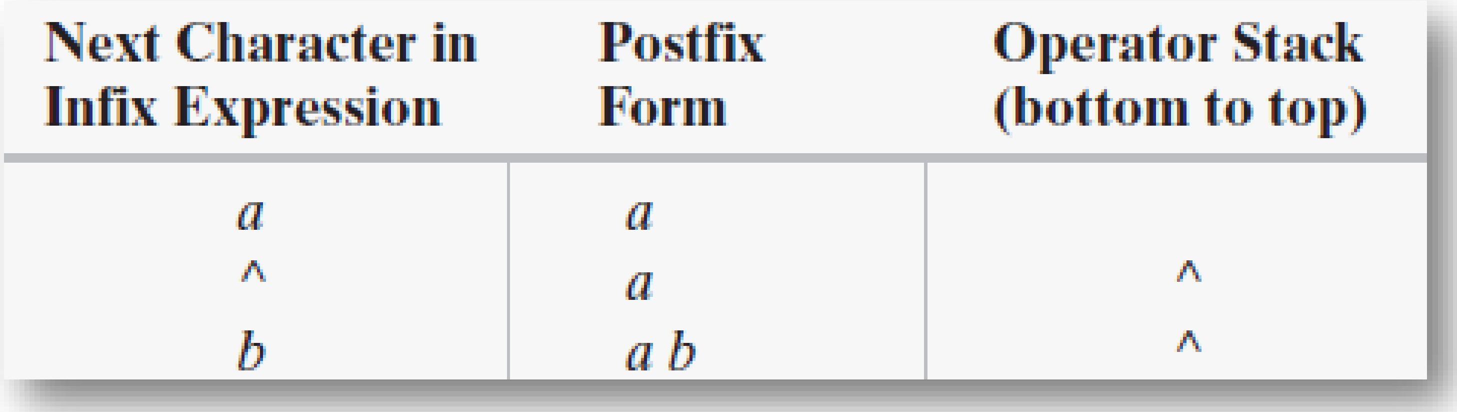
Converting an infix expression to postfix form: *a* ^ *b* ^ *c*



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
^	*a*	^

Another Example with Successive Operators with Same Precedence

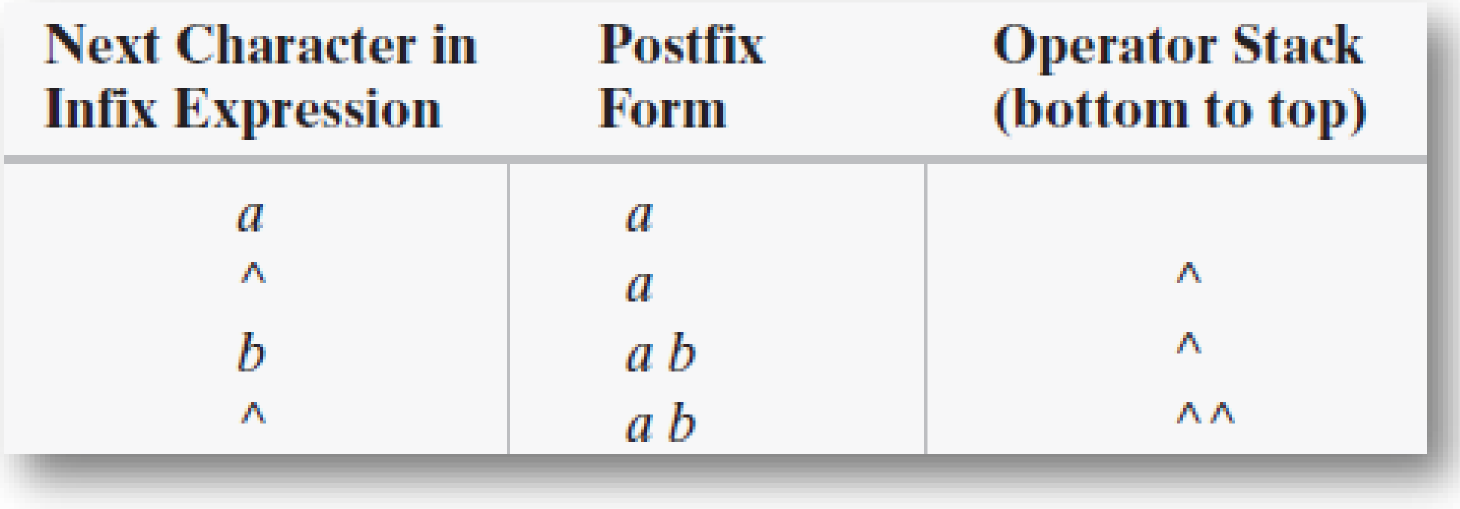
Converting an infix expression
to postfix form: *a* ^ *b* ^ *c*



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
^	*a*	^
b	*a b*	^

Another Example with Successive Operators with Same Precedence

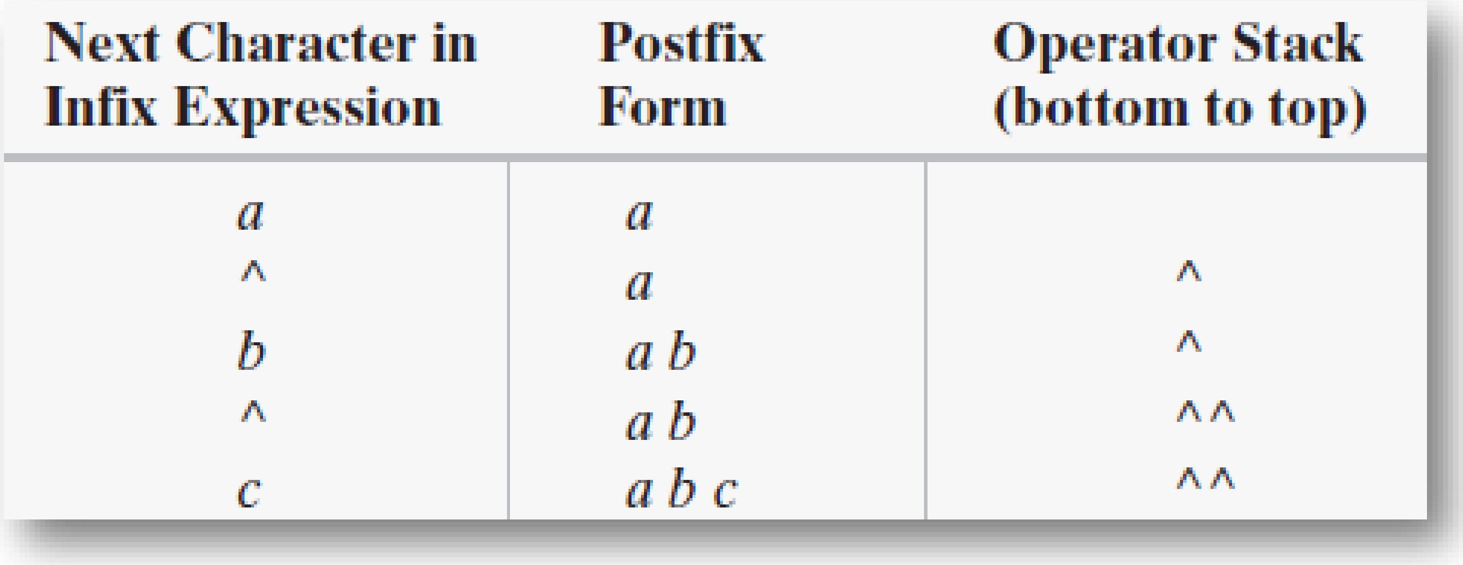
Converting an infix expression
to postfix form: *a* ^ *b* ^ *c*



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
^	*a*	^
b	*a b*	^
^	*a b*	^ ^

Another Example with Successive Operators with Same Precedence

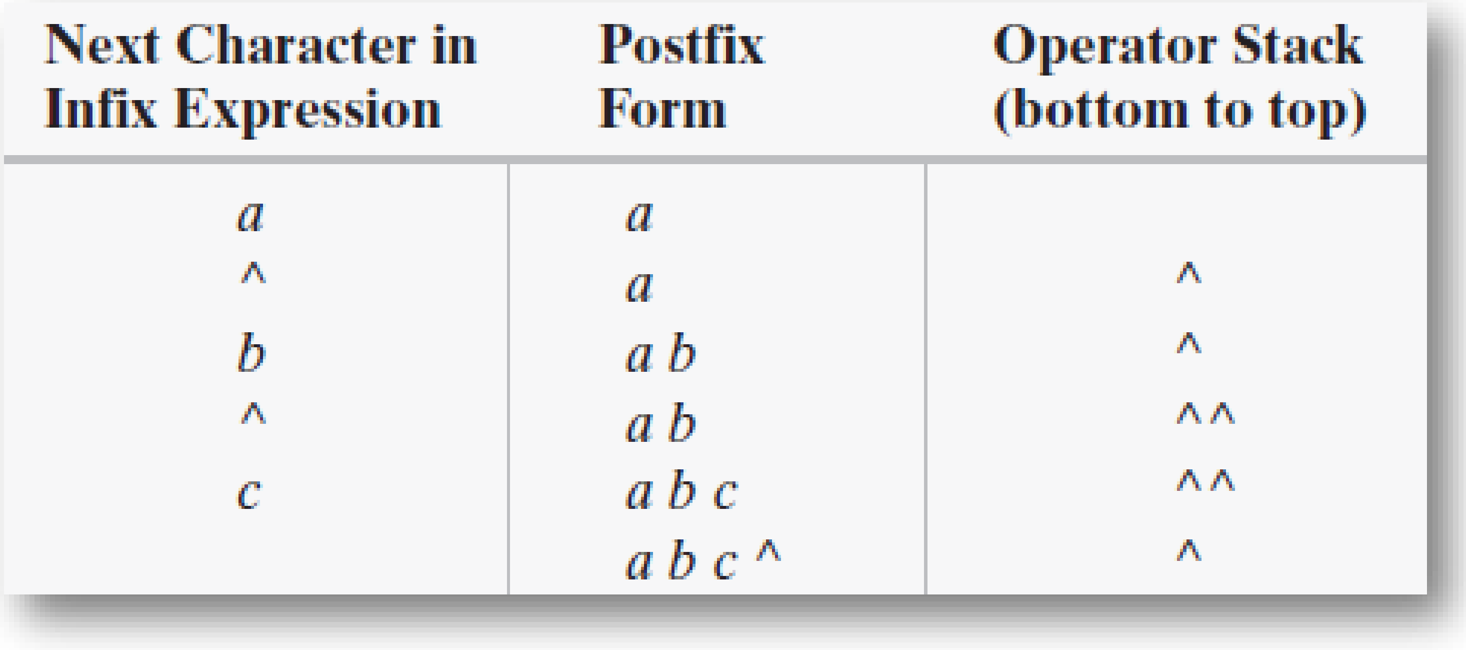
Converting an infix expression
to postfix form: *a* ^ *b* ^ *c*



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
^	*a*	^
b	*a b*	^
^	*a b*	^ ^
c	*a b c*	^ ^

Another Example with Successive Operators with Same Precedence

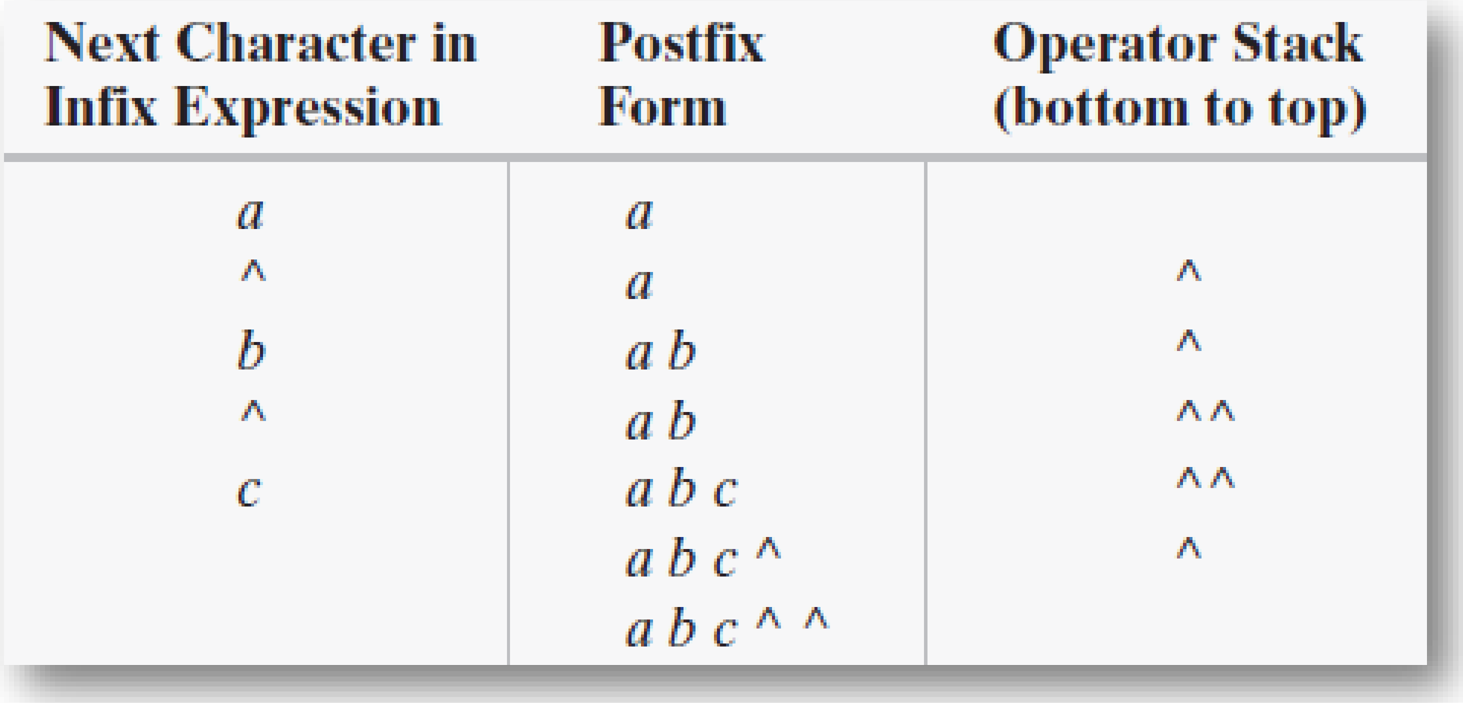
Converting an infix expression
to postfix form: *a* ^ *b* ^ *c*



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
^	*a*	^
b	*a b*	^
^	*a b*	^ ^
c	*a b c*	^ ^
	a b c ^	^

Another Example with Successive Operators with Same Precedence

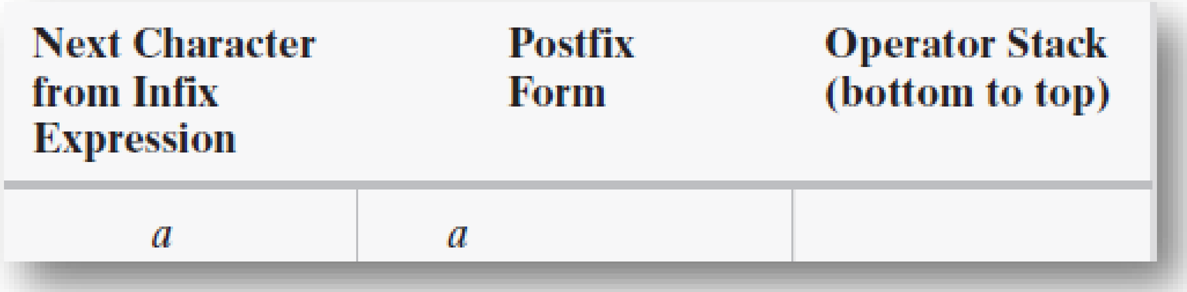
Converting an infix expression
to postfix form: *a* ^ *b* ^ *c*



Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
^	*a*	^
b	*a b*	^
^	*a b*	^ ^
c	*a b c*	^ ^
	a b c ^	^
	a b c ^ ^	

Infix to Postfix: Larger Example

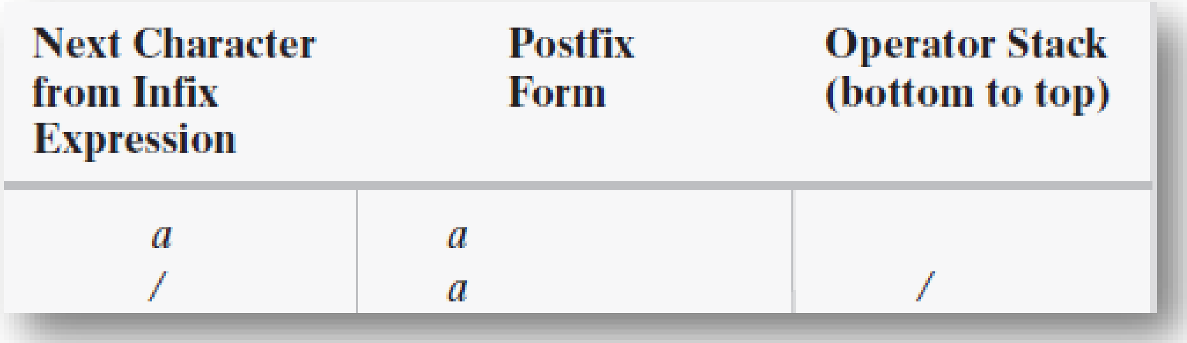
The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	

Infix to Postfix: Larger Example

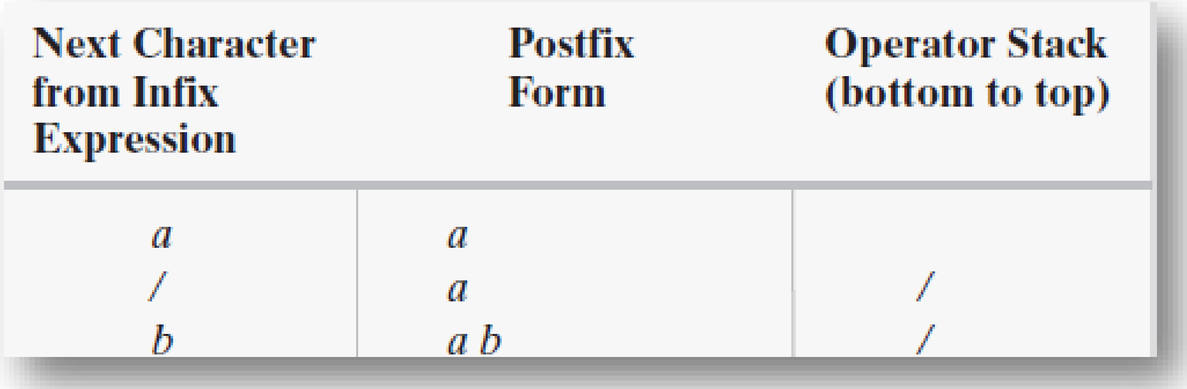
The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/

Infix to Postfix: Larger Example

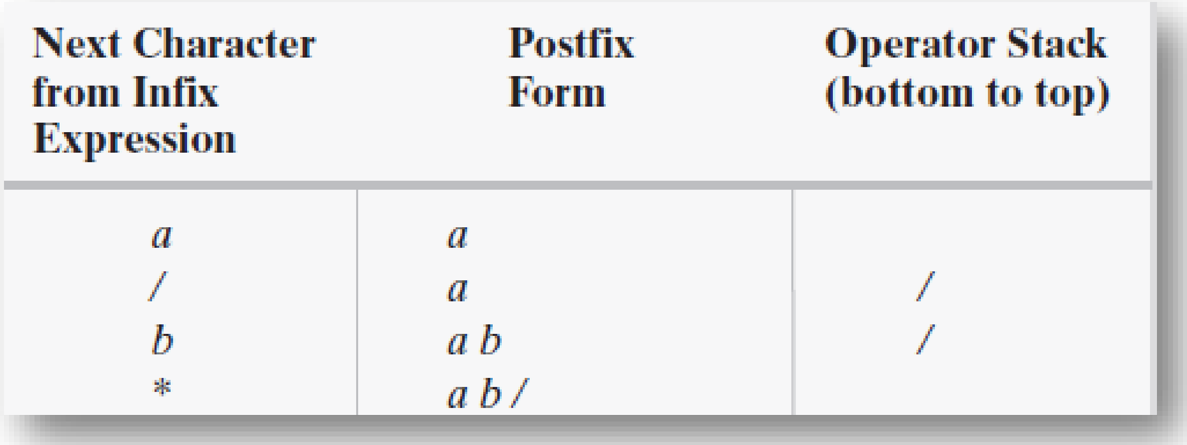
The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/

Infix to Postfix: Larger Example

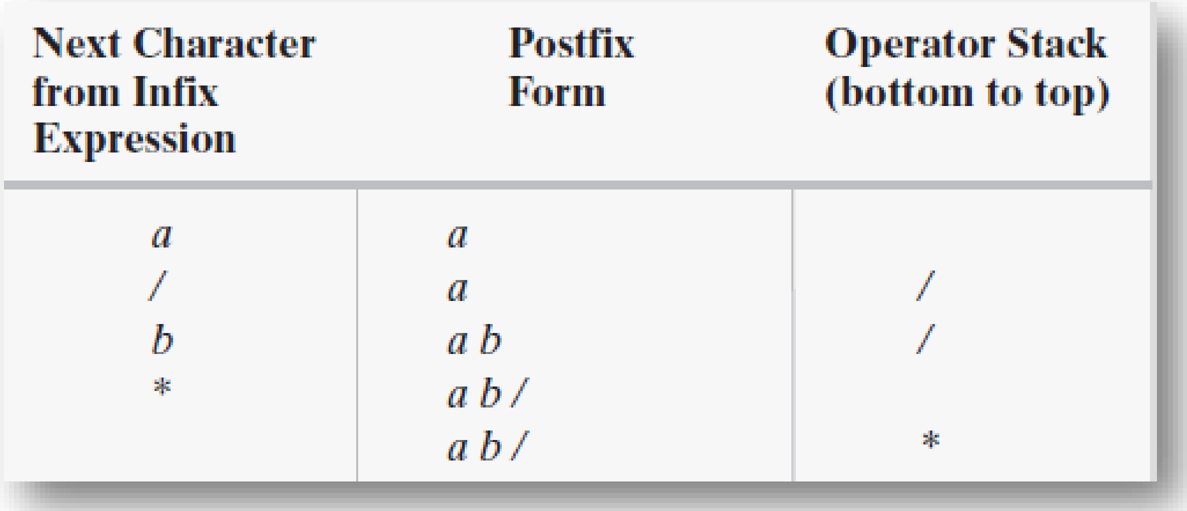
The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b* /	

Infix to Postfix: Larger Example

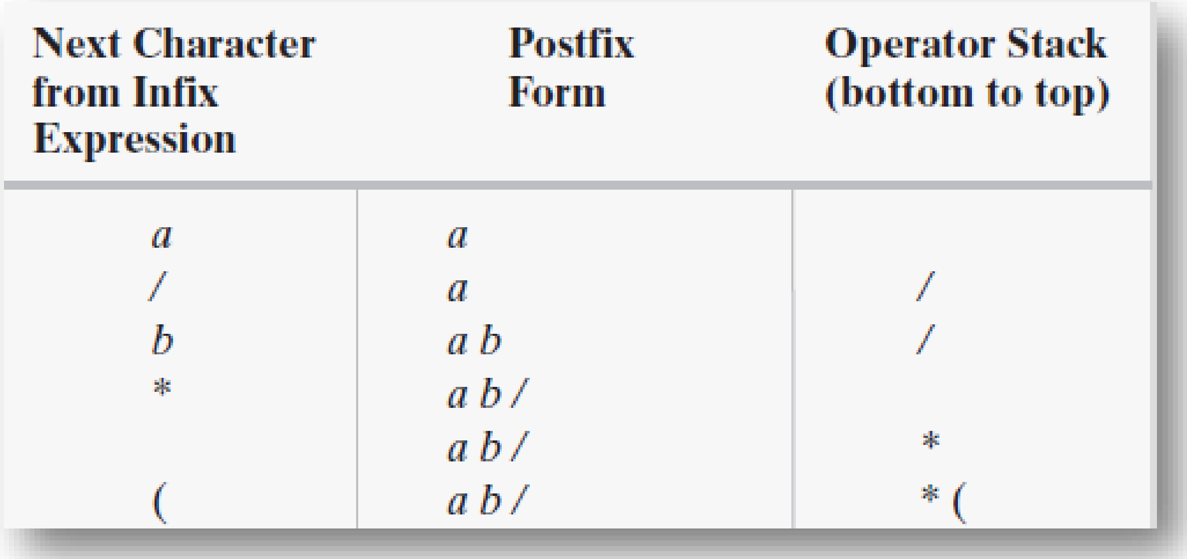
The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b /*	
	a b /	*

Infix to Postfix: Larger Example

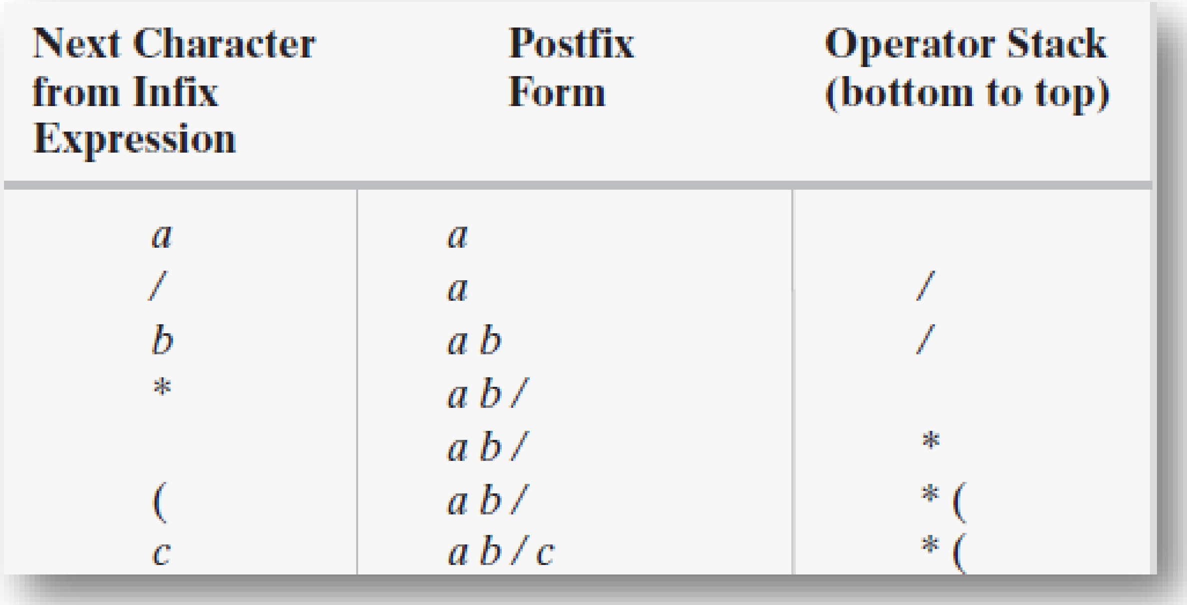
The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b* /	
	a b /	*
(	*a b* /	* (

Infix to Postfix: Larger Example

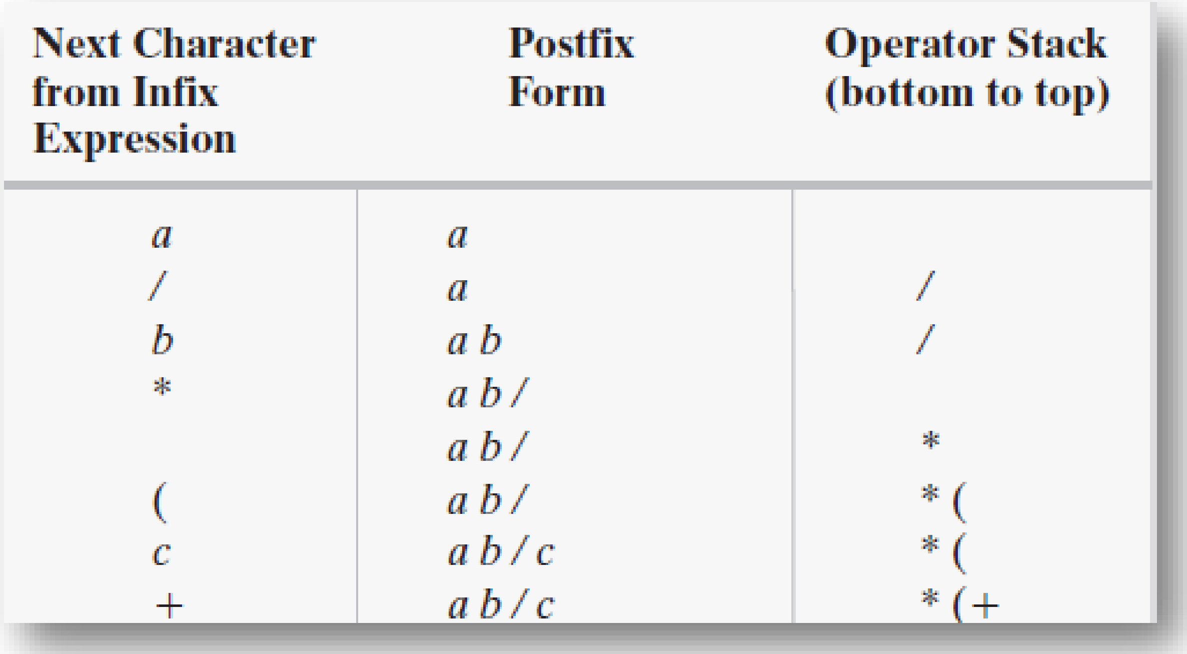
The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b* /	
	a b /	*
(	*a b* /	* (
c	*a b* / *c*	* (

Infix to Postfix: Larger Example

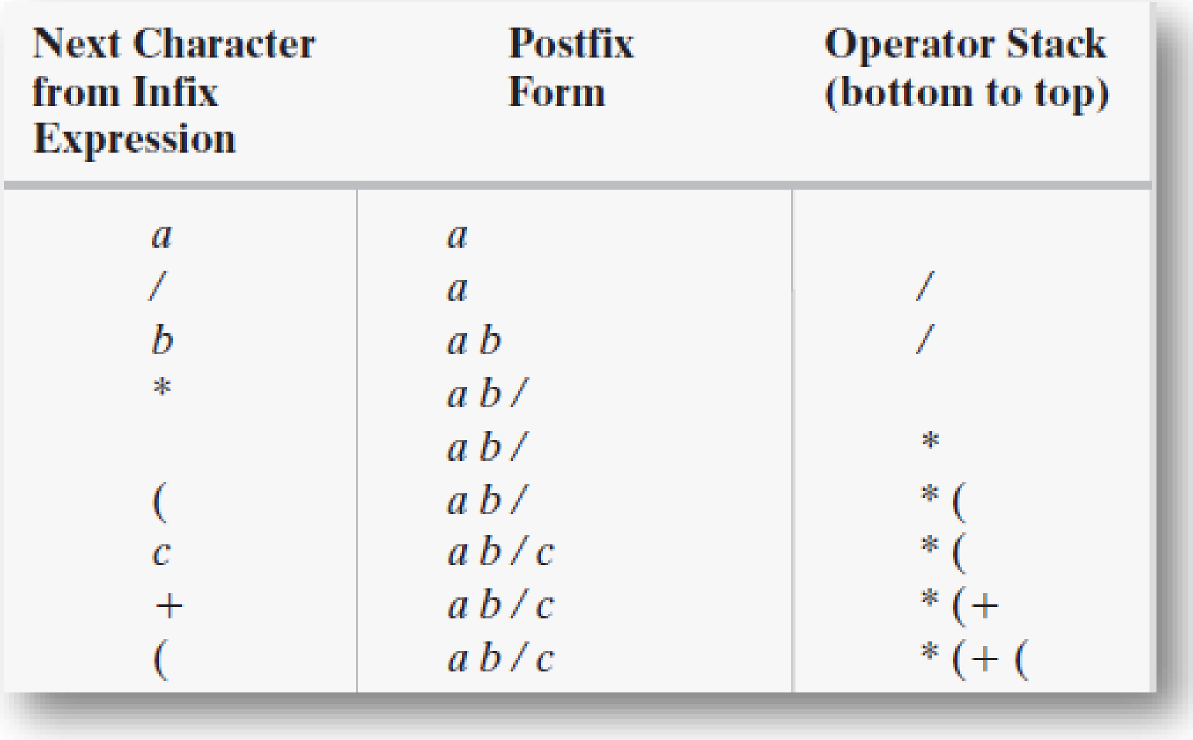
The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b /*	
	a b /	*
(	*a b /*	* (
c	*a b / c*	* (
+	*a b / c*	* (+

Infix to Postfix: Larger Example

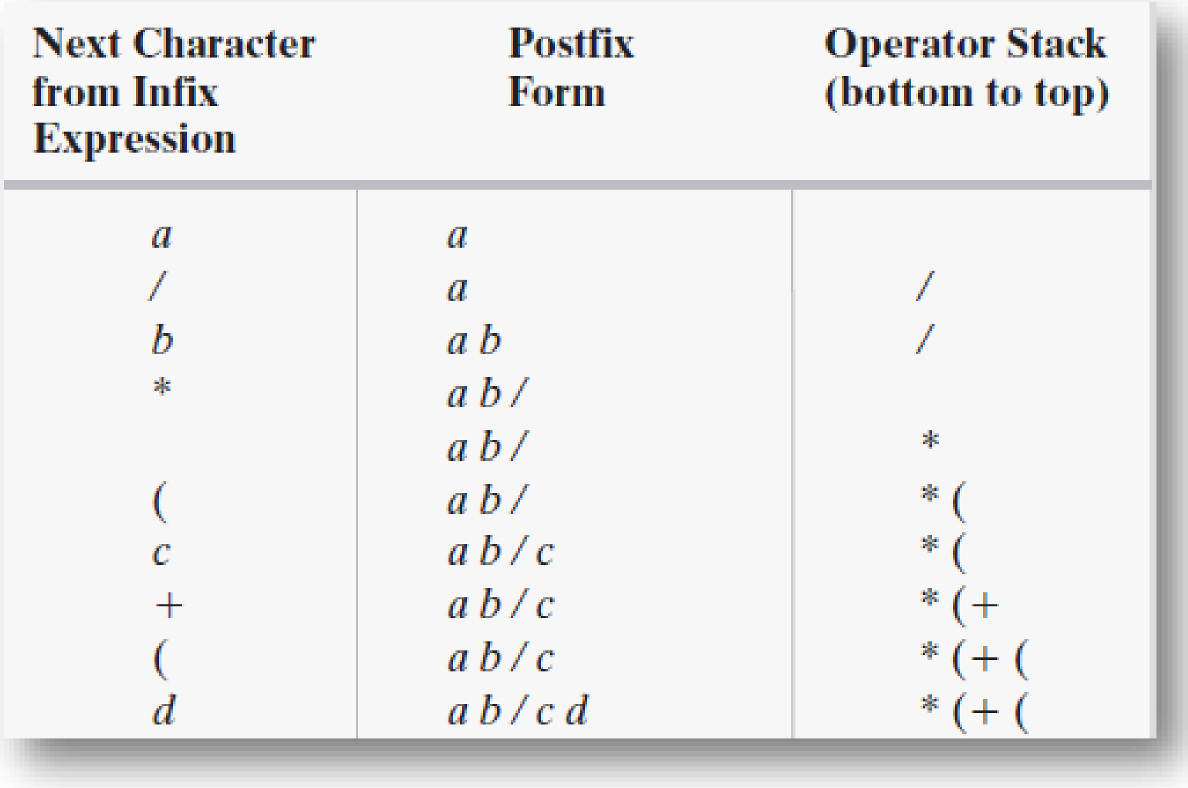
The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b* /	
	a b /	*
(	*a b* /	* (
c	*a b* / *c*	* (
+	*a b* / *c*	* (+
(	*a b* / *c*	* (+ (

Infix to Postfix: Larger Example

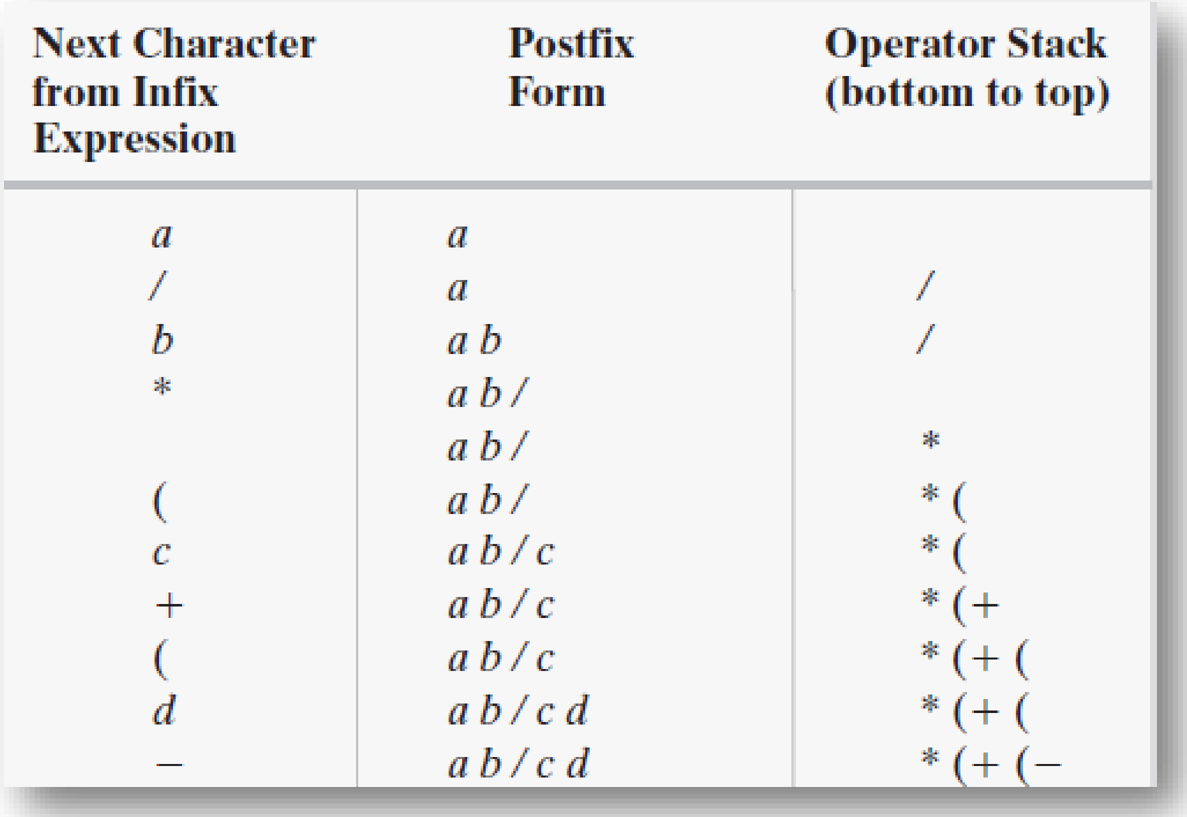
The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b* /	
	a b /	*
(	*a b* /	* (
c	*a b* / *c*	* (
+	*a b* / *c*	* (+
(	*a b* / *c*	* (+ (
d	*a b* / *c d*	* (+ (

Infix to Postfix: Larger Example

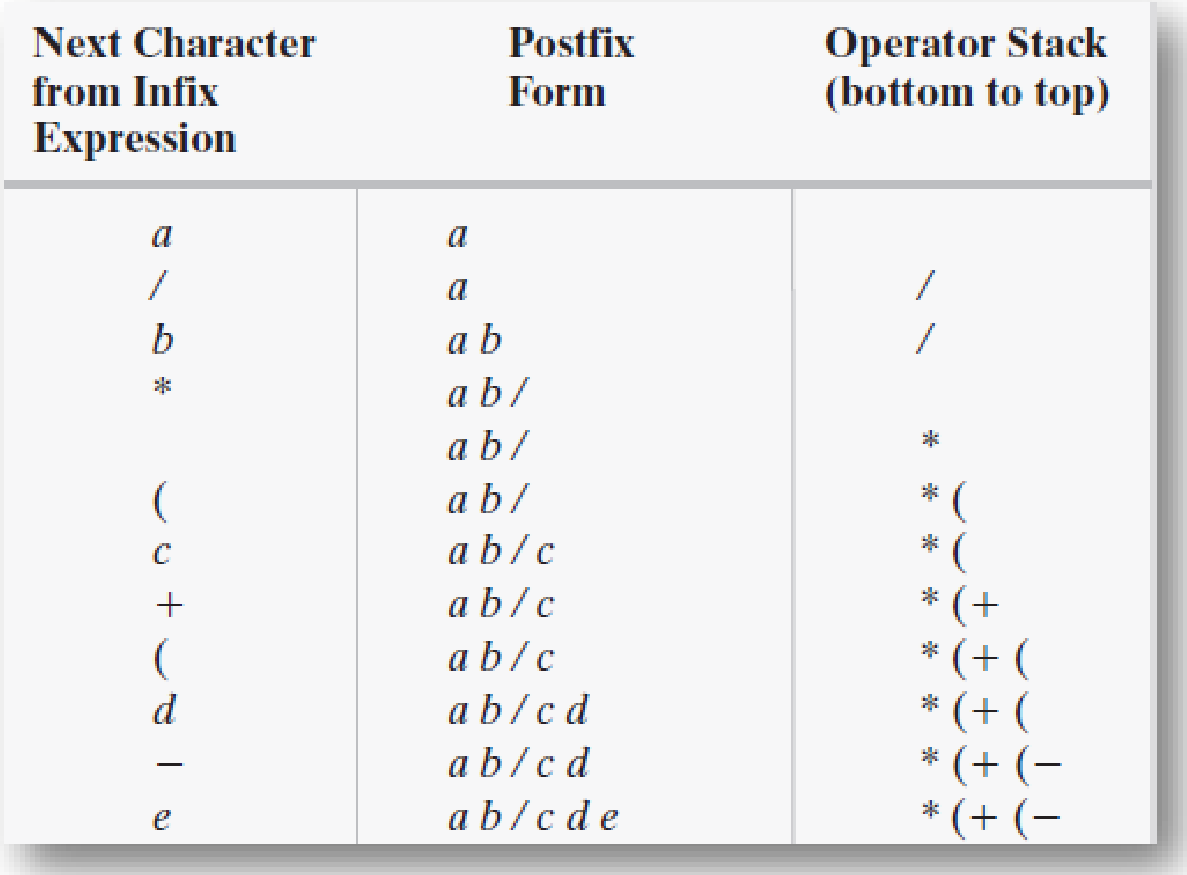
The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b* /	
	a b /	*
(	*a b* /	* (
c	*a b* / *c*	* (
+	*a b* / *c*	* (+
(	*a b* / *c*	* (+ (
d	*a b* / *c d*	* (+ (
–	*a b* / *c d*	* (+ (–

Infix to Postfix: Larger Example

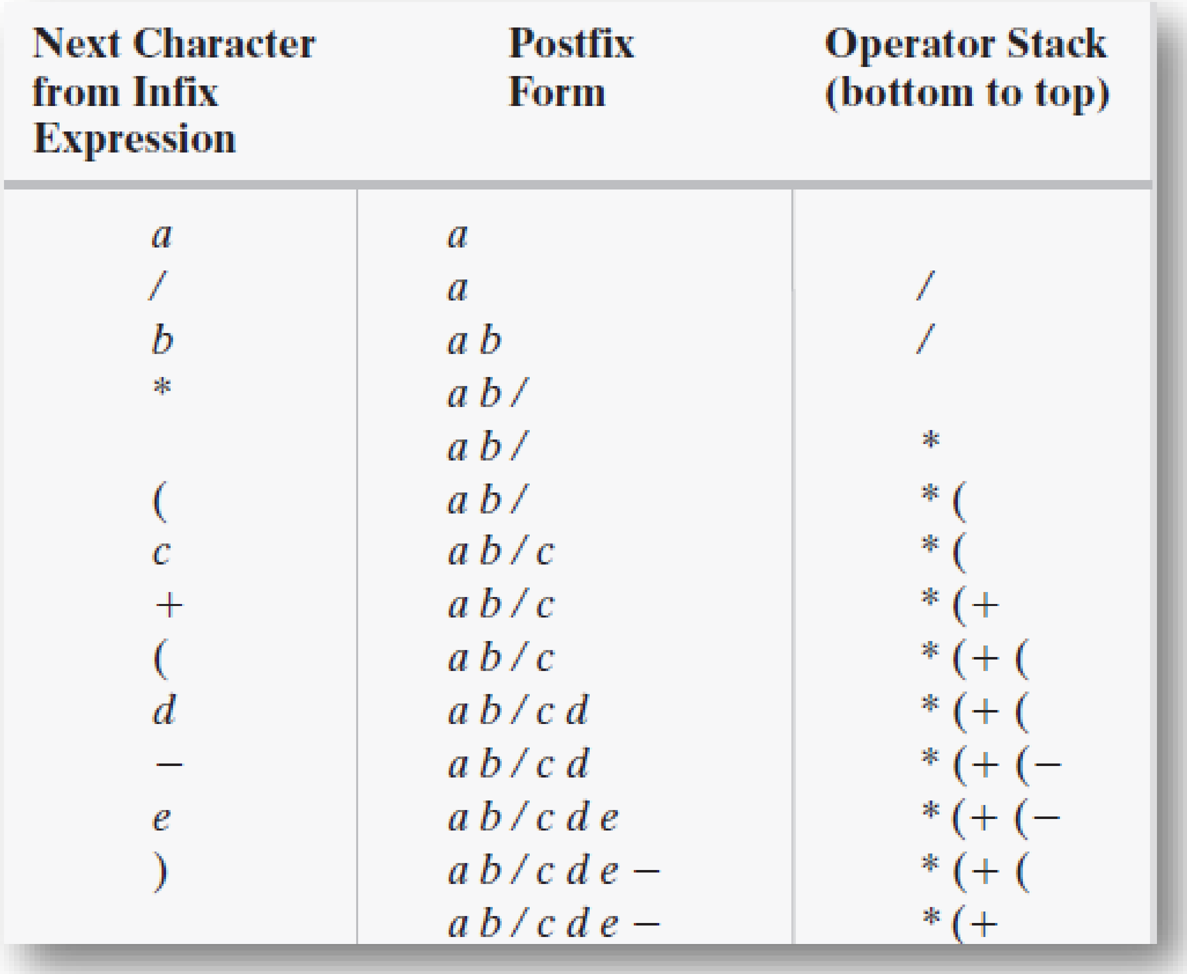
The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b* /	
	a b /	*
(	*a b* /	* (
c	*a b* / *c*	* (
+	*a b* / *c*	* (+
(	*a b* / *c*	* (+ (
d	*a b* / *c d*	* (+ (
–	*a b* / *c d*	* (+ (–
e	*a b* / *c d e*	* (+ (–

Infix to Postfix: Larger Example

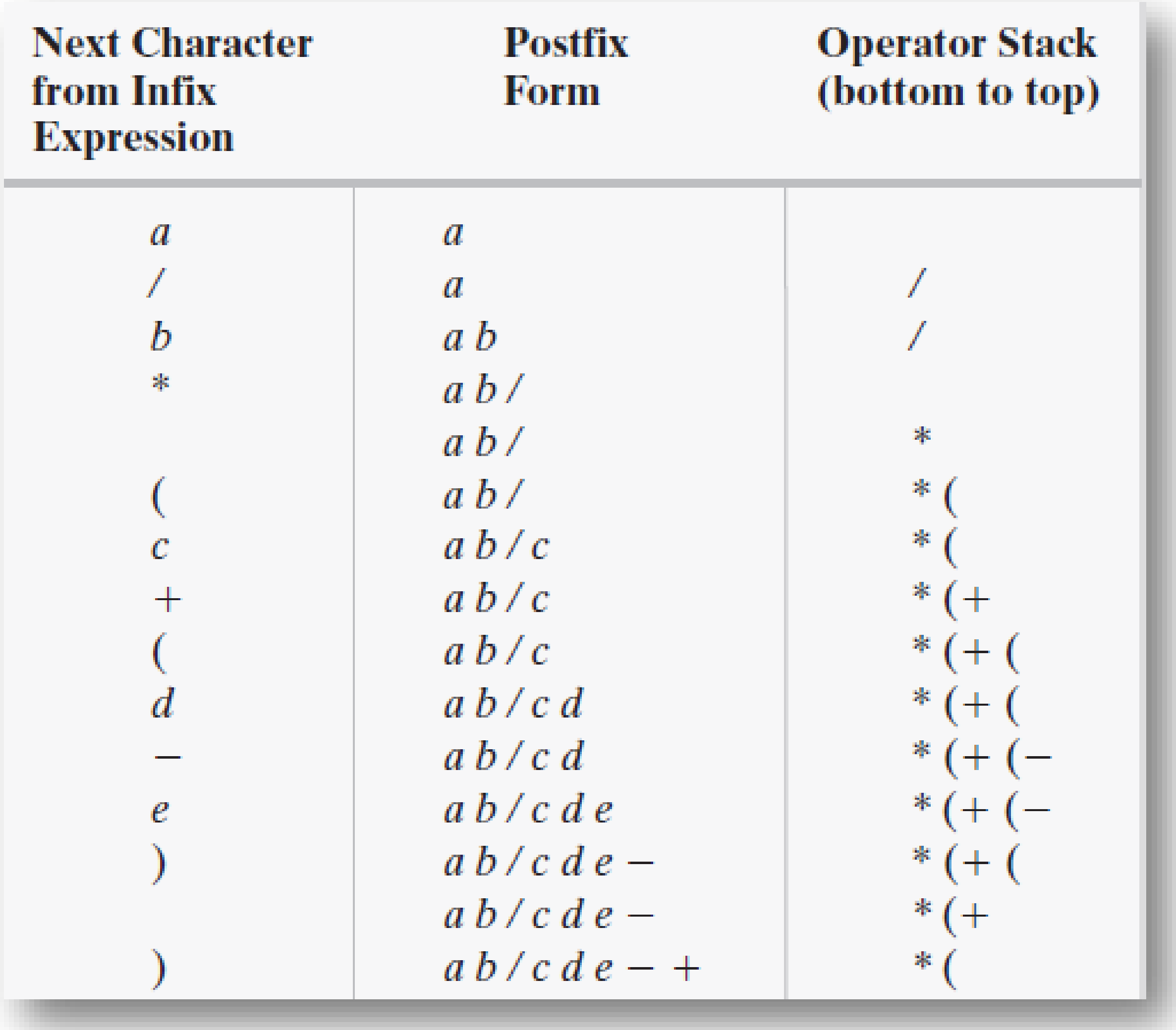
The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b* /	
	a b /	*
(	*a b* /	* (
c	*a b* / *c*	* (
+	*a b* / *c*	* (+
(	*a b* / *c*	* (+ (
d	*a b* / *c d*	* (+ (
–	*a b* / *c d*	* (+ (–
e	*a b* / *c d e*	* (+ (–
)	*a b* / *c d e* –	* (+ (
	a b / *c d e* –	* (+

Infix to Postfix: Larger Example

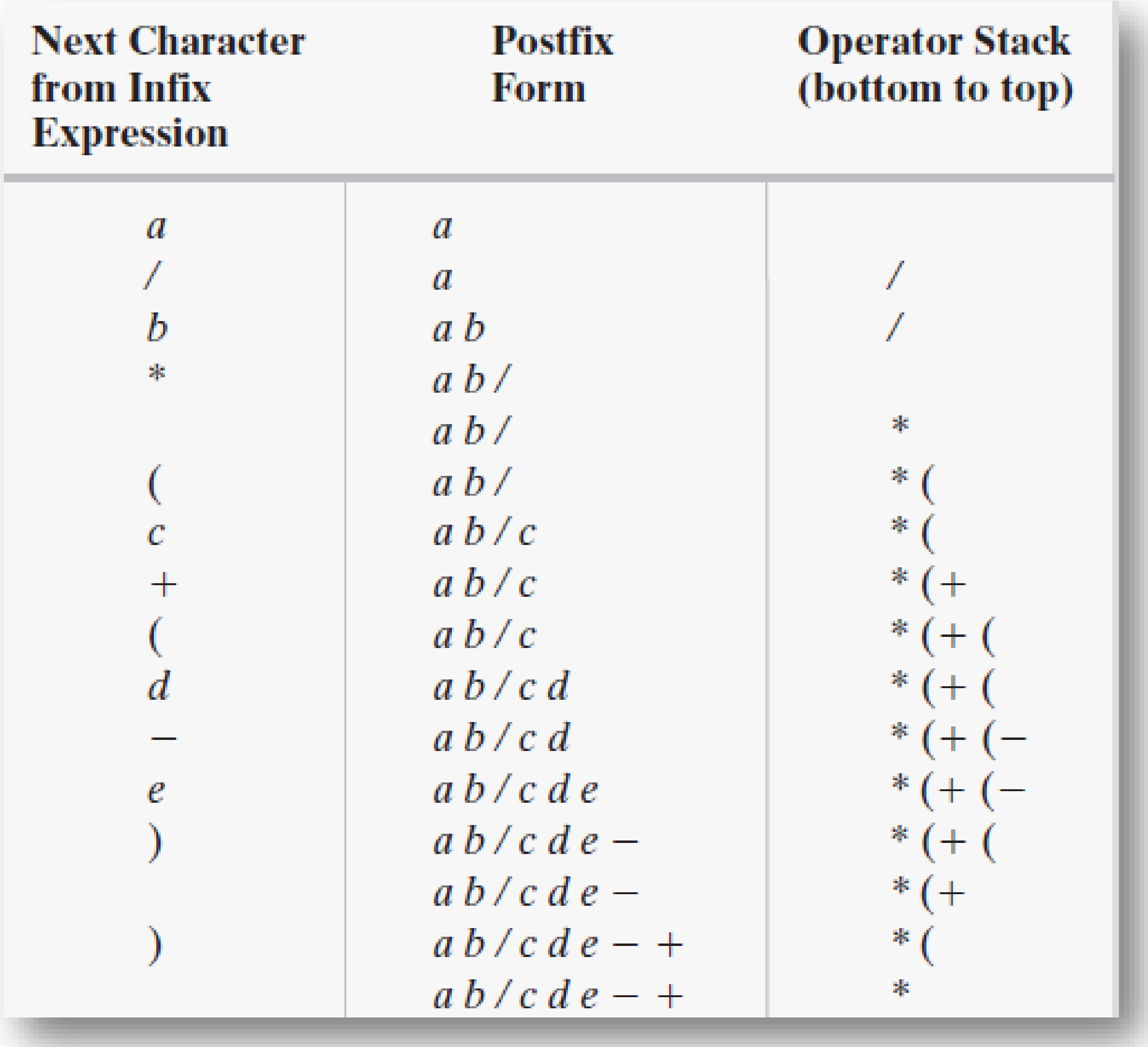
The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	a	
/	a	/
b	a b	/
*	a b /	
	a b /	*
(	a b /	* (
c	a b / c	* (
+	a b / c	* (+
(	a b / c	* (+ (
d	a b / c d	* (+ (
−	a b / c d	* (+ (−
e	a b / c d e	* (+ (−
)	a b / c d e −	* (+ (
	a b / c d e −	* (+
)	a b / c d e − +	* (

Infix to Postfix: Larger Example

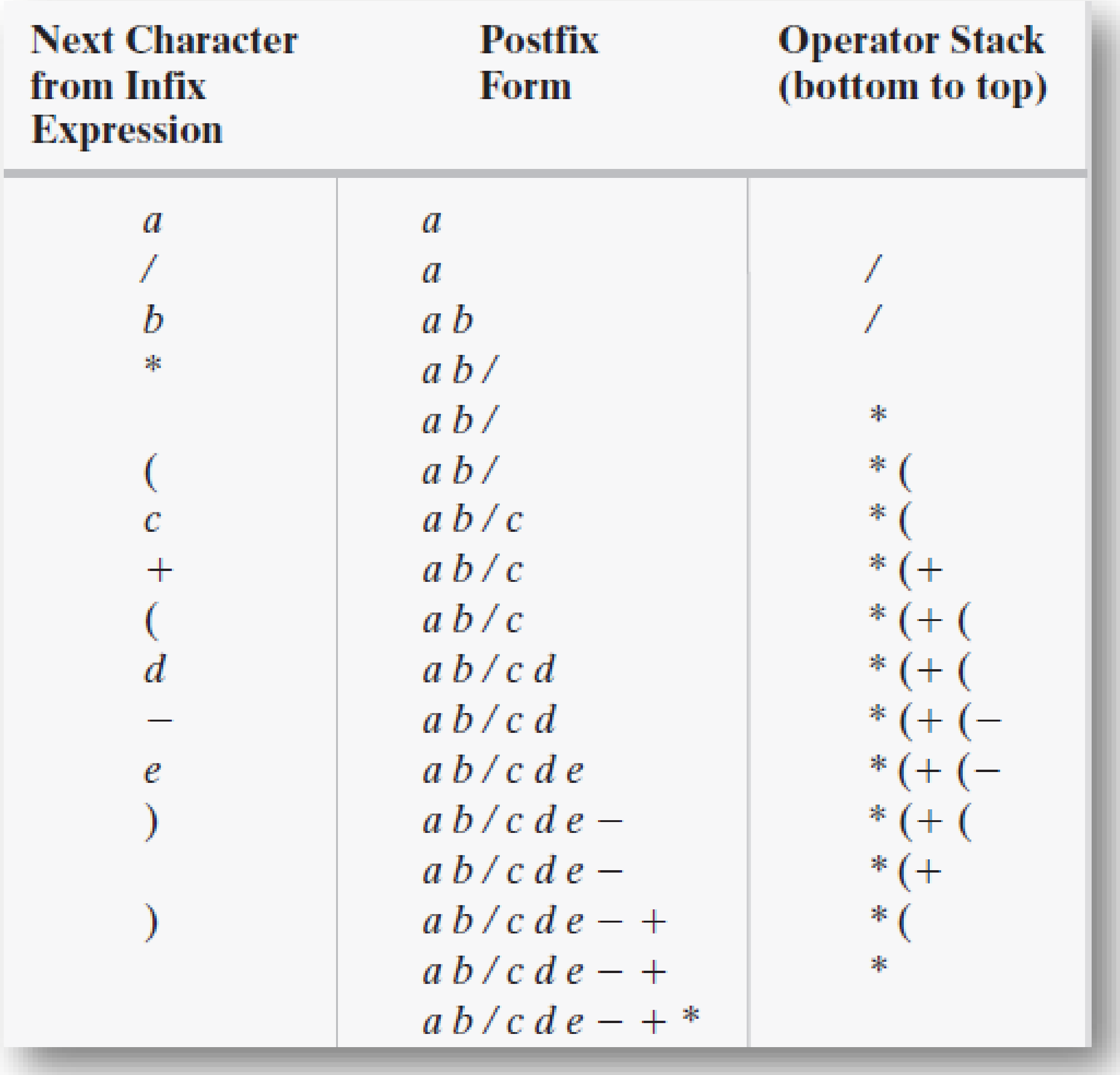
The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	a	
/	a	/
b	a b	/
*	a b /	
	a b /	*
(	a b /	* (
c	a b / c	* (
+	a b / c	* (+
(	a b / c	* (+ (
d	a b / c d	* (+ (
−	a b / c d	* (+ (−
e	a b / c d e	* (+ (−
)	a b / c d e −	* (+ (
	a b / c d e −	* (+
)	a b / c d e − +	* (
	a b / c d e − +	*

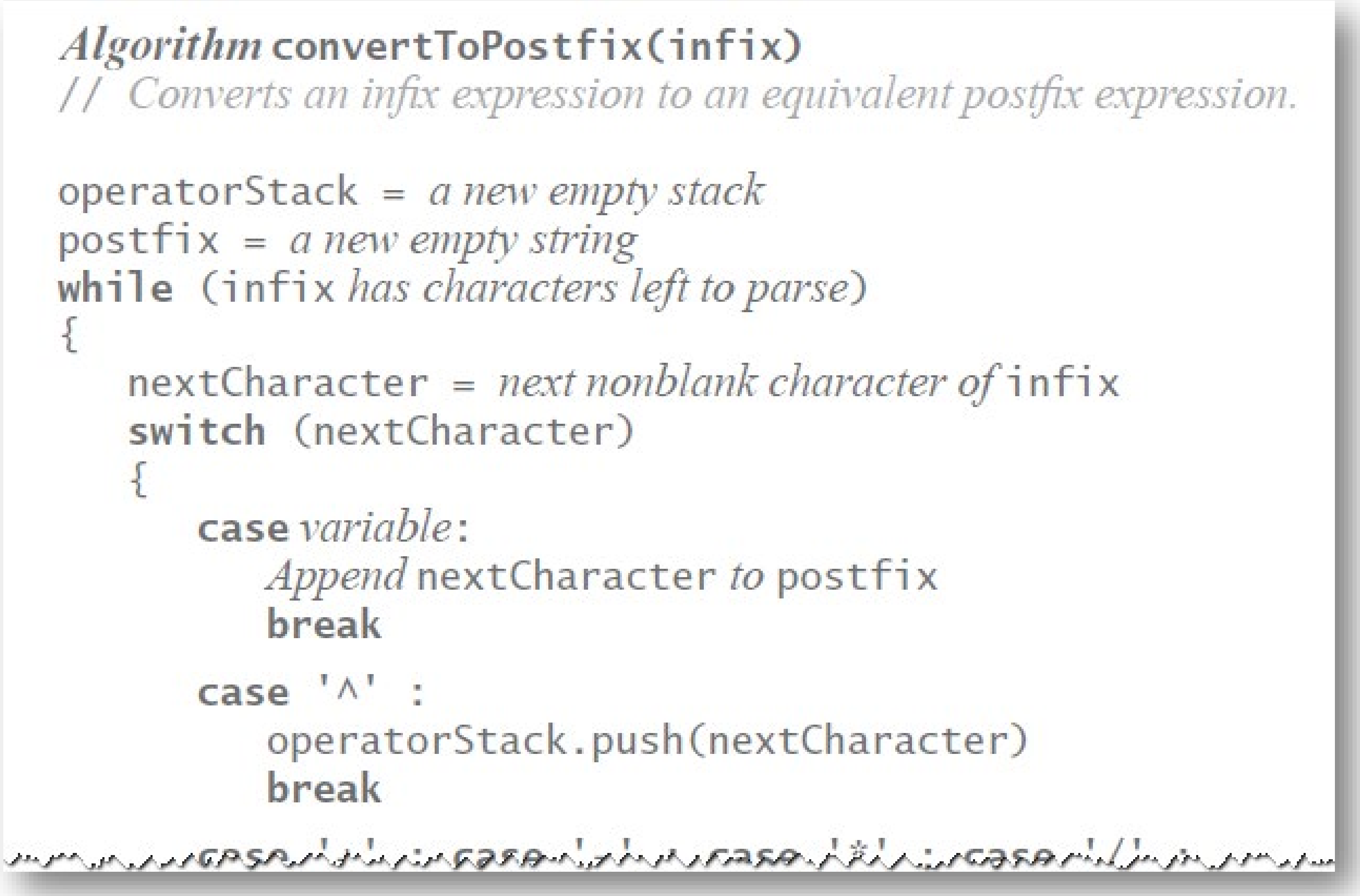
Infix to Postfix: Larger Example

The steps in converting the infix expression
*a / b * (c + (d - e))* to postfix form



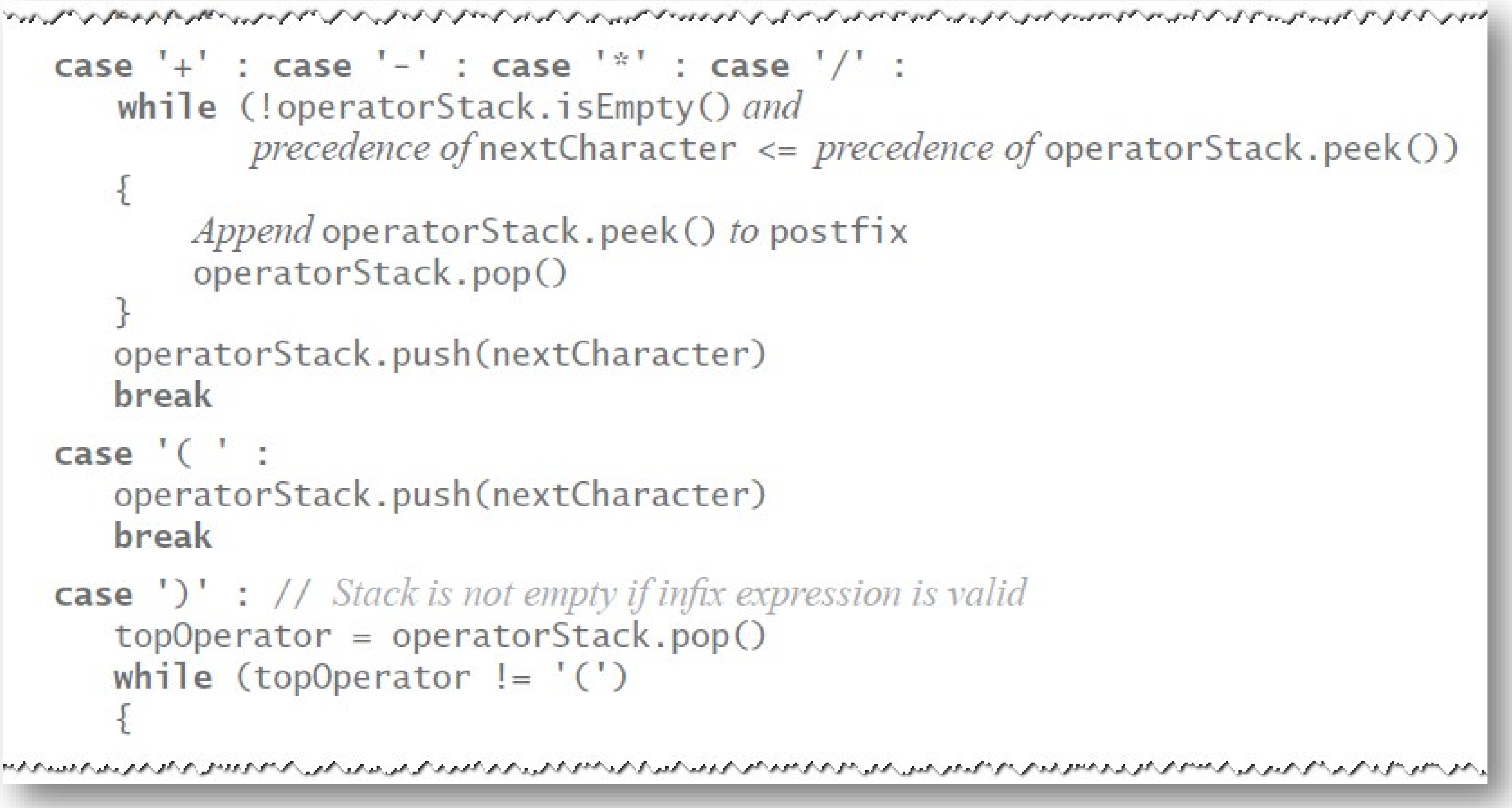
Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b* /	
	a b /	*
(	*a b* /	* (
c	*a b* / *c*	* (
+	*a b* / *c*	* (+
(	*a b* / *c*	* (+ (
d	*a b* / *c d*	* (+ (
−	*a b* / *c d*	* (+ (−
e	*a b* / *c d e*	* (+ (−
)	*a b* / *c d e* −	* (+ (
	a b / *c d e* −	* (+
)	*a b* / *c d e* − +	* (
	a b / *c d e* − +	*
	a b / *c d e* − + *	

Infix-to-postfix Algorithm



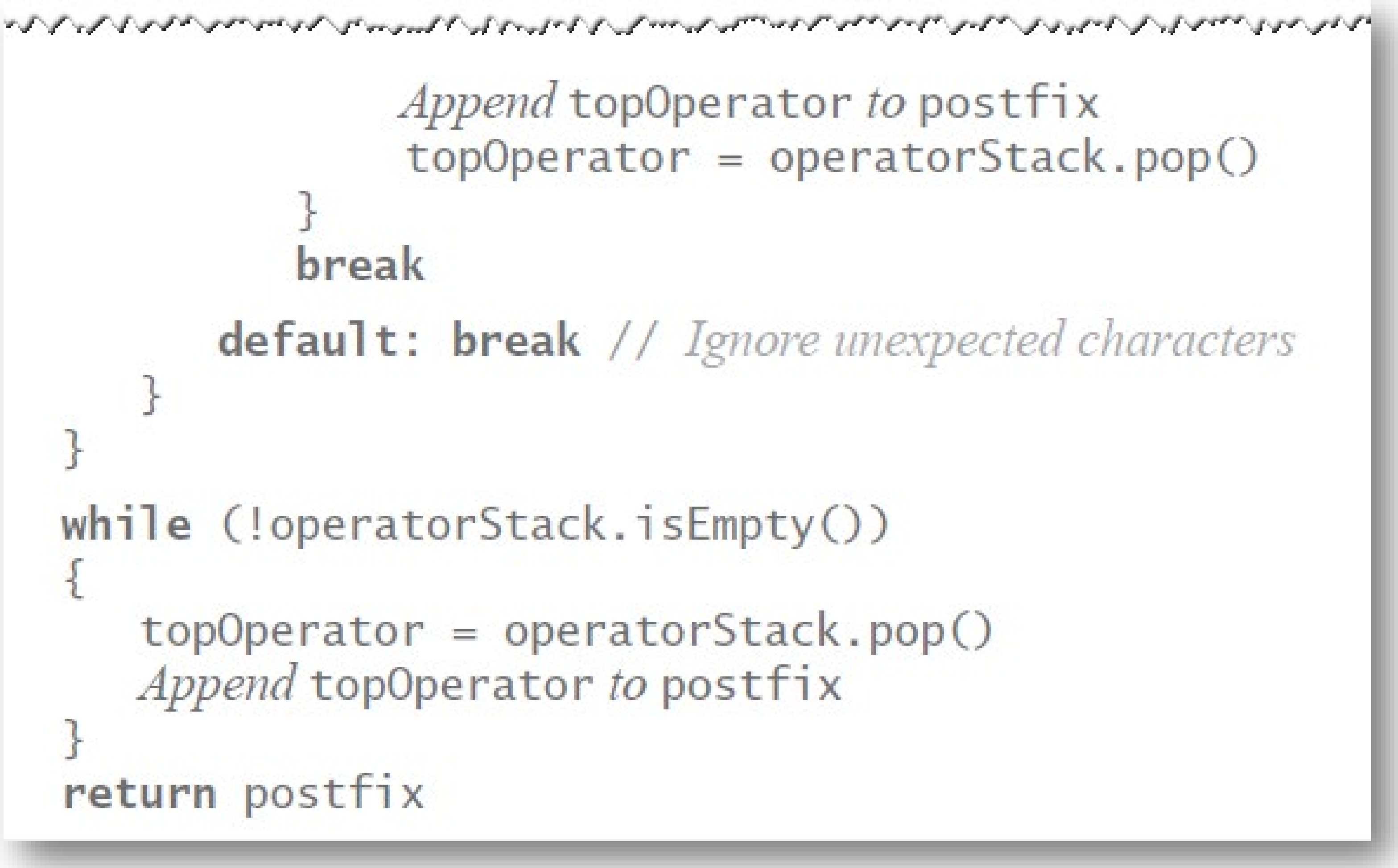
```
Algorithm convertToPostfix(infix)
// Converts an infix expression to an equivalent postfix expression.

operatorStack = a new empty stack
postfix = a new empty string
while (infix has characters left to parse)
{
    nextCharacter = next nonblank character of infix
    switch (nextCharacter)
    {
        case variable:
            Append nextCharacter to postfix
            break

        case '^' :
            operatorStack.push(nextCharacter)
            break

        case '+' : case '-' : case '*' : case '/' :
```

Infix-to-postfix Algorithm



```
case '+' : case '-' : case '*' : case '/' :
   while (!operatorStack.isEmpty() and
            precedence of nextCharacter <= precedence of operatorStack.peek())
   {
         Append operatorStack.peek() to postfix
         operatorStack.pop()
   }
   operatorStack.push(nextCharacter)
   break
case '( ' :
   operatorStack.push(nextCharacter)
   break
case ')' : // Stack is not empty if infix expression is valid
   topOperator = operatorStack.pop()
   while (topOperator != '(')
   {
```

Infix-to-postfix Algorithm



```
                Append topOperator to postfix
                topOperator = operatorStack.pop()
            }
            break

        default: break // Ignore unexpected characters
    }
}
while (!operatorStack.isEmpty())
{
    topOperator = operatorStack.pop()
    Append topOperator to postfix
}
return postfix
```

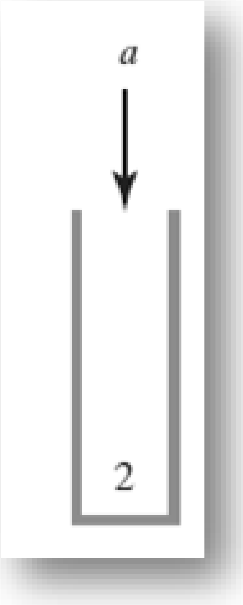
Step 3: Evaluating Postfix Expressions

1. Initialize an empty Stack
2. for each character in postfix expression
    1. if variable, push its value to Stack
    2. if operator
        1. pop second operand
        2. pop first operand
        3. apply operator to two operands
        4. push result
3. Return the remaining value in Stack

Step 3: Evaluating Postfix Expressions

The stack during the evaluation of the postfix expression

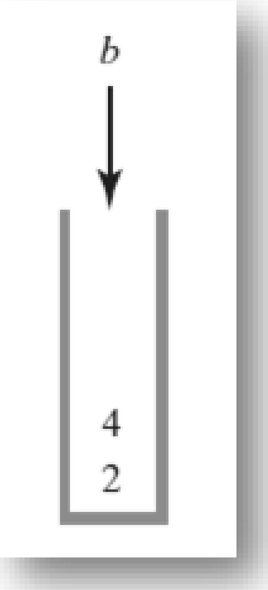
a b / when *a* is 2 and b is 4



Step 3: Evaluating Postfix Expressions

The stack during the evaluation of the postfix expression

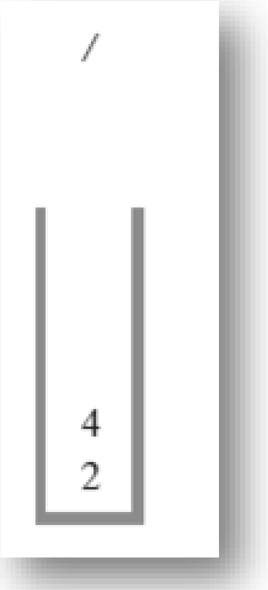
a b / when *a* is 2 and b is 4



Step 3: Evaluating Postfix Expressions

The stack during the evaluation of the postfix expression

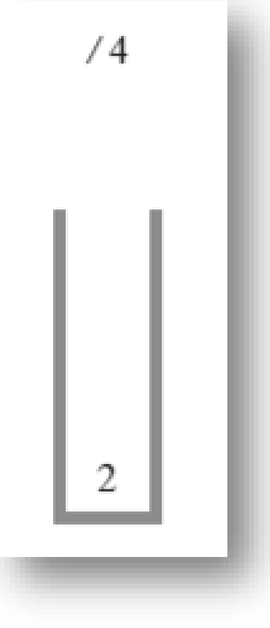
a b /   when *a* is 2 and b is 4



Step 3: Evaluating Postfix Expressions

The stack during the evaluation of the postfix expression

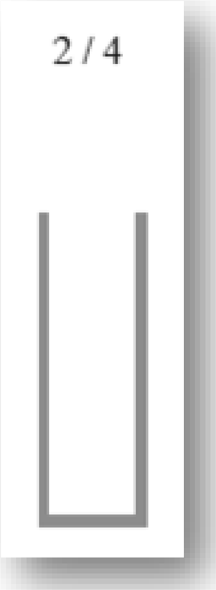
a b / when *a* is 2 and b is 4



Step 3: Evaluating Postfix Expressions

The stack during the evaluation of the postfix expression

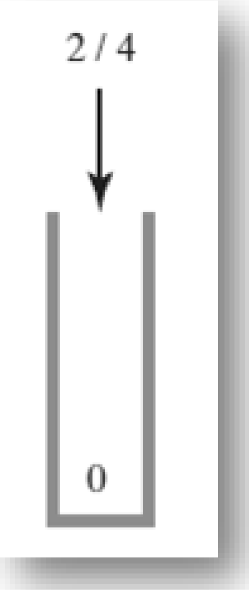
a b / when *a* is 2 and b is 4



Step 3: Evaluating Postfix Expressions

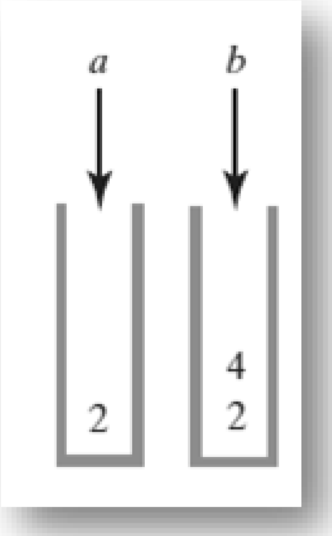
The stack during the evaluation of the postfix expression

a b / when *a* is 2 and b is 4



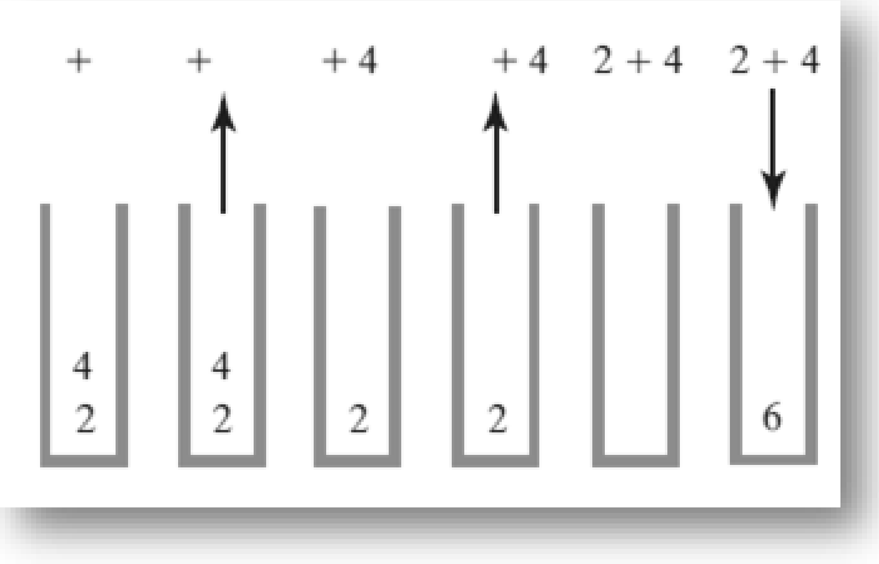
Evaluating Postfix Expressions: Example 2

The stack during the evaluation of the postfix expression a b + c / when *a* is 2, *b* is 4, and *c* is 3



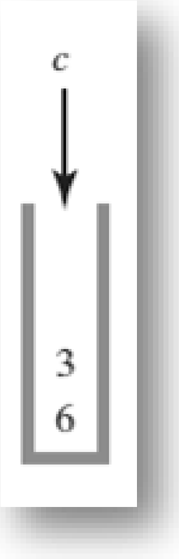
Evaluating Postfix Expressions: Example 2

The stack during the evaluation of the postfix expression a b + c / when *a* is 2, *b* is 4, and *c* is 3



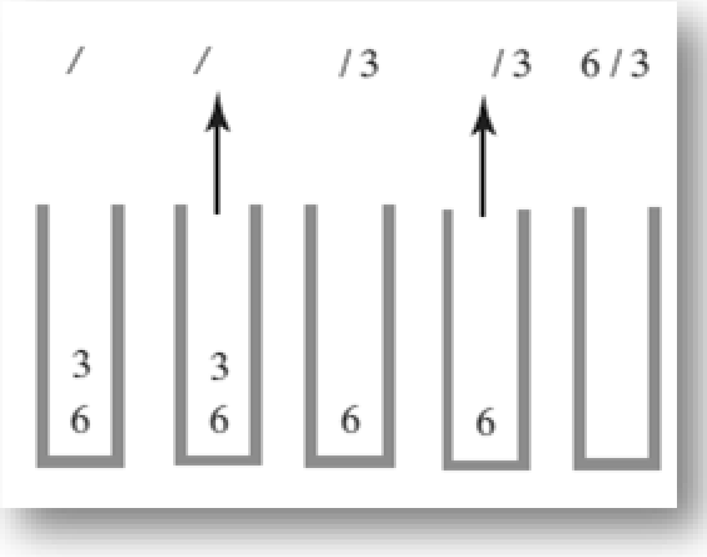
Evaluating Postfix Expressions: Example 2

The stack during the evaluation of the postfix expression a b + c / when *a* is 2, *b* is 4, and *c* is 3



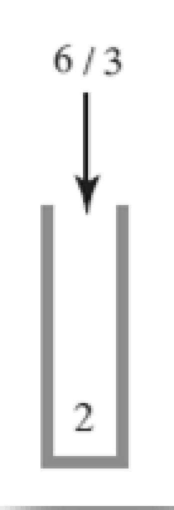
Evaluating Postfix Expressions: Example 2

The stack during the evaluation of the postfix expression a b + c / when *a* is 2, *b* is 4, and *c* is 3



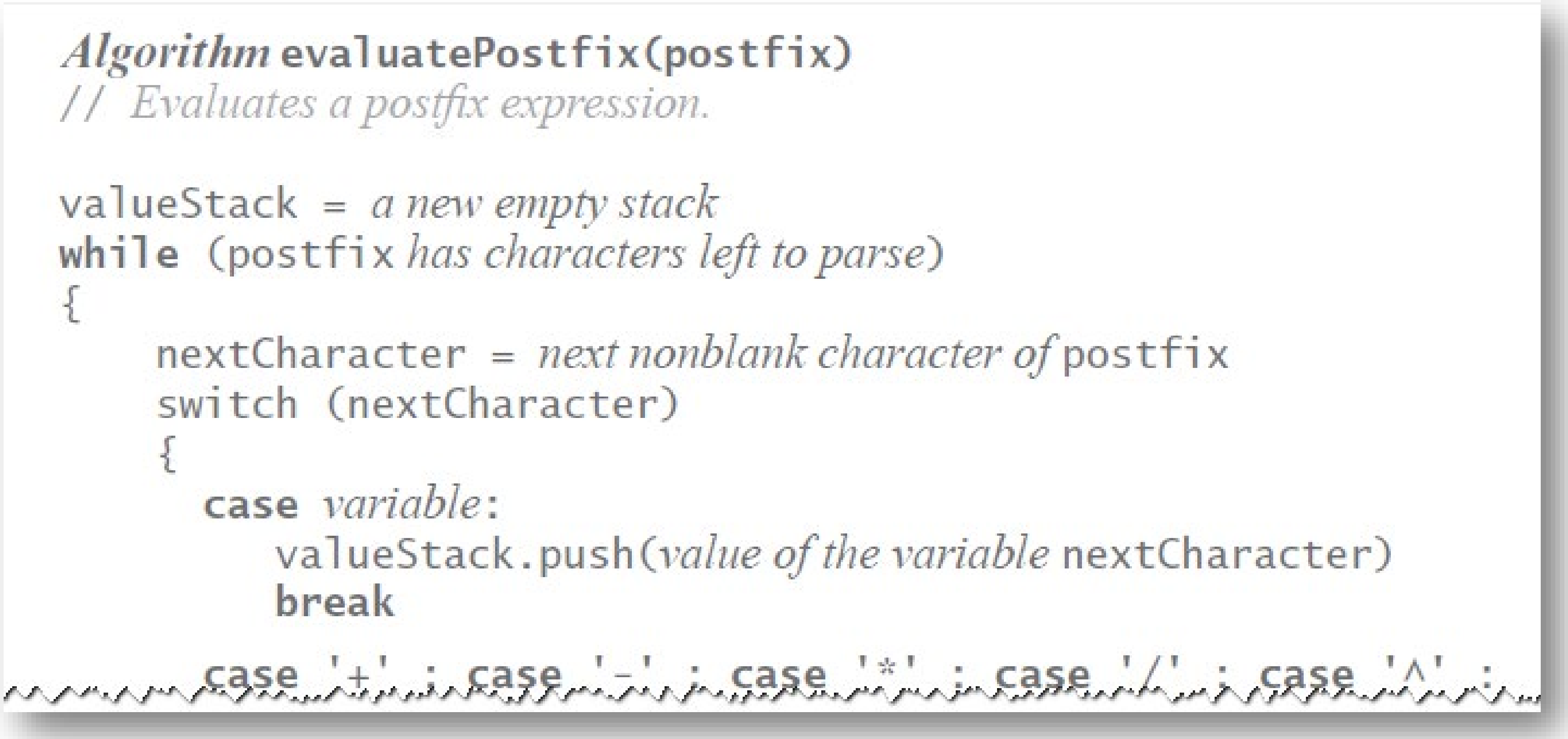
Evaluating Postfix Expressions: Example 2

The stack during the evaluation of the postfix expression a b + c / when *a* is 2, *b* is 4, and *c* is 3



Evaluating Postfix Expressions

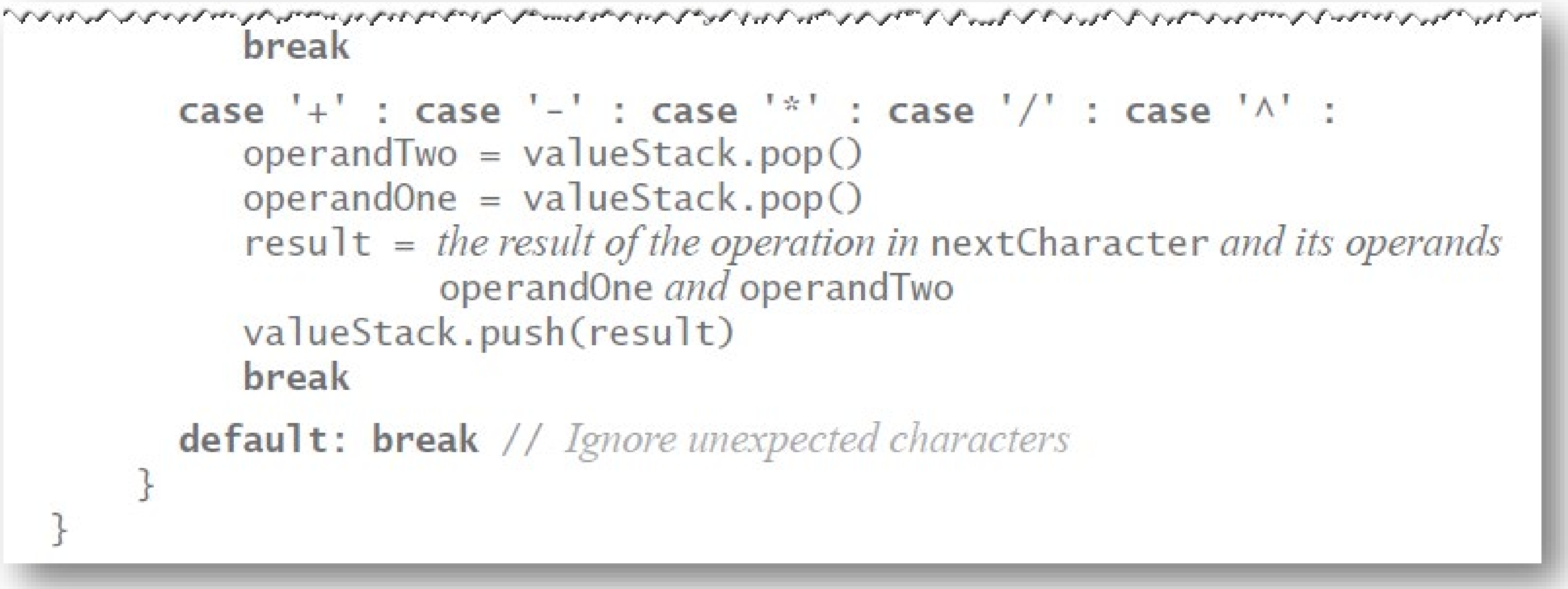
- Algorithm for evaluating postfix expressions.



```
Algorithm evaluatePostfix(postfix)
// Evaluates a postfix expression.

valueStack = a new empty stack
while (postfix has characters left to parse)
{
     nextCharacter = next nonblank character of postfix
     switch (nextCharacter)
     {
        case variable:
            valueStack.push(value of the variable nextCharacter)
            break

        case '+' : case '-' : case '*' : case '/' : case '^' :
```

Evaluating Postfix Expressions

- Algorithm for evaluating postfix expressions.



```
         break

      case '+' : case '-' : case '*' : case '/' : case '^' :
         operandTwo = valueStack.pop()
         operandOne = valueStack.pop()
         result = the result of the operation in nextCharacter and its operands
                  operandOne and operandTwo
         valueStack.push(result)
         break

      default: break // Ignore unexpected characters
    }
}
```

What is the running time?

- in terms of *n,* the length of the input prefix string
- Check balance
  - how many times does each character get pushed?
    - at most 1
  - how many times does each character get poped?
    - at most 1
  - What is the runtime of push and pop?
    - O(1)
  - O(n)
- Convert infix to postfix: O(n)
- Evaluate postfix: O(n)
- Total: O(3n) = O(n)
- Three passes!
- Can we do better?
- Yes! We can use two passes only
  - Expect to require more space
  - space-time tradeoff

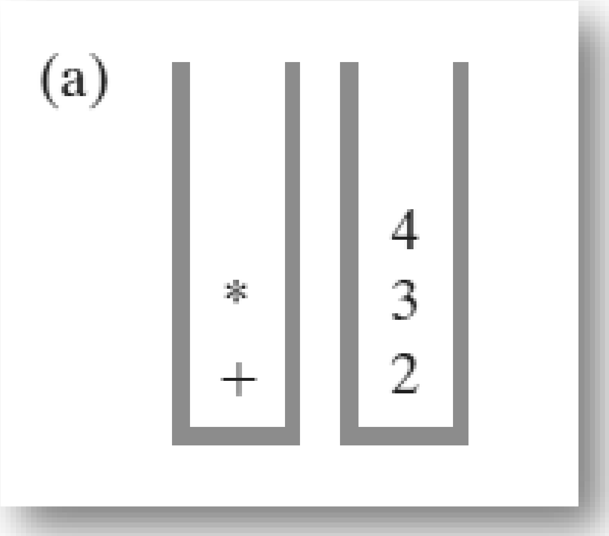
Evaluating Infix Expressions with 2 passes only

- We will use two stacks
  - Operator Stack
  - Operand stack
- Scan the expression once:
  - follow the steps of infix conversion to postfix,
  - **except**
    - instead of appending to postfix output, push to operand stack
    - when popping an operator, pop second then first operands, apply operator, push result to operand stack
- While operator stack not empty
  - pop an operator
  - pop second operand then first operand
  - apply the operator and push result to operand stack
- Result is the remaining value in the operand stack

Evaluating Infix Expressions with 2 passes only

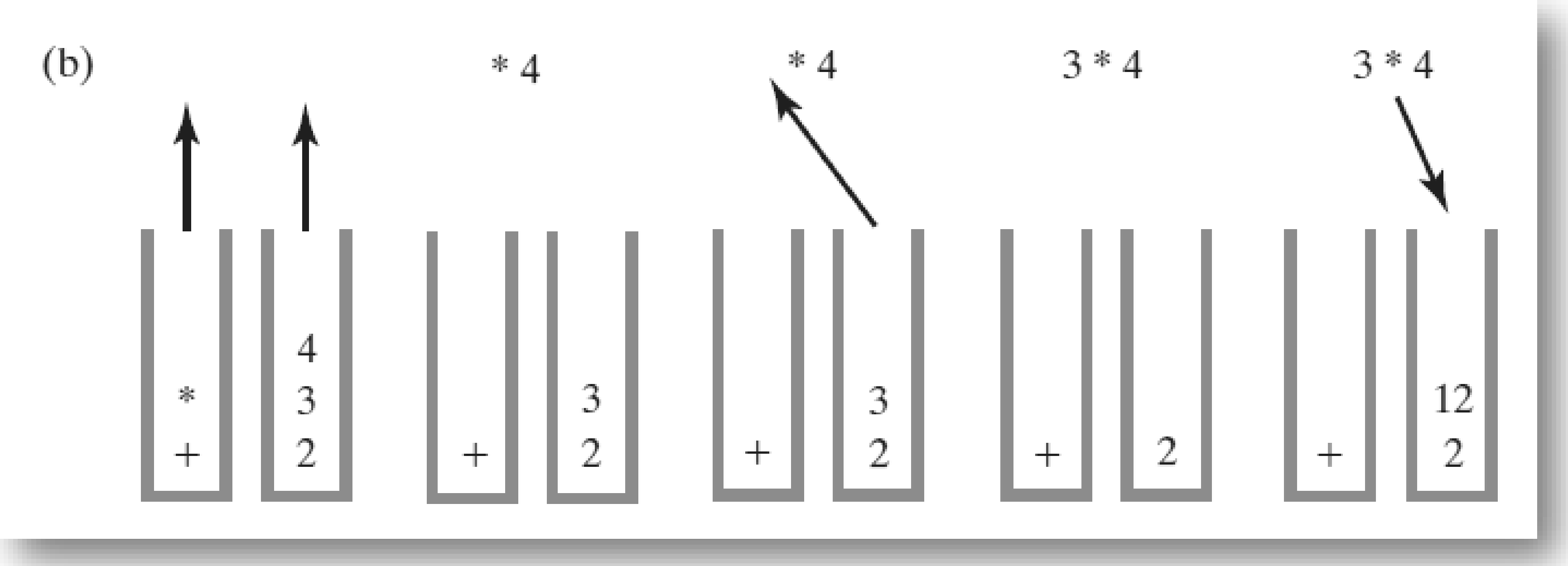
Two stacks during the evaluation of
a + b * c when a is 2, b is 3, and c is 4:

- after reaching the end of the expression;



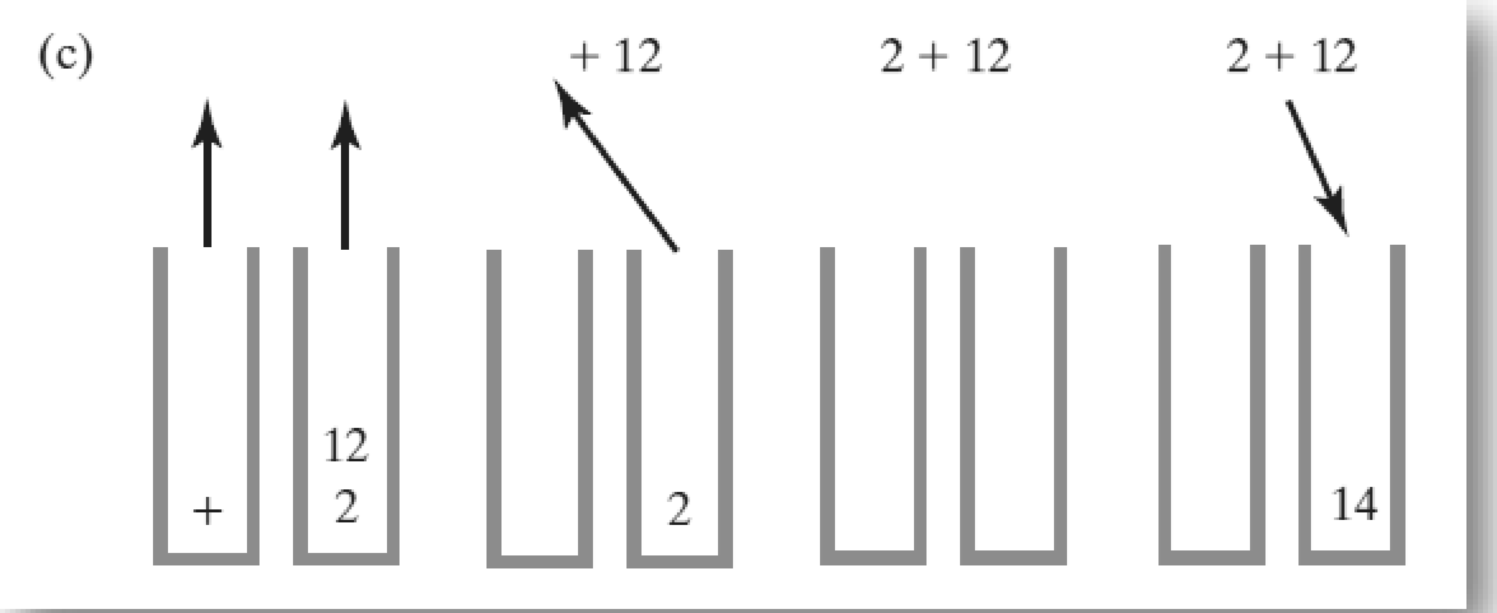
Evaluating Infix Expressions

Two stacks during the evaluation of
a + b * c when a is 2, b is 3, and c is 4:
while performing the multiplication;



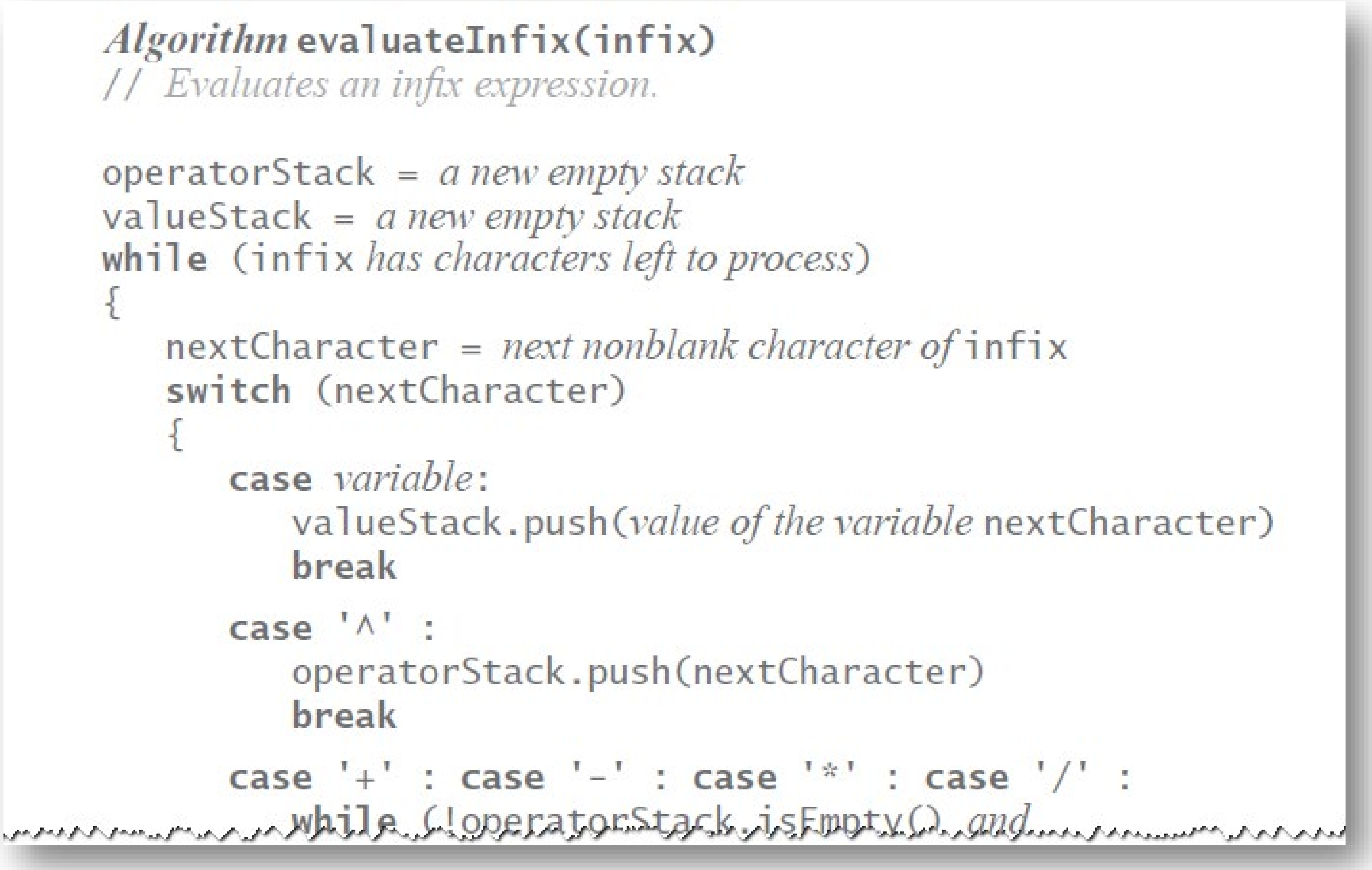
Evaluating Infix Expressions

Two stacks during the evaluation of
a + b * c when a is 2, b is 3, and c is 4:
(c) while performing the addition



Evaluating Infix Expressions

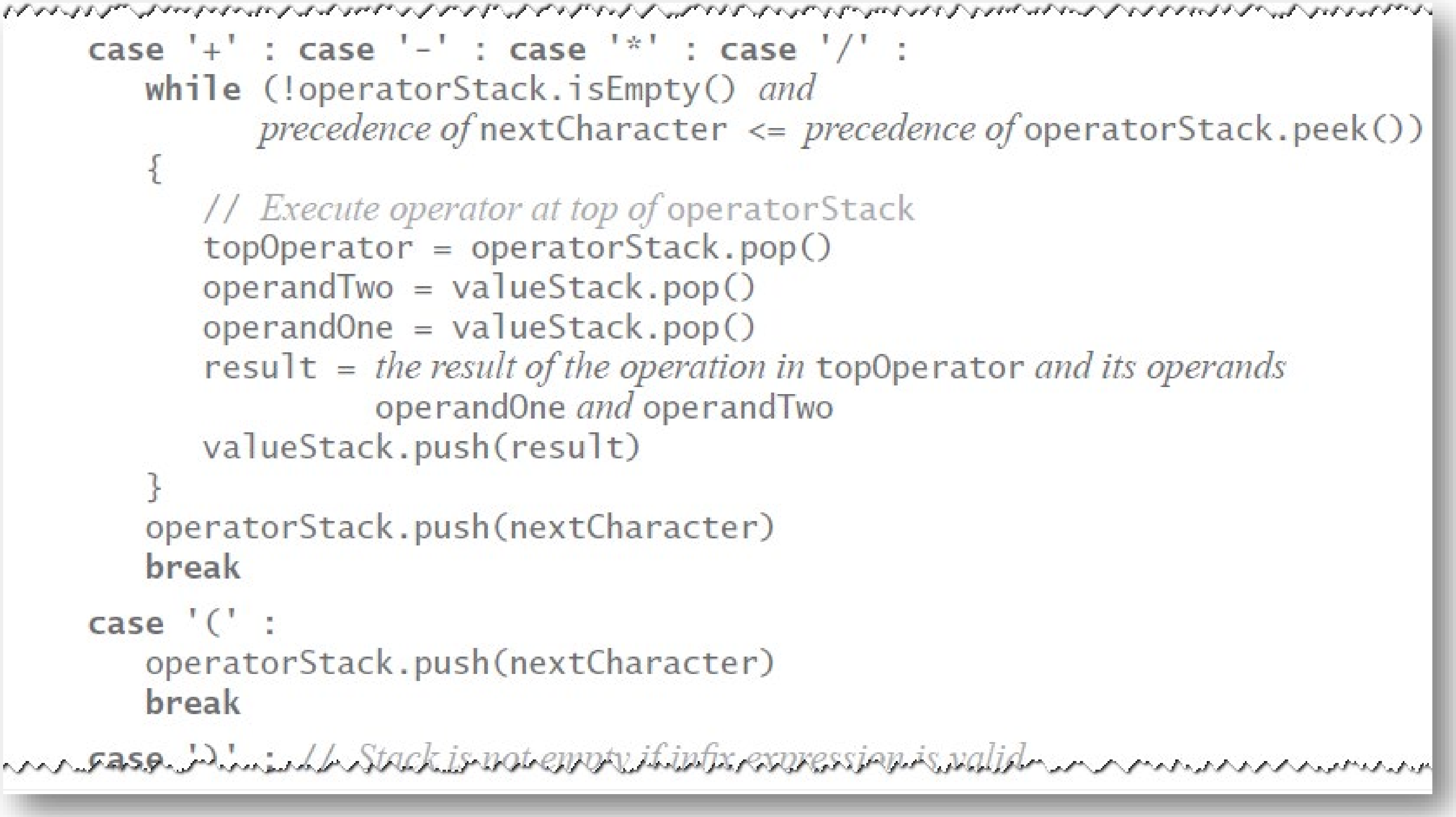
- Algorithm to evaluate infix expression.



```
Algorithm evaluateInfix(infix)
// Evaluates an infix expression.

operatorStack = a new empty stack
valueStack = a new empty stack
while (infix has characters left to process)
{
   nextCharacter = next nonblank character of infix
   switch (nextCharacter)
   {
      case variable:
         valueStack.push(value of the variable nextCharacter)
         break

      case '^' :
         operatorStack.push(nextCharacter)
         break

      case '+' : case '-' : case '*' : case '/' :
         while (!operatorStack.isEmpty() and
```

Evaluating Infix Expressions

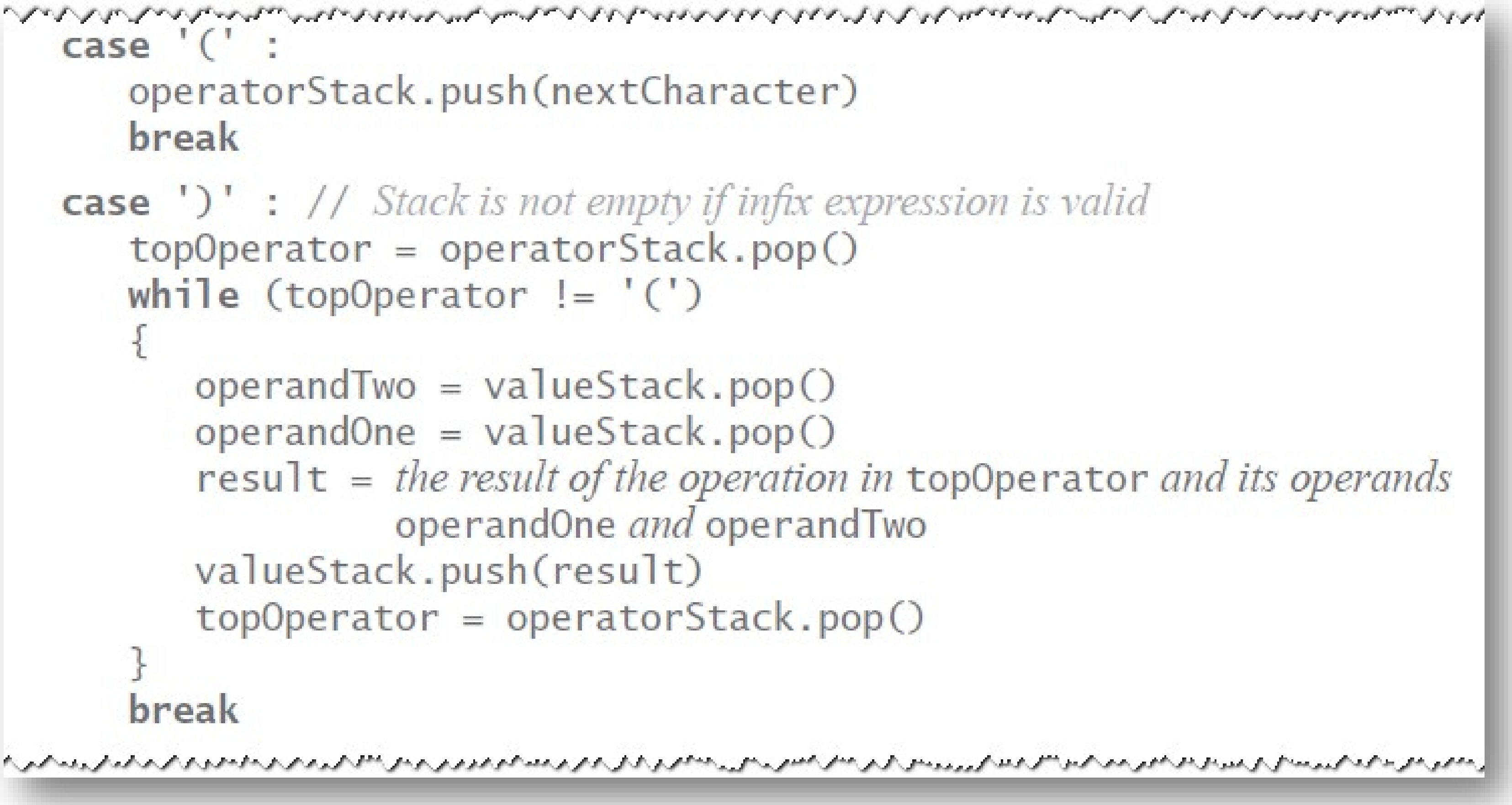
- Algorithm to evaluate infix expression.



```
case '+' : case '-' : case '*' : case '/' :
   while (!operatorStack.isEmpty() and
         precedence of nextCharacter <= precedence of operatorStack.peek())
   {
      // Execute operator at top of operatorStack
      topOperator = operatorStack.pop()
      operandTwo = valueStack.pop()
      operandOne = valueStack.pop()
      result = the result of the operation in topOperator and its operands
               operandOne and operandTwo
      valueStack.push(result)
   }
   operatorStack.push(nextCharacter)
   break

case '(' :
   operatorStack.push(nextCharacter)
   break

case ')' : // Stack is not empty if infix expression is valid
```

Evaluating Infix Expressions

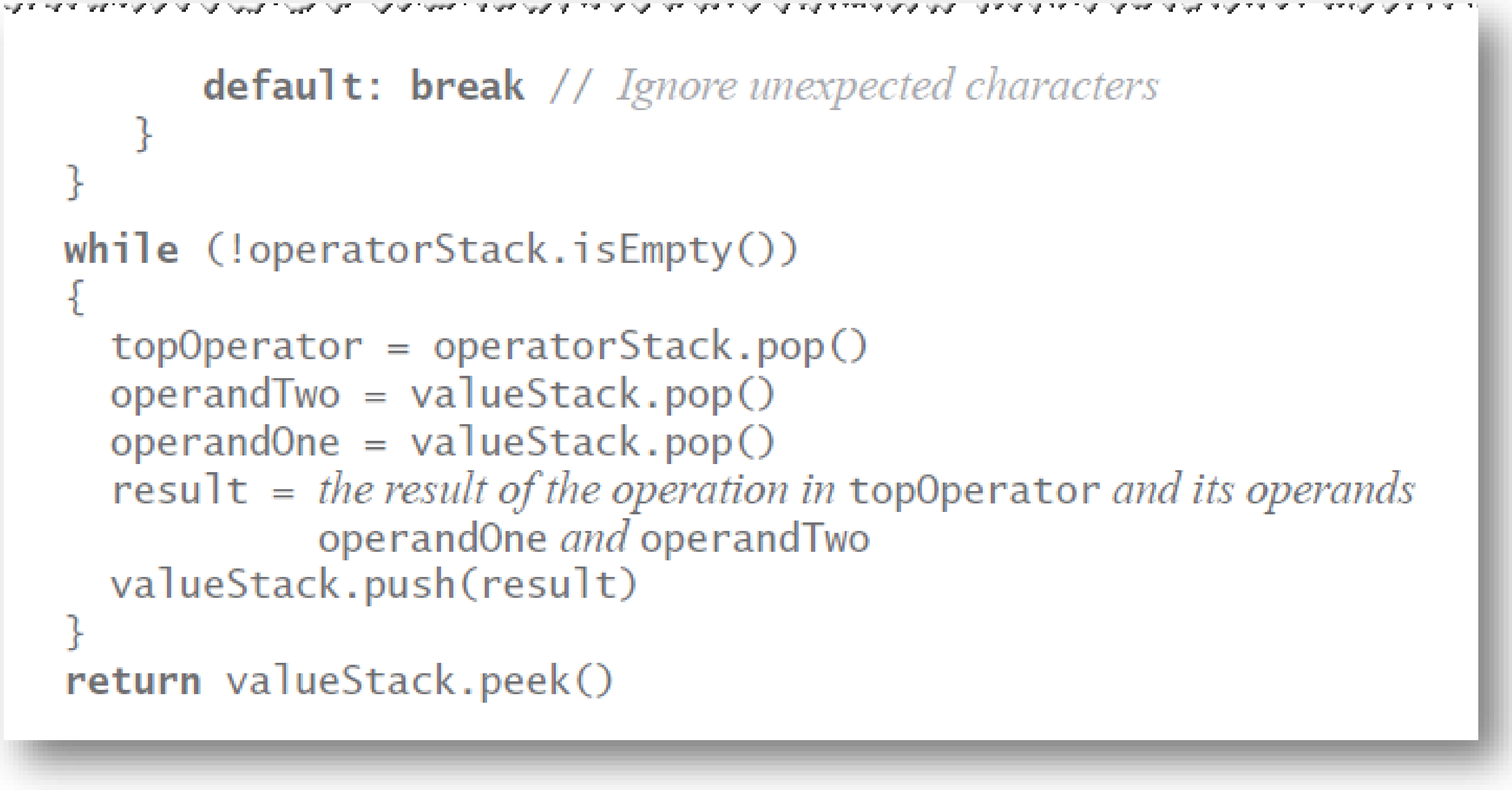
- Algorithm to evaluate infix expression.



```
case '(' :
   operatorStack.push(nextCharacter)
   break

case ')' : // Stack is not empty if infix expression is valid
   topOperator = operatorStack.pop()
   while (topOperator != '(')
   {
      operandTwo = valueStack.pop()
      operandOne = valueStack.pop()
      result = the result of the operation in topOperator and its operands
               operandOne and operandTwo
      valueStack.push(result)
      topOperator = operatorStack.pop()
   }
   break
```

Evaluating Infix Expressions

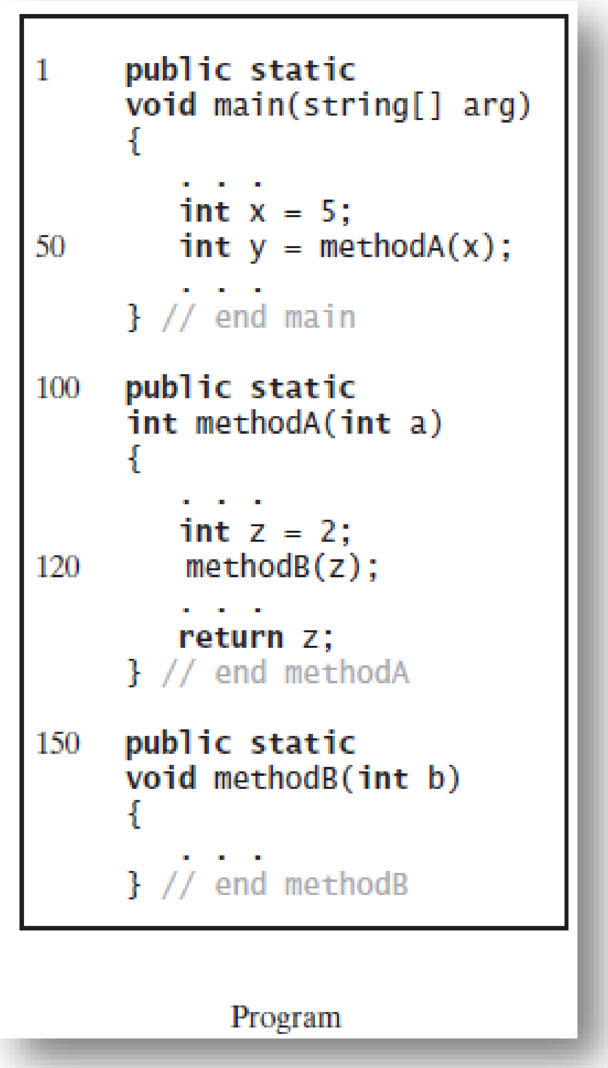
- Algorithm to evaluate infix expression.



```
            default: break // Ignore unexpected characters
        }
    }
    while (!operatorStack.isEmpty())
    {
        topOperator = operatorStack.pop()
        operandTwo = valueStack.pop()
        operandOne = valueStack.pop()
        result = the result of the operation in topOperator and its operands
                 operandOne and operandTwo
        valueStack.push(result)
    }
    return valueStack.peek()
```

The Program Stack (aka runtime stack)

The program stack at three points in time: (a) when main begins execution;

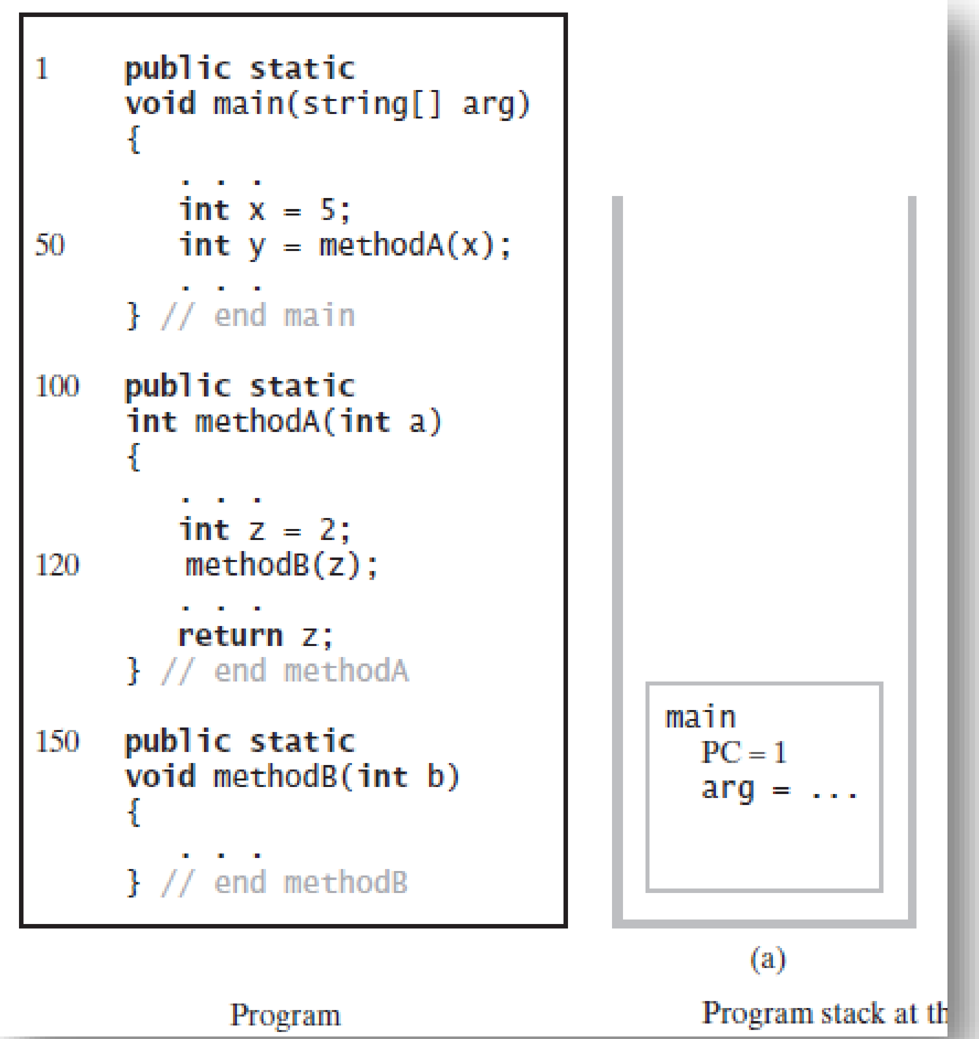


```
1      public static
       void main(string[] arg)
       {
          . . .
          int x = 5;
50        int y = methodA(x);
          . . .
       } // end main

100    public static
       int methodA(int a)
       {
          . . .
          int z = 2;
120        methodB(z);
          . . .
          return z;
       } // end methodA

150    public static
       void methodB(int b)
       {
          . . .
       } // end methodB
```

Program

The Program Stack

The program stack at three points in time: (a) when main begins execution



```
1     public static
      void main(string[] arg)
      {
         . . .
         int x = 5;
50       int y = methodA(x);
         . . .
      } // end main

100   public static
      int methodA(int a)
      {
         . . .
         int z = 2;
120       methodB(z);
         . . .
         return z;
      } // end methodA

150   public static
      void methodB(int b)
      {
         . . .
      } // end methodB
```

```
main
  PC = 1
  arg = . . .
```

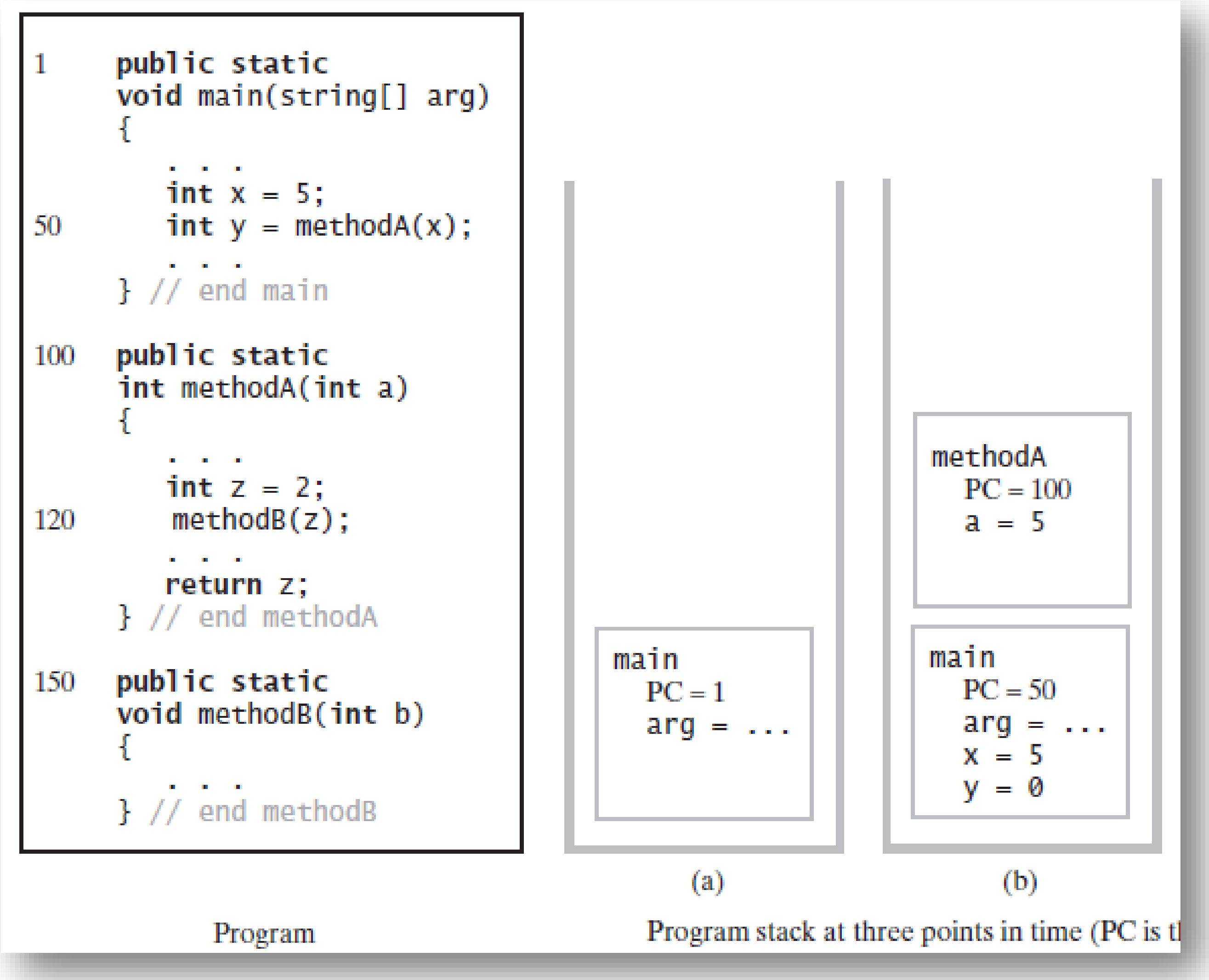
(a)

Program

Program stack at th

The Program Stack

The program stack at three points in time: (a) when main begins execution; (b) when methodA begins execution



```
1     public static
      void main(string[] arg)
      {
         . . .
         int x = 5;
50       int y = methodA(x);
         . . .
      } // end main

100   public static
      int methodA(int a)
      {
         . . .
         int z = 2;
120       methodB(z);
         . . .
         return z;
      } // end methodA

150   public static
      void methodB(int b)
      {
         . . .
      } // end methodB
```

(a)

main
PC = 1
arg = ...

(b)

methodA
PC = 100
a = 5

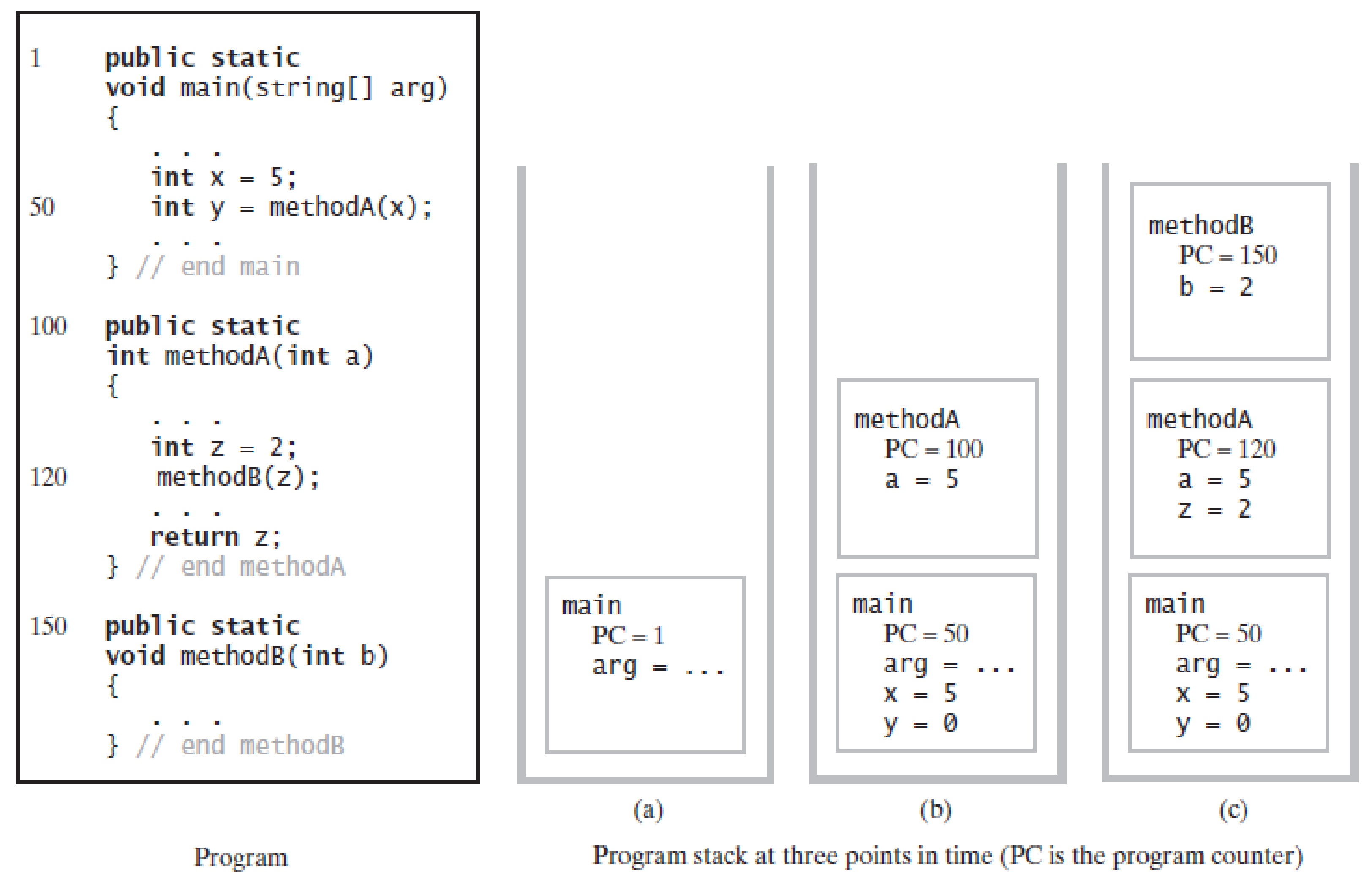
main
PC = 50
arg = ...
x = 5
y = 0

Program

Program stack at three points in time (PC is tl

The Program Stack

The program stack at three points in time: (a) when main begins execution; (b) when methodA begins execution; (c) when methodB begins execution



```
1     public static
      void main(string[] arg)
      {
         . . .
         int x = 5;
50       int y = methodA(x);
         . . .
      } // end main

100   public static
      int methodA(int a)
      {
         . . .
         int z = 2;
120       methodB(z);
         . . .
         return z;
      } // end methodA

150   public static
      void methodB(int b)
      {
         . . .
      } // end methodB
```

Program

Program stack at three points in time (PC is the program counter)

Java Class Library:The Class `Stack`

- Found in `java.util`

- Methods
  - A constructor – creates an empty stack
  - `public T push(T item);`
  - `public T pop();`
  - `public T peek();`
  - `public boolean empty();`