Lecture 4: Stack Applications

CS2504/CS4092– Algorithms and Linear Data Structures

Dr. Kieran T. Herley

Department of Computer Science
University College Cork

2013/14

Summary

Matching Parentheses Problem. Postfix notation for arithmetic expressions. Stack-based algorithm for evaluating postfix expressions. Translating infix to postfix.

Parentheses and Mathematical Expressions

- Mathematical expressions contain pairs of "grouping" symbols to indicate structure:
  - Parentheses: (and)
  - Braces: $\{$ and $\}$
  - Brackets [and]
  - Floor symbols: $\lfloor$ and $\rfloor$
  - Ceiling symbols $\lceil$ and $\rceil$
- Opening and closing symbols of each pair must "match up" for expression to make sense

Parentheses and Mathematical Expressions

- Mathematical expressions contain pairs of "grouping" symbols to indicate structure:
  - Parentheses: (and)
  - Braces: { and }
  - Brackets [and]
  - Floor symbols: $\lfloor$ and $\rfloor$
  - Ceiling symbols $\lceil$ and $\rceil$
- Opening and closing symbols of each pair must "match up" for expression to make sense
- For example

$$\overbrace{[\underbrace{(5+x)}_{B}-\underbrace{(y+z)}_{C}]}^{A}$$

Matching Parentheses Problem

Problem Read a string containing an expression and determine if the parentheses within are balanced.

Some Balanced Strings

() (()) ()()() ([]{})[]

Some Unbalanced Strings

)((([]]]

Notes We will allow (), [] and {} and ignore all non-parentheses.

More Formal Characterization

1. (), {} and [] are balanced
2. if S_1 and $S2$ are balanced, then so too is $S_1 \cdot S_2$
3. if S is balanced, then so too are (S), $\{S\}$ and $[S]$
4. Only those strings consistent with 1, 2, 3 above are balanced

Parentheses Matching and HTML Validation

```
<body>
<h1>My Page</h1>
<p>
Here is some text.
</p>
<ol>
    <li> One</li>
    <li>Two</li>
    <li>Three</li>
</ol>
</body>
```

- HTML tags (mostly) come in pairs
- Tag pairs akin to "parentheses"
- Must match properly

Fact

Many "syntax checking" algorithms (*e.g.* programming language parsers) are stack based.

Algorithm Sketch

- Read through the string character by character:
  - for each non-paren, simply ignore it
  - for each open paren *i.e* '(', '[' or '{', push it onto the stack
  - for each close paren *i.e* ')', ']' or '}',
    - pop the top symbol off the stack and test that it "matches" the current character
    - '(' matches ')' and vice versa, similarly with others

Algorithm Sketch

- Read through the string character by character:
  - for each non-paren, simply ignore it
  - for each open paren *i.e* '(', '[' or '{', push it onto the stack
  - for each close paren *i.e* ')', ']' or '}',
    - pop the top symbol off the stack and test that it "matches" the current character
    - '(' matches ')' and vice versa, similarly with others
- If any of the following events occur then the string is unbalanced:
  - Case 1: mismatch detected
  - Case 2: a close paren is encountered and the stack is empty
  - Case 3: the stack is non-empty at the end

  Otherwise it is balanced.

Some Illustrations

String	Stack
()	(
()	

  Verdict: OK

String	Stack
(]	(
(]	

  Verdict: mistatch

String	Stack
()[]	(
()[]	
()[]	[
()[]	

  Verdict: OK

String	Stack
([( )])	(

Some Illustrations

String	Stack
()	(
()	

  Verdict: OK

String	Stack
(]	(
(]	

  Verdict: mistatch

String	Stack
()[]	(
()[]	
()[]	[
()[]	

  Verdict: OK

String	Stack
([( )])	(
([( )])	([

Some Illustrations

String	Stack
()	(
()	

  Verdict: OK

String	Stack
(]	(
(]	

  Verdict: mistatch

String	Stack
()[]	(
()[]	
()[]	[
()[]	

  Verdict: OK

String	Stack
([( )])	(
([( )])	([
([( )])	([(

Some Illustrations

String	Stack
()	(
()	

  Verdict: OK

String	Stack
(]	(
(]	

  Verdict: mistatch

String	Stack
()[]	(
()[]	
()[]	[
()[]	

  Verdict: OK

String	Stack
([( )])	(
([( )])	([
([( )])	([(
([( )])	([

Some Illustrations

String	Stack
()	(
()	

  Verdict: OK

String	Stack
(]	(
(]	

  Verdict: mistatch

String	Stack
()[]	(
()[]	
()[]	[
()[]	

  Verdict: OK

String	Stack
([( )])	(
([( )])	([
([( )])	([(
([( )])	([
([( )])	(

Some Illustrations

String	Stack
()	(
()	

  Verdict: OK

String	Stack
(]	(
(]	

  Verdict: mistatch

String	Stack
()[]	(
()[]	
()[]	[
()[]	

  Verdict: OK

String	Stack
([( )])	(
([( )])	([
([( )])	([(
([( )])	([
([( )])	(
([( )])	

Some Illustrations

String	Stack
()	(
()	

  Verdict: OK

String	Stack
(]	(
(]	

  Verdict: mistatch

String	Stack
()[]	(
()[]	
()[]	[
()[]	

  Verdict: OK

String	Stack
([( )])	(
([( )])	([
([( )])	([(
([( )])	([
([( )])	(
([( )])	

  Verdict: OK

Pseudcode

```
Algorithm IsBalanced(s)

    Create empty parenStack

    for i ←0 to s.length() do
      current = s.charAt(i)

      if isOpenParen(current) then
          parenStack.push(current)
      else
      if isCloseParen(current) then
          if parenStack.isEmpty() then
              return false
          else
              top = parenStack.pop()
              if match(current) ≠top then
                  return false

    return parenStack.isEmpty()
```

- Semiformal programming notation
- Captures algorithmic essentials, but not all programming nitty-gritty
- Block structure indicated purely by indentation– i.e. no {} for loop bodies etc.

Algorithm Outline

Algorithm IsBalanced(s)

```
Create empty parenStack

for i ←0 to s.length() do
    current = s.charAt(i)

    /* "process" character current */

return parenStack.isEmpty()
```

Note: "process" returns false if problem encountered; otherwise balance depends on whether stack empty when completed (Case 3)

"Process" Current Character

```
if isOpenParen(current) then
    parenStack.push(current)
else
    /* handle close paren symbol */
```

"Process" Character current cont'd

Handling close paren: things OK unless

- stack empty (Case 2) or
- close paren and top stack symbol do not match (Case 1)

in either case the string is unbalanced so we return false.

```
if isCloseParen(current) then
    if parenStack.isEmpty() then /* Case 2 */
        return false
    else
        top = parenStack.pop()
        if match(current) ≠top then /* Case 1 */
            return false
```

Note: helper method match(x) returns "twin" of symbol x, i.e. '(' for ')' etc.

Putting It Together

```
Algorithm IsBalanced(s)

  Create empty parenStack

  for i ←0 to s.length() do
    current = s.charAt(i)

    if isOpenParen(current) then
        parenStack.push(current)
    else
    if isCloseParen(current) then
        if parenStack.isEmpty() then
            return false
        else
            top = parenStack.pop()
            if match(current) ≠top then
                return false

  return parenStack.isEmpty()
```

- Many "syntax checking" algorithms are stack based
- So to typically are many compilers' parsing engines
- Other useful tools are "syntax-driven" i.e. piggy-back their work on top of an underlying "syntax checking" process, *e.g.* indentation checker

Simple Postfix Calculator

"Standard" (Infix) Notation Operators $(+, -, *, /)$ appear between values/subexpressions to which they are applied, e.g.

$$1 \quad + \quad 2$$

Postfix Notation

- Postfix notation is an alternative notation for arithmetic expressions
- Operators appear *after* values/subexpressions to which it is applied, e.g.

$$1 \quad 2 \quad +$$

(Need spaces to separate operands in this notation)

Postfix Notation

- Every infix expression has an equivalent postfix expression (same result) and vice versa.
- Examples

Infix	Postfix
$1+2$	$1\ 2\ +$
$1+2+3$	$1\ 2\ +\ 3\ +$
$1*2+3+4$	$1\ 2\ *\ 3\ +\ 4\ +$
$(1+2)*3$	$1\ 2\ +\ 3\ *$

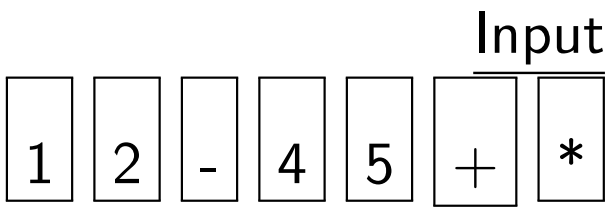
- Postfix offers some advantages over infix:
  - No parentheses required
  - Simpler evaluation algorithm than infix

Postfix Evaluation Algorithm

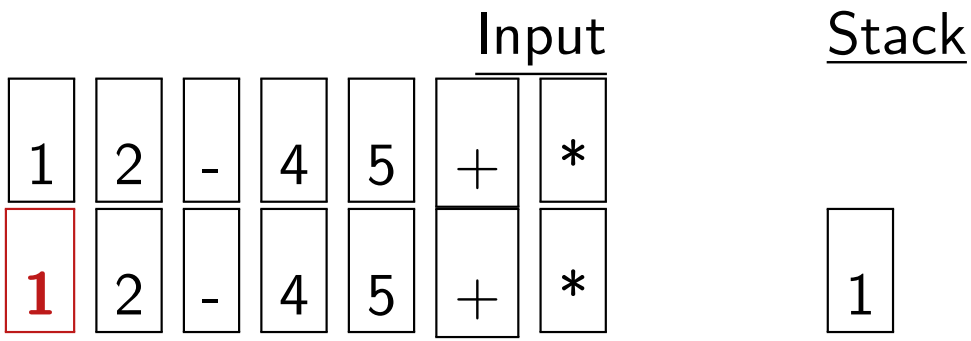
- Read the expression from left to right one operand/operator at a time:[1]
  - for each operand, push it onto a stack
  - for each operator, pop the top two operands off the stack, apply the operator to them and push the result onto the stack
- Once the end of the expression has been reached, the result is on the top of the stack.

[1] This algorithm handles positive operands and binary operators only; can be adapted to remove these restrictions

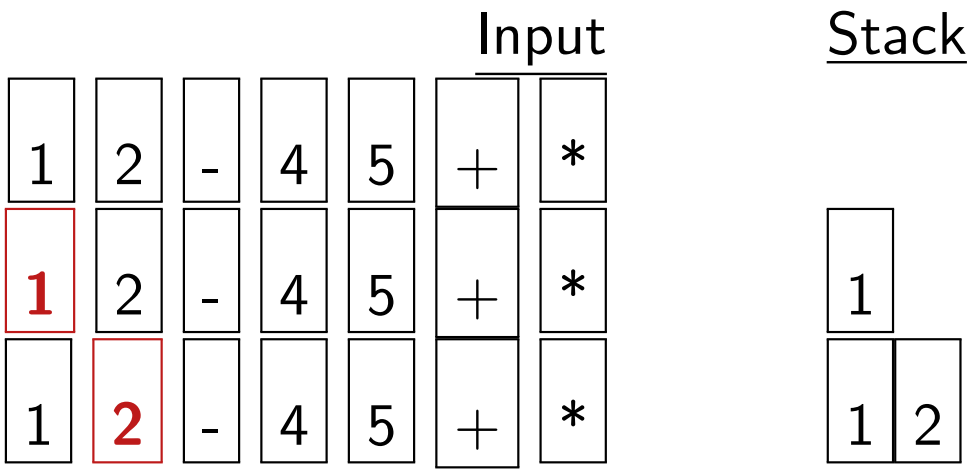
Illustration



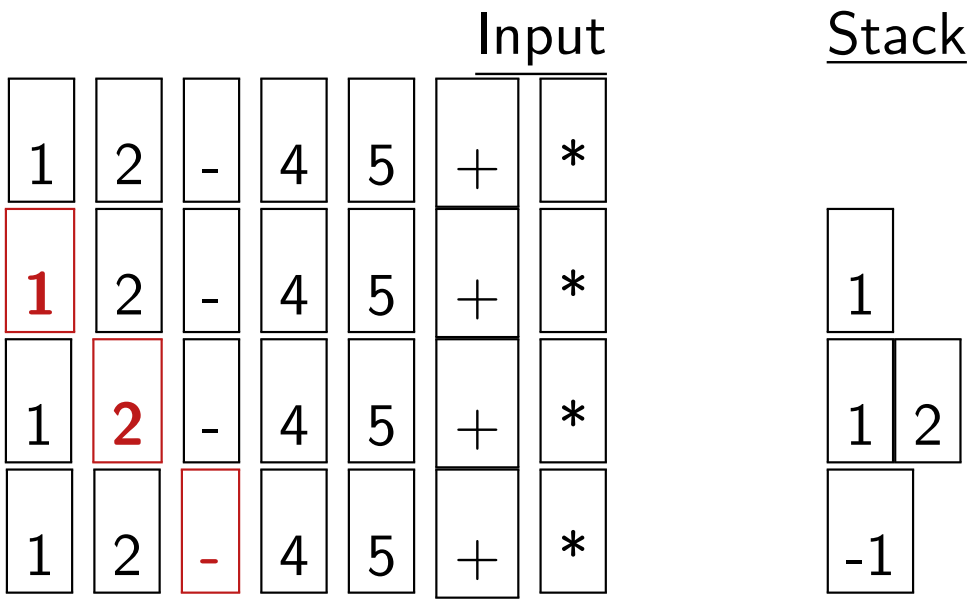
Illustration



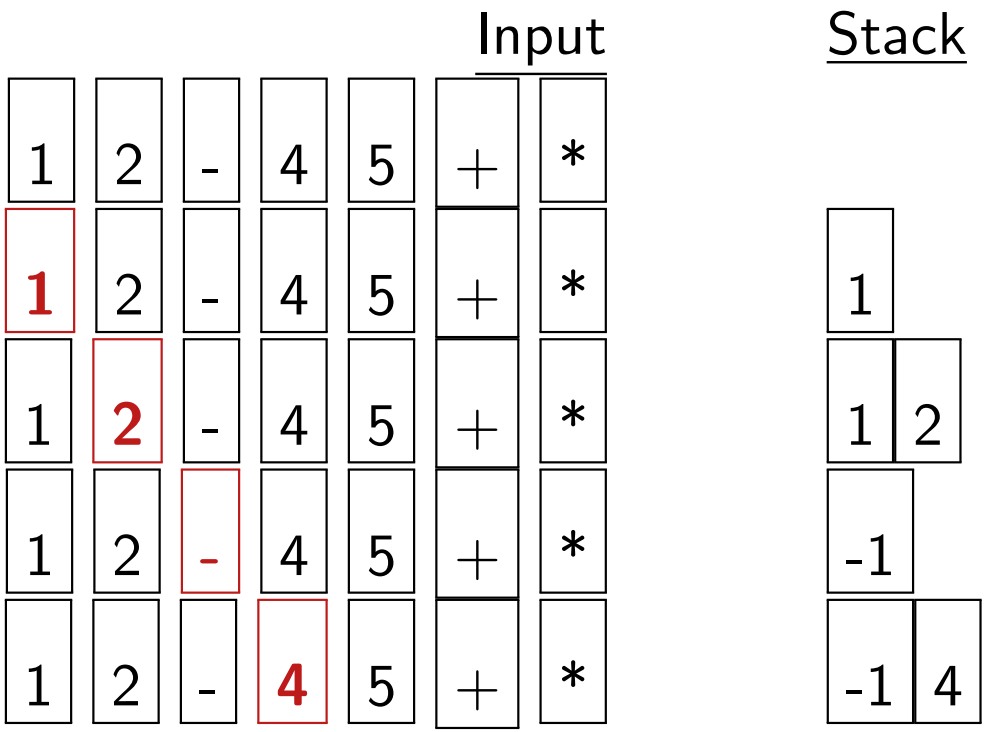
Illustration



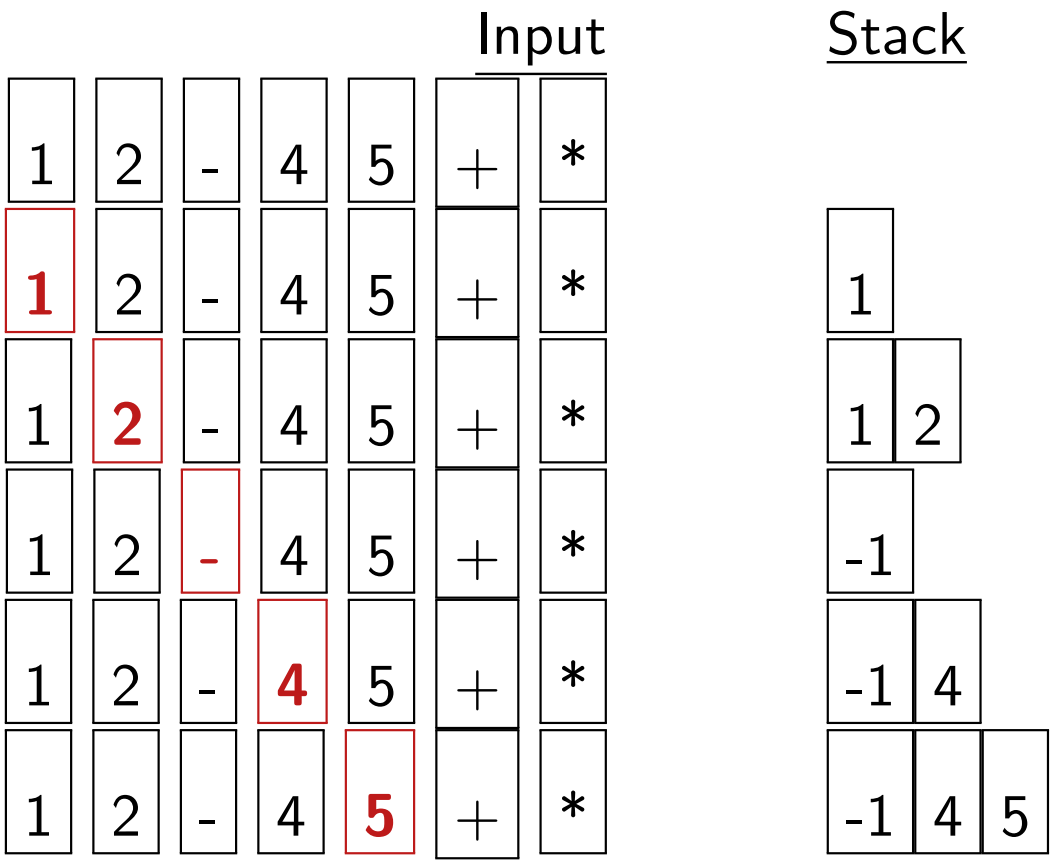
Illustration



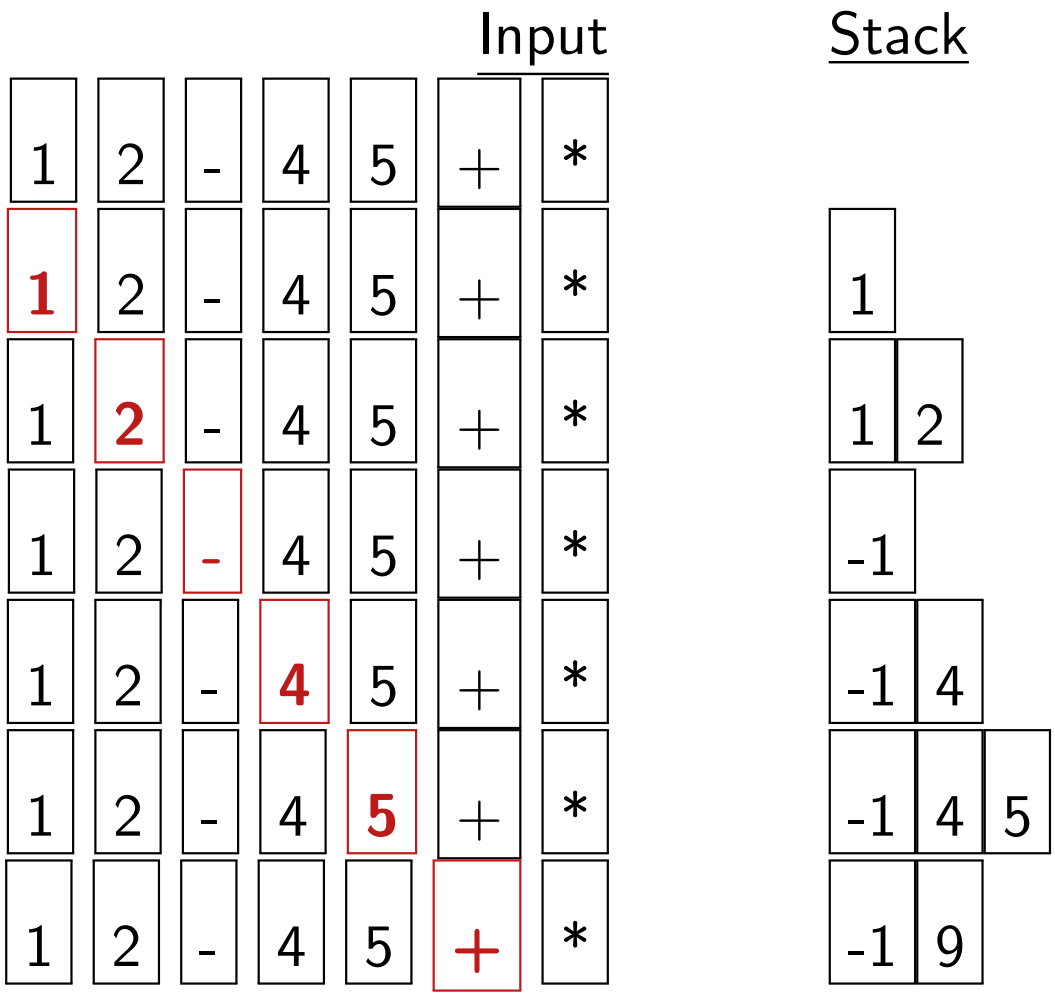
Illustration



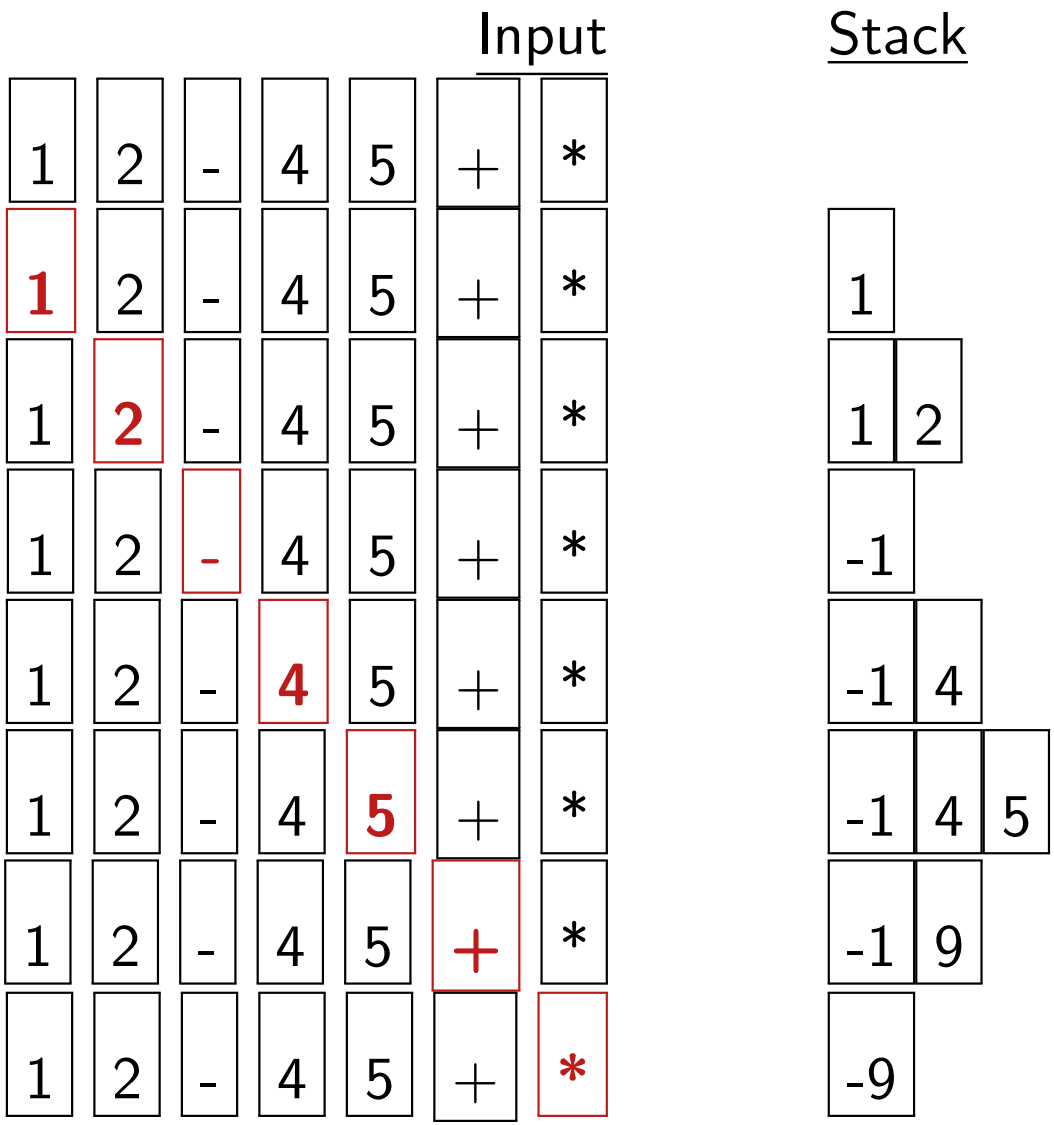
Illustration



Illustration



Illustration



Tokens

- Assume expression presented as a String
- Expression regarded as a sequence of *tokens* (also Strings)
- Our tokens are
  - *operands* – unsigned integers (*e.g.* "123")
  - *operators* – "+", "−", "∗", "/"
- "Whitespace" (spaces, tabs) between tokens is ignored
- Expression is assumed to be syntactically valid

ExprScanner

- Assume we have ExprScanner object that takes expression as String, *e.g.*

  `"1  2  +"`

  and "decomposes" it into its constituent tokens as String *e.g.*

  `"1"  "2"  "+"`

  .

- ExprScanner Operations:

  hasNext(): Return true if one or more tokens in expression remain unread and false otherwise. *Input: None*; *Output: boolean*.
  next(): Return the next unread token from the expression. Illegal if no tokens remain. *Input: None*; *Output: String (token)*.

Postfix Calculator – 1

Issue Iterating through the tokens one by one

Note e is the expression in String form

Code

```
expr ←new ExprScanner(e)
while expr.hasNext() do
    currTkn ←expr.next()
    /* handle currentTkn */
```

Postfix Calculator –2

Issue Handling token currenTkn

Code

```
if currTkn is a "+" then
  /* handle + operator */
else
if currTkn is a "-" then
  /* handle - operator */
else
  /* similar handling of * and / operators */
else
  /* handle operand */
```

Handling an Operator

Action Pop top two operands off stack, apply operator to them, and push the result back onto the stack

Code (for +)

```
secondOpnd ←opStack.pop()
firstOpnd ←opStack.pop()
opStack.push(firstOpnd + secondOpnd)
```

Note code for other ops similar

Handling an Operand

Action required Push value of operand onto the stack

Pseudocode

```
firstOpnd ←int value of currTkn
opStack.push(firstOpnd);
```

Putting It Together

```
Algorithm Evaluate(e)
    expr ←new ExprScanner(e)
    while expr.hasNext() do
        currTkn ←expr.next()
        if currTkn is a "+" then
            secondOpnd ←opStack.pop()
            firstOpnd ←opStack.pop()
            opStack.push(firstOpnd + secondOpnd)
        else
            /* similarly for "−", "*" and "/" */
        else
            firstOpnd ←int value of currTkn
            opStack.push(firstOpnd );
    return opStack.pop();
```

Translating Infix Into Postfix*

Create an empty stack. Process tokens as follows:

- Operand: print it immediately
- Left parentesis: push it onto stack
- Right parenthesis: Pop (and print) symbols off stack until (i) a left parenthesis appears which is popped (but not printed), or (ii) stack is empty.
- Operators: Pop (and print) symbols off stack until (i) an operator of *lower precedence* [2] appears on top, or (ii) stack is empty, and then push the current operator

Symbols on stack at end are popped and printed.

[2] Left parentheses have the lowest precedence, then additive operators (plus and minus), then multiplicative operators (times and division)

Illustration 1*

Expression $1 + 2 + 3$

Translation Process

Current Token	Output	Stack
1	1	
+		+
2	2	+
+	+	+
3	3	+
(end)	+	

The output (read top to botton)

$$1 \quad 2 \quad + \quad 3 \quad +$$

is the postfix expression.

Illustration 2*

Expression $(1 + 2) * 3$

Translation Process

Current Token	Output	Stack
(		(
1	1	(
+		(+
2	2	(+
)	+	
*		*
3	3	*
(end)	*	

The output (read top to botton) 1 2 + 3 * is the postfix expression.

Observation

Coupled with a postfix evaluator, the infix-to-postfix translator allows infix expressions to be evaluated.