2/2

Stack and Queue

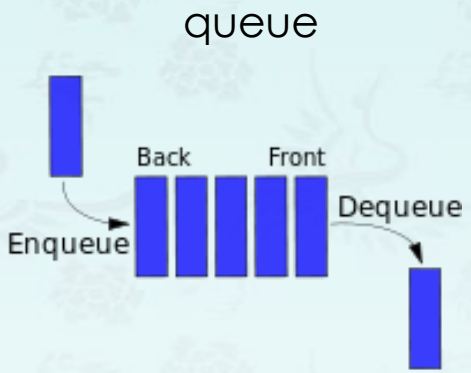
Data Structures
C++ for C Coders

한동대학교 김영섭 교수
idebtor@gmail.com

*applications - **infix to postfix***

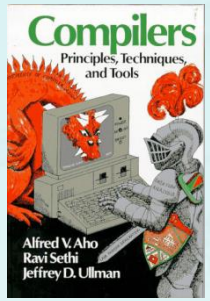
Queues

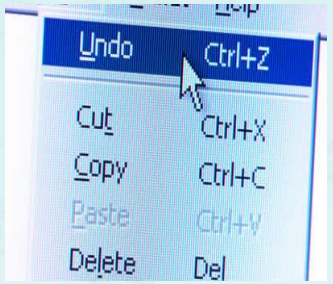
Queue: An ordered list in which **enqueues** (insertion or add) at the **rear** and **dequeues** (deletion or remove) take place at different end or **front**.
It is also known as a Fist-in-first-out(FIFO) list.

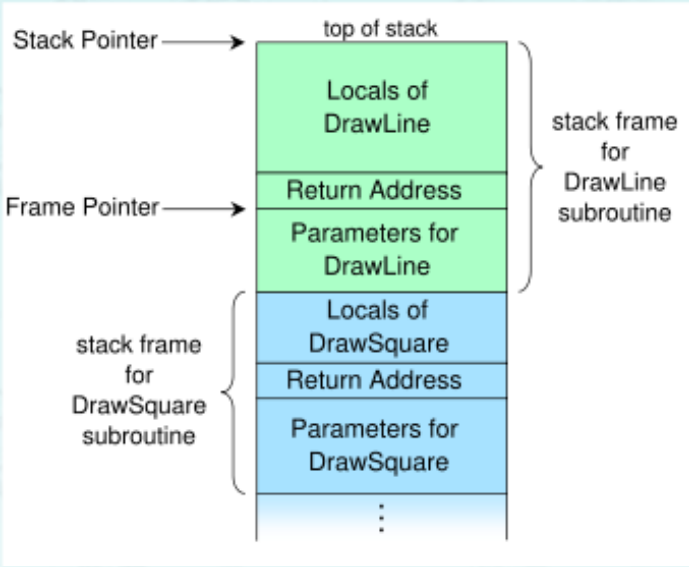


- Items can only be added at the **rear** of the queue and the only item that can be removed is the one at the **front** of the queue.

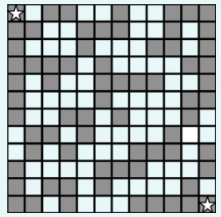
Stack and Queue Applications







- Parsing in a compiler. (p.127)
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Backtracking as in a maze (p.121)
- Implementing function calls in a compiler. (p.108)
- ...



Stack and Queue Applications

In a computer OS: Requests for services come in unpredictable order and timing, sometimes faster than they can be serviced .

- print a file
- need a file from the disk system
- send an email
- job scheduling

Arithmetic expression evaluation

Goal: Convert an **infix** expression to a **postfix** expression using a **stack**.

(1 + 2) * 3

operand operator

Stack: (Output:
Stack: (Output: 1
Stack: (+   Output: 1
Stack: (+   Output: 1 2
Stack: Output: 1 2 +
Stack: *   Output: 1 2 +
Stack: *   Output: 1 2 + 3
Stack: Output: 1 2 + 3 *

- Operands are output immediately
- Stack operators until right parens
- Unstack until left parens
  Delete left parens
- In general, higher precedence operator must be output before lower one.)

infix **(1 + ((2 + 3) * (4 * 5)))**

postfix

Arithmetic expression evaluation

Goal: Convert an **infix** expression to a **postfix** expression using a **stack**.

(1 + 2) * 3

operand operator

Stack: (Output:
Stack: (Output: 1
Stack: (+ Output: 1
Stack: (+ Output: 1 2
Stack: Output: 1 2 +
Stack: * Output: 1 2 +
Stack: * Output: 1 2 + 3
Stack: Output: 1 2 + 3 *

- Operands are output immediately
- Stack operators until right parens
- Unstack until left parens
  Delete left parens
- In general, higher precedence operator must be output before lower one.)

infix **(1 + ((2 + 3) * (4 * 5)))**

postfix **1 2 3 + 4 5 * * +**

Arithmetic expression evaluation

Goal: Evaluate infix expressions.

Two-stack algorithm. [E. W. Dijkstra]

Arithmetic expression evaluation

Goal: Evaluate infix expressions.

Two-stack algorithm. [E. W. Dijkstra]

- **Value**: push onto the **value stack.**
- **Operator:** push onto the **operator stack**.
- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.

Dijkstra's two-stack algorithm

value stack
(operand stack)

operator stack

infix expression
(fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra's two-stack algorithm

- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.

value stack
(operand stack)

operator stack

infix expression
(fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra's two-stack algorithm

- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.

value stack
(operand stack)

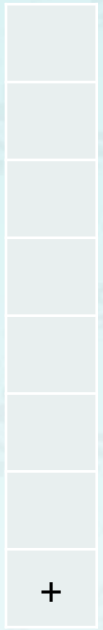
operator stack

infix expression
(fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra's two-stack algorithm

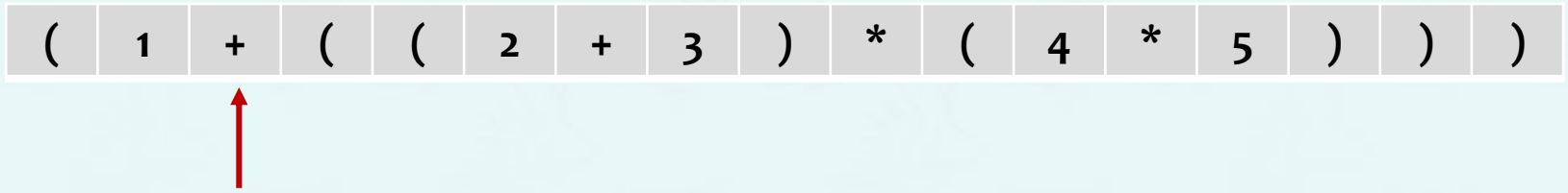
- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.



value stack
(operand stack)

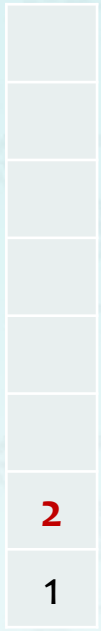
operator stack

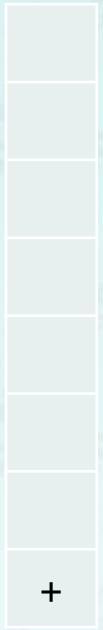
infix expression
(fully parenthesized)



Dijkstra's two-stack algorithm

- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.

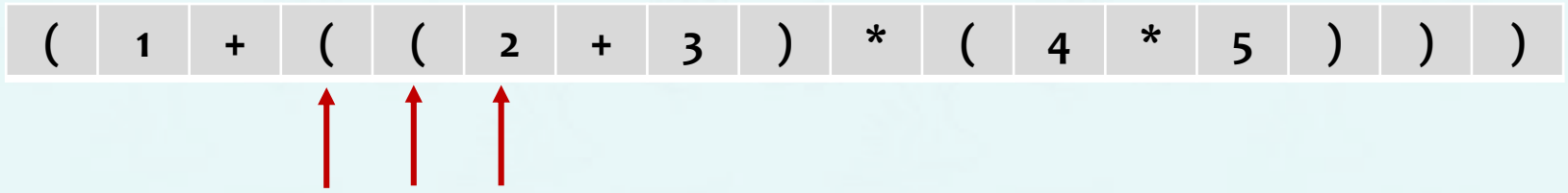




value stack
(operand stack)

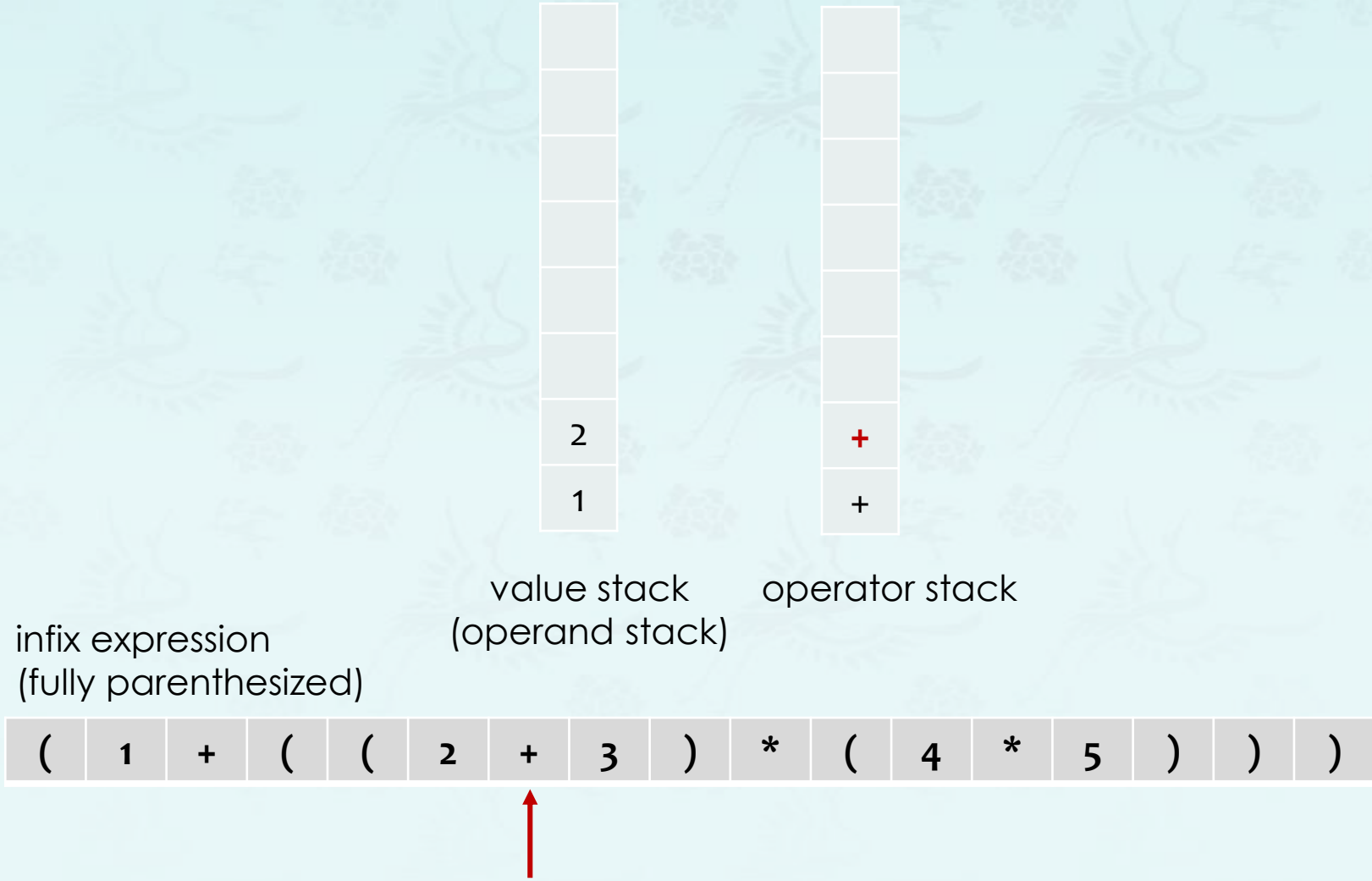
operator stack

infix expression
(fully parenthesized)



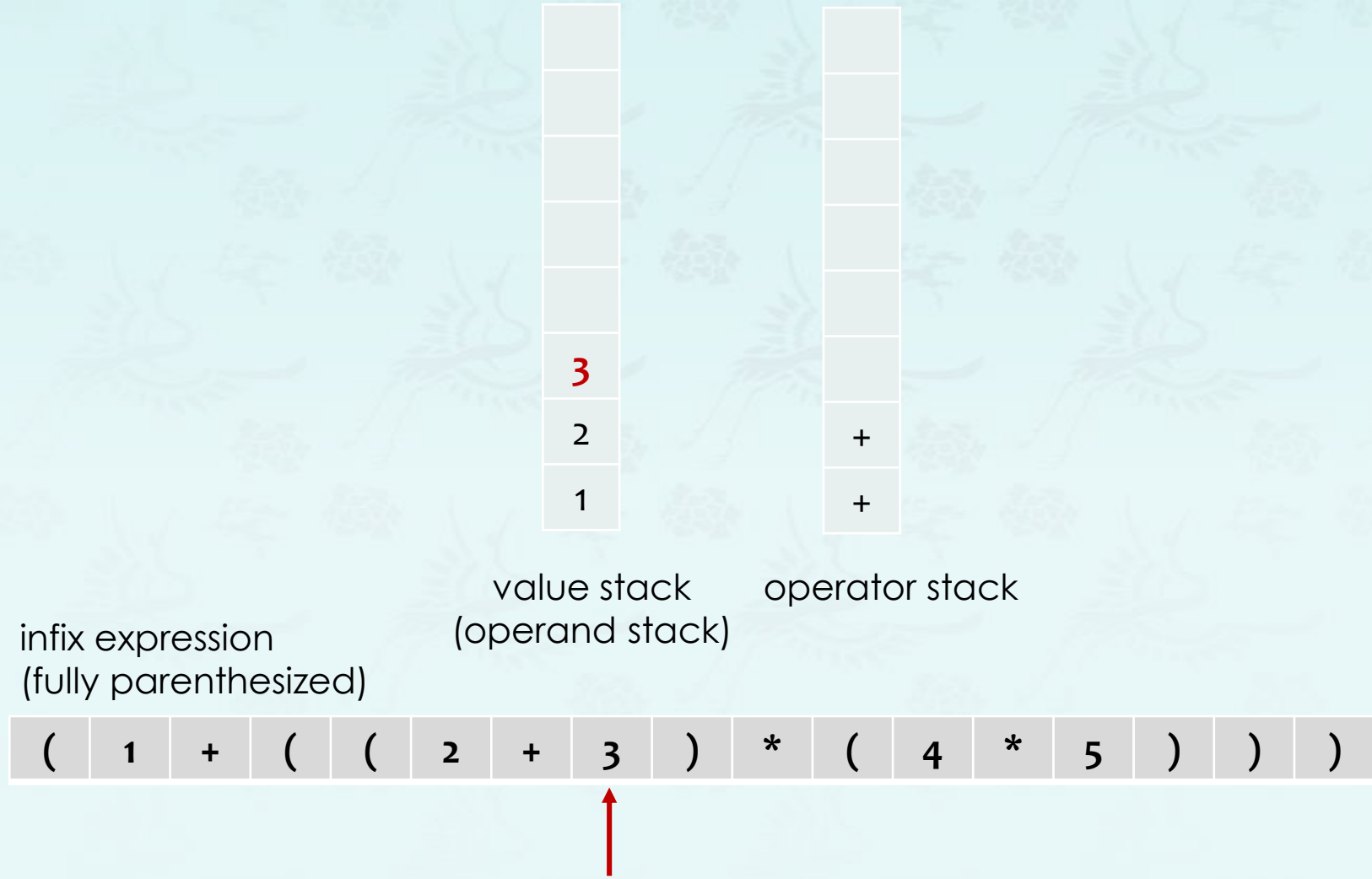
Dijkstra's two-stack algorithm

- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.



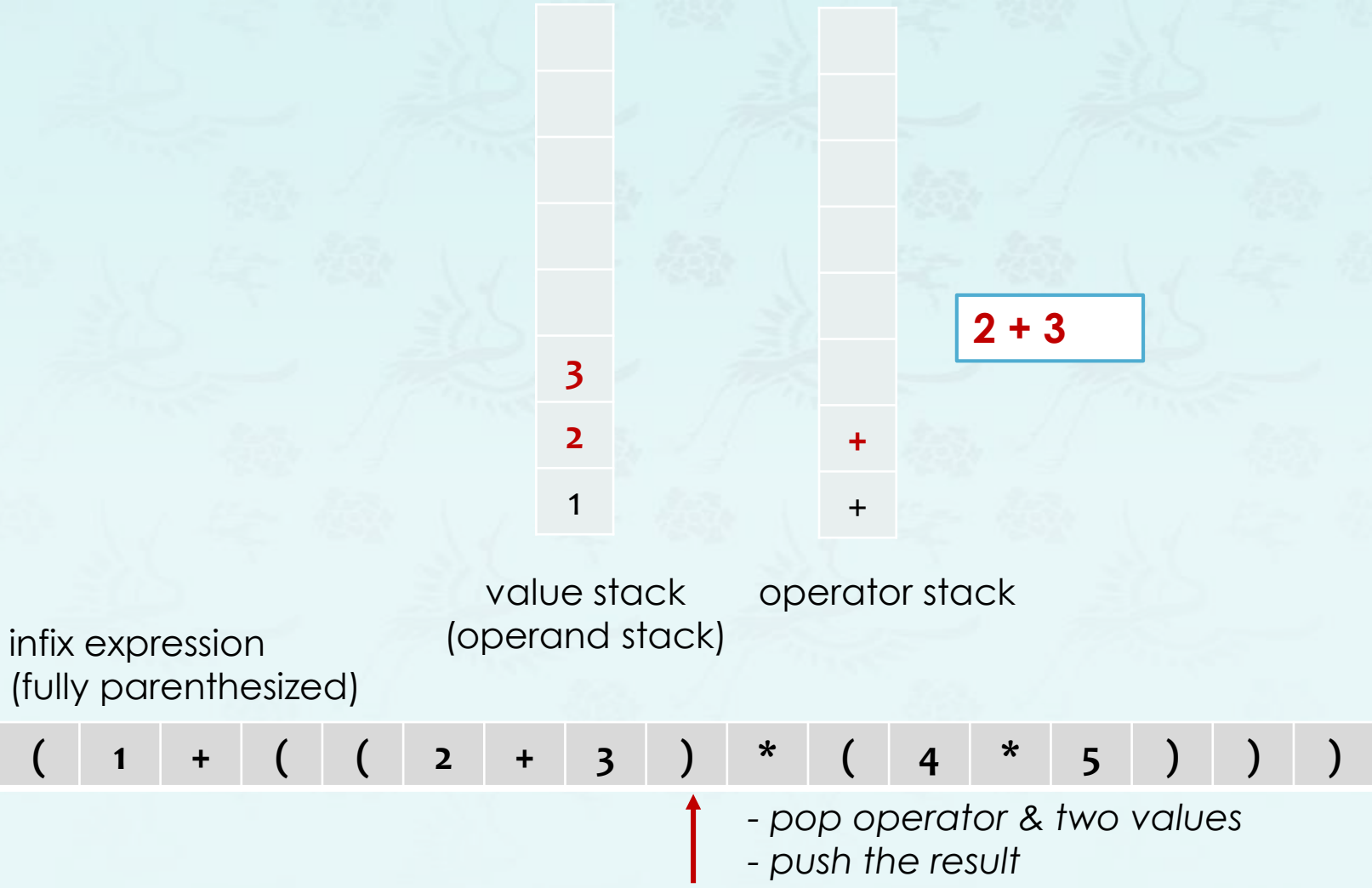
Dijkstra's two-stack algorithm

- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.



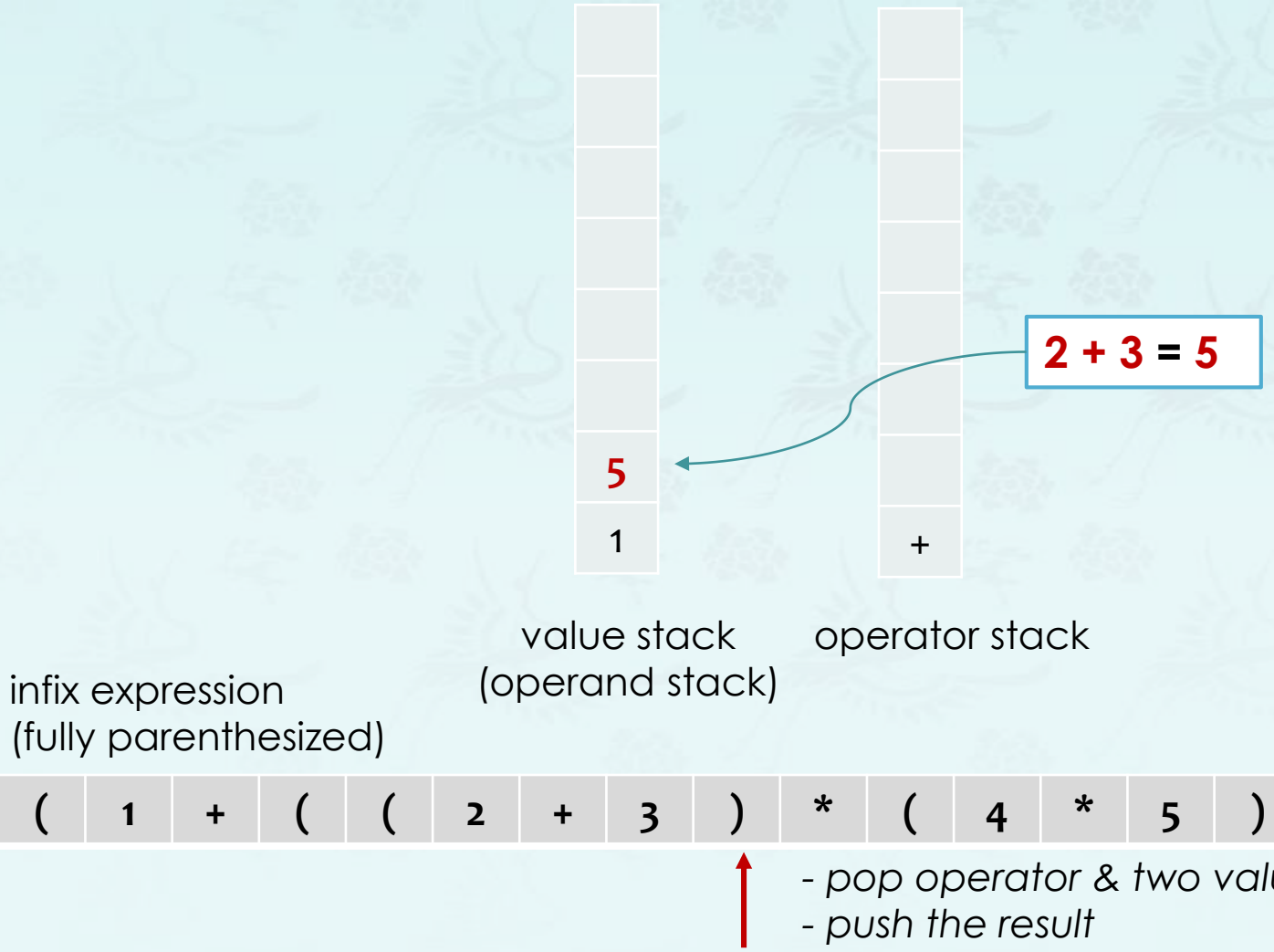
Dijkstra's two-stack algorithm

- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.



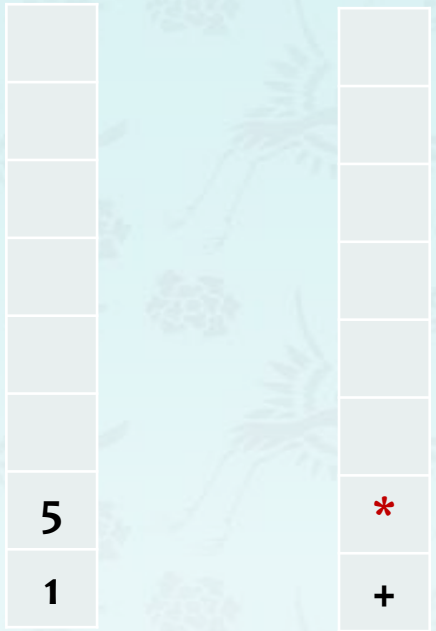
Dijkstra's two-stack algorithm

- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.



Dijkstra's two-stack algorithm

- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.



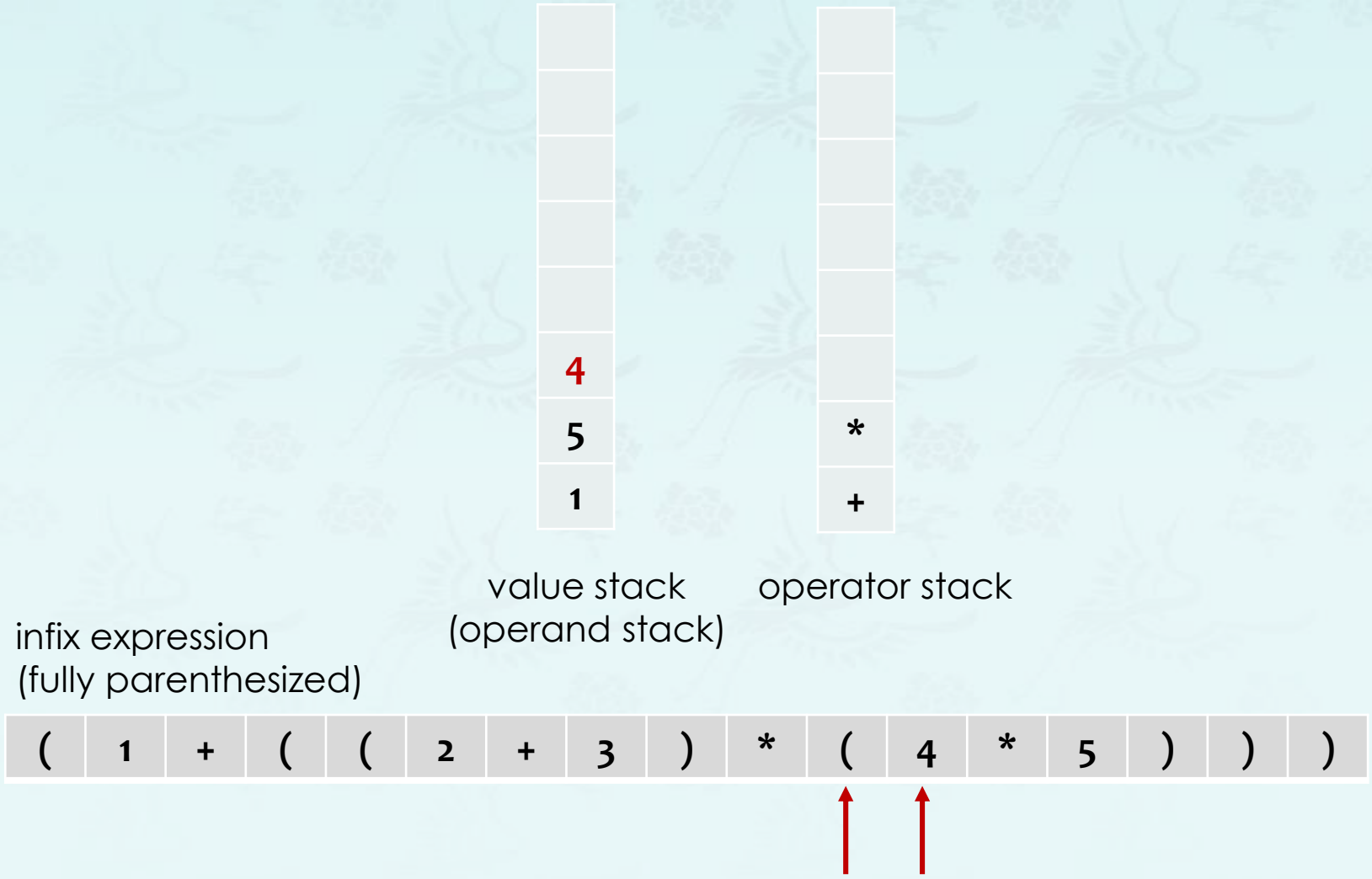
value stack (operand stack)

operator stack

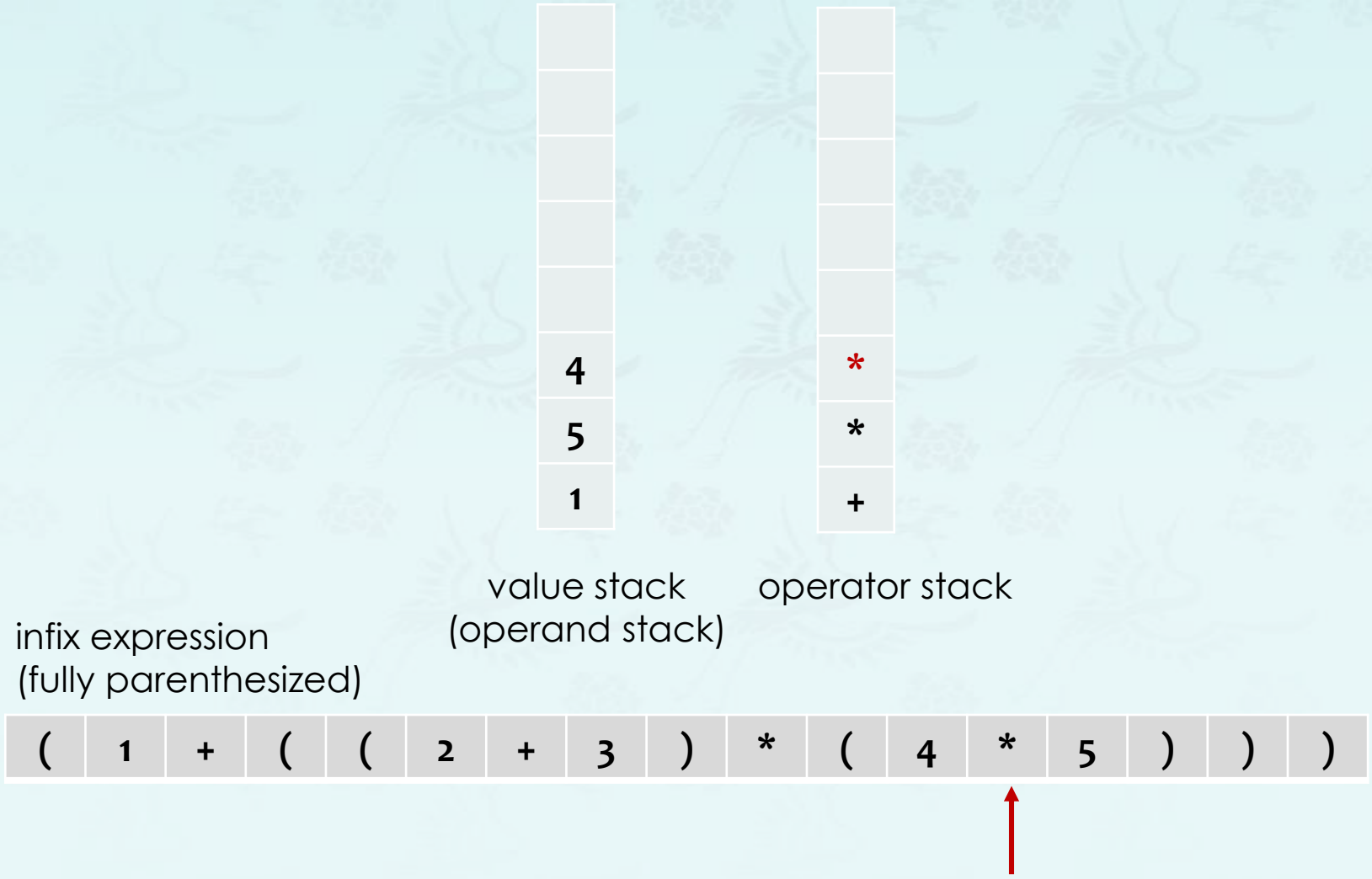
infix expression (fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

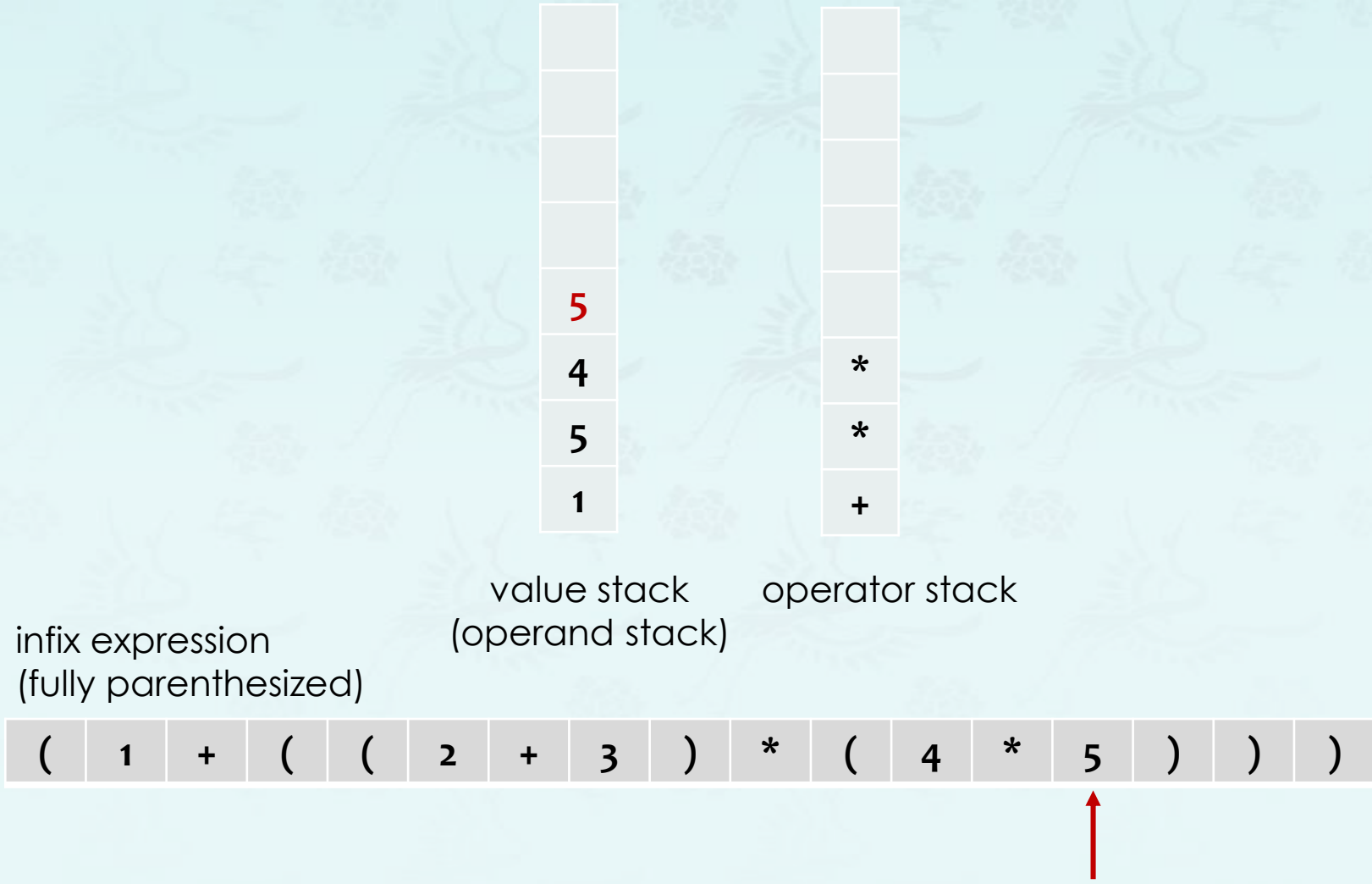
Dijkstra's two-stack algorithm



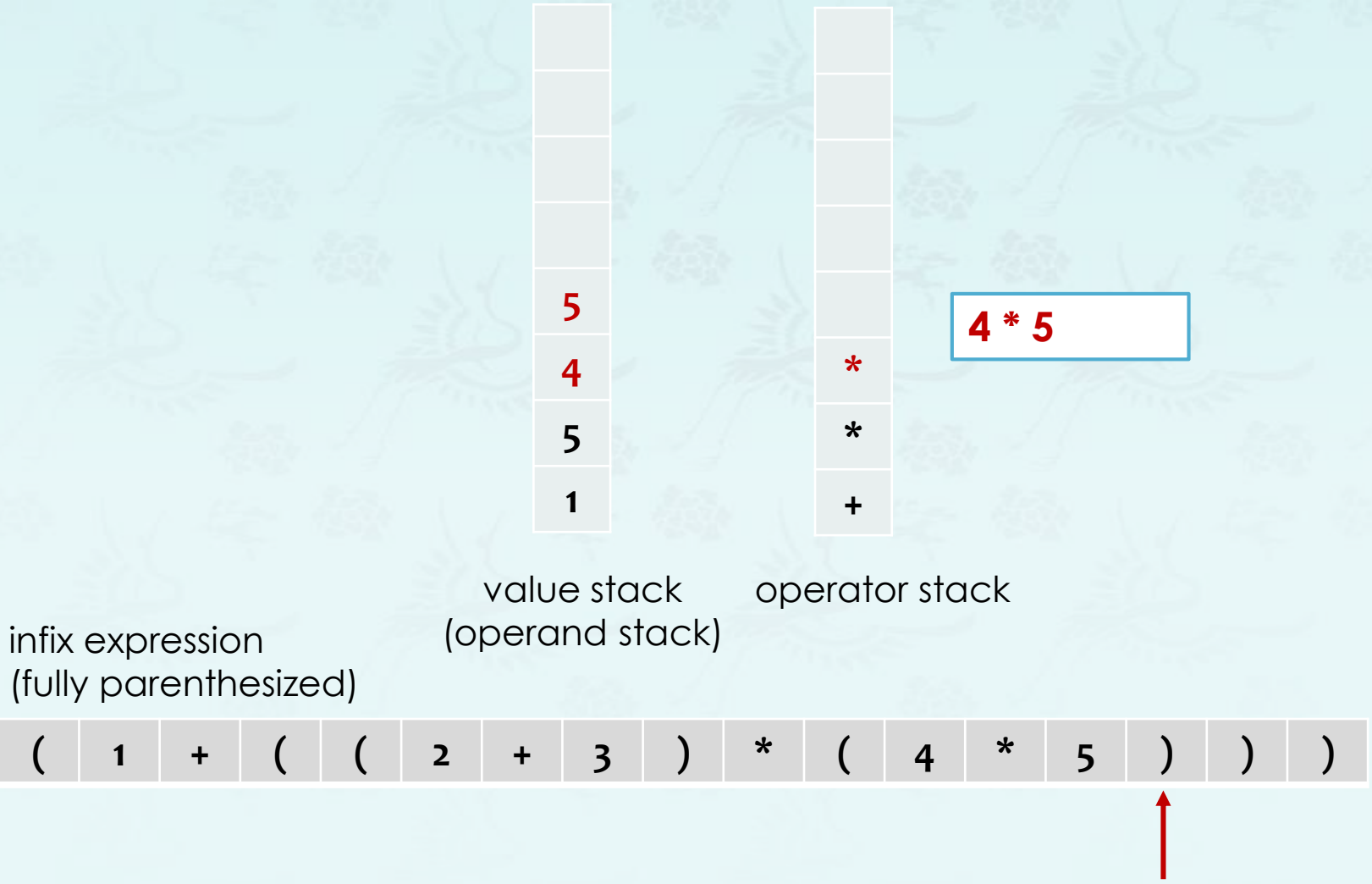
Dijkstra's two-stack algorithm



Dijkstra's two-stack algorithm

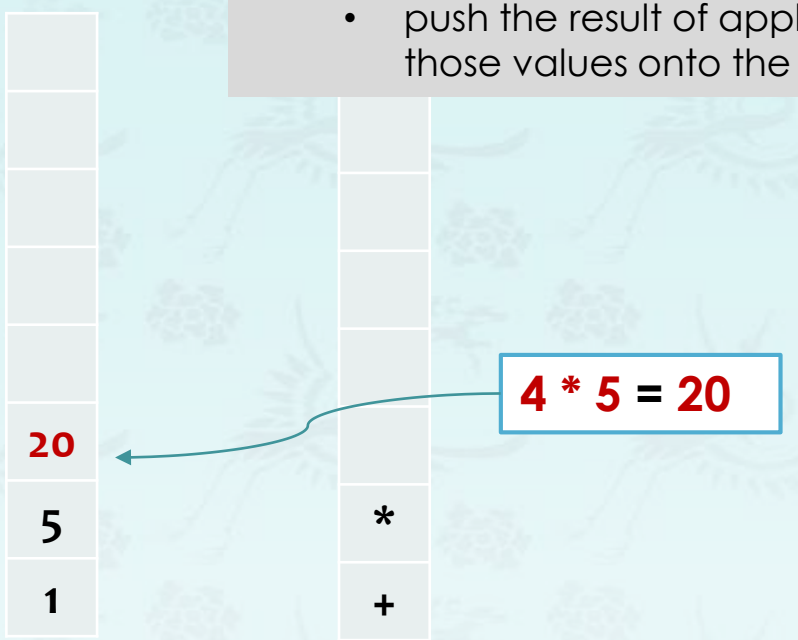


Dijkstra's two-stack algorithm



Dijkstra's two-stack algorithm

- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.



value stack (operand stack)

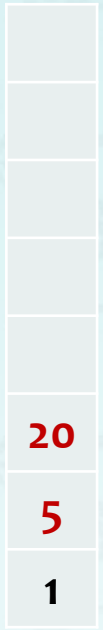
operator stack

infix expression (fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra's two-stack algorithm

- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.



5 * 20

value stack
(operand stack)

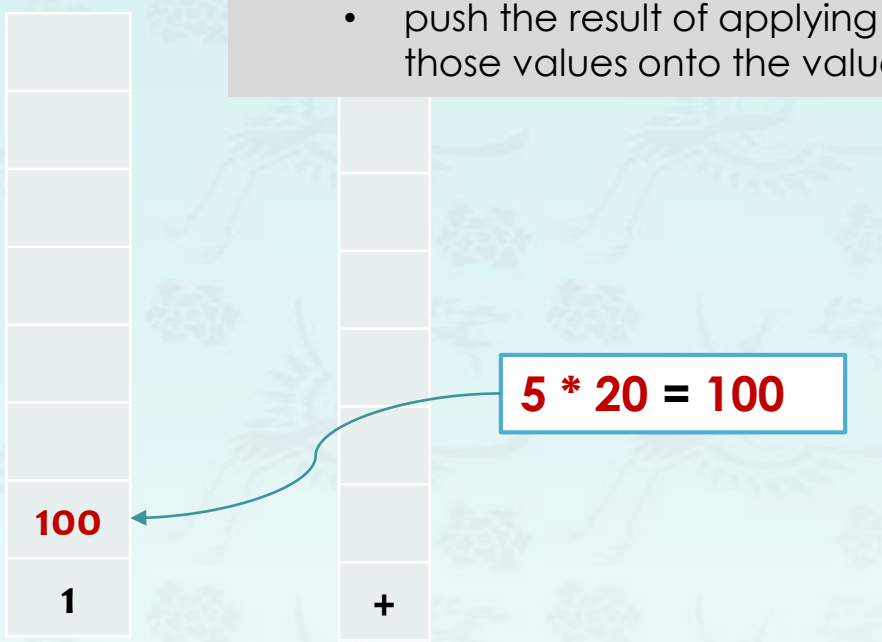
operator stack

infix expression
(fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra's two-stack algorithm

- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.



value stack (operand stack)

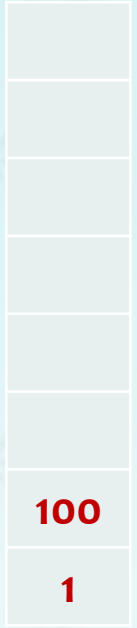
operator stack

infix expression (fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra's two-stack algorithm

- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.

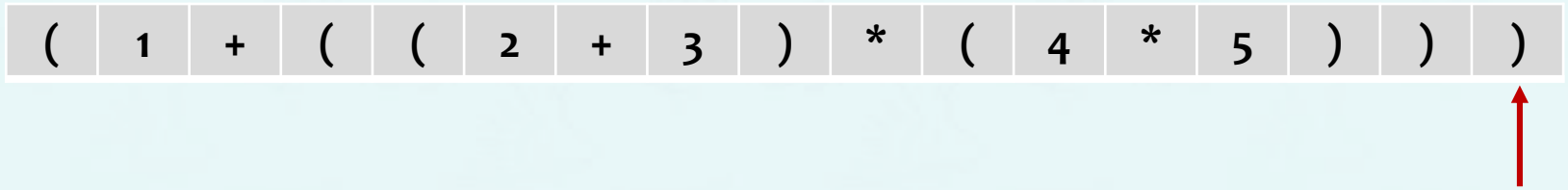


1 + 100

value stack
(operand stack)

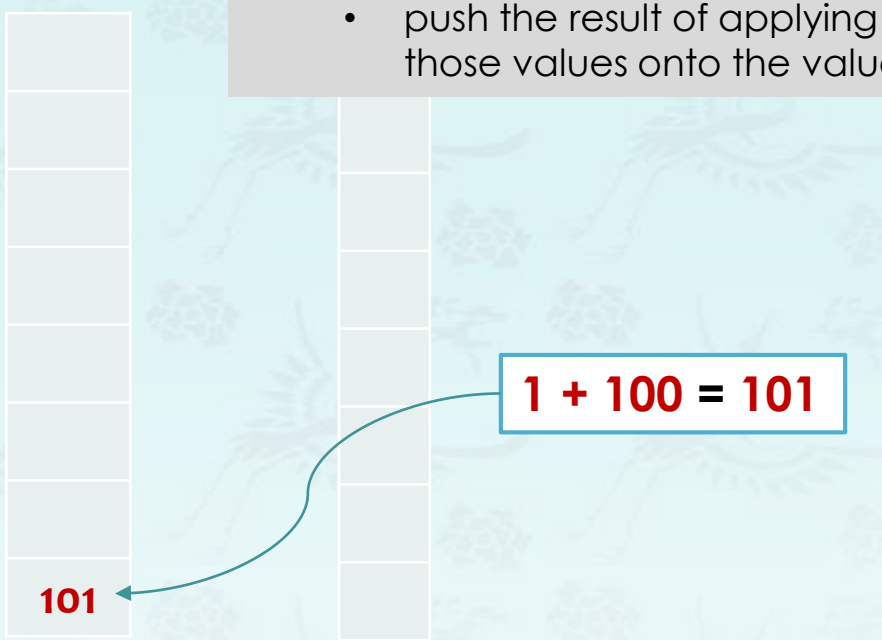
operator stack

infix expression
(fully parenthesized)



Dijkstra's two-stack algorithm

- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.



value stack
(operand stack)

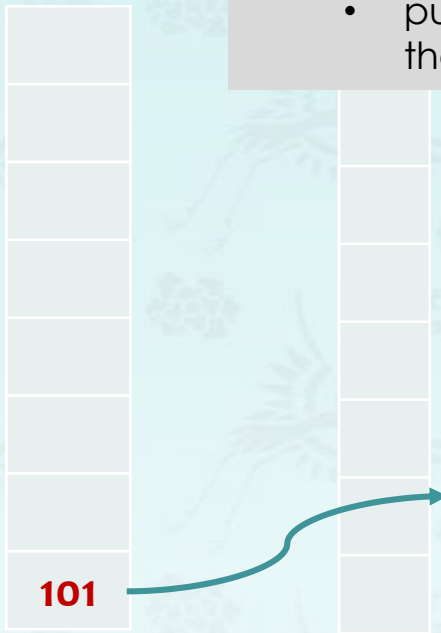
operator stack

infix expression
(fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra's two-stack algorithm

- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.



Empty the stack for the result at the end of expression

value stack
(operand stack)

operator stack

infix expression
(fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Arithmetic expression evaluation

Q: How does it work?

A: When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

as if the original input were:

```
( 1 + ( 5 * ( 4 * 5 ) ) )
```

Repeating the argument:

```
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

Extensions: More ops, precedence order, associativity.

Arithmetic expression evaluation

Observation 1. Dijkstra's two-stack algorithm computes the same value if the operator occurs **after** the two values.

(1 + ((2 + 3) * (4 * 5)))

(1 ((2 3 +) (4 5 *) *) +)

Observation 2.

Arithmetic expression evaluation

Observation 1. Dijkstra's two-stack algorithm computes the same value if the operator occurs **after** the two values.

(1 + ((2 + 3) * (4 * 5)))

(1 ((2 3 +) (4 5 *) *) +)

Observation 2. All of the parentheses are redundant!

Arithmetic expression evaluation

Observation 1. Dijkstra's two-stack algorithm computes the same value if the operator occurs **after** the two values.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

```
( 1 ( ( 2 3 + ) ( 4 5 * ) * ) + )
```

Observation 2. All of the parentheses are redundant!

```
1 2 3 + 4 5 * * +
```

Bottom line: Postfix or "reverse Polish" notation.
Applications: Postscript, calculators, JVM, ...

Dijkstra's two-stack algorithm

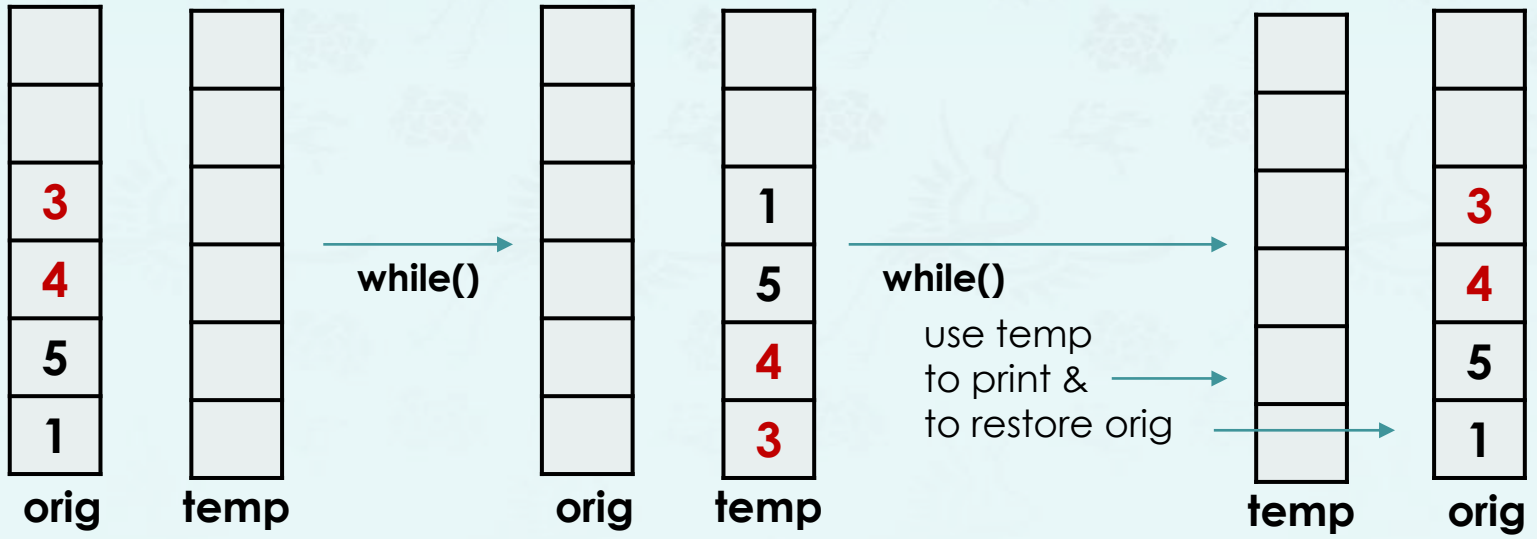
```
public class ArithmeticExpression {
  public static void main(String[] args) {
    Stack<Character> ops = new Stack<Character>();
    Stack<Double> vals = new Stack<Double>();
    String e = JOptionPane.showInputDialog(null,
      "Enter an expression", "Stack application", JOptionPane.QUESTION_MESSAGE);
    if (e == null) return;           // Check "Cancel"

    for (int i = 0; i < e.length(); i++) {
      Character c = e.charAt(i);
      if (c.equals(' ') || c.equals('(')) ;
      else if (c == '+') ops.push(c);
      else if (c == '*') ops.push(c);
      else if (c == ')') {
         Character op = ops.pop();
         if     (op.equals('+')) vals.push(vals.pop() + vals.pop());
         else if(op.equals('*')) vals.push(vals.pop() * vals.pop());
      }
      else {
         String s = "" + c;
         vals.push(Double.parseDouble(s));
      }
    }
    JOptionPane.showMessageDialog(null, e + " = " + vals.pop());
  }
}
```

printStack()

This function prints the contents of a stack from the bottom to top. The stack contents should be the same as before after printing.

Algorithm:

- Given a stack called **orig**.
- Create an empty stack called **temp.**
- While **orig** is not empty,
  - Top/Pop push an item from **orig** to **temp**.
- While **temp** is not empty,
  - Top/Pop an item from **temp**, print it and push it **orig**.



Task A: infix.cpp

```
1    While there are still tokens to be read in,
     1.1    Get the next token.
     1.2    If the token is:
          1.2.1  A space: ignore it
          1.2.2  A left brace: ignore it
          1.2.3  A number:
               1.2.3.1 read the number (it could be a multiple digit.)
               1.2.3.2 push it onto the value stack
          1.2.4  A right parenthesis:
               1.2.4.1 Pop the operator from the operator stack.
               1.2.4.2 Pop the value stack twice, getting two operands.
               1.2.4.3 Apply the operator to the operands, in the correct order.
               1.2.4.4 Push the result onto the value stack.
          1.2.5  An operator
               1.2.5.1 Push the operator to the operator stack
2    (The whole expression has been parsed at this point.
     Apply remaining operators in the op stack to remaining values in the value stack)
     While the operator stack is not empty,
     2.1    Pop the operator from the operator stack.
     2.2    Pop the value stack twice, getting two operands.
     2.3    Apply the operator to the operands, in the correct order.
     2.4    Push the result onto the value stack.
3    (At this point the operator stack should be empty, and the value stack should have only
     one value in it, which is the result.)
     Return the top item in the value stack.
```

Task B: infixall.cpp

1 While there are still tokens to be read in,
  1.1 Get the next token.
  1.2 If the token is:
    1.2.1 A space: ignore it
    1.2.2 A left brace: push it onto the operator stack.
    1.2.3 A number:
      1.2.3.1 read the number (it could be a multiple digit.)
      1.2.3.2 push it onto the value stack
    1.2.4 A right parenthesis:
      1.2.4.1 While the item on top of the operator stack is not a left brace,
        1.2.4.1.1 Pop the operator from the operator stack.
        1.2.4.1.2 Pop the value stack twice, getting two operands.
        1.2.4.1.3 Apply the operator to the operands, in the correct order.
        1.2.4.1.4 Push the result onto the value stack.
      1.2.4.2 Pop the left brace from the operator stack and discard it.
    1.2.5 An operator (let's call it **thisOp**)
      1.2.5.1 While the operator stack is not empty, and the top item on the operator stack has the same or greater precedence as thisOp,
        1.2.5.1.1 Pop the operator from the operator stack
        1.2.5.1.2 Pop the value stack twice, getting two values
        1.2.5.1.3 Apply the operator to two values in the correct order
        1.2.5.1.4 Push the result on the value stack
      1.2.5.2 Push the operator (**thisOp**) onto the operator stack
2 (The whole expression has been parsed at this point.
  Apply remaining operators in the op stack to remaining values in the value stack)
  While the operator stack is not empty,
  2.1 Pop the operator from the operator stack.
  2.2 Pop the value stack twice, getting two values.
  2.3 Apply the operator to two values, in the correct order.
  2.4 Push the result onto the value stack.
3 (At this point the operator stack should be empty, and the value stack should have only one value in it, which is the result.)
  Return the top item in the value stack.

Arithmetic expression evaluation

infix	postfix
2 + 3 * 4	2 3 4 * +
a * b + 5	a b * 5 +
(1 + 2) * 7	
a * b / c	
(a / (b – c + d)) * (e – a) * c	
a / b – c + d * e – a * c	

Infix and postfix notation

Arithmetic expression evaluation

infix	postfix
2 + 3 * 4	2 3 4 * +
a * b + 5	a b * 5 +
(1 + 2) * 7	1 2 + 7 *
a * b / c	a b * c /
(a / (b – c + d)) * (e – a) * c	a b c – d + / e a – * c *
a / b – c + d * e – a * c	a b / c – d e * + a c * –

Infix and postfix notation

Arithmetic expression evaluation

infix	postfix
2 + 3 * 4	2 3 4 * +
a * b + 5	a b * 5 +
	1 2 + 7 *
	a b * c /
	a b c – d + / e a – * c *
	a b / c – d e * + a c * –

Infix and postfix notation

Arithmetic expression evaluation

Goal: Evaluate postfix expressions.

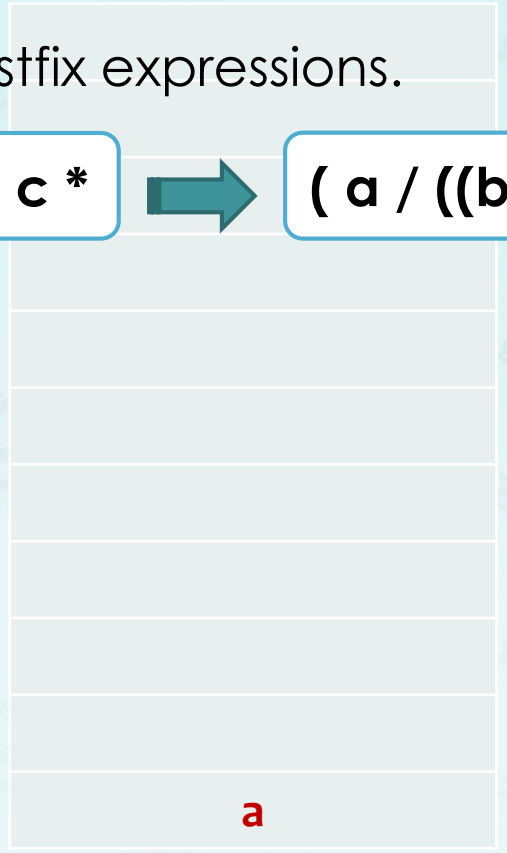
a b c – d + / e a – * c * ⇒ (a / ((b – c) + d)) * (e – a) * c

a	b	c	-	d	+	/	e	a	-	*	c	*

Arithmetic expression evaluation

Goal: Evaluate postfix expressions.

a b c – d + / e a – * c * ⇒ (a / ((b – c) + d)) * (e – a) * c



value stack
(operand stack)

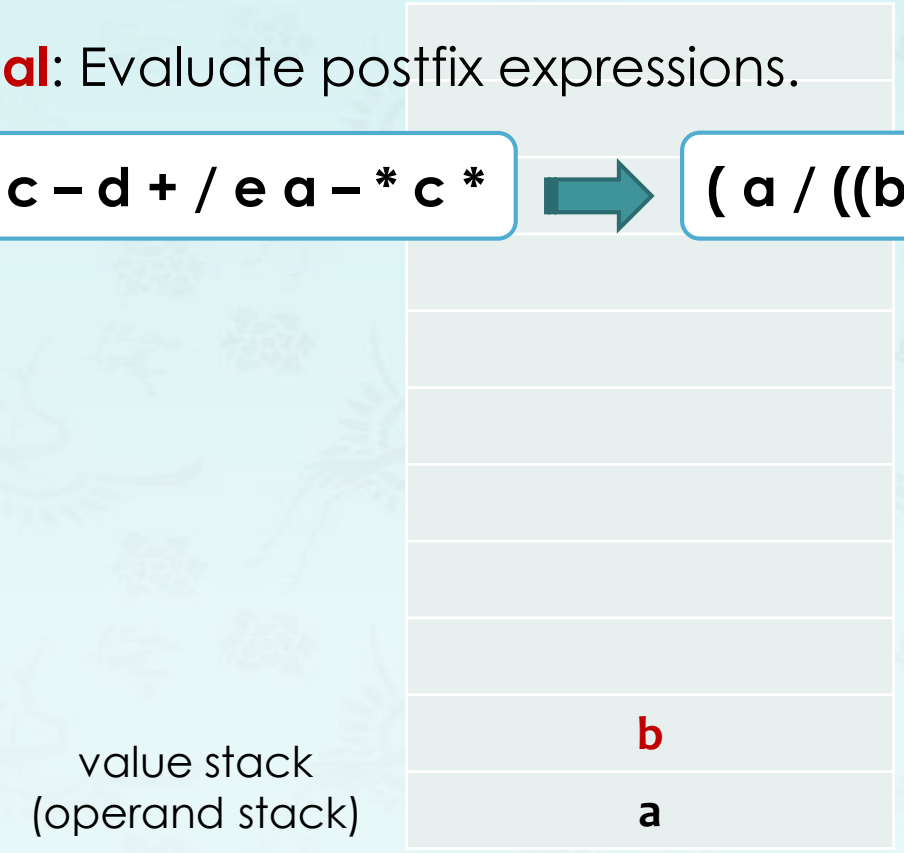
a	b	c	-	d	+	/	e	a	-	*	c	*

push the operands
until an operator comes up.

Arithmetic expression evaluation

Goal: Evaluate postfix expressions.

a b c – d + / e a – * c * ⇒ (a / ((b – c) + d)) * (e – a) * c

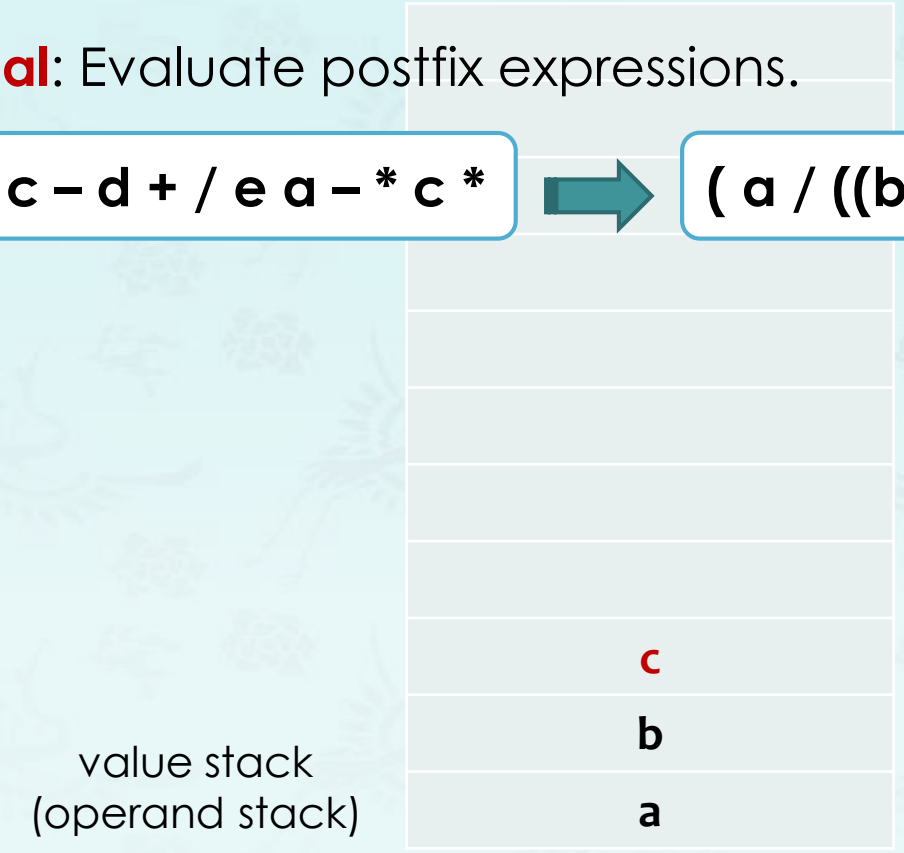


a	b	c	-	d	+	/	e	a	-	*	c	*

Arithmetic expression evaluation

Goal: Evaluate postfix expressions.

a b c – d + / e a – * c * ⇒ (a / ((b – c) + d)) * (e – a) * c

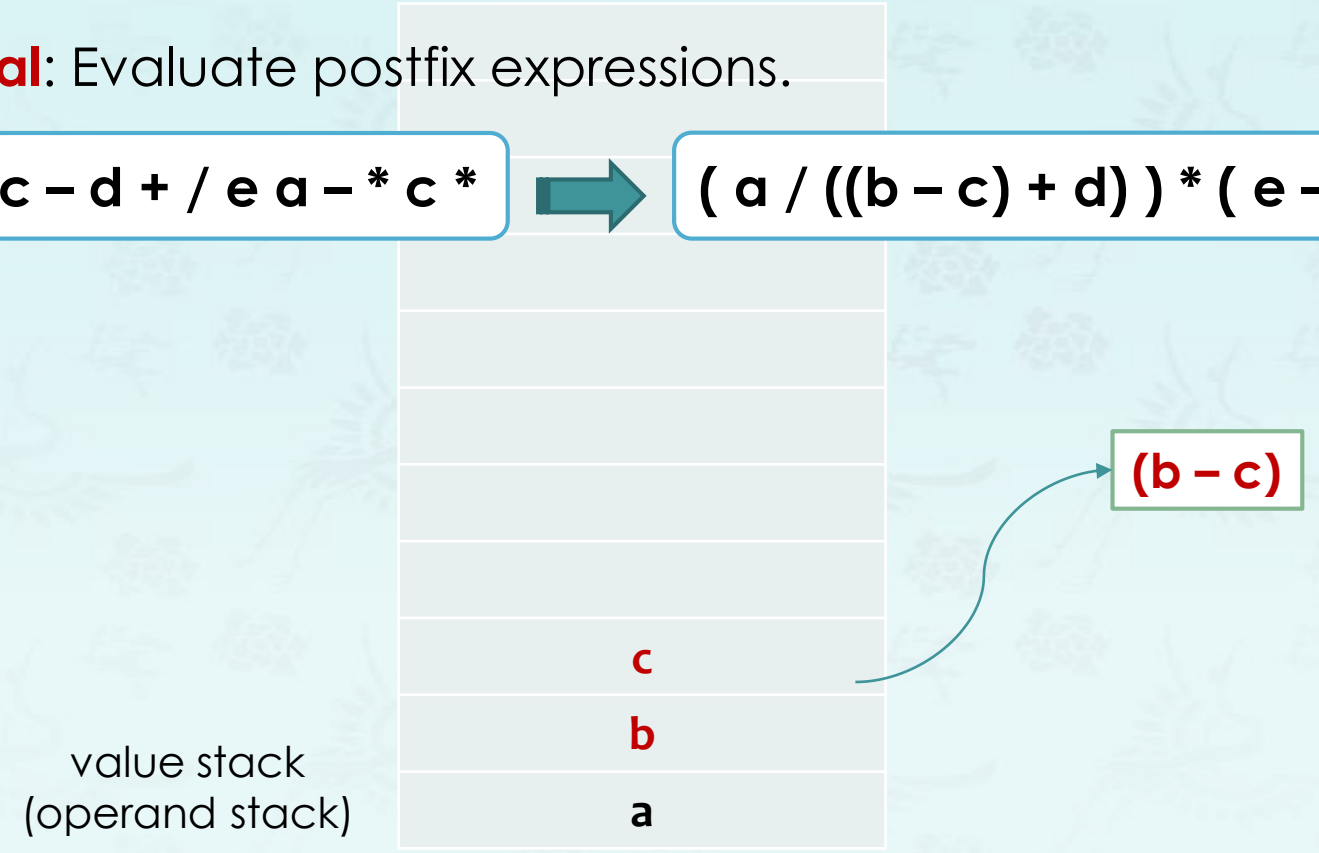


a	b	c	-	d	+	/	e	a	-	*	c	*

Arithmetic expression evaluation

Goal: Evaluate postfix expressions.

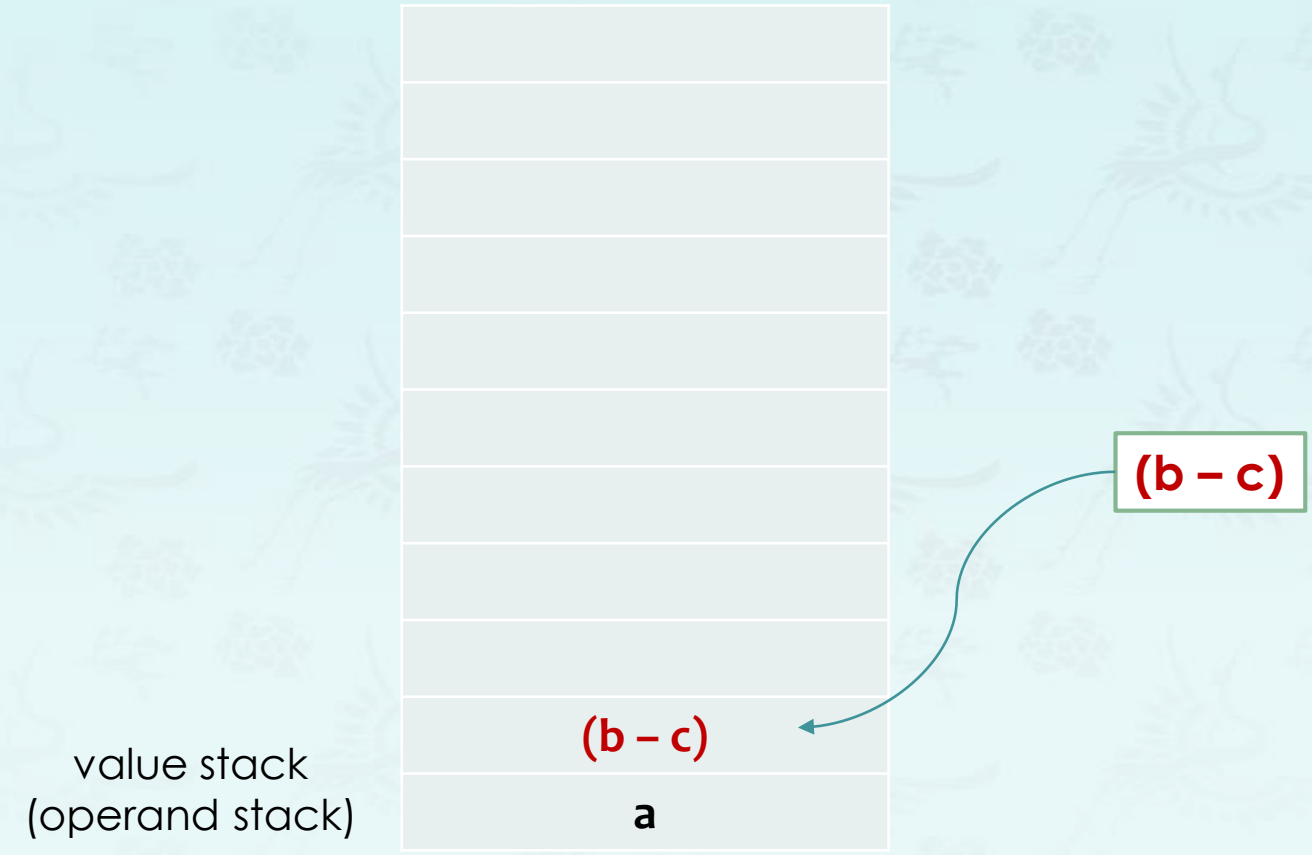
a b c – d + / e a – * c * ⇒ (a / ((b – c) + d)) * (e – a) * c



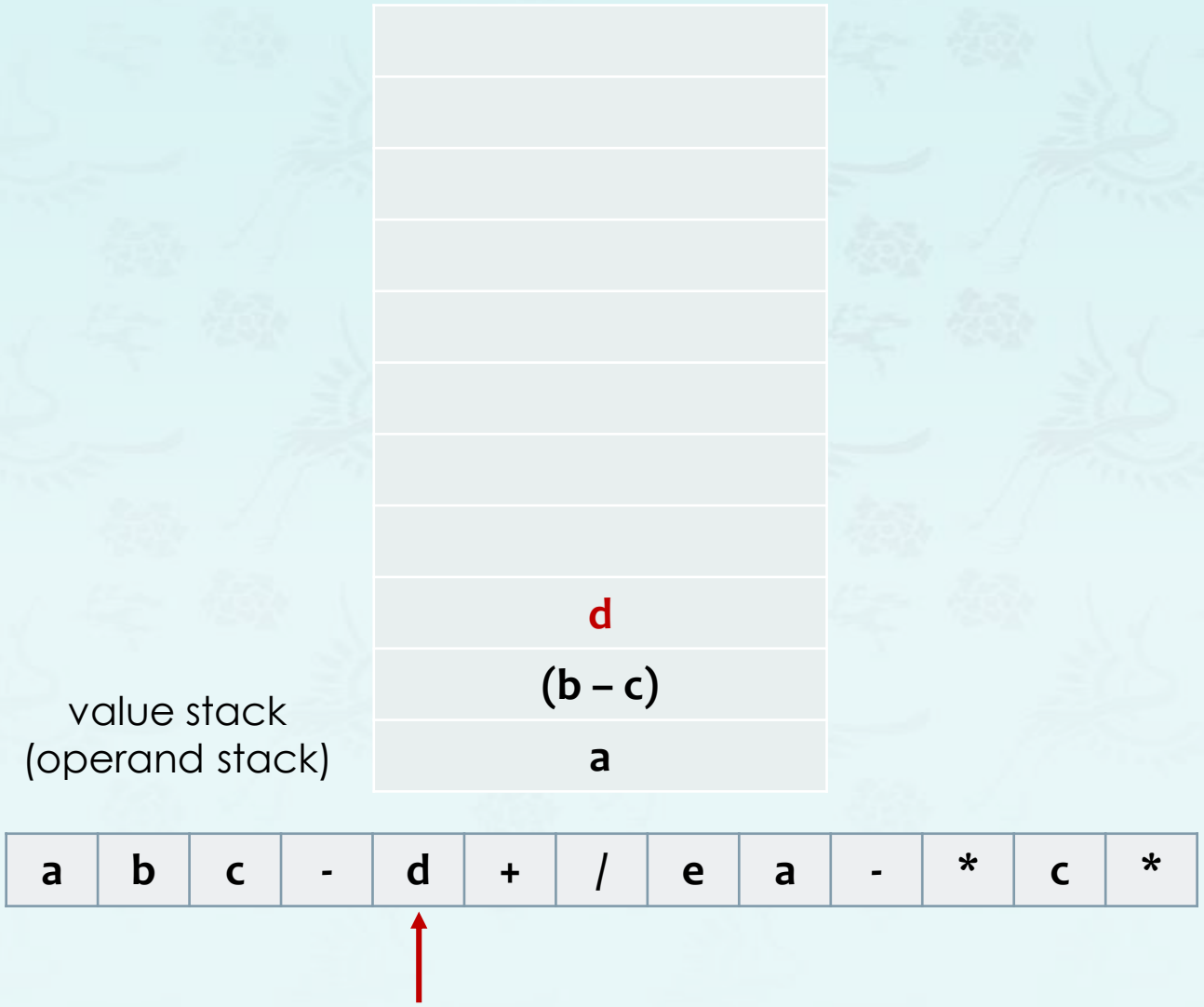
a	b	c	-	d	+	/	e	a	-	*	c	*

- pop two operands and evaluate them
- push the result

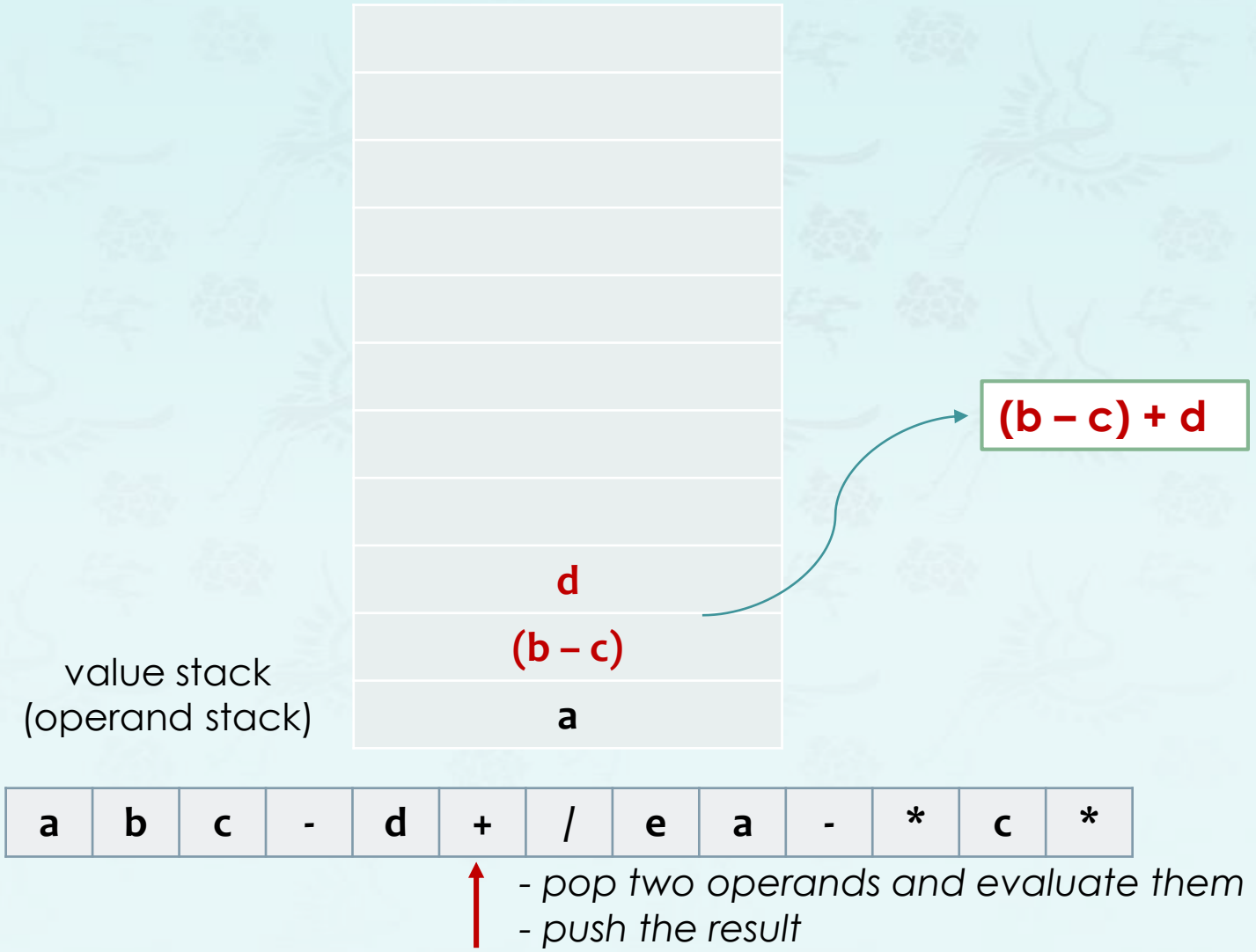
Arithmetic expression evaluation



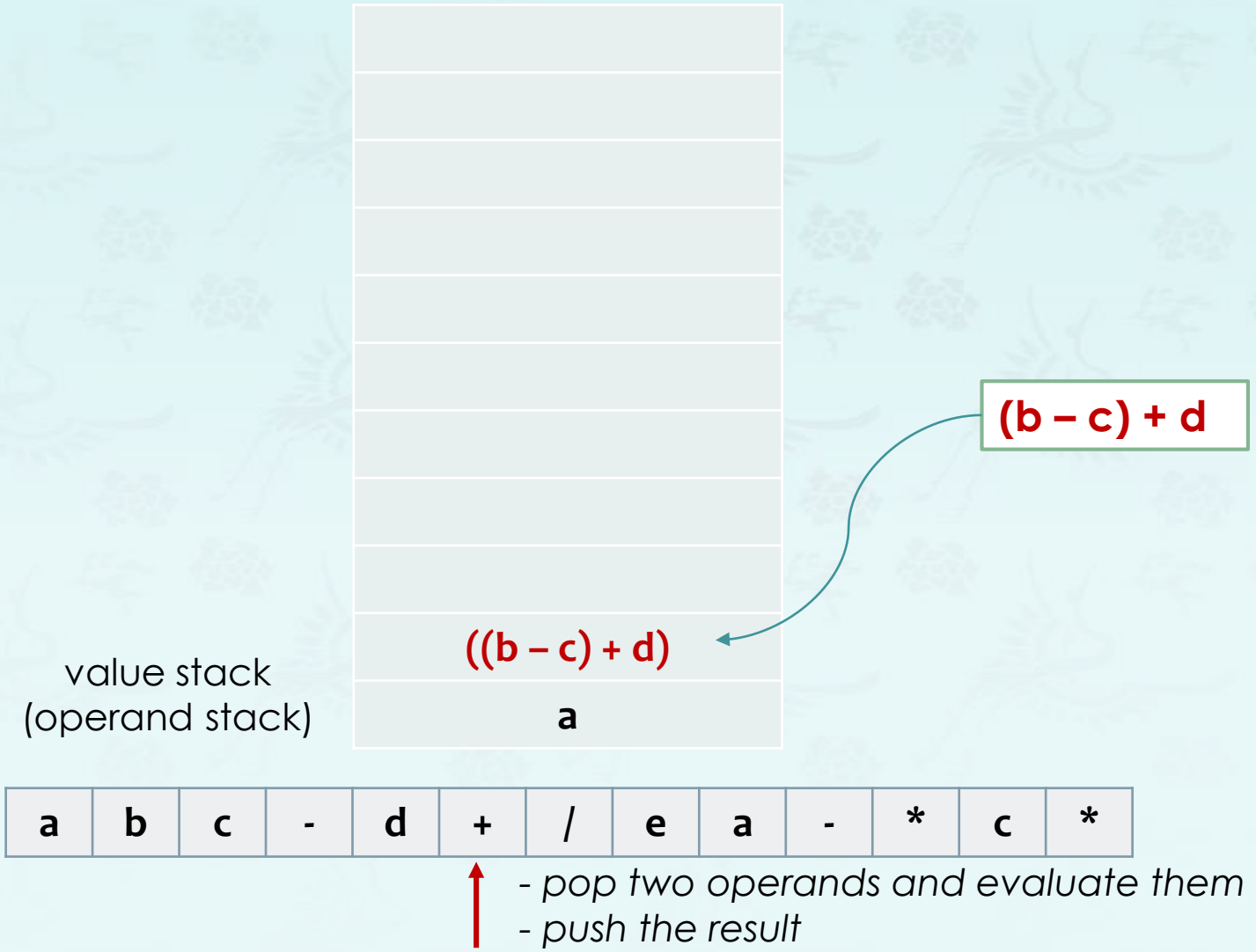
Arithmetic expression evaluation



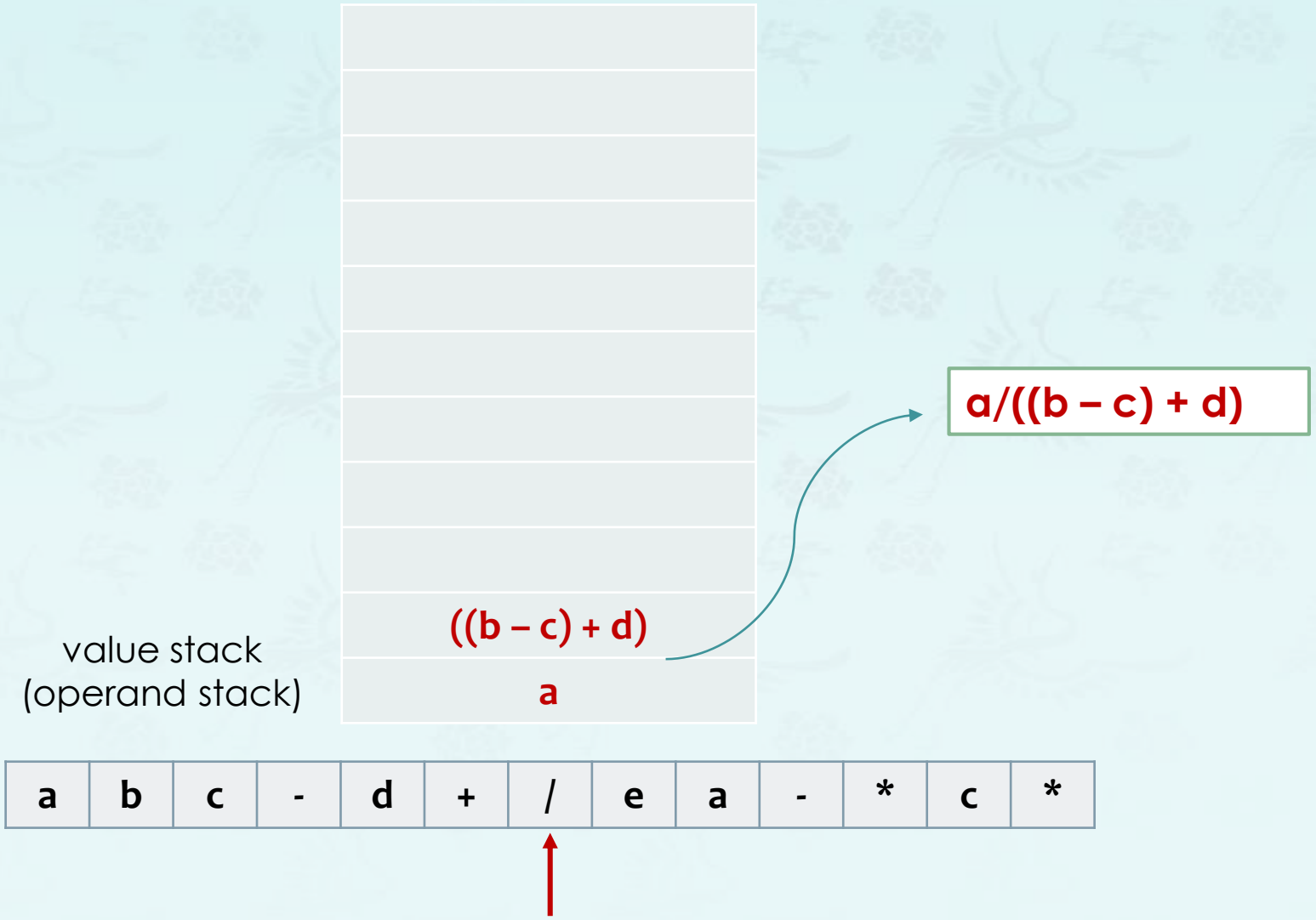
Arithmetic expression evaluation



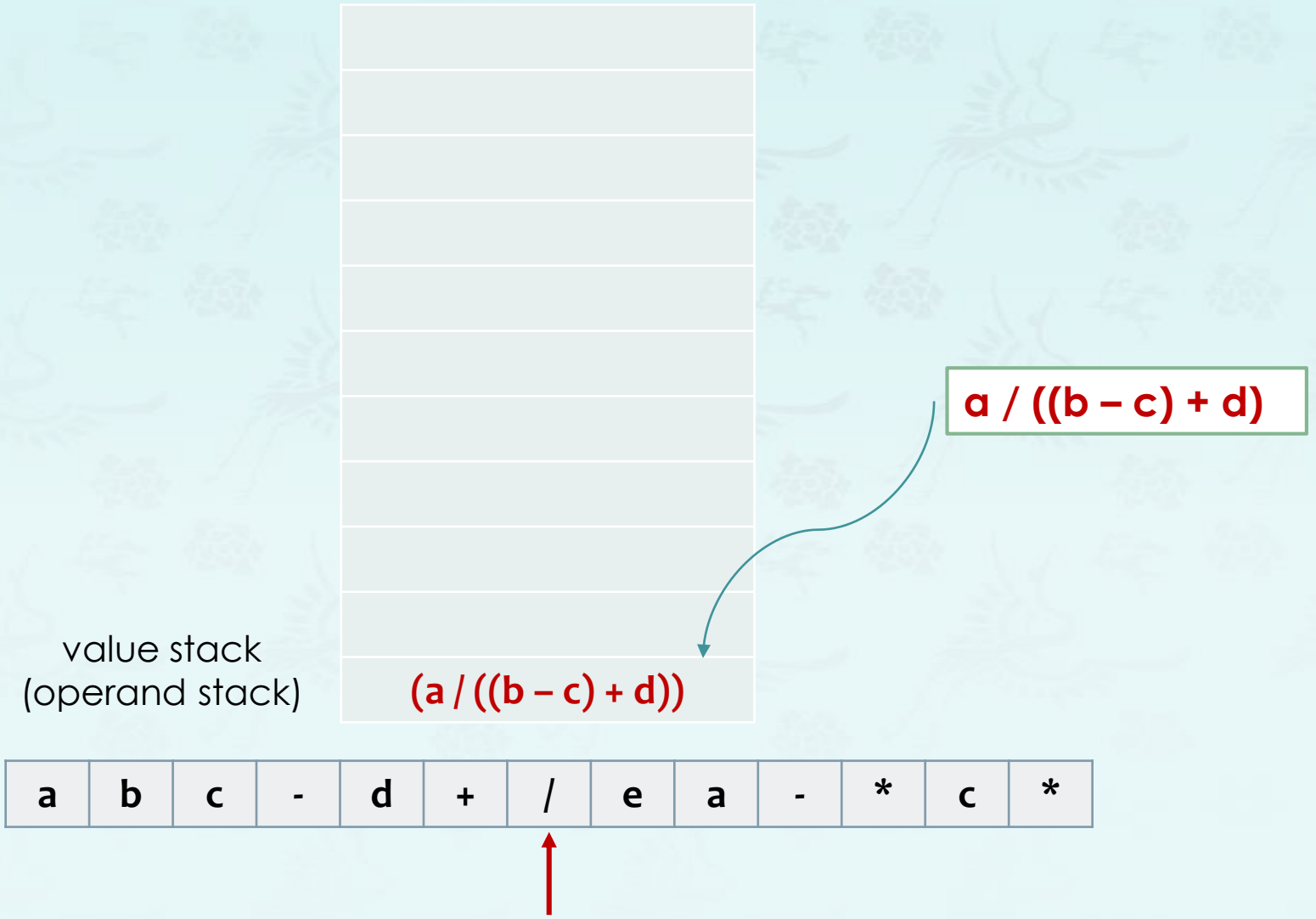
Arithmetic expression evaluation



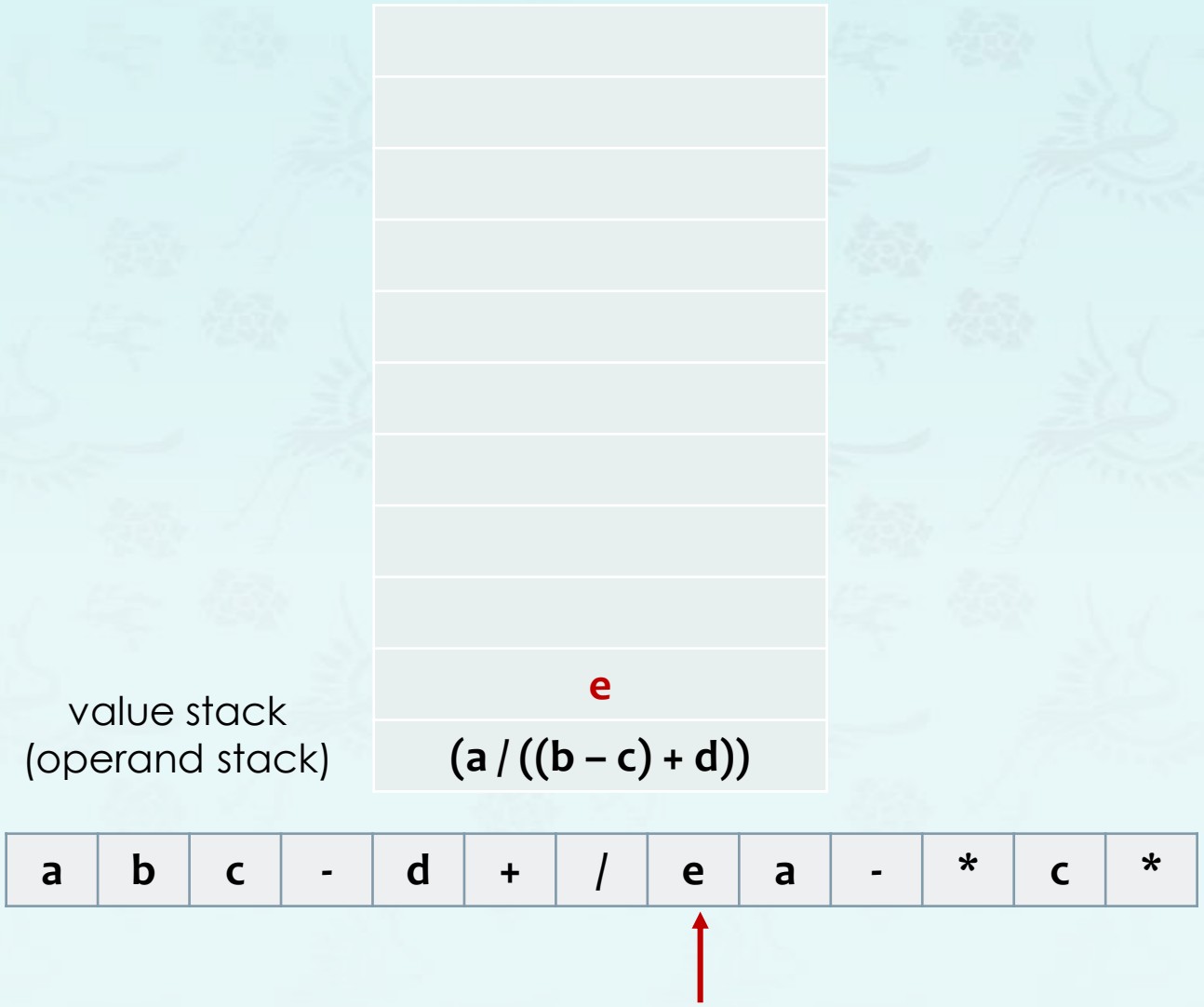
Arithmetic expression evaluation



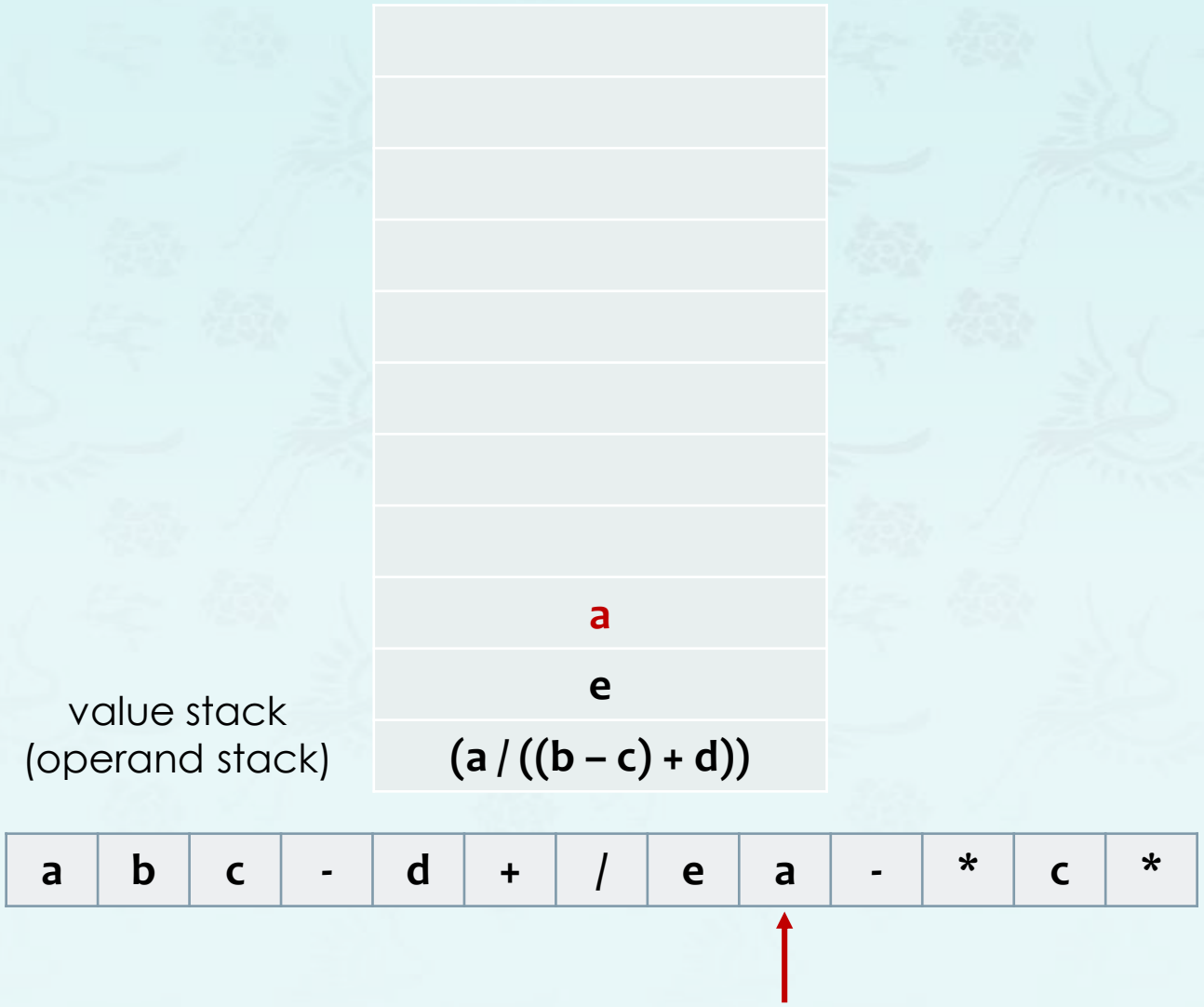
Arithmetic expression evaluation



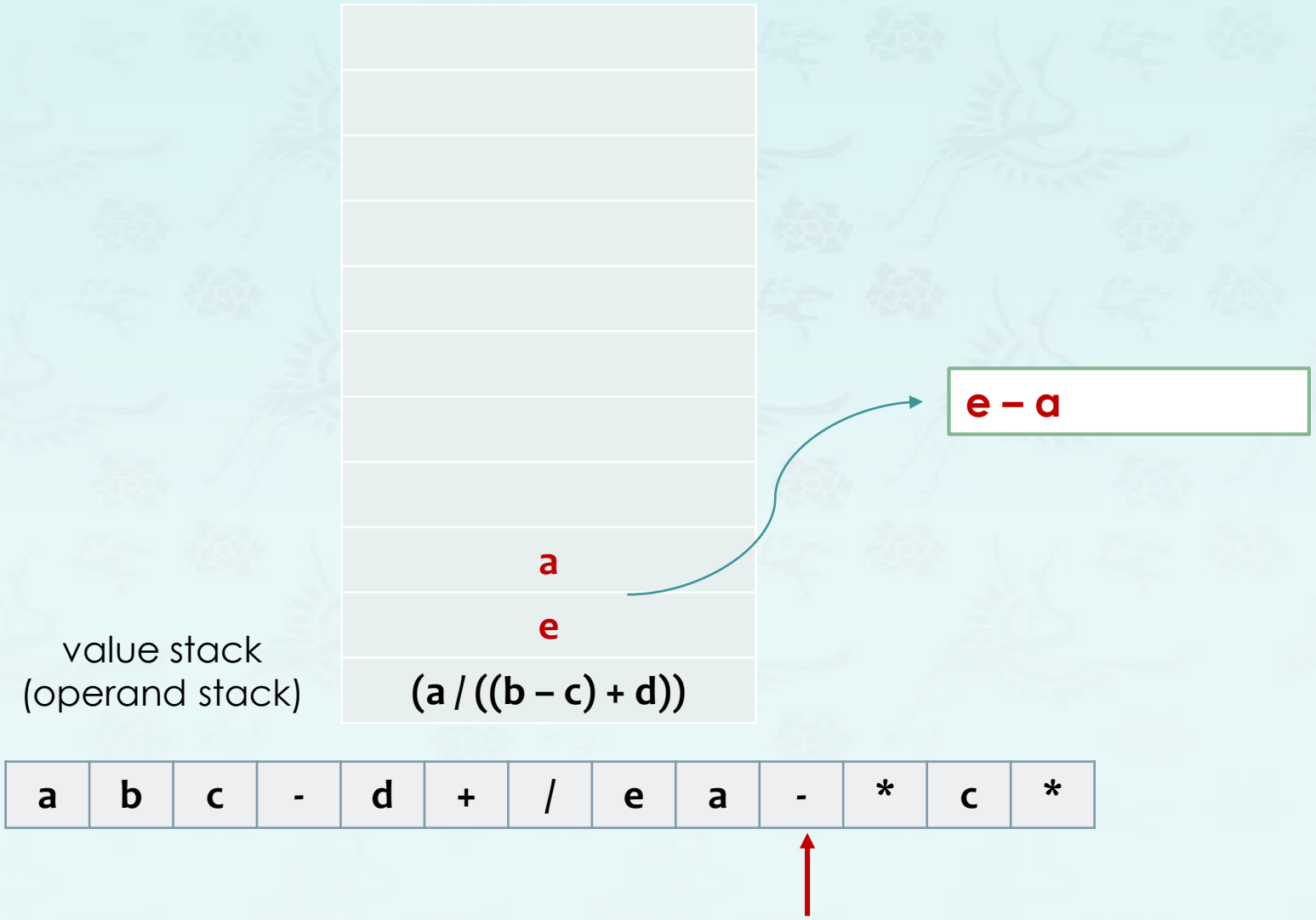
Arithmetic expression evaluation



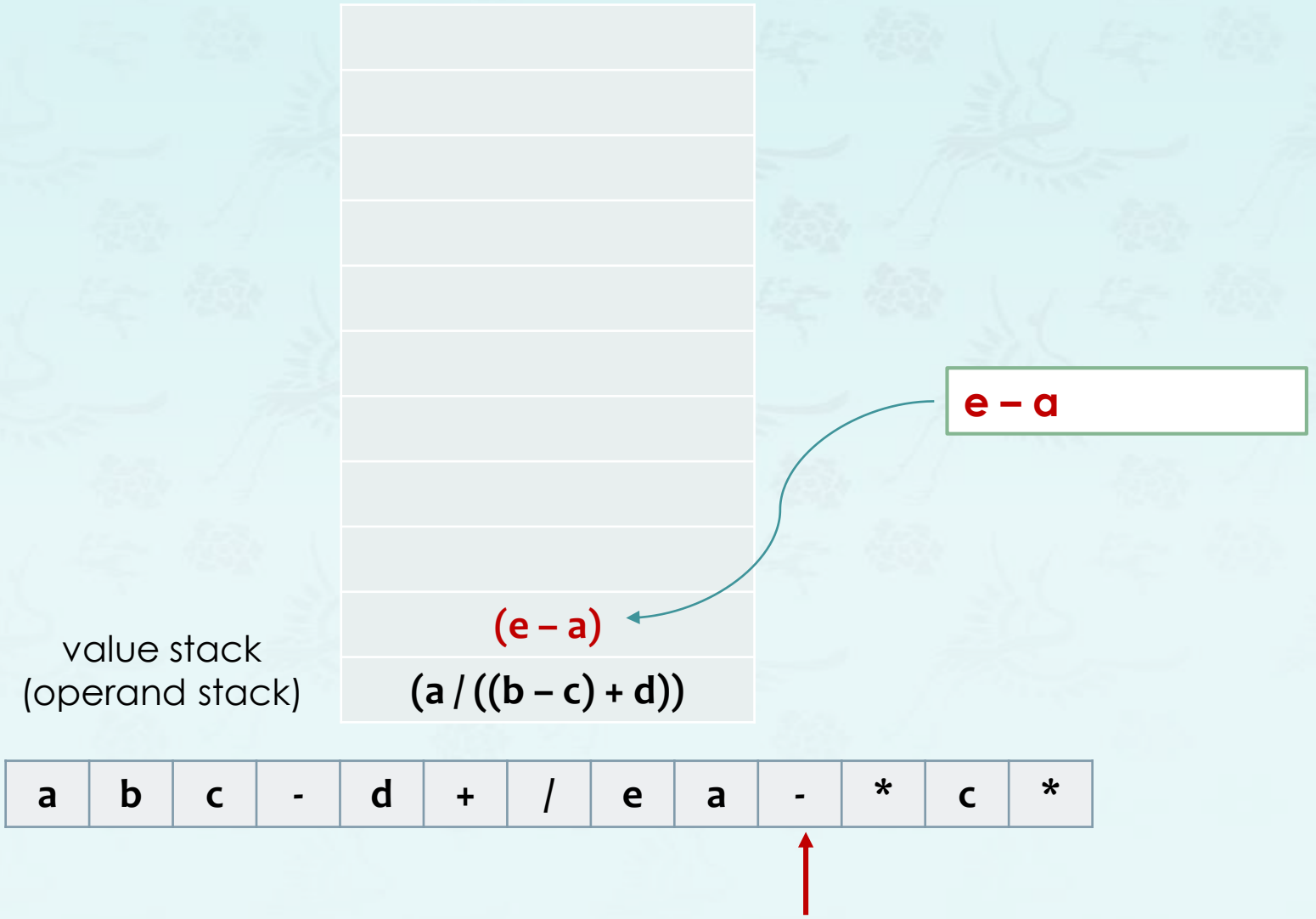
Arithmetic expression evaluation



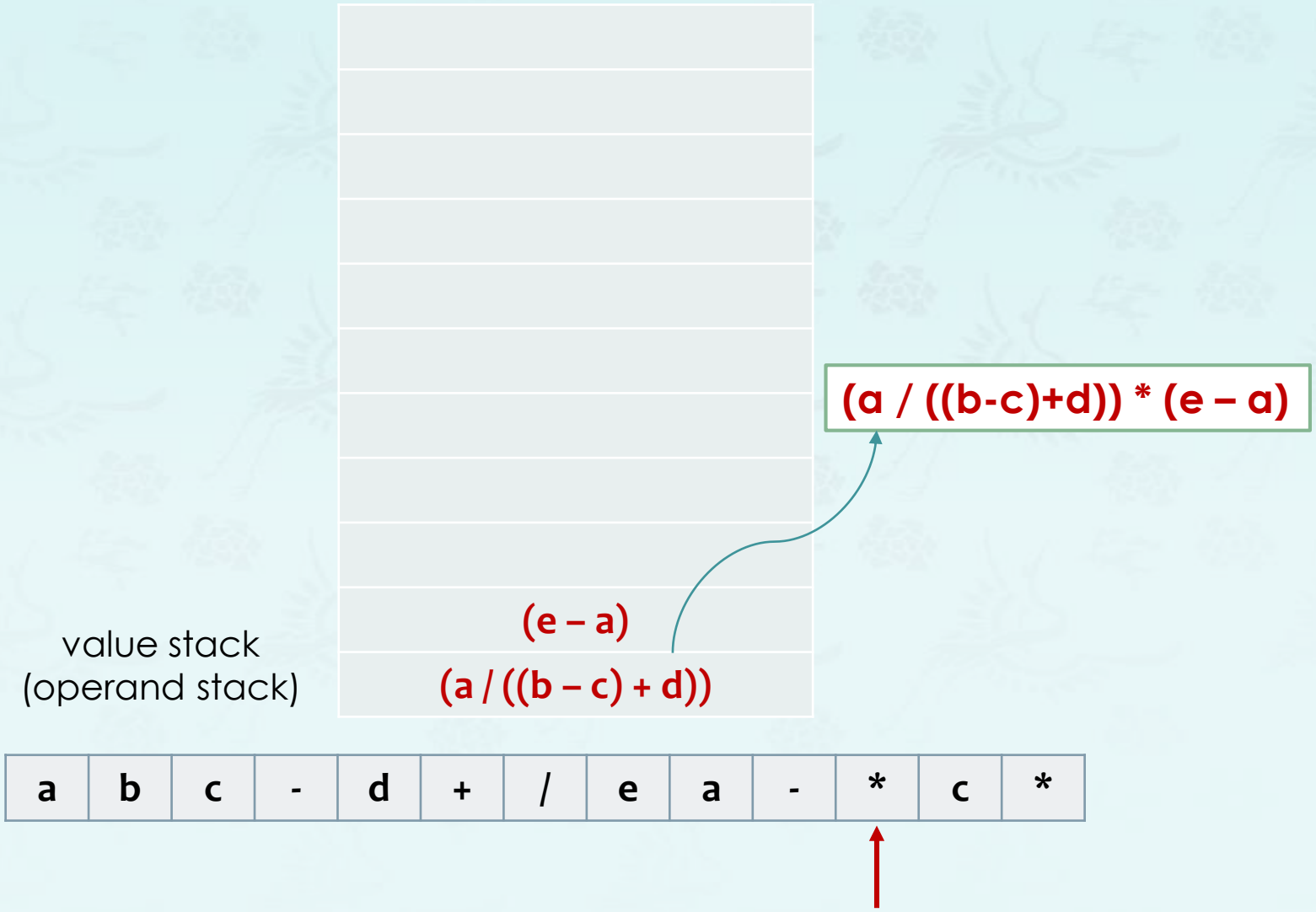
Arithmetic expression evaluation



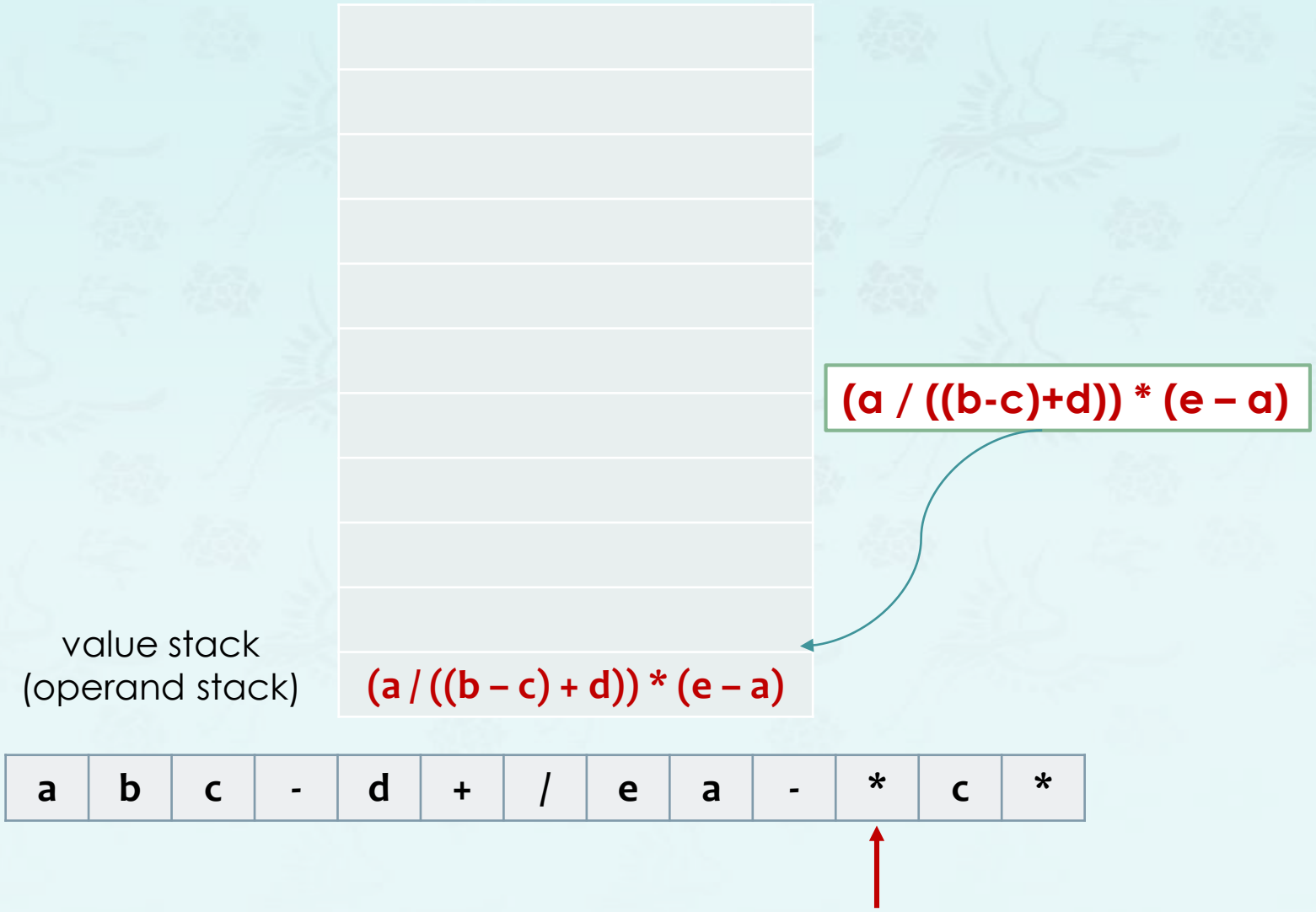
Arithmetic expression evaluation



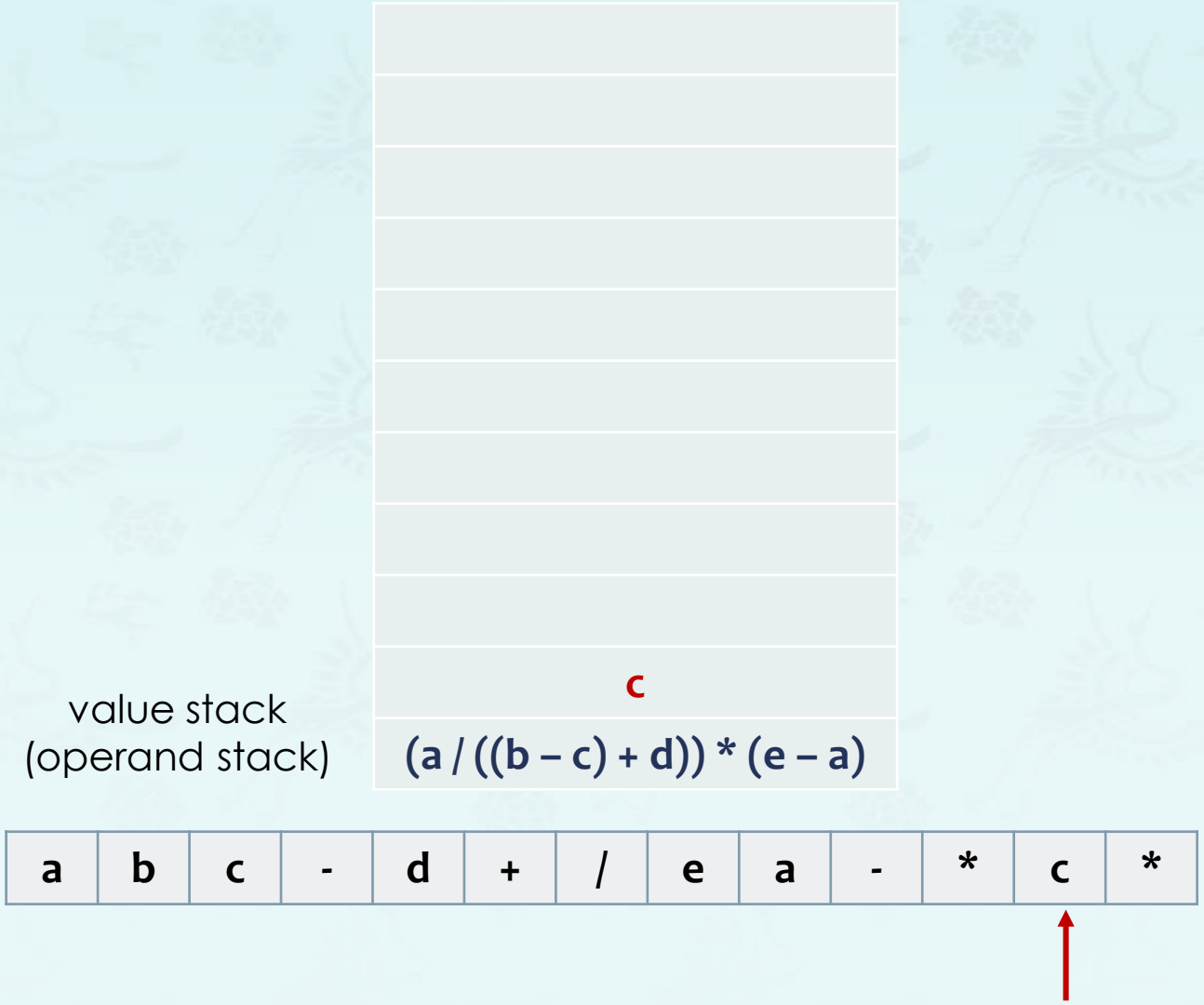
Arithmetic expression evaluation



Arithmetic expression evaluation



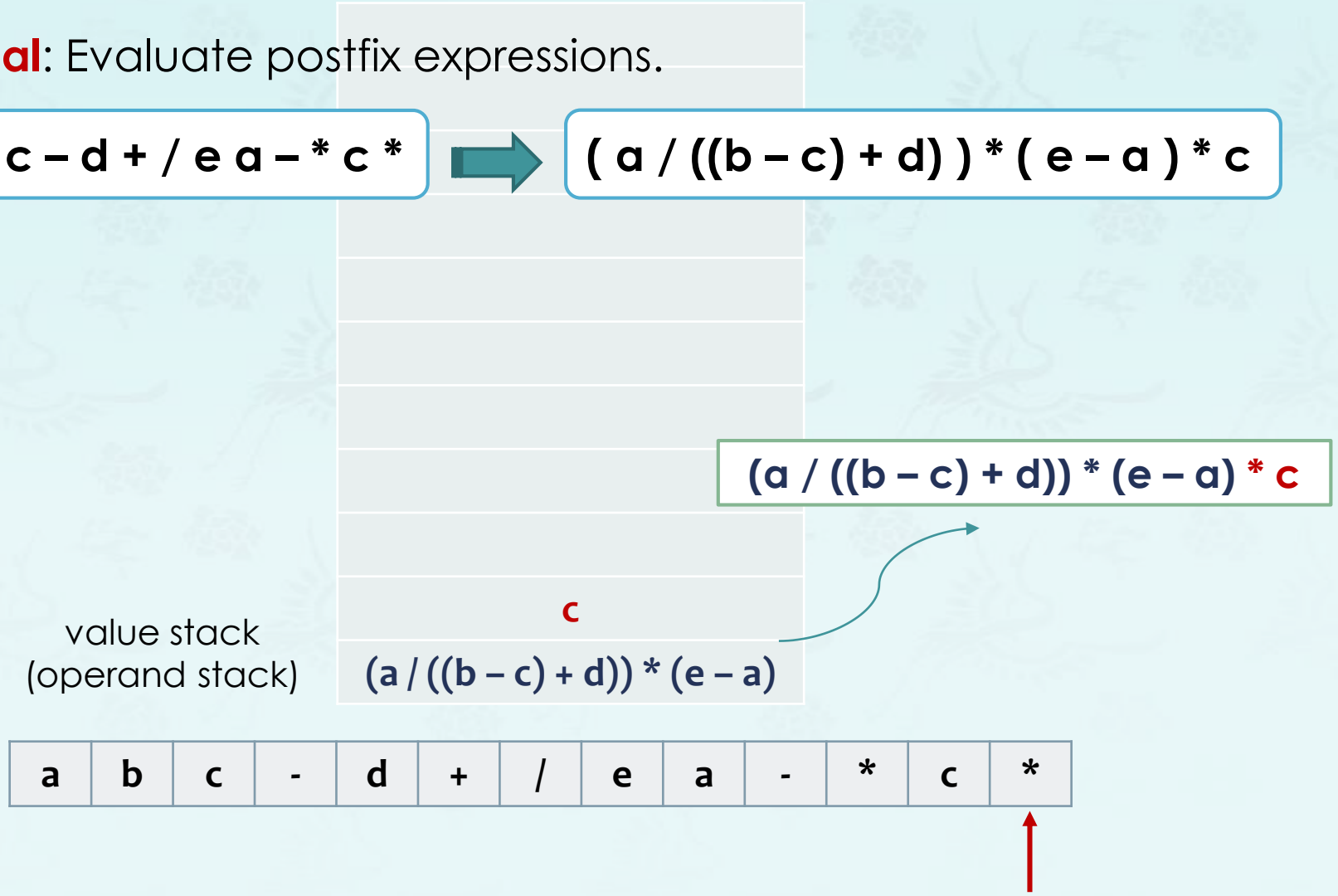
Arithmetic expression evaluation



Arithmetic expression evaluation

Goal: Evaluate postfix expressions.

a b c – d + / e a – * c * ⇒ (a / ((b – c) + d)) * (e – a) * c



Arithmetic expression evaluation - Infix to Postfix Conversion

We use a stack.

1. When an **operand** is read, output it.
2. When an **operator** is read,
   - **Pop** until the top of the stack has an element of **lower** precedence.
   - Then **push** it.
3. When) is found, pop until we find the matching (.
4. (has the lowest precedence when in the stack
   but has the highest precedence when in the input.
5. When we reach the end of input, pop until the stack is empty.

Arithmetic expression evaluation - Infix to Postfix Conversion

Example 1:

infix: 3+4*5/6

in	stack(bottom to top)	postfix
3		
+		
4		
*		
5		
/		
6		

Arithmetic expression evaluation - Infix to Postfix Conversion

Example 2:

infix: (1+3)*(4-2)/(5+7)

in	stack (bottom to top)	postfix
(	(	
1		1
+	(+	
3		1 3
)		1 3 +
*	*	
(	* (	
4		1 3 + 4
-	* (-	
2		1 3 + 4 2
)	*	1 3 + 4 2 -
/	/	1 3 + 4 2 - *

in	stack	postfix
(	/ (	1 3 + 4 2 - *
5		1 3 + 4 2 - * 5
+	/ (+	
7		1 3 + 4 2 - * 5 7
)		1 3 + 4 2 - * 5 7 +
		1 3 + 4 2 - * 5 7 + /

- Operands are output immediately
- Stack operators until right parens
- Unstack until left parens
  Delete left parens
- In general, higher precedence operator must be output before lower one.)

Arithmetic expression evaluation - Infix to Postfix Conversion

Example 3:

infix: a – (b + c * d) / e

in	stack(bottom to top)	postfix
a		
-		
(		
b		
+		
c		
*		
d		
)		
/		
e		

- Operands are output immediately
- Stack operators until right parens
- Unstack until left parens
  Delete left parens
- In general, higher precedence operator must be output before lower one.)

Arithmetic expression evaluation - Infix to Postfix Conversion

Example 3:

infix: A * (B + C * D) + E

	in	stack(bottom to top)	postfix
1	A		
2	*		
3	(		
4	B		
5	+		
6	C		
7	*		
8	D		
9	)		
10	+		
11	E		
12			

2/2

Stack and Queue

Data Structures
C++ for C Coders

한동대학교 김영섭 교수
idebtor@gmail.com

*applications - **infix to postfix***