

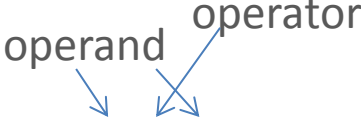
Stacks – Calculator Application

Alexandra Stefan

Last modified 2/6/2018

Infix and Postfix Notation

- The standard notation we use for writing mathematical expressions is called **infix notation**.
 - The operators are between the operands.
- There are two alternative notations:
 - **prefix notation**: the operator comes before the operands.
 - **postfix notation**: the operator comes after the operands.

- Example:
 - **infix:** $5 * ((9 + 8) * (4 * 6)) - 7$
 - **prefix:** $(* 5 (- (* (+ 9 8) (* 4 6)) 7))$
 - **postfix:** $5 9 8 + 4 6 * * 7 - *$
 - No parentheses needed.
 - Can be easily evaluated using a stack.

Processing a Symbolic Expression

- How do we process an expression such as:
 - $5 * ((9 + 8) * (4 * 6)) - 7$
 - postfix: $5\ 9\ 8\ +\ 4\ 6\ *\ *\ 7\ -\ *$
- Think of the input as a **list of tokens**.
 - Assume it is already tokenized
- A **token** is a **logical unit** of input, such as:
 - A number
 - An operator
 - A parenthesis.

Tokens

- A **token** is a **logical unit** of input, such as:
 - A number
 - An operator
 - A parenthesis.
- What are the tokens in:
 - $51 * (((195 + 8) * (4 - 6)) + 7)$
- Answer: 51, *, (, (, (, 195, +, 8,), *, (, 4, -, 6,),), +, 7,)
 - 19 tokens.
 - Note that a token is NOT a character. For example 195 is one token, but it contains 3 characters.
 - We will not discuss how to build tokens from characters.
 - The numbers are the difficult part.

Converting Infix to Postfix

Input: a list/stream of tokens in infix order.

Output: a list of tokens in postfix order.

Assumptions:

1. Each **operator has two operands**.
2. The input is **fully parenthesized**.

Every operation (that contains an operator and its two operands) is enclosed in parentheses.

Fully parenthesized	Not fully parenthesized (not allowed as input)
$(3+5)$	$3+5$
$(2+(5-4))$	$(2+5-4)$ $2+(5-4)$ $2+((5-4))$
$((2 + 9) - (4 + 5))$	$(2 + 9) - (4 + 5)$

T	op_stack	result list
5		5
*	*	
9		5 9
+	* +	
8		5 9 8
)	*	5 9 8 +
/	* /	
4		5 9 8 + 4
*	* / *	
6		5 9 8 + 4 6
)	* /	5 9 8 + 4 6 *
)	*	5 9 8 + 4 6 * /
-	* -	
7		5 9 8 + 4 6 * / 7
)	*	5 9 8 + 4 6 * / 7 -
)		5 9 8 + 4 6 * / 7 - *

: a stream of tokens in infix order.

 output: a list, result, of tokens in postfix order.

 (Uses a stack: op_stack)

 result = empty list

 op_stack = empty stack

 while(the input stream is not empty)

 T = next token

 If T is left parenthesis, ignore.

 If T is a number, insertAtEnd(result, T)

 If T is an operator, push(op_stack, T).

 If T is right parenthesis:

 op = pop(op_stack)

 insertAtEnd(result, op)

Infix → Postfix

 (5 * (((9 + 8) / (4 * 6)) - 7))

 - Numbers go in the result list

 - Operators go on the op_stack

 (the stack shown grows to the right)

 - Left parenthesis, (, are ignored.

 - At right parenthesis,), pop operator from op_stack and add it to the result list.

Evaluating Expressions in Postfix Notation

Input: a list tokens in infix order.

Output: the result of the calculation (a number).

Assumption: the list of tokens is be provided as input.

while(token list is not empty)

T = remove next token (number or operator) from list.

If T is a number, push(stack, T).

If T is an operator:

A = pop(stack)

B = pop(stack)

C = apply operator T on A and B

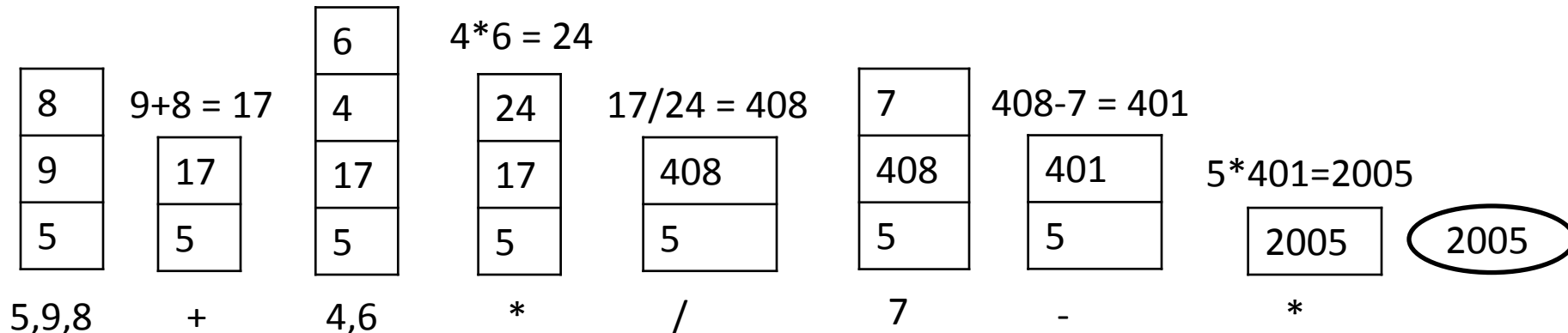
(order: B T A, e.g.: B-A)

push(stack, C)

final_result = pop(stack)

Postfix: 5 9 8 + 4 6 * * 7 - *

Token list: 5, 9, 8, +, 4, 6, *, *, 7, -, *



- A student pointed out that in the previous example, slide 7 was not evaluating the same postfix expression we got in slide 6. This is what was shown in class.
- Below I have modified the original infix expression slightly and redid the steps so that the same expression is evaluated. (2:40pm)

T	op_stack	result list
5		5
*	*	
2		5 2
+	* +	
8		5 2 8
)	*	5 2 8 +
/	* /	
6		5 2 8 + 6
-	* / -	
4		5 2 8 + 6 4
)	* /	5 2 8 + 6 4 -
)	*	5 2 8 + 6 4 - /
-	* -	
7		5 2 8 + 6 4 - / 7
)	*	5 2 8 + 6 4 - / 7 -
)		5 2 8 + 6 4 - / 7 - *

: a stream of tokens in infix order.
output: a list, result, of tokens in postfix order.
(Uses a stack: op_stack)
result = empty list
op_stack = empty stack
while(the input stream is not empty)
 T = next token
 If T is left parenthesis, ignore.
 If T is a number, insertAtEnd(result, T)
 If T is an operator, push(op_stack, T).
 If T is right parenthesis:
 op = pop(op_stack)
 insertAtEnd(result, op)

Infix → Postfix

(5 * (((2 + 8) / (6 - 4)) - 7))

- Numbers go in the result list
- Operators go on the op_stack
 (the stack shown grows to the right)
- Left parenthesis, (, are ignored.
- At right parenthesis,), pop operator from op_stack and add it to the result list.

Evaluating Expressions in Postfix Notation

Input: a list tokens in infix order.

Output: the result of the calculation (a number).

Assumption: the list of tokens is be provided as input.

while(token list is not empty)

T = remove next token (number or operator) from list.

If T is a number, push(stack, T).

If T is an operator:

A = pop(stack)

B = pop(stack)

C = apply operator T on A and B

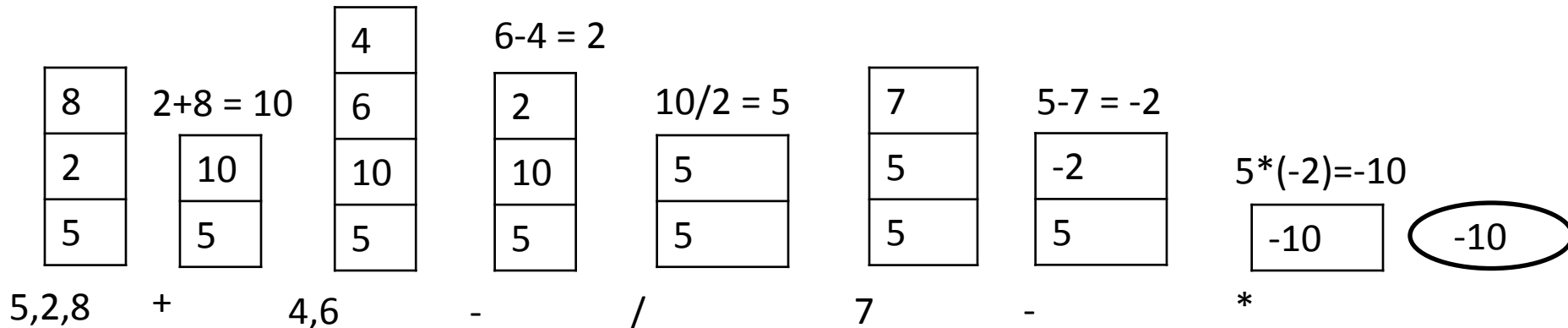
(order: B T A, e.g.: B-A)

push(stack, C)

final_result = pop(stack)

Postfix: 5 2 8 + 6 4 - / 7 - *

Token list: 5, 2, 8, +, 6, 4, -, /, 7, -, *



Here the * indicates the multiplication operator, not a pop() operation on the stack.

We do not explicitly show the pop operations. Instead, for each operator we pop, pop, calculate, push.