Infix Notation

- Each binary operator is placed between its operands.
- Each unary operator precedes its operand.

-2 + 3 * 5 ⟺ (-2) + (3 * 5)

Postfix expressions are easy to evaluate:

- no subexpressions
- precedence among operators already accounted for

But this is not the case for infix expressions!

e.g. 9 + (2 – 3) * 8

Infix Expression Evaluation

Two approaches to evaluate an infix expression:

- Use two stacks within one scan.

- Convert to equivalent postfix expression and then call the postfix evaluator.

Operator Associativity

Left associative: +, –, *, /, %

2 + 3 + 4 – 8 ⟺ ((2 + 3) + 4) – 8

8 / 4 * 3 ⟺ (8 / 4) * 3

Right associative: ^

2^7^6 + (3 – 2 * 4) % 5

2^(7^6) + ((3 – (2 * 4)) % 5)

Operator Precedence

() > ^ > * = % = / > + = –

12 + 23 * 4 ^ (3 – 7 / 11 ^ 2) % 25

12 + ((23 * (4 ^ (3 – (7 / (11 ^ 2))))) % 25)

Rank of Expression

Evaluates an infix expression based on *rank*.

1	for any operand
-1	for +, –, *, /, %, ^
0	for (,)

Cumulative rank: sum of the ranks of individual terms.

2 ^ 7 ^ 6 + (3 – 2 * 4) % 5

cumulative rank: 1 0 1 0 1 0 0 1 0 1 0 1 1 0 1

Necessary Condition for Correctness

The cumulative rank after each symbol is always 0 or 1, and for the entire expression must be 1.

(exactly one more operand than operator)

Invalid expression if condition is not satisfied.

2 4 + 3

However, the condition is not sufficient, i.e., satisfying the condition does not imply the correctness.

(4 + 3

8)) % 2

How to further check the correctness?

Convert it to postfix

and evaluate!

4 (+) 3 $\Rightarrow$ 4 + 3 error!

Infix-to-Postfix Conversion

During the scan of an expression:

- Write an operand immediately to the output string.
- No need to maintain an operand stack.

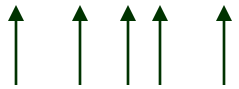
Operator stack

- Stores operators and left parentheses as soon as they appear.
- Manages the order of precedence and associativity of operators.
- Handles subexpressions.

Example 1

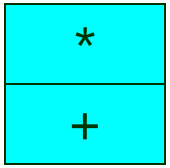
The stack temporally stores operators awaiting their right operand.

a + b * c



* has higher priority than +
$\Rightarrow$ add to the stack

Operator stack:



Postfix string:

a	b	c	*	+

Example 2

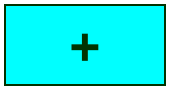
Use the stack to handle operators with same or lower precedence.

a * b / c + d

* has the same priority as /
⇒ pop * and write it to the postfix string before adding / to the stack.

/ has higher priority than +

Operator stack:



Postfix string:

a	b	*	c	/	d	+

Example 3

Use precedence values to handle ^ (right associative).

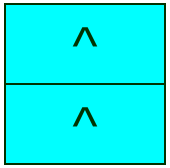
input precedence 4 when ^ is the input.
stack precedence 3 when ^ resides on the stack.

a ^ b ^ c

2^{nd} ^ has precedence 4 but 1^{st} ^ has only 3

$\Rightarrow$ 2^{nd} ^ goes to operator stack (so it will be popped before 1^{st} ^)

Operator stack:



Postfix string:

a	b	c	^	^

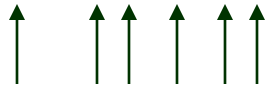
Example 4

Two precedence values for left parenthese (:

input precedence 5 which is higher than that of any operator.

(all operators on the stack must remain because a new subexpression begins.)

a * (b + c)



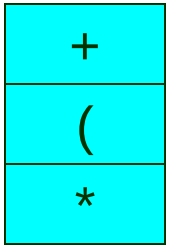
(has precedence 5 ⇒
it goes to the operator stack.

(now has precedence -1 ⇒
it stays on the operator stack.

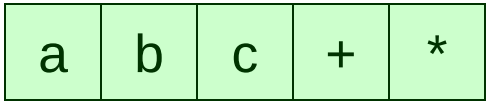
pop all operators from the stack
until (is at the top.

) cancels (at the top.

Operator stack:



Postfix string:



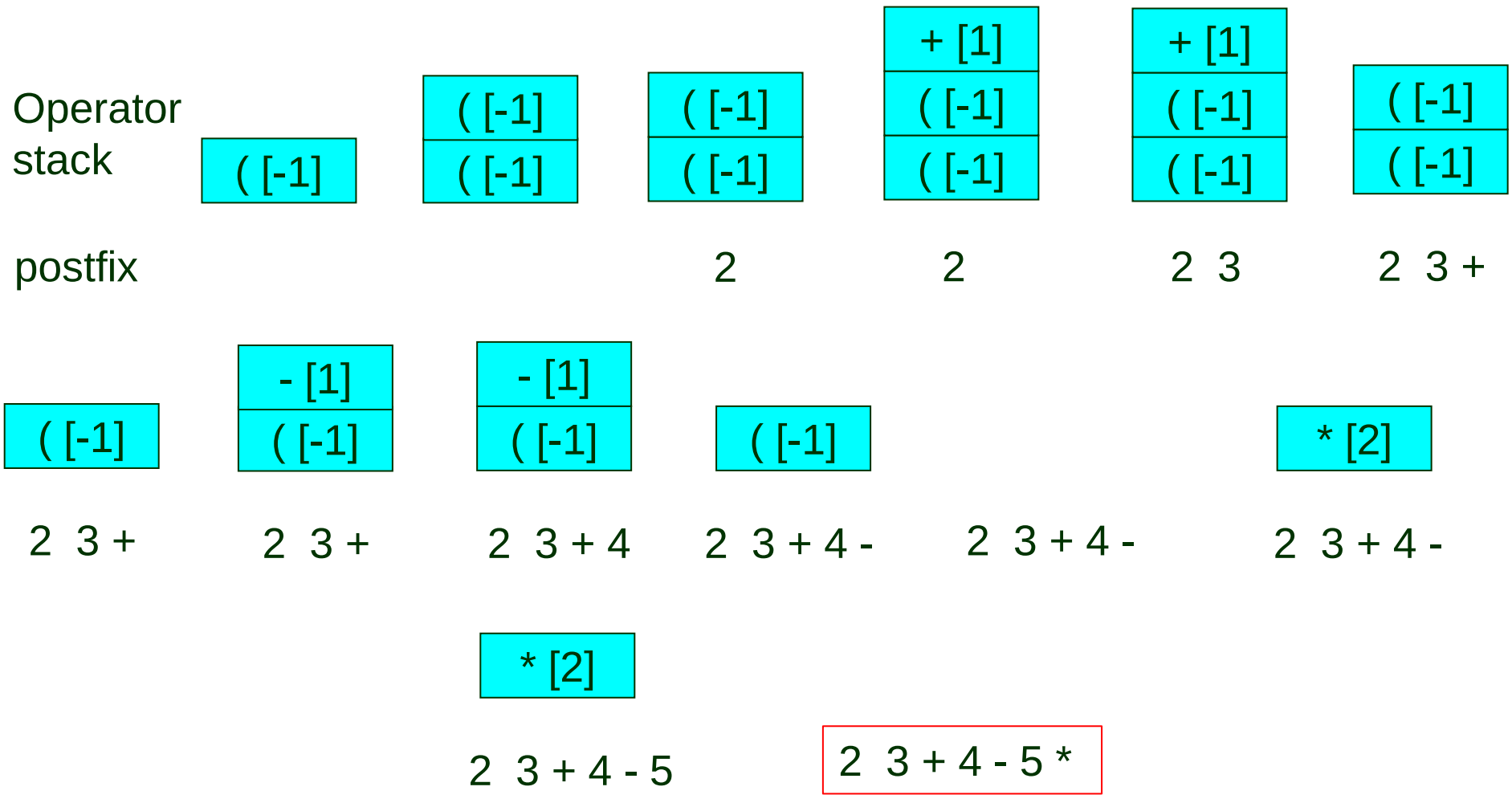
Input and Stack Precedence

Symbol	Input Precedence	Stack Precedence	Rank
+ –	1	1	-1
* / %	2	2	-1
^	4	3	-1
(	5	-1	0
)	0	0	0

(is also right associative with higher input precedence 5 than stack precedence -1. E.g., ((2 + 3) – 4) * 5.

(is Right Associative

((2 + 3) – 4) * 5

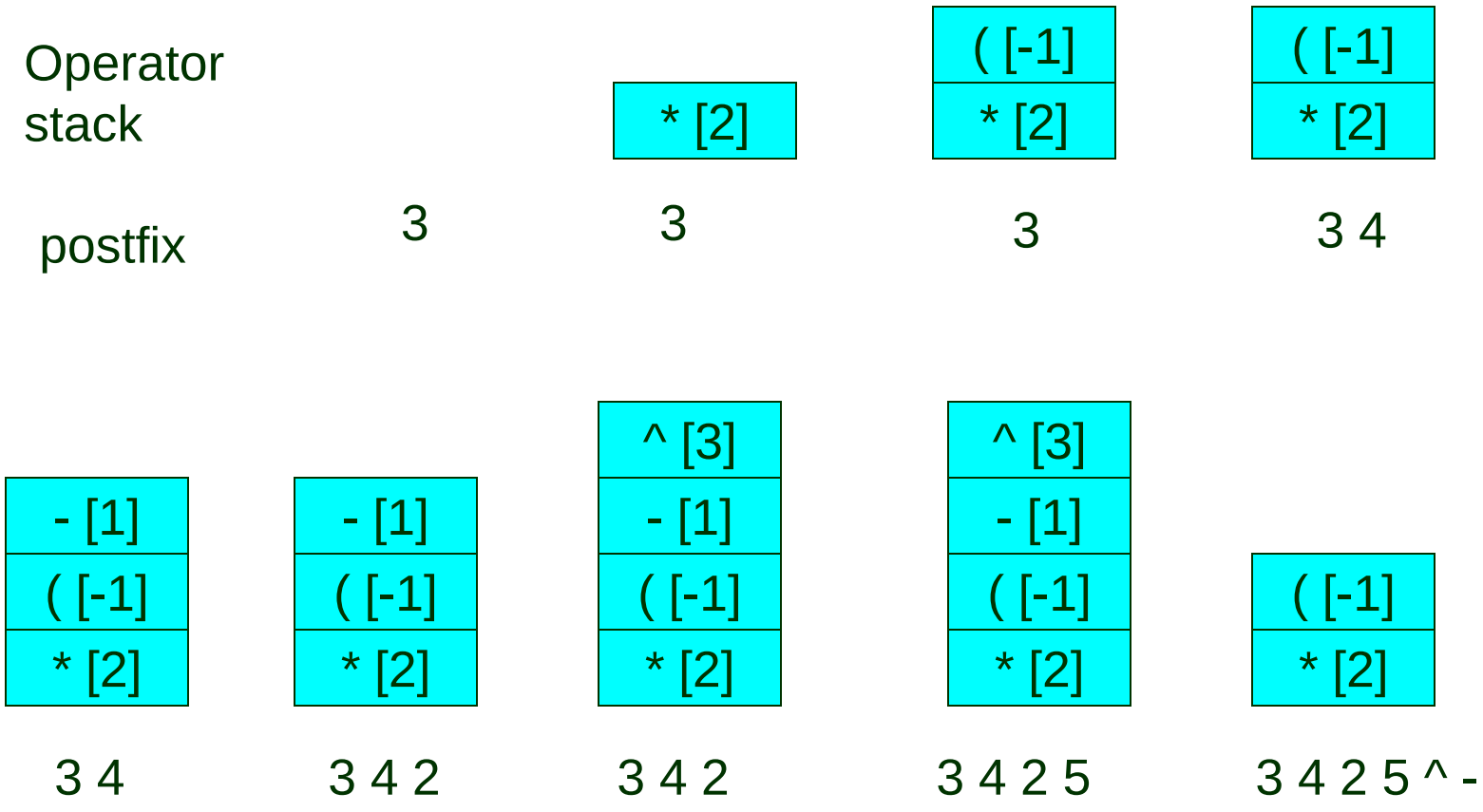


Rules for Evaluation

- Check the cumulative rank after each symbol (must be in the range from 0 to 1).
- Write the input to the postfix string if it is an operand.
- Upon input of an operator or a **(**, compare its input precedence with the stack precedence of the top operator on the stack.
- If the input is **)**, pop all operators from the stack until **(** and write them to the postfix string. Pop **(**.
- At the end of the infix expression, pop all remaining operators from the stack and write them to the postfix string.

A More Complete Example

3 * (4 – 2 ^ 5) + 6



cont'd

Pop (

3 * (4 – 2 ^ 5) + 6

* [2]	+ [1]	+ [1]
3 4 2 5 ^ -	3 4 2 5 ^ - *	3 4 2 5 ^ - * 6

3 4 2 5 ^ - * 6 +

The InfixExpression Class

outputHigherOrEqual()

- Pops the operator stack as long as the operator on the top of the stack has a stack precedence higher than or equal to the input precedence of the current operator *op*.
- Writes the popped operators to the postfix string.
- If *op* is a ')', and the top of the stack is a '(', also pops '(' from the stack but does not write it to the postfix.

Conversion to Postfix

postfix() scans an infix string and does the following:

- Skips a whitespace character.
- Writes an operand to the postfix string.
- Calls outputHigherOrEqual() with an operator.
- Also calls outputHigherOrEqual() when the input is).
- Terminates at the end of the expression or if an error occurs.

Running Time of Conversion

Suppose the infix string has n operators and operands:

- A call to outputHigherOrEqual() may pop $O(n)$ operators off the stack.

  1+ 2^3^4^…^100

- $O(n)$ such calls.

$O(n)$-time infix-to-postfix conversion.

Total time $O(n^2)$?

Not tight. Let's count write, push, and pop operations.

- Every operator or operand that's not '(' or ') ' is written to the postfix string. $O(n)$ writes.
- Every operator that is not ') ' gets pushed onto the stack. $O(n)$ pushes.
- #pops ≤ #pushes. So there are $O(n)$ pops in total.

Reporting Errors

postfix() also keeps track of the cumulative rank and catches five types of error :

- "Operator expected" if the rank goes above 1;
- "Operand expected" if the rank goes below 0;
- "Missing '('" if a scanned ')' in an empty stack without popping any '(' out;
- "Missing ')'" if a '(' is left unmatched on the stack at the end of the scan;
- "Invalid character" if the character is not a digit or operator.

Upcoming Events

Exam 2	Thursday March 28
Project 3 due	Saturday March 30