Programming Language Concepts

Programming Language Theory

Mid-term Exam

- **Schedule: 10/21 Wed 10AM~12PM (2 hours).**
- **Venue:** will be notified right before the exam.
- I'll post mid-term summary session videos by this Saturday (10/17).

Practice

- I made some simple examples of this week's topics.
- You can use 'git pull' on course GitHub repository to download examples in your PC.
- Examples are just to help you understand the topics.
    - You don't need to submit anything to prove that you finished practices.

Topics

- **Control Structure**
  - **Expressions and Their Evaluation**
  - **Statement**
  - Control Flow and Recursion
- Control Abstraction
  - Subprogram

Expression

- An ***Expression*** is a syntactic entity whose evaluation either ***produces a value or undefined*** (which fails to terminate).

- Expressions are one of the basic components of every programming language.

- Although there are languages such as functional languages which do not have statements, expressions exist in every language.

How to Represent?

- Operator and Operands.
  - `x + y, b - 1, f(3) >= 0`
- Prefix, Infix, Postfix notations.
  - Based on the location of operators,
  - `<prefix> ::= <op><prefix><prefix>` | ...
  - `<infix> ::= <infix><op><infix>` | ...
  - `<postfix> ::= <postfix><postfix><op>` | ...

Notations

- Consider mathematical equation: $a + b * c + d$
- Infix Notation: $(a + b) * (c + d)$
- Prefix Notation: $* + a\ b + c\ d$ - Also called *prefix Polish* notation.
    - or $(* \ (+ \ a \ b) \ (+ \ c \ d))$ - *Cambridge Polish* notation, puts operators inside parentheses.
- Postfix Notation
    - $a\ b + c\ d + *$

Semantics

- The semantics of expressions (or how they are evaluated) can be changed according to notations.

- For instance, infix expressions without parentheses may cause ambiguity in its evaluation.

  - $a + b * c + d$

  - $a + (b * c) + d$? or $(a + b) * (b + c)$?

- With *Infix Notation*, we need to consider ***Precedence*** and ***Associativity*** of operators.

Precedence

- Operator Precedence decides which operators should be considered first.

- We need to define such precedence to make evaluation of expression match to our intuition.

  - For `1 + 2 * 3`, we want its value to be 7, not 9.

    - `1 + (2 * 3) = 7 vs. (1 + 2) * 3 = 9`

- So we need precedence rules to prevent such cases.

Associativity

- However, precedence is not enough to correctly evaluate expressions.
- We also need to consider ***operator associativity***, tells us how an operator associates with its operands.
    - 10 - 5 - 3
        - (10 - 5) - 3 = 2 vs. 10 - (5 - 3) = 8
- Most of arithmetic operators associate from ***left to right***,
    - but there is a case like exponentiation,
    - 5^{3^2}= 5^3^2 ➞ 5^(3^2) vs. (5^3)^2

Precedence and Associativity

- Most languages have precedence and associativity which are not counter-intuitive.
- We need to carefully consider when writing code.
- If you have any suspicion, use parentheses to clarify your intention.
    - (1 + 2) * 3, (10 - 5) - 3, (5^3)^2

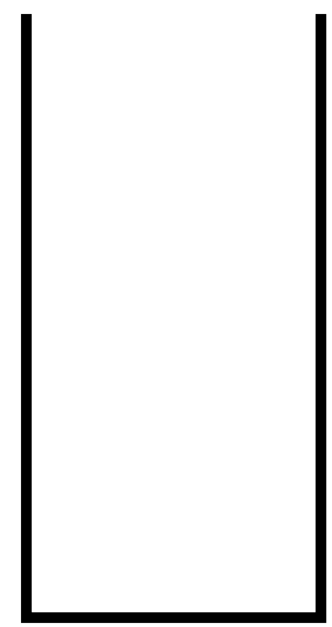
Prefix Notation

- Unlike infix notations, prefix notation has no such ambiguity, if we know the ***arity*** *(# of operands) of an operator.*

- We can consider a simple algorithm to evaluate prefix expressions with a stack and a counter.

  - $* + a\ 2 + b\ c$

  - $a = 1,\ b = 2,\ c = 3$



Input: $* + a\ 2 + b\ c$

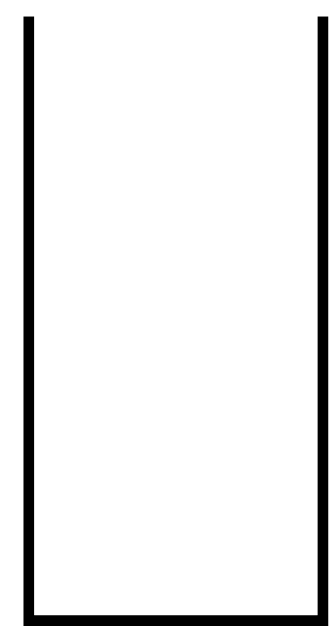
Counter $C = 0$

Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
  - if it is operator, update C with the arity.
  - each operand symbol, decrease C.
  - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
  - update C for new operator.

$a = 1,\ b = 2,\ c = 3$

Input: * + a 2 + b c



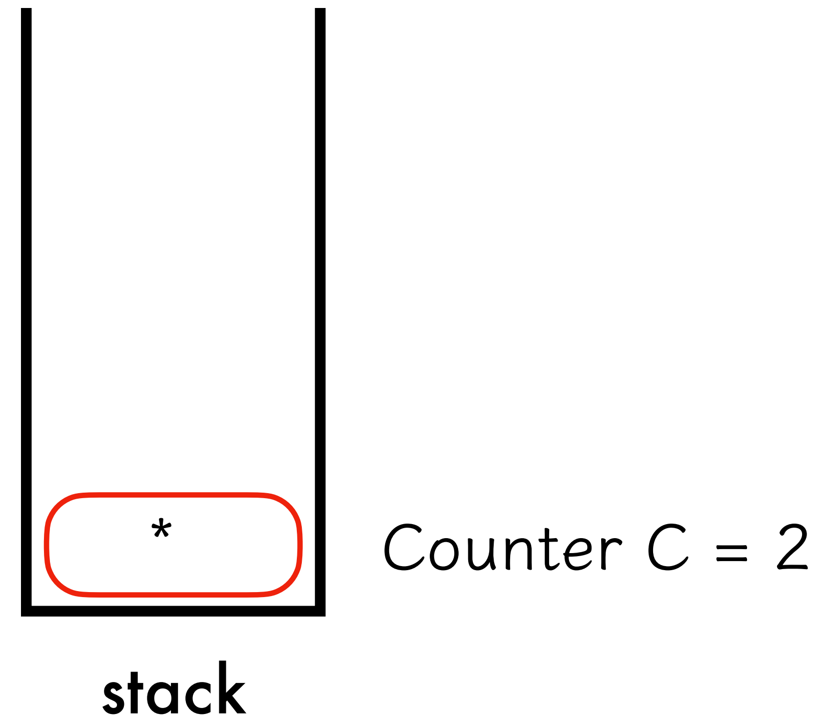
stack

Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
    - if it is operator, update C with the arity.
    - each operand symbol, decrease C.
    - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
    - update C for new operator.

a = 1, b = 2, c = 3

Input: * + a 2 + b c

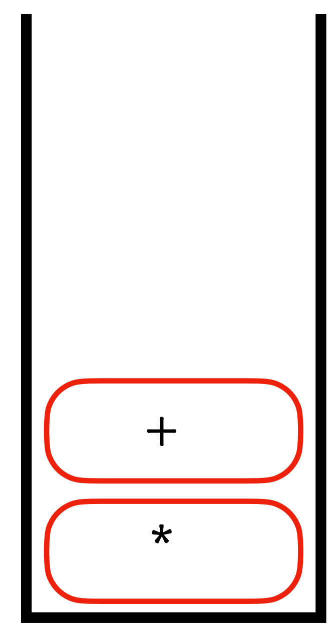


Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
  - if it is operator, update C with the arity.
  - each operand symbol, decrease C.
  - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
  - update C for new operator.

$a = 1,\ b = 2,\ c = 3$

Input: * + a 2 + b c



Counter C = 2

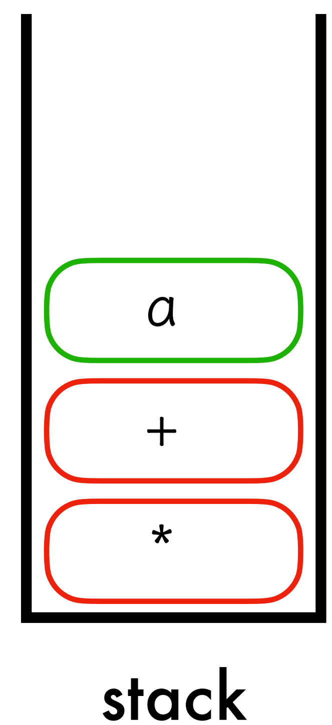
stack

Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
    - if it is operator, update C with the arity.
    - each operand symbol, decrease C.
    - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
    - update C for new operator.

a = 1, b = 2, c = 3

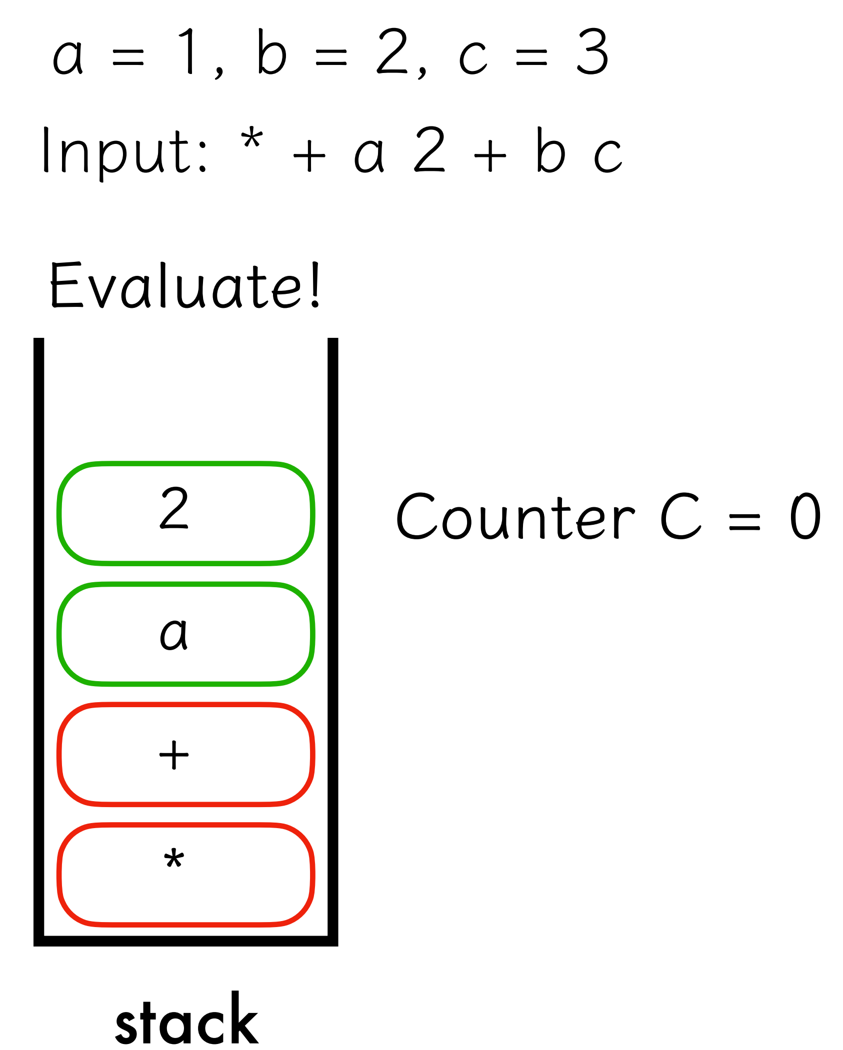
Input: * + a 2 + b c



Counter C = 1

Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
  - if it is operator, update C with the arity.
  - each operand symbol, decrease C.
  - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
  - update C for new operator.

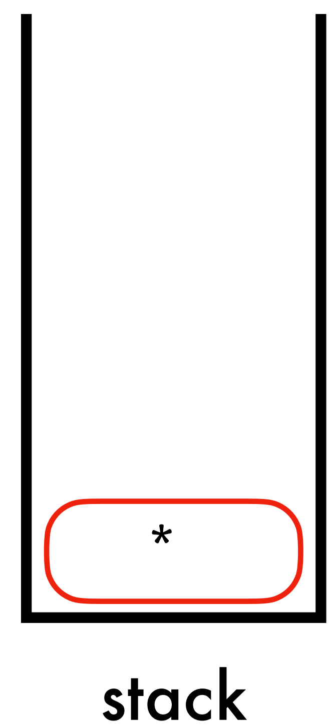


Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
  - if it is operator, update C with the arity.
  - each operand symbol, decrease C.
  - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
  - update C for new operator.

$a = 1,\ b = 2,\ c = 3$

Input: * + a 2 + b c

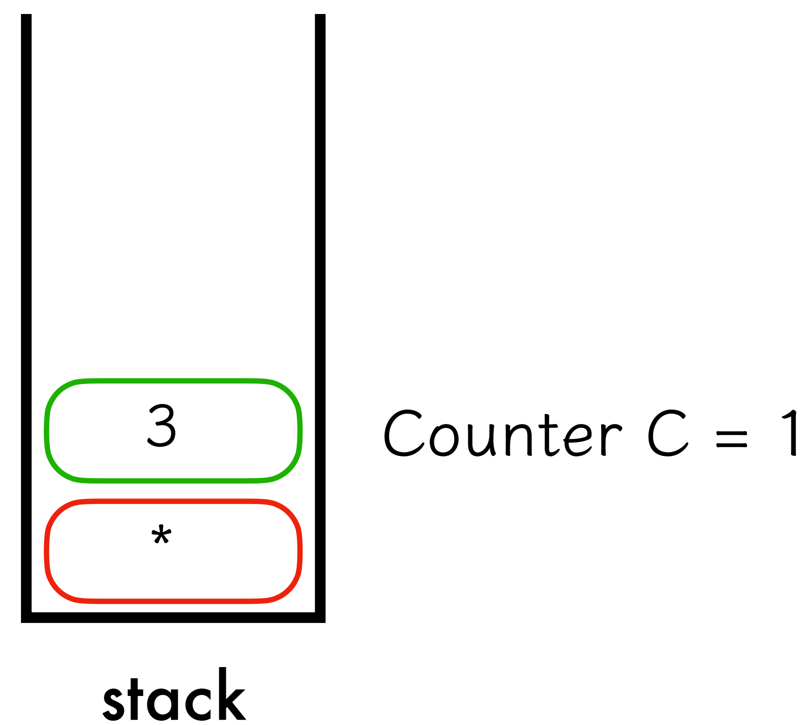


Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
    - if it is operator, update C with the arity.
    - each operand symbol, decrease C.
    - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
    - update C for new operator.

$a = 1,\ b = 2,\ c = 3$

Input: * + a 2 + b c

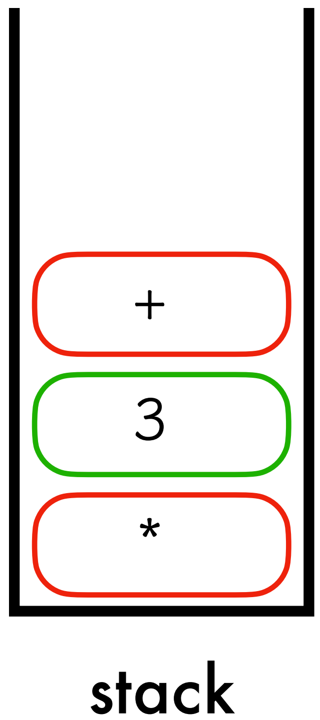


Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
  - if it is operator, update C with the arity.
  - each operand symbol, decrease C.
  - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
  - update C for new operator.

$a = 1,\ b = 2,\ c = 3$

Input: * + a 2 + b c



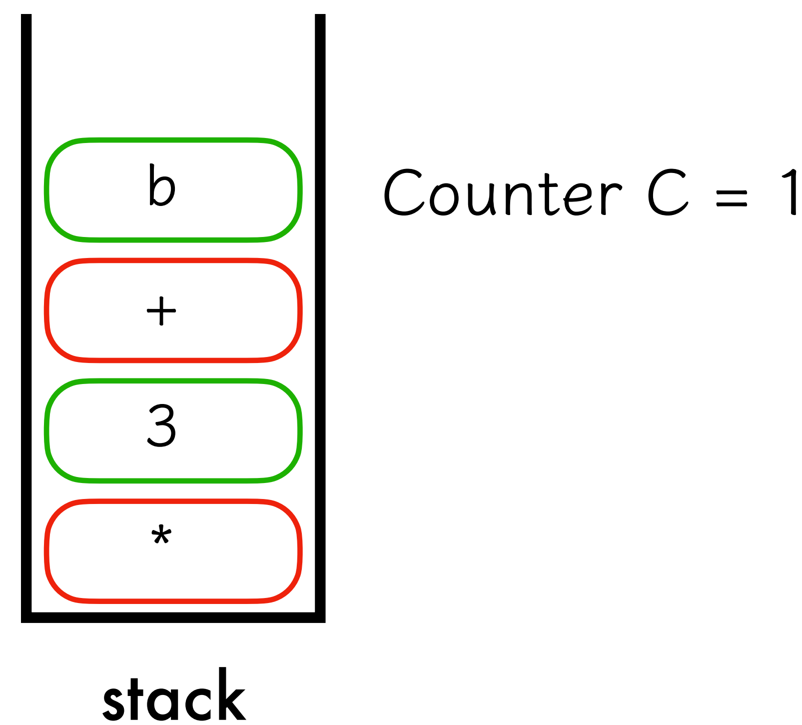
Counter C = 2

Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
    - if it is operator, update C with the arity.
    - each operand symbol, decrease C.
    - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
    - update C for new operator.

$a = 1,\ b = 2,\ c = 3$

Input: * + a 2 + b c



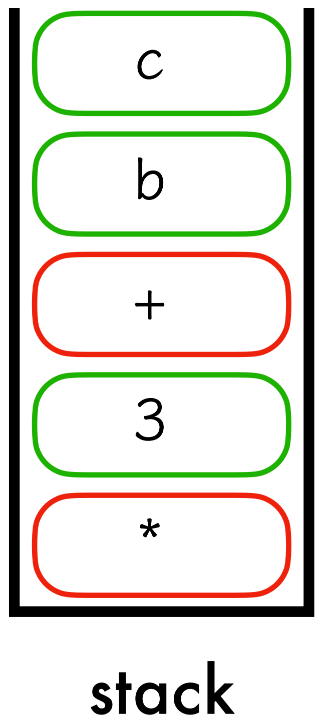
Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
  - if it is operator, update C with the arity.
  - each operand symbol, decrease C.
  - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
  - update C for new operator.

a = 1, b = 2, c = 3

Input: * + a 2 + b c

Evaluate!



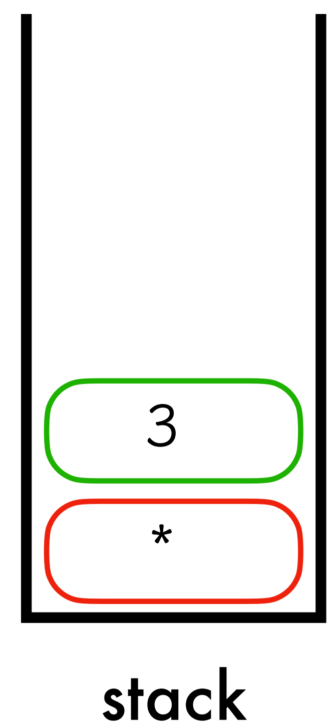
Counter C = 0

Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
    - if it is operator, update C with the arity.
    - each operand symbol, decrease C.
    - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
    - update C for new operator.

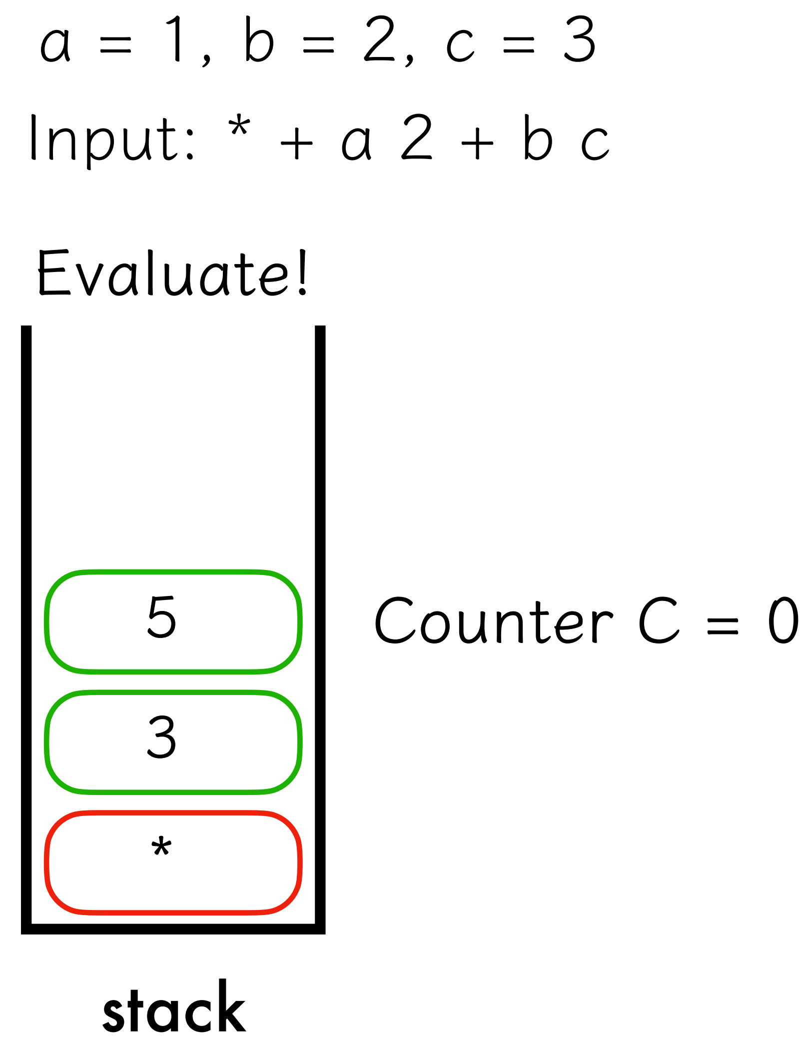
$a = 1,\ b = 2,\ c = 3$

Input: * + a 2 + b c



Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
  - if it is operator, update C with the arity.
  - each operand symbol, decrease C.
  - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
  - update C for new operator.

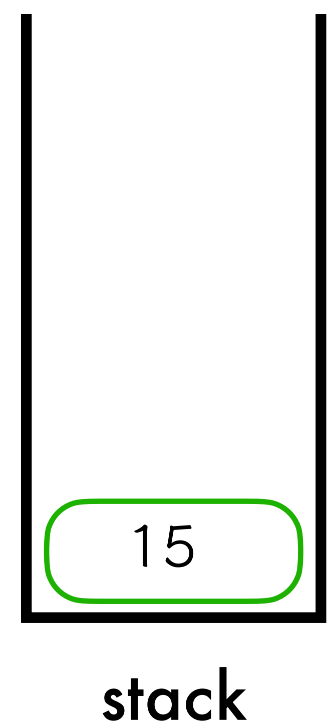


Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
  - if it is operator, update C with the arity.
  - each operand symbol, decrease C.
  - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
  - update C for new operator.

$a = 1,\ b = 2,\ c = 3$

Input: * + a 2 + b c

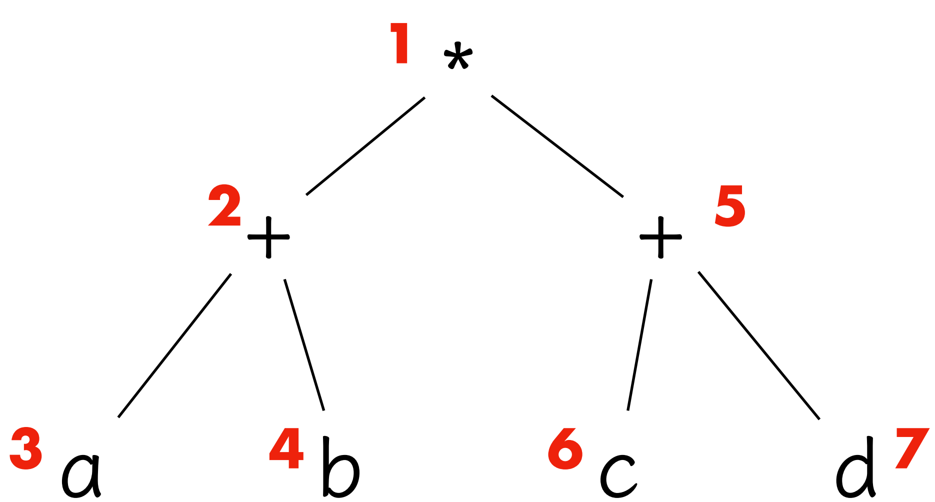


Postfix Notation

- In postfix notation, it is even simpler.
- We can read symbols from left to right, and every time we meet an operator, apply it to previous symbols based on its arity.
  - $a\ b + c\ d + *$, $a = 1$, $b = 2$, $c = 3$, $d = 4$
  - $\underline{a\ b\ +}\ c\ d + * \rightarrow 3\ \underline{c\ d\ +} \rightarrow \underline{3\ 7\ *} \rightarrow 21$

Using Syntax Tree

- We can also parse an expression into a syntax tree, then consider it with different traversal orders.
    - Non-leaf nodes are operators,
    - leaf nodes are operands.
- $a + b * c + d$



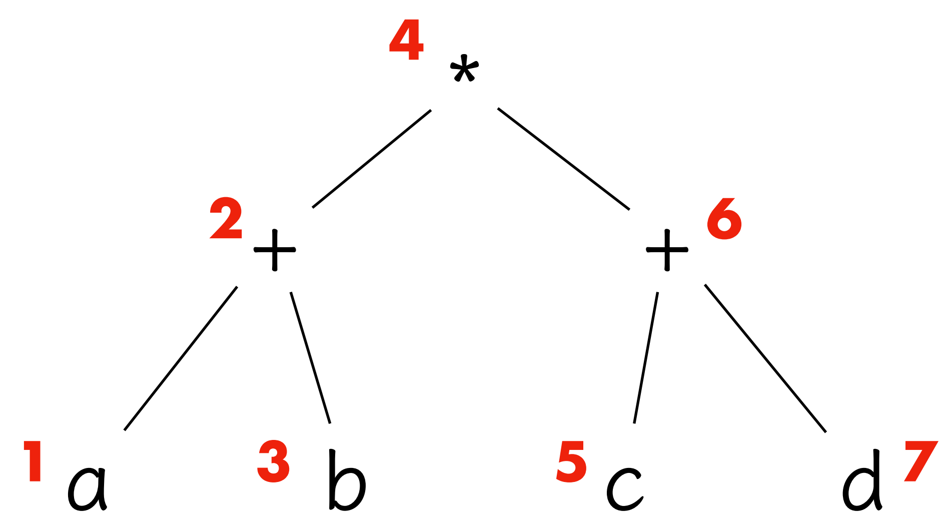
Pre-order $* + a\ b + c\ d$

In-order

Post-order

Using Syntax Tree

- We can also parse an expression into a syntax tree, then consider it with different traversal orders.
    - Non-leaf nodes are operators,
    - leaf nodes are operands.
- $a + b * c + d$



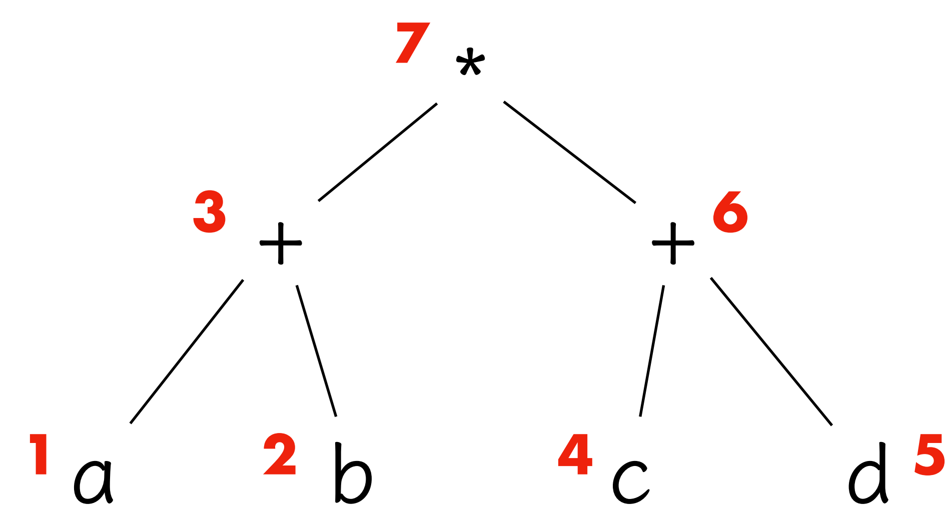
Pre-order $* + a\ b + c\ d$

In-order $a + b * c + d$

Post-order

Using Syntax Tree

- We can also parse an expression into a syntax tree, then consider it with different traversal orders.
  - Non-leaf nodes are operators,
  - leaf nodes are operands.
- a + b * c + d



Pre-order * + a b + c d

In-order a + b * c + d

Post-order a b + c d + *

Expression Evaluation

- In mathematics, $a - b + c$ and $a + c - b$ do not have different results - they are mathematically equivalent.
- However, in PL expressions, such ***Subexpression Evaluation Order*** can actually modify the result.
- Hence we have to consider subexpression evaluation order.
- There are several reasons why we should be careful.

Side Effect

- In imperative languages, it is possible that evaluation itself modifies the value of a variable through side effect.
- $(a + b{+}{+}) * (c + b{-}{-})$
- $(a + f(b)) * (c + f(d))$
- A component of a program has a side effect if it modifies the state of a program by execution.

Finite Arithmetic

- Numbers represented in a computer are ***finite***.
- e.g.) In C, we have different integer types such as `short`, `int`, and `long`, which can represent different range of integers.
- If the result of a computation (or evaluation of a subexpression) exceeds the boundaries, there will be overflow or underflow.
  - $a\text{-}b+c \rightarrow (a\text{-}b)+c$ vs. $(a+c)\text{-}b$, $b > c$
  - In computer there might be a problem for the latter, if $(a+c)$ is out of range, while the former can be OK due to $(a\text{-}b)$.

Undefined Operands

- Two strategies of Operator Application: ***eager evaluation*** or ***lazy evaluation***.
- *Eager evaluation* first computes all subexpressions, then apply operators.
- *Lazy evaluation* decides the evaluation of a subexpression later.
    - `a == 0 ? b : b/a` ➞ `"b over a" means "a divided by b".`
    - If we evaluate all the operands first, it will cause an error while evaluating `b/a`, since ***divide by 0 is undefined***.
    - But it is okay if we only evaluate an operand which need to be evaluated - `a == 0`, then `b` or `a != 0` then `b/a`.

Short-circuiting

- Short-circuiting is a technique to only evaluate a partial expression when the other is not required to be evaluated.
  - `if(str != null && str.length() > 0) ...`
  - If `str != null` is not satisfied, we don't need to evaluate `str.length()`.
  - Actually, evaluating `str.length()` before `str != null` will cause a problem.

Code Optimization

- The subexpression evaluation order may affect the efficiency of evaluation itself, considering code optimization.

  - ```
    a = array[i];
    b = a*a + c/d;
    ```

  - As you may already know, value of `a` should be read from the memory.

  - Hence it might be more efficient to evaluate `c/d` first.
    ```

Statement

- A *Statement* is a syntactic entity whose evaluation doesn't necessarily return a value, but can have a side effect.

- *Statements* are not present in all programming languages, but they are typically used by *Imperative Languages*.

- By executing (or evaluating) statements, we can keep changing a program's state.

    - e.g.) `print("Hello World!")`

Ambiguity in Definition

- We used the term "evaluation", which is not precisely and exactly defined, to define expression and statement.
- In different languages, an expression may have a side-effect, and a statement can have a return value.
  - In C, an assignment modifies the value of a variable, as well as returns the value.
- The key distinction is that when the state is fixed before the evaluation,
  - the result of expression evaluation is ***a value,***
  - while the result of statement evaluation is ***change of the state***.

The Concept of Variable

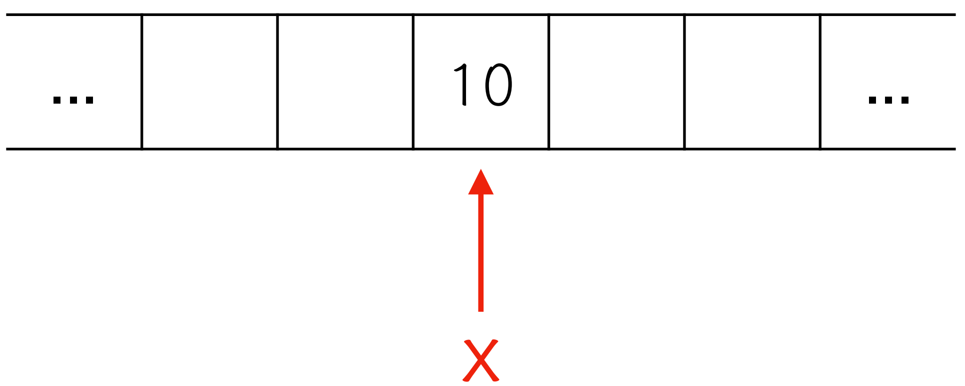
- In programming languages, two models of variables are employed.
- Modifiable Variable
  - A variable is considered as a container or location, which stores a value.
  - The value is "***modifiable***", by executing assignments.
- Reference Model
  - A variable is considered as a reference to a value stored in the memory, not a container of a value.

The Concept of Variable

- In modifiable variable, a variable itself is a container.
- In reference model, a variable is merely a reference to a memory location.
- Note that this is the concept of variable, and its implementation can be different in each language.

Modifiable Variable

Reference Model



Assignment

- ***Assignment*** is a statement which modifies a value associated with a modifiable variable.

- `<assign> ::= <expr1><opAssign><expr2>`

- For `<expr1>`, we use the ***l-value***, and for `<expr2>` we need the ***r-value***.

    - `x = 3; x = x + 1;`

    - On the left side, we use *l-value of* `x` *(the location)*, and on the right side, we use *r-value of* `x` *(value 3)*.

Assignment

- How assignment works with a variable of reference model?
  - `x = y`
  - It doesn't mean copying the value of `y` to variable `x`.
  - Rather they are now ***two references to the same object***.
    - We can modify `y` and it can be seen via `x`.
    - Similar to pointer variables, but in reference model, we can only modify the value indirectly with assignments.
- Java is a language employs reference model for variables of class types.

Summary

- Expressions and Notations
- Which should be considered for Expression Evaluation?
- Statements
- Variable and Assignment