EASTERN WASHINGTON UNIVERSITY
start something big

Stack 3

Computer Science Department
Eastern Washington University
Yun Tian (Tony) Ph.D.

Recall Last Class

- Demo of Linked List Stack
- Applications of Stack
  - Reverse List/Array
  - Decimal to Binary Number
  - Parenthesis Matching

Today

- Infix Expression and Postfix Expression
- Homework 5 and 6

Concept of Infix and Postfix Expressions

- Infix expressions
  - These are the ordinary expressions we use in programing.
    - It is a crucial task for the Compiler to evaluate them.
  - Operators are in between operands.
  - You have walk though the whole expression before starting to evaluate, **why?**
    - E.g. 3 + 4 * 2,
    - You cannot do 3 + 4 first.

Concept of Infix and Postfix Expressions

- Postfix expressions,
  - Precedence and associativity are build-in to the expression.
  - So you can evaluate it from left to right.
  - E.g 3 4 2 * +

Concept of Infix and Postfix Expressions

- To Evaluate Infix Expression
  - We can first transfer infix to postfix expression.
  - Then we evaluate postfix expression to yield a final results.
- Both steps above need Stack data structure.

Examples of Infix and Postfix

- Infix: 2 + 3
  - Postfix: 2 3 +
- Infix: 2 * 3 + 4
  - Postfix: 2 3 * 4 +
- Infix: 2 + 3 * 4
  - Postfix: 2 3 4 * +
- Infix: (2 + 3) * 4
  - Postfix: 2 3 + 4 *

Postfix Evaluation algorithm

```
While there are items in the the postfix expression
{
        if the current item is an operand,
                push it onto stack.
        else
                pop stack and attach operand to right of the operator
                pop stack again and attach operand to the left of the operator
                evaluate and push the results

        advanced to next item in postfix expression,
}
if only one number left on stack,
        pop and this is the final result.
else
        postfix has error in it.(probably the original infix has syntax error.)
```

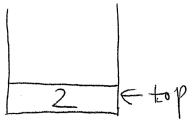
Postfix Evaluation Example

postfix: 2 3 * 4 +

while Loop first iteration

current item is '2', is operand;

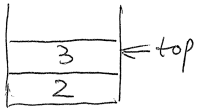
push('2');



while Loop second iteration

current is '3', is operand

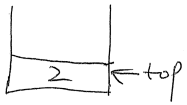
push('3')



while Loop third iteration

current is '*' is operator

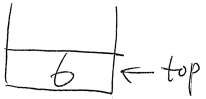
pop stack, right = 3



pop stack left = 2

evaluate: push(left * right)

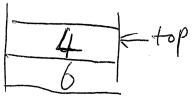
push(6)



while Loop Fourth iteration:

current item is '4', is operand.

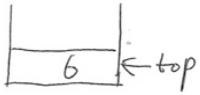
push('4')



while Loop ~~fifth~~ iteration

Current item is '+' ; operator

right = pop() = 4

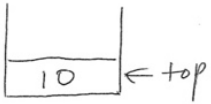


left = pop() = 6

evaluation

push (left '+' right)

push (10)



Then while Loop drops out!

only one number on the stack, is 10,

10 is the final result for the original postfix expression.

Note here:

① we learned that items on stack usually have the same data type.

② Here, we assume the operands and operators are stored in a string, and operands have only one digit. What if an operands have arbitrary number of digits?

③ How to solve this problem when you implement this algorithm?

Transform from infix to postfix

- Infix: 2 + 3 * 4
- Corresponding Postfix is, 2 3 4 * +
- We observe that operands remains the same order in both infix and postfix.
- Postfix is evaluated from left to right.
- The operators in postfix in the left has higher precedence or have to be performed earlier than the right ones.

```
//Pseudo code of converting infix to postfix expression
while there are items in Infix expression {
        if current items is an operand,
                write it to Postfix Expression;
        else if current item is a '('
                push it onto the stack;
        else if our current item is ')' {
                while the item on top of the stack is not a '('
                        pop stack and write the operator to the end of postfix expression.

                pop the '(' and throw it away.
        }
        else { // handle arithmetic operator, but here we only care about the binary operators.
                while Stack is not empty AND the precedence of item on top of stack is greater than
                    the current item. {
                        pop stack and write that operator to the end of postfix expression
                }
                push current item onto stack
        }
        advance to next item in infix expression
}
while stack is not empty {
        pop and write the operator to postfix expression.
}
```

Infix to Postfix Expression

- We observe that operands are not pushed onto stack, but only operators and ‘(‘.
  - Use stack to rearrange the order of the operators based on their precedence and associativity.
- Basic Idea of when processing an operator,
  - If we had some operators on the stack with higher precedence than the current item, we have to output those operators first to the postfix expression.

Infix to Postfix Expression

- How to determine the precedence of different operators?

Infix to Postfix Expression

- You have to use this precedence value lookup table in your program.
- ^ has associativity of right to left, the rest are left to right.

	(	)	^	*	/	%	+	-
On Stack	0	----	5	4	4	4	2	2
Current item	100	0	6	3	3	3	1	1

Infix to Postfix Expression

- *, / and % has Left to Right associativity.
  - Operators on the stack has bigger precedence value, and first performed than the current item.
    - Have to be written into postfix earlier, i.e. in the left.
    - Postfix is evaluated from left to right.
    - The operators in postfix in the left has higher precedence or have to be performed early than the right ones.
  - Same arguments with the + and – operator.

Infix to Postfix Expression

- The ^ operator (exponential) has Right to Left associativity.
  - The current item has a bigger precedence value than the one on the stack (previous one).
  - A similar argument could be made as in the previous slide as to the order of being written into the postfix expression.

Next Class

- Talk about next homework