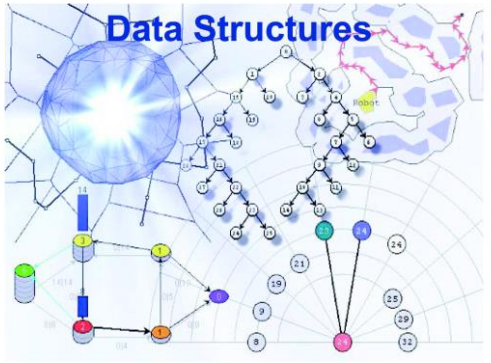
BBM 201
DATA STRUCTURES

Lecture 6:

EVALUATION of EXPRESSIONS



2018-2019 Fall

Evaluation of Expressions

- Compilers use stacks for the arithmetic and logical expressions.
- **Example:** x=a/b-c+d*e-a*c
- If a=4, b=c=2, d=e=3 what is x?
  - ((4/2)-2)+(3*3)-(4*2), (‘/’ and ‘*’ have a priority)
- There may be also parenthesis, such as:
  - a/(b-c)+d*(e-a)*c
  - **How does the compiler solve this problem?**

Infix, prefix, postfix

- Normally, we use ‘infix’ notation for the arithmetic expressions:
  - Infix notation: a+b
- However, there is also ‘prefix’ and ‘postfix’ notation:
  - Prefix notation: +ab
  - Postfix notation: ab+

- Infix : 2+3*4
- Postfix: 234*+
- Prefix: +2*34

Prefix

$+ 2 * 3 5 =$

$= + 2 \underline{* 3 5}$

$= \underline{+ 2 15} = 17$

$* + 2 3 5 =$

$= * \underline{+ 2 3} 5$

$= \underline{* 5 5} = 25$

Postfix

2 3 5 * + =

= 2 3 5 * +

= 2 15 + = 17

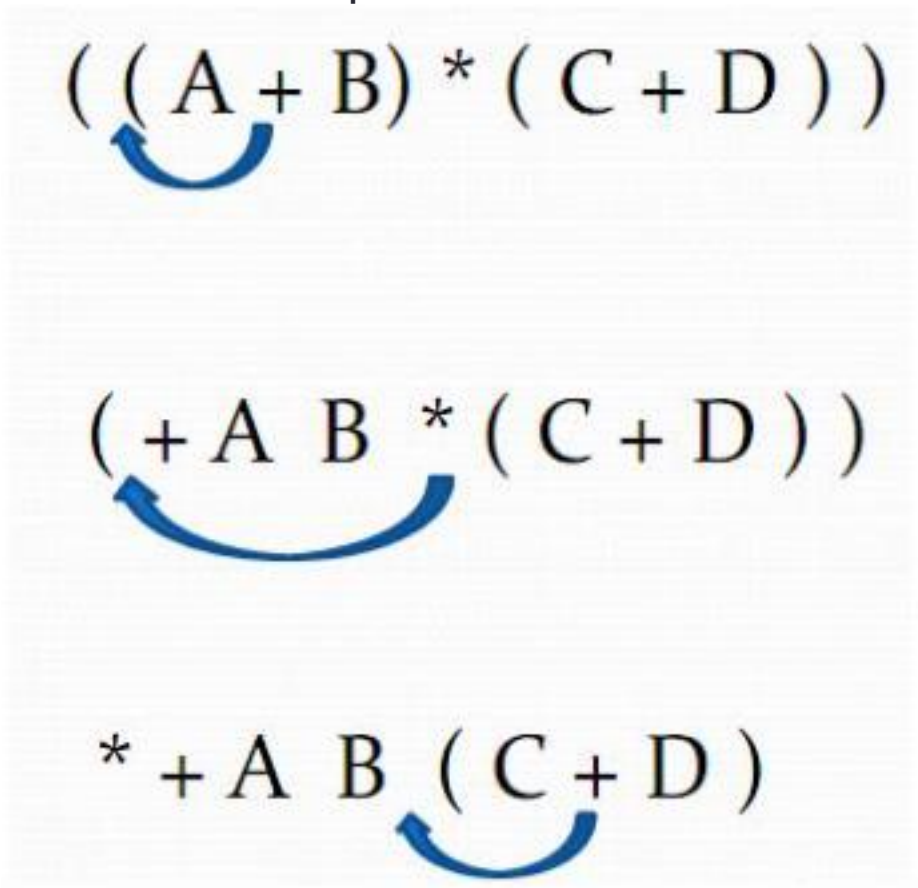
2 3 + 5 * =

= 2 3 + 5 *

= 5 5 * = 25

How to convert infix to prefix?

Move each operator to the left of the operands:

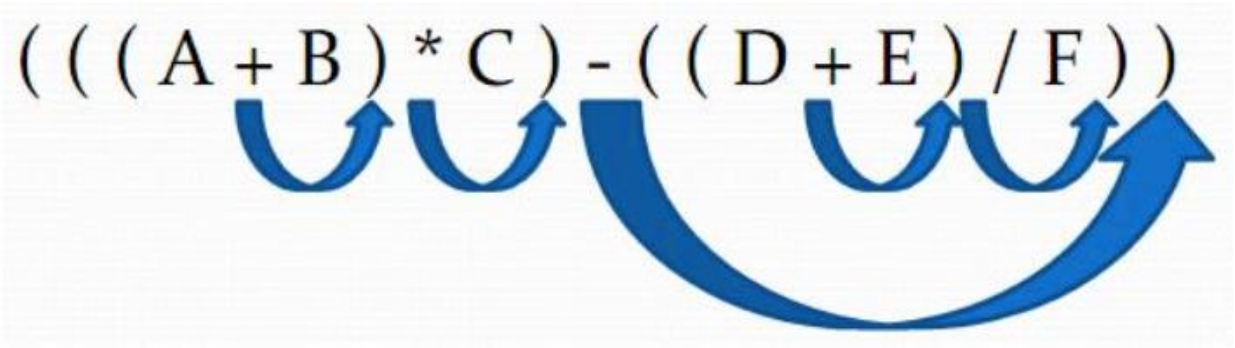


*** + A B + C D**

Operand order does not change!

How to convert infix to postfix?

Move each operator to the right of the operands:



- ((AB+* C) - ((D + E) / F))
- (AB+C* - ((D + E) / F))
- AB+C* ((D + E) / F)-
- AB+C* (DE+ / F)-
- A B + C * D E + F / -

Operand order still does not change!

Example - 1

- Infix: (a+b)*c-d/e

- Postfix: ???
- Prefix: ???

Example – 1 (solution)

- Infix: (a+b)*c-d/e

- Postfix: ab+c*de/-
- Prefix: -*+abc/de

Example - 2

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

- Postfix: ???
- Prefix: ???

Example – 2 (solution)

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

- Postfix: ab/c-de*+ac*-
- Prefix:-+-/abc*de*ac

Example – 3

- Infix: (a/(b-c+d))*(e-a)*c

- Postfix: ???
- Prefix: ???

Example – 3 (solution)

- Infix: (a/(b-c+d))*(e-a)*c

- Postfix: abc-d+/ea-*c*
- Prefix: **/a+-bcd-eac

Expressions

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

Infix, prefix, postfix

Infix	Postfix	Prefix
A+B-C	AB+C-	-+ABC
(A+B)*(C-D)	AB+CD-*	*+AB-CD
A^B*C-D+E/F/(G+H)	AB^C*D-EF/GH+/+	+-*^ABCD//EF+GH
((A+B)*C-(D-E))^(F+G)	AB+C*DE—FG+^	^-*+ABC-DE+FG
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

Why postfix?

- For the infix expressions we have two problems:
  - Parenthesis
  - Operation precedence

  - Example: ((4/2)-2)+(3*3)-(4*2) (infix)
  - 42/2-33*+42*- (postfix)

Operator PRECEDENCE

Operators						Associativity	Type
++	--	+	-	!	(type)	right to left	unary
*	/	%				left to right	multiplicative
+	-					left to right	additive
<	<=	>	>=			left to right	relational
==	!=					left to right	equality
&&						left to right	logical AND
||						left to right	logical OR
?:						right to left	conditional
=	+=	-=	*=	/=	%=	right to left	assignment
,						left to right	comma

Fig. 4.16 Operator precedence and associativity.

Parentheses are used to override precedence.

EVALUATION OF INFIX OPERATIONS
(fully Parenthesized)

1. Read one input character
2. Actions at the end of each input

Opening brackets	(2.1) *Push* into stack and then Go to step (1)
Number	(2.2) *Push* into stack and then Go to step (1)
Operator	(2.3) *Push* into stack and then Go to step (1)
Closing brackets	(2.4) *Pop* from character stack
	(2.4.1) if it is opening bracket, then discard it, Go to step (1)
	(2.4.2) *Pop* is used four times
	The first popped element is assigned to op2
	The second popped element is assigned to op
	The third popped element is assigned to op1
	The fourth popped element is the remaining opening bracket, which can be discarded
	Evaluate op1 op op2
	Convert the result into character and *push* into the stack
	Go to step (1)
New line character	(2.5) *Pop* from stack and print the answer *STOP*

(((2 * 5) - (1 * 2)) / (11 - 9))

Input Symbol	Stack (from bottom to top)	Operation
(	(	
(	((	
(	(((	
2	(((2	
*	(((2 *	
5	(((2 * 5	
)	((10	2 * 5 = 10 and *push*
-	((10 -	
(	((10 - (	
1	((10 - (1	
*	((10 - (1 *	
2	((10 - (1 * 2	
)	((10 - 2	1 * 2 = 2 & *Push*
)	(8	10 - 2 = 8 & *Push*
/	(8 /	
(	(8 / (	
11	(8 / (11	
-	(8 / (11 -	
9	(8 / (11 - 9	
)	(8 / 2	11 - 9 = 2 & *Push*
)	4	8 / 2 = 4 & *Push*
New line	Empty	*Pop* & Print

EVALUATION OF INFIX OPERATIONS

(Not fully Parenthesized)

(1) Read an input character

(2) Actions that will be performed at the end of each input

Opening parentheses	(2.1) *Push* it into character stack and then Go to step 1
Number	(2.2) *Push* into integer stack then Go to step 1
Operator	(2.3) Do the comparative priority check

(2.3.1) If the character stack's *top* contains an operator with equal or higher priority,

Then **process**

(2.3.1.1) If the character stack is empty, Go to step 2.3.2

(2.3.1.2) Else, Go to step 2.3

(2.3.2) Else, *Push* the input character to the character stack and Go to step 1

Closing par.	(2.4) *Peek* the *top* element of character stack

(2.4.1) If it is an opening parentheses then *Pop* from character stack and Go to step 1

(2.4.2) Else, **process** Go to the step 2.4

New line character	(2.5) If the character stack is not empty

(2.5.1) Then **process** and Go to step 2.5

(2.5.2) Else, print the result after popping from the integer stack ***STOP***

Process: (1) *Pop* from character stack to op

(2) *Pop* from integer stack to op2

(3) *Pop* from integer stack to op1

(4) Calculate op1 op op2 and *Push* the result into the integer stack

(2*5-1*2)/(11-9)

Input Symbol	Character Stack (from bottom to top)	Integer Stack (from bottom to top)	Operation performed
(	(		
2	(	2	
*	(*		Push as * has higher priority
5	(*	2 5	
-	(*		Since '-' has less priority, we do 2 * 5 = 10
	(-	10	We push 10 and then push '-'
1	(-	10 1	
*	(- *	10 1	Push * as it has higher priority
2	(- *	10 1 2	
)	(-	10 2	Perform 1 * 2 = 2 and push it
	(	8	Pop - and 10 - 2 = 8 and push, Pop (
/	/	8	
(	/ (	8	
11	/ (	8 11	
-	/ (-	8 11	
9	/ (-	8 11 9	
)	/	8 2	Perform 11 - 9 = 2 and push it
		4	Perform 8 / 2 = 4 and push it
New line		4	Print the output, which is 4

Evaluation of a prefix operation

Input: / - * 2 5 * 1 2 - 11 9

Output: 4

Data structure requirement: a character stack and an integer stack

1. Read one character input at a time and keep pushing it into the character stack until the new line character is reached
2. Perform *pop* from the character stack. If the stack is empty, go to step (3)

Number (2.1)	*Push* into the integer stack and then go to step (2)
Operator (2.2)	Assign the operator to op
	Pop a number from integer stack and assign it to op1
	Pop another number from integer stack and assign it to op2
	Calculate op1 op op2 and push the output into the int. stack.
	Go to step (2)

3. *Pop* the result from the integer stack and display the result

/ - * 2 5 * 1 2 -11 9

/	/		
-	/-		
*	/ - *		
2	/ - * 2		
5	/ - * 2 5		
*	/ - * 2 5 *		
1	/ - * 2 5 * 1		
2	/ - * 2 5 * 1 2		
-	/ - * 2 5 * 1 2 -		
11	/ - * 2 5 * 1 2 - 11		
9	/ - * 2 5 * 1 2 - 11	9	
\n	/ - * 2 5 * 1 2 - 11	9	
	/ - * 2 5 * 1 2 -	9 11	
	/ - * 2 5 * 1 2	2	11 - 9 = 2
	/ - * 2 5 * 1	2 2	
	/ - * 2 5 *	2 2 1	
	/ - * 2 5	2 2	1 * 2 = 2
	/ - * 2	2 2 5	
	/ - *	2 2 5 2	
	/ -	2 2 10	5 * 2 = 10
	/	2 8	10 - 2 = 8
	Stack is empty	4	8 / 2 = 4
		Stack is empty	Print 4

POSTFIX

Compilers typically use a parenthesis-free notation (postfix expression).

The expression is evaluated from the left to right using a stack:

- when encountering an operand: push it
- when encountering an operator: pop two operands, evaluate the result and push it.

Evaluation of a postfix expression

Token	Stack [0]	[1]	[2]	Top
4	4			0
2	4	2		1
/	4/2			0
2	4/2	2		1
-	(4/2)-2			0
3	(4/2)-2	3		1
3	((4/2)-2)	3	3	2
*	((4/2)-2)	3*3		1
+	((4/2)-2)+(3*3)			0
4	((4/2)-2)+(3*3)	4		1
2	((4/2)-2)+(3*3)	4	2	2
*	((4/2)-2)+(3*3)	4*2		1
-	((4/2)-2)+(3*3)-(4*2)			0

62/3-42*+

Token	Stack [0]	[1]	[2]	Top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
–	6/2–3			0
4	6/2–3	4		1
2	6/2–3	4	2	2
*	6/2–3	4*2		1
+	6/2–3+4*2			0

How to evaluate a postfix evaluation?

```
typedef enum
{left_parent,right_parent,add,subtract,multiply,divide,eos,operand}
precedence;

char expr[]= "422-3+/34-*2*";

precedence get_token(char* symbol, int* n){

    *symbol=expr[(*n)++];
     switch(*symbol){
              case '(': return left_parent;
              case ')': return right_parent;
              case '+': return add;
              case '-': return subtract;
              case '/': return divide;
              case '*': return multiply;
              case '\0': return eos;
              default: return operand;
     }
}
```

How to evaluate a postfix evaluation?

```
float eval(void){
    char symbol;
    precedence token;
    float op1, op2;
    int n=0;
    int top=-1;

    token = get_token(&symbol, &n);    //take a token

    while(token=eos){   //end of string?
            if(token==operand)
                push(&top, symbol- '0');
            else{
                op2=pop(&top);
                op1=pop(&top);
                switch(token){
                        case add: push(&top,op1+op2);
                        case subtract: push(&top,op1-op2);
                        case multiply: push(&top,op1*op2);
                        case divide: push(&top,op1/op2);
                 }
             }
            token=get_token(&symbol,&n);
     }
     return pop(&top);
}
```

CONVERT an INFIX to POSTFIX

a+b*c

Token	Stack [0]	[1]	[2]	Top	Output
a				−1	*a*
+	+			0	*a*
b	+			0	*ab*
*	+	*		1	*ab*
c	+	*		1	*abc*
eos				−1	*abc**+

a*(b+c)*d

Token	Stack [0]	[1]	[2]	Top	Output
a				−1	*a*
*	*			0	*a*
(	*	(		1	*a*
b	*	(		1	*ab*
+	*	(	+	2	*ab*
c	*	(	+	2	*abc*
)	*			0	*abc* +
*	*			0	*abc* +*
d	*			0	*abc* +**d*
eos	*			0	*abc* +**d**

How to convert infix to postfix?

```
char expr[]=" (4/(2-2+3))*(3-4)*2";
static int stack_pre[]={0,19,12,12,13,13,13,0};
static int pre[]={20,19,12,12,13,13,13,0};

void postfix(void){
  char symbol;
  precedence token;
  int n=0;
  int top=0;
  stack[0]=eos;

  for(token=get_token(&symbol,&n);token!=eos;token=get_token(&symbol,&n)){
          if(token==operand)
               printf("%c", symbol);
          else if(token==right_parent)
               while(stack[top]!=left_parent)
                  print_token(pop(&top));
               pop(&top);
          }
          else{
                while(stack_pre[stack[top]]>=pre[token])
                    print_token(pop(&top));
               push(&top, token);
           }
 }
 while((token=pop(&top))!=eos)
       print_token(token)
}
```

Exercise to do at home:

1. Write the code that converts infix to prefix.
2. Write the code that evaluates a prefix expression.