

Lecture Fourteen Infix, Prefix and Postfix Expressions

Algebraic Expression

- ☛ An algebraic expression is a legal combination of operands and the operators.
- ☛ Operand is the quantity (unit of data) on which a mathematical operation is performed.
- ☛ Operand may be a variable like x , y , z or a constant like 5 , 4 , 0 , 9 , 1 etc.
- ☛ Operator is a symbol which signifies a mathematical or logical operation between the operands. Example of familiar operators include $+$, $-$, $*$, $/$, $^$
- ☛ An example of expression as $x+y*z$.

Infix, Postfix and Prefix Expressions

- ☞ INFIX: the expressions in which operands surround the operator, e.g. $x+y$, $6*3$ etc this way of writing the Expressions is called infix notation.
- ☞ POSTFIX: Postfix notation are also Known as Reverse Polish Notation (RPN). They are different from the infix and prefix notations in the sense that in the postfix notation, operator comes after the operands, e.g. $xy+$, $xyz+*$ etc.
- ☞ PREFIX: Prefix notation also Known as Polish notation. In the prefix notation, operator comes before the operands, e.g. $+xy$, $*+xyz$ etc.

Operator Priorities

- ☞ How do you figure out the operands of an operator?
 - $a + b * c$
 - $a * b + c / d$
- ☞ This is done by assigning operator priorities.
 - $\text{priority}(*) = \text{priority}(/) > \text{priority}(+) = \text{priority}(-)$
- ☞ When an operand lies between two operators, the operand associates with the operator that has higher priority.

Tie Breaker

- When an operand lies between two operators that have the same priority, the operand associates with the operator on the left.

- $a + b - c$
- $a * b / c / d$

Delimiters

- Sub expression within delimiters is treated as a single operand, independent from the remainder of the expression.

- $(a + b) * (c - d) / (e - f)$

WHY??

- ☞ Why to use PREFIX and POSTFIX notations when we have simple INFIX notation?
- ☞ INFIX notations are not as simple as they seem specially while evaluating them. To evaluate an infix expression we need to consider Operators' Priority and Associative property
 - E.g. expression $3+5*4$ evaluate to 32 i.e. $(3+5)*4$ or to 23 i.e. $3+(5*4)$.
- ☞ To solve this problem Precedence or Priority of the operators were defined. Operator precedence governs evaluation order. An operator with higher precedence is applied before an operator with lower precedence.

Infix Expression is Hard To Parse

- ☞ Need operator priorities, tie breaker, and delimiters.
- ☞ This makes computer evaluation more difficult than is necessary.
- ☞ Postfix and prefix expression forms do not rely on operator priorities, a tie breaker, or delimiters.
- ☞ Both prefix and postfix notations have an advantage over infix that while evaluating an expression in prefix or postfix form we need not consider the Priority and Associative property (order of brackets).
 - E.g. $x/y*z$ becomes $*/xyz$ in prefix and $xy/z*$ in postfix. Both prefix and postfix notations make Expression Evaluation a lot easier.
- ☞ So it is easier to evaluate expressions that are in these forms.

Examples of infix to prefix and post fix

Infix	Postfix	Prefix
$A+B$	$AB+$	$+AB$
$(A+B) * (C + D)$	$AB+CD+*$	$*+AB+CD$
$A-B/(C*D^E)$	$ABCDE^*/-$	$-A/B*C^DE$

Saturday, April 18, 2015

Data Structure

9

Example: postfix expressions

- ☞ Postfix notation is another way of writing arithmetic expressions.
- ☞
- ☞ In postfix notation, the operator is written after the two operands.
 - infix: $2+5$ postfix: $2\ 5\ +$
- ☞ Expressions are evaluated from left to right.
- ☞
- ☞ Precedence rules and parentheses are never needed!!

Saturday, April 18, 2015

Data Structure

10

Suppose that we would like to rewrite $A+B*C$ in postfix

☞ Applying the rules of precedence, we obtained

$A+B*C$

$A+(B*C)$ Parentheses for emphasis

$A+(BC^*)$ Convert the multiplication, Let $D=BC^*$

$A+D$ Convert the addition

$A(D)+$

ABC^*+ **Postfix Form**

Postfix Examples

Infix	Postfix	Evaluation
$2 - 3 * 4 + 5$	$2 3 4 * - 5 +$	-5
$(2 - 3) * (4 + 5)$	$2 3 - 4 5 + *$	-9
$2 - (3 * 4 + 5)$	$2 3 4 * 5 + -$	-15

☞ Why ? No brackets necessary !

Algorithm for Infix to Postfix

- ☞ Examine the next element in the input.
- ☞ If it is operand, output it.
- ☞ If it is opening parenthesis, push it on stack.
- ☞ If it is an operator, then
 - i) If stack is empty, push operator on stack.
 - ii) If the top of stack is opening parenthesis, push operator on stack
 - iii) If it has higher priority than the top of stack, push operator on stack.
 - iv) Else pop the operator from the stack and output it, repeat step 4
- ☞ If it is a closing parenthesis, pop operators from stack and output them until an opening parenthesis is encountered. pop and discard the opening parenthesis.
- ☞ If there is more input go to step 1
- ☞ If there is no more input, pop the remaining operators to output.

Saturday, April 18, 2015

Data Structure

13

Suppose we want to convert $2*3/(2-1)+5*3$ into Postfix form,

Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(/(23*
2	/(23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/ 23*21-/5
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

Saturday, April 18, 2015

Data Structure

14

Example

☞ $(5 + 6) * 9 + 10$

will be

☞ $56 + 9 * 10 +$

Evaluation a postfix expression

- ☞ Each operator in a postfix string refers to the previous two operands in the string.
- ☞ Suppose that each time we read an operand we push it into a stack. When we reach an operator, its operands will then be top two elements on the stack
- ☞ We can then pop these two elements, perform the indicated operation on them, and push the result on the stack.
- ☞ So that it will be available for use as an operand of the next operator.

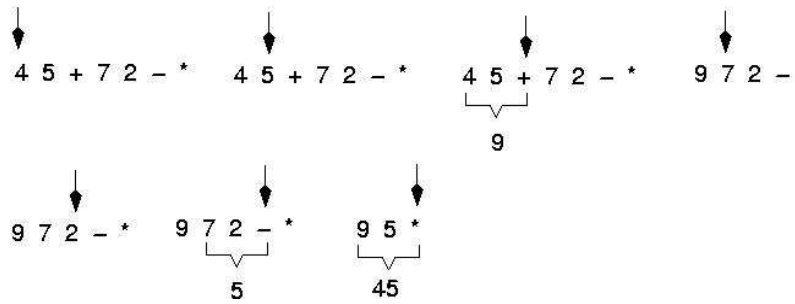
Evaluating Postfix Notation

- ☞ Use a stack to evaluate an expression in postfix notation.
- ☞ The postfix expression to be evaluated is scanned from left to right.
- ☞ Variables or constants are pushed onto the stack.
- ☞ When an operator is encountered, the indicated action is performed using the top elements of the stack, and the result replaces the operands on the stack.

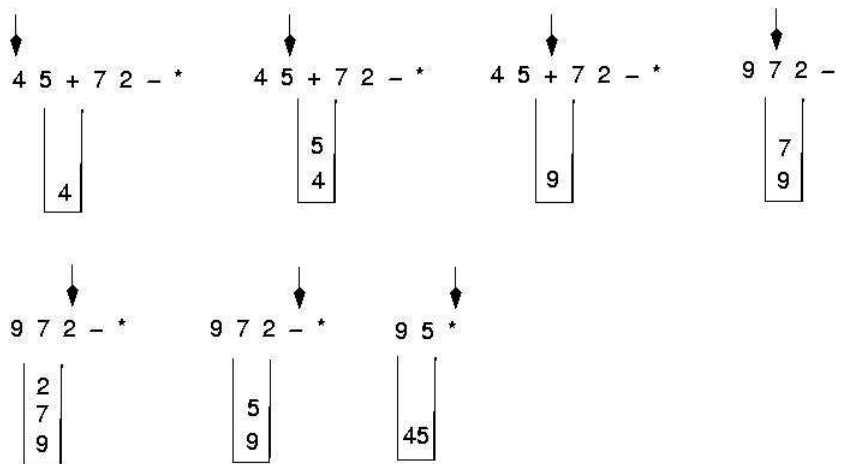
Evaluating a postfix expression

- ☞ Initialise an empty stack
- ☞ While token remain in the input stream
 - Read next token
 - If token is a number, push it into the stack
 - Else, if token is an operator, pop top two tokens off the stack, apply the operator, and push the answer back into the stack
- ☞ Pop the answer off the stack.

Example: Postfix Expressions



Postfix expressions: Algorithm using stacks (cont.)



Algorithm for evaluating a postfix expression (Cond.)

```

WHILE more input items exist
{
  If symb is an operand
    then push (opndstk,symb)
  else //symbol is an operator
  {
    Opnd1=pop(opndstk);
    Opnd2=pop(opndnstk);
    Value = result of applying symb to opnd1 & opnd2
    Push(opndstk,value);
  }
  //End of else
} // end while
Result = pop (opndstk);

```

Saturday, April 18, 2015

Data Structure

21

Question : Evaluate the following expression in postfix :
 $623+-382/+*2^3+$

Final answer is

- 49
- 51
- 52
- 7
- None of these

Saturday, April 18, 2015

Data Structure

22

Evaluate- 623+-382/+*2^3+

Symbol	opnd1	opnd2	value	opndstk
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3

Saturday, April 18, 2015

Data Structure

23

Evaluate- 623+-382/+*2^3+

Symbol	opnd1	opnd2	value	opndstk
8	6	5		1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3		4	7
	1, 7			
*	1	7	7	7
2	1	7	7	7, 2
^	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Saturday, April 18, 2015

Data Structure

24

Evaluating a Postfix Expression in C++

```
#include "stackd.h"
#include <assert.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#define MAX_SIZE_EXPRESSION 100
int main(void);
int evaluate_postfix(char *expression,
double *result);
int evaluate_operator(int operator_symbol,
double first_operand, double second_operand,
double *result);
```

Saturday, April 18, 2015

Data Structure

25

Evaluating a Postfix Expression in C++

```
#include "stackd.h"
#include <assert.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#define MAX_SIZE_EXPRESSION 100
int main(void);
int evaluate_postfix(char *expression,
double *result);
int evaluate_operator(int operator_symbol,
double first_operand, double second_operand,
double *result);
```

Saturday, April 18, 2015

Data Structure

26

Evaluating a Postfix Expression in C++

```
int main(void)
{
    char expression[MAX_SIZE_EXPRESSION];
    int position = 0;
    double value;
    do
    {
        expression[position] = getchar();
        position++;
    } while (expression[position - 1] != '\n'
        && position < MAX_SIZE_EXPRESSION);
```

Saturday, April 18, 2015

Data Structure

27

Evaluating a Postfix Expression in C++

```
expression[position - 1] = '\0';
if (!evaluate_postfix(expression, &value))
{
    return 1;
}
printf("Expression \"%s\" evaluates to %g.\n",
    expression, value);
return 0;
}
```

Saturday, April 18, 2015

Data Structure

28

Evaluating a Postfix Expression in C++

```
int evaluate_postfix(char *expression,
double *result)
{
int position;
stackd operand_stack;
static char *digits = "0123456789";
assert(NULL != result && NULL != expression);
if (!stackd_init(&operand_stack, 0))
return FALSE;
```

Saturday, April 18, 2015

Data Structure

29

Evaluating a Postfix Expression in C++

```
for (position = 0;
'\0' != expression[position]; position++)
{
if (NULL !=
strchr(digits, expression[position]))
{
if (!stackd_push(&operand_stack,
(double)(strchr(digits,
expression[position]) - &digits[0])))
break;
}
}
```

Saturday, April 18, 2015

Data Structure

30

Evaluating a Postfix Expression in C++

```

else
{
double first_operand, second_operand;
double value;
if (stackd_empty(&operand_stack))
break;
second_operand =
stackd_pop(&operand_stack);
if (stackd_empty(&operand_stack))
break;
first_operand =
stackd_pop(&operand_stack);

```

Saturday, April 18, 2015

Data Structure

31

Evaluating a Postfix Expression in C++

```

if (!evaluate_operator(
expression[position], first_operand,
second_operand, &value))
break;
if (!stackd_push(&operand_stack, value))
break;
} /* end else */
} /* end for */

```

Saturday, April 18, 2015

Data Structure

32

Evaluating a Postfix Expression in C++

```

if ('\0' != expression[position]
|| stackd_empty(&operand_stack))
{
printf("syntax error.\n");
stackd_deinit(&operand_stack);
return FALSE;
}
*result = stackd_pop(&operand_stack);

```

Saturday, April 18, 2015

Data Structure

33

Evaluating a Postfix Expression in C++

```

int evaluate_operator(int operator_symbol,
double first_operand, double second_operand,
double *result)
{
assert(NULL != result);
switch (operator_symbol)
{
case '+':
*result = first_operand + second_operand;
return TRUE;

```

Saturday, April 18, 2015

Data Structure

34

Evaluating a Postfix Expression in C++

```
case '-':
    *result = first_operand - second_operand;
    return TRUE;
case '*':
    *result = first_operand * second_operand;
    return TRUE;
case '/':
    if (0.0 == second_operand)
    {
        return FALSE;
    }
    else
    {
        *result = first_operand
        / second_operand;
        return TRUE;
    }
}
```

Saturday, April 18, 2015

Data Structure

35

Evaluating a Postfix Expression in C++

```
case '^':
    if (first_operand <= 0.0)
    {
        return FALSE;
    }
    else
    {
        *result = pow(first_operand,
        second_operand);
        return TRUE;
    }
}
```

Saturday, April 18, 2015

Data Structure

36

Evaluating a Postfix Expression in C++

```
default:  
return FALSE;  
} /* end switch */  
return FALSE; /* unreachable code */  
}
```