SWE 514 Computer Systems Project:

A86 Code Generator
Calculating Hexadecimal Expressions
And Converting Infix To Postfix

Boğaziçi University, Institute For Sciences Department Of Computer Engineering

Master of Science Program in Software Engineering

Yunus Sina Gülşen

Orkan Metin

Table of Contents

SWE 514 Computer Systems Project

In this Project, a program is implemented that generates A86 code for calculating the value of a hexadecimal expression using +, *, / operations and parenthesis.

Assumptions

All values and results of operations will fit into 16 bits.

All numbers are non-negative integers.

All values are written in hexadecimal format.

Output should be in hexadecimal format.

The expression may have parenthesis.

Infix To Postfix Conversion Using Stack

One of the applications of Stack is in the conversion of arithmetic expressions in high-level programming languages into machine-readable form. As our computer system can only understand and work on a binary language, it assumes that an arithmetic operation can take place in two operands only e.g., A+B, C*D,D/A etc. But in our usual form an arithmetic expression may consist of more than one operator and two operands e.g. (A+B)*C(D/(J+D)).

These complex arithmetic operations can be converted into polish notation using stacks which then can be executed in two operands and an operator form.

Infix Expression

It follows the scheme of <operand><operator><operand> i.e. an <operator> is preceded and succeeded by an <operand>. Such an expression is termed infix expression. E.g., A+B

Postfix Expression

It follows the scheme of <operand><operand><operator> i.e. an <operator> is succeeded by both the <operand>. E.g., AB+

Algorithm to Convert Infix To Postfix

Step 1: Scan the Infix expression from left to right for tokens (Operators, Operands & Parentheses) and perform the steps 2 to 5 for each token in the Expression.

Step 2: If token is operand, Append it in postfix expression.

Step 3: If token is a left parentheses "(", push it in stack.

Step 4: If token is an operator,

> Pop all the operators which are of higher or equal precedence then the incoming token and append them (in the same order) to the output Expression.
>
> After popping out all such operators, push the new token on stack.

Step 5: If ")" right parentheses is found,

> Pop all the operators from the Stack and append them to Output String, till you encounter the Opening Parenthesis "(".
>
> Pop the left parenthesis but don't append it to the output string (Postfix notation does not have brackets).

Step 6: When all tokens of Infix expression have been scanned. Pop all the elements form the stack and append them to the Output String.

The Output string is the Corresponding Postfix Notation.

Example;

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

Push "("onto Stack, and add ")" to the end of X.

Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.

If an operand is encountered, add it to Y.

If a left parenthesis is encountered, push it onto Stack.

If an operator is encountered ,then:

Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.

Add operator to Stack.

[End of If]

If a right parenthesis is encountered ,then:

Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.

Remove the left Parenthesis.

[End of If]

[End of If]

END.

Example;

Infix Expression: A+ (B*C-(D/E^F)*G)*H, where ^ is an exponential operator.

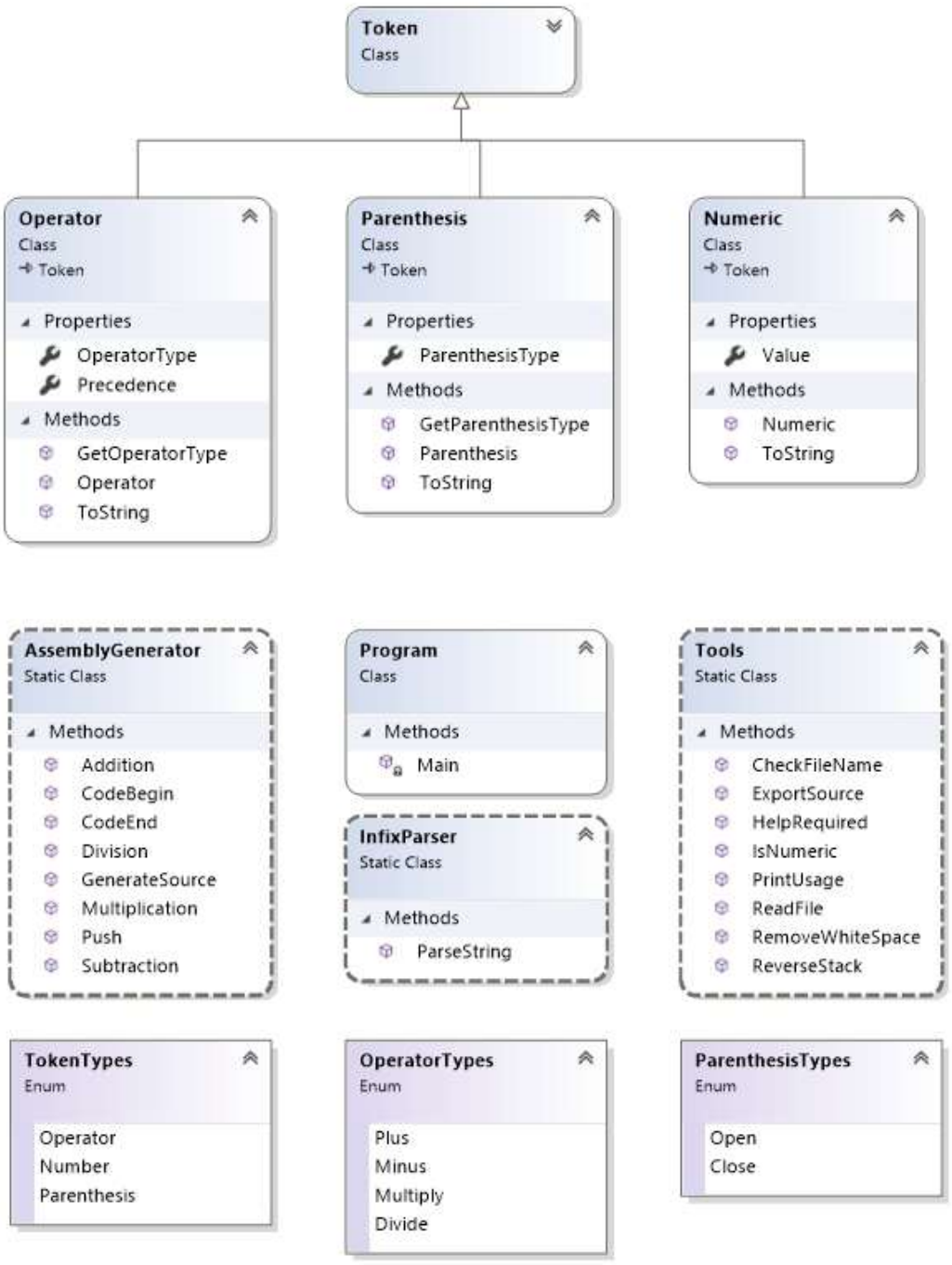
Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack , that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack , that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Resultant Postfix Expression: ABC*DEF^/G*-H*+

Advantage of Postfix Expression over Infix Expression

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

Class Diagram Of The Implementation



References:

Murugeshwari, T. (2011, March 04). Infix to postfix conversion. Retrieved from https://www.slideshare.net/Thenmurugeshwari/infix-to-postfix-conversion?qid=8e8c1c40-d2ed-4124-8d97-b522222bcb0a&v=&b=&from_search=3

Wolf, C. E., Professor Emerita. (n.d.). Infix to postfix conversion algorithm. Retrieved from http://csis.pace.edu/~wolf/CS122/infix-postfix.htm

Jain, A. (2017, June 14). Infix To Postfix Conversion Using Stack [with C program]. Retrieved from https://www.includehelp.com/c/infix-to-postfix-conversion-using-stack-with-c-program.aspx