# conversion from infix to prefix

I am preparing for an exam where i couldn't understand the convertion of infix notation to polish notation for the below expression:

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

Can any one tell step by step how the given expression will be converted to prefix?

data-structures

edited Feb 9 '12 at 14:26 — Eimantas — 36.4k 10 100 142

asked Dec 22 '09 at 14:59 — kar — 74 1 1 4

## 12 Answers

```
Algorithm ConvertInfixtoPrefix

Purpose: Convert an infix expression into a prefix expression. Begin
// Create operand and operator stacks as empty stacks.
Create OperandStack
Create OperatorStack

// While input expression still remains, read and process the next token.

while( not an empty input expression ) read next token from the input expression

    // Test if token is an operand or operator
    if ( token is an operand )
    // Push operand onto the operand stack.
        OperandStack.Push (token)
    endif

    // If it is a left parentheses or operator of higher precedence than the last, or the
stack is empty,
    else if ( token is '(' or OperatorStack.IsEmpty() or OperatorHierarchy(token) >
OperatorHierarchy(OperatorStack.Top()) )
    // push it to the operator stack
        OperatorStack.Push ( token )
    endif

    else if( token is ')' )
    // Continue to pop operator and operand stacks, building
    // prefix expressions until left parentheses is found.
    // Each prefix expression is push back onto the operand
    // stack as either a left or right operand for the next operator.
        while( OperatorStack.Top() not equal '(' )
            OperatorStack.Pop(operator)
            OperandStack.Pop(RightOperand)
            OperandStack.Pop(LeftOperand)
            operand = operator + LeftOperand + RightOperand
            OperandStack.Push(operand)
        endwhile

    // Pop the left parthenses from the operator stack.
    OperatorStack.Pop(operator)
    endif

    else if( operator hierarchy of token is less than or equal to hierarchy of top of the
operator stack )
    // Continue to pop operator and operand stack, building prefix
    // expressions until the stack is empty or until an operator at
    // the top of the operator stack has a lower hierarchy than that
    // of the token.
        while( !OperatorStack.IsEmpty() and OperatorHierarchy(token) lessThen Or Equal to
```

```
OperatorHierarchy(OperatorStack.Top()) )
            OperatorStack.Pop(operator)
            OperandStack.Pop(RightOperand)
            OperandStack.Pop(LeftOperand)
            operand = operator + LeftOperand + RightOperand
            OperandStack.Push(operand)
        endwhile
        // Push the lower precedence operator onto the stack
        OperatorStack.Push(token)
    endif
endwhile
// If the stack is not empty, continue to pop operator and operand stacks building
// prefix expressions until the operator stack is empty.
while( !OperatorStack.IsEmpty() ) OperatorStack.Pop(operator)
    OperandStack.Pop(RightOperand)
    OperandStack.Pop(LeftOperand)
    operand = operator + LeftOperand + RightOperand

    OperandStack.Push(operand)
endwhile

// Save the prefix expression at the top of the operand stack followed by popping // the
operand stack.

print OperandStack.Top()

OperandStack.Pop()

End
```

edited May 16 '15 at 8:13 — idmean — 7,660 5 21 50

answered Jan 9 '10 at 20:37 — rajdip — 249 1 2

---

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

Prefix notation (reverse polish has the operator last, it is unclear which one you meant, but the principle will be exactly the same):

1. `(/ f g)`
2. `(+ d e)`
3. `(- (+ d e) (/ f g))`
4. `(- a b)`
5. `(/ (- a b) c)`
6. `(* (/ (- a b) c) (- (+ d e) (/ f g)))`

edited May 16 '13 at 23:35

answered Dec 22 '09 at 19:24 — cjrh — 5,468 18 35

1 your prefix output seems to be incorrect. correct one should be `*/-abc-+de/fg` – Rohit Apr 17 '13 at 20:54

@Rohit: well spotted, thank you. – cjrh May 16 '13 at 23:35

---

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

step 1: `(a-b)/c*(d+e- /fg))`

step 2: `(a-b)/c*(+de - /fg)`

step 3: `(a-b)/c * -+de/fg`

Step 4: `-ab/c * -+de/fg`

step 5: `/-abc * -+de/fg`

step 6: `*/-abc-+de/fg`

This is prefix notation.

edited May 16 '12 at 7:00 — Nishant — 2,057 10 43 67

answered Mar 4 '12 at 14:44 — Rahul Kumar Sharma — 41 1

---

If there's something about what infix and prefix mean that you don't quite understand, I'd highly suggest you **reread that section of your textbook**. You aren't doing yourself any favors if you come out of this with the right answer for this one problem, but still don't understand the concept.

Algorithm-wise, its pretty darn simple. You just act like a computer yourself a bit. Start by puting parens around every calculation in the order it would be calculated. Then (again in order from first calculation to last) just move the operator in front of the expression on its left hand side. After that, you can simplify by removing parens.

answered Dec 22 '09 at 15:05

T.E.D.
**31k** 5 44 111

---

I saw this method on youtube hence posting here.

given infix expression : (a–b)/c*(d + e – f / g)

reverse it :

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

read characters from left to right.
maintain one stack for operators

```
  1. if character is operand add operand to the output
  2. else if character is operator or )
    2.1 while operator on top of the stack has lower or **equal** precedence than this
 character pop
    2.2 add the popped character to the output.
    push the character on stack

  3. else if character is parenthesis (
     3.1 [ same as 2 till you encounter ) . pop ) as well
  4. // no element left to read
    4.1 pop operators from stack till it is not empty
    4.2 add them to the output.

 reverse the output and print.
```

credits : youtube

answered Apr 17 '13 at 20:58

Rohit
**521** 6 23

---

(a–b)/c*(d + e – f / g)

remember scanning the expression from leftmost to right most start on parenthesized terms follow the WHICH COMES FIRST rule... *, /, % are on the same level and higher than + and -.... so (a-b) = -bc prefix (a-b) = bc- for postfix another parenthesized term: (d + e - f / g) = do move the / first then plus '+' comes first before minus sigh '-' (remember they are on the same level..) (d + e - f / g) move / first (d + e - (/fg)) = prefix (d + e - (fg/)) = postfix followed by + then - ((+de) - (/fg)) = prefix ((de+) - (fg/)) = postfix

(-(+de)(/fg)) = prefix so the new expression is now -+de/fg (1) ((de+)(fg/)-) = postfix so the new expression is now de+fg/- (2)

(a–b)/c* hence

1. (a-b)/c*(d + e – f / g) = -bc prefix [-ab]/c*[-+de/fg] ---> taken from (1) / c * do not move yet so '/' comes first before '*' because they on the same level, move '/' to the rightmost : /[-ab]c * [-+de/fg] then move '*' to the rightmost
   - / [-ab]c[-+de/fg] = remove the grouping symbols = */-abc-+de/fg --> Prefix
2. (a-b)/c*(d + e – f / g) = bc- for postfix [ab-]/c*[de+fg/-]---> taken from (2) so '/' comes first before '' *because they on the same level, move '/' to the leftmost: [ab-]*c[de+fg/-]/ then move '' *to the leftmost [ab-] c [de+fg/-]/* = remove the grouping symbols= a b - c d e + f g / - / * --> Postfix

answered Feb 7 '11 at 9:48

danny albay
**11** 1

---

here is an java implementation for convert infix to prefix and evaluate prefix expression (based on rajdip's algorithm)

```
import java.util.*;

public class Expression {
    private static int isp(char token){
        switch (token){
            case '*':
            case '/':
                return 2;
            case '+':
            case '-':
                return 1;
            default:
                return -1;
        }
    }
    private static double calculate(double oprnd1,double oprnd2,char oprt){
        switch (oprt){
            case '+':
                return oprnd1+oprnd2;
            case '*':
                return oprnd1*oprnd2;
            case '/':
                return oprnd1/oprnd2;
            case '-':
                return oprnd1-oprnd2;
            default:
                return 0;
        }
    }
    public static String infix2prefix(String infix) {
        Stack<String> OperandStack = new Stack<>();
        Stack<Character> OperatorStack = new Stack<>();
        for(char token:infix.toCharArray()){
            if ('a' <= token && token <= 'z')
                OperandStack.push(String.valueOf(token));
            else if (token == '(' || OperatorStack.isEmpty() || isp(token) >
isp(OperatorStack.peek()))
                OperatorStack.push(token);
            else if(token == ')'){
                while (OperatorStack.peek() != '(') {
                    Character operator = OperatorStack.pop();
                    String RightOperand = OperandStack.pop();
                    String LeftOperand = OperandStack.pop();
                    String operand = operator + LeftOperand + RightOperand;
                    OperandStack.push(operand);
                }
                OperatorStack.pop();
            }
            else if(isp(token) <= isp(OperatorStack.peek())){
                while (!OperatorStack.isEmpty() && isp(token)<= isp(OperatorStack.peek()))
{
                    Character operator = OperatorStack.pop();
                    String RightOperand = OperandStack.pop();
                    String LeftOperand = OperandStack.pop();
                    String operand = operator + LeftOperand + RightOperand;
                    OperandStack.push(operand);
                }
                OperatorStack.push(token);
            }
        }
        while (!OperatorStack.isEmpty()){
            Character operator = OperatorStack.pop();
            String RightOperand = OperandStack.pop();
            String LeftOperand = OperandStack.pop();
            String operand = operator + LeftOperand + RightOperand;
            OperandStack.push(operand);
        }
        return OperandStack.pop();
    }
    public static double evaluatePrefix(String prefix, Map values){
        Stack<Double> stack = new Stack<>();
        prefix = new StringBuffer(prefix).reverse().toString();
        for (char token:prefix.toCharArray()){
            if ('a'<=token && token <= 'z')
                stack.push((double) values.get(token));
            else {
                double oprnd1 = stack.pop();
                double oprnd2 = stack.pop();
                stack.push(calculate(oprnd1,oprnd2,token));
            }
        }
        return stack.pop();
    }
    public static void main(String[] args) {
        Map dictionary = new HashMap<>();
        dictionary.put('a',2.);
        dictionary.put('b',3.);
        dictionary.put('c',2.);
        dictionary.put('d',5.);
        dictionary.put('e',16.);
        dictionary.put('f',4.);
```

```
        dictionary.put('g',7.);
        String s = "((a+b)/(c-d)+e)*f-g";
        System.out.println(evaluatePrefix(infix2prefix(s),dictionary));
    }
}
```

answered May 12 at 12:19

Alireza Afzal ghaei
**55** 7

---

simple google search came up with this. Doubt anyone can explain this any simpler. But I guess after an edit, I can try to bring forward the concepts so that you can answer your own question.

Hint :

Study for exam, hard, you must. Predict you, grade get high, I do :D

Explanation :

It's all about the way operations are associated with operands. each notation type has its own rules. You just need to break down and remember these rules. If I told you I wrote (2*2)/3 as [* /] (2,2,3) all you need to do is learn how to turn the latter notation in the former notation.

My custom notation says that take the first two operands and multiple them, then the resulting operand should be divided by the third. Get it ? They are trying to teach you three things.

1. To become comfortable with different notations. The example I gave is what you will find in assembly languages. operands (which you act on) and operations (what you want to do to the operands).
2. Precedence rules in computing do not necessarily need to follow those found in mathematics.
3. Evaluation: How a machine perceives programs, and how it might order them at run-time.

edited Dec 22 '09 at 16:02

answered Dec 22 '09 at 15:01

Hassan Syed
**12.2k** 3 50 121

Link is dead :( – Anirudh Ramanathan Nov 4 '13 at 11:34

---

This algorithm will help you for better understanding .

Step 1. Push “)” onto STACK, and add “(“ to end of the A.

Step 2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty.

Step 3. If an operand is encountered add it to B.

Step 4. If a right parenthesis is encountered push it onto STACK.

Step 5. If an operator is encountered then: a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same or higher precedence than the operator. b. Add operator to STACK.

Step 6. If left parenthesis is encontered then a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encounterd) b. Remove the left parenthesis.

Step 7. Exit

answered Jun 15 '15 at 14:39

Sanjay Shiradwade
**1** 3

---

https://en.wikipedia.org/wiki/Shunting-yard_algorithm

> The shunting yard algorithm can also be applied to produce prefix notation (also known as polish notation). To do this one would simply start from the end of a string of tokens to be parsed and work backwards, reverse the output queue (therefore making the output queue an output stack), and flip the left and right parenthesis behavior (remembering that the now-left parenthesis behavior should pop until it finds a now-right parenthesis).

```
from collections import deque
def convertToPN(expression):
    precedence = {}
    precedence["*"] = precedence["/"] = 3
    precedence["+"] = precedence["-"] = 2
    precedence[")"] = 1

    stack  = []
    result = deque([])
    for token in expression[::-1]:
        if token == ')':
            stack.append(token)
        elif token == '(':
            while stack:
                t = stack.pop()
                if t == ")": break
                result.appendleft(t)
        elif token not in precedence:
            result.appendleft(token)
        else:
            # XXX: associativity should be considered here
            # https://en.wikipedia.org/wiki/Operator_associativity
            while stack and precedence[stack[-1]] > precedence[token]:
                result.appendleft(stack.pop())
            stack.append(token)

    while stack:
        result.appendleft(stack.pop())

    return list(result)

expression = "(a - b) / c * (d + e - f / g)".replace(" ", "")
convertToPN(expression)
```

step through:

```
step 1 : token ) ; stack:[ ) ]
result:[  ]
step 2 : token g ; stack:[ ) ]
result:[ g ]
step 3 : token / ; stack:[ ) / ]
result:[ g ]
step 4 : token f ; stack:[ ) / ]
result:[ f g ]
step 5 : token - ; stack:[ ) - ]
result:[ / f g ]
step 6 : token e ; stack:[ ) - ]
result:[ e / f g ]
step 7 : token + ; stack:[ ) - + ]
result:[ e / f g ]
step 8 : token d ; stack:[ ) - + ]
result:[ d e / f g ]
step 9 : token ( ; stack:[  ]
result:[ - + d e / f g ]
step 10 : token * ; stack:[ * ]
result:[ - + d e / f g ]
step 11 : token c ; stack:[ * ]
result:[ c - + d e / f g ]
step 12 : token / ; stack:[ * / ]
result:[ c - + d e / f g ]
step 13 : token ) ; stack:[ * / ) ]
result:[ c - + d e / f g ]
step 14 : token b ; stack:[ * / ) ]
result:[ b c - + d e / f g ]
step 15 : token - ; stack:[ * / ) - ]
result:[ b c - + d e / f g ]
step 16 : token a ; stack:[ * / ) - ]
result:[ a b c - + d e / f g ]
step 17 : token ( ; stack:[ * / ]
result:[ - a b c - + d e / f g ]

# the final while
step 18 : token ( ; stack:[  ]
result:[ * / - a b c - + d e / f g ]
```

edited Nov 27 '15 at 3:43

answered Nov 27 '15 at 3:30

Dyno Fu

**4,094** 2 19 38

---

Maybe you're talking about the Reverse Polish Notation? If yes you can find on wikipedia a very detailed step-to-step example for the conversion; if not I have no idea what you're asking :(

You might also want to read my answer to another question where I provided such an implementation: http://stackoverflow.com/questions/1933970/c-simple-operations-evaluation-class/1933976#1933976

edited Dec 22 '09 at 15:06

answered Dec 22 '09 at 15:01


Andreas Bonini
**22.6k** 19 99 139

Sort of. He's talking about the opposite. RPN is operators *after* values, prefix is operators *before* values. – Jeff Rupert Dec 22 '09 at 15:08

This is the algorithm using stack.
**Just follow these simple steps.**

1.Reverse the given infix expression.

2.Replace '(' with ')' and ')' with '(' in the reversed expression.

3.Now apply standard infix to postfix subroutine.

4.Reverse the founded postfix expression, this will give required prefix expression.

In case you find step 3 difficult consult http://scanftree.com/Data_Structure/infix-to-prefix where a worked out example is also given.

answered Jul 20 '14 at 14:12


user3694269
**1** 1