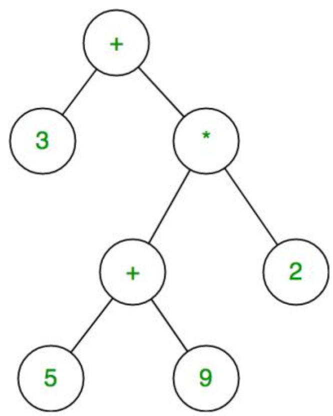
CS 260
Binary Parse Trees

Parse Trees

Parse Trees are binary trees, where each node has a parent and up to two children. They are used for parsing language and arithmetic expressions. This document focuses on parsing of arithmetic expressions.

Example Tree

In this example, the root contains the operator for addition, its left child is the number 3, its right child is the operator for multiplication, and so forth.



Expression Representation

Most people have been taught the use of infix notation. This is where the operator is written between the two operands and parentheses are used to indicate operations that are done out of the normal sequence. Other notations are postfix, where the operator follows its operands, and prefix, where the operator precedes its operands.

The expression shown in the above tree can be written in infix notation as **3 + (5 + 9) * 2**, or with full parentheses: (3 + ((5 + 9) * 2)). The same expression is written in postfix as **3 5 9 + 2 * +** and in prefix as: **+ 3 * + 5 9 2**.

Expression Evaluation

Most people have learned to evaluate an infix expression using the following rules.

1. Parentheses
2. Exponents
3. Multiplication/Division
4. Addition/Subtraction

and then evaluate the result left to right.

Computers normally use postfix notation because it does not require any knowledge of precedence of operators, use of parentheses, and is simply to evaluate using a stack. Postfix is also known as Reverse Polish Notation or RPN. Learning to read and understand it is a good skill for people who work with computers.

This work by Jim Bailey is licensed under a Creative Commons Attribution 4.0 International License.

The rules are simple:

1. Read the expression, one token at a time
    a. If operand, push on stack
    b. If operator, pop right side, pop left side, perform operation, push result on stack
2. When done reading expression, pop the result from stack

The evaluation of a postfix expression (3 5 9 + 2 * +) is illustrated in the following steps:

	Expression	Action	Stack
1	3 5 9 + 2 * +	Read 3 and push on stack	3
2	5 9 + 2 * +	Read 5 and push on stack	3, 5
3	9 + 2 * +	Read 9 and push on stack	3, 5, 9
4	+ 2 * +	Read +, pop 9, pop 5, push 5 + 9 on stack	3, 14
5	2 * +	Read 2 and push on stack	3, 14, 2
6	* +	Read *, pop 2, pop 14, push 14 * 2 on stack	3, 28
7	+	Read +, pop 28, pop 3, push 3 + 28 on stack	31
8		End of expression, pop 31 and return	

Displaying a Parse Tree

The same traversal methods are used to display the contents of a Binary Parse Tree as for any other binary tree. A prefix traversal yields the tree's expression in prefix notation, a postfix traversal yields it in postfix notation, and an infix traversal yields it in infix. For infix, it is necessary to modify the traversal code to add parentheses around each operator and its operands.

Parsing a postfix expression to build a Parse Tree

Since RPN (postfix) is so easy to use when evaluating an expression, it is also easy to use when creating a parse tree. The advantage of creating such a tree is that it provides an easy way to store an expression and then retrieve it in any of the three forms (postfix, prefix, or infix).

Instead of having a stack that holds operands and intermediate results, use a stack that holds the root node of a tree. Also realize that adding a node to such a stack is equivalent to adding the node and its entire subtree (since the subtree is attached via the left and right variables of the root node). The basic steps for creating such a tree from a postfix expression are shown below.

Look at each character in the string in order

- If a space
    - do nothing
- If an operand
    - create a new node for the operand
    - push it on the stack
- If an operator
    - create a new node for the operator
    - pop the top of the stack and attach it on the right of the new node
    - pop the top of the stack and attach it on the left of the new node
    - push new node and its subtree on the stack
- If end of expression
    - pop the top of the stack and save the node as root of the tree

This work by Jim Bailey is licensed under a Creative Commons Attribution 4.0 International License.

Pseudo Code for creating a ParseTree from a postfix expression

```
struct ParseNode
   char value
   ParseNode right
   ParseNode left

ParseNode doParse(string expression)

   // create a stack of ParseNodes
   stack theStack

   // Loop through the expression, getting each value in turn
   // if it is a space, ski
   // if it is a letter, push
   // if it is an operator, pop to right, pop to left, push
   for (i = 0; i < expression.length(); i++)

      char letter = expression[i]

      if (not isspace(letter))

         if (isOperand(letter))
            theStack.push(new ParseNode(letter))

         else
            ParseNode temp = new ParseNode(letter)

            Temp.right = theStack.top()
            theStack.pop()

            temp.left = theStack.top()
            theStack.pop()

            theStack.push(temp)

   return theStack.top()
```

This work by Jim Bailey is licensed under a Creative Commons Attribution 4.0 International License.

Parsing an infix expression to build a tree

The simplest way to parse an infix expression to build a tree is to convert it to postfix and then use the above algorithm. The algorithm for this conversion (assuming a string input and a string output) also uses a stack and is as shown below.

- Read the next character
  - if space
    - skip
  - If operand
    - add to output string
  - if L paren
    - push on the stack
  - if R paren
    - pop top item (continue until get L paren)
      - if not L paren, add to output string
      - else, discard and end popping
  - if operator
    - while stack not empty
      - pop top item
        - if operator with higher precedence, add to output string
        - else push back on stack, stop popping
          - it is either L paren
          - or operator of equal or lower precedence
      - push operator
- At end of expression
  - while the stack is not empty
    - pop and add to buffer

Consider this algorithm on the expression (A + B) / (C - D)

	Input	Action	Stack	Output String
1	(A+B)/(C-D)	L paren, push on stack	(	
2	A+B)/(C-D)	A, operand, output	(	A
3	+B)/(C-D)	+, operator, pop stack, (, push on stack	(, +	A
4	B)/(C-D)	B, operand, output	(,+	AB
5	)/(C-D)	R paren, pop adding to output until L paren		AB+
6	/(C-D)	/, operator, push on stack	/	AB+
7	(C-D)	L paren, push on stack	/,(	AB+
9	C-D)	C, operand, output	/,(	AB+C
10	-D)	-, operator, pop stack, (, push on stack	/,(,-	AB+C
11	D)	D, operand, output	/,(,-	AB+CD
12	)	R paren, pop adding to output until L paren	/	AB+CD-
13		End of expression, pop adding to output		AB+CD-/

Note that this algorithm assumes a fully parenthesized expression.

This work by Jim Bailey is licensed under a Creative Commons Attribution 4.0 International License.

Pseudo code for converting infix expression to postfix

```
string inOrder2PostOrder(string expression)
   stack theStack     // Stack of chars to store intermediate data
   string buffer = "" // String for building and returning result

   for (i = 0; i < static_cast<int>(expression.length()); i++)
      char value = expression[i]

      // add operands to buffer
      if (isOperand(value))
         buffer += value

      // push opening parens on stack
      else if (value == LPAREN)
         theStack.push(value)

      // if closing paren
      //   for each item on stack
      //     if opening paren, discard and quit
      //     else, add to buffer
      else if (value == RPAREN)
         bool isPopping = true

         while (!theStack.empty() and isPopping)
            char value = theStack.top()
            theStack.pop()

            if (value == LPAREN)
               isPopping = false
            else
               buffer += value

      // if an operator
      //   for each item on stack
      //     if operator with higher precedence, add item to buffer
      //     else push back on stack and quit
      //   push this value on stack
      else if (isOperator(value))
         bool isPopping = true

         while (!theStack.empty() and isPopping)
            char next = theStack.top()
            theStack.pop()

            if ( next == LPAREN )
               theStack.push(next)
               isPopping = false

            else
               if ( higherPrec(value, next) )
                  theStack.push(next)
                  isPopping = false

               else
                  buffer += next

          theStack.push(value)
```

This work by Jim Bailey is licensed under a Creative Commons Attribution 4.0 International License.

```
// get any remaining operators on stack
while (!theStack.empty())
   buffer += theStack.top()
   theStack.pop()

// return the result
return buffer
```

Traversing a Binary Parse Tree

When it is necessary to convert a Binary Parse Tree to a string for further processing or output, the three standard traversals are used. The algorithms for these are described in a separate document on Binary Tree Traversals.

The result of a pre-order traversal is to output the contents of the tree as a prefix expression. Similarly a post-order traversal provides a postfix expression and an in-order traversal an infix expression. Since prefix and postfix do not require any parentheses for proper processing the traversals do not require any special coding.

Infix expressions require parentheses to properly deal with operator precedence. For example the expression **A * (B + C)** without parentheses evaluates to **A * B + C** rather than the correct **A * B + A * C**. For this reason, it is necessary to modify the in-order traversal code to insert parentheses as the nodes are processed.

There are three levels of parentheses that can be generated. The first, and simplest to code, is a fully parenthesized expression where every subtree receives parentheses. For the above expression this would result in **((A) * ((B) + (C)))**. While this is mathematically correct, it is very cumbersome to look at.

The next level of sophistication is to only generate parentheses around subtrees that have an operator at the root. This results in **(A * (B + C))**. Again, this is mathematically correct, but remains more cumbersome, especially when more complex expressions are displayed.

The next step is to consider the precedence of the relative operators and only place parentheses around subtrees that have a root with a lower precedence than the parent node. This works fine for this expression resulting in **A * (B – C)**.

Unfortunately this solution does not deal well with the inverse operators of division and subtraction. As an example, consider the expression **A / (B * C)**. Since division and multiplication have the same precedence, the above algorithm would result in **A / B * C**, which is not mathematically the same. The solution to this problem is left as an exercise for the student.

This work by Jim Bailey is licensed under a Creative Commons Attribution 4.0 International License.