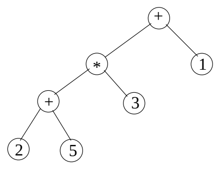
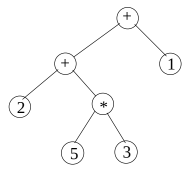
Binary Expression Trees

Binary expression trees capture the precedence and associativity of arithmetic operators:

(2 + 5) * 3 + 1



2 + 5 * 3 + 1



If we assume only binary operators, how do we algorithmically construct an expression tree given an *infix* expression?

Note that the evaluation of an expression tree uses a postorder traversal: the leftmost subtree is evaluated recursively then the rightmost subtree, then the operator node. If we traverse an expression tree and just print the element at each node, we get the following for the two examples shown above:

2 5 + 3 * 1 +
2 5 3 * + 1 +

The resulting expression are called *postfix* expressions. To convert an infix expression to a postfix expression or to evaluate a postfix expression, we need to use a stack.

Infix to Postfix Conversion

Converting an infix expression to a postfix expression uses a technique called *operator precedence parsing*, which is commonly used in the syntax parsing phase of a compiler. The technique is quite simple and consists of scanning an expression left to right, and using an operator stack.

To simplify the procedure, assume that the only binary operators we are interested in are: + - * /

Both + and - have the same precedence and * and / have the same precedence, which is greater than that for + and -. We also have to handle the fact that parenthesis override the regular precedence rules. We treat a left parenthesis as a higher precedence operator in the input, but a low precedence operator when on the operator stack.

1. start scanning the infix expression left to right, ignoring whitespace.
2. for each operand encountered, just output the value
3. If an operator is encountered on the input, we use an operator stack in the following manner:

Initially the operator stack has a special end-of-input marker on the top of the stack. If the operator on the top of the stack has a precedence less than the current operator, we just push the current operator onto the stack. If the operator on top has a higher or equal precedence, we pop it from the stack, send it to the output, and push the current operator onto the stack.

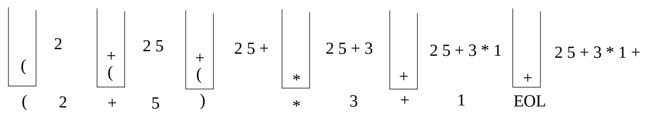
If we encounter a left paren, we push it on the stack. If we encounter a right parenthesis, we pop the stack and output all operators until we see a left paren. The left and right parenthesis symbols are discarded.

When the end-of-line marker in the input should be encountered only when the end-of-input marker is on the operator stack. This signifies the end of parsing. If not, the expression is not a proper expression.

Infix to Posfix Example

Infix expression: (2 + 5) * 3 + 1

The operator stack holds just the operators. Operands are sent to the output directly.



The conversion process rewrites the infix expression as a postfix expression, and send it to the standard output. That's nice, but not very useful.

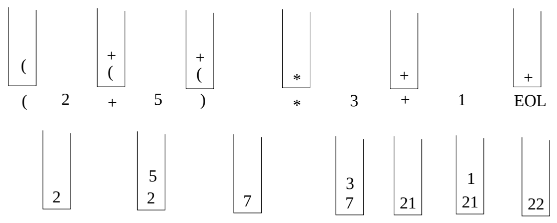
What if you wanted to actually evaluate the postfix expression as it was being constructed. How would you do this?

Instead of sending each operand to the output as it is encountered, you could have an operand stack on which you pushed each operand. When you pop an operator from the operator stack, you pop two operands from the operand stack and evaluate the operator using the two operands, then push the result back onto the operand stack. At the end of the parsing of the input expression, the final value on the operand stack is the value of the expression. This type of evaluator is called a *postfix machine*.

A Postfix Expression Evaluator

Infix expression: (2 + 5) * 3 + 1

The operator stack holds just the operators. Operands are pushed onto an operand stack.



Can the actions of a postfix evaluator be modified to allow us to construct a binary expression tree in a manner similar to the way the operand stack is manipulated?

Constructing A Binary Expression Tree

In the postfix machine, each time we see an operator we pop/push symbols onto the operator stack. Similarly, we push operands onto the operand stack, and when we pop an operator, we pop two operands from the operand stack and replace them with a new value.

A very similar process is used to construct a binary expression tree.

Each time we see an operator, do the same actions as we do for the operator precedence parsing algorithm. Each time we see an operand, we create a new BinaryNode object for the operand, and push a pointer to the node onto the operand stack.

When we pop an operator from the operator stack, we create a new BinaryNode for it.

When we go to pop two operands from the operand stack, we pop the first one and make it the right child of a operator node and pop the second one and make it the left child.

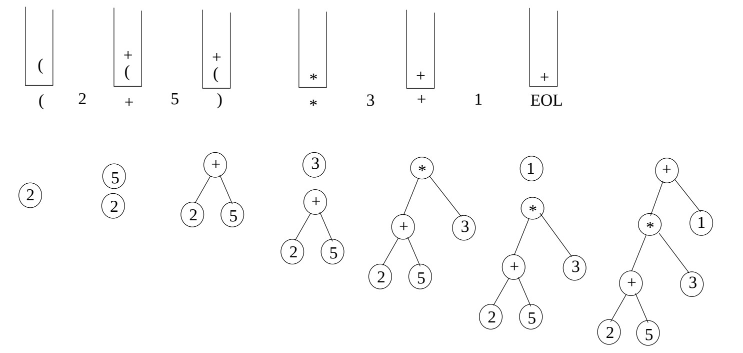
We then push the operator node pointer onto the operand stack.

At the end, the node object on the top of the operand stack is the root of the expression tree for the given infix expression.

Binary Expression Tree Example

Infix expression: (2 + 5) * 3 + 1

The operator stack holds just the operators. Operands are pushed onto an operand stack.



Expression Tree Evaluation

Once we have constructed the binary expression tree, to evaluate it, we just need to do a postorder traversal of the tree, and ask each node to evaluate itself. An operand node evaluates itself by just returning its value. An operator node has to apply the operator for that node to the result of evaluating its left subtree and its right subtree.

When we define our Binary Node class, we want to add a default virtual method for evaluating the node.

```
template<class T> class BinaryNode {
protected:
      T elem;
      BinaryNode *left, *right;
public:
      ...
      void setLeft(BinaryNode<T>* l) { left = l; }
      void setRight(BinaryNode<T>* r) { right = r; }
      virtual T eval() { return elem; }
};
```

We define the operand stack just using our stack template, but holding a BinaryNode<T> pointer:

```
Stack< BinaryNode<int>* > operands;
```

This default BinaryNode is fine when we create an operand node, since we just allocate a new BinaryNode<int>.

Question: What do the operator BinaryNode objects look like?

Expression Tree Evaluation

We can define derived node classes that inherit from BinaryNode<T> or each of the operators. We override the virtual eval method to provide the actual operator evaluation.

```
template<class T> class AddOperatorNode : public BinaryNode<T> {
public:
      virtual T eval() { return (left->eval() + right->eval()); }
};

template<class T> class SubOperatorNode : public BinaryNode<T> {
public:
      virtual T eval() { return (left->eval() - right->eval()); }
};

template<class T> class MultOperatorNode : public BinaryNode<T> {
public:
      virtual T eval() { return (left->eval() * right->eval()); }
};

template<class T> class DivOperatorNode : public BinaryNode<T> {
public:
      virtual T eval() { return (left->eval() / right->eval()); }
};
```

Since each operator node class for integer operators will be derived from BinaryNode<int>, there is no type incompatibility when pushing a new operator node object onto the operand stack.

```
BinaryNode<int> *addop = new AddOperatorNode();
addop->setRight(operands.pop());
addop->setLeft(operands.pop())
operands.push(addop);
```