Lecture 12

Simple Calculator

Simple Calculator

Evaluate:

- 1 + 2 * 3
- 10 - 4 - 3
- 2^3^3
- 4/2/2

We will use integer math for our examples

What about?

1 - 2 - 4 ^ 5 * 3 * 6 / 7 ^ 2 ^ 2

Postfix Notation

- ***infix*** notation -- Humans generally write expressions with the operators between the operands,

  as in 2 * 3 + 4
- post***fix*** notation -- notation in which the operators are put after their operands,

  as in 2 3 * 4 +

  This notation is preferred by computers

  – With "infix" (the method you are used to) :

    you put operators between operands: *a* + *b*
  – With "postfix" (the method computer prefers) :

    you put operators after operands: *a* *b* +

Postfix Notation

- Why Postfix Notation is preferred by the computer:
  - It is the most efficient method for representing arithmetic expressions
  - There is never any need to use ()'s with postfix notation and there is never any ambiguity
- To evaluate an infix expression, the compiler:
  - Converts the infix expression to postfix form
  - Evaluates the postfix expression

Application of stacks: Evaluating Postfix Expression

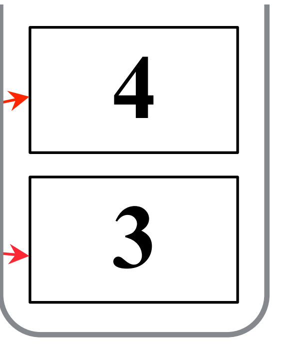
- Stacks are used by compilers to help in the process of evaluating expressions.
- Steps for evaluating postfix expressions

```
Make an empty stack s
For each token (operator * - + / or single digit integers 0, 1, …, 9) in the
postfix expression:
   if operand
     s.push(operand);
   if operator
     right =  s.top(); s.pop();
     left = s.top(); s.pop();
     push the value of the operator applied to the left and right
```

We use top and pop in pseudo code because we are writing the code in C++.

For the stack ADT, pop returns the value that is removed.

Example:
Evaluate *3 4 - 5 3 * -*

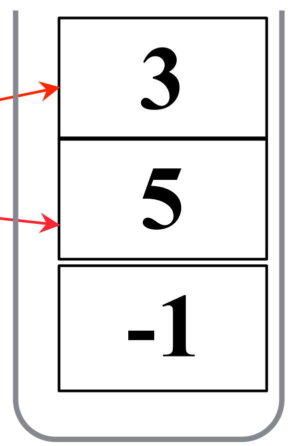



```
s.push(3);
s.push(4);
// found operator - so pop
right = s.top(); s.pop();
left = s.top(); s.pop();
s.push(left - right);
      // 3  -  4
```

- The stack now has one value `-1`
- The remainder of the expression: `5 3 * -`

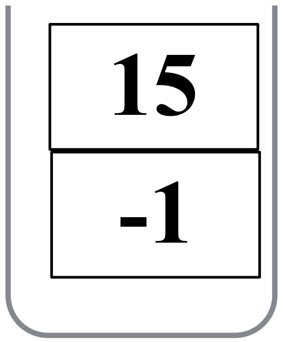
Continue with `5 3 * -`

```
s.push( 5 );
s.push( 3 );
right = s.top();s.pop();
left = s.top();s.pop();
s.push(left*right);
      // 5 * 3
```



- The Stack has 2 values
- Only one token remains

Continue with –




```
right = s.top();s.pop();  // found operator -
left = s.top();s.pop();
s.push(left-right);
        -1 - 15
```

- The expression has been processed.
- The value at the top of the stack is the value of the expression is -16
- Now evaluate `2 3 4 * 5 * -  ???`

Evaluate the Postfix expressions

1 2 3 * +

10 4 - 3 -

2 3 3 ^ ^

4 2 / 2 /

Running Time?

- Linear in the input size.

```
enum TokenType { EOL, VALUE, OPAREN, CPAREN, EXP,
                 MULT, DIV, PLUS, MINUS };
```

```
// PREC_TABLE matches order of Token enumeration
struct Precedence
{
    int inputSymbol;
    int topOfStack;
};
vector<Precedence> PREC_TABLE =
{
    { 0, -1 }, { 0, 0 },       // EOL, VALUE
    { 100, 0 }, { 0, 99 },     // OPAREN, CPAREN
    { 6, 5 },                  // EXP
    { 3, 4 }, { 3, 4 },        // MULT, DIV
    { 1, 2 }, { 1, 2 }         // PLUS, MINUS
};
```

Converting Infix Expressions to Equivalent Postfix Expressions

- An infix expression can be evaluated by first being converted into an equivalent postfix expression, and then the postfix version of the expression is evaluated
- Facts about converting from infix to postfix
  - Operands always stay in the same order with respect to one another
  - An operator will move only "to the right" with respect to the operands
  - All parentheses are removed

Converting Infix to Postfix

– `e.g. 1 + 2 * 3 ^ 4`

– in postfix `1 2 3 4 ^ * +`

Note: `^` *is a symbol in some languages for exponentiation*

– Operators are in reverse order in this example

- So we need to store them on a stack
- When an operator is encountered, pop higher order operators before pushing the lower order operator

Create the Postfix expression

- 1 + 2 * 3 -> 1 2 3 * +
- 10 - 4 - 3 -> 10 4 - 3 -
- 2^3^3 -> 2 3 3 ^ ^
- 4/2/2 -> 4 2 / 2 /

- Associativity
  - –Left associative: e.g. + , - ,* , /
  - –Right associative: e.g. ^ (exponential)
- Left-associative: Input + is lower than stack +

2 + 3 + 4 ⟶ 2 3 + 4 +

- Right-associative: Input ^ is higher than stack ^

2 ^ 3 ^ 4 ⟶ 2 3 4 ^ ^

Conversion Algorithm

- Algorithm:
  - Read infix expression as input
  - If input is operand, output the operand
  - If input is an operator +, -, *, /, then

    while (top of stack is an operator with greater precedence than the input operator)

      pop and output operator on top of stack

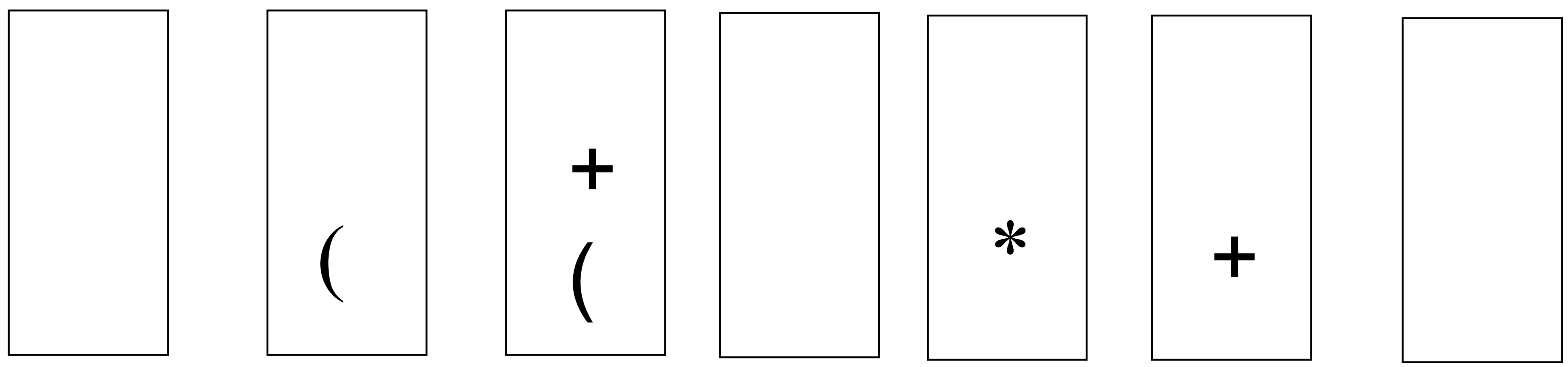
    push the input operator
  - If input is (, then push
  - If input is), then pop and output all operators until see a (on the stack. Pop the (without output
  - If no more input then pop and output all operators on stack

Running Time?

Example: $(2 + 4)*8 + 6$

- stacks

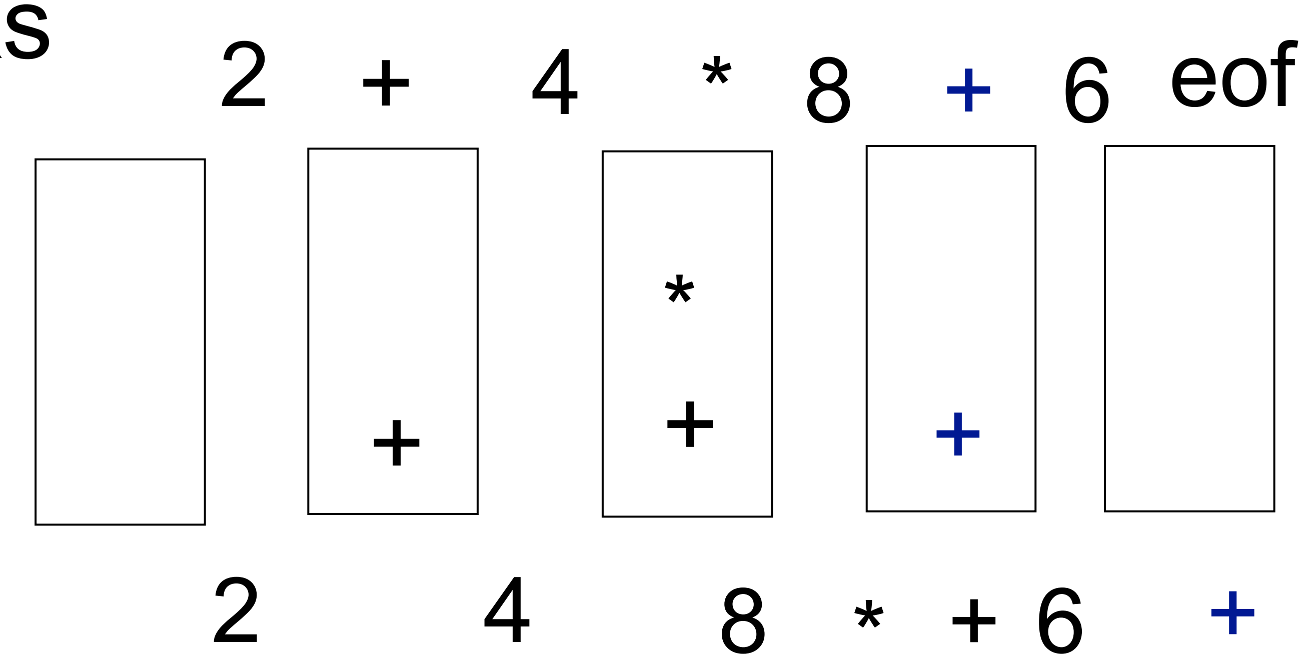
(2 + 4) * 8 + 6 eof



2 4 + 8 * 6 +

Example: $2 + 4*8 + 6$

- stacks



Conversion example

- Infix: 1 – 2 ^ 3 ^ 3 – (4 + 5 * 6) *7
- Postfix: 1 2 3 3 ^ ^ - 4 5 6 * + 7 * -
- -134217965

Enumerated types

using symbols instead of numbers for constant values

improves the readability of your code

- Simplest way to create your *own* type
  - you declare an enumerated type by using the **enum** keyword
  - you list all the values (the values are called *enumerators*) the type can hold:

enum Seasons { Winter, Spring, Summer, Fall };

0 1 2 3

  - every enumerator is assigned an integer value, either explicitly or by default.

A collection of named integer constants

```
#define EOL 0
#define VALUE 1
#define OPAREN 2
```

enum:

```
enum TokenType { EOL, VALUE, OPAREN, CPAREN, EXP, MULT, DIV, PLUS, MINUS};
```

It is possible to create explicit values:

```
enum seasons_t {spring = 10, summer = 100, fall = 50, winter = 5};

enum months_t {January = 1, February, March, April};
```

An alternative to if for multi-way branching if the condition being tested is equality for an integral type

```
enum Months { January = 1, February, March, April, May};
Months month = January;

switch (month)//expression must evaluate to an integral type
{
 case January:
   cout << "First month of the year!\n:";
 case February:
 case March:
   cout << "It is cold this month!\n";
   break;

 case April:
   cout << "Spring\n";
 default:
   cout << "One third of the year is over.\n";
   break;
}
```

Implementation

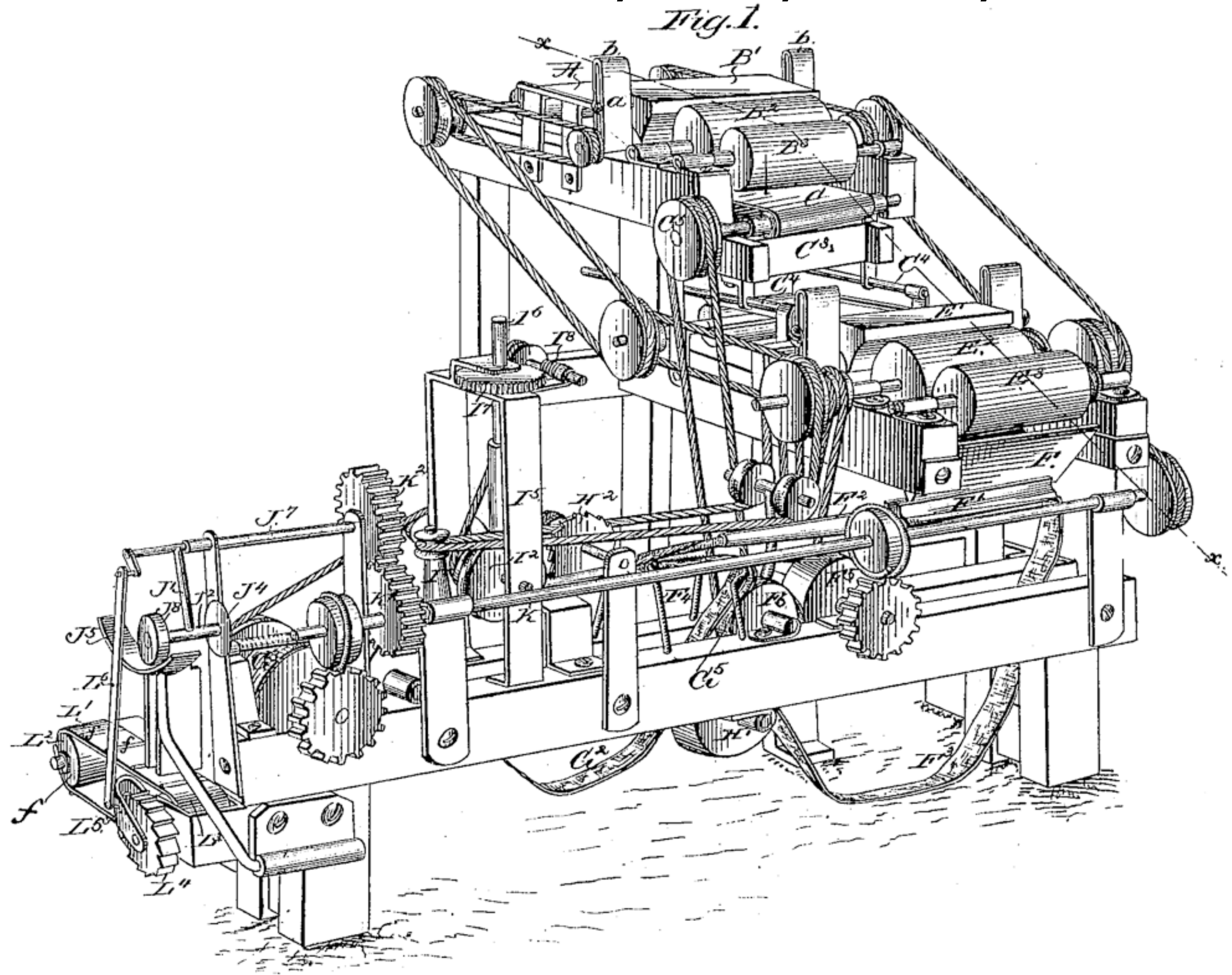
- Generate the tokens
  - Lexical Analysis - process of recognizing tokens in a stream of input.
  - Tokenizer - the program that does the lexical analysis of converting the input into tokens
  - Token - individual instance
- Evaluate the tokens
  - Template class called Evaluator
    - One stack to go from infix to postfix
    - One stack to evaluate a postfix experssion

Tokenizing

$$(2 + 4)*8 + 6$$

```
enum TokenType { EOL, VALUE, OPAREN, CPAREN, EXP,
                 MULT, DIV, PLUS, MINUS };
```

*)4 + 2(



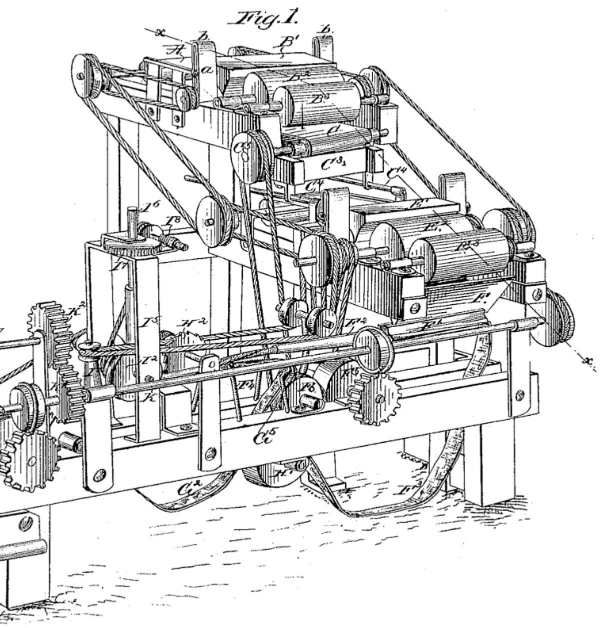
Machine picture from: http://upload.wikimedia.org/wikipedia/commons/c/ce/Bonsack_machine.png

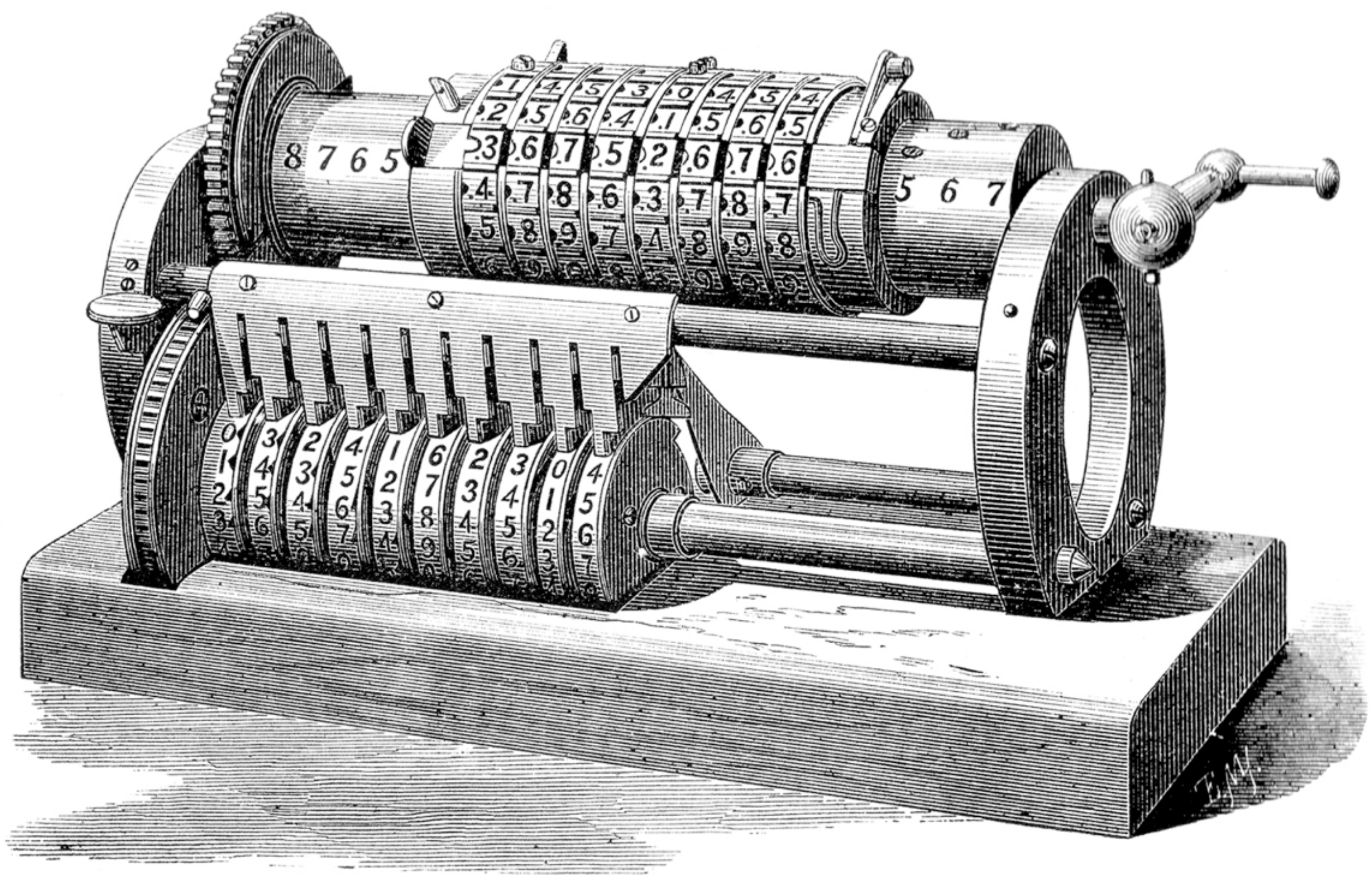
```
enum TokenType { EOL, VALUE, OPAREN, CPAREN, EXP, MULT, DIV, PLUS, MINUS };
```

Simple Calculator

(2 + 4)*8 + 6

Tokenizer





EOL

opStack

postFixStack

```
// PREC_TABLE
struct Precedence
{
    int inputSymbol;
    int topOfStack;
}
vector<Precedence> PREC_TABLE =
{
    { 0, -1 },    // EOL = 0
    { 0, 0 },     // VALUE = 1
    { 100, 0 }, // OPAREN =2
    { 0, 99 },   // CPAREN =3
    { 6, 5 },  // EXP = 4
    { 3, 4 },  // MULT =5
    { 3, 4 },  //  DIV =6
    { 1, 2 },  // PLUS = 7
    { 1, 2 }  // MINUS = 8
};
```

http://en.wikipedia.org/wiki/File:Grant_mechanical_calculating_machine_1877.jpg

$2 + 4$

VALUE,2 PLUS,0 VALUE,4

```
enum TokenType { EOL, VALUE, OPAREN, CPAREN, EXP, MULT, DIV, PLUS, MINUS };

template <class NumericType>
class Token
{
public:

    Token( TokenType tt = EOL, const NumericType & nt = 0 )
     : theType( tt ), theValue( nt ) { }

    TokenType getType( ) const{ return theType; }
    const NumericType & getValue( ) const{ return theValue; }

private:
    TokenType   theType;
    NumericType theValue;
};
```

notice the default values for the constructor!

VALUE,2 PLUS,0 VALUE,4

```
template <class NumericType>
class Tokenizer
{
public:
    Tokenizer( istream & is ): in( is ) { }

    Token<NumericType> getToken( );
```

Just one simple method!

```
private:
    istream & in;
};
```

where we get the input from

(12 + 4)

OPAREN,0 VALUE,12 PLUS,0 VALUE,4 CPAREN,0

```
// Find the next token, skipping blanks, and return it.
// Print error message if input is unrecognized.
template <class NumericType>
Token<NumericType> Tokenizer<NumericType>::getToken( )
{
    char ch;
    NumericType theValue;

    // Skip blanks
    while( in.get( ch ) && ch == ' ') ;

    if( in.good( ) && ch != '\n' && ch != '\0' ){
        switch( ch ){
            case '^': return EXP;
            case '/': return DIV;
            case '*': return MULT;
            case '(': return OPAREN;
            case ')': return CPAREN;
            case '+': return PLUS;
            case '-': return MINUS;
            default:
                in.putback( ch );
                if( !( in >> theValue ) ){
                    cerr << "Parse error" << endl;
                    return EOL;
                }
                return Token<NumericType>( VALUE, theValue );
        }
    }
    return EOL;
}
```

Simple Calculator

```
int main()
{
    string str;

    while( getline( cin, str ) )
    {
        Evaluator<int> e( str );
        cout << e.getValue( ) << endl;
    }
    return 0;
}
```

```
enum TokenType { EOL, VALUE, OPAREN, CPAREN, EXP, MULT, DIV, PLUS, MINUS };

template <class NumericType>
class Evaluator
{
public:
    Evaluator( const string & s ) : str( s )
                                 { opStack.push( EOL ); }

    // The only publicly visible routine
    NumericType getValue( );        // Do the evaluation

private:
    stack<TokenType>   opStack;      // Operator stack for conversion
    stack<NumericType> postFixStack; // Stack for postfix machine

    istringstream str;               // String stream

    // Internal routines
    NumericType getTop( );                                      // Get top of postfix stack
    void binaryOp( TokenType topOp );                           // Process an operator
    void processToken( const Token<NumericType> & lastToken );  // Handle LastToken
};
```

(12 + 4)

0,OPAREN | 12,VALUE | 0,PLUS | 4,VALUE | 0,CPARE

```
// Public routine that performs the evaluation.
// Examines the postfix machine to see if a single result
// is left and if so, returns it; otherwise prints error.
template <class NumericType>
NumericType Evaluator<NumericType>::getValue( )
{
    Tokenizer<NumericType> tok( str );
    Token<NumericType> lastToken;

    do {
        lastToken = tok.getToken( );
        processToken( lastToken );
    } while( lastToken.getType( ) != EOL );

    if( postFixStack.empty( ) )
    {
        cerr << "Missing operand!" << endl;
        return 0;
    }

    NumericType theResult = postFixStack.top( );
    postFixStack.pop( );
    if( !postFixStack.empty( ) )
        cerr << "Warning: missing operators!" << endl;

    return theResult;
}
```

Process each token till eof/eoln

Check if operand is missing

Output result check if operator is missing

```
// top and pop the postfix machine stack; return the result.
// If the stack is empty, print an error message.
template <class NumericType>
NumericType Evaluator<NumericType>::getTop( )
{
    if( postFixStack.empty( ) )
    {
        cerr << "Missing operand" << endl;
        return 0;
    }

    NumericType tmp = postFixStack.top( );
    postFixStack.pop( );
    return tmp;
}
```

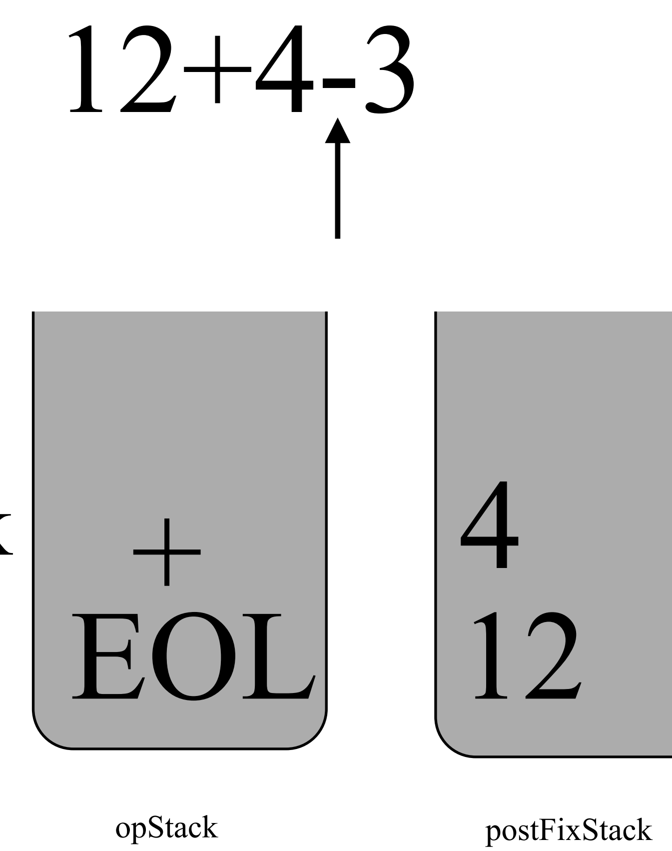
```
// Process an operator by taking two items off the postfix
// stack, applying the operator, and pushing the result.
// Print error if missing closing parenthesis or division by 0.
template <class NumericType>
void Evaluator<NumericType>::binaryOp( TokenType topOp ){

    if( topOp == OPAREN ){
        cerr << "Unbalanced parentheses" << endl;
        opStack.pop( );
        return;
    }
    NumericType rhs = getTop( );
    NumericType lhs = getTop( );

    if( topOp == EXP )
        postFixStack.push( pow( lhs, rhs ) );
    else if( topOp == PLUS )
        postFixStack.push( lhs + rhs );
    else if( topOp == MINUS )
        postFixStack.push( lhs - rhs );
    else if( topOp == MULT )
        postFixStack.push( lhs * rhs );
    else if( topOp == DIV )
        if( rhs != 0 )
            postFixStack.push( lhs / rhs );
        else {
            cerr << "Division by zero" << endl;
            postFixStack.push( lhs );
        }
    opStack.pop( );
}
```

Error check

top/pop operands

perform binary operation; put result on postFixStack

pop operator stack



0,OPAREN 12,VALUE 0,PLUS 4,VALUE AREN

```
// After token is read, use operator precedence parsing
// algorithm to process it; missing opening parentheses
// are detected here.
template <class NumericType>
void Evaluator<NumericType>::processToken(const Token<NumericType> & lastToken)
{
   TokenType topOp;
   TokenType lastType = lastToken.getType( );

   switch( lastType ){
     case VALUE:
          postFixStack.push( lastToken.getValue( ) );
          return;

     case CPAREN:
          while(  ( topOp = opStack.top( ) )  !=  OPAREN  &&  topOp != EOL  )
               binaryOp( topOp );
          if( topOp == OPAREN )
               opStack.pop( );  // Get rid of opening parentheseis
          else
               cerr << "Missing open parenthesis" << endl;
          break;

     default:    // General operator case
          while( PREC_TABLE[ lastType ].inputSymbol <=
                   PREC_TABLE[ topOp = opStack.top( ) ].topOfStack )
               binaryOp( topOp );
          if( lastType != EOL )
               opStack.push( lastType );
          break;
   }
}
```

Put operand on postFixStack

pop opStack and eval till "(' is found

pop operators with less precedence and eval

```
// PREC_TABLE
struct Precedence
{
   int inputSymbol;
   int topOfStack;
} ;
vector<Precedence =
{
   { 0, -1 },   // EOL = 0
   { 0, 0 },    //VALUE = 1
   { 100, 0 }, // OPAREN =
   { 0, 99 },  // CPAREN =
   { 6, 5 }, // EXP = 4
   { 3, 4 }, // MULT =5
   { 3, 4 }, //  DIV =6
   { 1, 2 }, // PLUS = 7
   { 1, 2 }  // MINUS = 8
};
```

Key Ideas

- Evaluator class
  - –Evaluates infix expression by “converting” to postfix expression and evaluating
  - –Does creation and evaluation of postfix expression in one step (evaluates as it creates)
- Precedence table
  - –Establishes precedence of operators
  - –Establishes whether operators are left or right associative

Expression Trees

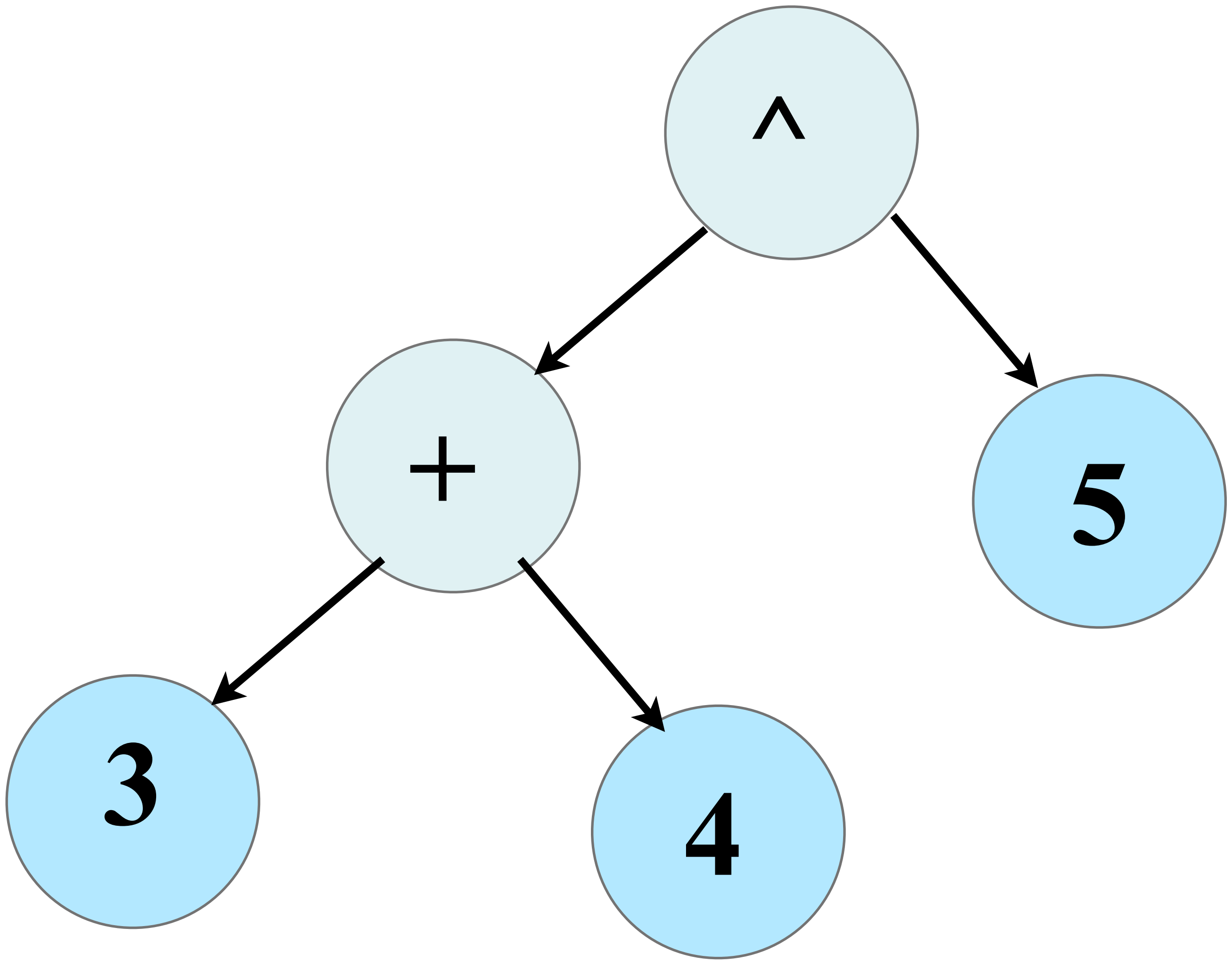
- leaves are operands
- other nodes are operators
- binary operators implies binary tree
- a node would have only one child if unary (e.g. -)
- evaluate by applying the operator at the root and recursively evaluating the left and right subtrees

Running Time?

- Every step involves a single push (and if it is an operator, two pops occur before the push.)
- E.g. 1 2 - 4 5 ^ 3 * 6 * 7 2 2 ^ ^ / -
- Has 9 operands and 8 operators, thus 17 steps and 17 pushes.
- Linear in the input size.

Expression Tree

$$(3 + 4)^5$$



Expression Trees

- Leaves contain operands (e.g., constants or variable names)
- Non-leaf nodes contain operators (e.g., ^, *, /, +, -)
- Nodes can have 1, 2, 3, or more children

Expression Tree

$3 + 4^\wedge 5$