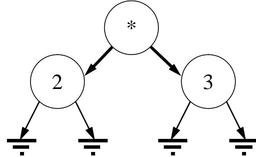
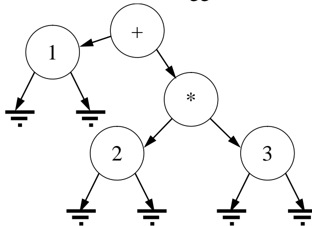
Example 2

```
1 + 2 * 3
```

- see 1, push 1 on number stack
- see +, push + on operator stack
- see 2, push 2 on operator stack
- see *, higher precedence than +, cannot add yet!
- see 3, push 3 on number stack
- no more input
  - “bind” operators on operator stack to values in num stack
    - pop 3 and 2 off number stack
    - pop * off operator stack
    - “bind” 2 * 3 to result in a small tree



    - push the small tree 2 * 3 on number stack
    - pop 2*3 and 1 off number stack
    - pop + off operator stack
    - bind 1 + 2*3 to result in a bigger tree



General Case: Shunting Yard Algorithm

```
1 + 2 * 3 - ( 4 - 5 ^ ( 6 ) )
```

Scan the equation left-to-right, for each element:

- if a number, push it on number stack
- if an operator, compare priority with one on top of stack
  - ***WHILE*** the top-of-stack operator is same or higher priority
    - stack operator should be done first
    - “bind” top 2 values in number stack to operator on stack
    - push the tree back on number stack
    - go to top of while, repeat as necessary
  - now stack operator is lower priority than new operator
    - push new operator on the operator stack
- if no more input
  - operators and numbers remain on stacks
  - top of operator stack has highest priority, bottom operator has lowest priority
  - while operator stack is not empty
    - bind top operator to top 2 numbers
    - replace top 2 numbers with one resulting tree

- what about parentheses ?
- relatively easy to handle...
  - (has lowest priority, put on stack as a marker
  -) has highest priority
    - when seen, bind everything on stack up to matching (
  - this replaces entire (...) expression with a tree (treated as a single number on the number stack)

- easy way to handle “no more input” case
  - place (...) around entire equation
  - when last) is seen, everything up to opening (is bound, i.e. the whole equation

Algorithm and Program Notes

- need to know if one operator has higher priority than another
  - write your own “priority” method

- cannot handle leading “-” sign for unary minus
  - pretend the “-” is part of the number, e.g. “-1”
  - use spaces to separate all numbers and operators

- store numbers and operators as a string
- convert to actual values at “last minute” as needed

- must put elements in
  - linked list (for a stack)
  - a binary tree (for expression tree)
  - need: next, left, right references
    - stack uses next reference
    - binary expression tree uses left, right references

Traversing a Tree

- 1 way to “walk” or traverse a linked list (visit all elements)
  - follow next!

- 3 ways to “walk a tree”
  - in order, preorder, postorder!
    - in order
      - self is in middle
      - visit left, then self, then right
      - see `toInfixString()` method in code
      - this is the approach taken to print an expression tree as a “normal” equation (must add parentheses to enforce proper ordering)
    - preorder
      - self is first
      - visit self, then left, then right
      - see `toPrefixString()` method in code
      - this is the order taken during a binary search (though you usually take only the left or the right branch, not both)
    - postorder
      - self is last
      - visit left, then right, then self
      - see `toPostfixString()` method in code
      - this is the approach taken to evaluate an expression tree (you must evaluate the left and right sides before combining them)

- left and right visits are actually recursive calls

See `Calc.java` source code!

```java
        if( opr == null ) {
            // oops, we ran out of equation terms and didn't find matching "("
            System.out.println( "Error, missing (");
            return;  // there was an error, quit early

        } else if( opr.val.equals("(") ) {
            // remove "(" from oprStack, its no longer needed
            oprStack.pop();
            return;  // there was no error, normal quit

        } else {
            // collapse top 2 subtrees on numStack and top operator
            // places resulting tree back on numStack
            bindTopOpr();
        }
    }
}


// store the new operator 'opr' on the stack because we haven't seen its
// right-hand argument yet.  before we do that however, we must check if
// the previous operator was higher priority; if so, it must bind to its
// two arguments now.
public static void pushNewOpr( Elem opr )
{
    Elem prevOpr = oprStack.peek();
    while( prevOpr != null
           && priority(prevOpr.val) >= priority(opr.val) )
    {
        // if prev opr was higher priority than this one,
        // collapse the previous expression into a tree
        bindTopOpr();
        prevOpr = oprStack.peek();
    }
    // now push our current operator on the oprStack
    oprStack.push( opr );
}


// parse equation, one term at a time.  (read comment at the top of this
// file to see what the input looks like).  the terms get pushed on stacks
// and "binding" is done when we can collapse a subexpression down into a
// tree.
public static void parseEquation( String equation )
{
    // start equation with an extra "("
    oprStack.push( new Elem("(") );

    while( ! equation.equals( "" ) ) {
        Elem firstPart = getFirstElem( equation );
        equation = removeFirstWord( equation );

        if( firstPart.val.equals(")") ) {
            // pop everying until the previous ( on the stack
            doBrackets();
        } else if( priority(firstPart.val) > 0 ) {
            // one of:   ^ * / + -
            pushNewOpr( firstPart );
        } else if( firstPart.val.equals("(") ) {
            // don't evaluate prev opr yet, wait till ) is seen
            oprStack.push( firstPart );
        } else {
            /* have a number */
            numStack.push( firstPart );
        }
    }

    // end equation with an extra ")"
    // this automatically does final cleanup for us :-)
    doBrackets();
}


// print the equation, as stored in the tree, 3 different ways.
// evaluate the equation and print the answer.
public static void printResult()
{
    Elem eqn = numStack.pop();

    System.out.println( "Printing infix version:"    );
    System.out.println( "\t" + eqn.toInfixString()   );

    System.out.println( "Printing prefix version:"   );
    System.out.println( "\t" + eqn.toPrefixString()  );

    System.out.println( "Printing postfix version:"  );
    System.out.println( "\t" + eqn.toPostfixString() );

    System.out.println( "Evaluating expression:"     );
    System.out.println( "\t" + eqn.eval()            );

    // sometimes errors leave things on the stacks,
    // so this forces the stacks to be emptied.
    while( ! numStack.isEmpty() )
        numStack.pop();
    while( ! oprStack.isEmpty() )
        oprStack.pop();
}


// repeatedly ask for a new equation, evaluate it, print answer.
// stop when the equation input is empty.
public static void main( String[] args )
{
    String input;
    String output;
    numStack = new Stack();
    oprStack = new Stack();

    while( true ) {
        // ask the user for an equation
        System.out.print("\nEnter equation:\n\t");
        input = Stdin.getString();

        input = input.trim();        // remove whitespace before and after
        if( input.equals("") )
            break;

        parseEquation( input );
        printResult();
    }
    System.out.println("Done.");
}

} // End of "class Calc"
```