Applications I

Agenda

Fun and games
- Word search puzzles
- Game playing

Stacks and compilers
- Checking for balanced symbols
- Operator precedence parsing
- Recursive descent parsing

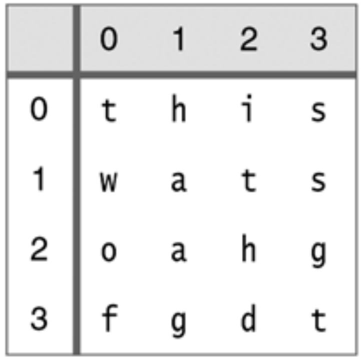
Word search puzzle

Problem: Given a two-dimensional array of characters and a list of words, find the words in the grid.

These words may be horizontal, vertical, or diagonal (for a total of 8 directions).

figure 10.1

A sample word search grid



	0	1	2	3
0	t	h	i	s
1	w	a	t	s
2	o	a	h	g
3	f	g	d	t

The grid contains the words: this, two, fat, and that.

Solution algorithms

An inefficient algorithm:

```
for each word W in the word list
    for each row R
        for each column C
            for each direction D
                check if W exists at row R, column C
                                  in direction D
```

Suppose $R = C = 32$, and $W = 40{,}000$
Number of string comparisons:
$W*R*C*8 = 40{,}000*32*32*8 =$ **327,680,000**

Improved algorithm:

```
for each row R
    for each column C
        for each direction D
            for each word length L
                check if L chars starting at row R
                      column C in direction D form a word
```

Suppose $R = C = 32$, $W = 40{,}000$, and $L_{max} = 20$
Maximum number of checks: $R*C*8*L_{max} = 32*32*8*20 = 163{,}840$

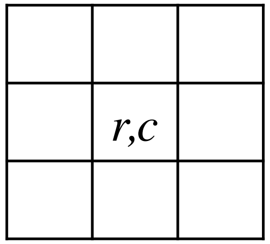
If the word list is sorted, we can use binary search and perform each check in roughly $\log_2 W$ string comparisons.
Total number of string comparisons ≈ 163,840*16 = **2,612,440**

For the example data, this algorithm is about 125 times faster than the previous one.

Further improved algorithm:

```
for each row R
    for each column C
        for each direction D
            for each word length L
                check if L chars starting at row R,
                      column C indirection D form a word
                if they do not form a prefix,
                    break;  // the innermost loop
```

Whether the *L* characters form a prefix may be determined by binary search.



Implementation in Java

```
int solvePuzzle() {
    int matches = 0;

    for (int r = 0; r < rows; r++)
        for (int c = 0; c < columns; c++)
            for (int rd = -1; rd <= 1; rd++)
                for (int cd = -1; cd <= 1; cd++)
                    if (rd != 0 || cd != 0)
                        matches += solveDirection(r, c, rd, cd);
    return matches;
}
```

```
int solveDirection(int r, int c, int rd, int cd) {
    int numMatches = 0;
    String prefix = "" + theBoard[r][c];

    for (int i = r + rd, j = c + cd;
          i >= 0 && j >= 0 && i < rows && j < columns;
          i += rd, j += cd) {
        prefix += theBoard[i][j];
        int index = prefixSearch(theWords, prefix);
        if (!theWords[index].startsWith(prefix))
            break;
        if (theWords[index].equals(prefix)) {
            numMatches++;
            System.out.println("Found " + prefix + " at " +
                                   r + " " + c + " to " +
                                   i + " " + j );
        }
    }
    return numMatches;
}
```

```
int prefixSearch(String[] a, String prefix) {
    int low = 0;
    int high = a.length - 1;

    while (low < high) {
        int mid = (low + high) / 2;
        if (a[mid].compareTo(prefix) < 0)
            low = mid + 1;
        else
            high = mid;
    }
    return low;
}
```

$\text{prefix} \le a[\text{low}] \wedge (\text{low} = 0 \vee \text{prefix} > a[\text{low} - 1])$

or

```
int prefixSearch(String[] a, String prefix) {
    int idx = Arrays.binarySearch(prefix);
    return idx >= 0 ? idx : -idx - 1;
}
```

Recall that the `binarySearch` method in the Collections API returns either the index of a match or the position of the smallest element that is at least as large as the target, plus 1 (as a negative number).

figure 10.2

The WordSearch class skeleton

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.IOException;

import java.util.Arrays;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;


// WordSearch class interface: solve word search puzzle
//
// CONSTRUCTION: with no initializer
// ******************PUBLIC OPERATIONS******************
// int solvePuzzle( )   --> Print all words found in the
//                          puzzle; return number of matches

public class WordSearch
{
    public WordSearch( ) throws IOException
      { /* Figure 10.3 */ }
    public int solvePuzzle( )
      { /* Figure 10.7 */ }

    private int rows;
    private int columns;
    private char theBoard[ ][ ];
    private String [ ] theWords;
    private BufferedReader puzzleStream;
    private BufferedReader wordStream;
    private BufferedReader in = new
                BufferedReader( new InputStreamReader( System.in ) );

    private static int prefixSearch( String [ ] a, String x )
      { /* Figure 10.8 */ }
    private BufferedReader openFile( String message )
      { /* Figure 10.4 */ }
    private void readWords( ) throws IOException
      { /* Figure 10.5 */ }
    private void readPuzzle( ) throws IOException
      { /* Figure 10.6 */ }
    private int solveDirection( int baseRow, int baseCol,
                                int rowDelta, int colDelta )
      { /* Figure 10.8 */ }
}
```

```java
/**
 * Constructor for WordSearch class.
 * Prompts for and reads puzzle and dictionary files.
 */
public WordSearch( ) throws IOException
{
    puzzleStream = openFile( "Enter puzzle file" );
    wordStream   = openFile( "Enter dictionary name" );
    System.out.println( "Reading files..." );
    readPuzzle( );
    readWords( );
}
```

figure 10.3

The `WordSearch` class constructor

```java
/**
 * Print a prompt and open a file.
 * Retry until open is successful.
 * Program exits if end of file is hit.
 */
private BufferedReader openFile( String message )
{
    String fileName = "";
    FileReader theFile;
    BufferedReader fileIn = null;

    do
    {
        System.out.println( message + ": " );

        try
        {
            fileName = in.readLine( );
            if( fileName == null )
                System.exit( 0 );
            theFile = new FileReader( fileName );
            fileIn  = new BufferedReader( theFile );
        }
        catch( IOException e )
          { System.err.println( "Cannot open " + fileName ); }
    } while( fileIn == null );

    System.out.println( "Opened " + fileName );
    return fileIn;
}
```

figure 10.4

The `openFile` routine for opening either the grid or word list file

```java
/**
 * Routine to read the dictionary.
 * Error message is printed if dictionary is not sorted.
 */
private void readWords( ) throws IOException
{
    List<String> words = new ArrayList<String>( );

    String lastWord = null;
    String thisWord;

    while( ( thisWord = wordStream.readLine( ) ) != null )
    {
        if( lastWord != null && thisWord.compareTo( lastWord ) < 0 )
        {
            System.err.println( "Dictionary is not sorted... skipping" );
            continue;
        }
        words.add( thisWord );
        lastWord = thisWord;
    }

    theWords = new String[ words.size( ) ];
    theWords = words.toArray( theWords );
}
```

figure 10.5

The `readWords` routine for reading the word list

```java
Arrays.sort(theWords);
```

```java
/**
 * Routine to read the grid.
 * Checks to ensure that the grid is rectangular.
 * Checks to make sure that capacity is not exceeded is omitted.
 */
private void readPuzzle( ) throws IOException
{
    String oneLine;
    List<String> puzzleLines = new ArrayList<String>( );

    if( ( oneLine = puzzleStream.readLine( ) ) == null )
        throw new IOException( "No lines in puzzle file" );

    columns = oneLine.length( );
    puzzleLines.add( oneLine );

    while( ( oneLine = puzzleStream.readLine( ) ) != null )
    {
        if( oneLine.length( ) != columns )
            System.err.println( "Puzzle is not rectangular; skipping row" );
        else
            puzzleLines.add( oneLine );
    }

    rows = puzzleLines.size( );
    theBoard = new char[ rows ][ columns ];

    int r = 0;
    for( String theLine : puzzleLines )
        theBoard[ r++ ] = theLine.toCharArray( );
}
```

figure 10.6

The `readPuzzle` routine for reading the grid

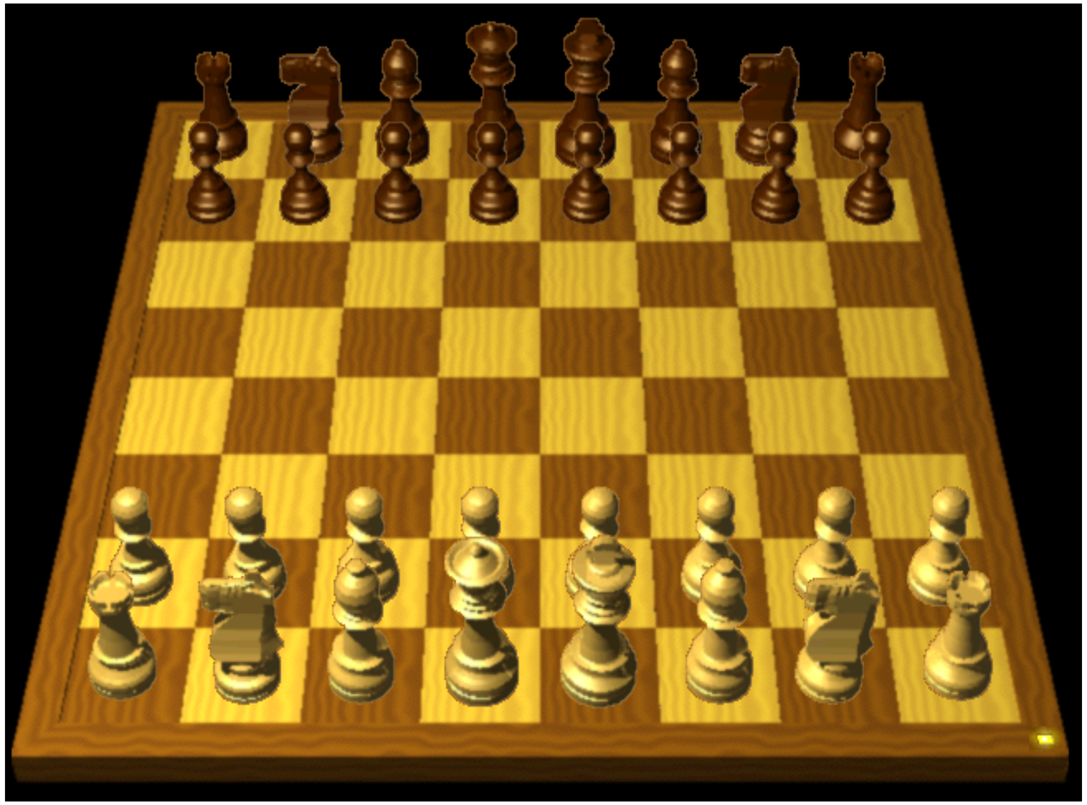
```java
char[][] theBoard;
```

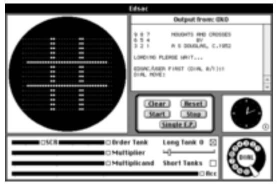
figure 10.9

A simple `main` routine for the word search puzzle problem

```
    // Cheap main
  public static void main( String [ ] args )
  {
      WordSearch p = null;

      try
      {
          p = new WordSearch( );
      }
      catch( IOException e )
      {
          System.out.println( "IO Error: " );
          e.printStackTrace( );
          return;
      }

      System.out.println( "Solving..." );
      p.solvePuzzle( );
  }
```

Games

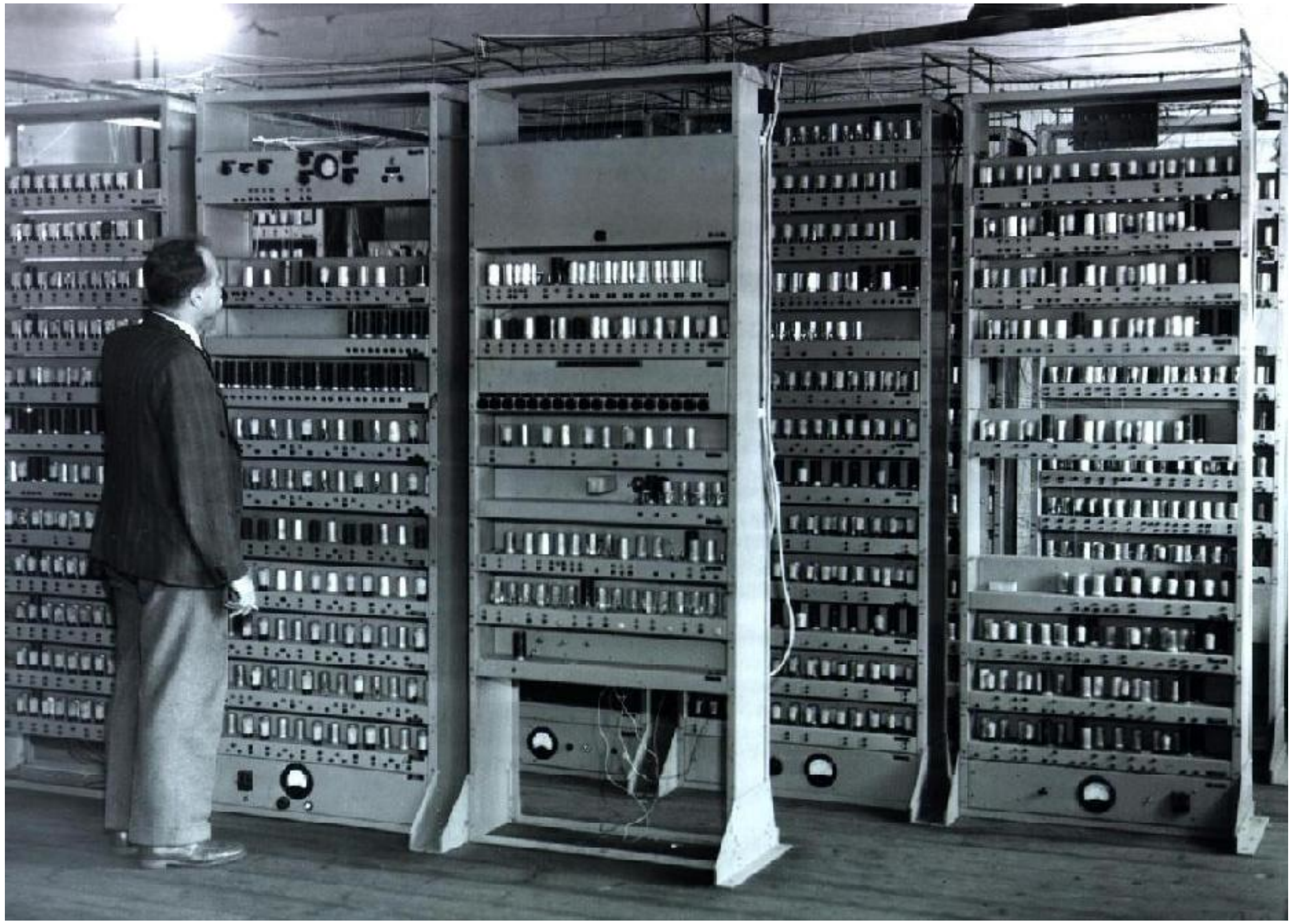




```
9 8 7        NOUGHTS AND CROSSES
6 5 4                BY
3 2 1        A S DOUGLAS, C.1952

LOADING PLEASE WAIT...

EDSAC/USER FIRST (DIAL 0/1):
DIAL MOVE:
```

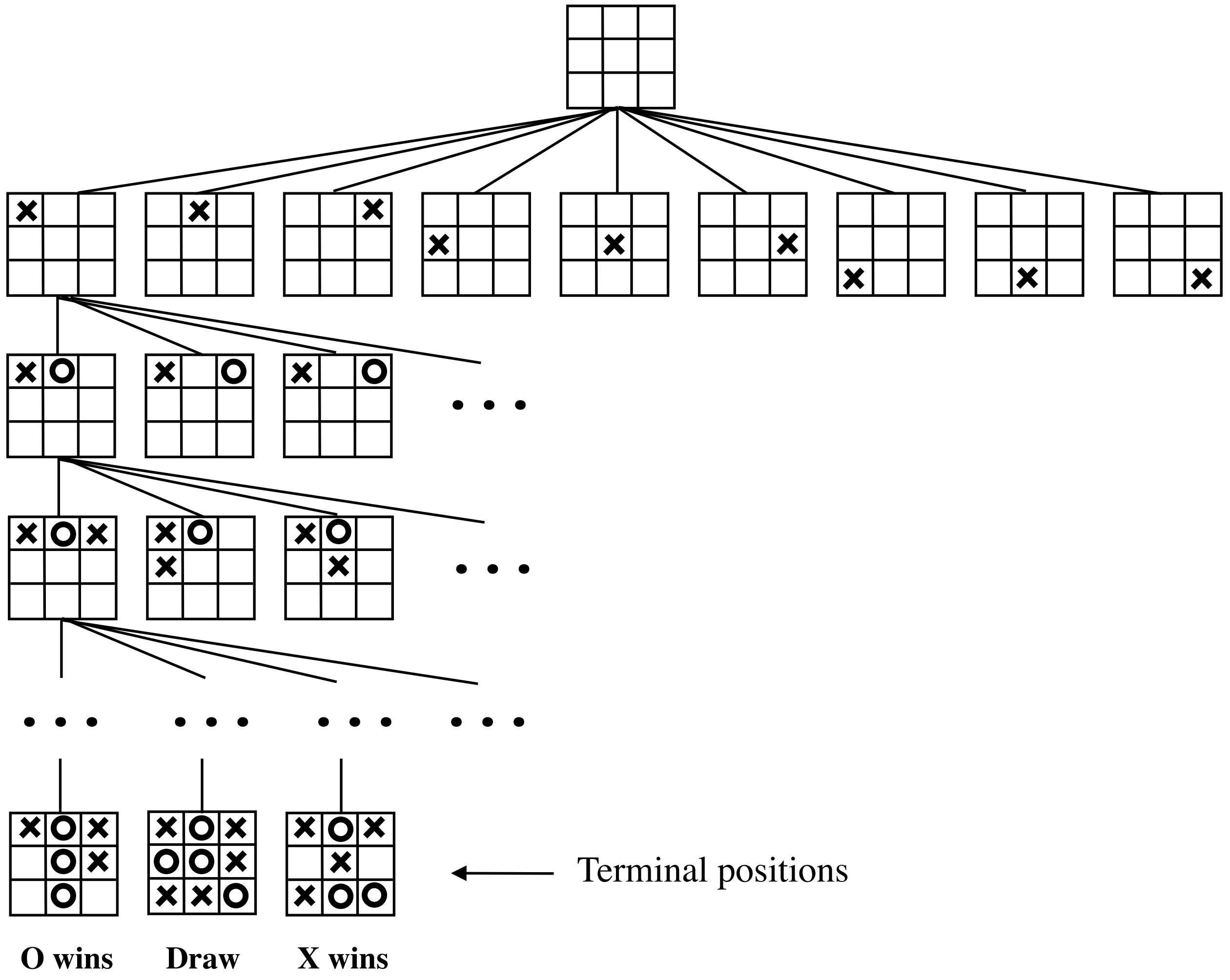
OXO for EDSAC, 1952

OXO was the first digital graphical game to run on a computer.



Electronic Delay Storage Automatic Calculator (EDSAC), 1949.
1024 locations, each containing 18 bits. One instruction per second.

The game of Tic-Tac-Toe



```java
public class TicTacToe {
    public static final int HUMAN        = 0;
    public static final int COMPUTER     = 1;
    public static final int EMPTY        = 2;

    public static final int HUMAN_WIN    = -1;
    public static final int DRAW         =  0;
    public static final int COMPUTER_WIN = +1;
    public static final int UNCLEAR      =  2;

    public TicTacToe() { clearBoard(); }

    public Best chooseMove(int side) { ... }
    public boolean playMove(int side, int row, int column) { ... }
    public void clearBoard() { ... }
    public boolean boardIsFull() { ... }
    public boolean isAWin(int side) { ... }

    private int[][] board = new int[3][3];
    private void place(int row, int column, int piece) { ... }
    private boolean squareIsEmpty(int row, int column) { ... }
    private int positionValue() { ... }
}
```

```
class Best {
    int row, column;
    int val;

    public Best(int v, int r, int c)
      { val = v; row = r; column = c; }

    public Best(int v)
      { this(v, 0, 0); }
}
```

```
Best chooseMove(int side)
```

The minimax strategy

1. A *terminal position* can immediately be evaluated, so if the position is terminal, return its value.

2. Otherwise, if it is the computer's turn to move, return the *maximum* value of all positions reachable by making one move. The reachable values are calculated recursively.

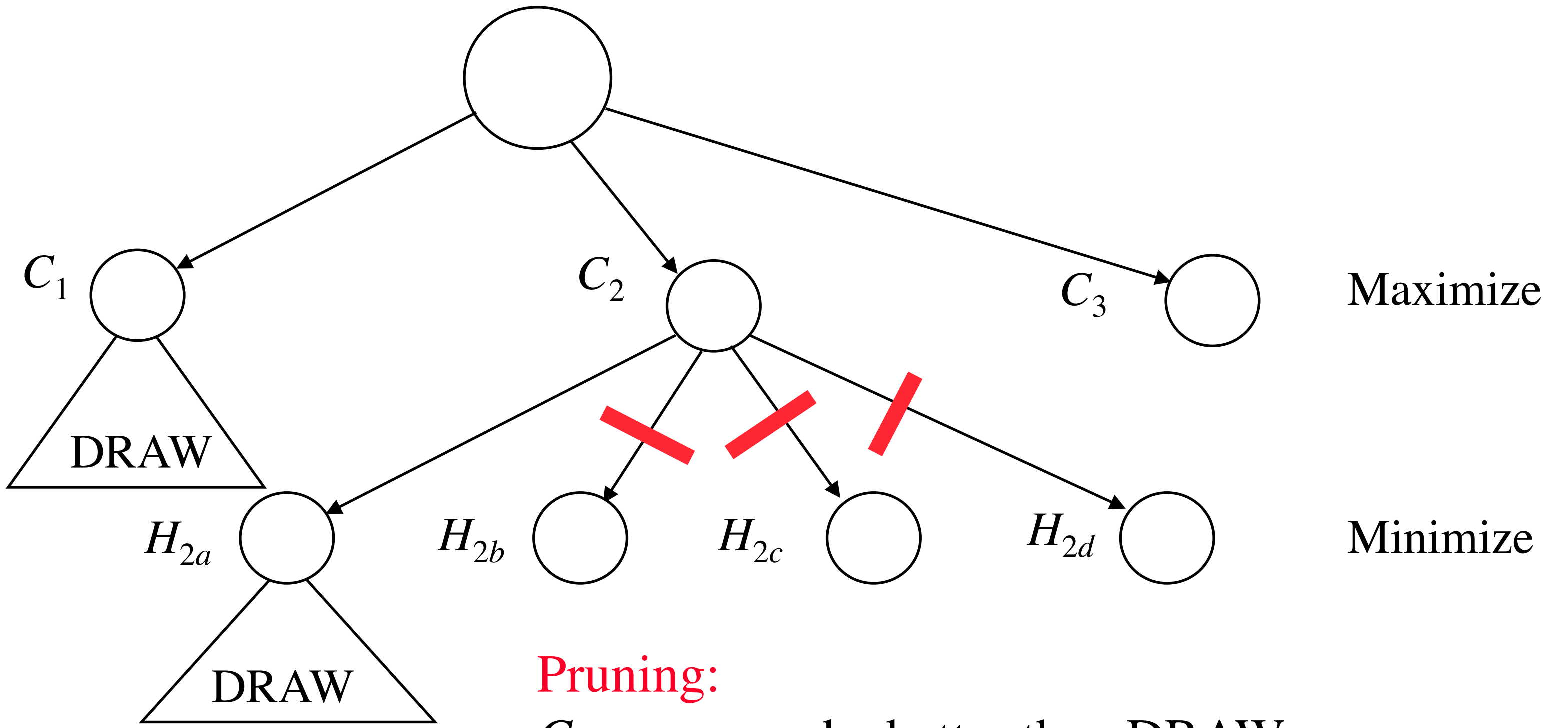
3. Otherwise, if it is the human player's turn to move, return the *minimum* value of all positions reachable by making one move. The reachable values are calculated recursively.

```
public Best chooseMove(int side) {
     int bestRow = 0, bestColumn = 0;
     int value, opp;

     if ((value = positionValue()) != UNCLEAR)
         return new Best(value);
     if (side == COMPUTER) { opp = HUMAN; value = HUMAN_WIN; }
     else { opp = COMPUTER; value = COMPUTER_WIN; }
     for (int row = 0; row < 3; row++)
         for (int column = 0; column < 3; column++)
             if (squareIsEmpty(row, column)) {
                 place(row, column, side);
                 Best reply = chooseMove(opp);
                 place(row, column, EMPTY);
                 if (side == COMPUTER && reply.val > value ||
                     side == HUMAN    && reply.val < value) {
                     value = reply.val;
                     bestRow = row; bestColumn = column;
                 }
             }
     return new Best(value, bestRow, bestColumn);
}
```

```
HUMAN_WIN = -1                COMPUTER_WIN = +1;
```

Minimax does more searching than necessary

Pruning



Pruning:
C_2 can never be better than DRAW

Alpha-beta pruning

Your enemy lost a bet and owes you one thing from a number of bags. You choose bag, but he chooses thing. Go through the bags one item at a time.

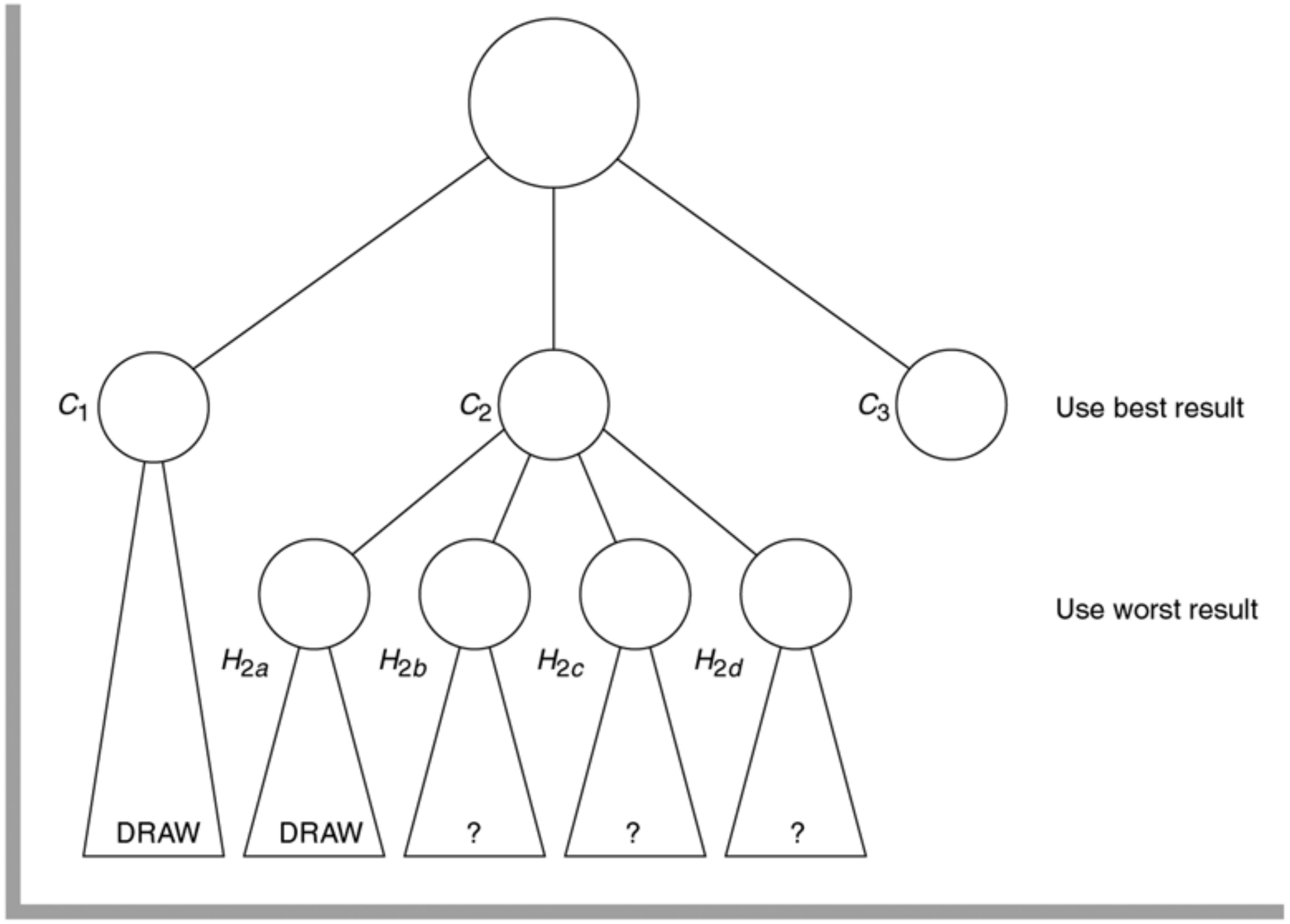
First bag: VM soccer tickets, sandwich, and $20
    He well choose the sandwich
Second bag: Dead fish, …
    He will choose dead fish. Doesn’t matter if the rest is a car and $50. You don’t need to look further in that bag.

Alpha-beta stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than or equal to a previously examined move.

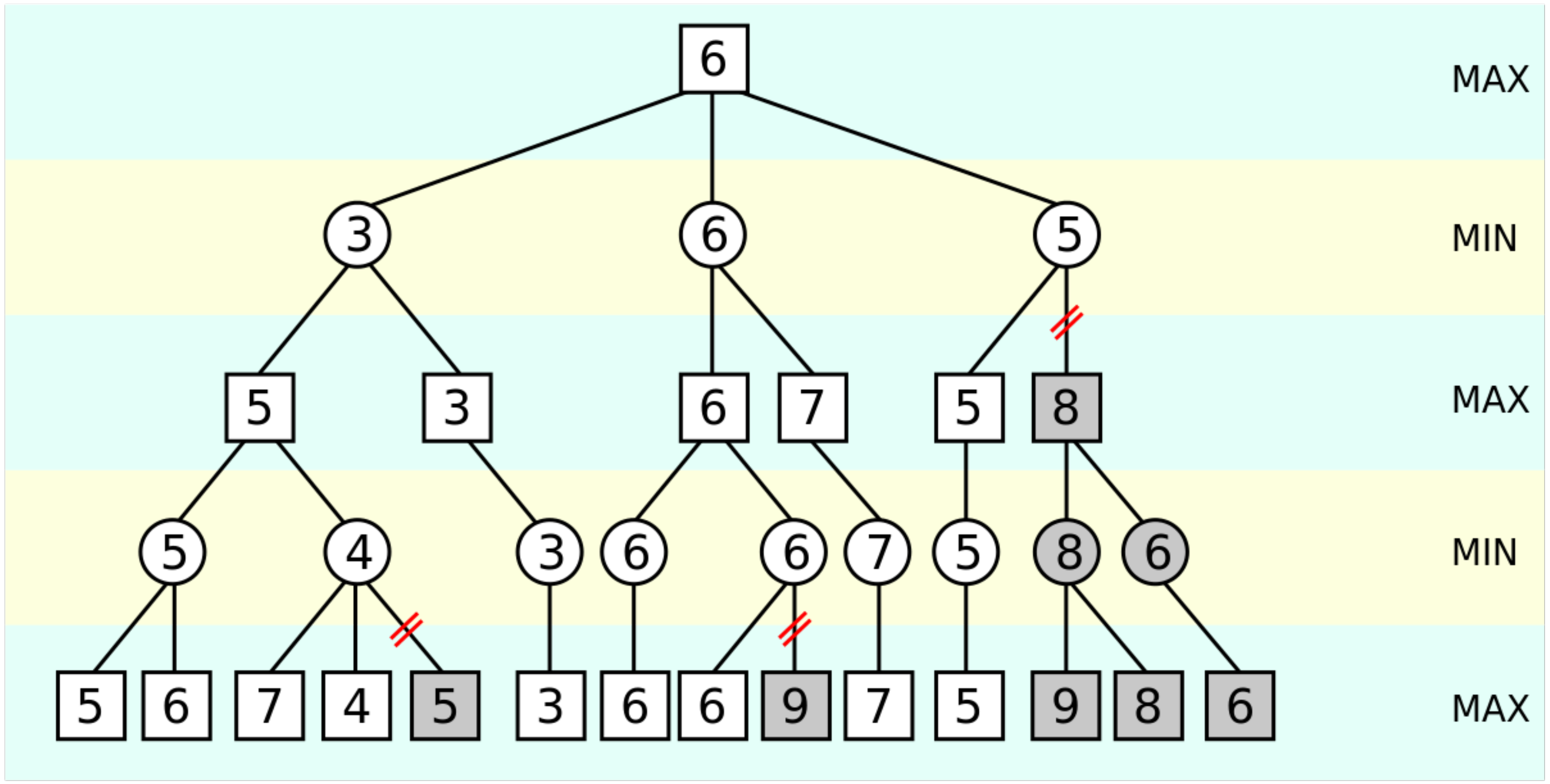
Alpha-beta pruning

figure 10.10

Alpha–beta pruning: After H_{2a} is evaluated, C_2, which is the minimum of the H_2's, is at best a draw. Consequently, it cannot be an improvement over C_2. We therefore do not need to evaluate H_{2b}, H_{2c}, and H_{2d} and can proceed directly to C_3.



Alpha-beta pruning example



Alpha-beta pruning

We say that the move H_{2a} is a *refutation* of the move C_2.
It proves that C_2 is not a better move than what already been seen.

alpha: The currently best value achieved by the computer (MAX)
beta: The currently best value achieved by the human player (MIN)

Prune

(1) when the human player achieves a value less than or equal to alpha.

(2) when the computer achieves a value greater than or equal to beta.

$\boxed{\text{Prune when alpha} \geq \text{beta}}$

refutation (en): *gendrivelse* (da)

```
public Best chooseMove(int side, int alpha, int beta) {
    int bestRow = 0, bestColumn = 0;
    int value, opp;
    if ((value = positionValue()) != UNCLEAR)
        return new Best(value);
    if (side == COMPUTER) { opp = HUMAN; value = alpha; }
    else { opp = COMPUTER; value = beta; }
Outer:
    for (int row = 0; row < 3; row++)
        for (int column = 0; column < 3; column++)
            if (squareIsEmpty(row, column)) {
                place(row, column, side);
                Best reply = chooseMove(opp, alpha, beta);
                place(row, column, EMPTY);
                if (side == COMPUTER && reply.val > value ||
                    side == HUMAN    && reply.val < value) {
                    value = reply.val;
                    if (side == COMPUTER) alpha = value;
                    else beta = value;
                    bestRow = row; bestColumn = column;
                    if (alpha >= beta)
                        break Outer;
                }
            }
    return new Best(value, bestRow, bestColumn);
}
```

Driver routine

```
Best chooseMove(int side) {
    return chooseMove(side, HUMAN_WIN, COMPUTER_WIN);
}
```

The effect of alpha-beta pruning

Alpha-beta pruning is most efficient if it searches the best move first.

In practice, alpha-beta pruning limits the searching to $O(\sqrt{N})$ nodes, where N is the number of nodes that would be examined without alpha-beta pruning.

Or, equivalently, the search can go twice as deep with the same amount of computation.

$$\sqrt{b^{2d}} = b^d$$

where b is the branching factor

Pruning by a transposition table

Avoid re-computations by saving evaluated positions in a table

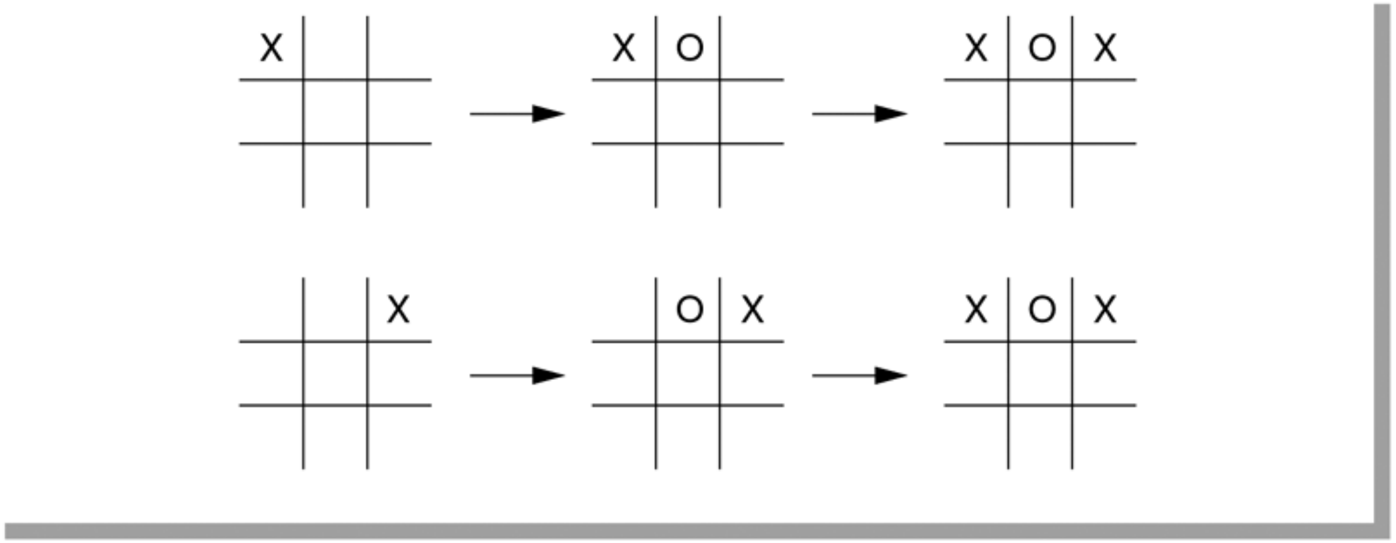


figure 10.12

Two searches that arrive at identical positions

Use a *transposition table*. Such a table is a hash table of each of the positions analyzed so far up to a certain depth. On encountering a new position, the program checks the table to see if the position has already been analyzed; this can be done quickly, in expected constant time

```
class Position {
    int[][] board;

    Position(int theBoard[][]) {
        board = new int[3][3];
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                board[i][j] = theBoard[i][j];
    }

    @override public boolean equals(Object rhs) {
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                if (board[i][j] != ((Position) rhs).board[i][j])
                    return false;
        return true;
    }

    @override public int hashCode() {
        int hashVal = 0;
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                hashVal = hashVal * 4 + board[i][j];
        return hashVal;
    }
}
```

```
private Map<Position,Integer> transpositions =
    new HashMap<Position,Integer>();
```

```
public Best chooseMove(int side, int alpha, int beta,
                       int depth) {
    int bestRow = 0, bestColumn = 0;
    int value, opp;
    Position thisPosition = new Position(board);

    if ((value = positionValue()) != UNCLEAR)
        return new Best(value);
    if (depth == 0)
        transpositions.clear();
    else if (depth >= 3 && depth <= 5) {
        Integer lookupVal = transpositions.get(thisPosition);
        if (lookupVal != null)
            return new Best(lookupVal);
    }
    ... chooseMove(opp, alpha, beta, depth + 1); ...
    if (depth >= 3 && depth <= 5)
        transpositions.put(thisPosition, value);
    return new Best(value, bestRow, bestColumn);
}
```

The effect of alpha-beta pruning and a transposition table for Tic-Tac-Toe

Alpha-beta pruning reduces the search from about 500,000 positions to about 18,000 positions.

The use of a transposition table removes about half of the 18,000 positions from consideration. The program's speed is almost doubled.

A general Java package for two-person game playing

by Keld Helsgaun

```
package twoPersonGame;
```

```
public abstract class Position {
    public boolean maxToMove;
    public abstract List<Position> successors();
    public abstract int value();
    public int alpha_beta(int alpha, int beta, int maxDepth) { ... };
    public Position bestSuccessor;
}
```

```
public int alpha_beta(int alpha, int beta, int maxDepth) {
    List<Position> successors;
    if (maxDepth <= 0 ||
        (successors = successors()) == null || successors.isEmpty())
        return value();
    for (Position successor : successors) {
        int value = successor.alpha_beta(alpha, beta, maxDepth - 1);
        if (maxToMove && value > alpha) {
            alpha = value;
            bestSuccessor = successor;
        } else if (!maxToMove && value < beta) {
            beta = value;
            bestSuccessor = successor;
        }
        if (alpha >= beta)
            break;
    }
    return maxToMove ? alpha : beta;
}
```

Reduction of code (negamax)

```
public int alpha_beta(int alpha, int beta, int maxDepth) {
    List<Position> successors;
    if (maxDepth <= 0 ||
        (successors = successors()) == null || successors.isEmpty())
        return (maxToMove ? 1 : -1) * value();
    for (Position successor : successors) {
        int value = -successor.alpha_beta(-beta, -alpha, maxDepth - 1);
        if (value > alpha) {
            alpha = value;
            bestSuccessor = successor;
        }
        if (alpha >= beta)
            break;
    }
    return alpha;
}
```

```java
import twoPersonGame.*;

public class TicTacToePosition extends Position {
    public TicTacToePosition(int row, int column,
                             TicTacToePosition predecessor) { ... }
    @Override public List<Position> successors() {
        List<Position> successors = new ArrayList<Position>();
        if (!isTerminal())
            for (int row = 0; row < 3; row++)
                for (int column = 0; column < 3; column++)
                    if (board[row][column] == '.')
                        successors.add(
                          new TicTacToePosition(row, column, this));
        return successors;
    }
    @Override public int value()
      { return isAWin('O') ? 1 : isAWin('X') ? -1 : 0; }
    public boolean boardIsFull() { ... }
    public boolean isAWin(char symbol) { ... }
    public boolean isTerminal() { ... }
    public void print() { ... }

    int row, column;
    char[][] board = new char[3][3];
}
```

Stacks and compilers

Balanced symbol-checker

Problem: Given a string containing parentheses, determine if for every left parenthesis there exists a matching right parenthesis.

For example the parentheses balance in "[()]", but not in "[(])".

In the following, we simplify the problem by assuming that the string only consists of parentheses.

Only one type of parenthesis

If there is only one type of parenthesis, e.g., ’(’ and ’)’, the solution is simple.

We can check the balance by means of a counter.

```
boolean balanced(String s) {
    int balance = 0;
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (c == '(')
            balance++;
        else if (c == ')') {
            balance--;
            if (balance < 0)
                return false;
        }
    }
    return balance == 0;
}
```

More than one type of parenthesis

However, if there is more than one type of parenthesis, the problem cannot be solved by means of counters.

However, we can check the balance by means of a stack:

```
1. Make an empty stack.
2. Read symbols until the end of the string.
   a. If the symbol is an opening symbol, push it onto the stack.
   b. If it is a closing symbol, do the following
        i. If the stack is empty, report an error.
       ii. Otherwise, pop the stack. If the symbol popped is
           not the corresponding opening symbol, report an error.
3. At the end of the string, if the stack is not empty, report an error.
```

Example

Symbols: ( ) [ ] { }

```
String s = "([]}"
```

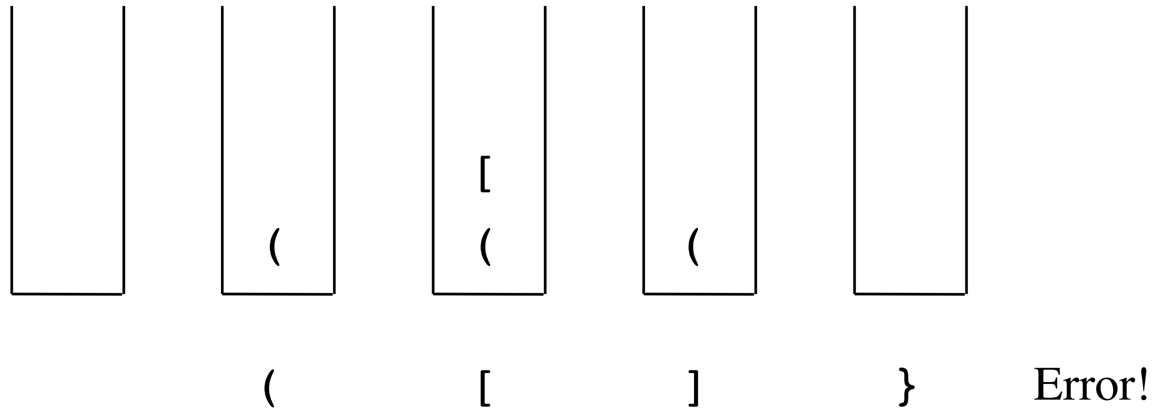
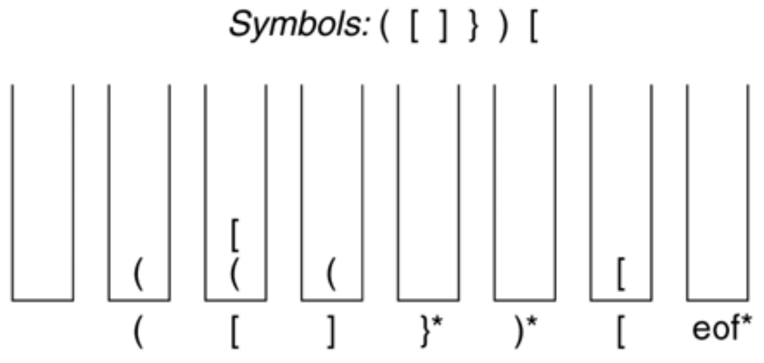


figure 11.1

Stack operations in a balanced-symbol algorithm



Errors (indicated by *):

} when expecting)

) with no matching opening symbol

[unmatched at end of input

Stack of characters

```
class CharStack {
    void push(char ch) { stack[++top] = ch; }
    char pop() { return stack[top--]; }
    boolean isEmpty() { return top == -1; }

    private char[] stack = new char[100];
    private int top = -1;
}
```

Java code

```java
boolean balanced(String s) {
    CharStack stack = new CharStack();
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (c == '(' || c == '[' || c == '{')
            stack.push(c);
        else if (stack.isEmpty() ||
            (c == ')' && stack.pop() != '(') ||
            (c == ']' && stack.pop() != '[') ||
            (c == '}' && stack.pop() != '{'))
            return false;
        }
    }
    return stack.isEmpty();
}
```

figure 11.2

The `Tokenizer` class skeleton, used to retrieve tokens from an input stream

```java
import java.io.Reader;
import java.io.PushbackReader;
import java.io.IOException;

// Tokenizer class.
//
// CONSTRUCTION: with a Reader object
// ******************PUBLIC OPERATIONS***********************
// char getNextOpenClose( ) --> Get next opening/closing symbol
// int getLineNumber( )     --> Return current line number
// int getErrorCount( )     --> Return number of parsing errors
// String getNextID( )      --> Get next Java identifier
//                               (see Section 12.2)
// ******************ERRORS**********************************
// Error checking on comments and quotes is performed

public class Tokenizer
{
    public Tokenizer( Reader inStream )
      { errors = 0; ch = '\0'; currentLine = 1;
        in = new PushbackReader( inStream ); }

    public static final int SLASH_SLASH = 0;
    public static final int SLASH_STAR  = 1;

    public int getLineNumber( )
      { return currentLine; }
    public int getErrorCount( )
      { return errors; }
    public char getNextOpenClose( )
      { /* Figure 11.7 */ }
    public char getNextID( )
      { /* Figure 12.29 */ }

    private boolean nextChar( )
      { /* Figure 11.4 */ }
    private void putBackChar( )
      { /* Figure 11.4 */ }
    private void skipComment( int start )
      { /* Figure 11.5 */ }
    private void skipQuote( char quoteType )
      { /* Figure 11.6 */ }
    private void processSlash( )
      { /* Figure 11.7 */ }
    private static final boolean isIdChar( char ch )
      { /* Figure 12.27 */ }
    private String getRemainingString( )
      { /* Figure 12.28 */ }

    private PushbackReader in;    // The input stream
    private char ch;              // Current character
    private int currentLine;      // Current line
    private int errors;           // Number of errors seen
}
```

```java
import java.io.Reader;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;

import java.util.Stack;


// Balance class: check for balanced symbols
//
// CONSTRUCTION: with a Reader object
// ******************PUBLIC OPERATIONS***********************
// int checkBalance( )   --> Print mismatches
//                           return number of errors
// ******************ERRORS**********************************
// Error checking on comments and quotes is performed
// main checks for balanced symbols.

public class Balance
{
    public Balance( Reader inStream )
      { errors = 0; tok = new Tokenizer( inStream ); }

    public int checkBalance( )
      { /* Figure 11.8 */ }

    private Tokenizer tok;
    private int errors;

    /**
     * Symbol nested class;
     * represents what will be placed on the stack.
     */
    private static class Symbol
    {
        public char token;
        public int  theLine;

        public Symbol( char tok, int  line )
        {
            token   = tok;
            theLine = line;
        }
    }

    private void checkMatch( Symbol opSym, Symbol clSym )
      { /* Figure 11.9 */ }
}
```

figure 11.3

Class skeleton for a balanced-symbol program

```java
/**
 * nextChar sets ch based on the next character in the input stream.
 * putBackChar puts the character back onto the stream.
 * It should be used only once after a call to nextChar.
 * Both routines adjust currentLine if necessary.
 */
private boolean nextChar( )
{
    try
    {
        int readVal = in.read( );
        if( readVal == -1 )
            return false;
        ch = (char) readVal;
        if( ch == '\n' )
            currentLine++;
        return true;
    }
    catch( IOException e )
      { return false; }
}

private void putBackChar( )
{
    if( ch == '\n' )
        currentLine--;
    try
      { in.unread( (int) ch ); }
    catch( IOException e ) { }
}
```

figure 11.4

The `nextChar` routine for reading the next character, updating `currentLine` if necessary, and returning `true` if not at the end of file; and the `putBackChar` routine for putting back `ch` and updating `currentLine` if necessary

```java
/**
 * Precondition: We are about to process a comment;
 *                 have already seen comment-start token
 * Postcondition: Stream will be set immediately after
 *                 comment-ending token
 */
private void skipComment( int start )
{
    if( start == SLASH_SLASH )
    {
        while( nextChar( ) && ( ch != '\n' ) )
            ;
        return;
    }

        // Look for a */ sequence
    boolean state = false;   // True if we have seen *

    while( nextChar( ) )
    {
        if( state && ch == '/' )
            return;
        state = ( ch == '*' );
    }
    errors++;
    System.out.println( "Unterminated comment!" );
}
```

figure 11.5

The skipComment routine for moving past an already started comment

figure 11.6

The `skipQuote` routine for moving past an already started character or string constant

```java
/**
 * Precondition: We are about to process a quote;
 *                    have already seen beginning quote.
 * Postcondition: Stream will be set immediately after
 *                    matching quote
 */
private void skipQuote( char quoteType )
{
    while( nextChar( ) )
    {
        if( ch == quoteType )
            return;
        if( ch == '\n' )
        {
            errors++;
            System.out.println( "Missing closed quote at line " +
                                currentLine );
            return;
        }
        else if( ch == '\\' )
            nextChar( );
    }
}
```

figure 11.7

The `getNextOpenClose` routine for skipping comments and quotes and returning the next opening or closing character, along with the `processSlash` routine

```java
 /**
  * Get the next opening or closing symbol.
  * Return false if end of file.
  * Skip past comments and character and string constants
  */
public char getNextOpenClose( )
{
    while( nextChar( ) )
    {
        if( ch == '/' )
            processSlash( );
        else if( ch == '\'' || ch == '"' )
            skipQuote( ch );
        else if( ch == '(' || ch == '[' || ch == '{' ||
                 ch == ')' || ch == ']' || ch == '}' )
            return ch;
    }
    return '\0';            // End of file
}

/**
 * After the opening slash is seen deal with next character.
 * If it is a comment starter, process it; otherwise putback
 * the next character if it is not a newline.
 */
private void processSlash( )
{
    if( nextChar( ) )
    {
        if( ch == '*' )
        {
            // Javadoc comment
            if( nextChar( ) && ch != '*' )
                putBackChar( );
            skipComment( SLASH_STAR );
        }
        else if( ch == '/' )
            skipComment( SLASH_SLASH );
        else if( ch != '\n' )
            putBackChar( );
    }
}
```

```
/**
 * Print an error message for unbalanced symbols.
 * @return number of errors detected.
 */
public int checkBalance( )
{
    char ch;
    Symbol match = null;
    Stack<Symbol> pendingTokens = new Stack<Symbol>( );

    while( ( ch = tok.getNextOpenClose( ) ) != '\0' )
    {
        Symbol lastSymbol = new Symbol( ch, tok.getLineNumber( ) );

        switch( ch )
        {
          case '(': case '[': case '{':
            pendingTokens.push( lastSymbol );
            break;

          case ')': case ']': case '}':
            if( pendingTokens.isEmpty( ) )
            {
                errors++;
                System.out.println( "Extraneous " + ch +
                                    " at line " + tok.getLineNumber( ) );
            }
            else
            {
                match = pendingTokens.pop( );
                checkMatch( match, lastSymbol );
            }
            break;

          default: // Cannot happen
            break;
        }
    }

    while( !pendingTokens.isEmpty( ) )
    {
        match = pendingTokens.pop( );
        System.out.println( "Unmatched " + match.token +
                            " at line "  + match.theLine );
        errors++;
    }
    return errors + tok.getErrorCount( );
}
```

figure 11.8

The `checkBalance` algorithm

```java
/**
 * Print an error message if clSym does not match opSym.
 * Update errors.
 */
private void checkMatch( Symbol opSym, Symbol clSym )
{
    if( opSym.token == '(' && clSym.token != ')' ||
        opSym.token == '[' && clSym.token != ']' ||
        opSym.token == '{' && clSym.token != '}' )
    {
        System.out.println( "Found " + clSym.token + " on line " +
               tok.getLineNumber( ) + "; does not match " + opSym.token
               + " at line " + opSym.theLine );
        errors++;
    }
}
```

figure 11.9

The `checkMatch` routine for checking that the closing symbol matches the opening symbol

```java
// main routine for balanced symbol checker.
// If no command line parameters, standard output is used.
// Otherwise, files in command line are used.
public static void main( String [ ] args )
{
    Balance p;

    if( args.length == 0 )
    {

        p = new Balance( new InputStreamReader( System.in ) );
        if( p.checkBalance( ) == 0 )
            System.out.println( "No errors!" );
        return;
    }

    for( int i = 0; i < args.length; i++ )
    {
        FileReader f = null;
        try
        {
            f = new FileReader( args[ i ] );

            System.out.println( args[ i ] + ": " );
            p = new Balance( f );
            if( p.checkBalance( ) == 0 )
                System.out.println( "   ...no errors!" );
        }
        catch( IOException e )
          { System.err.println( e + args[ i ] ); }
        finally
        {
            try
              { if( f != null ) f.close( ); }
            catch( IOException e )
              { }
        }
    }
}
```

figure 11.10

The `main` routine with command-line arguments

Evaluation of arithmetic expressions

Evaluate the expression

    1 * 2 + 3 * 4

Value = (1 * 2) + (3 * 4) = 2 + 12 = 14.

Simple left-to-right evaluation is not sufficient.

We must take into account that multiplication has higher **precedence** than addition (* binds more tightly than +).

Intermediate values have to be saved.

Associativity

If two operators have the same precedence, their **associativity** determines which one gets evaluated first.

The expression 4 - 3 - 2 is evaluated as (4 - 3) - 2, since minus associates **left-to-right**.

The expression 4 ^ 3 ^ 2 in which ^ is the exponentiation operator is evaluated as 4 ^ (3 ^ 2), since ^ associates **right-to-left**.

Parentheses

The evaluation order may be clarified by means of parentheses.

Example:

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

may be expressed as

```
( 1 - 2 ) - ( ( ( 4 * ( 5 ^ 3 ) ) * 6 ) / ( 7 ^ ( 2 ^ 2 ) ) )
```

Although parentheses make the order of evaluation unambiguous, they do not make the mechanism for evaluation any clearer.

Postfix notation

(Reverse Polish notation)

The normal notation used for arithmetic expressions is called **infix** notation (the operators are placed *between* its operands, e.g., 3 + 4).

Evaluation may be simplified by using **postfix** notation (the operators are placed *after* its operands (e.g., 3 4 +).

The infix expression

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

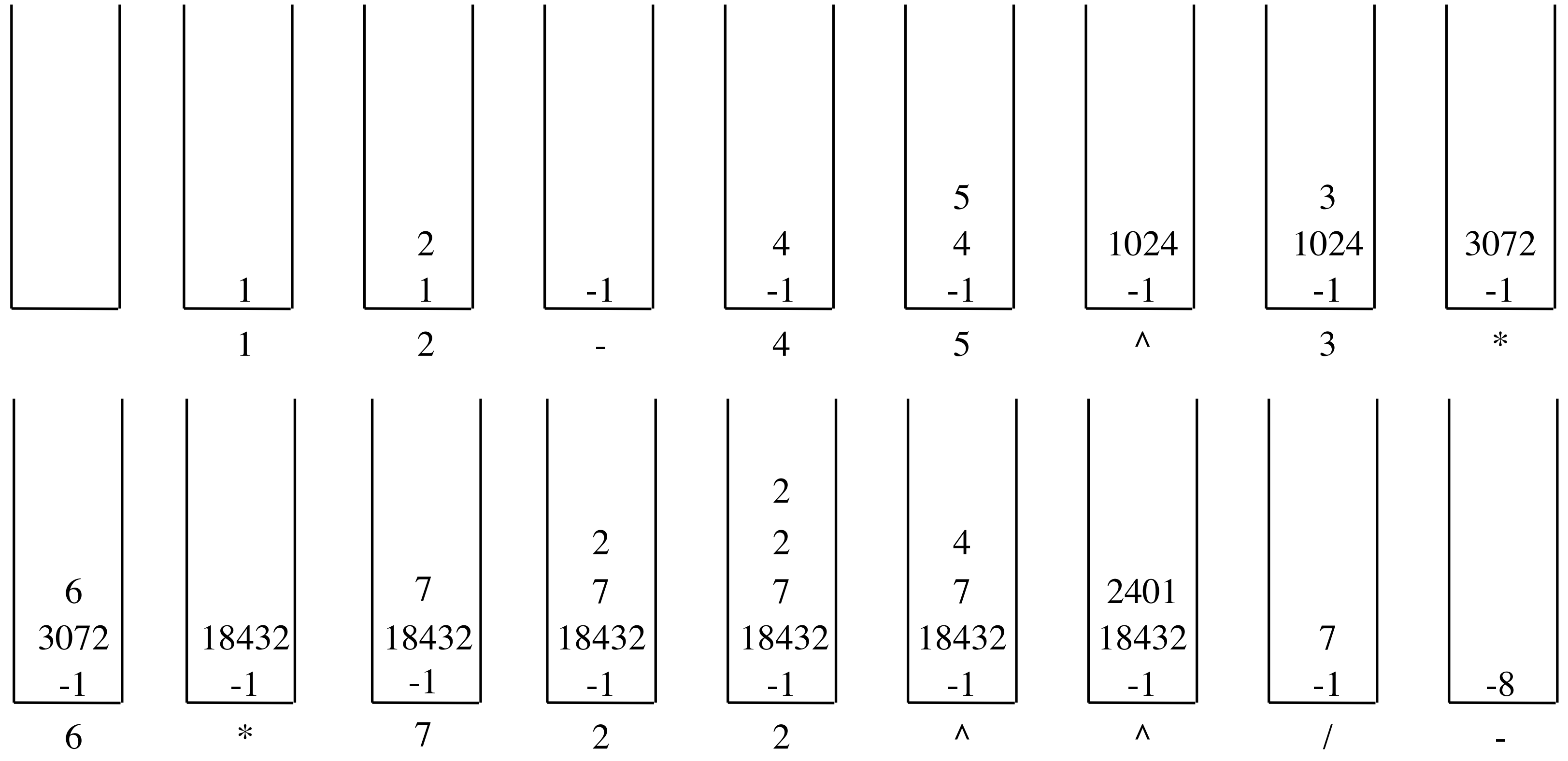
may be written in postfix notation as

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

Notice that postfix notation is **parenthesis-free**.

Evaluation of a postfix expression

1 2 - 4 5 ^ 3 * 6 * 7 2 2 ^ ^ / -   (postfix)

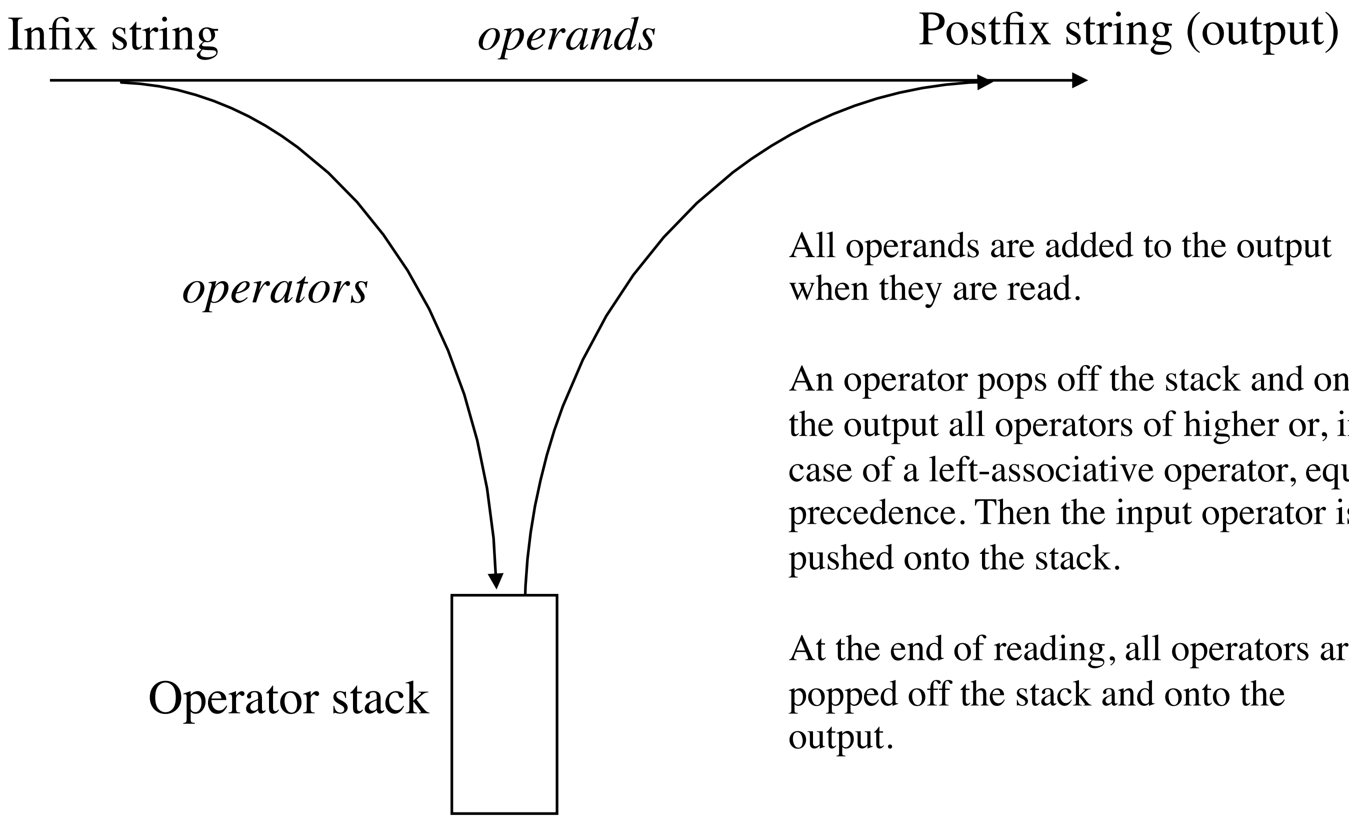


```
class Calculator {
    static int valueOf(String str) {
        IntStack s = new IntStack();
        for (int i = 0; i < str.length(); i++) {
            char c = str.charAt(i);
            if (Character.isDigit(c))
                s.push(Character.getNumericValue(c));
            else if (!Character.isWhitespace(c)) {
                int rhs = s.pop(), lhs = s.pop();
                switch(c) {
                case '+': s.push(lhs + rhs); break;
                case '-': s.push(lhs - rhs); break;
                case '*': s.push(lhs * rhs); break;
                case '/': s.push(lhs / rhs); break;
                case '^': s.push((int) Math.pow(lhs, rhs)); break;
                }
            }
        }
        return s.pop();
    }
}
```

We assume single-digit numbers

Conversion from infix to postfix

Dijkstra’s shunting-yard algorithm



All operands are added to the output when they are read.

An operator pops off the stack and onto the output all operators of higher or, in case of a left-associative operator, equal precedence. Then the input operator is pushed onto the stack.

At the end of reading, all operators are popped off the stack and onto the output.

Infix Expression	Postfix Expression	Associativity
`2 + 3 + 4`	`2 3 + 4 +`	Left-associative: Input + is lower than stack +.
`2 ^ 3 ^ 4`	`2 3 4 ^ ^`	Right-associative: Input ^ is higher than stack ^.

figure 11.12

Examples of using associativity to break ties in precedence

Conversion from infix to postfix

1 - 2 - 4 ^ 5 * 3 * 6 / 7 ^ 2 ^ 2 (infix)

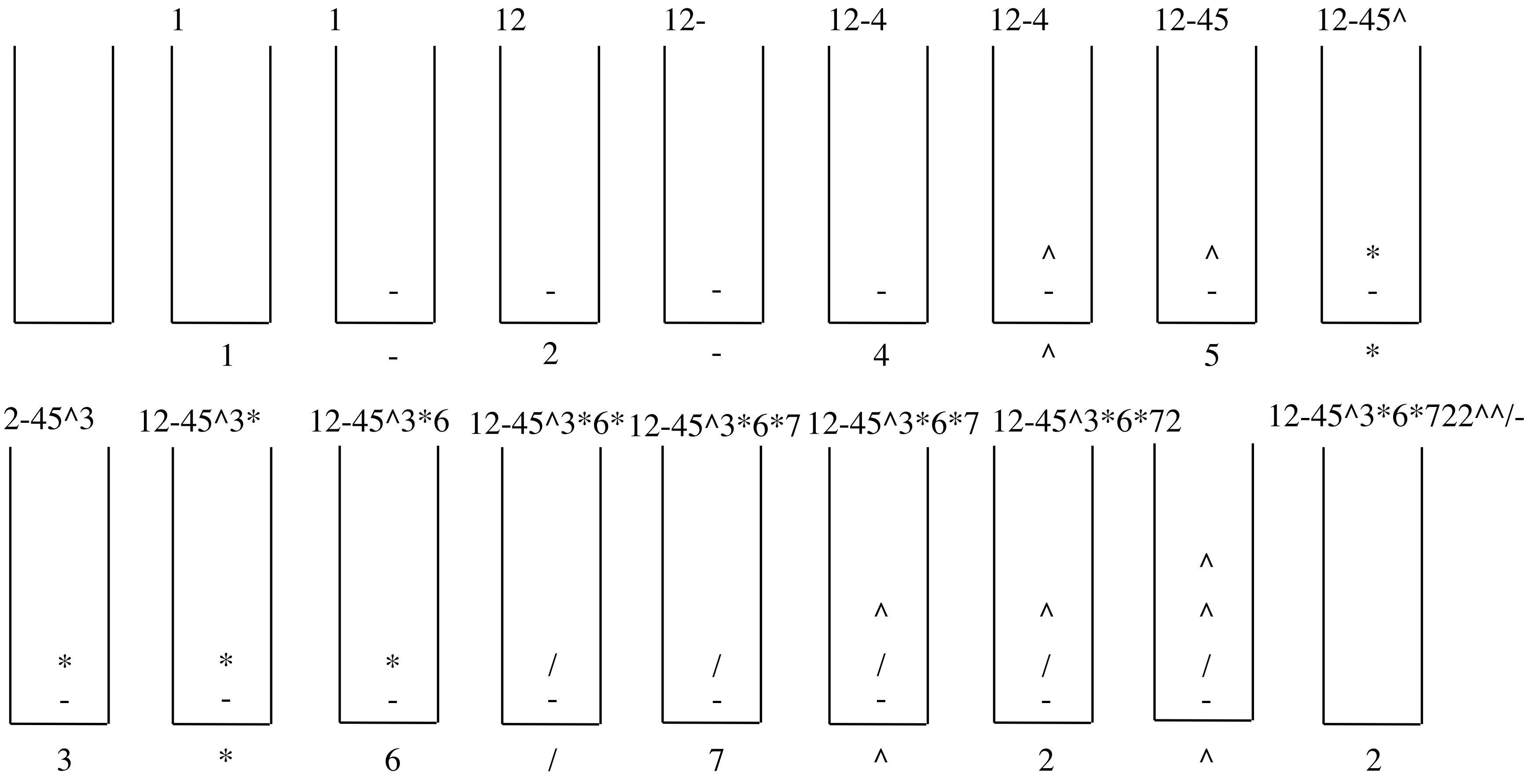
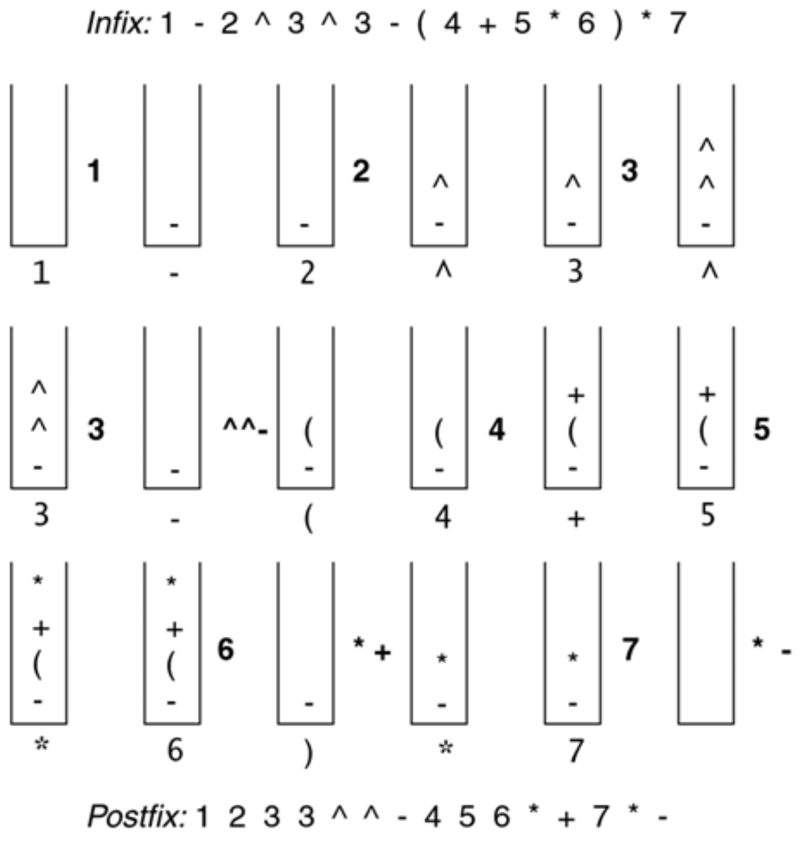


figure 11.13

Infix to postfix conversion



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

```java
import java.util.Stack;
import java.util.StringTokenizer;
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;

// Evaluator class interface: evaluate infix expressions.
//
// CONSTRUCTION: with a String
//
// ******************PUBLIC OPERATIONS***********************
// long getValue( )      --> Return value of infix expression
// ******************ERRORS**********************************
// Some error checking is performed

public class Evaluator
{
    private static class Precendence
      { /* Figure 11.20 */ }
    private static class Token
      { /* Figure 11.15 */ }
    private static class EvalTokenizer
      { /* Figure 11.15 */ }

    public Evaluator( String s )
    {
        opStack = new Stack<Integer>( ); opStack.push( EOL );
        postfixStack = new Stack<Long>( );
        str = new StringTokenizer( s, "+*-/^() ", true );
    }
    public long getValue( )
      { /* Figure 11.17 */ }

    private Stack<Integer>  opStack;      // Operator stack for conversion
    private Stack<Long>     postfixStack; // Stack for postfix machine
    private StringTokenizer str;          // StringTokenizer stream

    private void processToken( Token lastToken )
      { /* Figure 11.21 */ }
    private long getTop( )
      { /* Figure 11.18 */ }
    private void binaryOp( int topOp )
      { /* Figure 11.19 */ }
}
```

figure 11.14

The `Evaluator` class skeleton

figure 11.15

The Token and EvalTokenizer nested classes

```
private static class Token
{
    public Token( )
      { this( EOL ); }
    public Token( int t )
      { this( t, 0 ); }
    public Token( int t, long v )
      { type = t; value = v; }

    public int getType( )
      { return type; }
    public long getValue( )
      { return value; }

    private int type = EOL;
    private long value = 0;
}

private static class EvalTokenizer
{
    public EvalTokenizer( StringTokenizer is )
      { str = is; }

    /**
     * Find the next token, skipping blanks, and return it.
     * For VALUE token, place the
     * processed value in currentValue.
     * Print error message if input is unrecognized.
     */
    public Token getToken( )
      { /* Figure 11.16 */ }

    private StringTokenizer str;
}
```

```java
/**
 * Find the next token, skipping blanks, and return it.
 * For VALUE token, place the processed value in currentValue.
 * Print error message if input is unrecognized.
 */
public Token getToken( )
{
    long theValue;

    if( !str.hasMoreTokens( ) )
        return new Token( );

    String s = str.nextToken( );
    if( s.equals( " " ) ) return getToken( );
    if( s.equals( "^" ) ) return new Token( EXP );
    if( s.equals( "/" ) ) return new Token( DIV );
    if( s.equals( "*" ) ) return new Token( MULT );
    if( s.equals( "(" ) ) return new Token( OPAREN );
    if( s.equals( ")" ) ) return new Token( CPAREN );
    if( s.equals( "+" ) ) return new Token( PLUS );
    if( s.equals( "-" ) ) return new Token( MINUS );

    try
      { theValue = Long.parseLong( s ); }
    catch( NumberFormatException e )
    {
        System.err.println( "Parse error" );
        return new Token( );
    }

    return new Token( VALUE, theValue );
}
```

figure 11.16

The `getToken` routine for returning the next token in the input stream

figure 11.17

The getValue routine for reading and processing tokens and then returning the item at the top of the stack

```java
/**
 * Public routine that performs the evaluation.
 * Examine the  postfix machine to see if a single result is
 * left and if so, return it; otherwise print error.
 * @return the result.
 */
public long getValue( )
{
    EvalTokenizer tok = new EvalTokenizer( str );
    Token lastToken;

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

    if( postfixStack.isEmpty( ) )
    {
        System.err.println( "Missing operand!" );
        return 0;
    }

    long theResult = postFixTopAndPop( );
    if( !postfixStack.isEmpty( ) )
        System.err.println( "Warning: missing operators!" );

    return theResult;
}
```

figure 11.18

The routines for popping the top item in the postfix stack

```
/*
 * topAndPop the postfix machine stack; return the result.
 * If the stack is empty, print an error message.
 */
private long postfixPop( )
{
    if ( postfixStack.isEmpty( ) )
    {
        System.err.println( "Missing operand" );
        return 0;
    }
    return postfixStack.pop( );
}
```

```java
/**
 * 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.
 */
private void binaryOp( int topOp )
{
    if( topOp == OPAREN )
    {
        System.err.println( "Unbalanced parentheses" );
        opStack.pop( );
        return;
    }
    long rhs = postfixPop( );
    long lhs = postfixPop( );

    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
        {
            System.err.println( "Division by zero" );
            postfixStack.push( lhs );
        }
    opStack.pop( );
}
```

figure 11.19

The `BinaryOp` routine for applying `topOp` to the postfix stack

figure 11.20

Table of precedences used to evaluate an infix expression

```
private static final int EOL     = 0;
private static final int VALUE   = 1;
private static final int OPAREN  = 2;
private static final int CPAREN  = 3;
private static final int EXP     = 4;
private static final int MULT    = 5;
private static final int DIV     = 6;
private static final int PLUS    = 7;
private static final int MINUS   = 8;

private static class Precedence
{
    public int inputSymbol;
    public int topOfStack;

    public Precedence( int inSymbol, int topSymbol )
    {
        inputSymbol = inSymbol;
        topOfStack  = topSymbol;
    }
}

    // precTable matches order of Token enumeration
private static Precedence [ ] precTable =
{
    new Precedence(   0, -1 ),  // EOL
    new Precedence(   0,  0 ),  // VALUE
    new Precedence( 100,  0 ),  // OPAREN
    new Precedence(   0, 99 ),  // CPAREN
    new Precedence(   6,  5 ),  // EXP
    new Precedence(   3,  4 ),  // MULT
    new Precedence(   3,  4 ),  // DIV
    new Precedence(   1,  2 ),  // PLUS
    new Precedence(   1,  2 )  // MINUS
}
```

```java
/**
 * After a token is read, use operator precedence parsing
 * algorithm to process it; missing opening parentheses
 * are detected here.
 */
private void processToken( Token lastToken )
{
    int topOp;
    int lastType = lastToken.getType( );

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

      case CPAREN:
        while( ( topOp = opStack.peek( ) ) != OPAREN && topOp != EOL )
            binaryOp( topOp );
        if( topOp == OPAREN )
            opStack.pop( );  // Get rid of opening parenthesis
        else
            System.err.println( "Missing open parenthesis" );
        break;

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

figure 11.21

The `processToken` routine for processing `lastToken`, using the operator precedence parsing algorithm

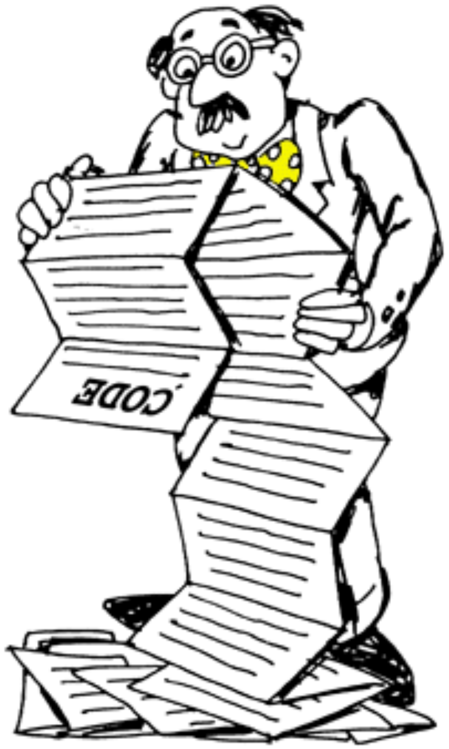
figure 11.22

A simple `main` for evaluating expressions repeatedly

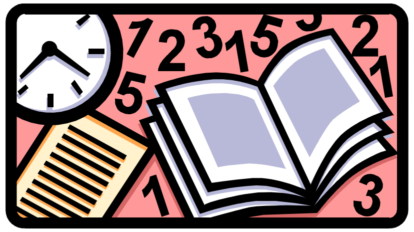
```
/**
 * Simple main to exercise Evaluator class.
 */
public static void main( String [ ] args )
{
    String str;
    BufferedReader in = new BufferedReader(
                            new InputStreamReader( System.in ) );

    try
    {
        System.out.println( "Enter expressions, 1 per line:" );
        while( ( str = in.readLine( ) ) != null )
        {
            System.out.println( "Read: " + str );
            Evaluator ev = new Evaluator( str );
            System.out.println( ev.getValue( ) );
            System.out.println( "Enter next expression:" );
        }
    }
    catch( IOException e ) { e.printStackTrace( ); }
}
```

Parsing

Parsing or **syntactic analysis** is the process of analyzing a string of symbols, either in natural language or in computer languages, according to the rules of a formal grammar



Example problem



Objective:

A program for reading and evaluating arithmetic expressions.

We solve an easier problem first:

Read a string and check that it is a legal arithmetic expression.

Grammar for arithmetic expressions

Use a *grammar* to specify legal arithmetic expressions:

```
<expression>  ::= <term> |
                  <term> + <expression> |
                  <term> - <expression>
<term>  ::= <factor> |
            <factor> * <term> |
            <factor> / <term>
<factor> ::= <number> |
            (<expression>)
```

The grammar is defined by **production rules** that consist of

(1) *nonterminal* symbols: *expression*, *term*, *factor*, and *number*

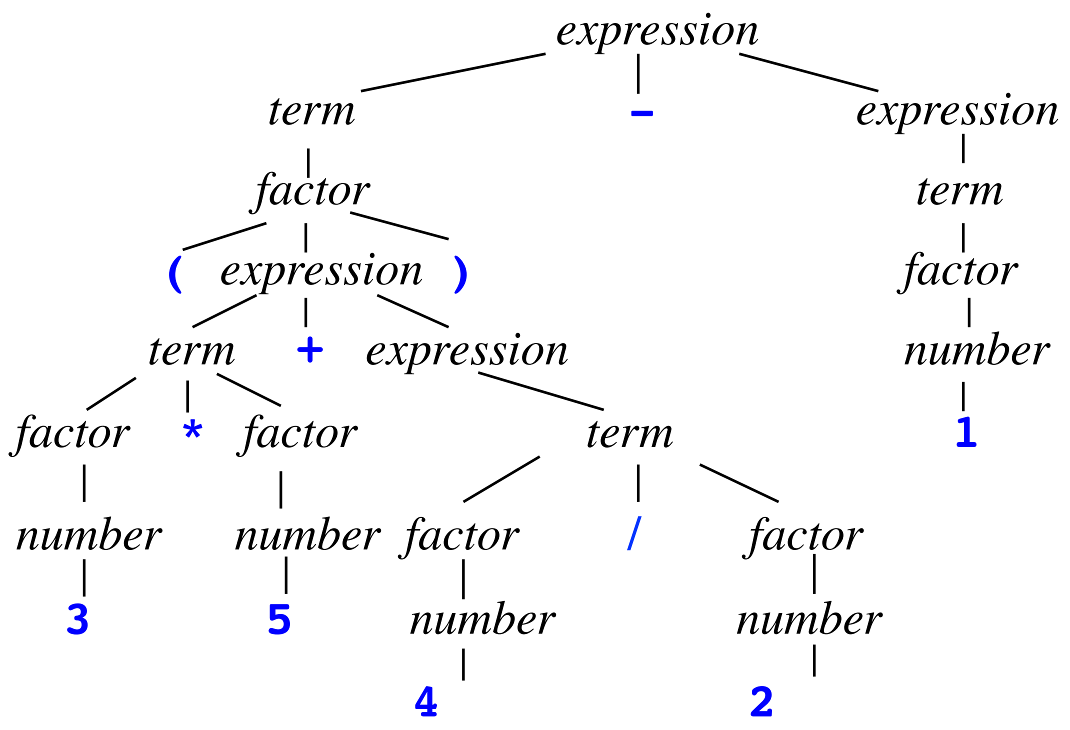
(2) *terminal symbols*: `+`, `-`, `*`, `/`, `(`, `)`, and digits

(3) *meta-symbols*: ::=, <, >, and |

Syntax trees

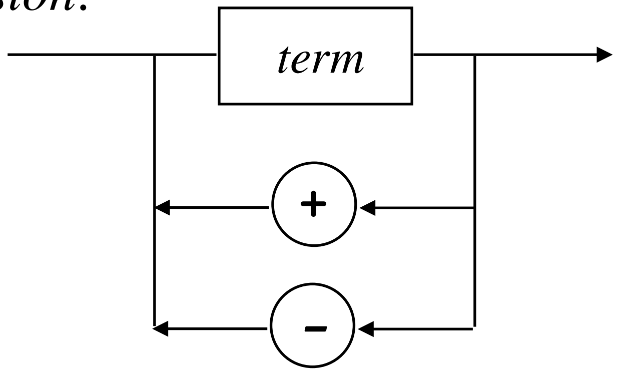
A string is an arithmetic expression if it is possible – using the production rules – to *derive* the string from <*expression*>.
In each step of a derivation we replace a nonterminal symbol with one of the alternatives of the right hand side of its rule.

Syntax tree for `(3 * 5 + 4 / 2) - 1`

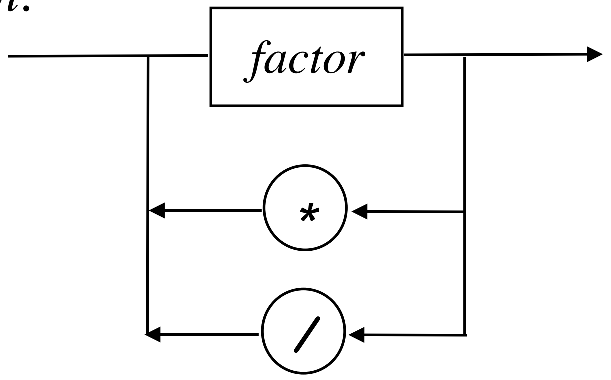


Syntax diagrams

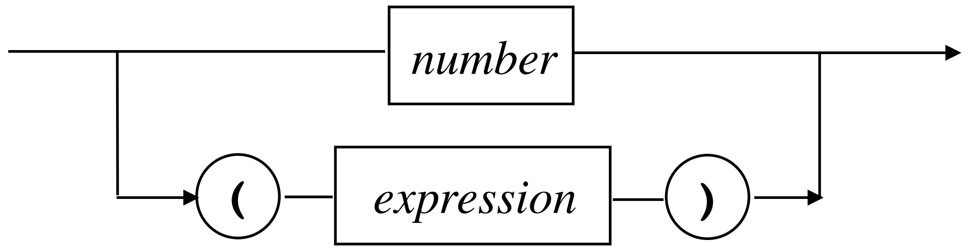
expression:



term:



factor:



Syntax analysis by recursive descent

A Java program for syntax analysis may be constructed directly from the syntax diagrams.

```
void expression() {
    term();
    while (token == PLUS || token == MINUS)
      { getToken(); term(); }
}
```

```
static final int PLUS   = 1, MINUS = 2,
                 MULT   = 3, DIV   = 4,
                 LPAR   = 5, RPAR  = 6,
                 NUMBER = 7, EOS   = 8;
int token;
```

```
void term() {
    factor();
    while (token == MULT || token == DIV)
        { getToken(); factor(); }
}
```

```
void factor() {
    if (token == NUMBER)
        ;
    else if (token == LPAR) {
        getToken();
        expression();
        if (token != RPAR)
            error("missing right parenthesis");
    } else
        error("illegal factor: " + token);
    getToken();
}
```

```
StringTokenizer str;
```

```
void parse(String s) {
    str = new StringTokenizer(s,"+-*/() ", true);
    getToken();
    expression();
}
```

Example of call:

```
parse("(3*5+4/2)-1");
```

```
void getToken() {
    String s;
    try {
        s = str.nextToken();
    } catch (NoSuchElementException e) {
        token = EOS;
        return;
    }
    if (s.equals(" ")) getToken();
    else if (s.equals("+")) token = PLUS;
    else if (s.equals("-")) token = MINUS;
    else if (s.equals("*")) token = MULT;
    else if (s.equals("/")) token = DIV;
    else if (s.equals("(")) token = LPAR;
    else if (s.equals(")")) token = RPAR;
    else {
        try {
            Double.parseDouble(s);
            token = NUMBER;
        } catch (NumberFormatException e)
          { error("number expected"); }
    }
}
```

Evaluation of arithmetic expressions

Evaluation may achieved by few simple changes of the syntax analysis program.

Each analysis method should return its corresponding value (instead of `void`).

```java
double valueOf(String s) {
    str = new StringTokenizer(s,"+-*/() ", true);
    getToken();
    return expression();
}
```

Example of call:

```java
double result = valueOf("(3*5+4/2)-1");
double expression() {
    double v = term();
    while (token == PLUS || token == MINUS)
        if (token == PLUS)
          { getToken(); v += term(); }
        else
          { getToken(); v -= term(); }
    return v;
}
```

```
double term() {
    double v = factor();
    while (token == MULT || token == DIV)
        if (token == MULT)
          { getToken(); v *= factor(); }
        else
          { getToken(); v /= factor(); }
    return v;
}
```

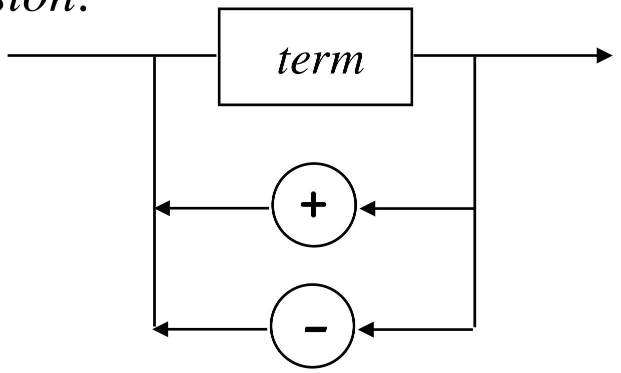
```
double factor() {
    double v;
    if (token == NUMBER)
        v = value;
    else if (token == LPAR) {
        getToken();
        v = expression();
        if (token != RPAR)
            error("missing right parenthesis");
    } else
        error("illegal factor: " + token);
    getToken();
    return v;
}
```

```
double value;
```

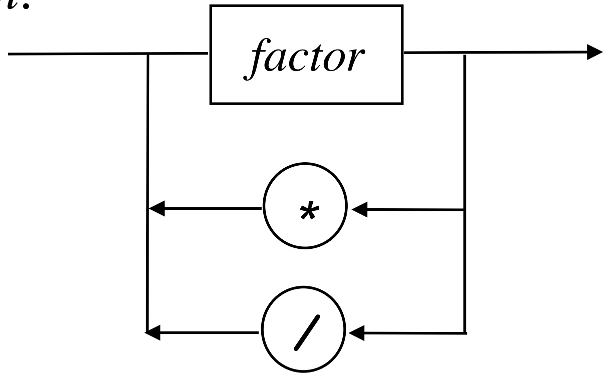
```
void getToken() {
    String s;
    try {
        s = str.nextToken();
    } catch(NoSuchElementException e) {
        token = EOS;
        return;
    }
    if (s.equals(" ")) getToken();
    else if (s.equals("+")) token = PLUS;
    else if (s.equals("-")) token = MINUS;
    else if (s.equals("*")) token = MULT;
    else if (s.equals("/")) token = DIV;
    else if (s.equals("(")) token = LPAR;
    else if (s.equals(")")) token = RPAR;
    else {
        try {
            value = Double.parseDouble(s);
            token = NUMBER;
        } catch(NumberFormatException e)
          { error("number expected"); }
    }
}
```

Syntax diagrams with ^

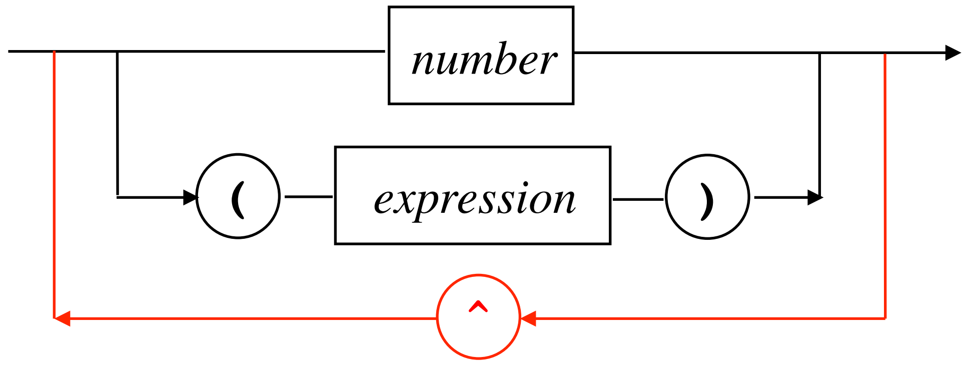
expression:



term:



factor:



```
double factor() {
    double v;
    if (token == NUMBER)
        v = value;
    else if (token == LPAR) {
        getToken();
        v = expression();
        if (token != RPAR)
            error("missing right parenthesis");
    } else
        error("illegal factor: " + token);
    getToken();
    if (token == POWER) {
        getToken();
        v = Math.pow(v, factor());
    }
    return v;
}
```