GIUSEPPE TURINI

CS-102: COMPUTING AND ALGORITHMS 2
LESSON 05
RECURSION (PART 2)

Kettering
UNIVERSITY

HIGHLIGHTS

Processing Arrays and Lists Recursively

- Processing Arrays Recursively
- Processing and Defining Linked Lists Recursively

Recursion as Backtracking Technique

- Backtracking and the Eight Queens Problem

Languages and Grammars

- Definition of Language and Grammar, and Recursive Grammars
- Java Identifiers, Palindromes, and A^nB^n Strings
- **Algebraic Expressions**
    - Infix/Prefix/Postfix Expressions, and Fully Parenthesized Expressions

Recursion and Mathematical Induction

- The Relationship Between Recursion and Mathematical Induction
- The Correctness of the Recursive Factorial Method
- The Cost of Towers of Hanoi

STUDY GUIDE

STUDY MATERIAL

- **This slides.**

SELECTED EXERCISES

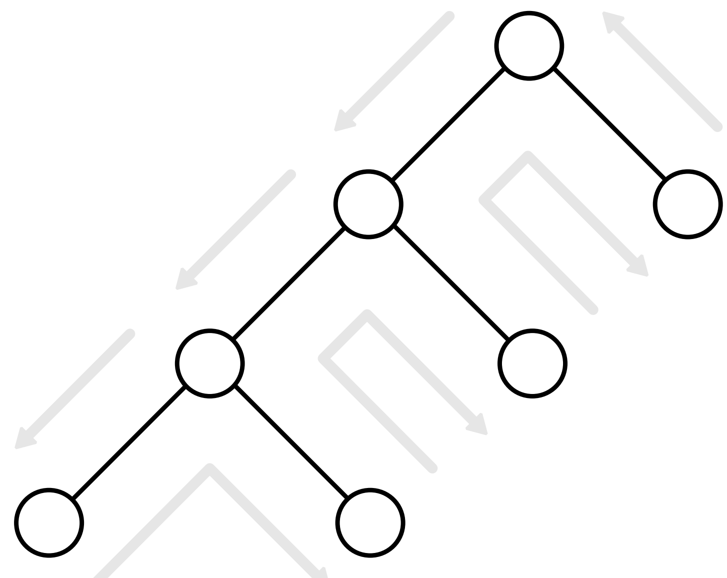
- **Set 1:** ex. 1a-b, ex. 2-3, ex. 7-8, ex. 16-20.
- **Set 2:** ex. 1-5, ex. 7, ex. 9, ex. 12-13, ex. 16-19, ex. 33-35.

ADDITIONAL RESOURCES

- **"Object-Oriented Data Structures Using Java (4th Ed.)", chap. 3.**
- "Data Abstraction and Problem Solving with Java (3rd Ed.)", chap. 3, 5-6.
- "Java Illuminated (5th Ed.)", chap. 13.
- visualgo.net/en/recursion

BACKTRACKING - RECURSION AND BACKTRACKING

Backtracking: A strategy to make successive guesses at a solution. If a particular guess leads to a dead end, **you back up to that guess and replace it with a different guess**. Retrace steps in reverse order and then try a new sequence of steps is called backtracking, and can be combined with recursion to solve problems.



BACKTRACKING - THE EIGHT QUEENS PROBLEM 1

THE EIGHT QUEENS PROBLEM: DEFINITION AND BRUTE-FORCE STRATEGY

Place **8** queens on a chessboard (**8x8**) so that no queen can attack another queen.
Find a solution between the **4426165368** ways to arrange **8** queens on **64** squares.
See: en.wikipedia.org/wiki/eight_queens_puzzle

Note: 4426165368 is the number of **combinations** (i.e. the order of items does not matter) without repetition of **8** (i.e. **k**) distinct items of a set of **64** (i.e. **n**) items, or:
See: en.wikipedia.org/wiki/binomial_coefficient

$$\frac{64!}{8!\,(64-8)!} = \frac{64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57}{8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1} = 4426165368$$

$$\frac{n!}{k!\,(n-k)!} = \binom{n}{k} = \text{binomial coefficient}$$

BACKTRACKING - THE EIGHT QUEENS PROBLEM 2

THE EIGHT QUEENS PROBLEM: DEFINITION AND BRUTE-FORCE STRATEGY

An observation that eliminates many arrangements from consideration is that: **no queen can reside in a row or a column that contains another queen**. So now only **40320** arrangements of queens need to be checked for attacks along diagonals.

Note: 40320 are the combinations without repetition of **8** distinct items:

$$\frac{(8 \times 8) \times (7 \times 7) \times \cdots \times (2 \times 2) \times (1 \times 1)}{8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1} = 40320$$

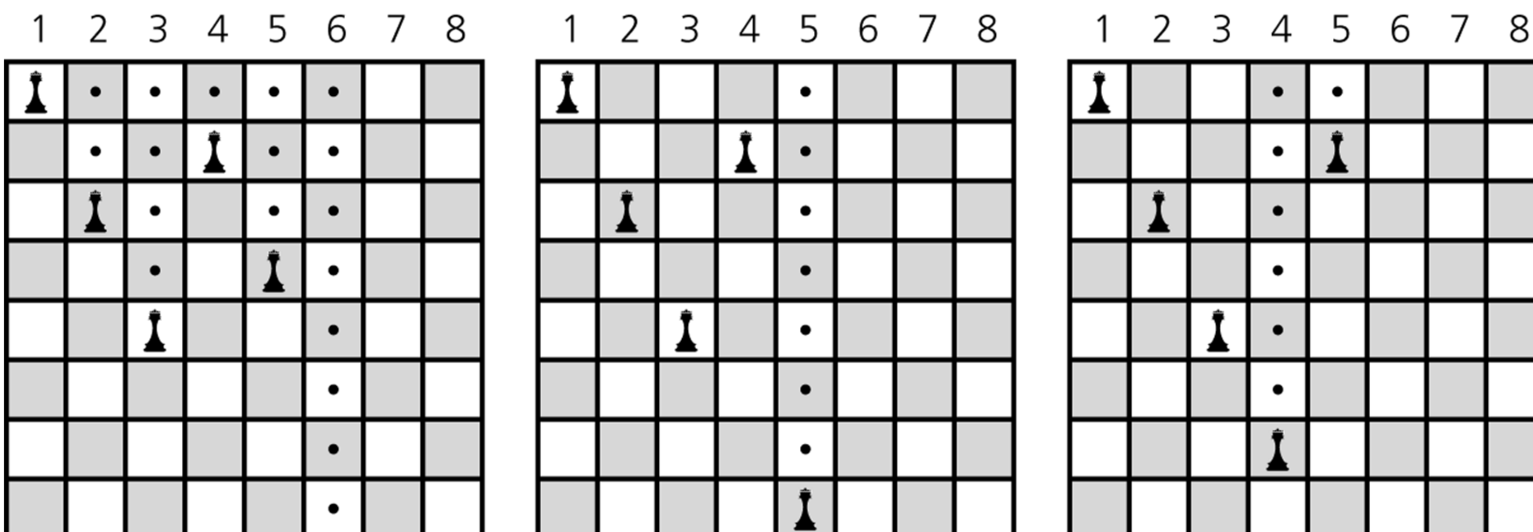
THE EIGHT QUEENS PROBLEM: BACKTRACKING STRATEGY

Providing organization for the guessing strategy:

- place queens one column at a time;
- if you reach an impasse, backtrack to the previous column.

BACKTRACKING - THE EIGHT QUEENS PROBLEM 3

Example: 5 queens that cannot attack each other, but that can attack all of column 6 **(a)**; backtracking to column 5 to try another square for the queen on column 5 **(b)**; backtracking to column 4 to try another square for the queen on column 4, then considering column 5 again **(c)**.



(a) (b) (c)

BACKTRACKING - THE EIGHT QUEENS PROBLEM

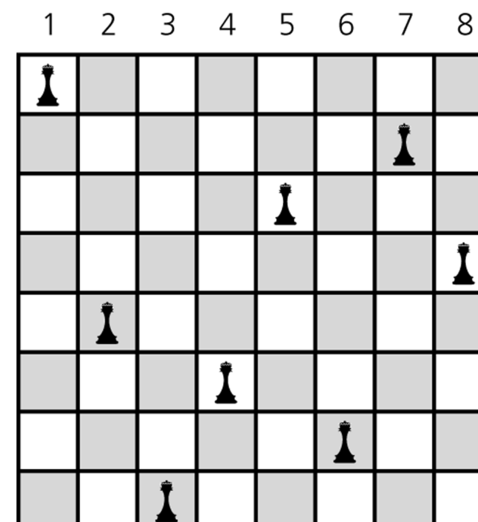
THE EIGHT QUEENS PROBLEM: A RECURSIVE ALGORITHM

Base Case: If there are no more columns to consider, you are finished.

Recursive Step:

1. if you successfully place a queen in current column,
    a. consider the next column (i.e. **smaller problem**);
2. if you cannot place a queen in the current column,
    a. you need to **backtrack**.

In figure, a **solution to the Eight Queens** problem:



See: en.wikipedia.org/wiki/eight_queens_puzzle

BACKTRACKING - THE EIGHT QUEENS PROBLEM

THE EIGHT QUEENS PROBLEM: ALGORITHMS AND DATA STRUCTURES

QUEENS A

```
// Class Queens implementing the chessboard for the Eight Queens problem.
public class Queens {
   public static final int BOARD_SIZE = 8; // Squares per row or column.
   public static final int EMPTY = 0; // Flag to indicate an empty square.
   public static final int QUEEN = 1; // Flag to indicate a square containing a queen.
   private int board[][]; // Chessboard.

   // Default constructor, creates an empty chessboard.
   public Queens() { board = new int[ BOARD_SIZE ][ BOARD_SIZE ]; }

   // Clears the chessboard setting all squares to EMPTY.
   public void clearBoard() {
      for( int row = 0; row < BOARD_SIZE; row++ ) {
         for( int col = 0; col < BOARD_SIZE; col++ ) { board[row][col] = EMPTY; } } }
```

BACKTRACKING - THE EIGHT QUEENS PROBLEM 6

QUEENS B

```
// Displays the chessboard.
public void displayBoard() {
   for( int row = 0; row < BOARD_SIZE; row++ ) {
      for( int col = 0; col < BOARD_SIZE; col++ ) {
         System.out.print( board[row][col] + " " ); }
      System.out.println(); } }
```

QUEENS C

```
// Places queens in columns of the board beginning at the column specified.
// Precondition: queens are placed correctly in columns 1 through col-1.
// Postcondition: returns true if a solution is found, otherwise returns false.
public boolean placeQueens( int col ) {
   if( col > BOARD_SIZE ) { return true; } // Base case.
   else {
      boolean queenPlaced = false; int row = 1;
      while( !queenPlaced && ( row <= BOARD_SIZE ) ) {
         // If square can be attacked by a queen, consider next square in column.
         if(isUnderAttack( row, col ) ) { row++; }
         else { // Place queen and consider next column.
            setQueen( row, col );
            queenPlaced = placeQueens( col + 1 );
            // If no queen is placeable in next column, backtrack.
            // Remove queen placed earlier, and try next square in column.
            if( !queenPlaced ) { removeQueen( row, col ); row++; } } }
      return queenPlaced; } } // Return result.
```

BACKTRACKING - THE EIGHT QUEENS PROBLEM 8

QUEENS D

```
// Sets a queen at square indicated by row and column.
private void setQueen( int row, int col ) { board[row-1][col-1] = QUEEN; }

// Remove (set EMPTY) a queen at square indicated by row and column.
private void removeQueen( int row, int col ) { board[row-1][col-1] = EMPTY; }

// Checks if specified square is under attack by queens in columns 1 through col-1.
private boolean isUnderAttack( int row, int col ) {
   // Check if there is a queen in the same row (in previous columns).
   for( int j = 1; j < col; j++ ) { if( board[row-1][j-1] == QUEEN ) { return true; }}
   // Check if there is a queen in the diagonal passing through the specified square.
   for( int i = row - 1, j = col - 1; ( i > 0 ) && ( j > 0 ); i--, j-- ) {
      if( board[i-1][j-1] == QUEEN ) { return true; } }
   // Check if there is a queen in reverse diagonal passing through specified square.
   for( int i = row + 1, j = col - 1; ( i <= BOARD_SIZE ) && ( j > 0 ); i++, j-- ) {
      if( board[i-1][j-1] == QUEEN ) { return true; } }
   return false; } } // Otherwise the specified square is not under attack.
```

BACKTRACKING - THE EIGHT QUEENS PROBLEM 9

TEST QUEENS

```java
// Test for the Queens class in order to solve the Eight Queens problem.
public class TestQueens {
   public static void main( String[] args ) {
      Queens test = new Queens();
      test.placeQueens(1);
      test.displayBoard(); } }
```

```
--------------------
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
--------------------
```

LANGUAGES & GRAMMARS - DEFINITIONS 1

Language: A set of strings of symbols from a finite alphabet (English, Java, etc.).

If we consider **a Java program as one long string of characters**, we can define the language of Java programs as:

$$\text{Java Programs} = \{\, w : w \text{ is a syntactically correct Java program} \,\}$$

Note: A language does not have to be a programming/communication language.

Example: The **set of algebraic expressions** can be defined as:

$$\text{Algebraic Expressions} = \{\, w : w \text{ is an algebraic expression} \,\}$$

LANGUAGES & GRAMMARS - DEFINITIONS 2

Grammar: A **grammar** states the rules for forming the strings in a language.

GRAMMARS: RECURSIVE GRAMMARS AND RECOGNITION ALGORITHMS

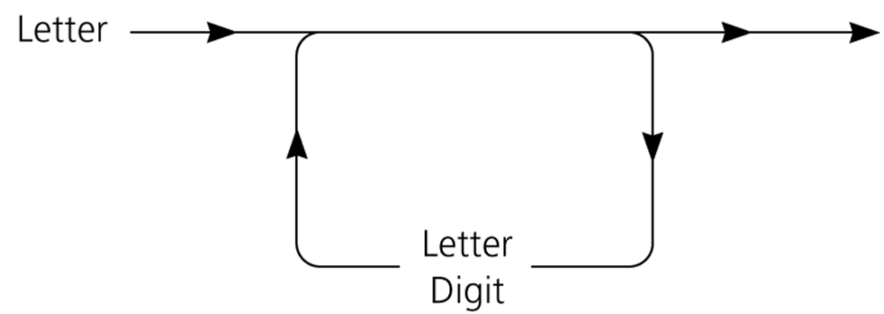
Recursive grammars ease the writing of **recognition algorithms** for languages. A recognition algorithm determines whether a given string is in the language.

GRAMMARS: SYMBOLS

$x \mid y$	means	x or y
xy	means	x followed by y
$x \cdot y$	means	x concatenated to y (or y appended to x)
$\langle word \rangle$	means	any instance of $word$ that the definition defines

LANGUAGES & GRAMMARS - JAVA IDENTIFIERS 1

Java Identifier: A Java identifier begins with a letter and is followed by zero or more letters/digits.



JAVA IDENTIFIERS: LANGUAGE

$$\text{Java Identifiers} = \{\, w : w \text{ is a legal Java identifier} \,\}$$

JAVA IDENTIFIERS: GRAMMAR (RECURSIVE)

$$\begin{aligned}
\langle identifier\rangle &= \langle letter\rangle \mid \langle identifier\rangle\langle letter\rangle \mid \langle identifier\rangle\langle digit\rangle \mid \\
&\qquad \$\,\langle identifier\rangle \mid _\,\langle identifier\rangle \\
\langle letter\rangle &= \mathrm{a} \mid \mathrm{b} \mid \cdots \mid \mathrm{z} \mid \mathrm{A} \mid \mathrm{B} \mid \cdots \mid \mathrm{Z} \\
\langle digit\rangle &= 0 \mid 1 \mid \cdots \mid 9
\end{aligned}$$

JAVA IDENTIFIERS: RECOGNITION ALGORITHM (RECURSIVE)

```
// Java identifiers recognition algorithm (pseudocode).
isJavaIdentifier( w ) {
   if( w is of length 1 ) {
      if( w is a letter ) { return true }
      else { return false } }
   else if( the first character of w is '$' or '_') {
      return isJavaIdentifier( w minus its first character ) }
   else if( the last character of w is a letter or a digit ) {
      return isJavaIdentifier( w minus its last character ) }
   else { return false } }
```

LANGUAGES & GRAMMARS - PALINDROMES 1

Palindrome: A palindrome is a string that reads the same backward or forward (e.g. “radar”, “racecar”, etc.).

PALINDROMES: LANGUAGE

$$\text{Palindromes} = \{\, w : w \text{ reads the same backward or forward} \,\}$$

PALINDROMES: GRAMMAR (RECURSIVE)

$$\langle pal \rangle = \text{emptystring} \mid \langle char \rangle \mid \text{a} \, \langle pal \rangle \, \text{a} \mid \text{b} \, \langle pal \rangle \, \text{b} \mid \cdots \mid \text{Z} \, \langle pal \rangle \, \text{Z}$$
$$\langle char \rangle = \text{a} \mid \text{b} \mid \cdots \mid \text{z} \mid \text{A} \mid \text{B} \mid \cdots \mid \text{Z}$$

LANGUAGES & GRAMMARS - PALINDROMES 2

PALINDROMES: RECOGNITION ALGORITHM (RECURSIVE)

```
// Palindromes recognition algorithm (pseudocode).
isPalindrome( w ) {
   if( ( w is the empty string ) or ( w is of length 1 ) ) { return true }
   else if( the first and last characters of w are the same letter ) {
      return isPalindrome( w minus its first and last characters ) }
   else { return false } }
```

LANGUAGES & GRAMMARS - $A^N B^N$ STRINGS 1

$A^n B^n$ String: An **$A^n B^n$** string consists of **n** consecutive **A** followed by **n** consecutive **B**.

$A^n B^n$ STRINGS: LANGUAGE

$$\mathrm{A^n B^n\ Strings} = \{\, w : w \text{ is of the form } \mathrm{A}^n \mathrm{B}^n \text{ for some } n \geq 0 \,\}$$

$A^n B^n$ STRINGS: GRAMMAR (RECURSIVE)

$$\langle word \rangle = \text{emptystring} \mid \mathrm{A}\, \langle word \rangle\, \mathrm{B}$$

LANGUAGES & GRAMMARS - $A^N B^N$ STRINGS 2

$A^n B^n$ STRINGS: RECOGNITION ALGORITHM (RECURSIVE)

```
// AⁿBⁿ strings recognition algorithm (pseudocode).
isAnBnString( w ) {
   if( the length of w is zero ) { return true }
   else if( ( w begins with character A ) and ( w ends with character B ) ) {
      return isAnBnString( w minus its first and last characters ) }
   else { return false } }
```

LANGUAGES & GRAMMARS - EXPRESSIONS 1

ALGEBRAIC EXPRESSIONS

The followings are 3 languages for algebraic expressions:

Infix Expressions: An operator appears between its operands. $((a+b)*c)$

Prefix Expressions: An operator appears before its operands. $* + a\ b\ c$

Postfix Expressions: An operator appears after its operands. $a\ b + c\ *$

Note: The 3 expressions on the right have the same meaning, but a different syntax.

LANGUAGES & GRAMMARS - EXPRESSIONS 2

CONVERT A FULLY PARENTHESIZED INFIX EXPRESSION TO A PREFIX FORM

1. move each operator to the position of its **corresponding open parenthesis**;
2. remove the parentheses;

CONVERT A FULLY PARENTHESIZED INFIX EXPRESSION TO A POSTFIX FORM

1. move each operator to the position of its **corresponding closing parenthesis**;
2. remove the parentheses.

ADVANTAGES OF PREFIX AND POSTFIX EXPRESSIONS

- they never need: precedence rules, association rules, and parentheses;
- they have: simple grammars, easy recognition and evaluation algorithms.

LANGUAGES & GRAMMARS - EXPRESSIONS 3

PREFIX EXPRESSIONS: GRAMMAR (RECURSIVE)

$$
\begin{aligned}
\langle prefix \rangle &= \langle identifier \rangle \mid \langle operator \rangle \langle prefix \rangle \langle prefix \rangle \\
\langle operator \rangle &= + \mid - \mid * \mid / \\
\langle identifier \rangle &= \mathrm{a} \mid \mathrm{b} \mid \cdots \mid \mathrm{z}
\end{aligned}
$$

PREFIX EXPRESSIONS: RECOGNITION ALGORITHM (RECURSIVE)

See: method **isPrefixExpression** in class **PrefixExpression**.

PREFIX EXPRESSIONS: EVALUATION ALGORITHM (RECURSIVE)

See: method **evalPrefixExpression** in class **PrefixExpression**.

LANGUAGES & GRAMMARS - EXPRESSIONS 4

PREFIX EXPRESSION A

```
import java.lang.String; // Note: String represent an immutable sequence of characters!
import java.lang.StringBuilder; // Better than StringBuffer in single-thread situations!

public class PrefixExpression {

   private String exp; // Stores the prefix expression.
   private float[] ids = new float[26]; // Stores identifier values (from 'a' to 'z').

   // Checks if a character is a valid identifier.
   private boolean isIdentifier( char c ) {
      if( ( c >= 'a' ) && ( c <= 'z' ) ) { return true; }
      return false; }

   // Checks if a character is a valid operator.
   private boolean isOperator( char c ) {
      if( ( c == '+' ) || ( c == '-' ) || ( c == '*' ) || ( c == '/' ) ) { return true; }
      return false; }
```

LANGUAGES & GRAMMARS - EXPRESSIONS 5

PREFIX EXPRESSION B

```
// Finds the end of a prefix expression, if one exists.
// Precondition: exp substring from index first through last with no blank characters.
// Postcondition: returns index of the last char in exp that begins at index first,
//                if it does not exist returns -1.
private int endPrefixExpression( int first, int last ) {
   if( ( first < 0 ) || ( first > last ) ) { return -1; }
   char ch = exp.charAt( first ); // Get character at first position of exp.
   if( isIdentifier( ch ) ) { return first; } // Last char index in simple prefix exp.
   else if( isOperator( ch ) ) {
      // Find the end of the first prefix expression.
      int firstEnd = endPrefixExpression( first + 1, last ); // Label: X.
      // If the end of 1st prefix exp was found, find the end of 2nd prefix exp.
      if( firstEnd > -1 ) {
         return endPrefixExpression( firstEnd + 1, last ); } // Label: Y.
      else { return -1; } }
   else { return -1; } }
```

LANGUAGES & GRAMMARS - EXPRESSIONS 6

PREFIX EXPRESSION C

```
// Default constructor.
public PrefixExpression() { exp = ""; }

// Constructor.
public PrefixExpression( String e ) { exp = e; }

// Determines whether exp is a prefix expression or not.
// Precondition: exp has been initialized with a string containing no blank chars.
// Postcondition: returns true if exp is in the prefix form, otherwise returns false.
public boolean isPrefixExpression() {
   // Get expression string size, and the result of endPrefixExpression.
   int size = exp.length();
   int lastChar = endPrefixExpression( 0, size - 1 );
   // Exploit endPrefixExpression to check if this is a prefix expression.
   if( ( lastChar >= 0 ) && ( lastChar == ( size - 1 ) ) ) { return true; }
   else { return false; } }
```

LANGUAGES & GRAMMARS - EXPRESSIONS 7

PREFIX EXPRESSION D

```
// Getter method for identifiers.
public float getIdentifier( char id ) {
   if( ( id >= 'a' ) && ( id <= 'z' ) ) {
      int index = id - 'a';
      return ids[index]; }
   return 0.0f; } // Return value in case of an error.

// Setter method for identifiers.
public void setIdentifier( char id, float value ) {
   if( ( id >= 'a' ) && ( id <= 'z' ) ) {
      int index = id - 'a';
      ids[index] = value; } }
```

LANGUAGES & GRAMMARS - EXPRESSIONS 8

PREFIX EXPRESSION E

```
// Evaluate the prefix expression e.
// Precondition: e is a prefix expression containing no blank characters.
// Postcondition: returns the value of the prefix expression e.
public float evalPrefixExpression( StringBuilder e ) { // StringBuilder is mutable!
   char ch = e.charAt(0); // Get character at first position of e.
   e.deleteCharAt(0); // We StringBuilder to perform this delete!
   if( isIdentifier( ch ) ) { return getIdentifier( ch ); } // Base case: single id.
   else if( isOperator( ch ) ) { // Recursive calls to retrieve the 2 operands.
      float operand1 = evalPrefixExpression( e ); // Label: X.
      float operand2 = evalPrefixExpression( e ); // Label: Y.
      switch( ch ) { // Apply the proper operator to return the result.
         case '+' : { return ( operand1 + operand2 ); }
         case '-' : { return ( operand1 - operand2 ); }
         case '*' : { return ( operand1 * operand2 ); }
         case '/' : { return ( operand1 / operand2 ); }
         default  : { break; } } }
   return 0.0f; } } // Return value in case of an error.
```

LANGUAGES & GRAMMARS - EXPRESSIONS 9

TEST PREFIX EXPRESSIONS

```
import java.lang.String; // Note: String represent an immutable sequence of characters!
import java.lang.StringBuilder; // Better than StringBuffer in single-thread situations!
public class TestPrefixExpressions {
   public static void main( String[] args ) {
      StringBuilder strExp = new StringBuilder( "+/ab-cd" ); // Set test expression.
      PrefixExpression exp = new PrefixExpression( strExp.toString() ); // Prefix exp.
      boolean res = exp.isPrefixExpression(); // Check if prefix expression is valid.
      exp.setIdentifier( 'a', 1.2f ); // Set the value of identifier a.
      exp.setIdentifier( 'b', 3.4f ); // Set the value of identifier b.
      exp.setIdentifier( 'c', 5.6f ); // Set the value of identifier c.
      exp.setIdentifier( 'd', 7.8f ); // Set the value of identifier d.
      float a = exp.getIdentifier( 'a' ); // Get the value of identifier a.
      float b = exp.getIdentifier( 'b' ); // Get the value of identifier b.
      float c = exp.getIdentifier( 'c' ); // Get the value of identifier c.
      float d = exp.getIdentifier( 'd' ); // Get the value of identifier d.
      float res1 = ( a / b ) + ( c - d ); // Correct result of prefix exp evaluation.
      float res2 = e.evalPrefixExpression( strExp ); } } // Custom prefix exp evaluation.
```

LANGUAGES & GRAMMARS - EXPRESSIONS 10

POSTFIX EXPRESSIONS: GRAMMAR (RECURSIVE)

$$\langle postfix \rangle = \langle identifier \rangle \mid \langle postfix \rangle \langle postfix \rangle \langle operator \rangle$$
$$\langle operator \rangle = + \mid - \mid * \mid /$$
$$\langle identifier \rangle = \mathrm{a} \mid \mathrm{b} \mid \cdots \mid \mathrm{z}$$

PREFIX-TO-POSTFIX EXPRESSION CONVERSION ALGORITHM (RECURSIVE)

```
// Converts prefix expression e to postfix form.
// Precondition: expression e is in valid prefix form.
// Postcondition: returns the equivalent postfix expression.
public StringBuilder convertPrefixToPostfix( StringBuilder e ) { // Need mutable string!
   char ch = e.charAt(0); // Get character at first position of e.
   e.deleteCharAt(0); // We StringBuilder to perform this delete!
   // Check first character of expression e.
   if( isIdentifier( ch ) ) { // Base case: single identifier.
      return new StringBuilder( Character.toString( ch ) ); }
   else { // First character is an operator.
      StringBuilder postfix1 = convertPrefixToPostfix( e ); // Label: X.
      StringBuilder postfix2 = convertPrefixToPostfix( e ); // Label: Y.
      return postfix1.append( postfix2.append( ch ) ); } } // Concatenate operator.
```

LANGUAGES & GRAMMARS - EXPRESSIONS 12

To avoid ambiguity, infix notation normally requires:

- precedence rules,
- rules for association, and
- parentheses.

FULLY PARENTHESIZED EXPRESSIONS

Fully parenthesized expressions do not require: precedence and association rules.

FULLY PARENTHESIZED EXPRESSIONS: GRAMMAR (RECURSIVE)

$$\begin{aligned}\langle infix\rangle &= \langle identifier\rangle \mid (\,\langle infix\rangle\langle operator\rangle\langle infix\rangle\,) \\ \langle operator\rangle &= + \mid - \mid \times \mid / \\ \langle identifier\rangle &= \mathrm{a} \mid \mathrm{b} \mid \cdots \mid \mathrm{z}\end{aligned}$$

RECURSION & INDUCTION - DEFINITION

A strong relationship exists between recursion and mathematical induction.

Recursion: Solves a problem by specifying a solution to one or more base cases, and then demonstrating how to derive the solution to a problem of an arbitrary size from the solutions to smaller problems of the same type.

Mathematical Induction: Proves a property about the natural numbers by proving the property about a base case (e.g. **0** or **1**), and then proving that the property must be true for an arbitrary natural number **n** if it is true for the natural numbers **<n**.

Induction can be used to:

1. prove that a recursive algorithm performs the task correctly,
2. prove that a recursive algorithm performs a certain amount of work.

RECURSION & INDUCTION - CORRECTNESS 1

1 - THE CORRECTNESS OF THE RECURSIVE FACTORIAL METHOD A

The recursive method **fact** that computes the factorial of a non-negative integer **n**:

```
// Recursive method to compute the factorial of a non-negative integer.
public int fact( int n ) {
   if( n == 0 ) { return 1; }
   else { return n * fact( n - 1 ); } }
```

Induction on **n** can prove that the method **fact** returns the values:

$$
\begin{aligned}
&fact(0) = 0! = 1 && \text{if } n = 0 \\
&fact(n) = n! = n \times (n-1) \times (n-2) \times \cdots \times 1 && \text{if } n > 0
\end{aligned}
$$

RECURSION & INDUCTION - CORRECTNESS 2

1 - THE CORRECTNESS OF THE RECURSIVE FACTORIAL METHOD B

Proof: The proof by induction is the following:

Basis: Show that the property is true for **n = 0**.
This is simply the base case of the recursive method: $fact(0) = 0! = 1$

Now prove that: if property is true for an arbitrary **k**, then property is true for **k+1**.

Inductive Hypothesis: Assume that the property is true for **n=k**.
That is, assume: $fact(k) = k * (k-1) * (k-2) * \cdots * 2 * 1$

Inductive Conclusion: Show that the property is true for **n=k+1**.
That is, show that: $fact(k+1) \ =? \ (k+1) * k * (k-1) * (k-2) * \cdots * 2 * 1$

RECURSION & INDUCTION - CORRECTNESS 3

1 - THE CORRECTNESS OF THE RECURSIVE FACTORIAL METHOD C

Proof (continued):

By the definition of method **fact**: $fact(k+1) = (k+1) * fact(k)$

By the inductive hypothesis: $fact(k) = k * (k-1) * (k-2) * \cdots * 2 * 1$

Thus:

$$\begin{aligned} fact(k+1) &= (k+1) * fact(k) = \\ &= (k+1) * [\, k * (k-1) * (k-2) * \cdots * 2 * 1 \,] \end{aligned}$$

The inductive proof is now complete, since we have proven that:

if property is true for an arbitrary k, then property is true for k+1.

RECURSION & INDUCTION - COST 1

2 - THE COST OF TOWERS OF HANOI A

The following is the solution to the Towers of Hanoi problem:

```java
// Class implementing the algorithm to solve the Towers of Hanoi problem.
public class TowersOfHanoiExample {

   // Algorithm to solve the Towers of Hanoi problem.
   public static void solveTowersOfHanoi(int count, char source, char dest, char spare) {
      if( count == 1 ) { // Base case.
         System.out.println("Move top disk from pole " + source + " to pole " + dest); }
      else { // Recursive calls.
         solveTowersOfHanoi( count-1, source, spare, dest ); // Label: a.
         solveTowersOfHanoi( 1, source, dest, spare ); // Label: b.
         solveTowersOfHanoi( count-1, spare, dest, source ); } } // Label: c.

   // Main method to test the solution of the Towers of Hanoi problem.
   public static void main( String[] args ) { solveTowersOfHanoi( 2, 'A', 'B', 'C' ); } }
```

RECURSION & INDUCTION - COST 2

2 - THE COST OF TOWERS OF HANOI B

For **n** disks, how many moves needs **solveTowersOfHanoi** to solve the problem?

Let: **moves(n)** be the number of moves made starting with **n** disks.

$$moves(n) = \begin{cases} 1, & \text{if } n = 1 \\ moves(n-1)_{\mathrm{a}} + moves(1)_{\mathrm{b}} + moves(n-1)_{\mathrm{c}}, & \text{if } n > 1 \end{cases}$$

So the **recurrence relation** for the number of moves required for **n** disks is:

$$moves(n) = \begin{cases} 1, & \text{if } n = 1 \\ (\, 2 \times moves(n-1) \,) + 1, & \text{if } n > 1 \end{cases}$$

RECURSION & INDUCTION - COST 3

2 - THE COST OF TOWERS OF HANOI C

A **closed-form formula** is a mathematical expression that can be evaluated in a finite number of operations (instead of a recursive form), allowing for example to substitute values for variables and obtain directly the final result of the formula.

The closed-form formula of the previous recurrence relation is:

$$moves(n) = 2^n - 1, \qquad \text{for all } \ n \geq 1$$

Induction on **n** can prove the formula above.

RECURSION & INDUCTION - COST 4

2 - THE COST OF TOWERS OF HANOI D

Proof: The proof by induction is the following:

Basis: Show that the property is true for **n = 1**.
So: $moves(1) = 2^1 - 1 = 1$

Now prove that: if property is true for an arbitrary **k**, then property is true for **k+1**.

Inductive Hypothesis: Assume that the property is true for **n=k**.
That is, assume that: $moves(k) = 2^k - 1$

Inductive Conclusion: Show that the property is true for **n=k+1**.
That is, show that: $moves(k+1) \ =?\ 2^{k+1} - 1$

RECURSION & INDUCTION - COST 5

2 - THE COST OF TOWERS OF HANOI E

Proof (continued):

By the recurrence relation: $moves(k+1) = (\ 2 \times moves(k)\ ) + 1$

By the inductive hypothesis: $moves(k) = 2^k - 1$

Thus:

$$moves(k+1) = (\ 2 \times moves(k)\ ) + 1 = \left(\ 2 \times \left(\ 2^k - 1\ \right)\ \right) + 1 =$$
$$= 2^{k+1} - 2 + 1 = 2^{k+1} - 1$$

The inductive proof is now complete, since we have proven that:

if property is true for an arbitrary k, then property is true for k+1.

Kettering
UNIVERSITY