ICOM 4015: Advanced Programming

Lecture 13

Chapter Thirteen: Recursion

Chapter Thirteen: Recursion

Chapter Goals

- To learn about the method of recursion
- To understand the relationship between recursion and iteration
- To analyze problems that are much easier to solve by recursion than by iteration
- To learn to "think recursively"
- To be able to use recursive helper methods
- To understand when the use of recursion affects the efficiency of an algorithm

Triangle Numbers

- Compute the area of a triangle of width *n*
- Assume each [] square has an area of 1
- Also called the *n*th *triangle number*
- The third triangle number is 6

```
[]
[][]
[][][]
```

Outline of `Triangle` Class

```
public class Triangle
{
   public Triangle(int aWidth)
   {
      width = aWidth;
   }
   public int getArea()
   {
      . . .
   }
   private int width;
}
```

Handling Triangle of Width 1

- The triangle consists of a single square
- Its area is 1
- Add the code `togetArea` method for width 1

```
public int getArea()
{
    if (width == 1) return 1;
    . . .
}
```

Handling the General Case

- Assume we know the area of the smaller, colored triangle

```
[]
[][]
[][][]
[][][][]
```

- Area of larger triangle can be calculated as:

```
smallerArea + width
```

- To get the area of the smaller triangle
  - *Make a smaller triangle and ask it for its area*

```
Triangle smallerTriangle = new Triangle(width - 1);
int smallerArea = smallerTriangle.getArea();
```

Completed `getArea` Method

```
public int getArea()
{
    if (width == 1) return 1;
    Triangle smallerTriangle = new Triangle(width - 1);
    int smallerArea = smallerTriangle.getArea();
    return smallerArea + width;
}
```

Computing the area of a triangle with width 4

- `getArea` method makes a smaller triangle of width 3
  - It calls `getArea` on that triangle
    - That method makes a smaller triangle of width 2
      - It calls `getArea` on that triangle
        - That method makes a smaller triangle of width 1
        - It calls `getArea` on that triangle
      - That method returns 1
    - The method returns `smallerArea + width = 1 + 2 = 3`
  - The method returns `smallerArea + width = 3 + 3 = 6`
- The method returns `smallerArea + width = 6 + 4 = 10`

Recursion

- A recursive computation solves a problem by using the solution of the same problem with simpler values
- For recursion to terminate,
  there must be special cases for the simplest inputs.
- To complete our Triangle example, we must handle width <= 0

```
if (width <= 0)  return 0;
```

- Two key requirements for recursion success:
  - *Every recursive call must simplify the computation in some way*
  - *There must be special cases to handle the simplest computations directly*

Other Ways to Compute Triangle Numbers

- The area of a triangle equals the sum

  `1 + 2 + 3 + . . . + width`

- Using a simple loop:

```
double area = 0;
for (int i = 1; i <= width; i++)
   area = area + i;
```

- Using math:

```
1 + 2 + . . . + n = n × (n + 1)/2
      => width * (width + 1) / 2
```

Animation 13.1 –

```
public static void main(String[] args)
{
   Triangle t = new Triangle(3);
   int area = t.getArea();
   System.out.println("Area: " + area);
}
. . .
public int getArea()
{
   if (width == 1) return 1;
   Triangle smallerTriangle = new Triangle(width - 1);
   int smallerArea = smallerTriangle.getArea();
   return smallerArea + width;
}
```

This animation demonstrates the recursive computation of the area of a `Triangle` object.

13-01 Recursion

ch13/triangle/Triangle.java

```java
/**
   A triangular shape composed of stacked unit squares like this:
   []
   [][]
   [][][]
   . . .
*/
public class Triangle
{
   /**
      Constructs a triangular shape.
      @param aWidth the width (and height) of the triangle
   */
   public Triangle(int aWidth)
   {
      width = aWidth;
   }

   /**
      Computes the area of the triangle.
      @return the area
   */
```

Continued

ch13/triangle/TriangleTester.java

```java
   public int getArea()
   {
      if (width <= 0) return 0;
      if (width == 1) return 1;
      Triangle smallerTriangle = new Triangle(width - 1);
      int smallerArea = smallerTriangle.getArea();
      return smallerArea + width;
   }

   private int width;
}
```

Output:

```
Enter width: 10
Area: 55
Expected: 55
```

Self Check 13.1

Why is the statement `if (width == 1) return 1;` in the `getArea` method unnecessary?

Answer: Suppose we omit the statement. When computing the area of a triangle with width 1, we compute the area of the triangle with width 0 as 0, and then add 1, to arrive at the correct area.

Self Check 13.2

How would you modify the program to recursively compute the area of a square?

Answer: You would compute the smaller area recursively, then return `smallerArea + width + width - 1`.

```
[][][][]
[][][][]
[][][][]
[][][][]
```

Of course, it would be simpler to compute

$$1+0+2+1+3+2+\cdots+n+n-1=\frac{n(n+1)}{2}+\frac{(n-1)n}{2}=n^2.$$

Permutations

- Design a class that will list all permutations of a string
- A permutation is a rearrangement of the letters
- The string `"eat"` has six permutations:

```
"eat"
"eta"
"aet"
"tea"
"tae"
```

Public Interface of `PermutationGenerator`

```
public class PermutationGenerator
{
   public PermutationGenerator(String aWord) { . . . }
   ArrayList<String> getPermutations() { . . . }
}
```

ch13/permute/PermutationGeneratorDemo.java

```java
import java.util.ArrayList;

/**
   This program demonstrates the permutation generator.
*/
public class PermutationGeneratorDemo
{
   public static void main(String[] args)
   {
      PermutationGenerator generator
            = new PermutationGenerator("eat");
      ArrayList<String> permutations = generator.getPermutations();
      for (String s : permutations)
      {
         System.out.println(s);
      }
   }
}

```

ch13/permute/PermutationGeneratorDemo.java (cont.)

Output:

```
eat
eta
aet
ate
tea
tae
```

To Generate All Permutations

- Generate all permutations that start with `'e'` , then `'a'` then `'t'`
- To generate permutations starting with `'e'`, we need to find all permutations of `"at"`
- This is the same problem with simpler inputs
- Use recursion

To Generate All Permutations

- `getPermutations`: loop through all positions in the word to be permuted
- For each position, compute the shorter word obtained by removing ith letter:

```
String shorterWord = word.substring(0, i) +
      word.substring(i + 1);
```

- Construct a permutation generator to get permutations of the shorter word

```
PermutationGenerator shorterPermutationGenerator
      = new PermutationGenerator(shorterWord);
ArrayList<String> shorterWordPermutations
      = shorterPermutationGenerator.getPermutations();
```

To Generate All Permutations

- Finally, add the removed letter to front of all permutations of the shorter word

```
for (String s : shorterWordPermutations)
{
   result.add(word.charAt(i) + s);
}
```

- Special case: simplest possible string is the empty string; single permutation, itself

ch13/permute/PermutationGenerator.java

```
import java.util.ArrayList;

/**
   This class generates permutations of a word.
*/
public class PermutationGenerator
{
   /**
      Constructs a permutation generator.
      @param aWord the word to permute
   */
   public PermutationGenerator(String aWord)
   {
      word = aWord;
   }

   /**
      Gets all permutations of a given word.
   */
   public ArrayList<String> getPermutations()
   {
```

Continued

ch13/permute/PermutationGenerator.java (cont.)

```
      ArrayList<String> result = new ArrayList<String>();

      // The empty string has a single permutation: itself
      if (word.length() == 0)
      {
         result.add(word);
         return result;
      }

      // Loop through all character positions
      for (int i = 0; i < word.length(); i++)
      {
         // Form a simpler word by removing the ith character
         String shorterWord = word.substring(0, i)
               + word.substring(i + 1);

         // Generate all permutations of the simpler word
         PermutationGenerator shorterPermutationGenerator
               = new PermutationGenerator(shorterWord);
         ArrayList<String> shorterWordPermutations
               = shorterPermutationGenerator.getPermutations();

```

Continued

ch13/permute/PermutationGenerator.java (cont.)

```
            // Add the removed character to the front of
            // each permutation of the simpler word,
            for (String s : shorterWordPermutations)
            {
               result.add(word.charAt(i) + s);
            }
        }
        // Return all permutations
        return result;
    }

    private String word;
}
```

Self Check 13.3

What are all permutations of the four-letter word `beat`?

> **Answer:** They are b followed by the six permutations of `eat`, `e` followed by the six permutations of `bat`, `a` followed by the six permutations of `bet`, and `t` followed by the six permutations of `bea`.

Self Check 13.4

Our recursion for the permutation generator stops at the empty string. What simple modification would make the recursion stop at strings of length 0 or 1?

Answer: Simply change `if (word.length() == 0)` to `if (word.length() <= 1)`, because a word with a single letter is also its sole permutation.

Tracing Through Recursive Methods

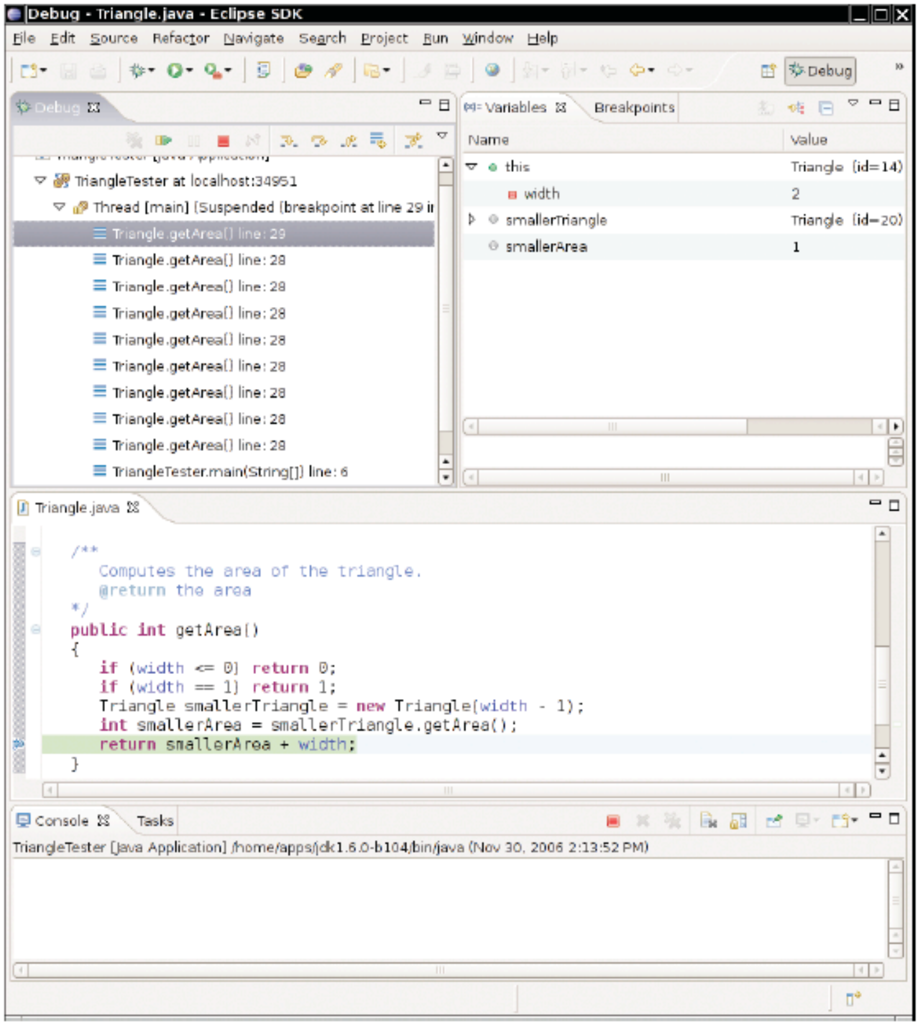


Figure 1 Debugging a Recursive Method

Thinking Recursively

- Problem: test whether a sentence is a palindrome
- Palindrome: a string that is equal to itself when you reverse all characters
  - *A man, a plan, a canal – Panama!*
  - *Go hang a salami, I'm a lasagna hog*
  - *Madam, I'm Adam*

Implement `isPalindrome` Method

```
public class Sentence
{
   /**
      Constructs a sentence.
      @param aText a string containing all characters of
            the sentence
   */
   public Sentence(String aText)
   {
      text = aText;
   }

   /**
      Tests whether this sentence is a palindrome.
      @return true if this sentence is a palindrome, false
            otherwise
   */
```

Continued

Implement `isPalindrome` Method (cont.)

```
public boolean isPalindrome()
   {
      . . .
   }
   private String text;
}
```

Thinking Recursively: Step-by-Step

1. Consider various ways to simplify inputs

    - Here are several possibilities:
    - Remove the first character
    - Remove the last character
    - Remove both the first and last characters
    - Remove a character from the middle
    - Cut the string into two halves

Thinking Recursively: Step-by-Step

2. Combine solutions with simpler inputs into a solution of the original problem

- Most promising simplification: remove first and last characters "adam, I'm Ada", is a palindrome too!
- Thus, a word is a palindrome if
  - *The first and last letters match, and*
  - *Word obtained by removing the first and last letters is a palindrome*
- What if first or last character is not a letter? Ignore it
  - *If the first and last characters are letters, check whether they match; if so, remove both and test shorter string*
  - *If last character isn't a letter, remove it and test shorter string*
  - *If first character isn't a letter, remove it and test shorter string*

Thinking Recursively: Step-by-Step

3. Find solutions to the simplest inputs

- Strings with two characters
  - *No special case required; step two still applies*
- Strings with a single character
  - *They are palindromes*
- The empty string
  - *It is a palindrome*

Thinking Recursively: Step-by-Step

4. Implement the solution by combining the simple cases and the reduction step

```
public boolean isPalindrome()
{
int length = text.length();
// Separate case for shortest strings.
if (length <= 1) return true;
// Get first and last characters, converted to
      lowercase.
char first = Character.toLowerCase(text.charAt(0));
char last = Character.toLowerCase(text.charAt(length -
1));
if (Character.isLetter(first) &&
      Character.isLetter(last))
{
   // Both are letters.
   if (first == last)
   {
```

Continued

Thinking Recursively: Step-by-Step (cont.)

```
         // Remove both first and last character.
         Sentence shorter = new Sentence(text.substring(1,
               length - 1));
         return shorter.isPalindrome();
      }
      else
         return false;
      }
      else if (!Character.isLetter(last))
      {
         // Remove last character.
         Sentence shorter = new Sentence(text.substring(0,
               length - 1));
         return shorter.isPalindrome();
      }
      else
      {
```

Continued

Thinking Recursively: Step-by-Step (cont.)

```
         // Remove first character.
        Sentence shorter = new
              Sentence(text.substring(1));
        return shorter.isPalindrome();
    }
}
```

Recursive Helper Methods

- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem
- Consider the palindrome test of previous slide
  It is a bit inefficient to construct new `Sentence` objects in every step

Continued

Recursive Helper Methods (cont.)

- Rather than testing whether the sentence is a palindrome, check whether a substring is a palindrome:

```
/**
   Tests whether a substring of the sentence is a
         palindrome.
   @param start the index of the first character of the
         substring
   @param end the index of the last character of the
         substring
   @return true if the substring is a palindrome
*/
public boolean isPalindrome(int start, int end)
```

Recursive Helper Methods

- Then, simply call the helper method with positions that test the entire string:

```
public boolean isPalindrome()
{
   return isPalindrome(0, text.length() - 1);
}
```

Recursive Helper Methods: `isPalindrome`

```
public boolean isPalindrome(int start, int end)
{
   // Separate case for substrings of length 0 and 1.
   if (start >= end) return true;
   // Get first and last characters, converted to
         lowercase.
   char first = Character.toLowerCase(text.charAt(start));
   char last = Character.toLowerCase(text.charAt(end));
   if (Character.isLetter(first) &&
         Character.isLetter(last))
   {
      if (first == last)
      {
         // Test substring that doesn't contain the
               matching letters.
         return isPalindrome(start + 1, end - 1);
      }
      else return false;
```

Continued

Recursive Helper Methods: `isPalindrome` (cont.)

```
    }
    else if (!Character.isLetter(last))
    {
       // Test substring that doesn't contain the last
             character.
       return isPalindrome(start, end - 1);
    }
    else
    {
       // Test substring that doesn't contain the first
             character.
       return isPalindrome(start + 1, end);
    }
}
```

Self Check 13.5

Do we have to give the same name to both `isPalindrome` methods?

Answer: No–the first one could be given a different name such as `substringIsPalindrome`.

Self Check 13.6

When does the recursive `isPalindrome` method stop calling itself?

Answer: When `start >= end`, that is, when the investigated string is either empty or has length 1.

Fibonacci Sequence

- Fibonacci sequence is a sequence of numbers defined by
  $f1 = 1$
  $f2 = 1$
  $fn = fn\text{-}1 + fn\text{-}2$
- First ten terms:
  1, 1, 2, 3, 5, 8, 13, 21, 34, 55

ch13/fib/RecursiveFib.java

```java
import java.util.Scanner;

/**
   This program computes Fibonacci numbers using a recursive
   method.
*/
public class RecursiveFib
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter n: ");
      int n = in.nextInt();

      for (int i = 1; i <= n; i++)
      {
         long f = fib(i);
         System.out.println("fib(" + i + ") = " + f);
      }
   }

```

Continued

ch13/fib/RecursiveFib.java (cont.)

```
   /**
      Computes a Fibonacci number.
      @param n an integer
      @return the nth Fibonacci number
   */
   public static long fib(int n)
   {
      if (n <= 2) return 1;
      else return fib(n - 1) + fib(n - 2);
   }
}
```

ch13/fib/RecursiveFib.java (cont.)

Output:

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

The Efficiency of Recursion

- Recursive implementation of `fib` is straightforward
- Watch the output closely as you run the test program
- First few calls to `fib` are quite fast
- For larger values, the program pauses an amazingly long time between outputs
- To find out the problem, lets insert trace messages

ch13/fib/RecursiveFibTracer.java

```java
import java.util.Scanner;

/**
   This program prints trace messages that show how often the
   recursive method for computing Fibonacci numbers calls itself.
*/
public class RecursiveFibTracer
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter n: ");
      int n = in.nextInt();

      long f = fib(n);

      System.out.println("fib(" + n + ") = " + f);
   }

```

Continued

ch13/fib/RecursiveFibTracer.java (cont,)

```
   /**
      Computes a Fibonacci number.
      @param n an integer
      @return the nth Fibonacci number
   */
   public static long fib(int n)
   {
      System.out.println("Entering fib: n = " + n);
      long f;
      if (n <= 2) f = 1;
      else f = fib(n - 1) + fib(n - 2);
      System.out.println("Exiting fib: n = " + n
            + " return value = " + f);
      return f;
   }
}
```

ch13/fib/RecursiveFibTracer.java (cont,)

Output:

Lauren – I'm not sure what's supposed to go here.

Call Tree for Computing `fib(6)`

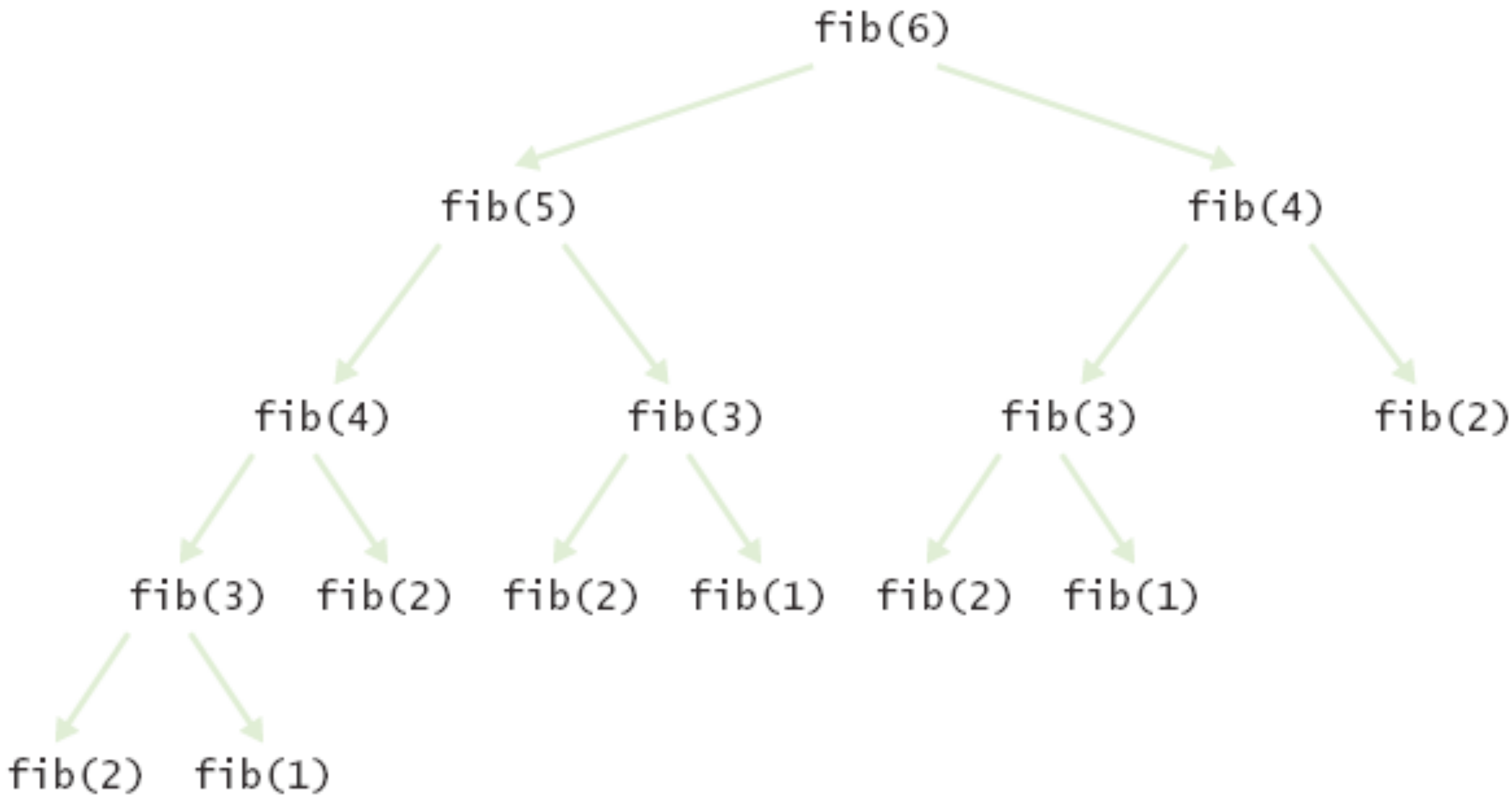


Figure 2 Call Pattern of the Recursive fib Method

The Efficiency of Recursion

- Method takes so long because it computes the same values over and over
- The computation of `fib(6)` calls `fib(3)` three times
- Imitate the pencil-and-paper process to avoid computing the values more than once

ch13/fib/LoopFib.java

```java
import java.util.Scanner;

/**
   This program computes Fibonacci numbers using an iterative method.
*/
public class LoopFib
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter n: ");
      int n = in.nextInt();

      for (int i = 1; i <= n; i++)
      {
         long f = fib(i);
         System.out.println("fib(" + i + ") = " + f);
      }
   }

```

Continued

ch13/fib/LoopFib.java (cont.)

```java
    /**
       Computes a Fibonacci number.
       @param n an integer
       @return the nth Fibonacci number
    */
    public static long fib(int n)
    {
       if (n <= 2) return 1;
       long fold = 1;
       long fold2 = 1;
       long fnew = 1;
       for (int i = 3; i <= n; i++)
       {
          fnew = fold + fold2;
          fold2 = fold;
          fold = fnew;
       }
       return fnew;
    }
}
```

ch13/fib/LoopFib.java (cont.)

Output:

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

The Efficiency of Recursion

- Occasionally, a recursive solution runs much slower than its iterative counterpart
- In most cases, the recursive solution is only slightly slower
- The iterative `isPalindrome` performs only slightly better than recursive solution
  - *Each recursive method call takes a certain amount of processor time*
- Smart compilers can avoid recursive method calls if they follow simple patterns
- Most compilers don't do that
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution
- "To iterate is human, to recurse divine.", L. Peter Deutsch

Iterative `isPalindrome` Method

```
public boolean isPalindrome()
{
   int start = 0;
   int end = text.length() - 1;
   while (start < end)
{
      char first =
            Character.toLowerCase(text.charAt(start));
      char last = Character.toLowerCase(text.charAt(end);
      if (Character.isLetter(first) &&
            Character.isLetter(last))
      {
         // Both are letters.
         if (first == last)
         {
            start++;
            end--;
         }
```

Continued

Iterative `isPalindrome` Method (cont.)

```
            else
               return false;
         }
         if (!Character.isLetter(last))
            end--;
         if (!Character.isLetter(first))
            start++;
      }
      return true;
   }
```

Self Check 13.7

You can compute the factorial function either with a loop, using the definition that $n! = 1 \times 2 \times \ldots \times n$, or recursively, using the definition that $0! = 1$ and $n! = (n - 1)! \times n$. Is the recursive approach inefficient in this case?

Answer: No, the recursive solution is about as efficient as the iterative approach. Both require $n - 1$ multiplications to compute $n!$.

Self Check 13.8

Why isn't it easy to develop an iterative solution for the permutation generator?

Answer: An iterative solution would have a loop whose body computes the next permutation from the previous ones. But there is no obvious mechanism for getting the next permutation. For example, if you already found permutations `eat`, `eta`, and `aet`, it is not clear how you use that information to get the next permutation. Actually, there is an ingenious mechanism for doing just that, but it is far from obvious–see Exercise P13.12.

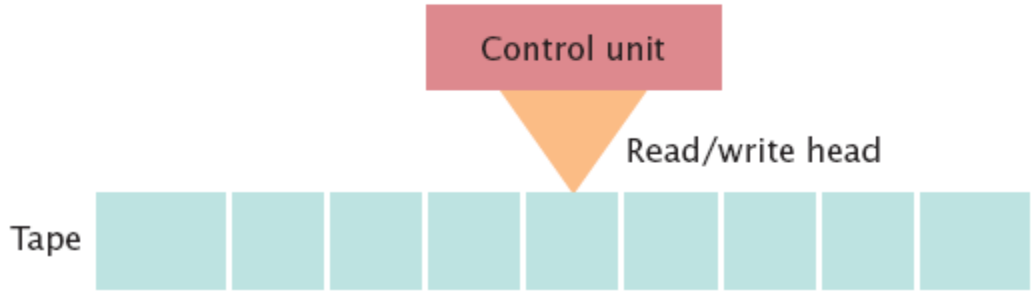
The Limits of Computation

Alan Turing

The Limits of Computation

Program

Instruction number	If tape symbol is	Replace with	Then move head	Then go to instruction
1	0	2	right	2
1	1	1	left	4
2	0	0	right	2
2	1	1	right	2
2	2	0	left	3
3	0	0	left	3
3	1	1	left	3
3	2	2	right	1
4	1	1	right	5
4	2	0	left	4



A Turing Machine

Using Mutual Recursions

- **Problem:** to compute the value of arithmetic expressions such as

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

- Computing expression is complicated
  - ** and / bind more strongly than + and −*
  - *parentheses can be used to group subexpressions*

Syntax Diagram for Evaluating an Expression

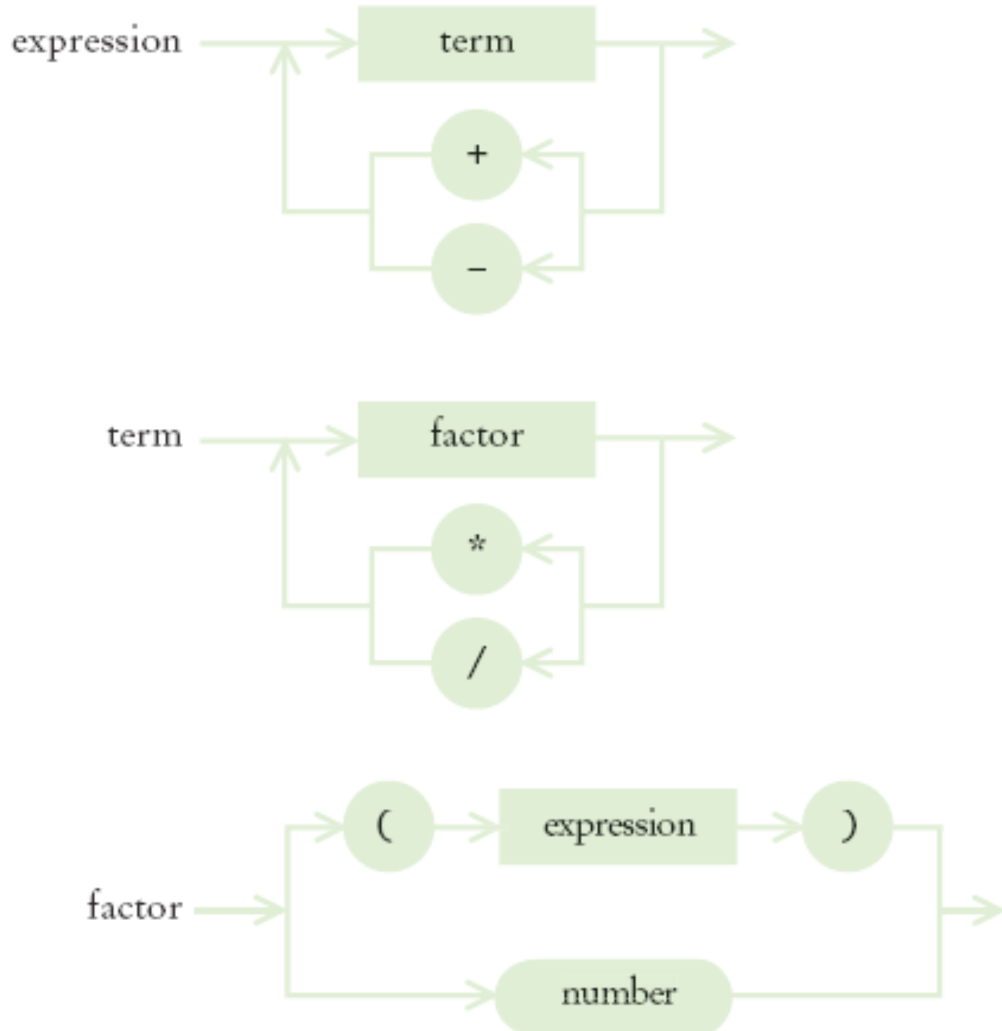


Figure 3 Syntax Diagrams for Evaluating an Expression

Using Mutual Recursions

- An expression can broken down into a sequence of terms, separated by + or -
- Each term is broken down into a sequence of factors, separated by * or /
- Each factor is either a parenthesized expression or a number
- The syntax trees represent which operations should be carried out first

Syntax Tree for Two Expressions

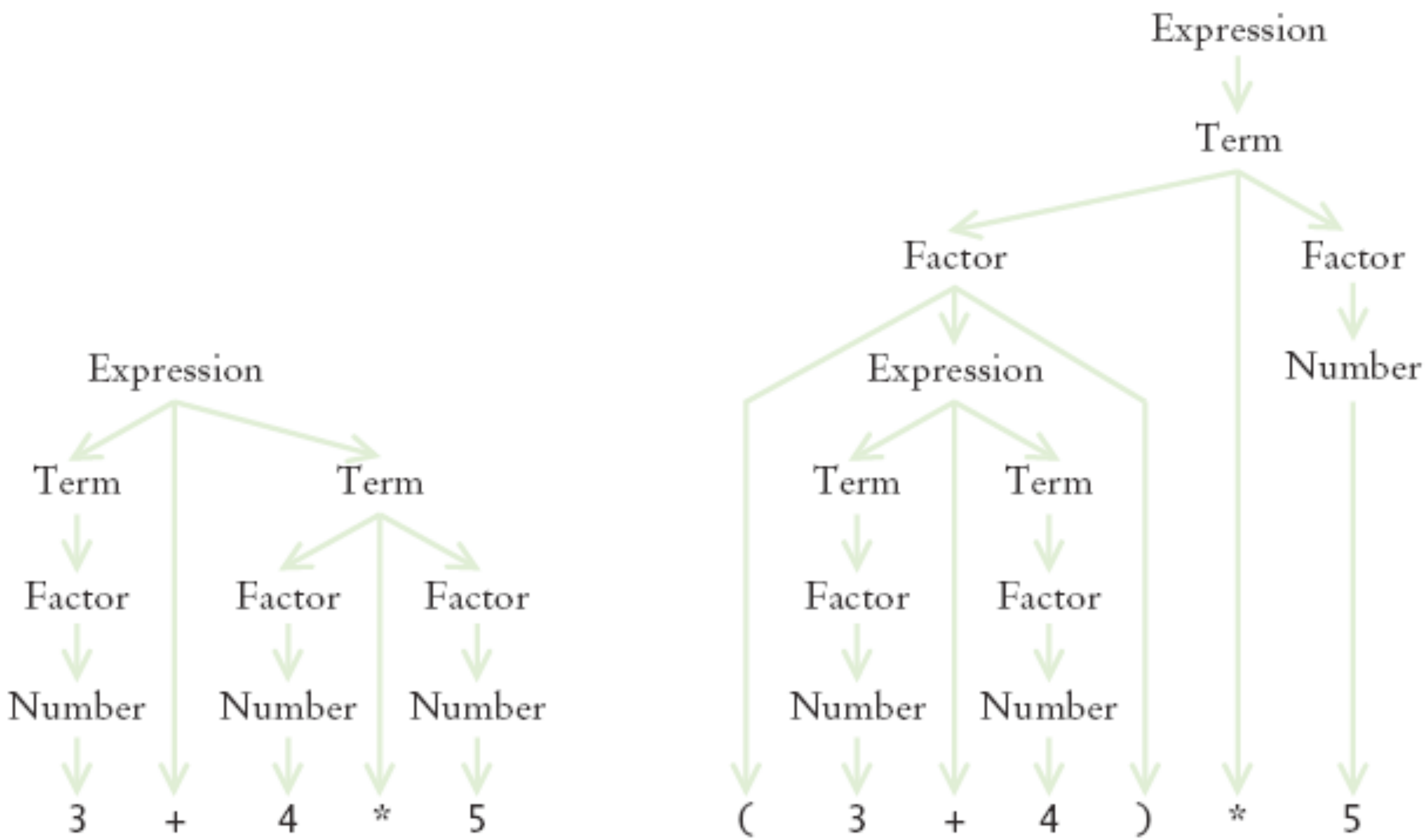


Figure 4 Syntax Trees for Two Expressions

Mutually Recursive Methods

- In a mutual recursion, a set of cooperating methods calls each other repeatedly
- To compute the value of an expression, implement 3 methods that call each other recursively
  - `getExpressionValue`
  - `getTermValue`
  - `getFactorValue`

The `getExpressionValue` Method

```
public int getExpressionValue()
{
   int value = getTermValue();
   boolean done = false;
   while (!done)
   {
      String next = tokenizer.peekToken();
      if ("+".equals(next) || "-".equals(next))
      {
         tokenizer.nextToken(); // Discard "+" or "-"
         int value2 = getTermValue();
         if ("+".equals(next)) value = value + value2;
         else value = value - value2;
      }
      else done = true;
   }
   return value;
}
```

The `getFactorValue` Method

```
public int getFactorValue()
{
   int value;
   String next =
   tokenpublic int getFactorValue()
{
   int value;
   String next = tokenizer.peekToken();
   if ("(".equals(next))
   {
      tokenizer.nextToken(); // Discard "("
      value = getExpressionValue();
      tokenizer.nextToken(); // Discard ")"
   }
   else
      value = Integer.parseInt(tokenizer.nextToken());
   return value;
}
```

Using Mutual Recursions

To see the mutual recursion clearly, trace through the expression `(3+4)*5`:

- `getExpressionValue` calls `getTermValue`
  - `getTermValue` calls `getFactorValue`
    - `getFactorValue` consumes the `(` input
    - `getFactorValue` calls `getExpressionValue`
      - `getExpressionValue` returns eventually with the value of 7, having consumed 3 + 4. This is the recursive call.
    - `getFactorValue` consumes the `)` input
    - `getFactorValue` returns `7`
  - `getTermValue` consumes the inputs * and 5 and returns 35
- `getExpressionValue` returns `35`

ch13/expr/Evaluator.java

```java
/**
   A class that can compute the value of an arithmetic expression.
*/
public class Evaluator
{
   /**
      Constructs an evaluator.
      @param anExpression a string containing the expression
      to be evaluated
   */
   public Evaluator(String anExpression)
   {
      tokenizer = new ExpressionTokenizer(anExpression);
   }

   /**
      Evaluates the expression.
      @return the value of the expression.
   */
   public int getExpressionValue()
   {
```

Continued

ch13/expr/Evaluator.java (cont.)

```
      int value = getTermValue();
      boolean done = false;
      while (!done)
      {
         String next = tokenizer.peekToken();
         if ("+".equals(next) || "-".equals(next))
         {
            tokenizer.nextToken(); // Discard "+" or "-"
            int value2 = getTermValue();
            if ("+".equals(next)) value = value + value2;
            else value = value - value2;
         }
         else done = true;
      }
      return value;
   }

   /**
      Evaluates the next term found in the expression.
      @return the value of the term
   */
```

Continued

ch13/expr/Evaluator.java (cont.)

```
   public int getTermValue()
   {
      int value = getFactorValue();
      boolean done = false;
      while (!done)
      {
         String next = tokenizer.peekToken();
         if ("*".equals(next) || "/".equals(next))
         {
            tokenizer.nextToken();
            int value2 = getFactorValue();
            if ("*".equals(next)) value = value * value2;
            else value = value / value2;
         }
         else done = true;
      }
      return value;
   }

```

Continued

ch13/expr/Evaluator.java (cont.)

```java
    /**
       Evaluates the next factor found in the expression.
       @return the value of the factor
    */
    public int getFactorValue()
    {
       int value;
       String next = tokenizer.peekToken();
       if ("(".equals(next))
       {
          tokenizer.nextToken(); // Discard "("
          value = getExpressionValue();
          tokenizer.nextToken(); // Discard ")"
       }
       else
          value = Integer.parseInt(tokenizer.nextToken());
       return value;
    }

    private ExpressionTokenizer tokenizer;
}
```

ch13/expr/ExpressionTokenizer.java

```java
/**
   This class breaks up a string describing an expression
   into tokens: numbers, parentheses, and operators.
*/
public class ExpressionTokenizer
{
   /**
      Constructs a tokenizer.
      @param anInput the string to tokenize
   */
   public ExpressionTokenizer(String anInput)
   {
      input = anInput;
      start = 0;
      end = 0;
      nextToken();
   }

```

Continued

ch13/expr/ExpressionTokenizer.java (cont.)

```java
   /**
      Peeks at the next token without consuming it.
      @return the next token or null if there are no more tokens
   */
   public String peekToken()
   {
      if (start >= input.length()) return null;
      else return input.substring(start, end);
   }

   /**
      Gets the next token and moves the tokenizer to the following token.
      @return the next token or null if there are no more tokens
   */
   public String nextToken()
   {
      String r = peekToken();
      start = end;
      if (start >= input.length()) return r;
      if (Character.isDigit(input.charAt(start)))
```

Continued

ch13/expr/ExpressionTokenizer.java (cont.)

```
         {
            end = start + 1;
            while (end < input.length()
                  && Character.isDigit(input.charAt(end)))
               end++;
         }
         else
            end = start + 1;
         return r;
      }

   private String input;
   private int start;
   private int end;
}
```

ch13/expr/ExpressionCalculator.java

```java
import java.util.Scanner;

/**
   This program calculates the value of an expression
   consisting of numbers, arithmetic operators, and parentheses.
*/
public class ExpressionCalculator
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter an expression: ");
      String input = in.nextLine();
      Evaluator e = new Evaluator(input);
      int value = e.getExpressionValue();
      System.out.println(input + "=" + value);
   }
}
```

ch13/expr/ExpressionCalculator.java (cont.)

Output:

```
Enter an expression: 3+4*5
3+4*5=23
```

Self Check 13.9

What is the difference between a term and a factor? Why do we need both concepts?

Answer: Factors are combined by multiplicative operators (`*` and `/`), terms are combined by additive operators (`+`, `-`). We need both so that multiplication can bind more strongly than addition.

Self Check 13.10

Why does the expression parser use mutual recursion?

Answer: To handle parenthesized expressions, such as `2+3*(4+5)`. The subexpression `4+5` is handled by a recursive call to `getExpressionValue`.

Self Check 13.11

What happens if you try to parse the illegal expression `3+4*)5`? Specifically, which method throws an exception?

Answer: The `Integer.parseInt` call in `getFactorValue` throws an exception when it is given the string `")"`.