Ý tưởng Đệ Quy

Đệ quy là gì?

- Một đối tượng được mô tả thông qua chính nó được gọi là mô tả đệ quy.
- Một bài toán mang tính chất đệ quy khi nó có thể được phân rã thành các bài toán nhỏ hơn nhưng mang cùng tính chất với bài toán ban đầu

Hàm đệ quy

- Một hàm được gọi là đệ quy khi nó gọi chính nó trong thân hàm → Rule
- Anchor or ground case → degenerate case

```
1. void Recusion()
2. {
3.     Recusion();
4. }
```


Thành phần của một hàm đệ quy

- Hàm đệ quy gồm 2 phần:
  - Phần cơ sở: Điều kiện thoát khỏi đệ quy
  - Phần đệ quy: Thân hàm có chứa lời gọi đệ quy

Thiết kế giải thuật đệ quy

- Thực hiện 3 bước sau:
  - Tham số hóa bài toán
  - Phân tích trường hợp chung: Đưa bài toán về bài toán nhỏ hơn cùng loại, dần dần tiến tới trường hợp suy biến → Phân rã bài toán tổng quát theo phương thức đệ quy → RECURSIVE RULE
  - Tìm trường hợp suy biến

Ưu và nhược điểm

- Biểu diễn bài toán, đồng thời làm gọn chương trình
- Không tối ưu về mặt thời gian (so với sử dụng vòng lặp), gây tốn bộ nhớ.

Một số loại đệ quy

- **Đệ quy tuyến tính (Linear Recursion)**
- **Đệ quy nhị phân (Binary Recursion)**
- **Đệ quy lồng (Nested Recursion)**
- **Đệ quy hỗ tương (Mutual Recursion)**
- **Quay lui (Backtracking)**

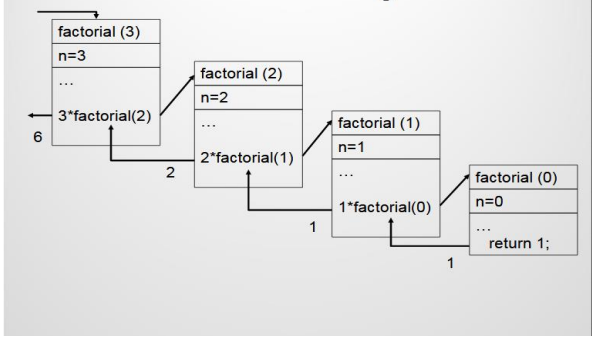
Linear Recursion

- Mỗi lần thực thi chỉ gọi đệ quy một lần

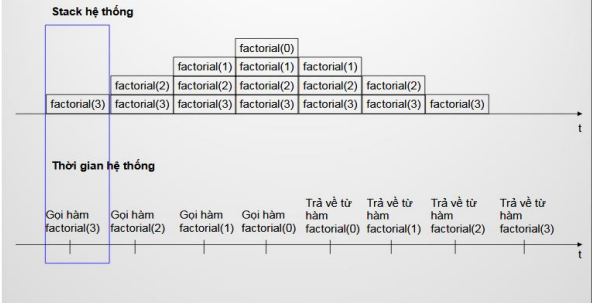
```
KieuDuLieu TenHam(Thamso)
{
   if(Dieu Kien Dung)
   {
      ...;
      return Gia tri tra ve;
   }
   ...;
   TenHam(Thamso)
   ...;
}
```

```
1. int Factorial(int n)
2. {
3.     if (n == 0)
4.     {
5.         return 1;
6.     }
7.     else
8.     {
9.         return n * Factorial(n - 1); // Linear Recursion
10.    }
11. }
```

Thi hành hàm tính giai thừa



Trạng thái hệ thống khi thi hành hàm tính giai thừa



Example

Tính S(n) = 1/(1*2) + 1/(2*3) + … + 1/(n*(n+1))

Binary Recursion

- Mỗi lần thực thi có thể gọi đệ quy 2 lần

```
KieuDuLieu TenHam(Thamso)
{
   if(Dieu Kien Dung)
   {
      …;
      return Gia tri tra ve;
   }
   …;
   TenHam(Thamso);
    …;
   TenHam(Thamso);
     …;
}
```

Ví dụ: Hàm FIBO(n) tính số hạng n của dãy FIBONACCI

```
int F(int n) {
    if ( n < 2 ) return 1;
    else
        return (F(n -1) + F(n -2));
}
```

Nested Recursion

- Tham số trong lời gọi đệ quy là một lời gọi đệ quy.
- Đệ quy lồng chiếm bộ nhớ rất nhanh
- (Đặc biệt) là đệ quy trực tiếp mà lời gọi đệ quy được thực hiện bên trong vòng lặp.

```
KieuDuLieu TenHam(Thamso)
{
   if(Dieu Kien Dung)
   {
      ...;
      return Gia tri tra ve;
   }
   ...;
   vonglap(dieu kien lap)
   {
      ...TenHam(Thamso)...;
   }
   return Gia tri tra ve;
}
```

14

Example

Ví dụ: Cho dãy { An } xác định theo công thức truy hồi :

$A_0 = 1$;

$A_n = n^2A_0+(n-1)^2A_1+ \ldots + 2^2A_{n-2}+ 1^2A_{n-1}$

Mutual Recursion

- Các hàm gọi đệ quy lẫn nhau

```
KieuDuLieu TenHamX(Thamso)
{
    if(Dieu Kien Dung)
    {
        ...;
        return Gia tri tra ve;
    }
    ...;
    return TenHamX(Thamso) <Lien
ket hai ham> TenHamY(Thamso);
}
```

```
KieuDuLieu TenHamY(Thamso)
{
    if(Dieu Kien Dung)
    {
        ...;
        return Gia tri tra ve;
    }
    ...;
    return TenHamY(Thamso)<Lien
ket hai ham>TenHamX(Thamso);
}
```


Example

- Xét tính chẵn lẻ của một số nguyên dương bằng đệ quy.

```
1.  bool isEven(unsigned int n)
2.  {
3.      if (n == 0)
4.      {
5.          return true;
6.      }
7.      else
8.      {
9.          return isOdd(n - 1);
10.     }
11. }
12.
13. bool isOdd(unsigned int n)
14. {
15.     if (n == 1)
16.     {
17.         return true;
18.     }
19.     else
20.     {
21.         return isEven(n - 1);
22.     }
23. }
```

Example

```
long  X(int n) {
   if(n==0)
      return 1;
   else
      return X(n-1) + Y(n-
1);
}

long Y(int n) {
   if(n==0)
      return 1;
   else
      return X(n-1)*Y(n-1);
}
```

```
void main(){
   int n;
   printf("\n Nhap n = ");
   scanf("%d",&n);
   printf( "\n  X = %d " ,X(n));
   printf( "\n Y = %d " ,Y(n));
   getch();
}
```


Exercise

- 1. Tìm ước chung lớn nhất

•Giải thuật đệ quy

```
int USCLN(int m , int n) {
    if (n == 0) return m;
    else USCLN(n, m % n);
}
```


Exercise

- In ra chuỗi giá trị đổi 1 số nguyên không âm Y ở cơ số 10 sang dạng cơ số k (2 <= k <= 9)

```
public static void Convert(int n, int k){
    if(n ==0 ) return;
    Convert(n/k, k);
    System.out.print(n%k);
}
```


Exercise

Luyện tập đệ quy

- Tính tổng sau: $S = 1^2+2^2+\ldots\ldots+n^2$ bằng đệ quy
- Đếm số lượng số chữ số của số nguyên dương bằng đệ quy
- Cho mảng một chiều các số nguyên. Viết hàm tính tổng các số chẵn trong mảng bằng đệ quy
- Cho mảng các số thực. Viết hàm đếm số lượng giá trị dương bằng đệ quy
- Viết hàm đệ quy xuất các giá trị của mảng các số nguyên
- Viết hàm đệ quy xuất các giá trị của mảng các số nguyên theo thứ tự từ trái qua phải (xuất ngược)
- Viết hàm đệ quy đếm số lượng giá trị phân biệt trong mảng các số nguyên
- Viết hàm đệ quy tính tích các giá trị lớn hơn giá trị đứng trước nó trong mảng
- Viết hàm đệ quy kiểm tra mảng các số thực có phải toàn số âm không
- Viết hàm đệ quy tìm số lớn nhất trong mảng các số thực
- Viết hàm đệ quy tìm vị trí mà có giá trị là bé nhất trong mảng các số thực
- Viết hàm đệ quy sắp xếp mảng tăng dần
- Viết hàm đệ quy săp xếp các giá trị chẵn trong mảng các số nguyên tăng dần và các giá trị lẻ vẫn giữ nguyên vị trị

Chapter 5

Recursion

Objectives

Discuss the following topics:

- Recursive Definitions
- Method Calls and Recursion Implementation
- Anatomy of a Recursive Call
- Tail Recursion
- Nontail Recursion
- Indirect Recursion

Objectives (continued)

Discuss the following topics:

- Nested Recursion
- Excessive Recursion
- Backtracking
- Case Study: A Recursive Descent Interpreter

Recursive Definitions

- **Recursive definitions** are programming concepts that define themselves
- A recursive definition consists of two parts:
  - The **anchor** or **ground case**, the basic elements that are the building blocks of all other elements of the set
  - Rules that allow for the construction of new objects out of basic elements or objects that have already been constructed

Recursive Definitions

- Here is the recurrence for **The Towers of Hanoi puzzle**:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + 2T(n-1) & \text{if } n \geq 1 \end{cases}$$

Recursive Definitions (continued)

- Recursive definitions serve two purposes:
  - **Generating** new elements
  - **Testing** whether an element belongs to a set
- Recursive definitions are frequently used to define functions and sequences of numbers

Method Calls and Recursion Implementation

- Activation records contain the following:
  - Values for all parameters to the method, location of the first cell if an array is passed or a variable is passed by reference, and copies of all other data items
  - Local (automatic) variables that can be stored elsewhere
  - The return address to resume control by the caller, the address of the caller's instruction immediately following the call

Method Calls and Recursion Implementation (continued)

- A dynamic link, which is a pointer to the caller's activation record
- The returned value for a method not declared as void

Method Calls and Recursion Implementation (continued)

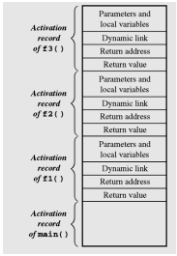


Figure 5-1 Contents of the run-time stack when `main()` calls method `f1()`, `f1()` calls `f2()`, and `f2()` calls `f3()`

Anatomy of a Recursive Call

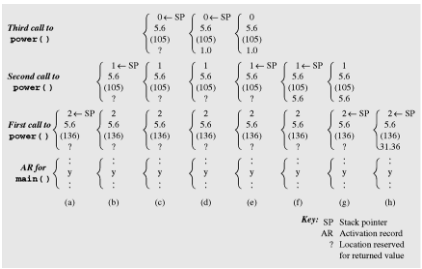


Figure 5-2 Changes to the run-time stack during execution of `power(5.6,2)`

Tail Recursion

- **Tail recursion** is characterized by the use of only one recursive call at the very end of a method implementation

```
void tail (int i) {
  if (i > 0) {
    System.out.print (i + "");
    tail(i-1);
  }
}
```

Nontail Recursion

Example of nontail recursion:

```
void nonTail (int i) {
  if (i > 0) {
    nonTail(i-1);
    System.out.print (i + "");
    nonTail(i-1);
  }
}
```


Nontail Recursion (continued)

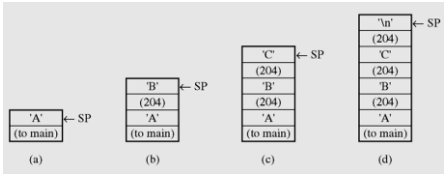


Figure 5-3 Changes on the run-time stack during the execution of `reverse()`

Nontail Recursion (continued)

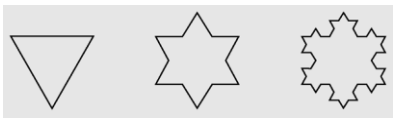


Figure 5-4 Examples of von Koch snowflakes

1. Divide an interval *side* into three even parts
2. Move one-third of *side* in the direction specified by *angle*

Nontail Recursion (continued)

3. Turn to the right 60° (i.e., turn –60°) and go forward one-third of *side*
4. Turn to the left 120° and proceed forward one-third of *side*
5. Turn right 60° and again draw a line one-third of *side* long

Nontail Recursion (continued)

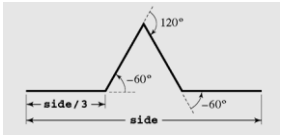


Figure 5-5 The process of drawing four sides of one segment of the von Koch snowflake

Nontail Recursion (continued)

```
drawFourLines (side, level)
   if (level = 0)
      draw a line;
   else
      drawFourLines(side/3, level-1);
      turn left 60°;
      drawFourLines(side/3, level-1);
      turn right 120°;
      drawFourLines(side/3, level-1);
      turn left 60°;
      drawFourLines(side/3, level-1);
```


Nontail Recursion (continued)

```
import java.awt.*;
import java.awt.event.*;

public class vonKoch extends Frame implements ActionListener {
   private TextField lvl, len;
   vonKoch() {
       super("von Koch snowflake");
       Label lvlLbl = new Label("level");
       lvl = new TextField("4",3);
       Label lenLbl = new Label("side");
       len = new TextField("200",3);
       Button draw = new Button("draw");
       lvl.addActionListener(this);
       len.addActionListener(this);
       draw.addActionListener(this);
       setLayout(new FlowLayout());
       add(lvlLbl);
       add(lvl);
       add(lenLbl);
       add(len);
       add(draw);
       setSize(600,400);
       setForeground(Color.white);
```

Figure 5-6 Recursive implementation of the von Koch snowflake

Nontail Recursion (continued)

```
       setBackground(Color.red);
       show();
       addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
               System.exit(0);
            }
       });
   }
   private double angle;
   private Point currPt, pt = new Point();
   private void right(double x) {
       angle += x;
   }
   private void left (double x) {
       angle -= x;
```

Figure 5-6 Recursive implementation of the von Koch snowflake (continued)

Nontail Recursion (continued)

```
}
private void drawFourLines(double side, int level, Graphics g) {
    if (level == 0) {
        // arguments to sin() and cos() must be angles given in
           radians,
        // thus, the angles given in degrees must be multiplied by
           PI/180;
        pt.x = ((int)(Math.cos(angle*Math.PI/180)*side)) + currPt.x;
        pt.y = ((int)(Math.sin(angle*Math.PI/180)*side)) + currPt.y;
        g.drawLine(currPt.x, currPt.y, pt.x, pt.y);
        currPt.x = pt.x;
        currPt.y = pt.y;
```

Figure 5-6 Recursive implementation of the von Koch snowflake (continued)

Nontail Recursion (continued)

```
    }
    else {
        drawFourLines(side/3.0,level-1,g);
        left (60);
        drawFourLines(side/3.0,level-1,g);
        right(120);
        drawFourLines(side/3.0,level-1,g);
        left (60);
        drawFourLines(side/3.0,level-1,g);
    }
}
public void actionPerformed(ActionEvent e) { // ActionListener
    repaint();
}
```

Figure 5-6 Recursive implementation of the von Koch snowflake (continued)

Nontail Recursion (continued)

```
    public void paint(Graphics g) {
        int level = Integer.parseInt(lvl.getText().trim());
        double side = Double.parseDouble(len.getText().trim());
        currPt = new Point(200,150);
        angle = 0;
        for (int i = 1; i <= 3; i++) {
            drawFourLines(side,level,g);
            right(120);
        }
    }
    static public void main(String[] a) {
        new vonKoch();
    }
}
```

Figure 5-6 Recursive implementation of the von Koch snowflake (continued)

Indirect Recursion

```
receive(buffer)
    while buffer is not filled up
        if information is still incoming
            get a character and store it in buffer;
        else exit();
        decode(buffer);

decode(buffer)
    decode information in buffer;
    store(buffer);

store(buffer)
    transfer information from buffer to file;
    receive(buffer);
```

Indirect Recursion (continued)

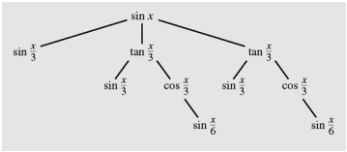


Figure 5-7 A tree of recursive calls for sin (*x*)

Excessive Recursion

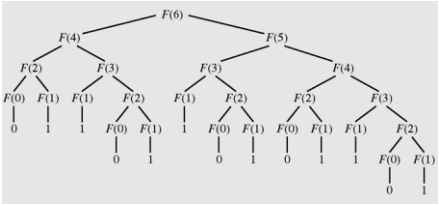


Figure 5-8 The tree of calls for `Fib(6)`

Excessive Recursion (continued)

n	Fib(n+1)	Number of Additions	Number of Calls
6	13	12	25
10	89	88	177
15	987	986	1,973
20	10,946	10,945	21,891
25	121,393	121,392	242,785
30	1,346,269	1,346,268	2,692,537

Figure 5-9 Number of addition operations and number of recursive calls to calculate Fibonacci numbers

Excessive Recursion (continued)

		Assignments	
n	Number of Additions	Iterative Algorithm	Recursive Algorithm
6	5	15	25
10	9	27	177
15	14	42	1,973
20	19	57	21,891
25	24	72	242,785
30	29	87	2,692,537

Figure 5-10 Comparison of iterative and recursive algorithms for calculating Fibonacci numbers

Backtracking

- **Backtracking** is a technique for returning to a given position (e.g., entry point) after trying other avenues that are unsuccessful in solving a particular problem

Backtracking (continued)

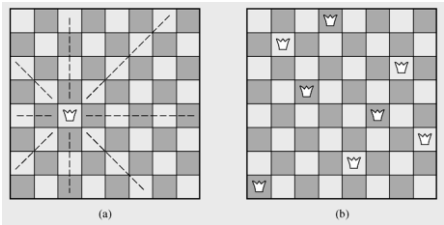


Figure 5-11 The eight queens problem

Backtracking (continued)

```
putQueen(row)
   for every position col on the same row
      if position col is available
         place the next queen in position col;
         if (row < 8)
           putQueen(row+1);
         else success;
         remove the queen from position col;
```


Backtracking (continued)

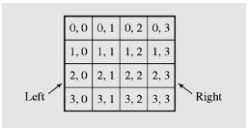


Figure 5-12 A 4 x 4 chessboard

Backtracking (continued)

```
import java.io.*;

class Queens {
    final boolean available = true;
    final int squares = 4, norm = squares - 1;
    int[] positionInRow = new int[squares];
    boolean[] column = new boolean[squares];
    boolean[] leftDiagonal  = new boolean[squares*2 - 1];
    boolean[] rightDiagonal = new boolean[squares*2 - 1];
    int howMany = 0;
    Queens() {
        for (int i = 0; i < squares; i++) {
            positionInRow[i] = -1;
            column[i] = available;
        }
        for (int i = 0; i < squares*2 - 1; i++)
            leftDiagonal[i] = rightDiagonal[i] = available;
```

Figure 5-13 Eight queens problem implementation

Backtracking (continued)

```
    }
    void PrintBoard(PrintStream out) {
        . . . . . .
    }
    void PutQueen(int row) {
        for (int col = 0; col < squares; col++)
            if (column[col] == available &&
                leftDiagonal [row+col] == available &&
                rightDiagonal[row-col+norm] == available) {
                positionInRow[row] = col;
                column[col] = !available;
                leftDiagonal[row+col] = !available;
                rightDiagonal[row-col+norm] = !available;
                if (row < squares-1)
                    PutQueen(row+1);
                else PrintBoard(System.out);
                column[col] = available;
                leftDiagonal[row+col] = available;
                rightDiagonal[row-col+norm] = available;
            }
```

Figure 5-13 Eight queens problem implementation (continued)

Backtracking (continued)

```
    }
    static public void main(String args[]) {
        Queens queens = new Queens();
        queens.PutQueen(0);
        System.out.println(queens.howMany + " solutions found.");
    }
}
```

Figure 5-13 Eight queens problem implementation (continued)

Backtracking (continued)

Move	Queen	row	col	
{1}	1	0	0	
{2}	2	1	2	failure
{3}	2	1	3	
{4}	3	2	1	failure
{5}	1	0	1	
{6}	2	1	3	
{7}	3	2	0	
{8}	4	3	2	

Figure 5-14 Steps leading to the first successful configuration of four queens as found by the method `putQueen()`

Backtracking (continued)

positionInRow	column	leftDiagonal	rightDiagonal	row
(0, 2, ,)	(!a,a,!a,a)	(!a,a,a,!a,a,a,a)	(a,a,!a,!a,a,a,a)	0, 1
{1}{2}	{1} {2}	{1} {2}	{2}{1}	{1}{2}
(0, 3, 1,)	(!a,!a,a,!a)	(!a,a,a,!a,!a,a,a)	(a,!a,a,!a,!a,a,a)	1, 2
{1}{3}{4}	{1} {4} {3}	{1} {4}{3}	{3} {1} {4}	{3}{4}
(1, 3, 0, 2)	(!a,!a,!a,!a)	(a,!a,!a,a,!a,!a,a)	(a,!a,!a,a,a,!a,!a)	0, 1, 2, 3
{5} {6} {7} {8}	{7} {5} {8} {6}	{5} {7} {6} {8}	{6} {5} {8} {7}	{5}{6}{7}{8}

Figure 5-15 Changes in the four arrays used by method `putQueen()`

Backtracking (continued)

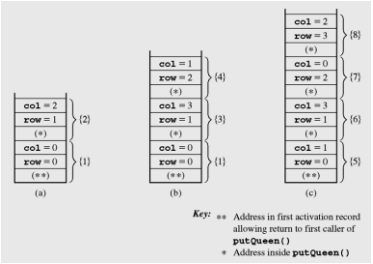


Figure 5-16 Changes on the run-time stack for the first successful completion of `putQueen()`

Backtracking (continued)

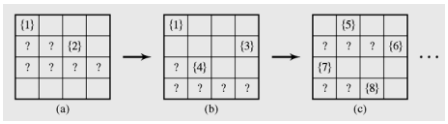


Figure 5-17 Changes to the chessboard leading to the first successful configuration

Backtracking (continued)

```
putQueen(1);
  col = 0;
  col = 1;
  col = 2;
  col = 3;
  putQueen(2)
    col = 0;
    putQueen(3)
      col = 0;
      col = 1;
      col = 2;
      success;
```

Figure 5-18 Trace of calls to `putQueen()` to place four queens (continued)

Case Study: A Recursive Descent Interpreter

- The process of translating one executable statement at a time and immediately executing it is called **interpretation**
- Translating the entire program first and then executing it is called **compilation**

Case Study: A Recursive Descent Interpreter (continued)

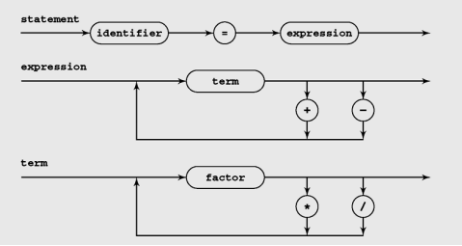


Figure 5-19 Diagrams of methods used by the recursive descent interpreter

Case Study: A Recursive Descent Interpreter (continued)

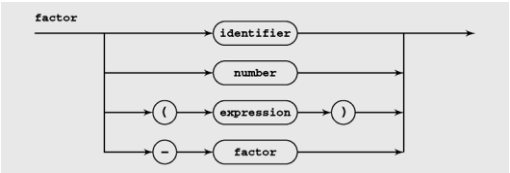


Figure 5-19 Diagrams of methods used by the recursive descent interpreter (continued)

Case Study: A Recursive Descent Interpreter (continued)

```
term()
    f1 = factor();
    while current token is either / or *
       f2 = factor();
       f1 = f1 * f2 or f1 / f2;
    return f1;

factor()
    process all +s and – s preceding a factor;
    if current token is an identifier
       return value assigned to the identifier;
    else if current token is a number
       return the number;
    else if current token is (
       e = expression();
       if current token is )
          return e;
```

Case Study: A Recursive Descent Interpreter (continued)

```
import java.io.*;

class Id {
    private String id;
    public double value;
    public Id(String s, double d) {
        id = s; value = d;
    }
    public boolean equals(Object node) {
        return id.equals(((Id)node).id);
    }
    public String toString() {
        return id + " = " + value + "; ";
    }
}
```

Figure 5-20 Implementation of a simple language interpreter

Case Study: A Recursive Descent Interpreter (continued)

```
public class Interpreter {
    private StreamTokenizer fIn = new StreamTokenizer(
                                  new BufferedReader(
                                  new InputStreamReader(System.in)));
    private java.util.LinkedList idList = new java.util.LinkedList();
    public Interpreter() {
        fIn.wordChars('$','$');// include underscores and dollar signs as
        fIn.wordChars('_','_');// word constituents; examples of identifiers:
                               // var1, x, _pqr123xyz, $aName;
        fIn.ordinaryChar('/'); // by default, '/' is a comment character;
        fIn.ordinaryChar('.'); // otherwise "n-123.45"
        fIn.ordinaryChar('-'); // is considered a token;
```

Figure 5-20 Implementation of a simple language interpreter (continued)

Case Study: A Recursive Descent Interpreter (continued)

```
    }
    private void issueError(String s) {
        System.out.println(s);
        Runtime.getRuntime().exit(-1);
    }
    private void addOrModify(String id, double e) {
        Id tmp = new Id(new String(id),e);
        int pos;
        if ((pos = idList.indexOf(tmp)) != -1)
            ((Id)idList.get(pos)).value = e;
        else idList.add(tmp);
    }
```

Figure 5-20 Implementation of a simple language interpreter (continued)

Case Study: A Recursive Descent Interpreter (continued)

```
private double findValue(String id) {
    int pos;
    if ((pos = idList.indexOf(new Id(id,0.0))) != -1)
         return ((Id)idList.get(pos)).value;
    else issueError("Unknown variable " + id);
    return 0.0;  // this statement is never reached;
}
private double factor() throws IOException {
    double val, minus = 1.0;
    fIn.nextToken();
    while (fIn.ttype == '+' || fIn.ttype == '-') { // take all '+'s
        if (fIn.ttype == '-')                      // and '-'s;
            minus *= -1.0;
        fIn.nextToken();
```

Figure 5-20 Implementation of a simple language interpreter (continued)

Case Study: A Recursive Descent Interpreter (continued)

```
    }
    if (fIn.ttype == fIn.TT_NUMBER || fIn.ttype == '.') {
        if (fIn.ttype == fIn.TT_NUMBER) {  // factor can be a number:
            val = fIn.nval;                // 123, .123, 123., 12.3;
            fIn.nextToken();
        }
        else val = 0;
        if (fIn.ttype == '.') {
            fIn.nextToken();
            if (fIn.ttype == fIn.TT_NUMBER) {
                String s = fIn.nval + "";
                s = "." + s.substring(0,s.indexOf('.'));
                val += Double.valueOf(s).doubleValue();
            }
            else fIn.pushBack();
        }
        else fIn.pushBack();
```

Figure 5-20 Implementation of a simple language interpreter (continued)

Case Study: A Recursive Descent Interpreter (continued)

```
    }
    else if (fIn.ttype == '(') {          // or a parenthesized
        val = expression();               // expression,
        if (fIn.ttype == ')')
            fIn.nextToken();
        else issueError("Right parenthesis is left out.");
    }
    else {
        val = findValue(fIn.sval);        // or an identifier;
    }
```

Figure 5-20 Implementation of a simple language interpreter (continued)

Case Study: A Recursive Descent Interpreter (continued)

```
        return minus*val;
    }
    private double term() throws IOException {
        double f = factor();
        while (true) {
            fIn.nextToken();
            switch (fIn.ttype) {
                case '*' : f *= factor(); break;
                case '/' : f /= factor(); break;
                default  : fIn.pushBack(); return f;
            }
        }
    }
```

Figure 5-20 Implementation of a simple language interpreter (continued)

Case Study: A Recursive Descent Interpreter (continued)

```
    }
    private double expression() throws IOException {
        double t = term();
        while (true) {
            fIn.nextToken();
            switch (fIn.ttype) {
                case '+' : t += term(); break;
                case '-' : t -= term(); break;
                default  : fIn.pushBack(); return t;
            }
        }
    }
```

Figure 5-20 Implementation of a simple language interpreter (continued)

Case Study: A Recursive Descent Interpreter (continued)

```
public void run() {
    try {
        System.out.println("The program processes statements in the "
                + "following format:\n"
                + "\t<id> = <expr>;\n\tprint <id>\n\tstatus\n\tend");
        while (true) {
            System.out.print("Enter a statement: ");
            fIn.nextToken();
            String str = fIn.sval;
            if (str.toUpperCase().equals("STATUS")) {
                 java.util.Iterator it = idList.iterator();
                 while (it.hasNext())
                     System.out.println(it.next());
            }
            else if (str.toUpperCase().equals("PRINT")) {
                 fIn.nextToken();
```

Figure 5-20 Implementation of a simple language interpreter (continued)

Case Study: A Recursive Descent Interpreter (continued)

```
            str = fIn.sval;
            System.out.println(str + " = " + findValue(str));
        }
        else if (str.toUpperCase().equals("END"))
            return;
        else {
            fIn.nextToken();
            if (fIn.ttype == '=') {
                 double e = expression();
                 fIn.nextToken();
                 if (fIn.ttype != ';')
                     issueError("There are some extras in the
                     statement.");
                 else addOrModify(str,e);
            }
            else issueError("'=' is missing.");
        }
```

Figure 5-20 Implementation of a simple language interpreter (continued)

Case Study: A Recursive Descent Interpreter (continued)

```
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public static void main(String args[]) {
        (new Interpreter()).run();
    }
}
```

Figure 5-20 Implementation of a simple language interpreter (continued)

Summary

- Recursive definitions are programming concepts that define themselves
- Recursive definitions serve two purposes:
  - Generating new elements
  - Testing whether an element belongs to a set
- Recursive definitions are frequently used to define functions and sequences of numbers

Summary (continued)

- Tail recursion is characterized by the use of only one recursive call at the very end of a method implementation.
- Backtracking is a technique for returning to a given position (e.g., entry point) after trying other avenues that are unsuccessful in solving a particular problem.
- The process of translating one executable statement at a time and immediately executing it is called interpretation.