Recursive Data Types

Spring 2014
Today’s Topics

- Personal Art competition winners
- Lists
- Immutable lists
- Immutable lists with Empty
- Datatype definitions
- Functions on recursive types
- Declared vs. actual types
- Beneficent mutation
- Example: Boolean formulas
2nd Place

```java
/**
 * Draw your personal, custom art.
 * Many interesting images can be drawn using the simple implementation of a turtle. For this
 * function, draw something interesting; the complexity can be as little or as much as you want.
 * We'll be peer-voting on the different images, and the highest-rated one will win a prize.
 */

public static void drawPersonalArt(Turtle turtle) {
    //change color to achieve illusion of depth
    PenColor[] colors = {PenColor.BLACK, PenColor.BLUE, PenColor.GRAY, PenColor.CYAN};
    //rotate slightly around vertical axis so that colors don't overlap completely
    double[] turnAngles = {0, 0.5, 359, 1.5};

    //draw multiple overlapping snowflakes
    for (int i=0; i< colors.length; i++) {
        turtle.color(colors[i]);
        turtle.turn(turnAngles[i]);
        drawSnowflake(turtle, 160);
    }
}
```

By Carmela Antoinette

© 6.005 Course Staff 07-13
public static void triangulateSegment(Turtle turtle, int segmentLength) {
    //determine the number of times segment is divided for triangulation based on
    //segmentLength for aesthetic purposes using change of base formula
    int expFactor = (int)Math.round(Math.log(segmentLength)/Math.log(3)) - 1;
    //uses recursion to triangulate with base case value of expFactor as 0
    if (expFactor == 0) {
        turtle.forward(segmentLength);
    } else {
        //draw the triangulated segment
        //angles and lengths chosen for aesthetics (mostly 30-60-90 triangles)
        double[] turnAngles = {60, 240, 60, 0};
        for (int i = 0; i < turnAngles.length; i++) {
            triangulateSegment(turtle, drawLength);
            turtle.turn(turnAngles[i]);
        }
    }
}

public static void drawSnowflakeBranch(Turtle turtle, int branchLength, boolean direction) {
    //initialize to angle of branches turned down
    double turnAngle = 120;
    //change turnAngle depending on direction
    if (direction) turnAngle = 180 - turnAngle;
    //draw the branch
    //angles and lengths chosen for aesthetics (mostly 30-60-90 triangles)
    turtle.turn(turnAngle);
    triangulateSegment(turtle, branchLength);
    turtle.turn(90);
    triangulateSegment(turtle, (int)branchLength/10);
    turtle.turn(90);
    triangulateSegment(turtle, (int)branchLength/10);
    turtle.turn(90);
    triangulateSegment(turtle, (int)branchLength/10);
    turtle.turn(180 - turnAngle);
}

public static void drawSnowflakeArm(Turtle turtle, int armLength) {
    //creating the inverted triangular base
    //height of the triangle is roughly 1/3 of the arm length
    //0.385 is roughly the ratio of hypotenuse of the 30-60-90 smaller triangle to the whole arm
    turtle.turn(30);
    triangulateSegment(turtle, (int)Math.round(0.385*armLength));
    turtle.turn(240);
    triangulateSegment(turtle, (int)Math.round(0.385*armLength)/2);
    turtle.turn(90);
    //moves forward to create the three branches using recursion
    //each branch is in turn a snowflake arm with three smaller branches
    int i = 5/12;  //first branch is about 5/12 through the arm, second about 2/3, third about 19/24
    int[] branchesAndDistances = new int[]{{(int)Math.round(armLength/3), (int)Math.round(armLength/3), (int)Math.round(armLength/4), (int)Math.round(armLength/3)},
                                           (int)Math.round(armLength/4), (int)Math.round(armLength/3)});
    //draw branches
    for (int i = 0; i < branchesAndDistances.length; i++) {
        turtle.forward(branchesAndDistances[i]);
        drawSnowflakeBranch(turtle, branchesAndDistances[i], true);
    }
    //form the tip
    turtle.forward((int)Math.round(5*armLength/24));
    turtle.turn(30);
    triangulateSegment(turtle, (int)armLength/15);
    vertex(turtle, armLength/4);
    //repeat everything in opposite order to create other side of arm
    triangulateSegment(turtle, (int)armLength/15);
    turtle.turn(30);
    forward(turtle, (int)Math.round(5*armLength/24));
    //draw reversed branches
    for (int i = 0; i < branchesAndDistances.length; i++) {
        drawSnowflakeBranch(turtle, branchesAndDistances[i], false);
        turtle.forward(branchesAndDistances[i]);
    }
    //each iteration draws 1 arm then rotates
    for (int numOfArms = 0; numOfArms < 6; numOfArms++)
        drawSnowflakeArm(turtle, armLength);
        turtle.turn(120);
}
public static void drawPersonalArt(Turtle turtle) {
    // This draws a picture using data encoded in the PictureData class.

    // The picture was produced from a PNG image that was processed by a
    // custom image processing script (included as /ext/improc.py).
    // This script first resizes the image to fit the dimensions of the
    // canvas, and then it recolors the image. The image uses 24-bit RGB
    // colors, while we only have 10 colors available to us (in PenColor),
    // so we have to recolor the image. To do this, the script transforms
    // both the image and the 10 colors into HSV color space and uses a
    // simple heuristic to choose the closest color for each pixel. It
    // then encodes this data as an array of strings and writes it to
    // PictureData (in /src/turtle/PictureData.java).

    // The data had to be encoded into a Java class (instead of something
    // more reasonable like a text file or resource file) because of the
    // way the auto-grader is implemented (file IO is not permitted, and
    // we can't package JARs ourselves).

    // starting from the center and pointing north,
    // move to starting point (top left) and point in the east direction
    turtle.forward(PictureData.HEIGHT / 2);
    turtle.turn(270.0);
    turtle.forward(PictureData.WIDTH / 2);
    turtle.turn(180.0);

    // image data is encoded in row-major order, so
    // color row by row, starting at top left, ending at bottom right
    for (int iRow = 0; iRow < PictureData.HEIGHT; iRow++) {
        // color an individual row by coloring each column in that row
        for (int iCol = 0; iCol < PictureData.WIDTH; iCol++) {
            char color = PictureData.DATA[iRow].charAt(iCol);
            // set pen color based on encoding
            switch (color) {
                case 'k':
                    turtle.color(PenColor.BLACK);
                    break;
                case 'a':
                    turtle.color(PenColor.GRAY);
                    break;
                case 'r':
                    turtle.color(PenColor.RED);
                    break;
                case 'p':
                    turtle.color(PenColor.PINK);
                    break;
                case 'o':
                    turtle.color(PenColor.ORANGE);
                    break;
                case 'y':
                    turtle.color(PenColor.YELLOW);
                    break;
                case 'g':
                    turtle.color(PenColor.GREEN);
                    break;
                case 'c':
                    turtle.color(PenColor.CYAN);
                    break;
                case 'b':
                    turtle.color(PenColor.BLUE);
                    break;
                case 'm':
                    turtle.color(PenColor.MAGENTA);
                    break;
            }
            turtle.forward(1); // unit movement
        }
        // from being at the end of a row, move down one row,
        // move back to the beginning of that row, and point
        // in the east direction to be ready to color the next row
        turtle.turn(90.0);
        turtle.forward(1);
        turtle.turn(90.0);
        turtle.forward(PictureData.WIDTH);
        turtle.turn(180.0);
    }
}
def encode_data_from_image_file(input_file):
    # Output file so that it can be drawn by a Java program
    output_file = '
    dimensions = (512, 512)
    template = "AUTO GENERATED FILE, DO NOT MODIFY"
    package turtle;
    public class PictureData {
        // width of the image
        public static final int WIDTH = %d;
        // height of the image
        public static final int HEIGHT = %d;
        // encoded image data for individual pixels
        // individual characters correspond to colors:
        // k => Black
        // a => Gray
        // r => Red
        // p => Pink
        // o => Orange
        // y => Yellow
        // g => Green
        // c => Cyan
        // b => Blue
        // m => Magenta
        // the data is stored in row-major order
        public static final String[] DATA = {%s};
        // in RGB
        def recolor(img):
            """Recolor the image from BGR color space (which is the OpenCV default) to the HSV color space to make it easier to map colors."""
            return cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
        
        def colormap(pixel):
            """Determine the closest color from MAPPING to recolor a certain pixel. Perform comparisons in the HSV color space and use a simple heuristic to weight hue, saturation, and value to determine the closest color."
            def min_index(lst):
                return min(((i, lst[i]) for i in range(len(lst))), key=lambda i: (i[1][0])
            def weight(color):
                return HUE_WEIGHT * abs(abs_output_file - int(color[0])) + 
                SAT_WEIGHT * abs(abs(input_file - int(color[1])) + 
                VAL_WEIGHT * abs(abs(color[2]) - int(color[2])))
            weights = [weight(color) for color in COLORS]
            return LETTERS[abs_index(weights)]
    
    def chunks(lst, n):
        """Chunk a sliceable into pieces."
        return (lst[i:i+n] for i in range(0, len(lst), n))
    
    def encoded(img):
        """Encode image data as a width, height, and array of Java strings."
        data = "\"" + \n                + \n                + \n            return tuple([DIMENSIONS] + [\', \'.join(data) + \n            def main():
                root = os.path.dirname(os.path.realpath(__file__))
                abs_input_file = os.path.join(root, INPUT_FILE)
                abs_output_file = os.path.join(root, OUTPUT_FILE)
                img = cv2.imread(abs_input_file, cv2.CV_LOAD_IMAGE_COLOR)
                img = resize(img)
                img = recolor(img)
                with open(abs_output_file, 'w') as fout:
                    fout.write(TEMPLATE % encoded(img))
    
if __name__ == '__main__':
    main()
Votes on Art

![Bar chart showing votes on different options, with option 1 having the highest votes and option 3 having a significant but lower number of votes.]

© 6.005 Course Staff 07-13
Today’s Topics

Personal Art competition winners

Lists
Immutable lists
Immutable lists with Empty
Datatype definitions
Functions on recursive types
Declared vs. actual types
Beneficent mutation
Example: Boolean formulas
public class List<E> {
    private E val;
    private List<E> rest;

    List(E val) {
        this.val = val;
        rest = null;
    }

    List(E val, List<E> rest) {
        this.val = val;
        this.rest = rest;
    }

    public E first()        { return val; }  
    public List<E> rest()   { return rest; } 
    public int length() {  
        if(rest == null)  
            return 1;  
        else  
            return 1 + rest.length();  
    }
}
public class ImList<E> {
    private final E val;
    private final ImList<E> rest;

    ImList(E val) {
        this.val = val;
        rest = null;
    }

    ImList(E val, ImList<E> rest) {
        this.val = val;
        this.rest = rest;
    }

    public E first() { return val; }

    public ImList<E> rest() { return rest; }

    public int length() {
        if(rest == null)
            return 1;
        else
            return 1 + rest.length();
    }
}
Using the list

1. Open your laptop

2. Find a partner (Optional)

3. Go to the following URL
   ➢ http://tinyurl.com/6005isbest

4. Answer the simple question

5. Close your laptop
Null vs Empty

Empty represented by null is a pain...

- Instead of

  ```java
  if(a.length() == b.length())
  ```

- We get

  ```java
  if((a == null && b == null) || (!a == null || b == null) &&
  (a.length() == b.length()))
  ```

Use a Sentinel Object to represent Empty

- Like the special last list element in the previous lecture

Sentinel objects behave like objects of the same data type

- E.g. can call `length()` on Empty
- Cleaner code, since instead of checking if a reference is null, we just call the method on the object
- Good practice in designing ADTs: will prevent bugs & save programmer effort
Immutable Lists

A fundamental data structure in many languages

- Scheme, Lisp, etc
- Can be shared safely
- Performance benefits: less time copying, less memory consumed

Four fundamental operations

- empty: void → ImList
  • Constructor
- cons: E × ImList → ImList
  • Returns a new list formed by adding a new element to the front of an existing list
- first: ImList → E
  • Returns the first element of a list. The list must be nonempty.
- rest: ImList → ImList
  • Returns the list of all elements of the list except the first. The list must be nonempty
Example Operations

empty() = []
cons(0, empty()) = [0]

x = cons(1, cons(2, cons(3, empty()))) = [1, 2, 3]
first(x) = 1
rest(x) = [2, 3]

first(rest(x)) = 2
rest(rest(x)) = [3]
rest(rest(rest(x))) = []

// fundamentally, for element e and list l,
first(cons(e, l)) = e
rest(cons(e, l)) = l
/** Immutable List interface */
public interface ImList<E> {
    /**
     * Cons adds a new item to the front of the list
     * @param e the element to add to the front
     * @return a new list consisting of e followed by this list */
    public ImList<E> cons (E e);

    /**
     * Returns the first element of the list. Requires the list be nonempty.
     * @return the first element of the list */
    public E first();

    /**
     * Returns the list except the first element. Requires the list to be nonempty.
     * @return the list except for the first element.
     */
    public ImList<E> rest();
Implementing ImList: Empty

/**
 * Implements the result of the empty() operation on immutable
 * lists.
 */

public class Empty<E> implements ImList<E> {

    public Empty() {
    }

    public ImList<E> cons(E e) { return new Cons<E>(e, this); }

    public E first() { threw new UnsupportedOperationException(); }

    public ImList<E> rest() {
        throw new UnsupportedOperationException();
    }
}
Implementing ImList: Cons

/**
 * Implements the result of a cons operation.
 */

public class Cons<E> implements ImList<E> {

    // the element
    private E e;

    // the rest of the list
    private ImList<E> rest;

    // constructor
    public Cons(E e, ImList<E> rest) {
        this.e = e;
        this.rest = rest;
    }

    public ImList<E> cons(E e) { return new Cons<E>(e, this); }

    public E first() { return e; }

    public ImList<E> rest() { return rest; }
}
ImList & Sharing

ImList<Integer> x = empty().cons(2).cons(1).cons(0)
ImList<Integer> y = x.rest().cons(4)

Is this sharing safe?
Data Type Definitions

Data type definition has

- Data type on the left
- Variants of the data type separated by “+” on the right
- Each variant = constructor with 0 or more named & typed arguments

MyInt = Int(num:int)

- MyInt is a synonym for the built-in type int with the name num

Boolean = true + false

Suit = Club + Diamond + Heart + Spade

- Example of an enum-like type

IntOrFloat = Int(num:int) + Float(num:float)

- Union-like type

© 6.005 Course Staff 07-13
Recursive Data Types

Data type definition for ImList

- ImList = Empty + Cons(first:E, rest:ImList)

ImList appears on both the left and right sides

- ImList is a recursive data type

Another example: binary tree

- Tree = Empty + Node(element:E, left:Tree, right:Tree)
Functions on Recursive Data Types

Using data type definitions makes defining operations easier

- Just think of operations in terms of one case per variant

Example: size()

- size: ImList \(\rightarrow\) Int
- Data type definition: ImList = Empty + Cons(first:E, rest:ImList)
- So, two cases
  - size(Empty) = 0
  - size(Cons(first:E, rest:ImList)) = 1 + size(rest)
- This recursive definition leads naturally to simple, understandable, recursive code
Functions on Recursive Data Types

**isEmpty: ImList \rightarrow boolean**
- isEmpty(Empty) = true
- isEmpty(Cons(first:E, rest:ImList)) = false

**contains: ImList x E \rightarrow boolean**
- contains(Empty, e:E) = false
- contains(Cons(first:E, rest:ImList), e:E) = first==e || contains(rest, e)
Functions on Recursive Data Types

1. Open your laptop

2. Find a partner (Optional)

3. Go to the following URL
   ➢ http://tinyurl.com/omg6005

4. Answer the two questions

5. Close your laptop
Functions on Recursive Data Types

**append**: \( \text{ImList} \times \text{ImList} \rightarrow \text{ImList} \)

- \( \text{append}(\text{Empty}, \text{list2}: \text{ImList}) = \text{list2} \)
- \( \text{append}(\text{Cons}(\text{first}: \text{E}, \text{rest}: \text{ImList}), \text{list2}: \text{ImList}) = \text{cons}(\text{first}, \text{append}(\text{rest}, \text{list2})) \)

**reverse**: \( \text{ImList} \rightarrow \text{ImList} \)

- \( \text{reverse}(\text{Empty}) = \text{empty()} \)
- \( \text{reverse}(\text{Cons}(\text{first}: \text{E}, \text{rest}: \text{ImList})) = \text{append}(\text{reverse}(\text{rest}), \text{cons}(\text{first}, \text{empty()})) \)
Declared Type vs Actual Type

In Java, two regimes for type checking: compile time and run time

Compile-time type checking
- Every variable has a declared type (the types in the declarations)
- Compiler enforces restrictions on declared types
  - E.g. only calling methods of declared type through its reference
  - E.g. not having statements like int foo = “hello”;

Run-time type checking
- Every object has an actual type dictated by the constructor
- E.g. ImList<Integer> a = new Empty()
  - Actual type of a is Empty, not ImList
- Some classes of type errors are caught at run time
Tuning the Rep

Adding a size method

```java
public interface ImList<E> {
    ...
    public int size();
}

public class Empty<E> implements ImList<E> {
    ...
    @Override
    public int size() {
        return 0;
    }
}

ImList = Empty + Cons(first:E, rest:ImList)
size: ImList → Int
So, two cases
size(Empty) = 0
size(Cons(first:E, rest:ImList)) = 1 + size(rest)
```
Tuning the Rep

Adding a size method

```java
public class Cons<E> implements ImList<E> {
    private final E e;
    private final ImList<E> rest;
    private final int size;
    // rep invariant
    // size = rest.size() + 1

    public Cons(E e, ImList<E> rest) {
        this.e = e;
        this.rest = rest;
        this.size = rest.size() + 1;
    }

    @Override
    public int size() {
        return size;
    }
}
```

So, two cases

- size(Empty) = 0
- size(Cons(first:E, rest:ImList)) = 1 + size(rest)
Tuning the Rep

Adding a size method, take II

```java
public class Cons<E> implements ImList<E> {
    private final E e;
    private final ImList<E> rest;
    private int size = 0;
    // rep invariant
    // size > 0 implies size = rest.size() + 1

    public Cons(E e, ImList<E> rest) {
        this.e = e;
        this.rest = rest;
    }
    ...
    @Override
    public int size() {
        if (size == 0)
            size = 1 + rest.size();
        return size;
    }
}
```

Immutable datatype
But mutable rep

Beneficent mutation
➢ Mutation that don’t change the abstract value represented by the object
Representing a Boolean function

Given a formula made up of Boolean variables and operators $\land$ (and), $\lor$ (or), $\neg$ (not), find an assignment of variables that makes the formula true

Example: $(P \lor Q) \land (\neg P \lor R)$
- Not true if $P=false$, $Q=false$, $R=false$
- Satisfied when $P=false$, $Q=true$, $R=true$
- Other solutions as well (E.g. $P=true$, $Q=false$, $R=true$)

Conjunctive Normal Form (CNF)
- Write a boolean formula as a “product of sums”
- i.e. Each clause is a sum (just $\lor$) and the clauses are combined with $\land$
- Standard way to write boolean formulas
Recursive Data Type for Boolean function

Formula = Var(name:String)
  + Not(formula:Formula)
  + Or(left:Formula, right:Formula)
  + And(left:Formula, right:Formula)

Example: $(P \lor Q) \land (\neg P \lor R)$
  - And(Or(Var(“P”), Var(“Q”)), Or(Not(Var(“P”)), Var(“R”)))

This is an abstract syntax tree for SAT

Why do we care about boolean satisfiability?
  - Theoretically speaking, SAT is a canonical difficult problem
  - 3-SAT (3 literals per clause) is known to be NP-complete [Cook, 1973]
  - Basis for theory of difficulty in computability
Summary

Recursive data types are ADTs that may reference instances of themselves

Datatype definitions: a powerful way to think about abstract data types, particularly recursive ones