Project 5

Fun with (Fun with (Fun with (Fun with (Fun with (Fun with (Fun with (recursion)

Recursion

- Definition: a method or structure that directly or indirectly invokes itself
- Must always have two situations:
  - At least one base (terminal) case that says when the recursion ends
  - At least one recursive case that results in it invoking itself while reducing the original problem in some way towards a base case

Examples of Recursive Data Structures

- A String is a sequence of characters
  - Base Case: Strings are null terminated – “\0”
- A node of a LinkedList is defined as having a value and a LinkedList
  - Base Case: Last node has a dummy (null) object for its LinkedList
- A node of a Tree is defined as having a value and two Trees
  - Base Case: A leaf is just a Tree node with null Trees

The Fibonacci Sequence

$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = 0$$

$$F(1) = 1$$

Application Contexts:

- Model the number of rabbit pairs as a function of time
- Prosody (meter) in Sanskrit
- Basis for Incan calculator for grain planting

A quick recursive program

The Fibonacci sequence is easily done with recursion:

```
public int fib (int num) {
  // Exercise: implement this method
}
```

A quick recursive program

The Fibonacci sequence is easily done with recursion:

```
public int fib (int num) {
  if (num == 0)
     return 0;
  else if (num == 1)
     return 1;
  else
     return (fib(num-1) + fib(num-2));}
```

Not very efficient though as fib(num) requires twice as many function calls as fib(num-1)

A higher level recursive program

- We’ve been sorting by comparing a number against all the numbers in the list
  - Very time intensive as lists get longer

- Two basic ideas help to speed up this process:
  - The shorter the list, the quicker it is to sort
  - If two lists are already sorted, putting them together is very simple.

Recursive sorting: Merge-Sort

- Base Case: a list of size 1 is already sorted!
- Recursive Case: Split the list in half and sort each half
- Then merge the sorted halves together

```
public void m_sort(List list)
{
  if (list.size == 1)
     return list;
  else
  {
     int middle = list.size/2;
     list left = list[0] to list[middle-1];
     list right = list[middle] to list[end];
     return (merge (m_sort(left),
                    m_sort(right)));
  }
}
```

```
public List merge(List left, List right) {
  int leftIndex = 0, rightIndex = 0;
  List output;

  while (leftIndex < left.size &&
           rightIndex < right.size){
      if (left[leftIndex] < right[rightIndex]) {
           output.add(left[leftIndex]);
           leftIndex++;
      }else{
           // right[rightIndex] <= left[leftIndex]
           output.add(right[rightIndex]);
           righttIndex++;
      }
  }
  // Handle trailing items
  if (leftIndex == left.size)
      output.add(the rest of right);
  else // rightIndex == right.size
      output.add(the rest of left);
}
```

Recursion vs Iteration

- Anything done recursively can be done through looping and vice versa
- Recursion Disadvantages:
  - Takes up memory space with each call
  - Takes time to make the function calls
  - Can result in StackOverflowErrors
- Recursion Advantages:
  - Can enable a cleaner and easier implementation
  - Some problems are just inherently recursive:
    - Towers of Hanoi, fractals

Project 5

Objectives:

- Implement recursive data structures and programs
- Use recursive structures to implement interesting Finch behavior.

New Execution Model

```
public void execute(Finch myFinch, double scale)
```

- Scale (0..1) specifies the “size” of a FinchAction
  - 1 = full size
  - 0 = zero size

- Our FinchActions to date will ignore scale (but the new ones will use it)

New Classes

- FinchTurn: turn the Finch by a specified angle at a specified velocity
- FinchJogScaled: the actual duration used is a product of the instance duration and the specified scale (hence, the distance traveled will be scaled)
- FinchMeta: a meta action that will conditionally execute a list of FinchActions

FinchMeta Properties

- String filter: name of actions in the master list to be executed in the recursive case
- int conditionType: determines the test that must be true in order for the recursive actions to be executed
- int loopType: determines whether the recursive actions should be executed once or repeatedly
- String filterTerminal: name of actions in the master list to be executed in the terminal case
- double scaleFactor: 0 .. 1. Specifies how much smaller the recursive actions are than the current one
- double minScale: 0 .. 1. Scale below which the terminal case is executed instead of the recursive case

FinchMeta: Condition Types

- conditionAll: always true

- conditionNoObstacle: true if there are no obstacles
- conditionLeftObstacle: true if there is an obstacle to the left
- conditionRightObstacle: true if there is an obstacle to the right
- conditionBothObstacle: true if there is an obstacle to both the left and right

- conditionLevel: true if the Finch is level
- conditionBeakUp: true if the Finch beak is up
- conditionBeakDown: true if the Finch beak is down
- conditionUpsideDown: true if the Finch is upsidedown

FinchMeta: “Loop” Types

- loopIf: the action sequence is executed if the
- condition is true
- loopNotIf: the action sequence is executed if the condition is **not** true
- loopWhile: the action sequence is executed repeatedly as long as the condition is true
- loopNotWhile: the action sequence is executed repeatedly as long as the condition is **not** true

FinchMeta Execution

Terminal case: the specified **scale** is less than **minScale**

- Actions that match **filterTerminal** are executed with a scale of **scale * scaleFactor**

Recursive case: the specified **scale** is greater than or equal to **minScale**.

- Actions that match **filter** are executed according to the **loopType** and **conditionType,** with a scale of **scale * scaleFactor**

FinchMeta Execution Example

FinchMeta action:

- filter = draw; filterTerminal = draw_end
- scaleFactor = .8; minScale = .2
- conditionType = BeakUp
- loopType = WhileNot

meta.execute(myFinch, .6):

- Non-terminal case: .6 * .8 > .2
- Checks with Finch to see if beak is up:
  - Yes: return
  - No: execute all of the actions in the maste rlist that match the name “draw”. Repeat with Check

Other Things to Handle

- Make sure that your new action classes are handled by the dialog box and the text file loader
- Create two interesting recursive programs (see the class web site for some references)