CS 241 — Introduction to Problem Solving and Programming

Capstone example

The animals game

April 27, 2005

Introduction

This example reviews

- Loops
- Recursion
- Linked structures
- Exceptions
- File I/O

And it also is an example of gaming and artificial intelligence.

Introduction

Animals is a game where a guesser asks a player a series of questions to deduce what animal the player is thinking of.

Specification:

> Write a program to play the roll of the guesser. If it guesses incorrectly (because the player is thinking of an animal unknown to it), it should *learn* a new animal and how to ask a question about that animal. It should retain this knowledge through successive runs of the program.

A program like that that demonstrates a growing expertise about something is an expert system.

Knowledge base

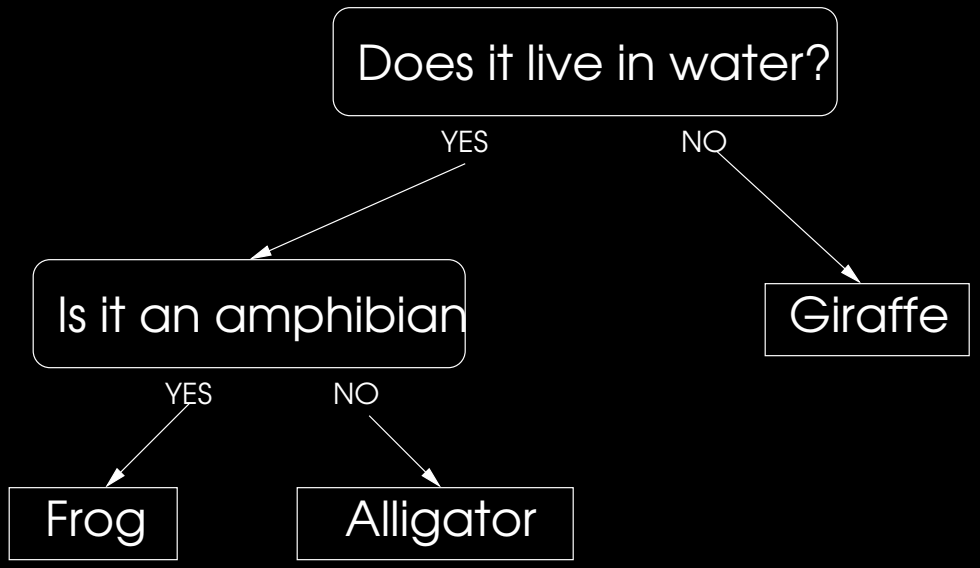
How can we represent/store/organize knowledge like this?

Ask if it lives in water

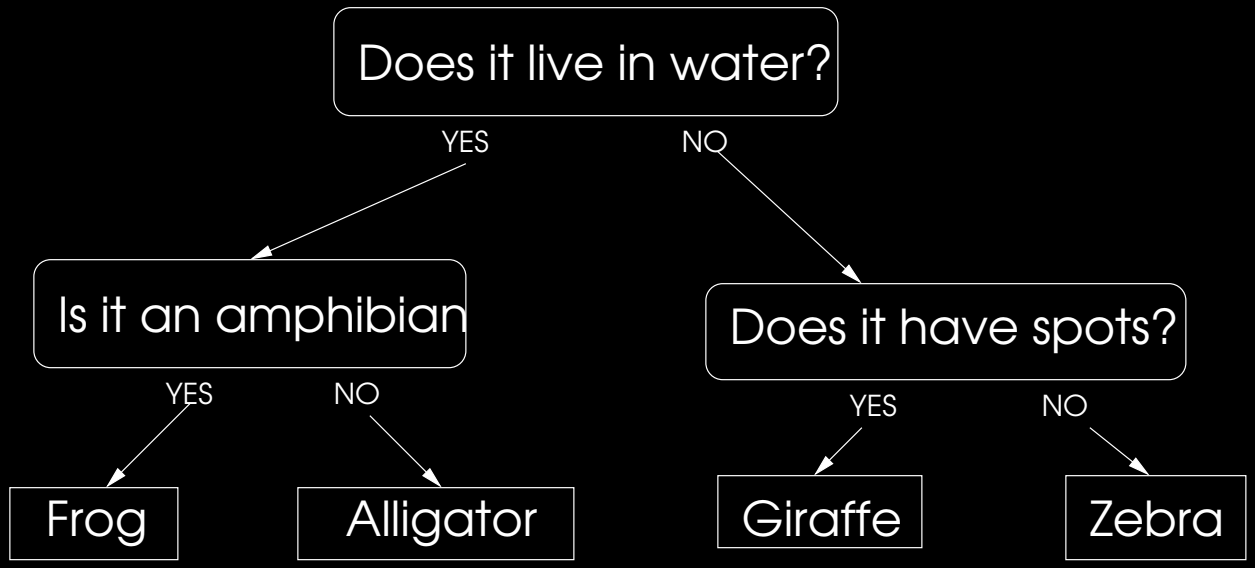
If so, ask if it is an amphibian
If so, guess it is a frog
If not, guess it is an alligator
if not, guess it is a giraffe

A data structure to hold knowledge is called a knowledge base.

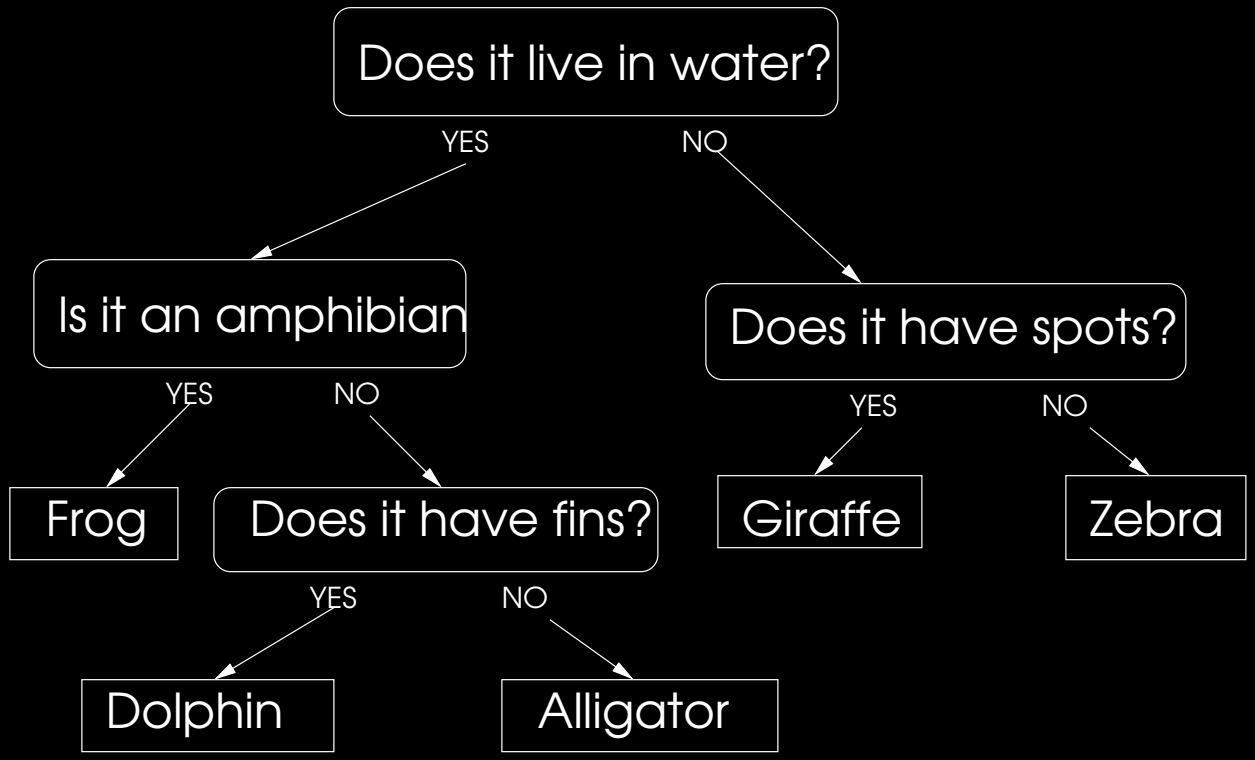
Knowledge base



Knowledge base



Knowledge base



Knowledge base

The general data structure used here is a binary tree. It is similar to the graphs we use for class hierarchies, except that every node has zero or two children.

We can store this with a linked structure, like linked lists, except (non-leaf) nodes need to have two references.

Note that leaf and non-leaf nodes will be implemented differently, but they are both nodes.

Knowledge base

The general data structure used here is a binary tree. It is similar to the graphs we use for class hierarchies, except that every node has zero or two children.

We can store this with a linked structure, like linked lists, except (non-leaf) nodes need to have two references.

Note that leaf and non-leaf nodes will be implemented differently, but they are both nodes.

Solution: Use an interface, implemented by two classes.

Knowledge base implementation

```
public interface Node

public class Question
        Node yes;
        Node no;
        String prompt;


public class Animal
        String animal;
```

General algorithm: Iterate through the tree, asking questions until we hit a leaf (animal) node; then guess.

Iterative algorithm

```
Node head =  ...

Node current = head;
while (current instanceof Question) {
     char query = DocsIO.readchar(((Question) current).prompt);
     if (query == 'y' || query == 'Y')
          current = ((Question) current).yes;
     else
          current = ((Question) current).no;
}

System.out.println("Is the animal you are thinking of a(n) "
                   + ((Animal) current).animal + "?");
```

Iterative algorithms

Drawbacks:

- Casting is confusing, takes a lot of typing, is inelegant, and generally should be avoided.

- Decision of what to do is based on type, which suggests using polymorphism.

Observation: In every case, a question is asked (for Animal nodes, the question is the guess); the differences is what to do next (ask a new question, or end the game). Leaf nodes are like a terminal case.

Recursion warm-up

To help think recursively about the tree, consider computing the number of animals in the tree.

```
public interface Node {
    public int numberOfAnimals();
}
public class Question {
    public int numberOfAnimals() {
        return yes.numberOfAnimals() + no.numberOfAnimals();
    }
}
public class Animal {
    public int numberOfAnimals() {
        return 1;
    }
}
```

Recursive algorithm

Solution:

Declare a method `ask()` in `Node`. Define it to be recursive in `Question` (it asks a question on one of its child nodes) and make its definition in `Animal` to be its base case.

```
public static void main(String[] args) {
    Node top = ... ;

    do
        top.ask();
    while (DocsIO.readchar("Do you want to play again?") == 'y');
}
```

Recursive implementation

```
public interface Node {
        public void ask();
}

public class Animal implements Node {
    private String animal;

    public Node ask() {
        System.out.println("Is the animal you are thinking of a(n) " +
                           animal + "?");
        char c = DocsIO.readchar();
        if (c == 'y' || c == 'Y')
            System.out.println("I got it!");
    }
}
```

Recursive implementation

```
public class Question implements Node {
    private Node yes;
    private Node no;
    private String prompt;

     public void ask() {
        System.out.println(prompt);
        char c = DocsIO.readchar();
        if (c == 'y' || c == 'Y')
            yes = yes.ask();
        else
            no = no.ask();
    }
}
```

Learning

We also want the system to be able to learn new animals when the guess is wrong.

What happens (or should happen) when we add something to the tree.

Learning algorithm

When a guess is a failure

- Ask what animal was being thought of.
- Ask for a question to differentiate between the two animals.
- Ask what the correct response (yes or no) would be for the new animal.
- Create a new question node based on the given question, with the current animal and the new one as children (arranged based on the answer to the question).
- Make the current node's parent point to the new question node.

Learning

We need constructors for `Animal` and `Question`

```
public class Question implements Node {
    ...
    public Question(String prompt, Node yes, Node no) {
        this.prompt = prompt;
        this.yes = yes;
        this.no = no;
    }
}
public class Animal implements Node {
    ...
    public Animal(String animal) {
        this.animal = animal;
    }
}
```

Learning

```
public void ask() {
    ...
    else {
        System.out.println("Ok, what animal were you thinking of?");
        String newAnimal = DocsIO.readString();
        Node newAnimalNode = new Animal(newAnimal);
        System.out.println("Please enter a question to "
                           + "differentiate between a(n) " + animal
                           + " and a(n) " + newAnimal + ".");
        String newQuestion = DocsIO.readString();
        char d =
            DocsIO.readchar("What would be the correct answer for a(n) "
                            + newAnimal + "?");
```

Learning

```
            Question newQuestionNode;
            if (d == 'y' || d == 'Y')
                newQuestionNode = new Question(newQuestion, newAnimalNode,
                                               this);
            else
                newQuestionNode = new Question(newQuestion, this,
                                               newAnimalNode);

            // Now what???
        }
    }
```

Learning

Problem: The node needs to change something in its parent, but it does not even know its parent.

A possible solution is to give each Animal node a parent link and give each Question node a setter method for yes and no (which would require also the Animal to know if it is its parent's yes or no).

That would make this more complicated, and wouldn't work if we were replacing the top node.

Learning

Solution: Make `ask()` return a node. Whenever we call `ask` on a node, replace that node with what `ask` returns.

```
public interface Node {
        public Node ask();
}
public class Animals {
    public static void main(String[] args) {
        ...
        do
            top = top.ask();
        while (DocsIO.readchar("Do you want to play again?") == 'y');
    }
}
```

Learning

```
public class Question implements Node {

     public Node ask() {
        System.out.println(prompt);
        char c = DocsIO.readchar();
        if (c == 'y' || c == 'Y')
            yes = yes.ask();
        else
            no = no.ask();
        return this;
     }
}
```

Learning

```
public class Animal implements Node {

    public Node ask() {
        System.out.println("Is the animal you are thinking of a(n) " +
                           animal + "?");
        char c = DocsIO.readchar();
        if (c == 'y' || c == 'Y') {
            System.out.println("I got it!");
            return this;
        }
        else {
            ...
            return newQuestionNode;
        }
    }
}
```

Retaining knowledge

Now to allow for the program to remember the knowledge base between runs of the program.

Obviously, this will require saving it to a file and loading it when the program is run next.

The first thing to figure out is how to format the file.

Saving the knowledge base in a file

```
<Node> ::= <Question> | <Animal>

<Question> ::= Q
               <string>
               <Node>
               <Node>

 <Animal> ::= A
              <string>
```

Writing to a file

```
public class Animals {
    public static void main(String[] args) {
        ...
        try {
            PrintWriter pw =
                new PrintWriter(new FileOutputStream("animals.dat"));
            top.print(pw);
            pw.close();
        } catch (IOException ioe) {
            System.out.println("Could not save animal knowledge.");
        }
```

Writing to a file

```
public class Question implements Node {
    ...
    public void print(PrintWriter pw) throws IOException {
        pw.println("Q");
        pw.println(prompt);
        yes.print(pw);
        no.print(pw);
    }
}
```

Writing to a file

```
public class Animal implements Node {
    ...
    public void print(PrintWriter pw) throws IOException {
        pw.println("A");
        pw.println(animal);
    }
}
```

Reading from file

Make the constructors recursive.

Each constructor will read off as much from the BufferedReader as it needs, assuming the sentinel lines are consumed, and call constructors for its children, passing them the BufferedReader.

Reading from a file

```
public class Animals {
    public static void main(String[] args) {
        ...
        try {
            BufferedReader br =
                new BufferedReader(new FileReader("animals.dat"));
            br.readLine();
            top = new Question(br);
            br.close();
        } catch (IOException ioe) {
            top = new Question("Does it live in water?",
                               new Animal("alligator"),
                               new Animal("giraffe"));
        }
```

Reading from a file

```
public class Question implements Node {
    ...
    public Question(BufferedReader br) throws IOException {
        prompt = br.readLine();
        if (br.readLine().equals("Q"))
            yes = new Question(br);
        else
            yes = new Animal(br);
        if (br.readLine().equals("Q"))
            no = new Question(br);
        else
            no = new Animal(br);
    }
}
```

Reading from a file

```
public class Animal implements Node {
    ...
    public Animal(BufferedReader br) throws IOException {
        this.animal = br.readLine();
    }
}
```