Structural Recursion and Induction

Of all the material in this course, this lesson is probably the hardest for students to grasp at first, but it's also one of the most fundamental. Once you understand it, everything else in the course becomes much easier, so we want you to play around with these ideas as much as you can. Come see us if you're having trouble; we really want this to make sense to you.

Objectives

At the end of this lesson, you should be able to do the following:

1. Implement recursively defined data structures.
2. Write structurally recursive functions that may require a helper function or an additional parameter.
3. Explain what structural induction is and why it is not a circular argument.
4. Use structural induction to prove the correctness of basic recursive programs over recursive data.

The exercises and problems included in this handout should help you learn this material. ***The problems are due at the beginning of class on Monday, July 1, along with the mathematical induction and recursion problems of the next handout, so don't wait until we've finished all of the material before starting!***

Using DrJava

At least for this part of the course, we'll be using a simple development environment called DrJava because it's very easy to install and it makes it possible to interactively experiment with our programs without having to write lots of skeleton code. The environment is being developed actively, and no documentation exists yet, but fortunately it's already quite usable for our purposes. Grab the latest copy from `http://drjava.sf.net/`. A walkthrough will be given in class.

Recursive Data and Recursive Functions

You should already be familiar with writing simple programs that use nonrecursive data structures like arrays, so today we're going to start off with a *recursive* implementation of lists. This is called a *singly linked list*, or simply a *linked list*. Enter Listing 1 into DrJava, save it as `List.java`, and compile it.

Probably you've not seen code written like this before, so let's walk through it. We're defining `List` to be an *abstract* type, and `Empty` and `Cons` are both *subtypes* of `List`. (`Cons` stands for *cons*truct – we're constructing a list from a datum and another list.) These are the *only* subtypes of `List`, which means that every `List` is either an instance of `Empty` or an instance of `Cons`. (We can't have any instance of just `List` because we declared `List` to be abstract.)

Now we can express lists of integers. Switch to the Interactions panel of DrJava and try:

```
public abstract class List {
}

class Empty extends List {
}

class Cons extends List {
  int first;
  List rest; // not null

  public Cons(int first, List rest) {
    this.first = first;
    this.rest = rest;
  }
}
```

Listing 1: Our initial `List` class.

```
> new Empty()
Empty@4b771a
> new Cons(3, new Empty())
Cons@443fbe
> new Cons(3, new Cons(5, new Empty()))
Cons@5aeb9e
```

Notice that when you enter a Java expression without a terminating semicolon, DrJava evaluates the expression and prints the result. Unfortunately, the results aren't very intelligible right now; all we know from the result of the last expression is that it was an instance of `Cons` and that it was distinct from the previous `Cons`. Let's fix this. If an expression evaluates to an object, DrJava prints it by calling the `toString` method of the object. All classes automatically have a `toString` method that prints the kind of results we've seen so far, but we want to write one that is more appropriate to `Lists`.

One way to do it is to writen a recursive method, mirroring the structure of the data. Add the methods from Listing 2 to the three classes we already have. In this example, we have an *abstract* `toString` method in the `List` class. In the `Empty` class, we implement the `toString` method to simply return the constant result appropriate for the empty list. The `Cons` class contains both a value and a reference to the rest of the list, so we implement `toString` to construct its result by *recursively* calling `toString` on the rest of the list and then *combining* this with the value at the current position.

Think of the recursion as creating separate *copies* of the method, each executing *with its own local variables*. It is *not* one set of local variables taking on many different values!

EXERCISE: Following this model, try to write a `sum` method that returns the sum of the elements in the list. Then write a `length` method that returns the number of elements in the list.

EXERCISE: Write a `mapSquare` method that returns a *new* list in which each element is the square of the corresponding element of the original list. Don't modify the original list.

```
public abstract class List {
  public abstract String toString();
}

class Empty extends List {
  public String toString() {
    return "<Empty>";
  }
}

class Cons extends List {
  public String toString() {
    return "(" + first + " " + rest.toString() + ")";
  }
}
```

Listing 2: A recursive `toString` implementation.

EXERCISE: Write an `isIncreasing` method that returns `true` if the list is in increasing order. Hint: This one requires a helper function.

Problems

1. Write a `max` method that returns the maximum value in the list. Think: what should `new Empty().max()` evaluate to?

2. Write a `filterEvens` method that returns a *new* list containing only the even numbers of the given list.

3. By now you should understand how lists are formed. Replace our `toString` implementation with one that produces results like `[2 1 1]` rather than `(2 (1 (1 <Empty>)))`. Think before you code: How will you be sure to include exactly one pair of brackets?

   To make constructing lists easier, also add the code from Listing 3. Now you should be able to write:

   ```
   > List.fromString("2 3 5 7").mapSquare()
   [4 9 25 49]
   ```

4. Write a recursive method `filterNonmultiples` that takes a parameter `n` and returns the sublist containing only those values that are *not* multiples of `n`. For example,

   ```
   List.fromString("4 7 9 12 16 19").filterNonmultiples(4)
   ```

   should return `[7 9 19]`.

5. In the style of `List`, implement a recursive data structure `Polynomial` that stores one-variable polynomials with integral powers. An example polynomial is $x^2 + 3.2x^{-17} + 0.9$. (Note that $0.9 = 0.9x^0$.) The variable name is irrelevant –you don't need to store it– and don't worry about the order of the terms. In addition to the usual `toString` method, write an `eval` method that evaluates

```
import java.io.*;

public abstract class List {
  public static List fromString(String str) throws IOException {
    StreamTokenizer st = new StreamTokenizer(new StringReader(str));
    st.resetSyntax(); // Let's keep it simple
    st.whitespaceChars(' ', ' ');  // Space is the only whitespace
    st.parseNumbers(); // We want numbers
    return fromStringTokenizer(st);
  }

  private static List fromStringTokenizer(StreamTokenizer st)
  throws IOException {
    switch (st.nextToken()) {
      case StreamTokenizer.TT_EOF:
        return new Empty();
      case StreamTokenizer.TT_NUMBER:
        return new Cons((int) st.nval, fromStringTokenizer(st));
      default:
        throw new IOException("Bad input string");
    }
  }
}
```

Listing 3: An implementation of `fromString`.

the polynomial at a given value for x. Test your code in DrJava, and submit it as `Polynomial.java`.

Think about how you could implement addition of polynomials or store more general mathematical expressions.

EXERCISE: What are the advantages of implementing `List` recursively rather than with arrays? What are the advantages of array-based implementations like `java.util.Vector`?

EXERCISE: Write an `append` method.

EXERCISE: Write a `reverse` method. Can you do it without calling `append`?

EXERCISE: Since a recursive function calls itself, how do we know that it will not loop forever?

Structural Induction

As we defined it, the set of all possible `Lists` is an *inductively defined set* because we can define it as

Let the set of all possible `Lists` be the smallest set such that:

Base case: `Empty` is a `List`
Inductive step: If e is a `List`, then `Cons(`e`)` also is a `List`

In general, an inductively defined set S is one that can be expressed as

Let the set S be the *smallest set* such that:

Base case: Some *base elements* are in the set S
Inductive step: If an element e is in the set S, then some manipulation of e according to some *rules* also is in the set S

EXERCISE: Inductively define the set of natural numbers.

EXERCISE: Inductively define the set of mathematical expressions involving natural numbers and the addition, subtraction, multiplication, and division operators.

By inductively defining a set, we succinctly and precisely describe a set that may contain an infinite number of elements. For example, the set of mathematical expressions described above is infinite and appears very difficult to enumerate (unlike the natural numbers), but it is very succinctly defined.

EXERCISE: Are all inductively defined sets infinite?

Conversely, each element of an inductively defined set can be derived from one of the *base elements* using only a finite number of applications of the *inductive step*.

EXERCISE: Given that 2, 3, and 5 are natural numbers, prove in terms of your definition that $2 + (3 \times 5)$ is a mathematical expression of the kind you defined earlier.

A *proof by induction* is a proof that some predicate is true for every element of an inductively defined set. There are different kinds of proof by induction, so to be specific we'll call this *proof by structural induction*. The basic form of such a proof looks like:

$P(e)$: State the predicate you want to prove about each element e.
Proof by structural induction:
Base case: Prove that the base elements of the set satisfy $P(e)$.
Induction step: Assume that $P(e)$ is true for any element e. (This is called the *induction hypothesis*.) Prove that the rules of the inductively defined set preserve the property $P(e)$.

Try to convince yourself that this makes sense.

EXERCISE: A common confusion is to think that we're trying to prove $P(e)$ from itself. Why is this not true?

EXERCISE: Prove that your `List sum` method is correct. Notice how the structure of the proof parallels the structure of the code.

EXERCISE: Prove that your `length` method is correct.

In your proofs, always be careful to follow the same basic structure: state exactly what you're trying to prove, how you're proving it (by structural induction over some set), what the base cases are, what the induction steps are, and where you use the induction hypothesis. Induction proofs are very straightforward when you do them correctly. Imprecise words like "eventually" have no place in induction proofs.

Problems

6. Inductively define the set of natural numbers not divisible by 5.

7. Prove that your `List append` method is correct.