Structural Recursion and Induction Solutions

Each problem was graded out of five points. In general, 5 points means you had a correct answer, 3 points means you had the right idea but had some problems carrying it out, and 1 point means an attempt was made. (2 and 4 points are somewhere in between.)

Problems

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

   The most commont solution, shown in Listing 1, mirrors the structure of the `isIncreasing` method. The advantage of this implementation is its simplicity – by assuming that the list is nonempty (or that `Integer.MIN_VALUE` is as good a value as any for the maximum value in an empty list), only one method needs to be implemented in both classes.

   An alternative approach, shown in Listing 2, uses a second method so that `new Empty().max()` doesn't return a maximum value, since an empty list really has no maximum value.

   The vast majority of students had no problems at all with their solutions to problems 1–4. One point was taken off if you had the wrong return type for `max`, and two points were taken if you were missing one of the recursive calls to what we called `max_startingAt` in the first solution.

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

   See Listing 1.

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?

   See Listing 3. Getting the spacing exactly right isn't important, though the solution shows one way to do it. The important feature is generating the one set of brackets with `toString` and using a helper function to recursively produce the string containing the elements.

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]`.

   See Listing 1. One point was taken off for returning those values that *are* multiples of `n` rather than what was asked.

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

```
public abstract class List {
  public int max() {
    return max_StartingAt(Integer.MIN_VALUE);
  }
  public abstract max_startingAt(int prevmax);
  public abstract List filterEvens();
  public abstract List filterNonmultiples(int n);
}

class Empty extends List {
  public int max_startingAt(int prevmax) {
    return prevmax;
  }
  public List filterEvens() {
    return new Empty();
  }
  public List filterNonmultiples(int n) {
    return new Empty();
  }
}

class Cons extends List {
  public int max_startingAt(int prevmax) {
    if (first > prevmax) {
      return rest.max_startingAt(first);
    } else {
      return rest.max_startingAt(prevmax);
    }
  }
  public List filterEvens() {
    if (first % 2 != 0) {
      return rest.filterEvens();
    } else {
      return new Cons(first, rest.filterEvens());
    }
  }
  public List filterNonmultiples(int n) {
    if (first % n != 0) {
      return new Cons(first, rest.filterNonmultiples(n));
    } else {
      return rest.filterNonmultiples(n);
    }
  }
}
```

Listing 1: Sample solution

```
public abstract class List {
  public abstract int max();
  public abstract int max_startingAt(int prevmax);
}

class Empty extends List {
  public int max() {
    throw new UnsupportedOperationException("Empty.max() undefined");
  }
  public int max_startingAt(int prevmax) {
    return prevmax;
  }
}

class Cons extends List {
  public int max() {
    return max_startingAt(first);
  }
  public int max_startingAt(int prevmax) {
    if (first > prevmax) {
      return rest.max_startingAt(first);
    } else {
      return rest.max_startingAt(prevmax);
    }
  }
}
```

Listing 2: An alternative implementation of `max`

```
public abstract class List {
  public String toString() {
    return "[" + toString_helper() + "]";
  }
  abstract protected String toString_helper();
}

class Empty extends List {
  protected String toString_helper() {
    return "";
  }
}

class Cons extends List {
  protected String toString_helper() {
    if (rest instanceof Empty) {
      return Integer.toString(first);
    } else {
      return first + " " + rest.toString_helper();
    }
  }
}
```

Listing 3: An implementation of `toString`.

addition to the usual `toString` method, write an `eval` method that evaluates the polynomial at a given value for x. Test your code in DrJava, and submit it as `Polynomial.java`.

See Listing 4. The coefficients must have type `double` or `float` in order to represent the example polynomial in the question.

One point was taken off if the `Empty` class did not implement the empty polynomial. One point was taken off if `toString` was missing, or `toString` or its helper function was not recursive, or `eval` was not recursive. Two points were taken off if `eval` was missing or implemented incorrectly. One common mistake was not writing `eval` to take an argument; this was probably due to a misunderstanding of the intended meaning of the function.

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

   Let the set S be the smallest set such that:
   Base case: 1, 2, 3, and 4 are in S.
   Inductive step: If n is in S, then $n + 5$ is in S.

   Although the majority of the class answered the two induction questions correctly, students had more trouble with these than they did with the recursion problems. Common mistakes inductively defining a set included:

   - Having too few base elements (e.g., using 1 as the only base element).
   - Having too many base elements (e.g., using all the naturals not divisible by 5 as the base elements).

```
public abstract class Polynomial {
  public abstract String toString();
  public abstract double eval(double x);
}

class Empty extends Polynomial {
  public String toString() {
    return "0";
  }
  public double eval(double x) {
    return 0.0;
  }
}

class Term extends Polynomial {
  double coefficient;
  int power;
  Polynomial rest;

  public Term(double coefficient, int power, Polynomial rest) {
    this.coefficient = coefficient;
    this.power = power;
    this.rest = rest;
  }
  public String toString() {
    return coefficient + "*x^" + power + rest.toString();
  }
  public double eval(double x) {
    return coefficient * Math.pow(x, (double) power) + rest.eval(x);
  }
}
```

Listing 4: `Polynomial.java`

- Writing an inductive step that conditionally added elements to the set (e.g., "If $n + 1 \bmod 5 \not\equiv 0$ then $n + 1$ is in S.")
- Being careless with the phrasing (e.g., "Base case: Let S be the smallest set such that 1, 2, 3, and 4 are in S." That just defines S to be $\{1, 2, 3, 4\}$.)

We took off one point if you failed to mention a set S that you were trying to define.

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

   Let $P($`e`$)$ be the predicate that for any `List e2`, `e.append(e2)` evaluates to a `List` containing all of the elements of `e` followed by all of the elements of `e2`. We will prove by induction over the structure of `Lists` that $P($`e`$)$ is true for all `Lists`.

   Base case: Let `e` be an instance of `Empty`. As given in the implementation of `append` in `Empty`, `e.append(e2)` evaluates to `e2` for any `List e2`. Thus $P($`e`$)$ is true when `e` is an empty list.

   Inductive step: Assume that $P($`e`$)$ is true for some `List e`. (This is our induction hypothesis.) Let `e1` be an instance of `Cons` containing `first` value `n` and `rest` value `e`. As given in the implementation of `append` in `Cons`, `e1.append(e2)` evaluates to `new Cons(n, e.append(e2))`. According to our induction hypothesis, `e.append(e2)` evaluates to a `List` containing all of the elements of `e` followed by all of the elements of `e2`, so `e1.append(e2)` evaluates to a `List` containing all of the elements of `e1` followed by all of the elements of `e2`. Thus $P($`e1`$)$ is true, and P is preserved by the `Cons` rule.

   People who had trouble with this problem should study the solution before the exam and come see us if you can't figure it out. Important parts to this problem include:

   - State the predicate $P($`e`$)$ that you're trying to prove for all `Lists e`. The predicate should mention `e`.
   - Realize that the result of `append` is not necessarily a `Cons`. (Consider appending an empty list onto an empty list.)
   - For the base case, prove $P($`e`$)$ for `e` being an instance of `Empty`. You prove this by citing the implementation in the `Empty` class.
   - For the inductive step, assume $P($`e`$)$ for some `List e` and then prove $P($`e1`$)$ for the list `e1 = new Cons(n,e)`. Point out your induction hypothesis, and point out when you make use of it.