Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue&Thu 12:00-13:15 @SH 212, Lab: Fri 10:00-11:15/11:30-12:45 @EB 103

Lab 7: Computations on Trees

Learning Objectives Upon successful completion, students will be able to:

- Apply the paradigm of using local computations at nodes to compute global results over trees.
- Code such computations on trees in an object-oriented style, using Java.

(*Note:* This lab is adapted from Prof. Andrew Tolmach's earlier version.)

Preparation

Download and unzip the file `lab7.zip`. You'll see a set of sub-directories, `00-05`. Each contains a Java program called `Example.java`. In all programs, simple binary trees are used. To compile and run the contents of each directory, `cd` to it and type

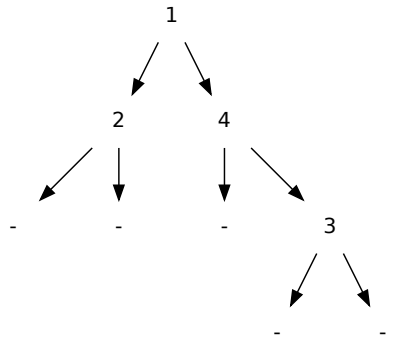
```
linux> javac Example.java
linux> java Example
```

You'll walk through these directories, in sequence.

00. Procedural Traversal

We begin with a program containing a very simple data type definition for binary trees with integers at the internal nodes (only), and code to calculate the sum of values in a tree.

Abstractly, there are two kinds of nodes in such a tree: internal nodes, which carry an integer value and pointers to two subtrees, and and leaf nodes, which carry no information. An example tree is shown on the right; its values sum to 10.



"from 00/Example.java"

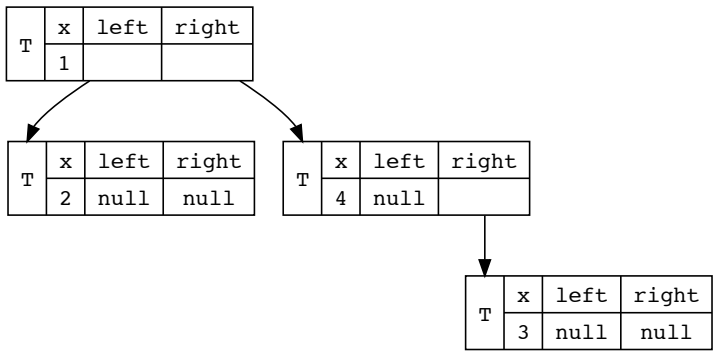
```java
class T {
  int x;        // node value
  T left;       // child
  T right;      // child
  T (int x, T left, T right) {
    this.x = x; this.left = left; this.right = right;
  }
}
```

"from 00/Example.java"

```
16 class Example {
17   public static void main (String argv[]) {
18     T t = new T (1, new T(2, null, null),
19                     new T(4, null, new T(3, null, null)));
20     System.out.println ("sum = " + sum(t));
21   }
22
23   static int sum (T t) {
24     if (t != null)
25       return t.x + sum(t.left) + sum(t.right);
26     else
27       return 0;
28   }
29 }
```

Some things to note about the code:

- Lines 7–14 define a class `T` to represent internal nodes, with the obvious instance variables. Recall that the instance variables of type `T` are actually *pointers* to objects of type `T`. Otherwise, this code looks essentially the same as the corresponding `struct` declaration in C.
- Unfortunately, Java requires us to write out a constructor for every class (lines 11–13), even when we want the "obvious" one given here. Note that in the constructor, we can use `this.x` to refer to the `x` instance variable; this lets us use `x` as the name of the corresponding constructor parameter too, saving us from needing to think up a different name for it. We'll use this convention consistently. Warning: when writing down trivial constructors like this, it is sadly easy to make silly mistakes that don't get caught by the Java compiler.
- By convention, we use the special Java constant `null` to represent (all) leaf nodes. This is a standard Java trick, taking advantage of the fact that `null` is a legal value of every class type.
- The `sum` function (lines 23–28) is written as an ordinary recursive procedure traversing the tree, passing the current node as a parameter. It does not use any object-oriented features (e.g., we could write essentially the same code in C). Question: in this traversal, in what order are nodes visited, and when do the additions actually take place?
- Note that we detect leaves by testing explicitly for `null` (at line 24); happily, this test also guards all the dereferences of `t` (which are all on line 25), so we expect no runtime errors due to null pointer dereferences.
- The `Example` class defines a `main` method that tests `sum` on a very simple example tree, built directly using constructors (lines 17–21), and prints out the result. In this case, the example tree corresponds to the one shown above. The formatting is just an aid to visualizing the tree shape; it is irrelevant to the Java compiler. The physical data structure Java builds for `t` looks like this:



01. Object-Oriented Traversal

Here is a more "object-oriented" version of the same program.

"from 01/Example.java"

```
7  class T {
8    private int x;
9    private T left;
10   private T right;
11   T (int x, T left, T right) {
12     this.x = x; this.left = left; this.right = right;
13   }
14
15   int sum () {
16     return x + (left != null ? left.sum() : 0) +
17                (right != null ? right.sum() : 0);
18   }
19 }
20
21 class Example {
22   public static void main (String argv[]) {
23     T t = new T (1, new T(2, null, null),
24                     new T(4, null, new T(3, null, null)));
25     System.out.println ("sum = " + t.sum());
26   }
27 }
```

- The definition of the tree node data structure is almost the same, but `sum` is now defined as a *method* of the class `T`. It is invoked by applying the method to a tree instance (e.g. `t.sum()` in line 25), rather than by passing the tree instance as an argument.
- Informally speaking, in this version we ask the tree "please tell me your sum," and it calculates its answer by in turn asking its children to tell it *their* sums, and so on. This is quite different from the version of `sum` in example 00, which worked by inspecting the tree's internal data. To emphasize this difference, the fields of `T` are now marked `private`, so the `sum` code in example 00 actually wouldn't compile against this definition.
- In attribute grammar terminology, we can view this code as computing a synthesized *sum* attribute value on all `T` nodes, given pre-existing x attributes and using a kind of artificial grammar rule $T \rightarrow TT$ and the attribute equation $T.sum := T.x + T_1.sum + T_2.sum$.

Note that in this case we are not actually storing the values of the computed *sum* attribute in the tree nodes, because all we really want is its value at the root.

- Unfortunately, the code in `sum` has to pay special attention (lines 16–17) to the cases where the child nodes are null. This makes the code more complicated, and damages the nice "just ask the children" metaphor. We will see how to fix this shortly.

Exercise: Computing Tree Size Modify the code so that it computes and prints out the *size* of a tree (defined as the number of non-leaf nodes) rather than the sum of its values. (You may want to copy the program to a new file, say `Exercise.java`, so that you can keep the original version.)

Hint: Only very small changes are required!

02. Multiple Node Types

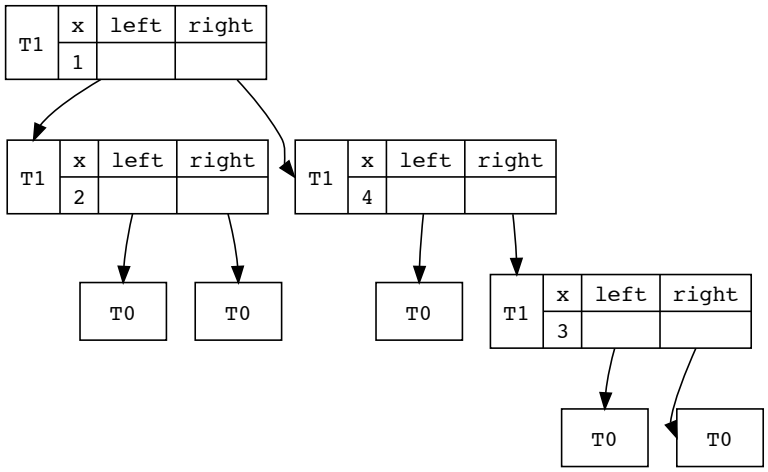
We can fix the ugliness associated with `null` values by using Java's *sub-classing* mechanism. The key idea is to introduce two separate sub-classes of `T` corresponding to internal and leaf nodes. This is also a good first step towards generalizing to trees with many different kinds of nodes.

"from 02/Example.java"

```
7  abstract class T {
8    abstract int sum();
9  }
10
11 class T0 extends T {
12   int sum() {
13     return 0;
14   }
15 }
16
17 class T1 extends T {
18   private int x;
19   private T left;
20   private T right;
21   T1 (int x, T left, T right) {
22       this.x = x; this.left = left; this.right = right; }
23
24   int sum() {
25     return x + left.sum() + right.sum();
26   }
27 }
28
29 class Example {
30   public static void main (String argv[]) {
31     T t = new T1 (1, new T1(2, new T0(), new T0()),
32                      new T1(4, new T0(), new T1(3, new T0(), new T0())));
33     System.out.println ("sum = " + t.sum());
34   }
35 }
```

- We redefine `T` to be an `abstract` class (lines 7–9), meaning that it is not actually instantiated. We give it two sub-classes (specified using `extends` on lines 11 and 17): `T0` for nodes with no values or children (leaves) and `T1` for nodes with one value and two children (internal nodes). Crucially, the instance variables representing children are declared (lines 19–20) to be of the super-class `T` , so they can filled with *either* `T0` or `T1` values.

- By convention, we no longer use `null` values, so every `T1` node is assumed to have two genuine children. Unfortunately, there is no way to tell Java that we wish to disallow `null` values here (indeed, this is a major flaw in Java's design).
- Although the abstract view of the example tree is the same as before, the corresponding physical data structure constructed in Java is different:



  Note that that identity of the constructing class (`T0` or `T1`) is recorded as a tag within the physical representation of each node.

- Class `T0` is somewhat "degenerate" in that it doesn't have any fields; hence, we do not need to write a constructor definition for it. Its instances are still useful, because they carry class tags, but all instances are essentially equivalent; there's usually no good reason to create more than one. In this situation, it is common to add a declaration like this in class `T0`:

  ```
  static T0 the_T0 = new T0();
  ```

  and then use `T0.the_T0` everywhere in place of calls to `new T0()`. But note that this would result in a different physical data layout in Java's memory, and programs can tell the difference between the two layouts because the equality operator (==) will reply `false` if given two different instances.

- As before, `sum` is an instance method defined for all `T` objects; this is the effect of the `abstract` method definition in class `T`. But now there are two *different* versions of `sum`, one for each sub-class. When `sum` is invoked on a particular object instance, the executing Java code performs a *dynamic dispatch*: it uses the object's class tag to determine which version of `sum` to execute.
- Finally, note that because of our convention about `null`, the code in `T1` no longer has to worry about whether its children are present. Now we really can just "ask the children." The attribute grammar interpretation is the same as for example 01, but now the code for `sum` in `T1` (lines 24–26) looks almost exactly like the attribute equation $T.sum := T.x + T_1.sum + T_2.sum$. The code for `sum` in `T0` (lines 12-14) corresponds to the attribute equation $T.sum := 0$.

Exercise: Extend the Possible Node Types Modify the code to add a third kind of tree node, `T2`, that carries *two* integer values and *three* children. Your revised `sum` function should add in the values of both integers from these nodes. Change the example tree to test the behavior of your new code.

Hint: You should be able to add support for `T2` nodes without changing any of the existing code (except for the test tree).

03. Richer Attribute Domains

Returning to the trees with node types `T0` and `T1` from example 02, consider a new problem: computing how many distinct integers occur in the tree.

"from 03/Example.java"

```
abstract class T {
  abstract ImmIntSet vals();
}

class T0 extends T {
  ImmIntSet vals() {
    return new ImmIntSet();
  }
}

class T1 extends T {
  private int x;
  private T left;
  private T right;
  T1 (int x, T left, T right) {
    this.x = x; this.left = left; this.right = right; }

  ImmIntSet vals() {
    return ImmIntSet.add(ImmIntSet.union(left.vals(), right.vals()), x);
  }
}
```

- A simple solution to this problem is to compute at each node the *set* of distinct values, called `vals()`, appearing in the subtree rooted at that node. Then, to find the number of distinct values in the entire tree, we can just take the cardinality of the `vals` computed for the overall root node: e.g., `t.vals().size()`.
- The cleanest approach to implementing `vals()` is to treat sets as *immutable*, i.e., rather than altering the contents of a given set over time, we create new sets when we need to, without altering the existing sets. This style is often called *functional*, because it allows us to write code that behaves like pure mathematical functions.
- From an attribute grammar perspective, we can view this code as computing the synthesized attribute *vals* on `T` nodes, with these equations:

  $T.vals := \{\}$ (for `T0` nodes)
  $T.vals := \{T.x\} \cup T_1.vals \cup T_2.vals$ (for `T1` nodes)

- A class for representing immutable integer sets is included in this program. It is defined as an extention to Java library's mutable `HashSet` class. For each of the three set operations, union, intersection, and add, a new set object is created to hold the result.

"from 03/Example.java"

```
class ImmIntSet extends HashSet<Integer> {
  public static ImmIntSet union(ImmIntSet s1, ImmIntSet s2) {
    ImmIntSet s = new ImmIntSet();
    s.addAll(s1);
    s.addAll(s2);
    return s;
  }

  public static ImmIntSet intersect(ImmIntSet s1, ImmIntSet s2) {
    ImmIntSet s = new ImmIntSet();
```

```
    for (int x : s1)
      if (s2.contains(x))
        s.add(x);
    return s;
  }

  public static ImmIntSet add(ImmIntSet s0, int x) {
    ImmIntSet s = new ImmIntSet();
    s.addAll(s0);
    s.add(x);
    return s;
  }
}
```

04. Using Imperative Sets

Operations on immutable sets are inherently more expensive than on mutable sets because the former are more powerful: each operation leaves its arguments unchanged, so they can be used again. In this particular problem, we don't actually need this extra power, so building up a single set by imperative update operations should be substantially more efficient. For this, we can use the Java library's `HashSet` class directly (imported on line 7).

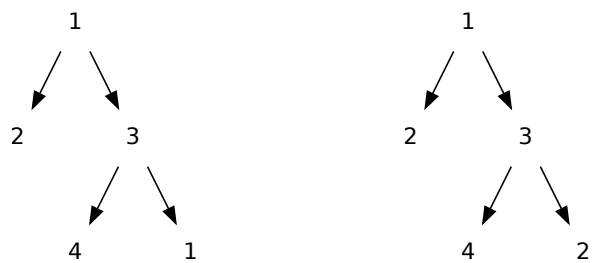
"from 04/Example.java"

```
9  abstract class T {
10   static Set<Integer> s;
11   abstract void addToVals();
12   static Set<Integer> vals(T t) {
13     s = new HashSet<Integer>();
14     t.addToVals();
15     return s;
16   }
17 }
18
19 class T0 extends T {
20   private int x;
21   T0 (int x) { this.x = x; }
22
23   void addToVals() {
24     s.add(x);
25   }
26 }
27
28 class T1 extends T {
29   private int x;
30   private T left;
31   private T right;
32   T1 (int x, T left, T right) {
33     this.x = x; this.left = left; this.right = right; }
34
35   void addToVals() {
36     s.add(x);
37     left.addToVals();
38     right.addToVals();
39   }
40 }
```

The hard work is done by an auxiliary instance method, `addToVals()`, which is defined for each node type. The wrapper function, `vals()`, is defined in `T` (lines 12–16), and can be invoked by, e.g., `T.vals(t)`.

05. Finding Duplicates on a Path

Now on to another problem: to compute and print out a boolean value calculated over the tree, which is true iff there is some path from root to leaf that contains the same value twice. For example, the tree on the left has a path with duplicates (the right-most one), but the tree on the right does not (even though there are two nodes containing 2).



"from 05/Example.java"

```
abstract class T {
  abstract ImmIntSet vals();
  abstract boolean hasDups();
}

class T0 extends T {
  ImmIntSet vals() {
    return new ImmIntSet();
  }
  boolean hasDups() {
    return false;
  }
}

class T1 extends T {
  private int x;
  private T left;
  private T right;
  T1 (int x, T left, T right) {
    this.x = x; this.left = left; this.right = right; }

  ImmIntSet vals() {
    return ImmIntSet.add(ImmIntSet.union(left.vals(), right.vals()), x);
  }
  boolean hasDups() {
    return left.hasDups() || right.hasDups() ||
           left.vals().contains(x) || right.vals().contains(x);
  }
}
```

- This simple solution directly uses the `vals()` function we defined in example 03. We could instead use the imperative version of the `vals()` function from example 04, which would probably be more efficient.

- From an attribute grammar perspective, we can think of this code as defining a new synthesized attribute *hasDups* (in addition to *vals*), with attribute equations:

  $T.hasDups := false$ (for `T0` nodes)
  $T.hasDups := T_1.hasDups \vee T_2.hasDups \vee T.x \in T_1.vals \vee T.x \in T_2.vals$ (for `T1` nodes)

- This code illustrates a useful, but slightly tricky, feature of Java. The language makes a fundamental distinction between *primitive* types, like `int` and `boolean`, and *object* types, which correspond to instances of classes like `T` and `T0`. For each primitive type, the Java standard library defines a corresponding *wrapper* class, such as `Integer`, `Boolean`, etc. Each instance of a wrapper class contains just a single value of the underlying primitive type. Wrappers are convenient for allowing primitive values to be used in situations where an object is required, for example in collections like `ImmIntSet`. To create a wrapped value from a primitive value, we can use the static `valueOf` method; to convert the other way, we can use an instance method such as `intValue`. For example:

  ```
  int i = 42;
  Integer v = Integer.valueOf(i);
  int j = v.intValue();
  System.out.println(j == i);  // prints true
  ```

  (We could also use a constructor to create the `Integer`, but that will always generate a completely fresh object, whereas `valueOf` can cache and re-use existing objects.)

  But observe that at lines 61 and 65, we pass bare `int`'s to `ImmIntSet`'s `add()` and `contains()` methods where `Integer`'s are expected. How does this type check? Answer: the Java compiler is automatically inserting calls to `Integer.valueOf` around the bare `int`s. The compiler performs this *autoboxing* process whenever a primitive value is passed to a method expecting an object or assigned to a variable declared as an object. (In fact, this was already happening at line 57 of example 03.) The compiler also performs automatic *unboxing* in the other direction, inserting `intValue` calls where needed, as will be seen in later examples.

Exercise: Marking Duplicates Consider the following variant of the duplicates-on-a-path problem from Example 06: instead of computing whether a duplicate exists on some path in the tree, we want to add a flag field to every node and set it to `true` iff the value of that node duplicates some value on the path from that node up to the root.

The template for the code is given in `Exercise.java`. You need to complete the definition of function `setDups` so that it sets the `isDup` flags correctly on all nodes. The key idea is to pass a suitable set *downwards* to each node. In attribute grammar terminology, you want to use an *inherited* set attribute. (Hint: The use of immutable sets is essential here.)

The printing code has been added so that you can check your results. For example, the output for the given test should be:

```
linux> java Exercise
1 false
 3 false
  4 false
  3 true
 1 true
```