Computations on Trees: `treelab`

CS 321 Languages and Compiler Design I, Winter Term 2015
Department of Computer Science, Portland State University
(Credit: Original lab developed by Prof. Andrew Tolmach)

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, especially ASTs.
- Code such computations on trees in an object-oriented style, using Java.
- Understand and apply the terminology of *attribute grammars* for defining computations over ASTs and parse trees.

Instructions

Download and unzip the file `treelab.zip`. You'll see a set of sub-directories, 00-10a. Each contains a Java program called `Example.java`; some directories contain other Java files as well. To compile and run the contents of each directory, `cd` to it and type

```
$ javac Example.java
$ java Example
```

We'll walk through many or all of these directories, in sequence. Some directories contain exercises; each of these directories also contains a subdirectory called `solution`. Do not look at the solution until you have attempted the exercise yourself! Directories whose names ends in `a` are auxiliary; they contain less essential material that may be skipped during the lab, if time is limited.

Computations over Trees

Trees are a pervasive data structure in computing, and particularly inside compilers. We have already noted the central role played by *abstract syntax trees (ASTs)*, which are an important internal representation of programs within the compiler. Trees are the natural way to represent programs in most high-level languages, because these languages have *recursive* structure: expressions can be built up out of arbitrary sub-expressions, statements out of arbitrary sub-statements (and expressions), and so on.

Once we have an AST represented in the compiler's memory, we need to perform *computations* over it, such as static analysis, interpretation, and code generation. (We will study static analysis in the last homework this term, and the other two topics next term.) In this lab, we will study how to organize such computations in general. In particular, we will explore a programming paradigm in which we try to keep computations as *localized* as possible, that is, to express the computation over the whole tree by combining small, largely

independent computations at each node of the tree. The order in which computations are combined is given by a tree *traversal* strategy; we will focus on strategies that can be easily coded using recursive descent.

We have also considered *parse trees*, which are induced by parsing the concrete syntax of programs. In the current parsing project, we are not actually constructing parse trees as explicit data structures within the compiler, choosing instead to build ASTs directly by executing actions during parsing. An alternative, used by some compilers, is to build a concrete parse tree data structure first, and then traverse over it to build the AST. Although we don't do that for the current project, we can still think of our parser as building a *virtual* (or imaginary) parse tree: instead of constructing concrete nodes, the data fields that would go in those nodes are passed directly to actions that construct the AST. In effect, it is as though the parse tree construction and traversal steps are fused together.

Lab Strategy

Since ASTs aren't fundamentally different from other kinds of trees—they just happen to have fairly large numbers of different node types and sometimes messy features like variable numbers of children—the issues involved in computing over them in Java can be explored in a simpler setting. So most of this lab will focus on simple binary trees carrying integer data; the last few examples look at simple ASTs and some of the specialized issues they raise.

Attribute Grammars

There is a well-established formalism called *attribute grammars (AGs)* for describing localized computations over (real or virtual) parse trees. This formalism is used in many textbooks, so it is worth being familiar with its basic terminology. (Although normally defined on parse trees, the same terminology can also be applied to ASTs, and indeed to trees in general.)

- An *attribute* is simply a piece of data associated with the tree nodes of a particular type. For example, nodes representing expressions might have attributes `type` or `value`.
- The values of attributes are defined by *attribute equations* relating them to other attributes of the node, its immediate children, or its immediate parent; thus, attribute definitions are intrinsically local in nature. For example, given the grammar production

  $exp \rightarrow exp - exp$

  we might have the associated attribute equation

  $exp.val := exp_1.val - exp_2.val$

  which states that the value (val) of the expression can be computed by subtracting the val of its second child from the val of its first child. (The subscripts are used to distinguish the two different exp children on the right-hand side of the production.)
- An attribute is *synthesized* if it depends only on the attributes of its descendents; it is *inherited* if its depends only on the attributes of its ancestors. In general, synthesized attributes can be computed by a bottom-up pass over the tree, and inherited ones by a top-down pass; a single recursive descent traversal can compute both. (To get the computation started, the values of any inherited attributes at the root node and of any synthesized attributes at leaf nodes must already be known.)
- The AG formalism doesn't specify whether or not attribute values are actually stored at tree nodes once they have been computed. This is a decision we have to make when turning AGs into real code. Storing attribute values takes space and may complicate our tree datatype definitions, but can save lots of time if an attribute value is used multiple times.

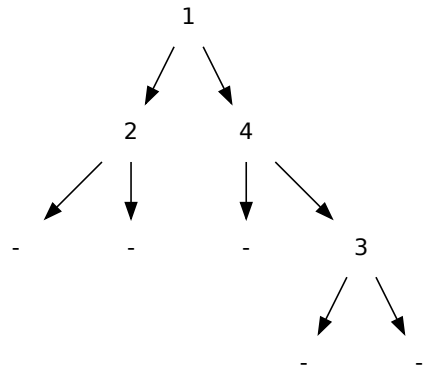
- In their purest form, AGs are *functional*; that is, attributes equations are just mathematical functions relating attribute values. Such AGs have the nice property that attributes can be evaluated in *any* order consistent with the dependencies among attribute values.
- However, it is common to extend the AG formalism to allow arbitrary *imperative* code, e.g. reading or writing global variables, or even performing output. In this setting, it is important to know the order of node evaluation (e.g., top-down, left-to-right).
- To compute a value over the whole tree, we can define a synthesized attribute for the root node type, evaluate it everywhere, and then read off its value at the root.
- When the dependency relationships among attributes permit, it may be possible to evaluate attributes "on the fly" during parsing without constructing a concrete parse tree data structure first. For example, our top-down parser can be viewed as computing the AST corresponding to each subtree as a synthesized attribute; and the AST for the whole program corresponds to the value of that attribute at the root. In general, a top-down recursive descent parser can evaluate both inherited and synthesized attributes on the fly; a bottom-up parser can only evaluate synthesized attributes (although special tricks can be played to loosen this restriction in certain cases).

As we progress through the examples in this lab, we'll point out how attribute grammar terminology can be used to give a high-level specification for what we're doing.

Procedural Traversal of Simple Binary Trees (00)

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.



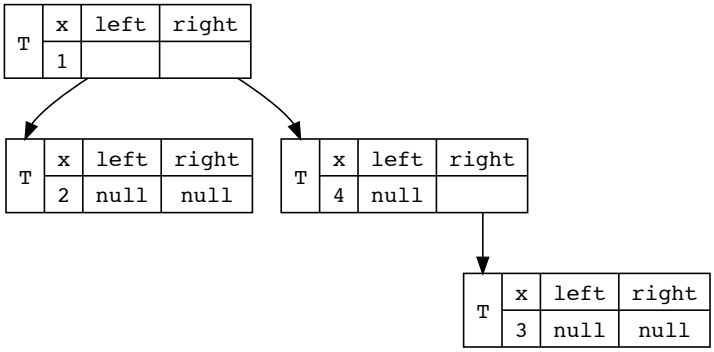
00/Example.java

```
1  class T {
2    int x;       // node value
3    T left;      // child
4    T right;     // child
5    T (int x, T left, T right) {
6      this.x = x; this.left = left; this.right = right;
7    }
8  }
```

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

Some things to note about the code:

- Lines 1–8 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 5–7), 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. (We'll revisit this choice soon.)
- The `sum` function (lines 20–25) 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 21); happily, this test also guards all the dereferences of `t` (which are all on line 22), 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 12–16), 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:



Object-oriented Traversal of Simple Binary Trees (01)

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

01/Example.java

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

- 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 22), 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 10–11) 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.

Imperative Computation over Simple Binary Trees (01a)

Another way to compute the sum is to accumulate it in a global variable which is updated as each node is visited.

01a/Example.java

```
1  class T {
2    private int x;
3    private T left;
4    private T right;
5    T (int x, T left, T right) {
6      this.x = x; this.left = left; this.right = right;
7    }
8
9    private static int s;
10
11   private void addToSum() {
12     s += x;
13     if (left != null)
14       left.addToSum();
15     if (right != null)
16       right.addToSum();
17   }
18
19   static int sum(T t) {
20     s = 0;
21     t.addToSum();
22     return s;
23   }
24 }
```

- The sum is accumulated into static variable `s` (line 9); recall that there is only one copy of each static variable, rather than a separate copy per object instance.

- The hard work is done by the instance method `addToSum` (lines 11–17); the static method `sum` (lines 19–23) is just a wrapper that initializes the accumulation variable, applies the worker method to the root, and returns the final accumulated value. Recall that static methods are not associated with a particular instance (and so cannot examine instance variables); they can be invoked from outside the class by prefixing with the class name (e.g. `T.sum(t)`).

- For this simple problem, using an imperative approach like this is verbose and awkward; the other solutions we've seen are better. But later on, we'll see circumstances where this approach can be much more efficient, so it is worth keeping in mind.

Exercise: Computing Tree Size (02)

Modify the code from example 01 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.

Hint: Only very small changes are required!

Multiple Node Types (03)

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.

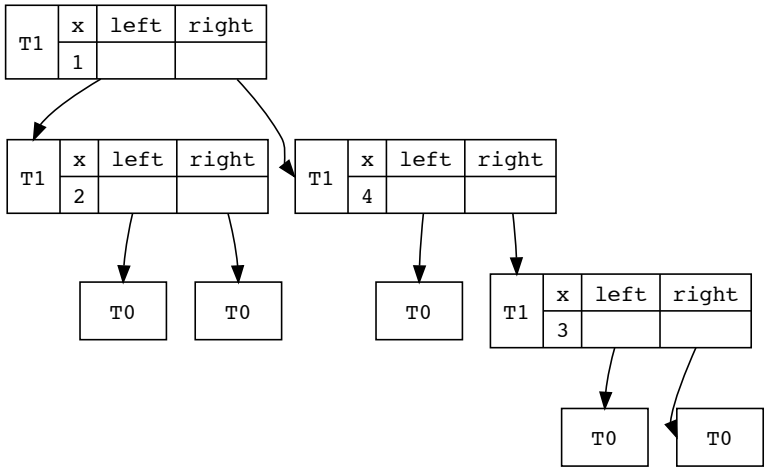
03/Example.java

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

- We redefine `T` to be an `abstract` class (lines 1–3), meaning that it is not actually instantiated. We give it two sub-classes (specified using `extends` on lines 5 and 14): `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 13–14) 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 18–20) looks almost exactly like the attribute equation $T.sum := T.x + T_1.sum + T_2.sum$. The code for `sum` in T0 (lines 6-8) corresponds to the attribute equation $T.sum := 0$.

Avoiding Dynamic Dispatch (03a)

Can we use the same sub-classing mechanism to define trees, but revert to writing `sum` as a non-OO procedure (as in example 00) ? Yes, but...

03a/Example.java

```
26  static int sum (T t) {
27    if (t instanceof T0)
28      return 0;
29    else if (t instanceof T1) {
30      T1 t1 = (T1) t;
31      return t1.x + sum(t1.left) + sum(t1.right);
32    } else
33      throw new Error("impossible");
34  }
```

- Java provide an boolean-valued expression form e `instanceof` c which evaluates to true just when the class tag on object e is c (or a sub-class of c).
- But using this feature makes code ugly! The problems are two-fold. First, the `instanceof` test only checks against one class at a time, so if we want to test against a series of possible sub-classes, we have to write an `if-then-else` chain, which is verbose and inefficient. Secondly, once we have matched e successfully against a class c, we have to explicitly *down-cast* e to c before accessing its fields (e.g. line 30), which again is verbose and inefficient (since the down-cast requires a runtime check), and can cause runtime exceptions if we make a mistake.
- Notice that control can never actually pass to line 33, where an exception is thrown, because `t` must either a `T0` or a `T1`. But there is no way for the Java compiler to know this for sure, because it has to consider the possibility that we might define further subclasses of `T` in other files.
- Overall, although there are times when `instanceof` is very useful, in general it should be avoided if a cleaner OO-style solution is available.

Exercise: Extend the Possible Node Types (04)

Modify the code from example 03 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).

Richer Attribute Domains (05)

Returning to the trees with node types T0 and T1 from example 03, consider a new problem: computing how many distinct integers occur in the tree.

05/Example.java

```
1  abstract class T {
2    abstract ImmSet<Integer> contents();
3  }
4
5  class T0 extends T {
```

```
6       ImmSet<Integer> contents() {
7         return ImmSet.<Integer>empty();
8       }
9     }
10
11    class T1 extends T {
12      private int x;
13      private T left;
14      private T right;
15      T1 (int x, T left, T right) {
16        this.x = x; this.left = left; this.right = right; }
17
18      ImmSet<Integer> contents() {
19        return ImmSet.plus(ImmSet.union(left.contents(),
20                                        right.contents()),
21                           x);
22      }
23    }
```

- A simple solution to this problem is to compute at each node the *set* of distinct values, called `contents()`, 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 contents computed for the overall root node: e.g., `t.contents().size()`.
- The cleanest approach to implementing `contents()` 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 *contents* on `T` nodes, with these equations:

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

- A small library of functions for operating on immutable sets is provided in the auxiliary file `ImmSet.java`. It uses the Java library's `HashSet` as the underlying set representation. (The library is *generic* over the type of elements in the set; don't worry if you don't fully understand this aspect of Java, since we'll discuss it in more detail later in the term.) This code is *not* a very efficient implementation of immutable sets, but it has the merit of simplicity.

05/ImmSet.java

```
17    public class ImmSet<T> {
18      private Set<T> s;   // the underlying set
19
20      private ImmSet(Set<T> s) {
21        this.s = s;
22      }
23
24      // Set Construction
25
26      // To call this, write something like ImmSet.<Integer>empty().
27      public static <T> ImmSet<T> empty() {
28        return new ImmSet<T>(new HashSet<T>());
29      }
30
```

```
31      public static <T> ImmSet<T> singleton(T x) {
32        Set<T> s = new HashSet<T>();
33        s.add(x);
34        return new ImmSet<T>(s);
35      }
36
37      public static <T> ImmSet<T> plus(ImmSet<T> a, T x) {
38        Set<T> s = new HashSet<T>(a.s);
39        s.add(x);
40        return new ImmSet<T>(s);
41      }
42
43      public static <T> ImmSet<T> union(ImmSet<T> a1,ImmSet<T> a2) {
44        Set<T> s = new HashSet<T>(a1.s);
45        s.addAll(a2.s);
46        return new ImmSet<T>(s);
47      }
48
49      public static <T> ImmSet<T> intersect(ImmSet<T> a1,ImmSet<T> a2) {
50        Set<T> s = new HashSet<T>();
51        for (T x : a1.s)
52          if (a2.s.contains(x))
53            s.add(x);
54        return new ImmSet<T>(s);
55      }
56
57      // Set Inspection
58
59      public boolean contains(T x) {
60        return s.contains(x);
61      }
62
63      public int size() {
64        return s.size();
65      }
66    }
```

Using Imperative Sets (05a)

Even with a cleverer representation than the one in `ImmSet.java`, 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 1).

05a/Example.java

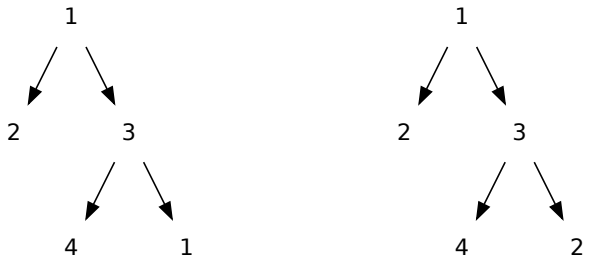
```
1   import java.util.*;
2
3   abstract class T {
4     static Set<Integer> s;
5     abstract void addToContents();
6     static Set<Integer> contents(T t) {
7       s = new HashSet<Integer>();
8       t.addToContents();
```

```
9         return s;
10      }
11    }
12
13    class T0 extends T {
14      private int x;
15      T0 (int x) { this.x = x; }
16
17      void addToContents() {
18        s.add(x);
19      }
20    }
21
22    class T2 extends T {
23      private int x;
24      private T left;
25      private T right;
26      T2 (int x, T left, T right) { this.x = x; this.left = left; this.right = right; }
27
28      void addToContents() {
29        s.add(x);
30        left.addToContents();
31        right.addToContents();
32      }
33    }
```

This code follows the pattern of example 01a. Again, the hard work is done by an auxiliary instance method, `addToContents()`, which is defined for each node type. The wrapper function, `contents()`, is defined in `T` (lines 6–10), and can be invoked by, e.g., `T.contents(t)`.

Finding Duplicates on a Path (06)

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).



06/Example.java

```
1   abstract class T {
2     abstract ImmSet<Integer> contents();
```

```
3       abstract boolean hasDups();
4     }
5
6     class T0 extends T {
7       ImmSet<Integer> contents() {
8         return ImmSet.<Integer>empty();
9       }
10      boolean hasDups() {
11        return false;
12      }
13    }
14
15    class T1 extends T {
16      private int x;
17      private T left;
18      private T right;
19      T1 (int x, T left, T right) { this.x = x; this.left = left; this.right = right; }
20
21      ImmSet<Integer> contents() {
22        return ImmSet.plus(ImmSet.union(left.contents(),
23                                        right.contents()),
24                           x);
25      }
26      boolean hasDups() {
27        return left.hasDups() || right.hasDups() ||
28               left.contents().contains(x) || right.contents().contains(x);
29      }
30    }
```

- This simple solution directly uses the `contents()` function we defined in example 05. We could instead use the imperative version of the `contents()` function from example 05a, 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 *contents*), with attribute equations:

  $T.hasDups := false$ (for T0 nodes)
  $T.hasDups := T_1.hasDups \vee T_2.hasDups \vee T.x \in T_1.contents \vee T.x \in T_2.contents$ (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 `ImmSet`. 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 24 and 28, we pass bare `int`'s to `ImmSet.plus` and `ImmSet.contains` 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 21 of example 05.) The compiler also performs automatic *unboxing* in the other direction, inserting `intValue` calls where needed, as will be seen in later examples.

Computing Multiple Results at a Node (06a)

The code in example 06 is not very efficient asymptotically, because it visits every node in the tree multiple times and does a great deal of repeated work. (Why?) Ideally, we would like to visit each node just once. We can achieve that goal by reimplementing the computation of *hasDups* and *contents* as a single traversal function that returns a *pair* of results for each node.

06a/Example.java

```
5   // Define a class to carry a pair of values.
6   class DC {
7     boolean hasDups;
8     ImmSet<Integer> contents;
9     DC (boolean hasDups, ImmSet<Integer> contents) {
10      this.hasDups = hasDups; this.contents = contents;
11    }
12  }
13
14  abstract class T {
15    abstract DC reduce();
16  }
17
18  class T0 extends T {
19    DC reduce() {
20      return new DC(false, ImmSet.<Integer>empty());
21    }
22  }
23
24  class T1 extends T {
25    private int x;
26    private T left;
27    private T right;
28    T1 (int x, T left, T right) { this.x = x; this.left = left; this.right = right; }
29
30    DC reduce() {
31      DC l = left.reduce();
32      DC r = right.reduce();
33      boolean hasDups = l.hasDups || r.hasDups ||
34                        l.contents.contains(x) || r.contents.contains(x);
35      ImmSet<Integer> contents = ImmSet.plus(ImmSet.union(l.contents,
36                                                          r.contents),
37                                             x);
38      return new DC(hasDups,contents);
39    }
40  }
```

- The code to calculate `hasDups` and `contents` at each node is essentially unchanged from example

06; indeed, the attribute grammar equations are exactly the same. The important difference has to do with the traversal strategy used to compute these attributes. By returning both pieces of information at the same time, we can compute them on the same visit.

- The only coding challenge is how to return multiple values from a Java function. The general answer is to define a fresh class for the purpose, here called `DC`. (Unfortunately, Java doesn't provide generic classes for pairs, triples, etc.)
- Since the name and fields of `DC` already explain what our recursive traversal function does, we have called it `reduce`, a common conventional name for functions that compute bottom-up over a tree. Another conventional name for functions of this kind is `fold`.
- Another approach to avoiding multiple traversals would be to *cache* the results of the `contents()` method in a new field `ccontents` associated with each node. Initially, `ccontents` is set to `null`. Whenever `contents()` is first called on a node, it detects that `ccontents` is null, computes its answer as usual and then overwrites `ccontents` with that answer before returning it; subsequent calls to `contents()` on that node can just use `ccontents`. In AG terms, this corresponds to storing the *contents* attribute explicitly rather than just computing it.

Exercise: Marking Duplicates (07)

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. 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:

```
$ java Example
1 false
 3 false
  4 false
  3 true
 1 true
```

Expression ASTs (08)

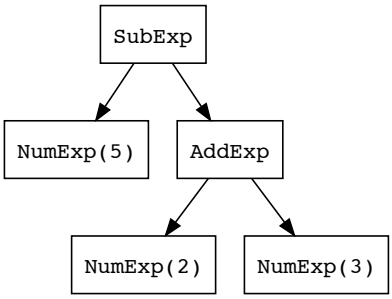
Now let's see how these tree traversal methods apply to ASTs. We start with a simple language of expressions involving numeric literals, addition, and subtraction, each represented by a different sub-class of class `Exp`. We define an `eval()` function that evaluates an expression tree to an integer; this would be suitable for use in an interpreter for the language.

08/Example.java

```
0  abstract class Exp {
1    abstract int eval();
2  }
3
4  class NumExp extends Exp {
```

```
5       private int num;
6       NumExp(int num) {this.num = num;}
7
8       int eval() {
9         return num;
10      }
11    }
12
13    class AddExp extends Exp {
14      private Exp left;
15      private Exp right;
16      AddExp (Exp left, Exp right) {this.left = left; this.right = right;}
17
18      int eval() {
19        return left.eval() + right.eval();
20      }
21    }
22
23    class SubExp extends Exp {
24      private Exp left;
25      private Exp right;
26      SubExp (Exp left, Exp right) {this.left = left; this.right = right;}
27
28      int eval() {
29        return left.eval() - right.eval();
30      }
31    }
32
33    class Example {
34      public static void main(String argv[]) {
35        Exp e = new SubExp(new NumExp(5),
36                           new AddExp(new NumExp(2),
37                                      new NumExp(3)));
38
39        System.out.println("value = " + e.eval());
40      }
41    }
```

- The following AST, which corresponds to the definition of `e` at lines 35–37, should evaluate to 0.



- `eval()` is implemented as just another recursive descent tree traversal, computing its answer bottom-

up. In attribute grammar terminology, it corresponds to a synthesized attribute val, with the following informal grammar and attribute equations:

$exp \to num \quad exp.val := num.val$
$exp \to exp + exp \quad exp.val := exp_1.val + exp_2.val$
$exp \to exp - exp \quad exp.val := exp_1.val - exp_2.val$

Here we assume that num nodes already have a pre-set val attribute; in a compiler, this would probably be attached to the numeric literal token returned by the lexical analyzer. As in the previous examples, the code doesn't actually store the val attribute at each node, because we are only interested in its value at the root.

- Most of the other functions we want to perform on ASTs have a similar structure, as we'll see in the coming weeks.

Exercise: Adding a Conditional Expression (09)

Extend the expression language with support for `ifnz` expressions and their evaluation. An `ifnz` expression has three sub-expressions $test$, nz, and z; its concrete syntax might be written `(`$test$ `?` nz `:` z`)` to match the conditional expression found in Java, C, and C++. If $test$ evaluates to 0, the `ifnz` expression as a whole evaluates to z; otherwise it evaluates to nz.

For example, the expression `((5-(2+3)) ? 10 : (21+21))` should evaluate to 42.

Hint: You should be able to add support for `ifnz` nodes without changing any of the existing code (except that you'll want to change the test tree).

Adding Variables and Environments (10)

Now consider what happens when we add *variables* to our expression language.

10/Example.java

```
1  abstract class Exp {
2    abstract int eval(ImmMap<String,Integer> env);
3  }
4
5  class LetExp extends Exp {
6    String x;
7    Exp d;
8    Exp e;
9    LetExp(String x, Exp d, Exp e) {this.x = x; this.d = d; this.e = e;}
10
11   int eval(ImmMap<String,Integer> env) {
12     int v = d.eval(env);
13     return e.eval(ImmMap.extend(x,v,env));
14   }
15 }
16
17 class VarExp extends Exp {
18   String x;
19   VarExp(String x) {this.x = x;}
20
21   int eval(ImmMap<String,Integer> env) {
22     Integer v = env.get(x);
23     if (v != null)
```

```
24          return v;
25        else
26          return 0;  // default value for undefined variables
27      }
28    }
29
30    class NumExp extends Exp {
31      int num;
32      NumExp(int num) {this.num = num;}
33
34      int eval(ImmMap<String,Integer> env) {
35        return num;
36      }
37    }
```

- Variables are identifiers (represented here as `Strings`) that carry an associated (integer) value.
- Uses of variables become another kind of leaf node (`VarExp`) in expressions. A variable evaluates to its value.
- To define variables and give them values, we use a new `LetExp` expression node, which has an associated variable x and two sub-expressions d and e; its concrete syntax might be written

  `let` x `=` d `in` e

  It is evaluated by first evaluating d, *binding* the resulting value to variable x and then evaluating e, whose value becomes the result of the entire `let` form.
- The value of a variable cannot be changed after its initial definition (so in fact "constant" might be a better name than "variable.")
- A key point is that the *scope* of x, i.e., the part of the program where its definition is visible, is just the sub-expression e. Outside the overall `let` expression (and also in d) the definition of x is not visible.
- It is possible for the same variable name to be bound more than once. If one of these bindings is nested within the e part of another, the inner binding takes precedence. (We say that the inner binding *shadows* the outer one.)
- By arbitrary convention, reference to an undefined variable evaluates to 0.
- Here are a number of small examples of `let` expressions, each of which evaluates to 42:

```
let x = 21 in
    x + x

let x = 20 in
    let y = x + 2 in
        x + y

(let x = 1 in
     x + x) +
(let y = 20 in
     y + y)

let x = 20 in
    let x = 21 in
        x + x

let x = 20 in
    let x = x + 1 in
        x + x

let x = 40 in
    (let x = 1 in
         x + x) +
    x
```

- This is just about the simplest way to introduce a variable declaration and initialization mechanism into a language. (`let` expressions occur in real functional languages such as Scheme, ML and Haskell.) Imperative languages, in which variables refer to mutable locations that themselves contain values, can be modeled in similar, but more complicated, ways.

- To implement the `eval()` function for this language, we give it an *environment* parameter `env`. The environment is a map from variable names to their values. `VarExp` nodes use the environment to look up their value; `LetExp` nodes compute an extended environment with a new binding, which they pass down to their e subexpression. All other nodes just pass the environment unchanged to their subexpressions.

- Not surprisingly, the cleanest treatment of environments is again based on immutable maps. The class `ImmMap` defines a type of immutable maps, built on top of the Java library's `HashMap`. (Again, this isn't a very efficient implementation of immutable maps, but it is simple and good enough for illustrative purposes.)

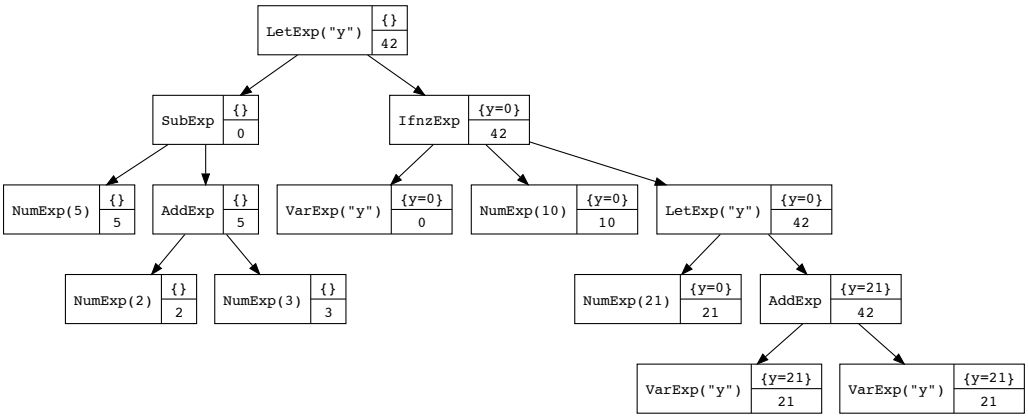
10/ImmMap.java

```
11      // Map construction
12
13      // Call this with something like ImmMap.<String,Integer>empty()
14      static <K,V> ImmMap<K,V> empty() {
15          return new ImmMap<K,V>(new HashMap<K,V>());
16      }
17
18      static <K,V> ImmMap<K,V> extend(K k, V v, ImmMap<K,V> a) {
19        Map<K,V> m = new HashMap<K,V>(a.m);
20        m.put(k,v);
21        return new ImmMap<K,V>(m);
22      }
23
24      // Map inspection
25
26      V get(K k) {
27        return m.get(k);
28      }
```

- From an attribute grammar perspective, `env` is an *inherited* attribute. We specify this attribute, and give equations for the val attribute on the new expression forms, as follows:

$exp \rightarrow$ `let` x `=` exp `in` exp	$exp_1.env := exp.env;\ exp_2.env = extend(exp.env, x, exp_1.val)$
	$exp.val := exp_2.val$
$exp \rightarrow x$	$exp.val := lookup(exp.env, x)$
$exp \rightarrow exp + exp$	$exp_1.env := exp.env;\ exp_2.env := exp.env$
$exp \rightarrow exp - exp$	$exp_1.env := exp_2.env := exp.env$ (a shorthand)

- It can be enlightening to annotate some example expression trees with the environments and values computed for each node. Here is the test tree from the example file with such annotations. (It can be even more useful to trace the order in which the computations are made and the annotations filled in...)



Imperative Environments (10a)

Environment management is an key part of many operations over ASTs, including static analysis, interpretation, code generation, and optimization. Moreover, real compilers make use of environments not just to keep track of variables, but also to manage many other kinds of identifiers, including function names, type names, module names, record field names, etc. Thus environments can get quite large, so handling them efficiently is important. Although it is possible to build quite efficient versions of immutable maps, mutable maps accessed using imperative operations can be more efficient still. Historically, most compilers have used mutable maps and imperative operations to manage environments, which are often called *symbol tables*. Unfortunately, in many languages name binding has characteristics that make imperative implementation complicated. In particular, the limited *scope* of bindings means that we cannot just *add* bindings into a single global environment: we must also *remove* them at the appropriate point in the tree traversal. Moreover, the possibility of shadowing means that removal must be done with care.

This example shows a possible design for an imperative environment implementation that supports retraction of bindings when exiting from a scope.

10a/Example.java

```
3   // A mutable environment datatype that supports retraction of bindings.
4   // (For simplicity, this implementation is specialized to String keys
5   //  and Integer values.)
6   class Env {
7     private Map<String,Stack<Integer>> m;
8
9     // Empty Environment
10    Env() {
11      m = new HashMap<String,Stack<Integer>>();
12    }
13
14    // Extend
15    void extend(String k, int v) {
16      Stack<Integer> s = m.get(k);
17      if (s == null) {
18        s = new Stack<Integer>();
19        m.put(k,s);
20      }
21      s.push(v);
```

```
22      }
23
24      // Retract
25      void retract(String k) {
26        Stack<Integer> s = m.get(k);  // should never be null
27        s.pop();
28        if (s.empty())
29          m.remove(k);
30      }
31
32      // Lookup
33      Integer lookup(String k) {
34        Stack<Integer> s = m.get(k);
35        if (s == null)
36          return null;
37        return s.peek();
38      }
39    }
40
41    abstract class Exp {
42      static Env env = new Env();
43      abstract int eval();
44    }
45
46    class LetExp extends Exp {
47      private String x;
48      private Exp d;
49      private Exp e;
50      LetExp(String x, Exp d, Exp e) {this.x = x; this.d = d; this.e = e;}
51
52      int eval() {
53        int v = d.eval();
54        env.extend(x,v);
55        v = e.eval();
56        env.retract(x);
57        return v;
58      }
59    }
60
61    class VarExp extends Exp {
62      private String x;
63      VarExp(String x) {this.x = x;}
64
65      int eval() {
66        Integer v = env.lookup(x);
67        if (v != null)
68          return v;
69        else
70          return 0;  // default value for undefined variables
71      }
72    }
```

- Instead of passing an (immutable) environment as a parameter to the `eval()` function, we use a single global environment declared as a `static` variable of class `Exp`.
- The evaluator for `LetExp`s extends the environment with its binding before evaluating the e subtree,

and then explicitly retracts the bound name from the environment again afterwards.

- We cannot just implement `retract` by removing the name from the environment; there may be an outer binding of the same name that was temporarily shadowed but should now be visible again. To handle the possibility of multiple bindings of the same name, the entries in the environment map are now *stacks* of values. If a name gets rebound in an inner scope, it is pushed onto the stack; the top of the stack is always used as the correct "current" binding; retraction just pops the stack, leaving any outer bindings untouched. We maintain the invariant that the map never contains an empty stack. We assume that `retract` is only called on currently bound identifiers.