# An Introduction to DemeterF (`Java`)

Bryan Chadwick

September 8, 2009

**Abstract**

This document is meant to be an introduction to Object Oriented Programming (OOP) and Functional Adaptive Programming (AP-F) using DemeterF. DemeterF is a traversal abstraction library (similar to the visitor pattern) that allows programmers to create mutation free traversals that support simple, automatic parallelization of traversal computation. We currently have implementations in Java, C#, and Scheme.

Here we cover the Java version of DemeterF, providing a detailed description of DemeterF traversals through a discussion of hand-written examples, DemeterF function objects, and demonstrations/explanations of useful functions and classes from the library.

## 1 OOP: Data Structures and Functions

In Object Oriented (OO) Programming Languages programmers generally encode functionality into cooperating classes and objects. Implementing a function over a collection of data types introduces code to each of the classes involved. As an example, consider the following Java classes representing `shapes` in Fig. 1.

```
// shape.java
abstract class shape{}

class circle extends shape{
   int radius;
   circle(int r){ radius = r; }
}
class square extends shape{
   int side;
   square(int s){ side = s; }
}
class pair extends shape{
   shape left,
         right;
   pair(shape l, shape r){ left = l; right = r; }
}
```

Figure 1: Classes representing shapes

### 1.1 Functions

There are many different things we might want to do with the different kinds of shapes. To start, we may want to calculate the *area* of a given shape. To compute a shape's area, we introduce an abstract method within the `shape` class, and implementations for each of the subclasses, shown in Fig. 2.

```
// In shape
abstract double area();
// In circle
double area(){ return Math.PI*radius*radius; }
// In square
double area(){ return side*side; }
// In pair
double area(){ return left.area()+right.area(); }
```

Figure 2: Implementing area calculation for shapes

If we add a new variant (or subclass) of `shape`, we just have to provide an implementation of all the abstract methods. A new class `rect` is shown in Fig. 3.

```
class rect extends shape{
   int width,
       height;
   rect(int w, int h){ width = w; height = h; }
   double area(){ return width*height; }
}
```

Figure 3: New shape subclass for rectangles

Adding a new (specialized) operation is more cumbersome; like above, we must add new methods to each of the related classes, including others that we might not be involved in writing. But, writing a more general function is simpler, *e.g.*, a function to *add* another shape to a given shape requires only a single method definition like that in Fig. 4.

```
// In shape
shape add(shape s){ return new pair(s,this); }
```

Figure 4: A method for combining shapes

## 1.2 Traversals

We call a single functionality over multiple (possibly recursive) classes a **traversal**, as the function implementation must *traverse* (or walk) the structures to compute a value. For example, the `area` function computes the total area of a shape by traversing nested composite shapes. In class based OOP languages it becomes tedious to implement such functions over reasonably complex data structures. So it's easier, more flexible, and modular, to write functionality around a more general traversal function.

The *visitor pattern* and other related tools such as DemeterJ were created to improve this situation by separating the implementation of functionality from the structure (or at least traversal) of data types. These visitor related solutions rely on mutation (also referred to as assignment or side effects) as the means to calculate values. In DemeterF we define a *functional* traversal that allows programmers to compute without mutation. Instead, the traversal passes related objects as arguments to methods of a special[1] kind of class, called a *function class*.

Fig. 5 contains a function class that implements the same functionality as the `area()` method written earlier, but it is completely specified within a single class.

```
import edu.neu.ccs.demeterf.*;

// Area function class
class Area extends ID{
   double combine(circle c, int r){ return Math.PI*r*r; }
   double combine(square s, int d){ return d*d; }
   double combine(rect r, int w, int h){ return w*h; }
   double combine(pair p, double l, double r){ return l+r; }

   static double area(shape s){
     return new Traversal(new Area()).<Double>traverse(s);
   }
}
```

Figure 5: Shape `area` calculation with DemeterF

The rest of this document discusses the various DemeterF interfaces and classes, and how they can be used to write functional traversal computations over data structures.

## 2 DemeterF Basics

Traversals drive computation in DemeterF; we use a dynamic traversal that calls methods within special (function) objects, supporting mutation-free computation. The class `Traversal` contains a *generic* method, `traverse(...)`, that implements a general depth-first traversal over a given object. The return type of `traverse` is parametrized to eliminate casting (hence the `<Double>` in Fig. 5).

Each instance of `Traversal` is constructed with a function object that computes values along the traversal. We divide the computation into two parts, represented by different named methods: `combine` and `update`. To aid programmers, we include two classes (`ID` and `Bc`) that provide a bit of predefined behavior. `ID`/`Bc` can then be extended to produce a specific solution to a traversal related problem.

### 2.1 `combine()` Methods

In our `shape` example, we can compare the two implementations of `area`: one internal (within the various `shape` classes), and one external. Fig. 2 shows methods that must be placed in each class. The method within `pair` is an interesting case, as we make (possibly recursive) calls to the abstract `shape.area()` method. Similarly, in Fig. 5, the `area` class describes how to calculate each `shape`'s area, by *combining*

[1]Function classes are special only in their use and the naming of special methods. They can, of course, contain fields/methods like any other Java class.

its fields during traversal; the recursive calls are done automatically. With the method signature of each `combine`, we tell DemeterF how to recursively replace the *constructor* of each concrete class.

For each definition of a class expected to be reached during traversal, say a class named `C` with $n$ fields, for instance, with a definition like this:

```
class C{
  F1 f1;
    /* ... */
  Fn fn;

  C(F1 f1, ..., Fn fn){ /* ... */ }
}
```

a function class would implement a method with the signature:

```
R combine(C c, R1 r1, ..., Rn rn){ ... }
```

Where `R` is what will be returned (`double` in the `area` case), and `R1` through `Rn` are the expected results of the recursive traversal of field types `F1` through `Fn`. The class that we extend, `ID`, contains methods for primitive fields (`int`, `double`, etc.) that just return them as-is (they are not traversed), as seen in Fig. 5 with the non-shape fields passed directly to the combine methods (`radius`, `size`, etc.).

### 2.1.1 Details : `ID`

To implement the `area` class and traversal, we `extend` the DemeterF function class `ID`. The `ID` class knows how to handle Java's *primitive* types (more on that later). We add `combine` methods to compute the `area` of a `shape`; these methods will be executed after all of an object's fields have been traversed.

For another example that uses a few more features of DemeterF, lets try counting the number of simple (non-compound) shapes, in a given `shape`. The function class that implements this is shown in Fig. 6. At a `pair` we add the two recursive calculations together, from the `left` and `right` fields, without counting the current one (`pair` is just a container). At all other shapes, we simply return `1`.

```
// Count function class
class Count extends ID{
   int combine(shape s){ return 1; }
   int combine(pair p, int l, int r){ return l+r; }

   static int count(shape s){
     return new Traversal(new Count()).<Integer>traverse(s);
   }
}
```

Figure 6: Simple shape `count` calculation

There are two different DemeterF features used in this example. First, the fields of non-`pair` shapes are ignored. Since the calculation has no use for the fields of simple shapes, the method does not need to mention them in its signature. If we want to refer to any fields or recursive results in a calculation we must also mention all previous fields: order matters. Second, we've abstracted three possible methods (one each for `circle`, `square`, and `rect`) into a single method for all `shape`s. Because `shape` is the *super-class* of all these classes, this method will be called when it is most suitable. The signature of the method for `pair` is more specific than the general `shape` method, so it is called whenever a `pair` is reached.

### 2.1.2 Predefined Behavior: `Bc`

What good is the simple `ID` class to solve bigger problems? Well, it enables, but doesn't really help you solve more complex problems. So, the DemeterF library provides a subclass of `ID` that *rebuilds* the object being traversed[2]. This behavior is implemented in `Bc` (the *building combiner*). Typically, we `extend Bc` to "functionally" transform a part of a data structure, leaving the rest intact. For example, consider the problem of changing all `circle`s into `square`s of the same size. Fig. 7 shows a class, `Circ2sqr`, that changes all circles in a given shape into squares of the same size.

[2]Some people might refer to this as "object copying".

```
// circle -> square  function class
class Circ2sqr extends Bc{
   shape combine(circle c, int r){ return new square(r); }

   static shape circ2sqr(shape s){
     return new Traversal(new Circ2sqr()).<shape>traverse(s);
   }
}
```

Figure 7: Circle to square transformation

The power here comes from the default behavior of `Bc`, rebuilding all other kinds of shapes. Our class overrides the `combine` method for `circle` (all user methods are more specific than the generic `Bc` behavior), returning an equal sized `square`. In this way, `Bc` can be extended to implement functional updates without using field accesses, and plain `ID` can be used when we don't need the rebuilding behavior.

### 2.1.3 Built-In Types

User defined classes have some number (maybe zero) fields as data members, but what happens at classes that **are** data? We refer to these types as *primitive* or *built-in* classes. For our purposes there is not much difference between *value* types like `int` or `double` and corresponding *reference* types, `Integer`and `Double`. Essentially the set of built-in classes describes the *leafs* of our data structures. In Java we have a few different types/classes that DemeterF considers primitives[3]. These are shown in Fig. 8.

| | | | |
|---|---|---|---|
| `short` | `Short` | `int` | `Integer` |
| `long` | `Long` | `float` | `Float` |
| `double` | `Double` | `char` | `Character` |
| `boolean` | `Boolean` | `String` | |

Figure 8: Java's primitives: boxed and unboxed

Programmers can write `combine` methods to override the default that `ID` provides, and for now, these methods take a single argument: the primitive to be transformed. Otherwise the primitive will be returned without modification. Let's use an example to illustrate; suppose we want to scale all the nested shapes within a given `shape`. In our specific case, because the size of each `shape` is stored as an integer (or `int`) we can simply scale all integers. Fig. 9 shows a function class that implements this in a single `combine` method, extending `Bc` to rebuild the containing shapes.

```
// Scale function class
class Scale extends Bc{
   int combine(int i){ return i*2; }

   static shape scale(shape s){
     return new Traversal(new Scale()).<shape>traverse(s);
   }
}
```

Figure 9: Shape scaling by transforming primitives

We are free to mix `combine` methods for built-in and user-defined types within the same function class, or extend our own function classes (instead of just `ID` and `Bc`). For example, we could have made `scale` extend `Circ2sqr`, resulting in a function class that will `scale` all shapes while converting `circle`s into `square`s in a single traversal.

## 2.2 `update()` Methods

The second part of traversal computation in DemeterF deals with an optional *traversal context*: an object that is passed around and modified (functionally, of course) during the traversal of a data structure. A

[3] Oddly, `Integer.class` (boxed) and `int.class` (unboxed) are different classes in Java, but DemeterF handles this issue in the method dispatch, so the boxed/unboxed types can be used interchangeably in method signatures.

traversal with a context is started by calling a different version of the `Traversal.traverse()` method that takes two parameters: the object to be traversed, and a root traversal context. DemeterF deals with passing the traversal context around, allowing the programmer to focus on when it needs to be *updated* or changed, and when it should be used.

When calling a `combine` method, the traversal context is added to the end of the parameter list and the most specific method is chosen. Because later parameters of `combine` methods can be left off (or ignored), we only need to mention the traversal context in a method signature when it is to be used. Here's a simple example: suppose we want to scale each `shape` by its *depth* from the top level. What we need to do is keep track of how many `pair`s we have traversed into, and scale non-`pair` shapes accordingly. Fig. 10 shows an implementation of this transformation as a single method within each class.

```
// In shape
abstract shape depthScl(int d);
// In circle
shape depthScl(int d){ return new circle(radius*d); }
// In square
shape depthScl(int d){ return new square(side*d); }
// In rect
shape depthScl(int d){ return new rect(width*d,height*d); }
// In pair
shape depthScl(int d){
  int nd = d+1;
  return new pair(left.depthScl(nd), right.depthScl(nd);
}
```

Figure 10: Implementing `depthScl` within shapes

The `depthScl` methods are generally the same for the non-`pair` shapes. The depth argument (`d`) is used to create a scaled version of each `shape`. In the `pair` case, we construct a new `pair` using the recursive results of the `left` and `right`. Before recurring, the depth is incremented and passed to recursive calls.

Fig. 11 contains the same transformation written using DemeterF. There are three major differences from earlier function classes. First, an integer argument is passed to the `traverse()` method. This becomes the initial traversal context; if no update methods were matched, the initial context (`1`) would be given as the last parameter to all combine methods. The second addition is that the `combine(int i, int d)` method has two parameters. The second parameter is the traversal context, which is of type `int`.

```
import edu.neu.ccs.demeterf.control.Fields;

// DepthScl function class
class DepthScl extends Bc{
   // Increase the depth for children of a pair
   int update(pair p, Fields.any f, int d){ return d+1; }

   // Scale integers by their depth
   int combine(int i, int d){ return i*d; }

   // Traverse, passing 1 as the starting depth
   static shape scale(shape s){
     return new Traversal(new DepthScl()).<shape>traverse(s,1);
   }
}
```

Figure 11: Implementing `depthScl` with a function class

The final addition is the most important: we use an `update` method and the DemeterF class `Fields.any`. The update method is similar to the `pair.depthScl(...)` method from Fig. 10 where the argument is incremented. The use of the type `Fields.any` signifies that the result of the `update` method should be used as a traversal context for *all* fields of a `pair`.

### 2.2.1 Details : Field classes

Notice where we used `Fields.any`? As the second parameter to an `update` method. This allows us to update the traversal context for *all* of the children of a given object, but what if we want different contexts to be passed for the `left` and `right` children of a `pair`? We have to add a few definitions to our class so DemeterF knows how to tell us when we are traversing into a specific field.

We signify the traversal of a field by passing an instance of a *field class* (if it exists) to the `update` method. A field class is a `public`, `static`, inner-class with the same name as the field it represents. Fig. 12 shows the additions that would be made to `pair` in order to support separate traversal contexts for its `left` and `right` fields.

```
import edu.neu.ccs.demeterf.control.Fields;

// In pair
public static class left extends Fields.any{}
public static class right extends Fields.any{}
```

Figure 12: Field classes for the `pair` class.

We extend `Fields.any` in order to allow our earlier function class, `DepthScl`, to continue working. Once we've added these field classes to `pair`, we can use them in place of `Fields.any`. Fig. 13 shows a function class that implements the scaling of only the shapes that are to the left of some `pair`. We do this by writing a more specific `update` method using `pair.right` that returns the original depth. The rest of the functionality is inherited from `DepthScl`.

```
// LeftScl function class
class LeftScl extends DepthScl{
   // Don't increase the depth for 'right' children
   int update(pair p, pair.right f, int d){ return d; }

   static shape scale(shape s){
     return new Traversal(new LeftScl()).<shape>traverse(s,1);
   }
}
```

Figure 13: Scaling just the `left` sides of pairs by their depth.

As a final example of `update` methods, we'll create a function class to generate *Scaled Vector Graphics* (SVG) images. SVG is an XML language for describing pictures made of shapes and text with support for complex compositing. With it we can draw images of our shapes and other things (as we'll see later). I've created a simple class, `svg`, that encapsulates a few of the details of the SVG format in `static` methods[4].

Fig. 14 shows a DemeterF function class that constructs a representative string in SVG format from a given `shape`. The traversal context (starting at 40) represents the `x` coordinate of the current shape. The `update` method increases this so shapes will be drawn across the image, assuming that nested `pair`s form a list to the right.

```
// Display function class
class Display extends ID{
   static int y = 50;
   int update(pair p, pair.right f, int x){ return x+45; }
   String combine(pair p, String l, String r){ return l+r; }

   int combine(int i){ return 3*i; }
   String combine(circle c, int r, int x){ return svg.circle(x,y,r); }
   String combine(square s, int sd, int x){ return svg.rect(x,y,sd,sd); }
   String combine(rect r, int w, int h, int x){ return svg.rect(x,y,w,h); }

   static String display(shape s){
      return svg.image(240, 100, new Traversal(new Display())
                                      .<String>traverse(s,40));
   }
}
```

Figure 14: Class to display a `shape` (generates an SVG String).

[4]See the corresponding source code file (`shape.java`) for details.

Using the `Display` class we can demonstrate each of the transformations that we created earlier. Fig. 15 shows the resulting images of calling the various transformations (`Circ2sqr`, `Scale`, `DepthScl`, and `LeftScl`) on the shape created by the expression:

```
new circle(5)
    .add(new square(5))
    .add(new circle(2))
    .add(new rect(4,9));
```

See `shape.java` (available wherever you got this) for more information on how these images were created and written to files.

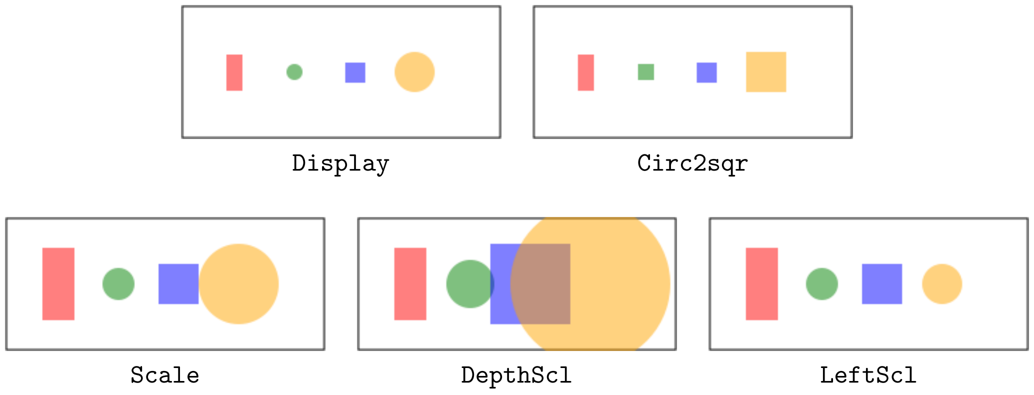

Figure 15: Images created using `Display` on a `shape` after transforming

You can see the starting `shapes` in the first image ("`Display`"). The `Circ2sqr` transformation is shown in the second image, and uniform `Scale` in the third. Because the shapes are added left to right, the `DepthScl` transformation scales them based on their distance from the left margin. `LeftScl` only scales shapes that are off the `left` of a `pair`; note that the right most circle remains unchanged.

That concludes the functional portions of DemeterF traversals. In the next section we introduce the final traversal parametrization, control, and show how it can be created and used effectively.

## 3 Traversal Control

The traversals we used previously have gone *everywhere*; in DemeterF this is the default traversal *control* (or *strategy*). It is possible to override the default in order to get all kinds of control/function combinations. In this section we introduce the classes needed to control traversals and show examples of how these can affect the *types* that are seen by `combine` methods.

### 3.1 Control Classes

In DemeterF we view class/interface declarations as the definition of a graph, usually called a *class-graph*. The edges of this graph represent relationships between classes, *is-a* and *has-a*, similar to those found in a UML class diagram. We focus on *has-a* relationships, since true object instances are only made up of concrete classes. Given a class definition, say:

```
class C{
    int i;
}
```

We say that there is an *edge* from `C` to `int` with the *label* `i`. To represent such edges, DemeterF has a class in the `edu.neu.ccs.demeterf.control` package named `Edge` that can be used to control where a traversal goes. `Edge` has a constructor that is used to create edges:

```
public Edge(Class c, String label);
```

For example, to represent the edge `C.i` we would construct an `Edge` with the expression:

```
new Edge(C.class, "i")
```

Each Traversal is parametrized by an instance of Control[5] (also in the control package) that tells the traversal which edges should be traversed from a given object instance. Control has a few static methods that are used to create instances to be passed to traversals.

## 3.2 Example Structures

Before we get started, let's introduce a simple data structure to be used as an example: *ternary trees.* We chose ternary trees[6] because they have more interesting paths and are not quite as trivial. Fig. 16 shows the simple class definitions for tree, none, one, and three, including the insert() method for each[7]. The insert method maintains an invariant for three trees similar to that of binary trees: all numbers in the left subtree are lessthan-or-equal to ldata; those in the mid subtree are greater-than ldata and lessthan-or-equal to rdata; and those in the right subtree are greater-than rdata.

```
// Ternary Trees
abstract class tree{
   abstract tree insert(int d);
}
// Empty tree
class none extends tree{
   tree insert(int d){ return new one(d); }
}
// Single integer data
class one extends tree{
   static tree e = new none();
   int data;
   one(int d){ data = d; }
   tree insert(int d){
     if(d <= data)
       return new three(e,d,e,data,e);
     return new three(e,data,e,d,e);
   }
}
// Three branches, two data's
class three extends tree{
   tree left;
   int ldata; tree mid;
   int rdata; tree right;

   three(tree l, int ld, tree m, int rd, tree r){
     left = l; ldata = ld;
     mid = m;
     rdata = rd; right = r;
   }
   tree insert(int d){
     if(d <= ldata)
       return new three(left.insert(d),ldata,mid,rdata,right);
     if(d <= rdata)
       return new three(left,ldata,mid.insert(d),rdata,right);
     return new three(left,ldata,mid,rdata,right.insert(d));
   }
}
```

Figure 16: Ternary tree structures with insertion.

To get going, we can implement a simple ToString function class for computing a string representation of a tree. Fig. 17 shows our implementation of toString for ternary trees. As long as that makes sense, we can now explain the various methods/kinds of Control.

## 3.3 Static Methods of Control

Each of the useful traversal control objects has a static function that returns a Control that implements the corresponding control behavior. We explain each of them in turn and provide lots of examples.

[5]or a subclass of, of course.
[6]As opposed to *binary* trees.
[7]All the code for the rest of the document can be found in threetree.java.

```
// In tree
public String toString()
{ return new Traversal(new ToString()).traverse(this); }

// ToString function class
class ToString extends ID{
   String combine(none n){ return "."; }
   String combine(one o, int i){ return "("+i+")"; }
   String combine(three t, String l, int ld, String m, int rd, String r){
     return "["+l+","+ld+","+m+","+rd+","+r+"]";
   }
}
```

Figure 17: `toString` for ternary trees.

### 3.3.1 `everywhere()`

The traversals we've seen so far have not specified any traversal control. The `Traversal` constructor is overloaded to supply the *everywhere* control when none is given. If you need to specify a `Control` explicitly for some reason and want this functionality simply use `everywhere()`.

### 3.3.2 `bypass(Edge ... edges)`

First is a method `bypass(Edge[])` that creates a mutable `Control` bypassing the given edges. The method is actually defined using Java's variable arguments syntax – `(Edge ... edges)` – so there's no need to wrap the edges in an array. We can use this bypassing functionality to implement a simple minimum operation on `tree`s. Fig. 18 shows a `Min` function class that returns the minimum number in a given `tree`, or `none` if the `tree` doesn't contain any numbers.

```
// Import Edge
import edu.neu.ccs.demeterf.control.*;

// Min function class
class Min extends ID{
   none combine(none n){ return n; }
   int combine(tree t, int i){ return i; }
   int combine(three t, tree l, int ld){ return ld; }

   static int min(tree t){
     Control c = Control.bypass(new Edge(three.class,"mid"),
                                new Edge(three.class,"right"));
     return new Traversal(new Min(),c).<Integer>traverse(t);
   }
}
```

Figure 18: A `Min` operation for ternary trees.

The `Control.bypass(...)` method is used to bypass the `mid` and `right` edges (*i.e.*, fields) of `three` trees[8]. This means that the recursive `Traversal` proceeds only down the `left` subtrees. The first method handles the empty tree case (`none`), the second method matches the both the single element scenario (`one`) and when the left tree has a minimum value (`left` is not `none`). The two element case when the `left` is `none` is handled in the third method, returning the `ldata` field.

One interesting note (to repeat) is that we have abstracted two methods:

```
   int combine(one  t, int i){ return i; }
   int combine(tree t, int i){ return i; }
```

into the single `tree` method... think about that for a minute. We can do this whenever it is convenient, or it makes code easier to follow.

[8]We are working on supporting a more concise `DemeterJ` style *strategy* expressions.

### 3.3.3 `bypass(String edges)`

To implement `Max` we've chosen to use a different (often, easier to use) version of the `Control.bypass()` method, shown in Fig. 19. Instead of constructing the `Edge`s by hand, we pass a `String` representation of the edges to be bypassed. The string contains *space delimited* edge descriptions of the form: "`package.Class.field`". In this case our classes are in the base (default) packageso the prefix can be omitted.

```
// Max function class
class Max extends ID{
   none combine(none n){ return n; }
   int combine(one t, int i){ return i; }
   int combine(three t, tree l, int ld){ return ld; }
   int combine(three t, tree l, int ld, tree m, int rd){ return rd; }
   int combine(three t, tree l, int ld, tree m, int rd, int r){ return r; }

   static int max(tree t){
     Control c = Control.bypass("three.left three.mid");
     return new Traversal(new Max(),c).<Integer>traverse(t);
   }
}
```

Figure 19: A `Max` operation for ternary trees.

### 3.3.4 `only(Edge ... edges )` / `only(String edges)`

The analog to the `Control.bypass(...)` methods (bypassing the given edges) are the methods `Control.only(...)` that create a `Control` that bypasses all edges *except* the given ones (*i.e.*, permits traversing only the given edges). Fig. 20 shows an alternative method using the `Min` function class from earlier. We use the `Control.only(String)` method to *only* traverse the `three.left` field. In this case the specification using `only(...)` is actually more concise than that using `bypass(...)`, but either one can be uses depending on the situation.

```
// In Min
static int minalt(tree t){
  Control c = Control.only("three.left");
  return new Traversal(new Min(),c).<Integer>traverse(t);
}
```

Figure 20: Alternative `min` function using `Control.only(...)`.

The other method, `Control.only(Edge[])`, has a similar correspondence to the `bypass(Edge[])` method, returning a `Control` object that permits only the traversal of the given edges.

### 3.3.5 `nowhere()`

Sometimes the general traversal order or depth is not quite right for the job-at-hand, or you might want to use just the matching functionality without a full traversal. For these situations there is a `Control` that tells a `Traversal` to bypass *all* edges. This type of traversal control is created using the `Control.nowhere()` method. Fig. 21 shows one use of the `nowhere` traversal control: implementing `isLeaf()` for ternary trees.

To make these kinds of traversals easier to create, `Traversal` contains a static method `onestep(...)` that takes a function object (e.g. `ID` or `Bc`) and returns a traversal with the `nowhere Control`. Fig. 22 shows an alternative implementation, `isLeaf2()`, that uses `onestep` to create the traversal.

When `nowhere` is used, each object is unfolded and its fields are passed to the matching combine method. The `combine` methods in the `Leaf` function class ignore any fields, matching solely on the type of the original object. We can also use the `onestep` (or `nowhere`) traversal to implement recursion by hand. Fig. 23 shows a more classical implementation of `tree` minimum with hand written recursion.

Here the one-step traversal is cached for efficiency, so we can call it repeatedly when needed. The simple cases (`none` and `one`) are the same as before, so we extend the earlier `Min` (just for brevity). The

```
class Leaf extends ID{
   boolean combine(none n){ return true; }
   boolean combine(tree t){ return false; }

   static boolean isLeaf(tree t){
     return new Traversal(new Leaf(), Control.nowhere())
                .<Boolean>traverse(t);
   }
}
```

Figure 21: `isLeaf` for ternary trees.

```
static boolean isLeaf2(tree t){
  return new Traversal.onestep(new Leaf()).<Boolean>traverse(t);
}
```

Figure 22: Alternative `isLeaf` implementation.

```
class Min2 extends Min{
   int combine(three t, none l, int ld){ return ld; }
   int combine(three t, tree l, int ld){ return min(l); }

   static Traversal trav = Traversal.onestep(new Min2());
   static int min(tree t){ return trav.<Integer>traverse(t); }
}
```

Figure 23: Alternative `Min` implementation.

new methods accept the unfolded `three` tree, and handle each case appropriately: if `left` is a `none`, then `ldata` is the minimum, otherwise recur on the `left` subtree[9].

### 3.3.6 `builtins(Class ... classes)`

The final method that `Control` provides has to do with the *built-in* concept of Section 2.1.3. While traversing, it's sometimes nice to be able to tell the traversal that instances of a specific class should be treated as *leafs* (or terminals) of a data structure, just as `int`, `double`, *etc.*, are. For this purpose, the programmer can add classes to the set of *built-ins* for a single traversal by creating a `Control` with `builtins(...)`. Any instance of the classes given will not be traversed, but the function class will be called with the object as the first parameter. If there is a traversal context, it will be passed as the `combine` method's second argument.

Fig. 24 shows another `isLeaf()` implementation (remember Figs. 21 and 22?). The difference between the `nowhere()` implementations and this one using `builtins(...)` has to do with the fields being accessible as parameters to combine methods.

```
static boolean isLeaf3(tree t){
   return new Traversal(new leaf(), Control.builtins(tree.class))
              .<Boolean>traverse(t);
}
```

Figure 24: Another alternative `isLeaf` implementation.

Using `nowhere()` (or equivalently `Traversal.onstep(...)`) unfolds each instance and calls the function object with it and all fields, followed by the optional traversal context. With `builtins(...)` instances are seen as being atomic, and the function object is passed just the instance and the optional traversal context. A true example requires more complex class structures and interactions, but the essential idea is that these *built-in* classes are treated just like `int` and `double`

[9]Notice that both methods have the same number of parameters? In saving the reimplementation of the `none`/`one` methods we are forced to *override* the original `min.combine(three, tree, int)` method. If we instead extend `ID`, we need the two other methods, but can do without the third parameter (`ld`) in the second `combine` method.

## 4 More Complex Examples

As a couple of more complex examples to ponder, we show how our ternary tree structures can be used to generate LaTeX pictures[10] and a possible implementation of tree *equality*.

### 4.1 Tree Additions

If you remember the tree class definitions (Fig. 16), in order to use separate `update` methods we need to add definitions for field classes to `one` and `three`.

```
// Field classes for 'one'
public static class data extends Fields.any{}

// Field classes for 'three'
public static class left  extends Fields.any{}
public static class ldata extends Fields.any{}
public static class mid   extends Fields.any{}
public static class rdata extends Fields.any{}
public static class right extends Fields.any{}
```

Figure 25: Field classes needed for trees.

As before, these classes are placed inside their respective class definitions. Now we can move on to drawing our trees within a LaTeX `picture` environment.

### 4.2 Drawing Trees

#### 4.2.1 Helper Classes

In order to draw a ternary tree, we use a helper class to keep track of the `x` and `y` coordinates and bounds. This is done with a *triple* of `double`s; the structure and a few simple (but useful) methods is shown in Fig. 26. The methods `left`, `mid`, and `right` calculate a new `trip` for each of the corresponding branches of a `three`. The little methods `w()`, `x()`, and `y()` return the *width*, *x-coord*, and *y-coord* of a tree with the given `trip`.

```
// Triple of left/right X bounds, and height (Y)
class trip{
   double lx,rx,h;
   trip(double lxx, double rxx, double hh)
   { lx = lxx; rx = rxx; h = hh; }

   trip left(double dh){ return new trip(lx, lx+w()-20, h+dh); }
   trip right(double dh){ return new trip(rx-w()+20, rx, h+dh); }
   trip mid(double dh){ return new trip(lx+w()+20, rx-w()-20, h+dh); }

   double w(){ return (rx-lx)/3; }
   double x(){ return (lx+rx)/2; }
   double y(){ return h; }
}
```

Figure 26: Helper class for picture coordinates.

As before with SVG, we encapsulate the TeX implementation specifics in a class, `tex`, that contains methods for a header/footer and creating shapes (circles, boxes, and text)[11]. Fig. 27 contains the function class that implements LaTeX `Display` for a given `tree`.

Let's walk through the `Display` function class from top to bottom. The static variable, `dH`, is simply the change in height between a tree and its subtrees. The three `update` methods update the traversal context, a `trip`, for `left`, `mid`, and `right` subtrees. The corresponding `trip` methods shift the bounds of the subtrees to the correct portion of the picture.

The left picture of Fig. 28 demonstrates the first case; for a `none` tree we just place an empty circle at the current `x`, with `y` adjusted for the circle radius. The center of Fig. 28 shows the results of the `one`

[10]This document is written in LaTeX, so the `picture` environment is a natural visual aid.
[11]This code is also in `threetree.java`.

```
// TeX based display for ternary trees
class Display extends ID{
   static double dH = -30;

   // Coordinates change for subtraversals
   trip update(three t, three.left f, trip c){ return c.left(dH); }
   trip update(three t, three.mid f, trip c){ return c.mid(dH); }
   trip update(three t, three.right f, trip c){ return c.right(dH); }

   // Place nodes in the picture
   String combine(none n, trip c){ return tex.circle(c.x(),c.y()+4,4); }
   String combine(one t, int i, trip c){ return tex.box(c.x(),c.y(),12,12,""+i); }
   String combine(three t, String l, int ld, String m, int rd, String r, trip c){
     return l+m+r+lines(c)+
            tex.box(c.x(),c.y(),28,12,ld+tex.space()+rd);
   }
   // Draw the lines/branches
   static String lines(trip c){
     double h = c.y()-6, nh = h+dH+12, cx = c.x();
     return (tex.line(cx,h,c.mid(0).x(),nh)+
             tex.line(cx-12,h,c.left(0).x(),nh)+
             tex.line(cx+12,h,c.right(0).x(),nh));
   }
   static String display(tree t, int w, int h){
     return (tex.head(w,h)+
             new Traversal(new Display())
                 .<String>traverse(t,new trip(12,w-12,h-20))
             +tex.foot());
   }
}
```

Figure 27: LaTeX picture generation function class.

combine method; we place a box with the given number into the picture. The final tree in Fig. 28 is a simple instance of three. The combine method appends the recursive results (l, m, and r) and adds lines, then places a larger box containing both numbers.

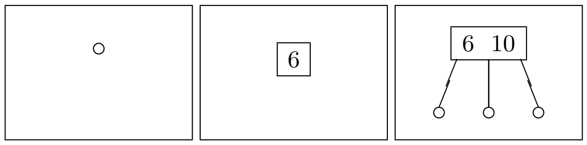

Figure 28: Three simple trees: none(), one(6) and one(6).insert(10).

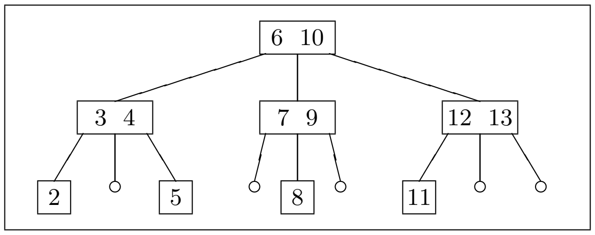

Figure 29: A more complex tree drawing.

The final picture, Fig. 29, is the result of calling display on a tree built from the expression below.

```
new none().insert(6).insert(10).insert(12).insert(4).insert(9).insert(3)
    .insert(7).insert(2).insert(5).insert(8).insert(13).insert(11);
```

The tex class and all the code for generating these pictures and writing them to a file can be found in the corresponding source file (threetree.java).

## 4.3 Tree Equality

Object (in this case tree) equality is a reasonably challenging problem for any programming framework or library to solve. This is mostly because the `equals` function involves the traversal of two objects simultaneously. In OO Languages this is usually done be type tests (instanceof), casting and hand coded “traversal”. Notice there hasn’t been a cast expression in this entire document? Why start now... we still don’t need it. Fig. 30 shows a function class that implements equality for our ternary trees.

```
// Tree equality function
class Equal extends ID{
   int update(one o, one.data f, one t){ return t.data; }

   int  update(three th, three.ldata f, three t){ return t.ldata; }
   int  update(three th, three.rdata f, three t){ return t.rdata; }
   tree update(three th, three.left  f, three t){ return t.left; }
   tree update(three th, three.mid   f, three t){ return t.mid; }
   tree update(three th, three.right f, three t){ return t.right; }

   boolean combine(){ return false; }
   boolean combine(none a, none b){ return true; }
   boolean combine(int a, int b){ return a == b; }
   boolean combine(one a, boolean d, one b){ return d; }
   boolean combine(three a, boolean l, boolean ld, boolean m,
                   boolean rd, boolean r, three b){
     return l && ld && m && rd && r;
   }

   static boolean equal(Object a, Object b){
     return new Traversal(new Equal()).<Boolean>traverse(a,b);
   }
}
```

Figure 30: Tree equality function class with DemeterF.

We use the `update` methods to *traverse* the second object (if it’s a `tree`). The default of equality is, of course, `false`, as in the first `combine` method. Once we get to leafs of the structure (`none` or `int`) we can return a result. For the more complex trees, we get back recursive results from subtraversals and make sure that they all returned `true`. Instead of comparing the type of the two objects, we mention the traversal context in each case, to be sure DemeterF will do that for us.

# 5 The End

That concludes our introduction to DemeterF (in Java); hopefully you’ve enjoyed the tour. There is more documentation to be found on-line at the DemeterF web site, including downloads, examples, and versions for other programming languages. See the DemeterF Website to get started.

# Appendices: System Setup

# A Basic Setup

If you are using some form of Unix or Linux, the setup is usually a breeze. If you haven’t considered a Linux distribution I would recommend trying Ubuntu. It seems to be a little lighter and a bit easier to use than RedHat distributions. Setup on these systems is mostly automatic, since Mono, Java and JavaCC can usually be obtained as installable packages.

On windows I suggest installing Cygwin and Emacs. The basic Cygwin installation has all the needed programs including, Bash and the base Unix programs like `ls`, `cp`, `mv`, etc. This also makes it easy to setup your `PATH` and `CLASSPATH` once and for all.

# B Java Setup

The JDK (Java Development Kit) is available from Sun Microsystems with installations for almost every operating system. For Linux distributions it is usually also available in package form. You should install at least Java 1.5 (J2SE 5.0) to allow support for generics. Once the JDK is installed, you can make sure that the installation works by running: `java -version` and `javac -version`. You should see a print out of your installed version information. If the commands cannot be found you will need to make sure they are your `path`. Also, your distribution might have a few different versions installed, and select a lesser version by default. The range of installations is too great to be covered here, but search for solutions on-line.

If you are running Windows, you should be able to run the `java/javac` commands within a Cygwin terminal. If the commands could not be found, or appear to be the wrong version, you may need to update your `PATH` environment variable. This can be done (usually) by right clicking on *My Computer* and selecting *Properties*. Under the *Advanced* tab, at the bottom of the window there should be a button labeled *Environment Variables*. You should add the directory of the Java installation (usually something like `C:\Program Files\Java\jdk1.5.0\bin`) to the `PATH` variable. Be sure to use the `jdk`, not the `jre` directory. Also remember, the *path separator* on Windows is actually a *semi-colon* (;), while on Unix/Linux it is a *colon* (:).

## C DemeterF Library and `CLASSPATH`

You can download the `demeterf.jar` from the DemeterF Website, then all you need to do is make sure that the location of the JAR is in your `CLASSPATH`. If you are using Eclipse then you can add the JAR to the Project libraries under the *Java Build Path* preferences.

On Windows, follow the directions above to add the location (be sure to include the full name, *e.g.*, `C:\path\demeterf.jar`) of the JAR to your `CLASSPATH` environment variable. If it does not exist then you will have to add a new one. Once it is added, you can test that it works by running:

```
java demeterf --help
```

in a new Cygwin terminal, which should print out a bunch of information on how to use the `DemeterF` class generator, included with DemeterF.

On Unix/Linux, you must add the location of the JAR to your `CLASSPATH` environment variable. This can be done by editing your `.bashrc/.tcshrc` file (depending on your shell). You should also include '`.`' so that the current directory can also be searched for classes.

You can do this by adding something like:

```
export CLASSPATH=.:/home/chadwick/www/demeterf/demeterf.jar
```

to your `.bashrc` file, or:

```
setenv CLASSPATH ".:/home/chadwick/www/demeterf/demeterf.jar"
```

to your `.tcshrc` file. Obviously the location of the JAR will probably be different on your system, unless you are using a Northeastern CCIS machine, where you can actually use the on-line version above to be sure it's up-to-date. Search on-line for more information and/or details.