An Introduction to **DEMETERF** (**C#**)  

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 C# 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 C# classes representing shapes in Fig. 1.

```csharp
// shape.cs
abstract class shape {}
class circle : shape{
    int radius;
    public circle(int r){ radius = r; }
}
class square : shape{
    int side;
    public square(int s){ side = s; }
}
class pair : shape{
    shape left,
    right;
    public 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.

```csharp
// In shape
public abstract double area();
// In circle
public override double area(){ return Math.PI*radius*radius; }
// In square
public override double area(){ return side*side; }
// In pair
public override 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.

```csharp
class rect : shape{
    int width,
    height;
    public rect(int w, int h){ width = w; height = h; }
    public override 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.
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 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.

```csharp
using edu.neu.ccs.demeterf;

// Area function class
class Area : 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; }

    public static double area(shape s)
    {
        return new Traversal(new Area()).traverse<double>(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 C# 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:

```java
class C{
    F1 f1;
    /* ... */
    Fn fn;
    public C(F1 f1, ..., Fn fn){ /* ... */ }
}
```
a function class would implement a method with the signature:

```java
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 C#'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, let’s 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.

```java
// Count function class
class Count : ID{
    int combine(shape s){ return 1; }
    int combine(pair p, int l, int r){ return l+r; }
    public static int count(shape s){
        return new Traversal(new Count()).traverse<int>(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 shapes. 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. 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 circles into squares of the same size. Fig.7 shows a class, Circ2sqr, that
changes all circles in a given shape into squares of the same size.

2Some people might refer to this as “object copying”.

4
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, Int32 and Double. Essentially the set of built-in classes describes the leaves of our data structures. In C# we have a few different types/classes that DemeterF considers primitives. These are shown in Fig. 8.

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.

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 circles into squares 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 Unlike Java, in C# int (value or unboxed) and Int32 (reference or boxed) are (pretty-much) synonyms, 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 `pairs` 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.

```java
// In shape
public abstract shape depthScl(int d);
// In circle
public override shape depthScl(int d){ return new circle(radius*d); }
// In square
public override shape depthScl(int d){ return new square(side*d); }
// In rect
public override shape depthScl(int d){ return new rect(width*d,height*d); }
// In pair
public override 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`.

```java
using Fields = edu.neu.ccs.demeterf.control.Fields;

// DepthScl function class
class DepthScl : 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
    public static shape scale(shape s){
        return new Traversal(new DepthScl()).traverse<shape>(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, 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.

```java
using Fields = edu.neu.ccs.demeterF.control.Fields;
// In pair
public class leftF : Fields.any{}
public class rightF : 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.

```java
// LeftScl function class
class LeftScl : DepthScl{
// Don't increase the depth for 'right' children
int update(pair p, pair.rightF f, int d){
    return d; }
}
public static shape scale(shape s){
    return new Traversal(new LeftScl()).traverse<shape>(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.

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 pairs form a list to the right.

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:

```java
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.

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.

---

4In DemeterF-C# duplicate names within a class are not allowed, so we append an ‘F’ to the field class, e.g., leftF.
5See the corresponding source code file (shape.cs) for details.
// Display function class
class Display : ID{
    static int y = 50;
    int update(pair p, pair rightF 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); }

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

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

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

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:

```java
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:

```java
public Edge(Type t, String label);
```

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

```java
new Edge(typeof(C), "i")
```

Each Traversal is parametrized by an instance of Control (also in the control package) that tells

---

6 or a subclass of, of course.
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 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.

The insert method maintains an invariant for three trees similar to that of binary trees: all numbers in the left subtree are less-than-or-equal to ldata; those in the mid subtree are greater-than ldata and less-than-or-equal to rdata; and those in the right subtree are greater-than rdata.

```csharp
// Ternary Trees
abstract class tree { }
public abstract tree insert(int d); }
// Empty Tree
class none : tree { }
public override tree insert(int d) { return new one(d); }
// Single integer data
class one : tree { internal int data; public one(int d) { data = d; }
public override 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 datas
class three : tree { internal tree left; internal int ldata; internal tree mid;
internal int rdata; internal tree right;
public three(tree l, int ld, tree m, int rd, tree r) { left = l; ldata = ld;
mid = m; rdata = rd; right = r; }
public override 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.

#### 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

---

7 As opposed to binary trees.
8 All the code for the rest of the document can be found in threetree.cs.
// In tree
public override string ToString()
{
    return new Traversal(new ToString()).traverse<string>(this);
}

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

Figure 17: toString for ternary trees.

explicitly for some reason and want this functionality simply use everywhere().

3.3.2 bypass(params Edge[] edges)

First is a method bypass(Edge[]) that creates a mutable Control bypassing the given edges. The method is actually defined using C#'s variable arguments syntax – (params 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 trees. 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
using Edge = edu.neu.ccs.demeterf.control.Edge;

// Min function class
class Min : ID{
    none combine(none n){ return n; }
    int combine(tree t, int i){ return i; }
    int combine(three t, none l, int ld){ return ld; }
    public static int min(tree t){
        Control c = Control.bypass(new Edge(typeof(three),"mid"),
                                   new Edge(typeof(three),"right"));
        return new Traversal(new Min(),c).traverse<int>(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. 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 ld data 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.

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 Edges by hand, we pass a String representation of the edges to be bypassed. The string contains space delimited edge descriptions of the form: "namespace.Class,assembly.field". In this case our classes are in the base (default) namespace (and assembly) so the prefix can be omitted.

9 We are working on supporting a more concise DemeterJ style strategy expressions.
3.3.4 only(params 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 used depending on the situation.

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.
public static bool isLeaf2(tree t){
    return newTraversal.onstep(newLeaf()).traverse<bool>(t);
}

Figure 22: Alternative isLeaf implementation.

class Min2 : Min{
    int combine(three t, none l, int ld){ return ld; }
    int combine(three t, tree l, int ld){ return min(l); }
    staticTraversal trav =Traversal.onstep(newMin2());
    public static int min(tree t){ return trav.traverse<int>(t); }
}

Figure 23: Alternative Min implementation.

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

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 leaves (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.

public static bool isLeaf3(tree t){
    return newTraversal(newLeaf(), Control.builtins(typeof(tree))
        .traverse<bool>(t);
}

Figure 24: Another alternative isLeaf implementation.

Using nowhere() (or equivalentlyTraversal.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.

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\footnote{This document is written in \LaTeX{}, so the picture environment is a natural visual aid.} and a possible implementation of tree equality.

\footnote{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.}
4.1 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.

```java
// Field classes for 'one'
public class dataF : Fields.any {}

// Field classes for 'three'
public class leftF : Fields.any{}
public class ldataF : Fields.any{}
public class midF : Fields.any{}
public class rdataF : Fields.any{}
public class rightF : 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 doubles; 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.

```java
// Triple of left / right X bounds , and height (Y)
class trip {
    double lx, rx, h;
    public trip(double lxx, double rxx, double hh) {
        lx = lxx;
        rx = rxx;
        h = hh;
    }
    public trip left(double dh) {
        return new trip(lx, lx+w() - 20, h+dh);
    }
    public trip right(double dh) {
        return new trip(rx - w() + 20, rx, h+dh);
    }
    public trip mid(double dh) {
        return new trip(lx+w() + 20, rx - w() - 20, h+dh);
    }
    public double w() { return (rx-lx)/3; }
    public double x() { return (lx+rx)/2; }
    public 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). 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 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.

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

This code is also in threetree.cs.

12 This code is also in threetree.cs.
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 by 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.

```csharp
// Tree equality function
class Equal : ID{
    int update(one o, one.dataF f, one t){ return t.data; }
    int update(tree th, three.dataF f, three t){ return t.1data; }
    int update(tree th, three.dataF f, three t){ return t.2data; }
    int update(tree th, three.dataF f, three t){ return t.3data; }
    tree update(tree th, three.dataF f, three t){ return t.1mid; }
    tree update(tree th, three.dataF f, three t){ return t.2mid; }
    tree update(tree th, three.dataF f, three t){ return t.3mid; }

    bool combine(){ return false; }
    bool combine(none a, none b){ return true; }
    bool combine(int a, int b){ return a == b; }
    bool combine(one a, bool d, one b){ return d; }
    bool combine(three a, bool l, bool l1, bool m, bool n, bool rd, bool r, three b){
        return l && l1 && m && n && rd && r;
    }

    public static bool equal(object a, object b){
        return new Traversal(new Equal()).traverse<bool>(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 C#); 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 C# Compiler

In order to compile C# programs you will obviously need a C# compiler. On windows you can download Visual C# Studio Express. This allows you to compile C# source files from within the IDE (Visual Studio), and it also installs the command line C# compiler, csc.exe. Later, you can simply add an assembly reference within Visual Studio, but if you intend to compile things by hand (from the command line) then you will probably need to add the .NET base directory to your PATH.

On my machine the directory is located at C:\WINDOWS\Microsoft.NET\Framework\v3.5. You can add this to your PATH environment variable by right clicking on My Computer, choosing preferences, and selecting Environment Variables under the Advanced tab. If you already have a PATH variable, edit it by
placing the directory path at the end preceded by a semi-colon (;). Otherwise, just add a new variable named \texttt{PATH} with the directory as its value.

Unfortunately, \texttt{csc.exe} doesn’t interact very well with Cygwin, so it’s easier to run it under the Windows terminal. I won’t get into this here, but if you want to use C# on Windows you should figure out how it works anyway.

On Unix/Linux you should install \texttt{Mono}, a C# compiler, \texttt{gmcs}, and \texttt{.NET} runtime. This is what I use on Linux, so it’s much easier to use. You should be able to find an installable package for your distribution, or follow the directions from the website.

\section*{C \ DemeterF Library/DLL}

Finally, you will need the \texttt{DemeterF.dll} library available from the \texttt{DemeterF Website}. Installing it into your \texttt{assembly cache} is beyond the scope of this discussion, but there is more information on the Internet (I’m not sure about Windows), specifically on the \texttt{Mono Website}.

On Linux, if you place the DLL in a common place, you can create symbolic links to it and place them in the current directory when needed. When you compile a source file from the command line your command will look something like this:

\begin{verbatim}
gmcs -r:DemeterF.dll source.cs
\end{verbatim}

You can also give an absolute path for the DLL, instead of creating symbolic links, but I usually create a symbolic link in the current directory when compiling:

\begin{verbatim}
ln -s /path/to/DemeterF.dll ./
\end{verbatim}

which saves a little typing when compiling.

As I mentioned before, in Visual Studio you can add an assembly reference to your project. I haven’t had too much experience with this, but if you have success/failure please let me know.