

Mondrian for .NET

*“A language for tomorrow,
brought to you today,
by people from the future”*

Jason Smith & Nigel Perry & Erik Meijer

[About the Authors: Nigel Perry is a Senior Fellow at the University of Canterbury, Christchurch, New Zealand. Jason Smith works as a Research Assistant to Dr Perry. New Zealand is of course “in the future” compared to most of the rest of the world. Erik Meijer is a Program Manager with the Common Language Runtime team at Microsoft and adjunct professor of Computer Science at the Oregon Graduate Institute.]

Mondrian is a modern purely functional language that is specifically designed to leverage the possibilities of the .NET framework. It brings powerful algorithm expression and scripting techniques to the .NET programmer. For web programmers Mondrian introduces groundbreaking multi-language ASP.NET, where both C# and Mondrian code can be included on the same page. Mondrian is available to run under Visual Studio.NET; and best of all, it is free! You can download Mondrian from <http://www.mondrian-script.org>.

Mondrian is a purely functional language specifically designed to inter-work with object-oriented languages; as such it is a blend of the two paradigms. From its functional heritage, in particular that of Haskell, it offers:

- Higher-order Functions: This means functions are first-class values. They may be passed as function arguments, returned as results, and new functions may be created dynamically;
- JIT Evaluation: “Just In Time” evaluation where work is not done until needed and cached once the work has been done the first time (usually termed “lazy” or “non-strict” evaluation in the functional world); and
- Monadic I/O: which allows complex side-effecting computations to be constructed from simpler ones.

From the .NET common language runtime (CLR) and C#, influenced by Massey Hope[†]C and Haskell, it offers:

- “Object-oriented friendly” Types: The ways of defining and using types are rather different in the functional and object-oriented paradigms. The type system of Mondrian is designed to provide the flexibility of functional language type systems, while providing maximum compatibility with object-oriented languages. The syntax of type declarations also leans towards the object-oriented style.
- Threads: Multiple threads and thread synchronization primitives are provided, programs may consist of threads written in different languages; and
- Exceptions: Full support for exceptions is provided, including cross-language throwing and catching.

The language syntax resembles a meld of C# and traditional functional languages to simplify usage by object-oriented programmers.

Mondrian code can call routines written in other .NET hosted languages, and one of its design goals was to be useful as a scripting language. Mondrian also supports standalone programming, and being called from other .NET hosted languages. The latter enables programmers to exploit the particular strengths of functional languages in projects primarily written in other languages.

In the rest of this article we first briefly argue why functional languages have a place in the programmers toolbox, and then introduce Mondrian by example.

Why Use Functional Languages?

Functional languages are so named as they are based entirely on *functions*, the term being used here in the mathematical sense. Functional languages contain no conventional assignment or flow-of-control statements; algorithms are expressed as mappings from input values to output values. This means that in functional languages the programmer can be more concerned over the higher-level details of *what* they want accomplished, and not over the lower-level details of *how* they want it accomplished. In turn this reduces both development and maintenance costs.

Expressing an algorithm is often clearer and more concise than in traditional imperative languages. Furthermore JIT evaluation, where work is not done unless it is needed, opens up new ways of solving problems. We give examples of both of these later in the article, but first present a couple of “real world” examples where functional programming has been successfully exploited.

Composing Financial Contracts

Recent work in evaluating financial contracts has been reported by Simon Peyton Jones, of Microsoft Research, Jean-Marc Ebar, of LexiFi Technologies, and Julian Seward, of the University of Glasgow, in a paper “Composing contracts: an adventure in financial engineering”.

Financial contracts can become quite complex, but are usually composed from a set of basic operations. Higher-order programming, which allows larger functions to be composed from smaller ones, enabled them to flexibly construct more complex contracts from these basic operations; paralleling the “real world” process.

To evaluate a contract over a period of time “value trees” are used to represent a discreet approximation of the continuous process e.g. the interest rate evolution. Computing a value tree can be quite intensive since its size is quadratic in the number of time steps it covers. Furthermore complex contracts result in the combining together of many value trees, so evaluating financial contracts is traditionally computationally intensive. However Peyton Jones, Eber and Seward were able to significantly reduce the computation required by employing a functional language. This occurs as only a path through the value tree is needed, and JIT evaluation *automagically* performs just enough work to compute that path. In a traditional imperative, or object-oriented, language the whole value tree is produced, doing much unneeded computation in the process.

Designing Chips

The Fast Fourier Transform (FFT) is an algorithm familiar to engineers. Traditional implementations of FFT involve repeated iterations over arrays. In recent years researchers around the world, from places as far flung as Oxford (UK) and Auckland (NZ), have been working on new formulations of traditional algorithms. This work has produced purely functional implementations of a number of algorithms, including FFT. By “purely functional” we mean that the algorithm is defined as the composition of a number of functions. In a simple linear composition data is fed into the first function, its result becomes the input to the second function, and so on. Composition doesn't need to be linear, “networks” of functions can be created and the data “flows” around it. Function composition is a basic feature of functional programming languages and can be expressed clearly and concisely.

Digital circuits are made up of a number of “functional units” (gates, etc.) that are connected together by “wires” (connections on the chip) and functional composition is a direct model of this. This connection between functional programs and digital circuitry has caught the interested of fabricators, and functional languages are now being used to design and model real chips. Work in this area, for example, is been done at Chalmers University (Sweden) in association with various commercial companies including Xilinx.

Enter .NET

We have only scratched the surface of the real world applications of functional programming; which range from programming telephone exchanges to graphical animation packages (more info on real world applications of functional programming is available on <http://www.haskell.org/practice.html>). However it would be wrong to suggest that functional languages are the best tools for all programming tasks. Indeed it would be wrong to claim that for *any* language or paradigm; as the history of PL/1 shows, no single programming language can be completely “general purpose”.

This is where .NET enters the picture, by allowing programmers to choose the most appropriate language for different parts of their solutions. Mondrian for .NET allows you to program your whole solution in a functional language if you wish. However, just as important, it allows you to mix'n'match languages to exploit the particular power of functional programming to improve your solutions whenever the use of the functional programming paradigm is most appropriate.

Algorithm Specification: Keep It Readable

To demonstrate the clarity and conciseness functional programming can achieve we shall compare QuickSort coded in C[#] and Mondrian.

The QuickSort algorithm can be described as follows, where with “collection” we mean any collection type (list, array, etc.):

- 1) If the collection has only one element it is sorted.
- 2) Select an element for the collection, call this the “pivot”. Any element will do, often the first or last is chosen.
- 3) Partition the remaining elements in the collection into two. The first partition should contain all those elements less than the pivot, the second all those greater or equal to the pivot.

- 4) Recursively perform the algorithm on the two partitions.
- 5) Join the (now sorted) first partition, the pivot, and the second partition together to form the final sorted collection.

The core of Quick Sort written in C# shown Example 1, a full program demonstrating its use is given in Listing 1 at the end.

```

delegate bool SortComp(Object lexpr, Object rxpr);

static void QSort(Object[] Data, SortComp Cmp)
{
    QSortPart(Data, 0, Data.Length-1, Cmp);
}

static void QSortPart(Object[] Data, int Left, int Right, SortComp Cmp)
{
    if(Right <= Left) return;

    int NewPivot = Partition(Data, Left, Right, Cmp);

    QSortPart(Data, Left, --NewPivot, Cmp);
    QSortPart(Data, ++NewPivot, Right, Cmp);
}

static int Partition(Object[] Data, int Left, int Right, SortComp Cmp)
{
    int iLeft = Left - 1;
    int iRight = Right;
    int iPivot = Right; // Pick right element as the pivot
    Object PivotValue = Data[iPivot];

    while(true)
    {
        while(Cmp(Data[++iLeft], PivotValue));

        while(Cmp(PivotValue, Data[--iRight]))
            if(iRight == iLeft) break;

        if(iLeft >= iRight) break;

        Swap(Data, iLeft, iRight);
    }

    Swap(Data, iLeft, iPivot);
    return iLeft;
}

static void Swap(Object[] Data, int i, int j)
{
    Object x = Data [i];
    Data[i] = Data[j];
    Data[j] = x;
}

```

Example 1: QuickSort algorithm in C#.

This algorithm is not particularly complex, yet the correctness of it, in particular the separate function, is hard to determine. How much time over the years has been lost correcting invalid array index computations in algorithms such as this?

In contrast the core of QuickSort written in Mondrian is shown in Example 2, while a full listing of a test program is included as Listing 2 at the end.

```

// qsort : forall a. (a -> a -> Boolean) -> List<a> -> List<a>
qsort = compare -> 1 ->
  switch(1)
  {
    case []: [];
    case (pivot::t):
      let

```

```

        before = filter (x -> compare x pivot)      t;
        after  = filter (x -> not (compare x pivot)) t;
    in
        (qsort compare before)
        ++ (pivot :: (qsort compare after));
};

```

Example 2: QuickSort in Mondrian

This implementation is certainly more concise and clearer than the C# version. Its correctness is quite simple to determine, no danger of catching indexitis here.

Our implementation uses Mondrian's standard `filter` function to split the data into the two partitions required by the algorithm. The `filter` function takes a predicate function and a list, and returns all items in the list for which the predicate is true; for those interested it's definition is given in Listing 5 at the end. The code fragment:

```
x -> compare x pivot
```

denotes an inline anonymous function which uses the supplied comparison argument, `compare`, to define a predicate which selects all items less than the pivot value. The second application of `filter` selects all items greater than or equal to the pivot by simply negating (`not`) the result of the comparison function. The operator “`++`” is list concatenation.

In case some think we are “cheating” in this comparison by using the pre-defined function `filter`, here is `Partition` written in Mondrian:

```

partition = pred -> data -> before -> after ->
    switch(data)
    { case []:
        Pair{ a = before;
              b = after;
            };
      case (first::tail):
        if(pred first)
            partition pred tail (first::before) after
        else
            partition pred tail before (first::after);
    };

```

Example 3: Partition algorithm in Mondrian

The type `Pair` is standard in Mondrian and enables, among other things, functions to easily return two values. Its definition is trivial and it is equivalent to a two field structure in C#. Listing 3 shows QuickSort using `partition`.

We will return to QuickSort later when we discuss Mondrian for ASP.NET.

JIT Evaluation: Don't Do Unnecessary Work!

JIT evaluation is a key concept in lazy functional programming, where it is usually termed “non strict” or “lazy” evaluation. JIT evaluation simply means that a computation is not actually performed until its result is needed, and that once the computation has been performed its value is cached. In particular, this means that unlike most programming languages, arguments to functions are not evaluated unless the function actually needs the value. JIT evaluation also allows the creation of “infinite” data structures; that is data structures whose definition specifies a large, or infinite,

size but which are only actually constructed as far as needed by the application. For example the following Mondrian code defines the “infinite” list of all integers • n.

```
// from : Integer -> List<Integer>;  
from = n -> n :: from(n + 1);
```

where “::” is Mondrian’s list construction operator.

A call such as `from 2` returns immediately as the recursive call is not actually performed until the tail of the list is required. In contrast, if `from` would be defined in a strict language such as C# then the call `from(2)` either a stack overflow would occur, or, if you have a huge amount of memory, an overflow exception would result as C# tried to create a list of all integers • 2.

Another example is the Fibonacci sequence.

```
// fibList : List<Integer>;  
fibList =  
  let  
    fibHelper = a -> b -> a :: (fibHelper b (a+b));  
  in  
    fibHelper 1 1;
```

The Fibonacci function `fibList` defines an “infinite” list of all the Fibonacci numbers. Again, the result will be evaluated as needed, that is as numbers are extracted from the list.

The Sieve of Eratosthenes

Primes are useful in many algorithms, for example cryptography algorithms and random number generators often use primes. The *Sieve of Eratosthenes* is a well-known and simple algorithm for generating primes:

- 1) Initialize some collection (array, list, set, etc.) to contain the integers starting from 2, up to some limit.
- 2) Remove the least number from this collection; it is a prime.
- 3) If another prime is required then remove from the collection all multiples of the prime found in step (2).
- 4) Go to step 2.

To code Sieve in a procedural language an array of some fixed size, n, is used which is then sieved to produce all the primes • n. A C# version is given in Example 3.

```
static ArrayList primes(int limit)  
{  
  int [] sieve = new int[limit];  
  ArrayList found = new ArrayList();  
  
  // initialise array  
  for (int i = 2; i < limit; i++) sieve[i] = i;  
  
  // find primes  
  for (int cursor = 2; cursor < limit; cursor++)  
  {  
    if (sieve[cursor] != 0)  
    {  
      // found a prime  
      found.Add(cursor);  
      // Sieve the array  
      for (int j = cursor + 1; j < limit; j++)
```

```

        if (sieve[j]%cursor == 0)
            sieve[j] = 0;
    }
}
return found;
}

```

Example 3

where `ArrayList` is from the .NET Framework and provides an extensible array. Note however that the algorithm cannot use an extendible array for the collection from which primes are generated, this must be a fixed sized array, or the sieve would not work.

The Sieve algorithm coded in Mondrian uses the build-in `from` function to generate a list of all the natural numbers and then sieves this list to calculate a list of all the primes “on-demand.” In contrast to the procedural version this implementation can determine the first n , for all n , primes. Example 4 shows the Sieve coded in Mondrian.

```

// sieve : List<Integer> -> List<Integer>;
sieve = xs ->
  switch (xs)
  { case (y::ys):
      // keep head, remove all multiples of head from tail,
      // then recursively sieve tail
      y :: sieve (filter (x -> x % y != 0) ys);
  };
primes = sieve (from 2);

```

Example 4

As with `QuickSort` above we use the standard `filter` function. In this case the predicate is the inline function:

```
x -> x % y != 0
```

that returns true if x is not a multiple of y .

The Best of Both Worlds

The Mondrian realization of the Sieve of Eratosthenes is both simpler and more flexible than the C# version; it can provide both all the primes $\leq n$ and the first n primes. However calculating primes is probably only a small part of a particular solution and other parts of that solution might be better written, for various reasons, in another language, such as C#. Can we combine the benefits of using Mondrian for algorithms such as these with the benefits provided by other languages in other areas, for example in producing GUI's? The answer with .NET is of course yes! One of the strengths of .NET is its support from multi-language programming. Whether you wish to use a whole library written in another language, or just a single routine, .NET through it's Common Language Runtime (CLR) provides a simple way to achieve this. Well for some languages anyway.

As you might guess Mondrian with its JIT evaluation is not compiled in the quite the same way as typical object-oriented and procedural languages. This doesn't mean JIT evaluation cannot be compiled well onto .NET, it just means that calling a Mondrian function from, say, C# is a little, but only a little, different than calling, say, a Visual Basic function.

The code in Example 5 is a C# class which provides an object-oriented iterator-style interface to the Mondrian prime generator shown in Example 4.

```
class PrimeIterator
{ private Object list = primeGenerator.primes.Apply();

  public int Current()
  { return (int)mondrian.prelude.hd.Apply(list);
  }

  public void Next()
  { list = mondrian.prelude.tl.Apply(list);
  }
}
```

Example 5

A Mondrian function, such as `primes`, is accessible from other .NET languages as a class with a method `Apply`. The `Apply` method handles the interface between Mondrian's JIT evaluation model and the strict model of .NET, similar to the standard class `System.Delegate` provides an `invoke` method. The functions/classes `mondrian.prelude.hd` and `mondrian.prelude.tl` are the standard Mondrian functions for returning the head and tail of a list respectively.

The class `PrimeIterator` provides a more flexible prime generator than the C# code shown in Example 3. This demonstrates some of the power of .NET; users of the `PrimeIterator` class never need to know what language it was written in, they never need to even have heard of JIT evaluation; all they see is a very clever class which they've no idea how to write themselves in C#, C++ or Visual Basic!

Scripting: Control Using Mondrian

Mondrian's "function plus data" model is rather different than .NET's "object plus method" model and this causes a slight impedance match when calling Mondrian from typical object-oriented languages, as shown in the last section. However the "object plus method" model fits well into Mondrian's command expressions (known as *monads* by functional programmers). Command expressions enable Mondrian to be used very effectively to script code written in other .NET hosted languages. As with functions, individual command expressions can be combined to produce more complex operations. Normal functions and JIT evaluation may also be exploited to produce scripts not easily written in other languages.

Example 6 demonstrates the use of Mondrian to call and connect functions written in another CLR hosted language. In this case the functions are from the .NET Framework, whatever language(s) that is written in, as this is .NET it doesn't matter!

```
// readLinesFromURL :: String -> IO<List<String>>;
readLinesFromURL = url ->
{ try
  { req <- HttpRequest.Create(url);
    rsp <- req # HttpRequest.GetResponse();
    str <- rsp # WebResponse.GetResponseStream();
    readLines str;
  }
  catch (e : Exception)
  { result Nil;
  };
};
```

Example 6. Reading from a URL using Mondrian and the .NET Framework.

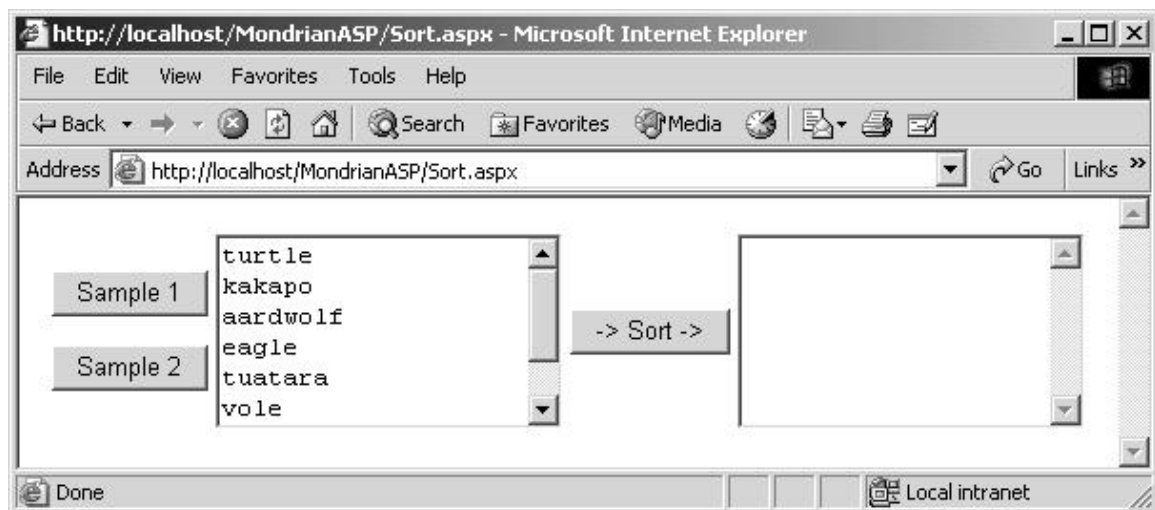
The classes `HttpRequest` and `WebResponse` are from the .NET Framework, the first three lines of the `readLinesFromURL` function open a URL and return a stream from which the contents of the item referred to by the URL may be read. The command function `readLines` is provided by Mondrian, it takes a stream and returns a list of all the lines read from the URL. This function relies on JIT evaluation; the complete stream is not read in one go, it could be gigabytes in length, it is read as *needed*. JIT evaluation enables the URL stream to be processed as though it was all in memory at once, avoiding the need to coordinate processing the current line, reading the next line, etc.

The `try/catch` command construct shown above mirrors that of .NET and languages such as C#. Unusually for a functional language Mondrian can handle general exceptions, including those thrown by other .NET hosted languages it calls, and throw its own.

ASP.NET: Multi-language Web Scripting

Any language hosted on .NET can be used for coding ASP.NET pages if the language provider chooses to implement "CodeDom" support. The ease of supporting CodeDom ranges from trivial, for languages such as C# and Visual Basic, to very involved, for languages which are far removed from C# or Visual Basic. Mondrian of course falls into the latter category. However by devising a new approach to CodeDom support, not only does Mondrian support ASP.NET, in the true spirit of .NET it also provides multi-language ASP.NET pages. Currently a mixture of Mondrian and C# is allowed, other languages may be added in the future.

But why would you want to use multiple languages on your ASP.NET pages? The answer is of course much the same as for other types of programming; to exploit the best tools for solving the problem. We will explore Mondrian for ASP.NET by coding a simple form that provides a string sorting service:



On this form text may be typed into the left field, or one of the two sample buttons used to enter sample text. Pressing the sort button sorts the lines in the left field and places the result into the right field. To produce this we first need to define an ASP.NET form, as shown in Example 7.

```
<form method="POST" action="SortMondrian.aspx" runat=server>  
<table border=0 cellpadding=4>
```

```

<tr>
  <td valign=center>
    <asp:Button id="I1" text="Sample 1" OnClick="Init1"
      runat="server" />
    <br>&nbsp;<br>
    <asp:Button id="I2" text="Sample 2" OnClick="Init2"
      runat="server" /><br>
  </td>
  <td valign=center>
    <asp:TextBox id="T" rows="6" textmode="multiline" text=""
      runat="server" />
  </td>
  <td valign=center align=center>
    <asp:Button id="B" text="-> Sort ->" OnClick="DoSort"
      runat="server" />
  </td>
  <td valign=center>
    <asp:TextBox id="T2" rows="6" textmode="multiline" text=""
      runat="server" />
  </td>
</tr>
</table>
</form>

```

Example 7: ASP.NET form definition

In this form every button has a handler method defined for the `OnClick` event. Defining the two initialization methods in C# is trivial, as shown in Example 8.

```

void Init1(object sender, EventArgs e)
{ T.Text = "turtle\nkakapo\nnaardwolf\nneagle\ntuatara\nvole\nbadger";
}

void Init2(object sender, EventArgs e)
{ T.Text = "reversed\ninput\ndata";
}

```

Example 8: C# handlers for Sample buttons

The handler for the sort button however is a little more involved. The contents of an `asp:TextBox` control is a single string, so to sort the lines this string must be broken up into the individual lines, these lines sorted, and then the re-ordered lines joined back together into a single string. Such data manipulation is one of Mondrian's strong points; a function to do the above is shown in Example 9.

```

SortLines = cs ->
  let
    l = lines (stringToList cs);
    r = qsort stringLT l;
  in
    listToString (unlines r);

```

Example 9

Lists are the natural data type in Mondrian, so the input string is first converted to a list using `stringToList`, and the final result is converted to a string using `listToString`. The `qsort` function was defined earlier in Example 2, while `lines` and `unlines` are standard Mondrian functions. To attach `SortLines` to the ASP.NET button another small C# routine is used, as shown in Example 10.

```

void Clicked(object sender, EventArgs e)
{ T2.Text = (string)SortLines.Apply(T.Text);
}

```

Example 10

Finally the above code and HTML fragments need to be combined into a single ASP.NET page. To do this requires Mondrian's multi-language support, the page outline is shown in Example 11, the complete page is shown in Listing 4.

```
<%@ Page Language="Mondrian" %>

<script runat="server">
// qsort - Example 2
// SortLines - Example 9
</script>

<script runat="server">[C#]
// C# handlers - examples 8 and 10
</script>

<html>
<body>
// the form - Example 7
</body>
</html>
```

Example 11: Mondrian for ASP.NET page outline

In a Mondrian ASP.NET page the default language of a script is Mondrian. To include C# code a language marker [C#] is added after the opening <script> tag. Mondrian and C# routines can call each other and both can access elements on the page.

Visual Studio Integration

Of course the end user experience wouldn't be complete unless you could program in your favorite functional language i.e. Mondrian in your equally favorite RAD environment such as Visual Studio .NET. To that end, we have integrated Mondrian into the Visual Studio .NET environment. Example 12 shows a screen shot of the Mondrian/C# Sieve of Eratosthenes discussed above.


```

        QSort(a, new SortComp(StrComp));

        for (int i = 0; i < a.Length; i++)
        {
            System.Console.WriteLine(a[i]);
        }

        System.Console.WriteLine("Please press Enter");
        System.Console.Read();
    }

    static bool StrComp(Object lxpr, Object rxpr)
    {
        return ((String)lxpr).CompareTo(rxpr) < 0;
    }

    // QSort sort an array of objects
    // Params:
    //   Data - Data of object's to sort
    //   Cmp - Comparison function that order's the list items.

    public delegate bool SortComp(Object lxpr, Object rxpr);

    static void QSort(Object[] Data, SortComp Cmp)
    {
        QSortPart(Data, 0, Data.Length-1, Cmp);
    }

    static void QSortPart(Object[] Data,
                          int Left, int Right,
                          SortComp Cmp)
    {
        if(Right <= Left) return;

        int NewPivot = Partition(Data, Left, Right, Cmp);

        QSortPart(Data, Left, --NewPivot, Cmp);
        QSortPart(Data, ++NewPivot, Right, Cmp);
    }

    static int Partition(Object[] Data,
                        int Left, int Right,
                        SortComp Cmp)
    {
        int iLeft = Left - 1;
        int iRight = Right;
        int iPivot = Right; // Pick right element as the pivot
        Object PivotValue = Data[iPivot];

        while(true)
        {
            while(Cmp(Data[++iLeft], PivotValue));

            while(Cmp(PivotValue, Data[--iRight]))
                if(iRight == iLeft) break;

            if(iLeft >= iRight) break;

            Swap(Data, iLeft, iRight);
        }

        Swap(Data, iLeft, iPivot);
        return iLeft;
    }

    static void Swap(Object[] Data, int i, int j)
    {
        Object x = Data [i];
        Data[i] = Data[j];
        Data[j] = x;
    }
}
}

```

Listing 1.

```
package Sorting;
```

```

import mondrian.prelude;

// qsort : forall a. (a -> a -> Boolean) -> List<a> -> List<a>;
qsort = compare -> l ->
  switch(l)
  { case []: [];
    case (pivot::t):
      let
        before = filter (x -> compare x pivot) t;
        after  = filter (x -> not (compare x pivot)) t;
      in
        (qsort compare before)
        ++ (pivot :: (qsort compare after));
  };

main =
  let
    data = map stringToList ["the", "quick", "brown",
                              "fox", "jumped"];
    sort = qsort stringLT data;
  in
  { putStrLn (listToString (unlines sort));
    putStrLn "Please press Enter";
    getChar;
  };

```

Listing 2.

```

package Sorting;

import mondrian.prelude;

// qsort : forall a. (a -> a -> Boolean) -> List<a> -> List<a>;
qsort = compare -> l ->
  switch(l)
  { case []: [];
    case (pivot::t):
      let
        pair = partition (x -> compare x pivot) t [] [];
      in
        (qsort compare (fst pair)
         ++ (pivot :: (qsort compare (snd pair))));
  };

// partition: forall a. (a -> Boolean)
//                -> List<a> -> List<a> -> List<a>
//                -> Pair<List<a>, List<a>>
partition = pred -> data -> before -> after ->
  switch(data)
  { case []:
    Pair{ a = before;
          b = after;
    };
    case (first::tail):
    if(pred first)
      partition pred tail (first::before) after
    else
      partition pred tail before (first::after);
  };

main =
  let
    data = map stringToList ["the", "quick", "brown",
                              "fox", "jumped"];
    sort = qsort stringLT data;
  in
  { putStrLn (listToString (unlines sort));
    putStrLn "Please press Enter";
    getChar;
  };

```

Listing 3.

```
<%@ Page Language="Mondrian" %>
<script runat="server">
    qsort = compare -> l ->
        switch(l)
        { case []: [];
          case (h::t):
            let
                before = filter (x -> compare x h) t;
                after = filter (x -> not (compare x h)) t;
            in
                (qsort compare before)
                ++ (h :: (qsort compare after));
        };
    SortLines = cs ->
        let
            l = lines (stringToList cs);
            r = qsort stringLT l;
        in
            listToString (unlines r);
</script>
<script runat="server">[C#]
    void Clicked(object sender, EventArgs e)
    { T2.Text = (string)SortLines.Apply(T.Text);
    }
    void Init1(object sender, EventArgs e)
    { T.Text =
"turtle\nkakapo\naardwolf\nneagle\ntuatara\nvole\nbadger\nswordfish";
    }
    void Init2(object sender, EventArgs e)
    { T.Text = "reversed\ninput\ndata";
    }
</script>
<html>
<body>
<form method="POST" action="SortMondrian.aspx" runat=server>
<table border=0 cellpadding=4>
    <tr>
        <td valign=center>
            <asp:Button id="I1" text="Sample 1" OnClick="Init1"
                runat="server" /><br>
            &nbsp;&nbsp;&nbsp;<br>
            <asp:Button id="I2" text="Sample 2" OnClick="Init2"
                runat="server" /><br>
        </td>
        <td valign=center>
            <asp:TextBox id="T" rows="6" textmode="multiline" text=""
                runat="server" />
        </td>
        <td valign=center align=center>
            <asp:Button id="B" text="-> Sort ->" OnClick="Clicked"
                runat="server" />
        </td>
        <td valign=center>
            <asp:TextBox id="T2" rows="6" textmode="multiline" text=""
                runat="server" />
        </td>
    </tr>
</table>
```

```
</form>  
</body>  
</html>
```

Listing 4

```
// filter : forall a. (a -> Boolean) -> List<a> -> List<a>;  
filter = pred -> elems ->  
  switch(elems)  
  { case Nil:  
    Nil;  
    case Cons{h = head; t = tail;};  
    if (pred h)  
      then h :: (filter pred t)  
      else filter pred t  
  };
```

Listing 5