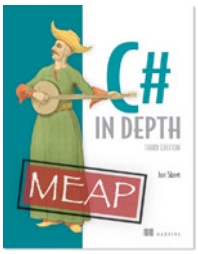


# C# Implementation Dive: Compiler Transformation

**By Jon Skeet, author of *C# in Depth, Third Edition***

*Asynchrony has been a thorn in the side of developers for years. It's been known to be useful as a way to avoid tying up a thread while waiting for some arbitrary task to complete, but it's also been a pain in the neck to implement correctly. In this article, based on chapter 15 of C# in Depth, Third Edition, author Jon Skeet discuses compiler transformations for async.*

You may also be interested in…

I vividly remember the evening of October 28, 2010. Anders Hejlsberg was presenting async/await at the Microsoft Professional Develpers Conference (PDC), and shortly before his talk started, an avalanche of downloadable material was made available—including a draft of the changes to the C# specification, a Community Technology Preview (CTP) of the C# 5 compiler, and the slides Anders was presenting. At one point, I was watching the talk live and skimming through the slides while the CTP installed. By the time Anders had finished, I was writing async code and trying things out.

In the next few weeks, I started taking bits apart—looking at exactly what code the compiler was generating, trying to write my own simplistic implementation of the library that came with the CTP and generally poking at it from every angle. As new versions came out, I worked out what had changed and became more and more comfortable with what was going on behind the scenes. The more I saw, the more I appreciated how much boilerplate code the compiler is happy to write on our behalf. It's like looking at a beautiful flower under a microscope: the beauty is still there to be admired, but there's so much more to it than can be seen at first glance.

Not everyone is like me, of course. If you just want to rely on the behavior I've already described, and simply trust that the compiler will do the right thing, that's *absolutely fine*. It's unlikely that you'll ever have to debug your code down to the level that we'll look at here, but I believe this article will give you more insight into how the whole feature hangs together. The awaitable pattern certainly makes a lot more sense once you've looked at the generated code, and you'll see some of the types that the framework provides to help the compiler. Some of the scariest details are only present due to optimization; the design and implementation are very carefully tuned to avoid unnecessary heap allocations and context switches, for example.

As a rough approximation, we'll pretend the C# compiler performs a transformation from "C# code using async/await" to "C# code without using async/await." In reality, the internals of the compiler aren't available to us, and it's more than likely that this transformation occurs at a lower level than C#. Certainly the generated IL can't always be expressed in non-async C#, as C# has tighter restrictions around flow control than IL does. But it's simpler for us to think of it as C#, in terms of how the jigsaw of code fits together.

The generated code is somewhat like an onion, with layers of complexity. We'll start from the very outside, working our way in toward the tricky bit—`await` expressions and the dance of awaiters and continuations.

## Overview of the generated code

Still with me? Let's get started. I won't go into *all* the depth I could here—that could fill hundreds of pages—but I'll give you enough background to understand the overall structure, and then you can either read the various blog posts I've written over the past couple of years for more intricate detail, or simply write some asynchronous code and decompile it. Also, I'll only cover asynchronous methods—that will include all the interesting

machinery, and you won't need to deal with the extra layer of indirection that asynchronous anonymous functions present.

**WARNING, BRAVE TRAVELER—HERE BE IMPLEMENTATION DETAILS!**

This section documents some aspects of the implementation found in the Microsoft C# 5 compiler, released with .NET 4.5. A few details changed pretty substantially between CTP versions and in the beta, and they may well change again in the future. But I think it unlikely that the fundamental *ideas* will change much though. If you understand enough of this article to be comfortable that there's no magic involved, just really clever compiler-generated code, you should be able to take any future changes to the details in stride.

The implementation (both in this approximation and in the code generated by the real compiler) is basically in the form of a *state machine*. The compiler will generate a private nested struct to represent the asynchronous method, and it must also include a method with the same signature as the one you've declared. I call this the *skeleton method*—there's not much to it, but everything else hangs off it.

The skeleton method needs to create the state machine, make it perform a single step (where a *step* is whatever code executes before the first genuinely waiting `await` expression), and then return a task to represent the state machine's progress. (Don't forget that until you hit the first `await` expression that actually needs to wait, execution is synchronous.) After that, the method's job is done—the state machine looks after everything else, and continuations attached to other asynchronous operations simply tell the state machine to perform another step. The state machine signals when it's reached the end by giving the appropriate result to the task that was returned earlier. Figure 1 shows a flow diagram of this, as best I can represent it.

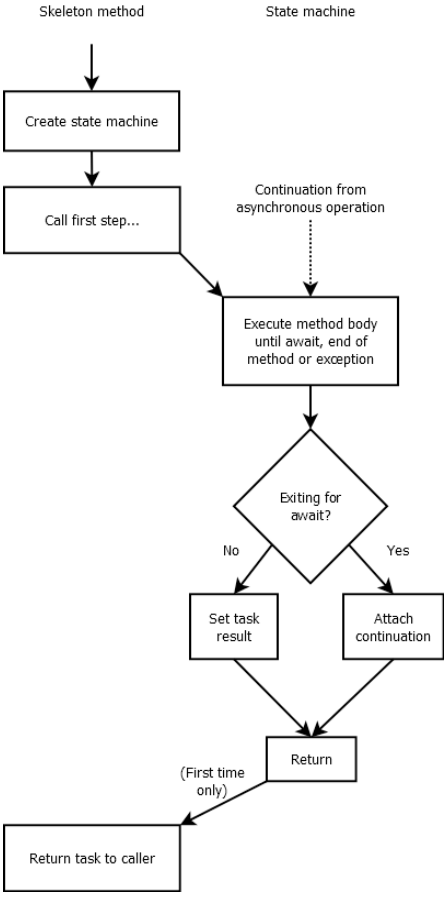


Figure 1 Flowchart of generated code

Of course the "execute method body" step only starts from the beginning of the method the first time it's called, from the skeleton method. After that, each time you get to that block, it's via a continuation, at which point execution effectively continues from where it left off.

We now have two things to look at: the skeleton method and the state machine. For most of the remainder of this section, I'll use a single sample asynchronous method, shown in the following listing.

**Listing 1 Simple async method to demonstrate compiler transformations**

```
static async Task<int> SumCharactersAsync
                     (IEnumerable<char> text)
{
    int total = 0;
    foreach (char ch in text)
    {
        int unicode = ch;
        await Task.Delay(unicode);
        total += unicode;
    }
    await Task.Yield();
    return total;
}
```

Listing 1 doesn't do anything *useful*, but we're really just interested in the control flow. It's worth noting a few points before we start:

- The method has a parameter (`text`).
- It contains a loop that you effectively need to jump back into when the continuation executes.
- It has two `await` expressions of different types: `Task.Delay` returns a `Task`, but `Task.Yield()` returns a `YieldAwaitable`.
- It has obvious local variables (`total`, `ch`, and `unicode`) that you'll need to keep track of across calls.
- It has an implicit local variable created by calling `text.GetEnumerator()`. It returns a value at the end of the method.

The original version of this code had `text` as a `string` parameter, but the C# compiler knows about iterating over strings in an efficient way, using the `Length` property and the indexer, which made the decompiled code more complicated.

I won't present the *complete* decompiled code, although it's in the downloadable source. In the next few sections we'll look at a few of the most important parts. You won't see exactly this code if you decompile the code yourself; I've renamed variables and types so they're rather more legible, but it's effectively the same code.

Let's start off with the simplest bit—the skeleton method.

## Structure of the skeleton method

Although the code in the skeleton method is simple, it offers some hints about the state machine's responsibilities. The skeleton method generated for listing 1 looks like this:

```
[DebuggerStepThrough]
[AsyncStateMachine(typeof(DemoStateMachine))]
static Task<int> SumCharactersAsync(IEnumerable<char> text)
{
    var machine = new DemoStateMachine();
    machine.text = text;
    machine.builder = AsyncTaskMethodBuilder<int>.Create();
    machine.state = -1; machine.builder.Start(ref machine);
    return machine.builder.Task;
}
```

The `AsyncStateMachineAttribute` type is just one of the new attributes introduced for `async`. It's really for the benefit of tools—you're unlikely to ever need to consume it yourself, and you shouldn't start decorating your own methods with it.

You can see three of the fields of the state machine already:

1. One for the parameter (`text`). Obviously there are as many fields here as there are parameters.

2. One for an `AsyncTaskMethodBuilder<int>`. This struct is effectively responsible for tying the state machine and the skeleton method together. There's a nongeneric equivalent for methods returning just `Task` and an `AsyncVoidMethodBuilder` structure for methods returning `void`.
3. One for `state`, starting off with a value of `-1`. The initial value is always `-1`, and we'll take a look at what various possible values mean later.

Given that the state machine is a struct, and `AsyncTaskMethodBuilder<int>` is a struct, you haven't knowingly performed *any* heap allocation yet. It's entirely possible for the various calls you're making to have done so for you, of course, but it's worth noting that the code tries to avoid them as far as possible. The nature of asynchrony means that if any `await` expressions need to really wait, you'll need a lot of these values on the heap, but the code makes sure that they're only boxed when they need to be. All of this is an implementation detail, just like the heap and the stack are implementation details, but in order for `async` to be practical in as many situations as possible, the teams involved in Microsoft have worked closely to reduce allocations to the bare minimum.

The `machine.builder.Start(ref machine)` call is an interesting one. The use of pass-by-reference here allows you to avoid creating a copy of the state machine (and thus a copy of the builder)—this is for both performance and correctness. The compiler would really *like* to treat both the state machine and the builder as classes, so `ref` is used liberally throughout the code. In order to use interfaces, various methods take the builder (or awaiter) as a parameter using a generic type parameter that's constrained to implement an interface (such as `IAsyncStateMachine` for the state machine). That allows the members of the interface to be called without any boxing being required. The *action* of the method is simple to describe—it makes the state machine take the first step, synchronously, returning only when the method has either completed or reached a point where it needs to wait for an asynchronous operation.

Once that first step has completed, the skeleton method asks the builder for the task to return. The state machine uses the builder to set results or exceptions when it finishes.

## Structure of the state machine

The overall structure of the state machine is pretty straightforward. It always implements the `IAsyncStateMachine` interface (introduced in .NET 4.5) using explicit interface implementation. The two methods declared by that interface (`MoveNext` and `SetStateMachine`) are the only two methods it contains. It also has a bunch of fields—some private, some public.

For example, this is the collapsed declaration of the state machine for listing 1:

```
[CompilerGenerated]
private struct DemoStateMachine : IAsyncStateMachine
{
    public IEnumerable<char> text;  #1

    public IEnumerator<char> iterator;  #2
    public char ch;
    public int total;
    public int unicode;

    private TaskAwaiter taskAwaiter;  #3
    private YieldAwaitable.YieldAwaiter yieldAwaiter;

    public int state;  #4
    public AsyncTaskMethodBuilder<int> builder;

    private object stack;

    void IAsyncStateMachine.MoveNext() { ... }

    [DebuggerHidden]
    void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine machine)
    { ... }
}
```

**#1 Fields for parameters**
**#2 Fields for local variables**
**#3 Fields for awaiters**
**#4 Common infrastructure**

In this example I've split the fields up into various sections. You've already seen that the `text` field (#1) representing the original parameter is set by the skeleton method, along with the `builder` and `state` fields, which are common infrastructure shared by all state machines.

Each local variable also has its own field (#2), as you need to preserve the values across invocations of the `MoveNext()` method. Sometimes there are local variables that are only ever used between two particular `await` expressions, and don't really *need* to be preserved in fields, but in my experience the current implementation always hoists them up to be fields anyway. Aside from anything else, this improves the debugging experience, as you wouldn't generally expect local variables to lose their values, even if there's nothing in the code that uses them any further.

There's a single field for each type of awaiter used in the asynchronous method if they're value types, and one field for all awaiters that are reference types (in terms of their compile-time type). In this case, you've got two `await` expressions that use two different types of awaiter structures, so you've got two fields (#3). If the second `await` expression had also used a `TaskAwaiter`, or if `TaskAwaiter` and `YieldAwaiter` were both classes, you'd just have a single field. Only one awaiter can ever be live at a time, so it doesn't matter that you can only store one value at a time. You have to propagate awaiters across `await` expressions so that once the operation has finished, you can get the result.

Out of the common infrastructure fields (#4), you've already seen `state` and `builder`. Just as a reminder, `state` is used to keep track of where you've got to, so the continuation can get to the right point in the code.

`builder` is used for various things, including creating a `Task` or `Task<T>` for the skeleton method to return—a task that will then be populated with the right result when the asynchronous method finishes. The `stack` field is a little more arcane—it's used when an `await` expression occurs as part of a statement that needs to keep track of some extra state that isn't represented by normal local variables. You'll see an example of that further in this article—it's not used in the state machine generated for listing 1.

The `MoveNext()` method is where all the compiler smarts really come into play, but before I describe that, we'll take a *very* quick look at `SetStateMachine`. It has the same implementation in every state machine, and it looks like this:

```
void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine machine)
{
    builder.SetStateMachine(machine);
}
```

In brief, this method is used to allow a boxed copy of the state machine to have a reference to itself, within the builder. I won't go into the details of how all the boxing is managed—all you need to understand is that the state machine *is* boxed where necessary, and the various aspects of the async machinery ensure that after boxing, the single boxed copy is used consistently. This is really important, as we're talking about a *mutable value type* (shudder!). If some changes were applied to one copy of the state machine, and some changes were applied to another copy, the whole thing would fall apart very quickly.

If you want to think of it another way—and this will be important if you ever start *really* thinking about how the instance variables of the state machine are propagated—the state machine is a `struct` to avoid unnecessary heap allocations early on, but most of the code tries to *act* like it's really a class. The reference juggling around `SetStateMachine` makes it all work.

Right! Now we've got everything in place except for the actual code that was in the asynchronous method. Let's dive into `MoveNext()`.

## One entry point to rule them all

If you ever decompile an async method—and I really hope you will—you'll see that the `MoveNext()` method in the state machine gets very long, very fast, mostly as a function of how many `await` expressions you have. It contains all the logic in the original method, *and* the delicate ballet required to handle all the state transitions,[1] *and* some wrapper code to handle the overall result or exception.

When writing asynchronous code by hand, you'd typically put continuations into separate methods: start in one method, then continue in another, and maybe finish in a third. But that makes it hard to handle flow control, such

[1] It really does feel like a dance, with intricate steps that have to be performed at exactly the right time and place.

as loops, and it's unnecessary for the C# compiler. It's not like the readability of the generated code matters. The state machine has a single entry point, `MoveNext()`, which is used from the start and for the continuations for all `await` expressions. Each time `MoveNext()` is called, the state machine works out where in the method to get to via the `state` field. This is either the logical starting point of the method or the end of an `await` expression, when you're ready to evaluate the result. Each state machine is executed only once. Effectively there's a switch statement based on `state`, with different cases corresponding to `goto` statements with different labels.

The `MoveNext()` method typically looks something like this:

```
void IStateMachine.MoveNext()
{
    // For an asynchronous method declared to return Task<int>
    int result;
    try
    {
        bool doFinallyBodies = true;
        switch (state)
        {
            // Code to jump to the right place...
        }

        // Main body of the method
    }
    catch (Exception e)
    {
        state = -2;
        builder.SetException(e);
        return;
    }
    state = -2;
    builder.SetResult(result);
}
```

The initial state is always `-1`, and that's *also* the state when the method is executing your code (as opposed to being paused while awaiting). Any non-negative states indicate the target of a continuation. The state machine ends up in state `-2` when it's completed. In state machines created in debug configurations, you'll see reference to a state of `-3`—it's never expected that you'll actually *end up* in that state. It's there to avoid having a degenerate switch statement, which would result in a poorer debugging experience.

The `result` variable is set during the course of the method, at the point where the original async method had a `return` statement. This is then used in the `builder.SetResult()` call, when you reach the logical end of the method.

Even the nongeneric `AsyncTaskMethodBuilder` and `AsyncVoidMethodBuilder` types have `SetResult()` methods; the former communicates the fact that the method has completed to the task returned from the skeleton method, and the latter signals completion to the original `SynchronizationContext`. (Exceptions are propagated to the original `SynchronizationContext` in the same way. It's a rather dirtier way of keeping track of what's going on, but it provides a solution for situations where you really *must* have `void` methods.)

The `doFinallyBodies` variable is used to work out whether any `finally` blocks in the original code (including implicit ones from `using` or `foreach` statements) should be executed when execution leaves the scope of a `try` block.

Conceptually, you only want to execute a `finally` block when you leave the `try` block in a normal way. If you're just returning from the method early having attached a continuation to an awaiter, the method is logically "paused" so you don't want to execute the `finally` block. Any `finally` blocks would appear within the `Main body of the method` section of code, along with the associated `try` block.

Most of the body of the method is recognizable in terms of the original async method. Admittedly, you need to get used to all the local variables now appearing as instance variables in the state machine, but that's not too hard. The tricky bits are all around `await` expressions—as you might expect.

## Control around await expressions

Just as a reminder, any `await` expression represents a fork in terms of possible execution paths. First the awaiter is fetched for the asynchronous operation being awaited, and then its `IsCompleted` property is

checked. If that returns `true`, you can get the results immediately and continue. Otherwise, you need to do the following:

- Remember the awaiter for later.
- Update the state to indicate where to continue from.
- Attach a continuation to the awaiter.
- Return from `MoveNext()`, ensuring that any `finally` blocks are *not* executed.

Then, when the continuation is called, you need to jump to the right point, retrieve the awaiter, and reset your state before continuing.

As an example, the first `await` expression in listing 1 looks like this:

```
await Task.Delay(unicode);
```

The generated code looks like this:

```
TaskAwaiter localTaskAwaiter = Task.Delay(unicode).GetAwaiter();
   if (localTaskAwaiter.IsCompleted)
   {
       goto DemoAwaitCompletion;
   }
   state = 0;
   taskAwaiter = localTaskAwaiter;
   builder.AwaitUnsafeOnCompleted(ref localTaskAwaiter, ref this);
   doFinallyBodies = false;
   return;
DemoAwaitContinuation:
   localTaskAwaiter = taskAwaiter;
   taskAwaiter = default(TaskAwaiter);
   state = -1;
DemoAwaitCompletion:
   localTaskAwaiter.GetResult();
   localTaskAwaiter = default(TaskAwaiter);
```

If you'd been awaiting an operation that returned a value—for example, assigning the result of `await client.GetStringAsync(...)` using an `HttpClient`—the `GetResult()` call near the end would be where you'd get the value.

The `AwaitUnsafeOnCompleted` method attaches the continuation to the awaiter, and the switch statement at the start of the `MoveNext()` method would ensure that when `MoveNext()` executes again, control passes to `DemoAwaitContinuation`.

**AwaitOnCompleted vs. AwaitUnsafeOnCompleted**

Earlier, I showed you a notional set of interfaces, where `IAwaiter<T>` extended `INotifyCompletion` with its `OnCompleted` method. There's also an `ICriticalNotifyCompletion` interface, with an `nsafeOnCompleted` method. The state machine calls `builder.AwaitUnsafeOnCompleted` for awaiters that implement `ICriticalNotifyCompletion`, or `builder.AwaitOnCompleted` for awaiters that only implement `INotifyCompletion`. We'll look at the differences between these two calls further in this article when we discuss how the awaitable pattern interacts with contexts.

Note that the compiler wipes both the local and instance variables for the awaiter, so that it can be garbage-collected where appropriate.

Once you can identify a block like this as corresponding to a single `await` expression, the generated code really isn't *too* bad to read in decompiled form.

There may be more `goto` statements (and corresponding labels) than you'd expect, due to CLR restrictions, but getting your head round the `await` pattern is the biggest hump in understanding, in my experience.

There's one thing I still need to explain—the mysterious `stack` variable in the state machine.

## Keeping track of a stack

When you think of a stack frame, you probably think about the local variables you've declared in the method. Sure, you may be aware of some hidden local variables like the iterator for a `foreach` loop, but that's not all that

goes on the stack, at least logically.[2] In various situations, there are intermediate expressions that can't be used until some other expressions are evaluated. The simplest examples of these are binary operations like addition, and method invocations.

As a trivial example, consider this line:

```
var x = y * z;
```

In stack-based pseudocode, that's something like this:

```
push y
push z
multiply
store x
```

Now suppose you have an `await` expression in there:

```
var x = y * await z;
```

You need to evaluate `y` and store it somewhere before you await `z`, but you might well end up returning from the `MoveNext()` method immediately, so you need a logical stack to store `y` on. When the continuation executes, you can restore the value and perform the multiplication. In this case, the compiler can assign the value of `y` to the `stack` instance variable. This does involve boxing, but it means you get to use a single variable.

That's a simple example. Imagine you had something where multiple values needed to be stored, like this:

```
Console.WriteLine("{0}: {1}", x, await task);
```

You need both the format string and the value of `x` on your logical stack. This time, the compiler creates a `Tuple<string, int>` containing the two values, and stores that reference in `stack`. Like the awaiter, you only ever need a single logical stack at a time, so it's fine to always use the same variable.[3]

In the continuation, the individual arguments can be fetched from the tuple and used in the method call. The downloadable source code contains a complete decompilation of this sample, with both of the preceding statements (`LogicalStack.cs` and `LogicalStackDecompiled.cs`).

The second statement ends up using code like this:

```
          string localArg0 = "{0} {1}";
          int localArg1 = x;
    localAwaiter = task.GetAwaiter();
    if (localAwaiter.IsCompleted)
    {
        goto SecondAwaitCompletion;
    }
  var localTuple = new Tuple<string, int>(
    localArg0, localArg1);
    stack = localTuple;
    state = 1;
    awaiter = localAwaiter;
    builder.AwaitUnsafeOnCompleted(ref awaiter, ref this);
    doFinallyBodies = false;
    return;
SecondAwaitContinuation:
   localTuple = (Tuple<string, int>) stack;
    localArg0 = localTuple.Item1; localArg1 = localTuple.Item2; stack = null;
    localAwaiter = awaiter;
    awaiter = default(TaskAwaiter<int>);
    state = -1;
SecondAwaitCompletion:
    int localArg2 = localAwaiter.GetResult();
    Console.WriteLine(localArg0, localArg1, localArg2); // Bold
```

The bold lines here are the ones involving elements of the logical stack.

At this point, we've probably gone as far as we need to—if you've made it this far successfully, you know more about the details of what's going on under the hood than 99 percent of developers are ever likely to. It's fine if

[2] As Eric Lippert is fond of saying, the stack is an implementation detail—some variables you might *expect* to go on the stack actually end up on the heap, and some variables may end up only existing in registers. For the purposes of this section, we're just talking about what logically happens on the stack.

[3] Admittedly there are times when the compiler could be smarter about the type of the variable, or avoid including one at all if it's never needed, but all of that may be added in a later version, as a further optimization.

you didn't quite follow everything first time through—if your experience is anything like mine has been when reading the code of these state machines, you'll want to wait a little while, and then come back to it.

## Finding out more

Want even more details? Crack out a decompiler. I'd urge you to use very small programs to investigate what the compiler does—it's very easy to get lost in a maze of twisty little continuations, all alike, if you write anything nontrivial. You may need to reduce the level of optimization the decompiler performs in order to get it to show you a fairly close-to-the-metal view of the code, rather than an interpretation. After all, a perfect decompiler would just reproduce your async functions, which would defeat the whole purpose of the exercise!

The code that the compiler generates can't always be decompiled into valid C#.

There's *always* the problem of it deliberately using unspeakable names for both variables and types, but more important, there are some cases where valid IL has no direct equivalent in C#. For example, in IL it's legitimate to branch to an instruction that's within a loop—after all, IL doesn't even *have* the concept of a loop, as such. In C#, you can't `goto` a label within a loop from outside the loop, so such an instruction can't be represented entirely correctly. Even the C# compiler can't have it all its own way: IL still has some restrictions on jump targets, so you'll often find the compiler has to go through a series of jumps to get to the right place.

Similarly, I've seen some decompilers get a little confused as to the exact ordering of assignment statements around the logical stack, occasionally moving the assignment of the temporary variables (`localArg0` and `localArg1`, for example) to the wrong side of the `IsCompleted` check. I believe this is due to the code not being quite like the normal output of the C# compiler. It's not too bad when you know what to look for, but it does mean that occasionally you'll probably end up dropping down to IL.

## Summary

I hope what we covered hasn't obscured the elegance of the asynchronous features of C# 5. The ability to write efficient asynchronous code in a more familiar execution model is a huge step forward, and I believe it will be transformative—once it's well understood. It's been my experience when giving presentations about async that many developers get easily confused by the feature the first time they see and use it.

**Here are some other Manning titles you might be interested in:**

Dependency Injection in .NET
Mark Seemann

Windows Phone 7 in Action
Timothy Binkley-Jones, Massimo Perga and Michael Sync

Real-World Functional Programming
Tomas Petricek

Last updated: 8/2/13