Unit Testing in F#

Unit testing is a code testing method designed to demonstrate the correctness of code at the *unit* level. A *unit* is in terms of whatever the smallest functional unit is within a given language or project. For example, in functional code, a unit is often thought of as a module, function, or primitive operation. In object-oriented code, a unit is usually a class and its methods. More generally, unit testing ensures that an abstract data type (which is an abstract data structure and associated operations) produces expected inputs and outputs.

It should be noted that unit testing is only one form of testing. Furthermore, test procedures—unless they exhaustively test all possible inputs—are not *sufficient* to ensure the correctness of a unit. Nevertheless, tests are one of the easiest ways to check that a program behaves as expected and tests are one of the most important steps toward correctness. Tests are especially useful in helping to ensure that the addition of new features to a codebase does not change the expected behavior. Consequently, test methods like unit testing are widely practiced in the software industry.

Running example

We will be revisiting the code we built together as a part of the parser combinator tutorial: the code that parses sentences into a list of words. If you don't remember what we did, please revisit the reading on Parser Combinators.

In this tutorial, I encourage you to follow along on your own machine.

MsTest

Microsoft .NET comes equipped with a unit test framework called MsTest. Since MsTest has F# language bindings, we can write MsTest unit tests natively in F#. The `dotnet new` command is capable of generating an F# test project, however, in order to make such a test project useful, it needs to be combined with an existing F# console or library project. In .NET, the facility for combining two projects together is an organizational feature called a *solution*.

.NET solutions

Let's start by generating a solution that will tie an F# library and unit test project together. Solutions are the standard way of combining projects in .NET, and as long as all projects can be compiled on the .NET platform, they can be combined. For example, a solution can be composed of F#, C#, and Visual Basic projects, along with test projects, and so on.

First, create a new directory to house your solution and `cd` into it.

```
$ mkdir test_tutorial
$ cd test_tutorial
```

Now type:

```
$ dotnet new sln
```

If the solution is created successfully, you will see:

```
The template "Solution File" was created successfully.
```

Next, let's create a very simple parser library. We will reuse the sentence parser code developed in the chapter on Parser Combinators.

```
$ mkdir SentenceParser
$ cd SentenceParser
$ dotnet new classlib -lang F#
The template "Class library" was created successfully.
```

First, download the Combinator.fs library[67]. Next, download the SentenceParser.fs library[68]. Note that these downloads are slightly different than example we worked through in Parser Combinators. I've added some extra information to both the `Success` and `Failure` types

[67] `https://williams-cs.github.io/cs334-f22-www/assets/code/Combinator.fs.txt`

[68] `https://williams-cs.github.io/cs334-f22-www/assets/starter/SentenceParser.fs.txt`

to help with debugging. After downloading, you should have at least the following files in your `SentenceParser` folder:

```
$ ls
Combinator.fs.    Library.fs    SentenceParser.fs    SentenceParser.fsproj
```

Delete the auto-generated `Library.fs` file.

```
$ rm Library.fs
```

Open the `SentenceParser.fsproj` file and add `Parsers.fs` and `SentenceParser.fs` as compile targets. Remove the `Library.fs` target. Your `SentenceParser.fsproj` should look like this:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="Combinator.fs" />
    <Compile Include="SentenceParser.fs" />
  </ItemGroup>

</Project>
```

Now, `cd` back into the parent directory and add the `SentenceParser` project to the solution.

```
$ cd ..
$ dotnet sln add SentenceParser/SentenceParser.fsproj
```

If the project is added successfully, you will see:

```
Project `SentenceParser/SentenceParser.fsproj` added to the solution.
```

We should now be able to build our solution.

```
$ dotnet build
```

Note that, in a solution, all `.fs` library files must be inside a `module` or a `namespace`. In the files supplied, the code in `Combinator.fs` is under the `Combinator` module, and the code in `SentenceParser.fs` is under the `SentenceParser` module. Go ahead, have a look. If you forget to

do this for your own project, you will see a compile-time message like:
`Files in libraries or multiple-file applications must begin with a namespace or module declaration.`

Creating the MsTest project

Now we can create the MsTest project and test our code. In the solution directory, create a new directory called `SentenceParserTests`, `cd` into it, and then use the `dotnet` tool to create an MsTest project.

```
$ mkdir SentenceParserTests
$ cd SentenceParserTests
$ dotnet new mstest -lang F#
```

If you did everything correctly, you should see:

```
The template "Unit Test Project" was created successfully.
```

We now need to make the `SentenceParser` a compile-time dependency of the `SentenceParserTests` project so that the test framework can call our library from test code.

```
$ dotnet add reference ../SentenceParser/SentenceParser.fsproj
Reference `..\SentenceParser\SentenceParser.fsproj` added to the project.
```

Finally, we need to `cd` back into our parent directory and add the `SentenceParserTests` project to the solution.

```
$ cd ..
$ dotnet sln add SentenceParserTests/SentenceParserTests.fsproj
Project `SentenceParserTests/SentenceParserTests.fsproj` added to the solution.
```

Again, running `dotnet build` should successfully build the entire project.

Understanding the test format

Let's open up the `SentenceParserTests/Tests.fs` file and have a look.

```
$ cat SentenceParserTests/Tests.fs
namespace SentenceParserTests

open System
open Microsoft.VisualStudio.TestTools.UnitTesting
```

```
[<TestClass>]
type TestClass () =

    [<TestMethod>]
    member this.TestMethodPassing () =
        Assert.IsTrue(true);
```

This file contains one test, called `TestMethodPassing`. Since `MsTest` was originally designed to test C#, tests utilize classes for organization.

Test suites. A collection of tests is called a *test suite*. Generally, a test suite is a set of tests designed to test *one unit*. For example, an entire suite might test different aspects of the same single algorithm. You might, for instance, write a test that checks for the common case for a sorting routine, another test that tests the corner case where the input is already sorted, and another test that tests another corner case where the input is empty (e.g., an empty list). All of these tests are packaged together in a *test class*, which houses the test suite. Test classes that house test suites must have the `[<TestClass>]` annotation as above.

Test methods. Each test is called a *test method*. In MsTest, each test method must literally be a method inside a test class. The test suite shown above has a single test called `TestMethodPassing`. There are two important facts to note about test methods. First, the method is prefixed with the `[<TestMethod>]` annotation. Second, test methods must be no-parens functions; or more precisely, they need to F# functions that take `unit`. The test above does nothing; it simply asserts `true`, which forces a test to pass.

Note that it is *up to you* how you want to organize your tests into test suites. Choose the organization that you find most useful. Out of laziness, I usually just put all the tests for an entire `module` inside a single test suite, and only break it into separate test suites once the test suite has grown to an unmanageable size. Remember, programming is an art, not a science!

Running the tests

If you are in the `SentenceParserTests` folder, you can run `dotnet test` and you should see output that looks a bit like this.

```
$ dotnet test
  ... some output omitted ...
Microsoft (R) Test Execution Command Line Tool Version 17.0.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed! - Failed: 0, Passed: 1, Skipped: 0, Total: 1, Duration: 15 ms
```

Above, you can see that the entire test suite consisting of a single test passed (`Passed!`) and took 15 milliseconds to run.

You can also run the `dotnet test` command from the parent directory which contains the solution. In that case, you will see test output from every project that actually contain tests.

Adding a new test

Finally, let's add a new test that actually tests our parser. In fact, let's get rid of the silly parser that always succeeds.

At the highest level, a hand-wavy description of our parser is that it takes a string representing a sentence and turns it into a list of words. The purpose of a test is to ensure that such hand-wavy descriptions are backed up with real code that does what you say and is checked every time you run the test suite. A nice side-effect of such tests is that they serve to document use cases for your code.

First, add an `open` statement to the top of your `SentenceParserTests/Tests.fs` file so that it can access your `SentenceParser` library.

```
open SentenceParser
```

Next, replace the test `TestClass` with a new one. Here is the complete code for the `Tests.fs` file:

```
namespace SentenceParserTests

open System
open Microsoft.VisualStudio.TestTools.UnitTesting
open SentenceParser

[<TestClass>]
type TestClass () =

    [<TestMethod>]
    member this.ValidSentenceReturnsAWordList() =
        let input = "The quick brown fox jumped over the lazy dog."
        let expected = [ "The"; "quick"; "brown"; "fox"; "jumped"; "over"; "the"; "lazy"; "dog" ]
        let result = parse input
        match result with
        | Some ws ->
            Assert.AreEqual(expected, ws)
        | None ->
            Assert.IsTrue false
```

The logic is as follows. We supply an input called `input`, which is a sentence. We also supply an `expected` value, which is the output we *expect* `parse` to produce when given the `input`. Next, we call `parse` with `input` and store it in `result`. Since `parse` returns an `option` type (`Some` if the parser succeeds, `None` if it does not), we pattern-match on `result`. Finally,

1. if we get back `Some` word list `ws`, we check that `ws` is exactly the same as the word list `expected`. Note the position of the `expected` parameter. While `Assert.AreEqual` will fail anytime its two arguments differ, when it fails, it returns a helpful message based on the contents of the `expected` parameter. Otherwise,
2. if we get back `None`, then the parse failed when it should have succeeded. In this case, we force the test to fail by supplying `Assert.IsTrue false`.

Running `dotnet test` reports:

```
$ dotnet test
  ... some output omitted ...
Microsoft (R) Test Execution Command Line Tool Version 17.0.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed! - Failed: 0, Passed: 1, Skipped: 0, Total: 1, Duration: 86 ms
```

So far so good.

Test-driven development

One of the many buzzwords you may hear out in industry is something called "test-driven development" or TDD. The idea behind TDD is to write tests *before* you write your implementation code. While there are many fads in software development, I believe that this is genuinely a good idea. For starters, providing an example of input and output often focuses your implementation efforts. Second, input and output examples fit nicely with functional programming, since, if you're doing it correctly, every function should be *pure* and every input should unambiguously produce the same output every time. [69]

[69] For deterministic functions.

Let's add a test for a feature we do not yet have: the ability to parse questions.

```
[<TestMethod>]
member this.ValidQuestionReturnsAWordList() =
    let input = "Does the quick brown fox jump over the lazy dog?"
    let expected = [ "Does"; "the"; "quick"; "brown"; "fox"; "jump"; "over"; "the"; "lazy"; "dog" ]
    let result = parse input
    match result with
    | Some warr ->
        Assert.AreEqual(expected, warr)
    | None ->
        Assert.IsTrue false
```

Running `dotnet test` produces our first failing test, because we do not yet support this feature.

```
$ dotnet test
  ... some output omitted ...
Microsoft (R) Test Execution Command Line Tool Version 17.0.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
```

```
A total of 1 test files matched the specified pattern.
  Failed ValidQuestionReturnsAWordList [35 ms]
  Error Message:
   Assert.IsTrue failed.
  Stack Trace:
     at SentenceParserTests.TestClass.ValidQuestionReturnsAWordList() in [...] Tests.fs:line 30

  Standard Error Messages:
 [attempting: grammar on "Does the quick brown fox jump over the lazy dog?", next char: 0x44]
 [attempting: sentence on "Does the quick brown fox jump over the lazy dog?", next char: 0x44]
 [attempting: sprefix on "Does the quick brown fox jump over the lazy dog?", next char: 0x44]
 [attempting: upperword on "Does the quick brown fox jump over the lazy dog?", next char: 0x44]
 [success: upperword, consumed: "Does", remaining: " the quick brown fox jump over the lazy dog?", next char: 0x20]
 [attempting: words0 on " the quick brown fox jump over the lazy dog?", next char: 0x44]
 [attempting: word on "the quick brown fox jump over the lazy dog?", next char: 0x44]
 [success: word, consumed: "the", remaining: " quick brown fox jump over the lazy dog?", next char: 0x20]
 [attempting: word on "quick brown fox jump over the lazy dog?", next char: 0x44]
 [success: word, consumed: "quick", remaining: " brown fox jump over the lazy dog?", next char: 0x20]
 [attempting: word on "brown fox jump over the lazy dog?", next char: 0x44]
 [success: word, consumed: "brown", remaining: " fox jump over the lazy dog?", next char: 0x20]
 [attempting: word on "fox jump over the lazy dog?", next char: 0x44]
 [success: word, consumed: "fox", remaining: " jump over the lazy dog?", next char: 0x20]
 [attempting: word on "jump over the lazy dog?", next char: 0x44]
 [success: word, consumed: "jump", remaining: " over the lazy dog?", next char: 0x20]
 [attempting: word on "over the lazy dog?", next char: 0x44]
 [success: word, consumed: "over", remaining: " the lazy dog?", next char: 0x20]
 [attempting: word on "the lazy dog?", next char: 0x44]
 [success: word, consumed: "the", remaining: " lazy dog?", next char: 0x20]
 [attempting: word on "lazy dog?", next char: 0x44]
 [success: word, consumed: "lazy", remaining: " dog?", next char: 0x20]
 [attempting: word on "dog?", next char: 0x44]
 [success: word, consumed: "dog", remaining: "?", next char: 0x3f]
 [success: words0, consumed: " the quick brown fox jump over the lazy dog", remaining: "?", next char: 0x3f]
 [success: sprefix, consumed: "Does the quick brown fox jump over the lazy dog", remaining: "?", next char: 0x3f]
 [attempting: period on "?", next char: 0x44]
 [failure at pos 48 in rule [pchar '.']: period, remaining input: "", next char: EOF]
 [failure at pos 48 in rule [pchar '.']: sentence, remaining input: "", next char: EOF]
 [failure at pos 48 in rule [pchar '.']: grammar, remaining input: "", next char: EOF]


Failed! - Failed: 1, Passed: 1, Skipped: 0, Total: 2, Duration: 116 ms
```

This output says that the `ValidQuestionReturnsAWordList` test failed. It failed, of course, because we have not yet implemented this feature. Observe that, since we used the `debug` feature, the failing test printed out what the program echoed. Tests *only print output when they fail*.

Let's implement the feature.

A Question Parser

I am not going to belabor parsers again here, so let's fast-forward to the most intuitive feature addition. First, add a `qmark` parser.

```
let qmark = (pchar '?') <!> "question mark"
```

Next, modify the `sentence` parser so that it accepts either a period or a question mark.

```
let sentence = pleft prefix (period <|> qmark) <!> "sentence"
```

The complete, modified code is as follows:

```
module SentenceParser

open Combinator

let qmark = (pchar '?') <!> "question mark"
let period = (pchar '.') <!> "period"
let word = pfun (pmany1 pletter) (fun cs -> stringify cs) <!> "word"
let upperword = pseq pupper (pmany0 pletter) (fun (x,xs) -> stringify (x::xs)) <!> "upperword"
let words0 = pmany0 (pright pws1 word) <!> "words0"
let prefix = pseq upperword words0 (fun (w,ws) -> w::ws) <!> "sprefix"
let sentence = pleft prefix (period <|> qmark) <!> "sentence"
let grammar = pleft sentence peof <!> "grammar"

let parse input : string list option =
    match grammar (prepare input) with
    | Success(ws,_) -> Some ws
    | Failure(_,_) -> None
```

Let's test it again.

```
$ dotnet test
  ... some output omitted ...
Microsoft (R) Test Execution Command Line Tool Version 17.0.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed! - Failed: 0, Passed: 2, Skipped: 0, Total: 2, Duration: 79 ms
```

Looks good!

Conclusion

In this tutorial, we learned:

- How to create a solution.
- How to add a project to a solution.
- How to add a test project to a solution.
- How to add a test.
- How to run tests.
- How to do test-driven development, where tests are written before implementation code.

I encourage you to add tests to your own projects. This means that you will probably need to ”wrap” your existing projects in a solution, but the above tutorial should be enough of a guide to get you started.

There are many additional `Assert` methods beside the `AreEqual` method. For additional information, see the documentation on the MsTest `Assert` class[70]. After clicking on the link, look for the ”Methods” dropdown in the left column.

Finally, if you want another tutorial, have a look at Microsoft’s official F# unit test tutorial[71] which goes into more detail than this tutorial.

[70] `https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert?view=mstest-net-1.2.0`

[71] `https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-fsharp-with-mstest`