|  |
| --- |
|  |
| PIT – F# to JS Compiler |
| UG Documentation |

|  |
| --- |
| Fahad  11/14/2011 |

# Introduction

F#, a functional-first language in the .NET platform is a very nice language that includes the benefits of functional programming + OO. We decided to leverage F# as a language that would target JavaScript in a way to help Functional Programmers / .NET users. This helps in writing code that is concise, succinct, and easy to manage. Additionally, we have added some wrapping code that helps you do real-time debugging with the F# code and is easy when you are deploying the generated JavaScript.

## What does Pit solve?

PIT is F# to JavaScript compiler that leverages the beauty of F# and also JavaScript. It helps the Developer to quickly code, debug and manage from small to very large libraries easily. We have also kept the motto of the F# language in keeping the generated JavaScript very simple, so it could be directly used in JavaScript itself. The core idea of Pit is to leverage the F# language simplicity and also being powerful in representing core ideas. Most of the F# features are supported and thus would allow a developer to write maximum efficient code (that is functional too).

Pit Framework fits for writing any JavaScript based Web apps or Web sites (HTML5 based games etc., etc.). More like a glue between your F# program and the Web-browser. It is not a full end-to-end framework. On the other hand, F# language can be used in server-side areas, With Pit it becomes complete with pure client side programming in F# itself!

## JavaScript

Pit generates clean JavaScript that is readable and can be directly used from JavaScript itself. The F# features explained in the next section will have details of each transformation that it does with the F# code. It is very important to understand that Pit should be used to write F# code for the Web and not generally as any .NET application. The Framework has all the basic tooling for core JavaScript available in most browsers.

*Note: There is no JS Lint integration done yet, we don’t intend to do it now. But the code is very readable and also has lexical scoping for most objects.*

Contents

[Introduction 1](#_Toc308605872)

[What Does Pit really Solve? 1](#_Toc308605873)

[Javascript 1](#_Toc308605874)

[Installation 3](#_Toc308605875)

[Minimum requirements 3](#_Toc308605876)

[Machine Settings 3](#_Toc308605877)

[Working with Pit 4](#_Toc308605878)

[Getting Started 4](#_Toc308605879)

[To Create a New Pit project with Web Application 4](#_Toc308605880)

[To Create a Pit Project with existing Web Application 7](#_Toc308605881)

[Pit Attributes Syntax 8](#_Toc308605882)

[Pit Terms 9](#_Toc308605883)

[F# Language Integration 9](#_Toc308605884)

[Supported Features 9](#_Toc308605885)

[Javascript Types 31](#_Toc308605886)

[Pit Namespace Diagram 35](#_Toc308605887)

[FAQ 38](#_Toc308605888)

[Match expression with same alias objects 38](#_Toc308605889)

[Match expressions as in .NET types 38](#_Toc308605890)

[Member functions with inner let functions accessing global variables 39](#_Toc308605891)

# Installation

Pit setup installs Compiler, Libraries, Debug libraries, Silverlight runtime, JavaScript helper files and New Project templates for VS 2010.



**Pit Installer**

*Note: In 64 bit version machines Pit is installed in Program files x86*

## Minimum requirements

1. Visual Studio 2010 (Professional, Premium, Ultimate)
2. Silverlight 4 tools for Visual Studio 2010

***Note: Silverlight is required for debugging mode.***

## Machine Settings

* New Environment Variable named “PitLocation” is added to User level environment variables, required for Pit compiler to pick assembly locations
* Installation is available only for current user

## Mac OSX with MonoDevelop

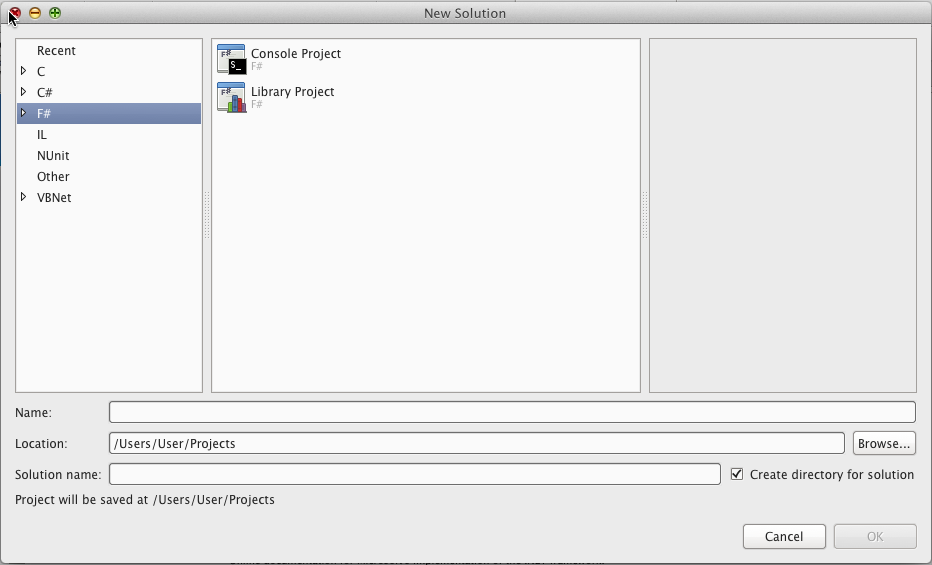
Pit works on Mac OSX with mono. Below are the install instructions to execute on a Mac with MonoDevelop plugins enabled. One important note here is, we are not bringing the “Debug” mode for Mac users, it will be strictly only used for JavaScript code generation with F#.

* Install Mono + MonoDevelop in Mac ([link](http://monodevelop.com/Download)).
* Install fsharpbinding addin for monodevelop from the following location
  + Download the forked FSharpbinding code from this location <https://github.com/karthikvishnu/fsharpbinding> which refers the latest version binaries(MonoDevelop – 2.8.4)
* To compile the plugin open a terminal window and use the following commands

*./configure.sh*

*sudo make*

*sudo make install*



**F# Addin**

* Install Pit, Pit addin to add pit library project templates to MonoDevelop . You can download required files for Mac Lion OS X from Pit [site](http://pitfw.posterous.com).

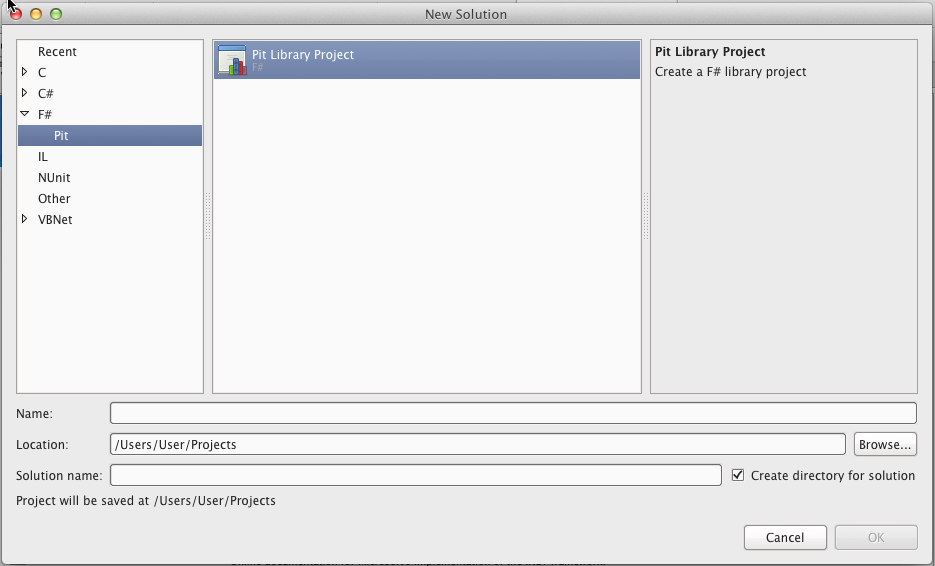
1. To install the libraries and addin use the following commands

*sudo make*

*sudo make install-fw*

*sudo make install-addin*

*./PitReg.sh*



**Pit addin**

*Note: Log off and log in to update the new environment variables*

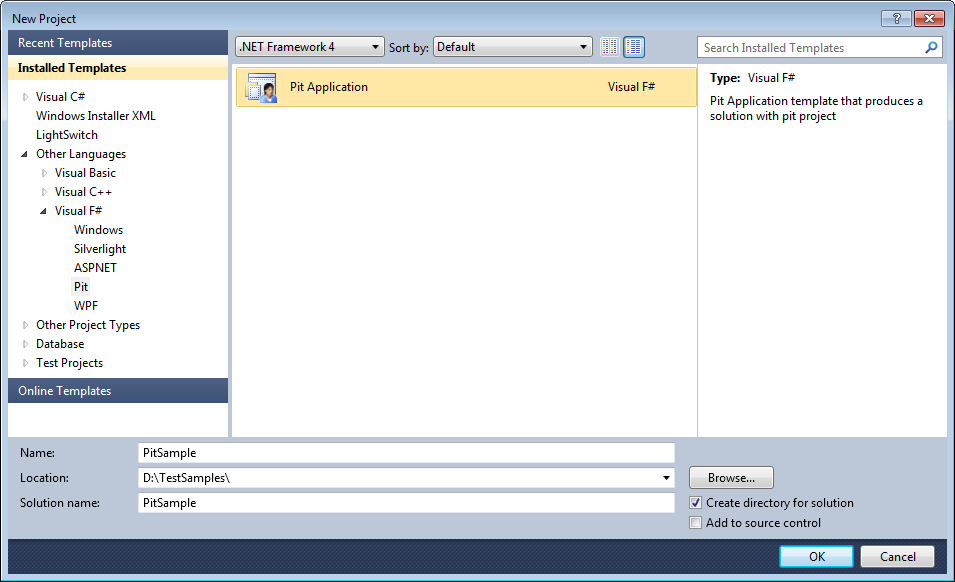
# Working with Pit

## Getting Started

Pit application can be created with the help of installed Visual Studio Pit project templates.

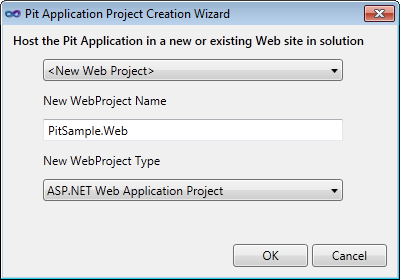
### Create New Pit Web Application

* On the Windows **Start** menu, click **All Programs** > **Microsoft Visual Studio 2010** > **Microsoft Visual Studio**.
* On the **File** menu, click **New Project**. The New project dialog opens
* In **Installed Templates** list, expand Visual F# node
* In **Pit** node, select **Pit Application** template and enter desired name for project

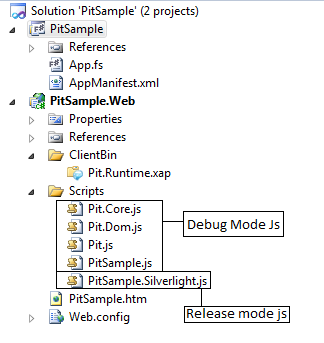


**New Project Window**

* Click Ok to display Pit project application creation wizard



* Choose Web project name and Web project type
* Click Ok to create Pit application



**Created project files**

* Added Html page contains following code

|  |
| --- |
| <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  <html xmlns="http://www.w3.org/1999/xhtml">  <head>      <title>PitSample</title>      <!--Debug mode-->      <!--Comment following lines in release mode-->      <script src="Scripts/PitSample.Silverlight.js" type="text/javascript"></script>        <!--Release mode-->      <!--Uncomment following lines in release mode-->      <!--<script src="Scripts/Pit.js" type="text/javascript"></script>      <script src="Scripts/Pit.Core.js" type="text/javascript"></script>      <script src="Scripts/Pit.Dom.js" type="text/javascript"></script>      <script src="Scripts/PitSample.js" type="text/javascript"></script>-->  </head>  <body>  </body>  </html> |

**Debug mode html page**

* Select “Debug” in Configuration manager to launch the page in debug mode
* In “Debug” mode debugging is enabled for fsharp code



**Debug sample**

* Select “Release” in configuration manager and uncomment release mode scripts to launch the page in release mode. Ensure to comment “Debug” mode scripts

|  |
| --- |
| <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  <html xmlns="http://www.w3.org/1999/xhtml">  <head>      <title>PitSample</title>      <!--Debug mode-->      <!--Comment following lines in release mode-->      <!--<script src="Scripts/PitSample.Silverlight.js" type="text/javascript"></script>-->        <!--Release mode-->      <!--Uncomment following lines in release mode-->      <script src="Scripts/Pit.js" type="text/javascript"></script>      <script src="Scripts/Pit.Core.js" type="text/javascript"></script>      <script src="Scripts/Pit.Dom.js" type="text/javascript"></script>      <script src="Scripts/PitSample.js" type="text/javascript"></script>  </head>  <body>  </body>  </html> |

**Release mode js**

* Project built in “Release” mode all converts FSharp code in “PitSample” to equivalent JavaScript code with Pit Compiler



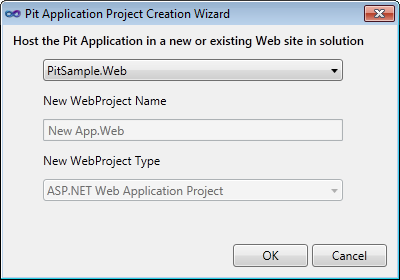
**Fsharp code**



**Equivalent JavaScript code**

### Create Pit Project with existing web application

* Click Add New project
* Expand Pit Node, select installed Pit application template with a new project name
* Click OK to launch Pit project creation wizard

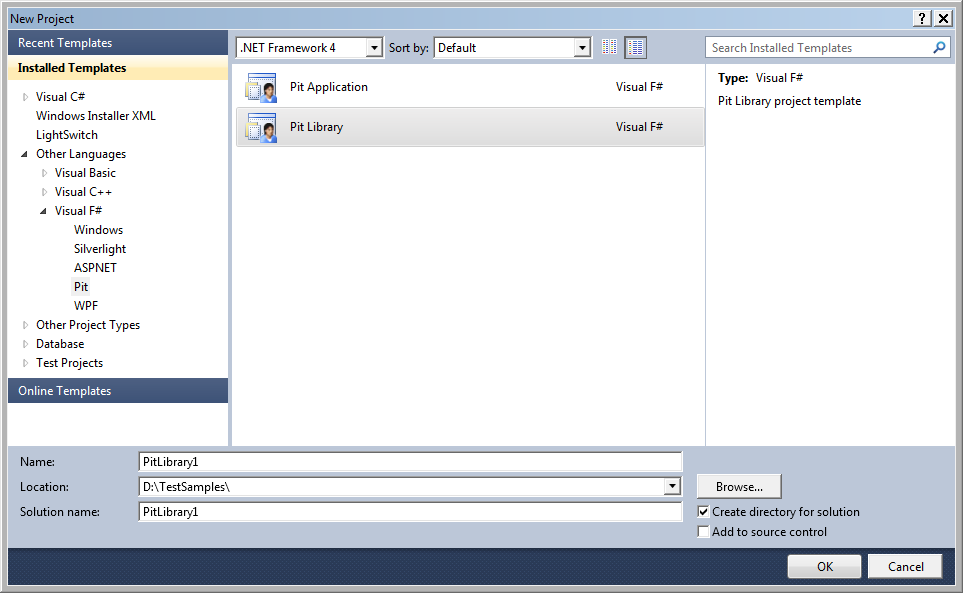


**Pit App creation Wizard**

* Select any existing web project in drop down and create a new Pit application project

### Create Pit Library Project

* On the Windows **Start** menu, click **All Programs** > **Microsoft Visual Studio 2010** > **Microsoft Visual Studio**.
* On the **File** menu, click **New Project**. The New project dialog opens
* In **Installed Templates** list, expand Visual F# node
* In **Pit** node, select **Pit Library** template and enter desired name for project



* Generated js can be found in Bin location for the project or if reference to a webproject js file is copied to Scripts folder.

## Pit Attributes Syntax

Pit depends on F# quotation expressions for generating JS. It uses .NET attributes to mark a function with some annotation. Below is a simple example of an F# function that is annotated,

*F#*

|  |
| --- |
| [<Js>] // type alias for ReflectedDefinitionAttribute  let f x = x + 1 |

Likewise all methods, members, constructors should be marked with this attribute, More details of the usage would be explained in the below sections in [F# Language Integration](#_Supported_Features).

Below are the listed Pit Attributes that allows you to generate / alter the JS code generation,

|  |  |
| --- | --- |
| **Attribute Name** | **Usage** |
| Js | Type Alias for ReflectedDefinitionAttribute. Used to declare a function / member/ constructor for JS code generation. |
| AliasAttribute | Alias name for a type. Usually used to name a different variable than the declared variable. This is used when JS is generated for the type. |
| CompileToAttribute | Used in member functions which are wrappers for JavaScript methods. Ex:  [<CompileTo(“getElementById”)>]  member x.GetElementById(…) |
| JsObjectAttribute | Used to generate Javascript-Objects for F# records.  Ex:  [<JsObject>]  type t = {  id : int  name : string  } |
| JsOverloadMember | Alias to CompileTo attribute, used for marking overloaded members with different names to generate in JavaScript. |

Note: no need to suffix “Attribute” when using in F#, as it is a normal .NET attribute and follows the same rules.

## Pit Terms

**Debug Mode**

Pit provides support for F# debugging using the Silverlight runtime. The Silverlight runtime is used as a simulation for debugging code using the Silverlight-JavaScript Bridge. Pit implements all HTML DOM and HTML5 DOM code in the libraries and provide full inter-op between the “Debug” and “Release” modes.

**Release Mode**

Pit generates the equivalent JavaScript for all the F# code that gets compiled into an assembly marked with attributes (ReflectedDefinitionAttribute). The generated code works seamlessly between in debugging (Silverlight) as well as in JavaScript, thus enabling very fast development tooling built right into VS 2010 for developers.

## F# Language Integration

F# Language provides a way to compile the F# code as quotations, and we take these quotations to compile it to JavaScript.

### Supported Features

Below are the list of F# features that is currently supported,

|  |  |
| --- | --- |
| Top-Level Features | Yes/No |
| Functions | Yes |
| Values | Yes |
| Tuples | Yes |
| List | Yes |
| Option | Yes |
| Sequence | Yes |
| Array | Yes |
| Generics \* | Yes |
| Records | Yes |
| Discriminated Unions | Yes |
| Enumerations | Yes |
| Reference Cell | Yes |
| Classes | Yes |
| Structures \*\* | No |
| Inheritance \* | No |
| Abstract \*\* | No |
| Members | Yes |
| Type Extensions | Yes |
| Operator Overloading | Yes |
| Flexible Types | Yes |
| Delegates | Yes |
| Object Expressions \*\* | No |
| Casting and Conversions | Yes |
| Access Control | Yes |
| Loops: for..to | Yes |
| Loops: for..in \* | Yes |
| Assertions \*\* | No |
| Exception Handling | Yes |
| Attributes \*\* | No |
| use Keyword | Yes |
| Namespaces | Yes |
| Modules | Yes |
| Units of Measure | Yes |
| Lazy Computation | Yes |
| Computation Expression | Yes |
| Async \*\* | No |
| Code Quotations \*\* | No |

*Note: Marked fields are features that are still not added to PIT, this table will be consistently revised in our future versions.*

*\*\*: Features that are not supported either by F# quotations or not really required in JavaScript-domain*

*\*: Features that are not fully supported by F# quotations or partially implemented (ex: Inheritance)*

The sections would be explained with a brief introduction from F# MSDN reference link (as most are the same F# features), and it will also have a generated JavaScript reference code.

#### Namespaces

A namespace lets you organize code into areas of related functionality by enabling you to attach a name to a grouping of program elements. Pit uses a namespaces collection to register globally. Below is an example of namespaces getting generated in JavaScript.

**F#**

|  |
| --- |
| namespace Widgets  module WidgetModule =  … |

***JS***

|  |
| --- |
| registerNamespace("Widgets.WidgetModule"); |

*Note: for more information check this MSDN* [*link*](http://msdn.microsoft.com/en-us/library/dd233219.aspx)*.*

#### Modules

In the context of the F# language, a *module* is a grouping of F# code, such as values, types, and function values, in an F# program. Grouping code in modules helps keep related code together and helps avoid name conflicts in your program.

*Note: for more information check this MSDN* [*link*](http://msdn.microsoft.com/en-us/library/dd233221.aspx)*.*

#### Literals

JavaScript literals are very limited, only the following is supported,

* Integer
* Float
* String
* Bool

All other F# literals found [here](http://msdn.microsoft.com/en-us/library/dd233193.aspx) are not supported.

#### Functions (F#)

 F# function has a name, can have parameters and take arguments, and has a body. F# also supports functional programming constructs such as treating functions as values, using unnamed functions in expressions, composition of functions to form new functions, curried functions, and the implicit definition of functions by way of the partial application of function arguments.

|  |
| --- |
| // Non-recursive function definition.  let [inline] function-name parameter-list [ : return-type ] = function-body  // Recursive function definition.  let rec function-name parameter-list = recursive-function-body |

A simple function definition resembles the following:

***F#***

|  |
| --- |
| [<Js>]  let f x = x + 1 |

***JS***

|  |
| --- |
| **var** f = **function**(x) {  **return** (x + 1);  }; |

Function with parameters,

***F#***

|  |
| --- |
| [<Js>]  let cylinderVolume radius length =  // Define a local value pi.  let pi = 3.14159  length \* pi \* radius \* radius  [<Js>]  let vol = cylinderVolume 2. 3. |

***JS***

|  |
| --- |
| var cylinderVolume = function(radius) {  return function(length) {  var pi = 3.14159;  return (((length \* pi) \* radius) \* radius);  };  };  var vol = cylinderVolume(2)(3); |

*Note: For more details visit this MSDN* [*link*](http://msdn.microsoft.com/en-us/library/dd233229.aspx)*.*

#### Values (F#)

Values in F# are quantities that have a specific type; values can be integral or floating point numbers.

**Binding a Value**

The term *binding* means associating a name with a definition.

***F#***

|  |
| --- |
| let a = 1  let str = "text"  // A function value binding.  let f x = x + 1 |

**Immutability**

Immutable values are values that cannot be changed throughout the course of a program's execution. In pure functional languages, there are no variables, and functions behave strictly as mathematical functions. F# is not a pure functional language, yet it fully supports functional programming. Using immutable values is a good practice because doing this allows your code to benefit from an important aspect of functional programming.

**Mutable Variables**

You can use the keyword mutable to specify a variable that can be changed.

***F#***

|  |
| --- |
| [<Js>]  let mutable x = 1  x <- x + 1 |

Note: More details can be found in this MSDN [link](http://msdn.microsoft.com/en-us/library/dd233185.aspx).

#### Generics

F# function values, methods, properties, and aggregate types such as classes, records, and discriminated unions can be *generic*. Generic constructs contain at least one type parameter, which is usually supplied by the user of the generic construct. Generic functions and types enable you to write code that works with a variety of types without repeating the code for each type. Making your code generic can be simple in F#, because often your code is implicitly inferred to be generic by the compiler's type inference and automatic generalization mechanisms. Pit would translate the typed F# code to untyped Javascript. If all the libraries are used within F# (Pit) then the types will adhere just like a normal F# code.

One more important point when using generics in Pit is that inner generic functions (functions inside another function) cannot be generic because of the limitation in F# quotations.

Note: for more details on this topic check this MSDN [link](http://msdn.microsoft.com/en-us/library/dd233215.aspx).

#### Loops: for..to

The for...to expression is used to iterate in a loop over a range of values of a loop variable.

***F#***

|  |
| --- |
| // A simple for...to loop.  [<Js>]  let function1() =  let mutable x = 0  for i = 1 to 10 do  x <- x + i  x |

***JS***

|  |
| --- |
| var function1 = function() {  var x = 0;  for (var i = 1; i <= 10; i++) {  x = (x + i);  };  return x;  }; |

***Note: F# quotations doesn’t support for..downto, and so Pit also doesn’t support it.***

Note: for more information check this MSDN [link](http://msdn.microsoft.com/en-us/library/dd233236.aspx).

#### Loops: for..in

This looping construct is used to iterate over the matches of a pattern in an enumerable collection such as a range expression, sequence, list, array, or other construct that supports enumeration.

|  |
| --- |
| for pattern in enumerable-expression do  body-expression |

The following code examples illustrate the use of for...in expression.

***F#***

|  |
| --- |
| let seq1 = seq { for i in 1 .. 10 -> (i, i\*i) }  for (a, asqr) in seq1 do  //dosomething |

***JS***

|  |
| --- |
| var seq1 = Pit.FSharp.Core.Operators.CreateSequence(Pit.FSharp.Collections.SeqModule.Delay(function() {  return Pit.FSharp.Collections.SeqModule.Map(function(i) {  return {  Item1: i,  Item2: (i \* i)  };  })(Pit.FSharp.Core.Operators.op\_Range(1)(10));  }));  var enumerator = seq1.IEnumerable1\_GetEnumerator();  (function(thisObject) {  try {  while (enumerator.IEnumerator\_MoveNext()) {  var forLoopVar = enumerator.IEnumerator1\_get\_Current();  var asqr = forLoopVar.Item2;  var a = forLoopVar.Item1;  // do something  }  } finally {  (function(thisObject) {  if (enumerator.containsInterface("\_\_getIDisposable")) {  return enumerator.IDisposable\_Dispose();  } else {  return null;  }  })(thisObject)  }  })(this); |

Note: for more information check this MSDN [link](http://msdn.microsoft.com/en-us/library/dd233227.aspx).

#### Loops: while..do

The while...do expression is used to perform iterative execution (looping) while a specified test condition is true.

|  |
| --- |
| while test-expression do  body-expression |

The following example illustrates the use of the while...do expression.

***F#***

|  |
| --- |
| let mutable x = 0  while x < 10 do  x <- x + 1 |

***JS***

|  |
| --- |
| var x = 0;  while (x < 10) {  x = (x + 1);  }; |

Note: for more information check this MSDN [link](http://msdn.microsoft.com/en-us/library/dd233208.aspx).

#### Access Control

*Access control* refers to declaring which clients can use certain program elements, such as types, methods, and functions. All F# features of access control can be enforced.

*Note: Pit only generates pure JavaScript functions, so access control is not valid inside of JavaScript.*

*Note: for more information on this topic refer to this MSDN* [*link*](http://msdn.microsoft.com/en-us/library/dd233188.aspx)*.*

#### Classes

*Classes* are types that represent objects that can have properties, methods, and events.

|  |
| --- |
| type [access-modifier] type-name [type-params] [access-modifier] ( parameter-list ) [ as identifier ] =  [ class ]  [ inherit base-type-name(base-constructor-args) ]  [ let-bindings ]  [ do-bindings ]  member-list  ...  [ end ]  // Mutually recursive class definitions:  type [access-modifier] type-name1 ...  and [access-modifier] type-name2 ... |

Pit generates a single function for the whole class type with lexical scoping to the prototype functions in Javascript. Below is an example F# class type,

***F#***

|  |
| --- |
| type Employee [<Js>](id: int, name: string) =  let mutable salary = 0  [<Js>]  member x.ID = id // property with get  [<Js>]  member x.Name = name // property with get  [<Js>]  member x.Salary // property with get/set  with get() = salary  and set(v: int) = salary <- v  [<Js>]  member x.GetName() = name // method declaration  [<Js>]  static member GetAnnualSalary(e: Employee) =  e.Salary \* 12  let e = new Employee(0, "Micheal")  let name = e.GetName()  let salary = e.Salary  let annualAmt = Employee.GetAnnualSalary(e) |

***JS***

|  |
| --- |
| Employee = (function() {  function Employee(id, name) {  this.id = id;  this.name = name;  this.salary = 0;  }  Employee.prototype.GetName = function() {  return this.name;  };  Employee.prototype.set\_Salary = function(x) {  return function(v) {  return this.salary = v;  };  };  Employee.prototype.get\_ID = function() {  return this.id;  };  Employee.prototype.get\_Name = function() {  return this.name;  };  Employee.prototype.get\_Salary = function() {  return this.salary;  };  return Employee;  })();  Employee.GetAnnualSalary = function(e) {  return (e.get\_Salary() \* 12);  };  var e = new Employee(0, "Micheal");  var name = e.GetName();  var salary = e.get\_Salary();  var annualAmt = Employee.GetAnnualSalary(e); |

Static functions are direct functions in JavaScript, so they are not part of the prototype extensions in the generated JS code.

*Note: Pit doesn’t support “Inheritance” or “Abstract” classes yet, so for heterogeneous types use* [*Discriminated union types*](#_Discriminated_Unions)*.*

Note: for more information on this topic refer to this MSDN [link](http://msdn.microsoft.com/en-us/library/dd233205.aspx).

#### Enumerations

*Enumerations*, also known as *enums*, are integral types where labels are assigned to a subset of the values. You can use them in place of literals to make code more readable and maintainable.

|  |
| --- |
| type enum-name =  | value1 = integer-literal1  | value2 = integer-literal2  ... |

The following code shows the declaration and use of an enumeration.

***F#***

|  |
| --- |
| type Color =  | Red = 0  | Green = 1  | Blue = 2  // Use of an enumeration.  [<Js>]  let col1 : Color = Color.Red |

***JS***

|  |
| --- |
| var Color = {  Red: {},  Green: {},  Blue: {}  };  var col1 = Program.Color.Red; |

Note: for more details check this MSDN [link](http://msdn.microsoft.com/en-us/library/dd233216.aspx).

#### Type Extensions

Type extensions let you add new members to a previously defined object type.

|  |
| --- |
| // Intrinsic extension.  type typename with  member self-identifier.member-name =  body  ...  [ end ]  // Optional extension.  type typename with  member self-identifier.member-name =  body  ...  [ end ] |

There are two forms of type extensions that have slightly different syntax and behavior. An *intrinsic extension* is an extension that appears in the same namespace or module, in the same source file, and in the same assembly (DLL or executable file) as the type being extended. An *optional extension* is an extension that appears outside the original module, namespace, or assembly of the type being extended.

***F#***

|  |
| --- |
| module MyModule1 =  // Define a type.  type MyClass() =  [<Js>]  member this.F() = 100  // Define type extension.  type MyClass with  [<Js>]  member this.G() = 200  module MyModule2 =  [<Js>]  let function1 (obj1: MyModule1.MyClass) =  // Call an ordinary method.  let r1 = obj1.F()  // Call the extension method.  let r2 = obj1.G()  r1 + r2 |

***JS***

|  |
| --- |
| MyModule1.MyClass = (function() {  function MyClass() {}  MyClass.prototype.F = function() {  return 100;  };  MyClass.prototype.G = function() {  return 200;  };  return MyClass;  })();  MyModule2.function1 = function(obj1) {  var r1 = obj1.F();  var r2 = obj1.G();  return (r1 + r2);  }; |

Note: for more information on this topic check this MSDN [link](http://msdn.microsoft.com/en-us/library/dd233211.aspx).

#### Delegates

A delegate represents a function call as an object. In F#, you ordinarily should use function values to represent functions as first-class values; however, delegates are used in the .NET Framework and so are needed when you interoperate with APIs that expect them. They may also be used when authoring libraries designed for use from other .NET Framework languages.

|  |
| --- |
| type delegate-typename = delegate of type1 -> type2 |

In Javascript delegates are just function lambdas, so Pit just compiles the delegate types as functions without generating any type declaration JS code. Below is an example of using a delegate,

***F#***

|  |
| --- |
| type Delegate1 = delegate of (int \* int) -> int  type Delegate2 = delegate of int \* int -> int  [<Js>]  let f1 = new Delegate1(fun (a,b) -> a + b)  let r = f1.Invoke((1,2)) |

***JS***

|  |
| --- |
| var f1 = function(tupledArg) {  var a = tupledArg.Item1;  var b = tupledArg.Item2;  return (a + b);  };  var r = f1({  Item1: 1,  Item2: 2  });  var f1 = function(a, b) {  return (a + b);  };  var r = f1(1, 2); |

Pit doesn’t generate JS for delegate types; the resulting usage of the delegate will generate a lambda function in JavaScript.

*Note: for more information on this topic check this MSDN* [*link*](http://msdn.microsoft.com/en-us/library/dd233206.aspx)*.*

#### Operator Overloading

The following code defines an operator +?.

***F#***

|  |
| --- |
| [<Js>]  let (+?) (x: int) (y: int) = x + 2\*y |

***JS***

|  |
| --- |
| op\_PlusQmark = function(x) {  return function(y) {  return (x + (2 \* y));  };  };  var res = op\_PlusQmark(10)(1); |

Types/Union cases can also be extended with custom operators,

***F#***

|  |
| --- |
| type t = {  x: int  y: int  } with  [<Js>]  static member (+) (t1:t, t2:t) =  { x = t1.x + t2.x; y = t1.y + t2.y }  let t1 = {x=10; y=10;}  let t2 = {x=20; y=20;}  let t3 = t1 + t2 |

***JS***

|  |
| --- |
| t = (function() {  function t(x, y) {  this.x = x;  this.y = y;  }  t.prototype.get\_x = function() {  return this.x;  };  t.prototype.get\_y = function() {  return this.y;  };  return t;  })();  t.op\_Addition = function(tupledArg) {  var t1 = tupledArg.Item1;  var t2 = tupledArg.Item2;  return new Pit.Test.OperatorOverloadTests.t((t1.get\_x() + t2.get\_x()), (t1.get\_y() + t2.get\_y()));  };  var t1 = new t(10, 10);  var t2 = new t(20, 20);  var t3 = t.op\_Addition({  Item1: t1,  Item2: t2  }); |

Note: There might be situations in which instead of “Tupled args” it may require function parameters to be passed normally, Use JsIgnore(IgnoreTuple=true).

#### Exception Handling

Exception handling is the standard way of handling error conditions in the .NET Framework. Javascript also supports exception handling with try/catch/finally blocks.

**Try..with exceptions**

Pit supports exceptions of the below types,

***F#***

|  |
| --- |
| exception Error1 of string  exception Error2 of string \* int  [<Js>]  let TryWith1() =  let function1 (x:int) (y:int) =  try  if x = y then raise (Error1("x"))  else raise (Error2("x", 10))  with  | Error1(str) -> Assert.AreEqual "TryWith Error1" "x" str  | Error2(str, i) -> Assert.AreEqual "TryWith Error2" 10 i  function1 10 10  function1 10 20 |

***JS***

|  |
| --- |
| var function1 = function (x) {  return function (y) {  return (function (thisObject) {  try {  return (function (thisObject) {  if (x == y) {  throw new Pit.Test.TryWithTests.Error1("x")  } else {  throw new Pit.Test.TryWithTests.Error2("x", 10)  }  })(thisObject);  } catch (matchValue) {  (function (thisObject) {  if (matchValue instanceof Pit.Test.TryWithTests.Error1) {  var str = matchValue.get\_Data0();  return Assert.AreEqual("TryWith Error1", "x", str);  } else {  return (function (thisObject) {  if (matchValue instanceof Pit.Test.TryWithTests.Error2) {  var i = matchValue.get\_Data1();  var str = matchValue.get\_Data0();  return Assert.AreEqual("TryWith Error2", 10, i);  } else {  return Pit.FSharp.Core.Operators.Reraise();  }  })(thisObject);  }  })(thisObject)  }  })(this);  };  }; |

Note: raise or failwith method calls will get translated to “throw” statements

**Try..finally**

***F#***

|  |
| --- |
| let mutable x = 0  try  doSomething()  finally  x <- 1 |

***JS***

|  |
| --- |
| var x = 0;  (function(thisObject) {  try {  return Program.TestModule.doSomething();  } finally {  x = 1;  }  })(this); |

Note: for more information check this MSDN [link](http://msdn.microsoft.com/en-us/library/dd233218.aspx).

#### use Keyword

The use keyword has a form that resembles that of the let binding:

use *value* = *expression*

It provides the same functionality as a let binding but adds a call to Dispose on the value when the value goes out of scope. Note that the compiler inserts a null check on the value, so that if the value is null, the call to Dispose is not attempted.

***F#***

|  |
| --- |
| let s1 = seq { 0.. 10 }  use e = s1.GetEnumerator()  while e.MoveNext() do  doSomething() |

***JS***

|  |
| --- |
| var s1 = Pit.FSharp.Core.Operators.CreateSequence(Pit.FSharp.Core.Operators.op\_Range(0)(10));  var e = s1.IEnumerable1\_GetEnumerator();  return (function(thisObject) {  try {  while (e.IEnumerator\_MoveNext()) {  Program.TestModule.doSomething()  };  return null;  } finally {  (function(thisObject) {  if (e.containsInterface("\_\_getIDisposable")) {  return e.IDisposable\_Dispose();  } else {  return null;  }  })(thisObject)  }  })(this); |

Note: for more information check this MSDN [link](http://msdn.microsoft.com/en-us/library/dd233240.aspx).

#### Options

The option type in F# is used when an actual value might not exist for a named value or variable. An option has an underlying type and can hold a value of that type, or it might not have a value.

***F#***

|  |
| --- |
| [<Js>]  let keepIfPositive (a : int) = if a > 0 then Some(a) else None |

***JS***

|  |
| --- |
| var keepIfPositive = function(a) {  return (function(thisObject) {  if (a > 0) {  return new Pit.FSharp.Core.FSharpOption1.Some(a);  } else {  return new Pit.FSharp.Core.FSharpOption1.None();  }  })(this);  }; |

The value None is used when an option does not have an actual value. Otherwise, the expression Some( ... ) gives the option a value. The values Some and None are useful in pattern matching, as in the following function exists, which returns true if the option has a value and false if it does not.

F#

|  |
| --- |
| [<Js>]  let exists (x : int option) =  match x with  | Some(x) -> true  | None -> false |

***JS***

|  |
| --- |
| var exists = function(x) {  return (function(thisObject) {  if (x instanceof Pit.FSharp.Core.FSharpOption1.None) {  return false;  } else {  var x1 = x.get\_Value();  return true;  }  })(this);  }; |

Note: More details can be found on this [link](http://msdn.microsoft.com/en-us/library/dd233245.aspx).

#### Discriminated Unions

*Discriminated unions* provide support for values that can be one of a number of named cases, possibly each with different values and types. Discriminated unions are useful for heterogeneous data; data that can have special cases, including valid and error cases; data that varies in type from one instance to another; and as an alternative for small object hierarchies. In addition, recursive discriminated unions are used to represent tree data structures.

|  |
| --- |
| type type-name =  | case-identifier1 [of type1 [ \* type2 ...]  | case-identifier2 [of type3 [ \* type4 ...]  ... |

Simple example is given below

***F#***

|  |
| --- |
| type Shape =  | Circle of float  | Square of float  [<Js>]  let pi = 3.141592654  [<Js>]  let area myShape =  match myShape with  | Circle radius -> pi \* radius \* radius  | Square s -> s \* s  [<Js>]  let radius = 15.0  [<Js>]  let myCircle = Circle(radius)  [<Js>]  let squareSide = 10.0  [<Js>]  let mySquare = Square(squareSide) |

***JS***

|  |
| --- |
| Shape = function() {  this.Tag = 0;  this.IsSquare = false;  this.IsCircle = false;  };  Shape.Circle = function(item) {  this.Item = item;  };  Shape.Circle.prototype = new Shape();  Shape.Circle.prototype.Equality = function(compareTo) {  var result = true;  result = result && this.get\_Item() == compareTo.get\_Item();  return result;  };  Shape.Circle.prototype.get\_Item = function() {  return this.Item;  };  Shape.Square = function(item) {  this.Item = item;  };  Shape.Square.prototype = new Shape();  Shape.Square.prototype.Equality = function(compareTo) {  var result = true;  result = result && this.get\_Item() == compareTo.get\_Item();  return result;  };  Shape.Square.prototype.get\_Item = function() {  return this.Item;  };  Shape.prototype.get\_Tag = function() {  return this.Tag;  };  Shape.prototype.get\_IsSquare = function() {  return this instanceof Shape.Square;  };  Shape.prototype.get\_IsCircle = function() {  return this instanceof Shape.Circle;  };    var pi = 3.141592654;  var area = function(myShape) {  return (function(thisObject) {  if (myShape instanceof Shape.Square) {  var s = myShape.get\_Item();  return (s \* s);  } else {  var radius = myShape.get\_Item();  return ((pi \* radius) \* radius);  }  })(this);  };  var radius = 15;  var myCircle = new Shape.Circle(radius);  var squareSide = 10;  var mySquare = new Shape.Square(squareSide); |

*Note: above code doesn’t include namespaces, but usually all JS code generated will have namespaces associated with it.*

*Note: for more information on this topic check this MSDN* [*link*](http://msdn.microsoft.com/en-us/library/dd233226.aspx)*.*

#### Records

Records represent simple aggregates of named values, optionally with members.

|  |
| --- |
| [ attributes ]  type [accessibility-modifier] typename = {  [ mutable ] label1 : type1;  [ mutable ] label2 : type2;  ...  }  member-list |

Pit has two forms of representation for F# records. As already stated in the [Attributes Syntax](#_Pit_Attributes_Syntax), a record can be annotated with [<JsObjectAttribute>] to generate Javascript like object notations, this could be useful in generating less code. Below are examples given for both forms,

***F#***

|  |
| --- |
| type Point = { x : float; y: float; z: float; } |

***JS***

|  |
| --- |
| Point = (function() {  function Point(x, y, z) {  this.x = x;  this.y = y;  this.z = z;  }  Point.prototype.Equality = function(compareTo) {  var result = true;  result = result && this.get\_X() == compareTo.get\_X();  result = result && this.get\_Y() == compareTo.get\_Y();  result = result && this.get\_Z() == compareTo.get\_Z();  return result;  };  Point.prototype.get\_x = function() {  return this.x;  };  Point.prototype.get\_y = function() {  return this.y;  };  Point.prototype.get\_z = function() {  return this.z;  };  return Point;  })(); |

***F#***

|  |
| --- |
| [<JsObject>]  type Point = { x : float; y: float; z: float; }  [<Js>]  let p = { x = 2.0; y = 3.0; z = 2.0 } |

***JS***

|  |
| --- |
| var p = {  x: 2,  y: 3,  z: 2  }; |

*Note: JS for Types is not created when [<JsObject>] attribute is used.*

Pit supports object equality for F# records, this means record types of both the modes can be pattern matched, and compared.

##### Nested Records

Pit supports nested records i.e. any inner member or named values can refer another record type

***F#***

|  |
| --- |
| [<JsObject>]  type Address = { street: string;  pincode : int  }    [<JsObject>]  type Employee = {  id : int; name : string; address : Address }  [<Js>]  let emp:Employee = { id = 0; name = "Robert" ; address = { street = "Green street" ; pincode = 420 } } |

***JS***

|  |
| --- |
| Address = (function() {      function Address(street, pincode) {          this.street = street;          this.pincode = pincode;      }      Address.prototype.get\_street = function() {          return this.street;      };      Address.prototype.get\_pincode = function() {          return this.pincode;      };      return Address;  })();  Employee = (function() {      function Employee(id, name, address) {          this.id = id;          this.name = name;          this.address = address;      }      Employee.prototype.get\_id = function() {          return this.id;      };      Employee.prototype.get\_name = function() {          return this.name;      };      Employee.prototype.get\_address = function() {          return this.address;      };      return Employee;  })();  var emp = {          id: 0,          name: "Robert",          address: {              street: "Green street",              pincode: 420          }      }; |

*Note: Above code generated with [<JsObject>] type attribute declared*

***JS***

|  |
| --- |
| var emp = new Employee1(0, "Robert", new Pit.Test.RecordsTests.Address("Green street", 420)); |

*Note: Above code generated without [<JsObject>] type attribute declared*

Record extension with the use of “with” keyword is also supported in Pit.

***F#***

|  |
| --- |
| type CustomPoint = {x : int ; y : int} with          [<Js>]member this.xy = this.x \* this.y |

*Note:[<JsObject>] type attribute cannot be declared while extending records with the use of “with” keyword*

***JS***

|  |
| --- |
| CustomPoint = (function() {      function CustomPoint(x, y) {          this.x = x;          this.y = y;      }      CustomPoint.prototype.get\_x = function() {          return this.x;      };      CustomPoint.prototype.get\_y = function() {          return this.y;      };      CustomPoint.prototype.get\_xy = function() {          return (this.get\_x() \* this.get\_y());      };      return CustomPoint;  })(); |

Note: for more information on this topic check this MSDN [link](http://msdn.microsoft.com/en-us/library/dd233184.aspx).

#### Tuples

A *tuple* is a grouping of unnamed but ordered values, possibly of different types.

***F#***

|  |
| --- |
| // Tuple of two integers.  ( 1, 2 )  // Triple of strings.  ( "one", "two", "three" ) |

Example tuple code generation in JS is as below,

***F#***

|  |
| --- |
| [<Js>]  let (a, b) = (1, 2) |

***JS***

|  |
| --- |
| var patternInput = {  Item1: 1,  Item2: 2  };  var b = patternInput.Item2;  var a = patternInput.Item1; |

Note: for more information check this MSDN [link](http://msdn.microsoft.com/en-us/library/dd233200.aspx).

#### Reference Cells

*Reference cells* are storage locations that enable you to create mutable values with reference semantics.

***F#***

|  |
| --- |
| // Declare a reference.  let refVar = ref 6  // Change the value referred to by the reference.  refVar := 50 |

***JS***

|  |
| --- |
| var refVar = 6;  refVar = 50; |

Pit identifies reference variables and declares as normal Javascript variables. The reference cells are typical F# features and so in Javascript mode it doesn’t require to be like that.

Note: for more information on this topic check this MSDN [link](http://msdn.microsoft.com/en-us/library/dd233186.aspx).

#### List

A list in F# is an ordered, immutable series of elements of the same type.

***F#***

|  |
| --- |
| [<Js>]  let list123 = [ 1; 2; 3 ]  [<Js>]  let listOfSquares = [ for i in 1 .. 10 -> i\*i ] |

***JS***

|  |
| --- |
| var list123 = new Pit.FSharp.Collections.FSharpList1.Cons(1, new Pit.FSharp.Collections.FSharpList1.Cons(2, new Pit.FSharp.Collections.FSharpList1.Cons(3, new Pit.FSharp.Collections.FSharpList1.Empty())));  var listOfSquares = Pit.FSharp.Collections.SeqModule.ToList(Pit.FSharp.Core.Operators.CreateSequence(Pit.FSharp.Collections.SeqModule.Delay(function() {  return Pit.FSharp.Collections.SeqModule.Map(function(i) {  return (i \* i);  })(Pit.FSharp.Core.Operators.op\_Range(1)(10));  }))); |

The generated JS creates the objects for FSharpList Cons to build up the list. All the higher-order methods are supported for the List data structure. You can find more information about the F# List higher order methods in this [link](http://msdn.microsoft.com/en-us/library/ee353738.aspx).

#### Sequence

A *sequence* is a logical series of elements all of one type. Sequences are particularly useful when you have a large, ordered collection of data but do not necessarily expect to use all the elements. Individual sequence elements are computed only as required, so a sequence can provide better performance than a list in situations in which not all the elements are used. Sequences are represented by the seq<'T> type, which is an alias for [IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx). The Pit FW has full support for System.IEnumerable<T> and works in eagerly (on-demand) when a seq type is iterated.

**Sequence Expressions**

A *sequence expression* is an expression that evaluates to a sequence. Sequence expressions can take a number of forms. The simplest form specifies a range. For example, seq { 1 .. 5 } creates a sequence that contains five elements, including the endpoints 1 and 5.

***F#***

|  |
| --- |
| // Sequence that has an increment.  seq { 0 .. 10 .. 100 }  [<Js>]  let multiples = seq { for i in 1 .. 10 do yield i \* i } |

***JS***

|  |
| --- |
| var multiples = Pit.FSharp.Core.Operators.CreateSequence(Pit.FSharp.Collections.SeqModule.Delay(function() {  return Pit.FSharp.Collections.SeqModule.Map(function(i) {  return (i \* i);  })(Pit.FSharp.Core.Operators.op\_Range(1)(10));  })); |

Pit also supports all higher-order Sequence functions ([link](http://msdn.microsoft.com/en-us/library/ee353635.aspx)).

Note: for more details you can check this MSDN link.

#### Array

Arrays are fixed-size, zero-based, mutable collections of consecutive data elements that are all of the same type. You can create arrays in several ways. You can create a small array by listing consecutive values between [| and |] and separated by semicolons, as shown in the following example,

***F#***

|  |
| --- |
| [<Js>]  let array1 = [| 1; 2; 3 |] |

***JS***

|  |
| --- |
| var array1 = [1, 2, 3]; |

Arrays get mapped directly to Javascript arrays. Javascript arrays can be heterogeneous, but since F# is strongly-typed you won’t be able to declare directly. Creating an “object” array would allow the arrays to be heterogeneous.

***F#***

|  |
| --- |
| [<Js>]  let array1: obj[] = [| box(1); box("hello");|] |

***JS***

|  |
| --- |
| var array1 = [1, "hello"]; |

*Note: It is not recommended to use box’ed arrays like above in most situations. It may be valid for scenarios where you are creating wrappers for other JS libraries.*

Pit supports all the higher-order functions of F# Array module; get more information on the Array module from this [link](http://msdn.microsoft.com/en-us/library/ee370273.aspx).

*Note: Array.Parallel is not implemented in Pit as it is not valid for JS mode.*

#### Array2D

F# and CLI multi-dimensional arrays are typically zero-based. However, CLI multi-dimensional arrays used in conjunction with external libraries (for examples, libraries associated with Visual Basic) be non-zero based, using a potentially different base for each dimension. The operations in this module will accept such arrays, and the basing on an input array will be propagated to a matching output array on the [Array2D.map](http://msdn.microsoft.com/en-us/library/ee353722.aspx) and [Array2D.mapi](http://msdn.microsoft.com/en-us/library/ee353734.aspx) operations. Non-zero-based arrays can also be created using[Array2D.zeroCreateBased](http://msdn.microsoft.com/en-us/library/ee340214.aspx), [Array2D.createBased](http://msdn.microsoft.com/en-us/library/ee340258.aspx) and [Array2D.initBased](http://msdn.microsoft.com/en-us/library/ee353632.aspx).

Pit in the underlying wraps uses the normal JavaScript arrays to hold 2D arrays.

***F#***

|  |
| --- |
| let array = Array2D.init 2 2 (fun i j -> i + j)  let i = Array2D.get arr 1 1 |

***JS***

|  |
| --- |
| var array = Pit.FSharp.Collections.Array2DModule.Initialize(2)(2)(function(i) {  return function(j) {  return (i + j);  };  });  var i = Pit.FSharp.Collections.Array2DModule.Get(array)(1)(1); |

*Note: for more information on this check the MSDN* [*link*](http://msdn.microsoft.com/en-us/library/ee353794.aspx)*.*

#### Set

Immutable sets based on binary trees, where comparison is the F# structural comparison function, potentially using implementations of the [IComparable](http://msdn.microsoft.com/en-us/library/system.icomparable.aspx) interface on key values. Pit supports Set data structure fully.

*Note: for more information on this check the MSDN* [*link*](http://msdn.microsoft.com/en-us/library/ee353619.aspx)*.*

#### Events

Events enable you to associate function calls with user actions and are important in GUI programming.

Pit enables all DOM events as first-class F# types. It is easy to hook-up to the DOM events and process them with simple F# code. Below is the list of DOM events,

|  |  |
| --- | --- |
| Event.click | DOM click event |
| Event.change | DOM change event |
| Event.blur | DOM blur event |
| Event.dblclick | DOM double click event |
| Event.error | DOM error event |
| Event.focus | DOM focus event |
| Event.keydown | DOM key down event |
| Event.keypress | DOM key press event |
| Event.mousedown | DOM mouse down event |
| Event.mouseover | DOM mouse over event |
| Event.mouseup | DOM mouse up event |
| Event.resize | DOM resize event |
| Event.select | DOM select event |
| Event.unload | DOM unload event |

***F#***

|  |
| --- |
| let el = document.getElementbyId(“btn”)  el  |> Event.click  |> Event.add(fun \_ -> alert(“Button clicked”) |

**Creating Custom Events**

***F#***

|  |
| --- |
| type MyClassWithCLIEvent [<Js>]() =  let event1 = new Event<\_>()  [<Js>] member this.Event1 = event1.Publish  [<Js>] member this.TestEvent(arg) =  event1.Trigger(this, arg) |

***JS***

|  |
| --- |
| MyClassWithCLIEvent = (function() {  function MyClassWithCLIEvent() {  this.event1 = new Pit.FSharp.Control.FSharpEvent1();  }  MyClassWithCLIEvent.prototype.TestEvent = function(arg) {  return this.event1.Trigger({  Item1: this,  Item2: arg  });  };  MyClassWithCLIEvent.prototype.get\_Event1 = function() {  return this.event1.get\_Publish();  };  return MyClassWithCLIEvent;  })(); |

#### Observables

F# observables are again first-class much like Events. The list of Observable functions can be found [here](http://msdn.microsoft.com/en-us/library/ee370313.aspx). Pit leverages the F# observables and provides full support for reactive programming. Below is an example showing a mouse event for several text elements,

***F#***

|  |
| --- |
| [<Js>]  let hook (i:int) (el:DomElement) (mouseMove:IEvent<HtmlEventArgs>) =  mouseMove  |> Observable.map(fun e -> e.ClientX, e.ClientY)  |> Observable.delay(i \* 100)  |> Observable.subscribe(fun (x,y) ->  el.Style.Left <- x.ToString() + "px"  el.Style.Top <- y.ToString() + "px"  )  |> ignore |

***JS***

|  |
| --- |
| hook = function (i) {  return function (el) {  return function (mouseMove) {  return Pit.FSharp.Core.Operators.op\_PipeRight  (Pit.FSharp.Core.Operators.op\_PipeRight(Pit.FSharp.Core.Operators.op\_PipeRight  (Pit.FSharp.Core.Operators.op\_PipeRight(mouseMove)(function (source) {  return Pit.FSharp.Control.ObservableModule.Map(function (e) {  return {  Item1: e.clientX,  Item2: e.clientY  };  })(source);  }))(function (w) {  return Pit.Rx.Observable.delay((i \* 100))(w);  }))(function (source) {  return Pit.FSharp.Control.ObservableModule.Subscribe(function (tupledArg) {  var x = tupledArg.Item1;  var y = tupledArg.Item2;  el.style.left = (x.toString() + "px");  return el.style.top = (y.toString() + "px");  })(source);  }))(function (value) {  return Pit.FSharp.Core.Operators.Ignore(value);  });  };  };  }; |

#### Lazy Computation

*Lazy computations* are computations that are not evaluated immediately, but are instead evaluated when the result is needed. This can help to improve the performance of your code.

***F#***

|  |
| --- |
| let x = 10  let result = lazy (x + 10)  let y = result.Force() |

***JS***

|  |
| --- |
| var x = 10;  var result = Pit.FSharp.Control.LazyExtensions.Create(function() {  return (x + 10);  });  var y = Pit.FSharp.Control.LazyExtensions.Force(result); |

*Note: Pit also uses Lazy evaluations in the implementation of Seq.initInfinite.*

*Note: for more information on this check the MSDN* [*link*](http://msdn.microsoft.com/en-us/library/dd233247.aspx)*.*

#### Computation Expression

Computation expressions in F# provide a convenient syntax for writing computations that can be sequenced and combined using control flow constructs and bindings. They can be used to provide a convenient syntax for *monads*, a functional programming feature that can be used to manage data, control, and side effects in functional programs.

* Pit also has full support for seq monad for creating sequences.
* Async is not supported in Pit as the Javascript environment is not suitable for those scenarios. In HTML5, Web Workers would allow getting work done asynchronously, this is more like a MailboxProcessor, and we intend to implement that for async support.

Note: for more information check this MSDN [link](http://msdn.microsoft.com/en-us/library/dd233182.aspx).

##### Maybe monad

Maybe monad uses Option<T> to say if an evaluation was successful or not. Below is the code for it,

***F#***

|  |
| --- |
| module Monads =  type Maybe<'a> = option<'a>  [<Js>]  let succeed x = Some(x)  [<Js>]  let bind p rest =  match p with  | None -> None  | Some r -> rest r  [<Js>]  let delay f = f()  type MaybeBuilder() =  [<Js>]  member b.Return(x) = succeed x  [<Js>]  member b.Bind(p, rest) = bind p rest  [<Js>]  member b.Delay(f) = delay f  [<Js>]  member b.Let(p,rest) : Maybe<'a> = rest p  [<Js>]  member b.ReturnFrom(x) = x  [<Js>]  let maybe = MaybeBuilder() |

***JS***

|  |
| --- |
| Pit.Monads.MaybeBuilder = (function() {  function MaybeBuilder() {}  MaybeBuilder.prototype.Return = function(x) {  return Pit.Monads.succeed(x);  };  MaybeBuilder.prototype.Bind = function(tupledArg) {  var p = tupledArg.Item1;  var rest = tupledArg.Item2;  return Pit.Monads.bind(p)(rest);  };  MaybeBuilder.prototype.Delay = function(f) {  return Pit.Monads.delay(f);  };  MaybeBuilder.prototype.Let = function(tupledArg) {  var p = tupledArg.Item1;  var rest = tupledArg.Item2;  return rest(p);  };  MaybeBuilder.prototype.ReturnFrom = function(x) {  return x;  };  return MaybeBuilder;  })();  Pit.Monads.succeed = function(x) {  return new Pit.FSharp.Core.FSharpOption1.Some(x);  };  Pit.Monads.bind = function(p) {  return function(rest) {  return (function(thisObject) {  if (p instanceof Pit.FSharp.Core.FSharpOption1.Some) {  var r = p.get\_Value();  return rest(r);  } else {  return new Pit.FSharp.Core.FSharpOption1.None();  }  })(this);  };  };  Pit.Monads.delay = function(f) {  return f();  };  Pit.Monads.get\_maybe = new Pit.Monads.MaybeBuilder(); |

Usage of maybe monad is given below,

***F#***

|  |
| --- |
| [<Js>]  let failIfBig n = maybe {if n > 1000 then return! None else return n}  [<Js>]  let safesum (inp1,inp2) =  maybe { let! n1 = failIfBig inp1  let! n2 = failIfBig inp2  let sum = n1 + n2  return sum }  [<Js>]  let result1() = safesum (999,1000) |

***JS***

|  |
| --- |
| Pit.MonadsTest.failIfBig = function(n) {  return function(builder) {  return builder.Delay(function() {  return (function(thisObject) {  if (n > 1000) {  return builder.ReturnFrom(new Pit.FSharp.Core.FSharpOption1.None());  } else {  return builder.Return(n);  }  })(this);  });  }(Pit.Monads.get\_maybe);  };  Pit.MonadsTest.safesum = function(tupledArg) {  var inp1 = tupledArg.Item1;  var inp2 = tupledArg.Item2;  return function(builder) {  return builder.Delay(function() {  return builder.Bind({  Item1: Pit.MonadsTest.failIfBig(inp1),  Item2: function(\_arg2) {  var n1 = \_arg2;  return builder.Bind({  Item1: Pit.MonadsTest.failIfBig(inp2),  Item2: function(\_arg1) {  var n2 = \_arg1;  var sum = (n1 + n2);  return builder.Return(sum);  }  });  }  });  });  }(Pit.Monads.get\_maybe);  };  Pit.MonadsTest.result1 = function() {  return Pit.MonadsTest.safesum({  Item1: 999,  Item2: 1000  });  }; |

#### Units of Measure

Floating point and signed integer values in F# can have associated units of measure, which are typically used to indicate length, volume, mass, and so on. By using quantities with units, you enable the compiler to verify that arithmetic relationships have the correct units, which helps prevent programming errors.

***F#***

|  |
| --- |
| [<Measure>] type C  [<Measure>] type F  [<Js>]   let to\_farenheit (x : float<C>) = x \* (9.0<F>/5.0<C>) + 32.0<F>  [<Js>]   let to\_celsius (x : float<F>) = (x - 32.0<F>) \* (5.0<C>/9.0<F>)    [<Js>]   let UOMeasure1() =      let f = to\_farenheit 20.<C>      f = 68 |

***JS***

|  |
| --- |
| Pit.Test.UOMTest.to\_farenheit = function(x) {      return ((x \* (9 / 5)) + 32);  };  Pit.Test.UOMTest.to\_celsius = function(x) {      return ((x - 32) \* (5 / 9));  };  Pit.Test.UOMTest.UOMeasure1 = function() {      var f = Pit.Test.UOMTest.to\_farenheit(20);      return Assert.AreEqual("Units Of Measure To Farenheit", 68, f);  }; |

Units of measure are used for static type checking. When floating point values are compiled, the units of measure are eliminated, so the units are lost at run time. Therefore, any attempt to implement functionality that depends on checking the units at run time is not possible. For example, implementing a ToString function to print out the units is not possible. And therefore, Pit would just generate the resulting JavaScript without any types.

#### Compiler Directives

The **pfc** compiler executable is used to compile F# project/DLL from command line. Supported commmand for Pit compiler is as stated below:

* pfc.exe “projectFilePath” /o:“jsOutputFilePath” /ft:true
* pfc.exe “dllFilePath” /o:“jsOutputFilePath” /ft:false

|  |  |
| --- | --- |
| /o: | JavaScript output file |
| /ft | Format JavaScript (true/false) |

## Javascript

Pit directly maps Integer, float and bool to the JS type. String requires special decoration since the JS string and the .NET string are different in APIs. Below are the mapping classes that Pit exposes (under Pit.Javascript namespace),

|  |  |
| --- | --- |
| **Javascript** | **Pit** |
| String | JsString |
| Array | JsArray |
| Regex | RegExp |
| DateTime | DateTime |
| Math | Math |

### JsString

JsString is a wrapper for Javascript string APIs. Below table shows the list of functions that is available,

|  |  |
| --- | --- |
| Length | Returns the length of the string |
| CharAt | Returns the character from an index |
| CharCodeAt | Returns the character code from an index |
| Concat | Concats two strings |
| IndexOf | Returns the index based on search string |
| LastIndexOf | Returns the last index based on search string |
| Match | Returns an array of matches against the regex |
| Replace | Replaces a string with another string |
| Search | Returns the index of the search string |
| Slice | Removes characters based on index |
| Split | Splits the string based on identifier |
| Substring | Returns the substring of a larger string |
| ToLower | Converts to lower string case |
| ToUpper | Converts to upper string case |

To initialize a JsString use the below code,

***F#***

|  |
| --- |
| let str = new JsString(“Hello World”) |

***JS***

|  |
| --- |
| var str = “Hello World” |

### JsArray

JsArray is a strongly-typed wrapper for Javascript Array APIs. Below table shows the list of functions that is available,

|  |  |
| --- | --- |
| Length | Returns the length of the array |
| Item | Indexer property that returns an item from the array |
| Join | Joins all the array elements as a string |
| Pop | Removes the last element of an array and returns that element |
| Push | Adds new elements to the end of an array and returns the new length |
| Reverse | Reverses the order of the elements in an array |
| Shift | Removes the first element of an array and returns that element |
| Slice | Selects a part of an array, and returns that array |
| Sort | Sort the elements of an array |
| Splice | Adds/Removes the elements from an array |
| Unshift | Adds new elements to the beginning of an array and returns the new length |

Sample F# code for all the functions,

***F#***

|  |
| --- |
| let a = [|1;2;3|]  let js = new JsArray<\_>(a)  let item = js.[1]  let str = js.Join()  let p = js.Pop()  let i = js.Push(4)  let i = js.Push([|4;5;|])  let rev = js.Reverse()  let revlen = rev.Length  let s = js.Shift()  let s = js.Slice(1)  let s = js.Slice(0, 2)  let s = js.Sort()  js.Splice(1, 0, 4)  let s = js.Unshift(4) |

In Debug mode, one important thing to understand here is the usage of “integer” in arrays will be treated as “floats”, this is because the Silverlight-Javascript Bridge identifies float as a common data type for marshaling data between Silverlight and Javascript. There is no way to work around this if you are using Javascript arrays. Best are to use normal F# arrays that will get compiled to JS arrays automatically.

### Math

Math implementation is a module type. Below is the list of functions that are available,

|  |  |
| --- | --- |
| E | Returns Euler's number (approx. 2.718) |
| LN2 | Returns the natural logarithm of 2 (approx. 0.693) |
| LN10 | Returns the natural logarithm of 10 (approx. 2.302) |
| LOG2E | Returns the base-2 logarithm of E (approx. 1.442) |
| LOG10E | Returns the base-10 logarithm of E (approx. 0.434) |
| PI | Returns PI (approx. 3.14159) |
| SQRT1\_2 | Returns the square root of 1/2 (approx. 0.707) |
| SQRT2 | Returns the square root of 2 (approx. 1.414) |
| abs(x) | Returns the absolute value of x |
| acos(x) | Returns the arccosine of x, in radians |
| asin(x) | Returns the arcsine of x, in radians |
| atan(x) | Returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians |
| atan2(y,x) | Returns the arctangent of the quotient of its arguments |
| ceil(x) | Returns x, rounded upwards to the nearest integer |
| cos(x) | Returns the cosine of x (x is in radians) |
| exp(x) | Returns the value of Ex |
| floor(x) | Returns x, rounded downwards to the nearest integer |
| log(x) | Returns the natural logarithm (base E) of x |
| [pow(x,y)](http://www.w3schools.com/jsref/jsref_pow.asp) | Returns the value of x to the power of y |
| [random()](http://www.w3schools.com/jsref/jsref_random.asp) | Returns a random number between 0 and 1 |
| [round(x)](http://www.w3schools.com/jsref/jsref_round.asp) | Rounds x to the nearest integer |
| [sin(x)](http://www.w3schools.com/jsref/jsref_sin.asp) | Returns the sine of x (x is in radians) |
| [sqrt(x)](http://www.w3schools.com/jsref/jsref_sqrt.asp) | Returns the square root of x |
| [tan(x)](http://www.w3schools.com/jsref/jsref_tan.asp) | Returns the tangent of an angle |

### RegExp

RegExp is implemented as a type. Below is the list of functions available,

|  |  |
| --- | --- |
| Exec | Tests for a match in a string. Returns the first match |
| Test | Tests for a match in a string. Returns true or false |
| Compile | Compiles a regular expression |

***F#***

|  |
| --- |
| let str = "Every man in the world! Every woman on earth!"  let r = new RegExp("man", "g")  let jsStr = new JsString(str)  let str2 = jsStr.Replace(r, "person") |

### Date

Date is implemented as a type. Below is the list of functions available,

|  |  |
| --- | --- |
| getDate() | Returns the day of the month (from 1-31) |
| getDay() | Returns the day of the week (from 0-6) |
| getFullYear() | Returns the year (four digits) |
| getHours() | Returns the hour (from 0-23) |
| getMilliseconds() | Returns the milliseconds (from 0-999) |
| getMinutes() | Returns the minutes (from 0-59) |
| getMonth() | Returns the month (from 0-11) |
| getSeconds() | Returns the seconds (from 0-59) |
| getTime() | Returns the number of milliseconds since midnight Jan 1, 1970 |
| getTimezoneOffset() | Returns the time difference between GMT and local time, in minutes |
| getUTCDate() | Returns the day of the month, according to universal time (from 1-31) |
| getUTCDay() | Returns the day of the week, according to universal time (from 0-6) |
| getUTCFullYear() | Returns the year, according to universal time (four digits) |
| getUTCHours() | Returns the hour, according to universal time (from 0-23) |
| getUTCMilliseconds() | Returns the milliseconds, according to universal time (from 0-999) |
| getUTCMinutes() | Returns the minutes, according to universal time (from 0-59) |
| getUTCMonth() | Returns the month, according to universal time (from 0-11) |
| getUTCSeconds() | Returns the seconds, according to universal time (from 0-59) |
| parse() | Parses a date string and returns the number of milliseconds since midnight of January 1, 1970 |
| setDate() | Sets the day of the month (from 1-31) |
| setFullYear() | Sets the year (four digits) |
| setHours() | Sets the hour (from 0-23) |
| setMilliseconds() | Sets the milliseconds (from 0-999) |
| setMinutes() | Set the minutes (from 0-59) |
| setMonth() | Sets the month (from 0-11) |
| setSeconds() | Sets the seconds (from 0-59) |
| setTime() | Sets a date and time by adding or subtracting a specified number of milliseconds to/from midnight January 1, 1970 |
| setUTCDate() | Sets the day of the month, according to universal time (from 1-31) |
| setUTCFullYear() | Sets the year, according to universal time (four digits) |
| setUTCHours() | Sets the hour, according to universal time (from 0-23) |
| setUTCMilliseconds() | Sets the milliseconds, according to universal time (from 0-999) |
| setUTCMinutes() | Set the minutes, according to universal time (from 0-59) |
| setUTCMonth() | Sets the month, according to universal time (from 0-11) |
| setUTCSeconds() | Set the seconds, according to universal time (from 0-59) |
| toDateString() | Converts the date portion of a Date object into a readable string |
| toLocaleDateString() | Returns the date portion of a Date object as a string, using locale conventions |
| toLocaleTimeString() | Returns the time portion of a Date object as a string, using locale conventions |
| toLocaleString() | Converts a Date object to a string, using locale conventions |
| toString() | Converts a Date object to a string |
| toTimeString() | Converts the time portion of a Date object to a string |
| toUTCString() | Converts a Date object to a string, according to universal time |
| UTC() | Returns the number of milliseconds in a date string since midnight of January 1, 1970, according to universal time. This is a static method. |

### DOM Events

DOM elements have a set of standard events as defined in W3C. Below is the list of events present under “Pit.Dom” namespace,

|  |  |
| --- | --- |
| Event.click | Mouse click on an object |
| Event.change | The content of a field changes |
| Event.blur | An element loses focus |
| Event.dblclick | Mouse double click on an object |
| Event.error | An error occurs when loading a document or an image |
| Event.focus | An element gets focus |
| Event.keydown | A keyboard key is pressed |
| Event.keypress | A keyboard key is pressed or held down |
| Event.keyup | A keyboard key is released |
| Event.load | A page or image is finished loading |
| Event.mousedown | A mouse button is pressed |
| Event.mousemove | Mouse is moved |
| Event.mouseout | Mouse is moved off an element |
| Event.mouseover | Mouse is moved over an element |
| Event.mouseup | Mouse button is released |
| Event.resize | Window or frame is resized |
| Event.select | Text is selected |
| Event.unload | The user exits the page |

### AJAX

Pit supports AJAX wrappers to XHR and ActiveXObject (for IE). Below is an example F# code that shows how to initialize and use the XMLHttpRequest with a GET request,

***F#***

|  |
| --- |
| module Ajax =  [<Js>]  let createRequest () =  if window.ActiveXObject <> null then  new XMLHttpRequestIE("Msxml2.XMLHTTP") :> XMLHttpRequest  else  new XMLHttpRequest()  [<Js>]  let createGet url =  let req = createRequest()  req.Open("GET", url, false)  req  [<Js>]  let createAsyncGet url =  let req = createRequest()  req.Open("GET", url, true)  req |

With IE, we do a check using “window.ActiveXObject <> null” and initialize the XMLHttpRequestIE(“Msxml2.XMLHTTP”), this will generate the JS accordingly.

Assuming we have some file “data.txt” in the same domain,

|  |
| --- |
| let request = Ajax.createAsyncGet("data.txt")  request.OnReadyStateChange <- fun () ->  if request.ReadyState = 4 && request.StatusText = "OK" then  // do something  request.Send(null) |

*Note: For more details on AJAX refer this article <http://ajaxpatterns.org/XMLHttpRequest_Call>*

***Pit currently doesn’t resolve the response text as JSON, This is under plans and would be done in future releases.***

## Pit Namespace Diagram

This section will detail the namespaces and classes that are present under Pit.

**Pit Namespace**

**Pit.Javascript Namespace**

**Pit.Dom Namespace**

**Pit.Dom.HTML5 Namespace**

# Working with Pit source

Pit is open-sourced based on Apache 2.0 license and it could be found [here](http://www.apache.org/licenses/LICENSE-2.0.html). This section details the list of things required to work with Pit source.

## Project Structure

The project is basically split into two, Compiler and Library.

*Note: use the suffix “dbg” for debug projects.*

## Building Projects

* To build all project double click “run.bat” in root folder. Ensure to update “run.bat” with the location in root folder

**run.bat content**

|  |
| --- |
| "C:\Windows\Microsoft.NET\Framework\v4.0.30319\msbuild.exe" "D:\Work\Pit\BuildAll.proj"  pause |

* Generated assemblies can be found in the following relative path

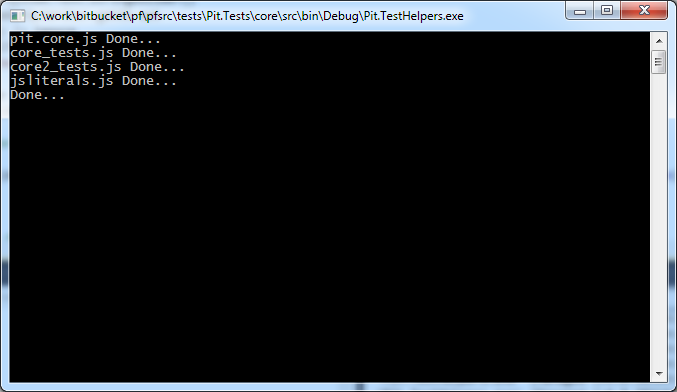
build\Pit.Setup\ProductBuild

* “BuildAll.proj” file contains xmls definitions to build all projects and to copy the output to specified location.

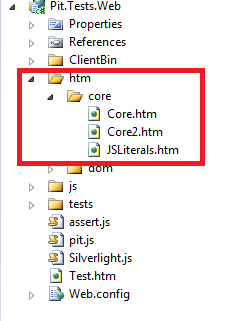
## Tests

Pit uses a simple way of testing the generated JS. It uses simple “Asserts” based on the F# code, and in turn the JavaScript uses “console” to log the results. There are 3 project types each having different sets of tests to execute. Follow the below steps to execute the tests,

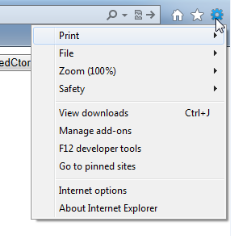
* Open “Pit.Tests.sln” solution from the tests folder.
* Execute the console application to generate the JS.
  + Note: Pit.Tests project should be the default “Startup” project.



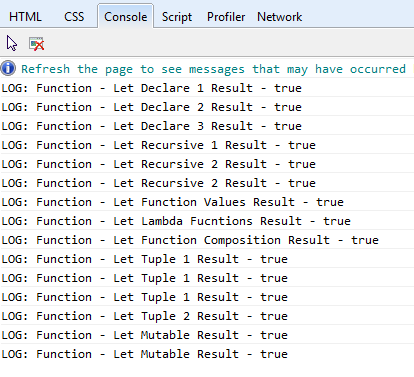
* Execute “run.bat” (present in the project folder) to format the JS and place it in the scripts folder in the Web project.
* The HTM files present under htm\core are the tests file.



* To run the HTM files,
  + Right-click and click “View in browser”
    - IE users can select F12 developer tools / FF users can just enable Firebug to enable the console logging.



* + Click the buttons that are available in the page. The results should be printed out in the console window,



Creating and adding new test cases is also very easy. Below is a sample F# test,

***F#***

|  |
| --- |
| [<Js>]  let LetDeclare() =  let x = 1  Assert.AreEqual "Let Declare 1" x 1 |

***JS***

|  |
| --- |
| Pit.Test.LetTests.LetDeclare = function() {  var x = 1;  return Assert.AreEqual("Let Declare 1", x, 1);  }; |

*Note: Check the tests that are available in the project.*

# FAQ

## Match expression with same alias objects

match g with

| :? Bool as g -> () // here we can't have both the values as g

This creates Javascript scoping issues. Easiest way to work around this is to write the expression as,

match gen with

| :? Bool as g -> ()

## Match expressions with .NET types

match arrays with

| :? ('T[][]) as ts -> ts |> concatArrays

| \_ -> arrays |> Seq.toArray |> concatArrays

The above expressions are specific to .NET and can't be supported in Javascript. Best is to write up code that would automatically handle these scenarios in a single function call, avoiding the match expression.

## Member functions with inner let functions accessing global variables

Member functions that has inner let functions such as these,

let started = ref false

….

interface IEnumerable<T> with

member x.MoveNext() =

let rec next() =

if not !started then started := true;

e.MoveNext() && (f e.Current || next())

next()

Have problem in JS generation, as the variable scope is incorrect when it gets translated to JS. This will be fixed later, As of now workaround would be to move the next() method as a member function to the declared type and then call it directly inside the interface method.

## Member functions that are synonymous to String struct

If you have a member function that is synonymous with "String", example "Join" method that returns a string

[<CompileTo("join")>]

member x.Join() = ""

The call AST for this generates

Let (j, Call (Some (arr), System.String Join(), []), Value (<null>))

The method declaring type is returned as "System.String" for whatever reason we are not sure of. Easiest workaround is to change the API name and have the CompileToAttribute still set as "join",

[<CompileTo("join")>]

member x.JoinAll() = ""

## Arguments in type members (functions)

Usually in F# if you have 2 or more arguments in a type member it will consider it as a Tuple. This would work normally if you have custom type classes etc., There are some scenarios in which you need to ignore the tuple generation and make it generate as a parameterized function,

* Operator overloads

[<Js;JsIgnore(IgnoreTuple=true)>]

static member (+) (a: Set<'T>, b: Set<'T>) =

* Interface members

type GenericComparer<'T when 'T : comparison>() =

interface System.Collections.Generic.IComparer<'T> with

[<Js;JsIgnore(IgnoreTuple=true)>]

member this.Compare(x,y) =

Pit will then generate the methods as normal parameterized functions in JS.

## Overloaded methods in types

Overloaded methods need to be marked with "JsOverloadMember(\_NewName\_)" attribute,

member x.GetName() = ""

[<JsOverloadMember("GetName1")>]

member x.GetName(surname) = ""

In general FP, overloaded members are not really the idea to use, so we don't check for overloaded methods in the compiler. This would be the apt way of dealing this scenario.