

---

# TITLE - F SHARP

---

**Jueping Wang**  
NetId: juepingwang  
University of Arizona  
juepingwang@email.arizona.edu

**Cameron Kazmierski**  
NetId: cameron168  
University of Arizona  
cameron168@email.arizona.edu

April 30, 2020

## ABSTRACT

F# is a unique programming language designed by Microsoft to fill a functional programming language gap in its .NET development framework. F# includes data types and structures similar to most major programming languages. F# differentiates itself by providing all the functionality of a functional programming language as well as the functionality of objects.

## 1 Introduction

F# was originally created by Microsoft to serve as its functional language in the .NET framework of software development tools. The language has data types similar to other programming languages, such as integers, floats, tuples, and lists. F# differs in that it allows more control over the size of the data type in memory. F#'s control structures are also very similar to common object-oriented programming languages with if-elif-else statements, for loops, and while loops. Like other functional programming languages, F# also has match expressions. Subprograms of F# include support for functions, classes, and modules. These characteristics differentiate F# from both object-oriented and functional programming languages, creating a multi-paradigm language with the characteristics of both types of programming languages.

## 2 History

F# was originally developed at Microsoft Research [1], and the first version was released in May 2005 [2]. F# is a functional-first, multi-paradigm language [1]. The syntax of F# provides a friendly way to write code. Therefore, F# code is relatively easy to maintain compared to C#. In addition, F# is a powerful computing language for manipulating data, such as in the fields of data analysis, data visualizations, and high-performance analytics. Nowadays, F# is commonly utilized in data science and financial fields. For example, FsLab is a collection of popular F# open-source libraries for data science applications.

F# improves the language interoperability on the .NET Framework[1]. One of the most important features of the .NET Framework is to allow developers using one language to interact with the other languages, C# and Visual Basic [1]. Therefore, the .NET Framework provides a solution for developers to solve more complex tasks. Before F# emerged, C# and Visual Basic were the two languages in the .NET Framework ecosystem[1]. Even they have some aspects of functional programming, but F# was designed specifically to bring together the utility of functional programming with the .NET library. After F# released, it became the perfect complement to C# and Visual Basic in the .NET Framework.

## 3 Data Types

F# is the newest programming language in .NET family. It inherits a lot of features from C# and Visual Basic. F# also has data types, such as int and float. Additionally, F# has similar arithmetic operators as well. A type is a concept or abstraction, and is primarily about enforcing safety [3]. F# is statically typed, which means that the type checking will be done at compile time [1]. For Example, if we try to assign a string to an integer type, we will receive a compiler error.

Table 1: Integer Data Types[3][4]

Type	Suffix	Range	Remarks	Example
byte	uy	0 to 255	8-bit unsigned integer	42uy
sbyte	y	128 to 127	8-bit signed integer	-11uy
int16	s	32,768 to 32,767	16-bit signed integer	42s
uint16	us	0 to 65,535	16-bit unsigned integer	12222us
int, int32		$-2^{31}$ to $2^{31} - 1$	32-bit signed integer	1248
uint32	u	0 to $2^{32} - 1$	32-bit unsigned integer	200u
int64	L	$-2^{63}$ to $2^{63} - 1$	64-bit signed integer	200L
uint64	UL	up to $2^{64} - 1$	64-bit unsigned integer	200UL
float32	F	1.5E-45 to 3.41E+38	32-bit signed floating point number (7 significant digits)	-11.0F
float		5.0E-324 to 1.7E+308	64-bit signed floating point number (15-16 significant digits)	-11.0
decimal	M	1.0E-28 to 7.9E28	128-bit signed floating point number (28-29 significant digits)	-11.0M
bool	true or false	Stores boolean values	true , false	
char			Single unicode characters	'a'
string			Unicode text	"Hello"

```

1 // Compiler Error
2 let a : int = 1.20F
3 printfn "a: %d" a
4
5 //error FS0001: This expression was expected to have type
6 //      'int'
7 //but here has type
8 //      'float32'
9 //compiler exit status 1

```

**Caution:** The **\*\*** operator, only works for float and float32 types. To raise the power of an integer value, you must either convert it to a floating-point number first or use the **pown** function [3].

```

1 // Simple example of Data type and Arithmetic operators
2 // initialize variables
3 let a : float32 = 1.20F
4 let b : float32 = 2.21F
5 printfn "a: %f" a
6 printfn "b: %f" b
7 printfn "sum: %f" (a + b)
8
9 // output:
10 // a: 1.200000
11 // b: 2.210000
12 // sum: 3.410000

```

**Immutability:** When you read some F# documentation, it may be very hard to find the word, variable. In some programming languages, we are allowed to re-assign a new value to a previously instantiated variable. However, in F#,

Table 2: Arithmetic operators[3][4]

Operator	Description	Example	Result
+	Addition	1 + 2	3
-	Subtraction	1 - 2	-1
	Multiplication	2 * 3	6
/	Division	8L / 3L	2L
*	Power[*]	2.0 ** 5.0	32.0
%	Modulus	7 % 3	1
&&	And	true && false	false
	Or	true    false	true
not	Not	not false	true

we can not change the state after the declaration. In other words, we can not change the value of the "declared variable". If we want to change the state, we have to use the keyword "**mutable**" to change the state.

```

1 // example of key word mutable
2 let mutable x : int = 10
3 x <- 15
4 printfn "x: %d" x
5
6 // output:
7 // x: 15

```

**Tuple:** A tuple is an ordered collection of data and an easy way to group common pieces of data together, and tuples can contain any number of values of any type [3].

```

1 // example of Tuple
2
3 let names = ("John", 23)
4 printfn "name: %s" (fst names)
5 printfn "age: %d" (snd names)
6
7 // output:
8 // name: John
9 // age: 23

```

**List:** A list in F# is an ordered, immutable series of elements of the same type [5]. The syntax used is semicolon-delimited, or to separate each element with a semicolon in closed brackets. The empty list is denoted as []. In addition, list in F# supports list comprehension syntax.

```

1 // Basic syntax of List
2 let odds = [1; 3; 5; 7; 9]
3 let evens = [2; 4; 6; 8; 10]

```

```

1 // Examples:
2 // From: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/lists
3 let list1 = [ 1; 2; 3 ]
4
5 printfn "list1.IsEmpty is %b" (list1.IsEmpty)
6 printfn "list1.Length is %d" (list1.Length)
7 printfn "list1.Head is %d" (list1.Head)

```

Table 3: Some built-in List functions

[3]		
Function	Type	Description
List.length	'a list -> int	Returns the length of a list.
List.head	'a list -> 'a	Returns the first element in a list.
List.tail	'a list -> 'a list	Returns the given list without the first element.
List.zip	('a -> bool) -> 'a list -> 'a list	Given two lists with the same length, returns a joined list of tuples.

## 4 Control Structures

Control structures have a significant role in programming languages. When a program is in its executable stage, control structures allow computers to make running decisions via test expressions. There are three types of decisions in code that can be performed via control structures, **Sequential mode**, **Selection mode**, and **Repetition mode** [6]. Sequential mode is sequential execution of code statements. In other words, execution always begins at the first statement of the program, and statements are executed one at a time, in order, and from top to bottom [7]. Selection mode differs from sequential mode by choosing between two or more alternative paths [6]. The final discussion mode type, repetition, is used for repeating a piece of code multiple times [6].

### 4.1 Sequential mode

Sequential execution is the most commonly used case of a control structure. It executes the first statement at the top of the code and sequentially runs through each statement. The following example demonstrates sequential execution:

```

1 /* example of sequential execution */
2 let x : int = 2           // 1th code statement
3 let y : int = 5.         // 2th code statement
4 let z : int = x + y      // 3th code statement
5 printfn "x + y = %d" z   // final code statement
6
7 // output:
8 // x + y = 7

```

### 4.2 Selection mode

**if-then statement** The if statement uses the if and then keywords. The test expression must be of type Boolean and if it evaluates to true, then the code following the then statement is executed. [3].

```

1 /* example of if-then statement */
2
3 let condition : bool = true
4
5 if condition then
6     printfn "conditional expression is true , so I ran ."
7
8 printfn "Hello , I am here now ."
9
10 // output:
11 // conditional expression was true , so I ran .
12 // Hello , I am here now .

```

If the test expression (at line 5) evaluates to false, the output will only be "Hello, I am here now.", because the then statement in the if-then expression will not run.

**if-then-else statement** If the test expression evaluates to true, then the first block of code executes; otherwise, the second block of code executes[3]. In other words, the if-else statement will choose which branch needs to be executed.

```

1  /* example of if-then-else statement */
2
3  let x : int = 3
4
5  if x % 2 = 1 then
6      printfn "x is odd."
7  else
8      printfn "x is even."
9
10 // output:
11 // x is odd.

```

An example of the else branch in the code above occurs when x is equal to 4, and x mod 2 is not equal to 1. The result of this test expression is false. Therefore, the else part will be executed. In this case, the output will be "x is even."

**if-then-elif-else statement** This statement allows developers to have multiple branches (or choices) and only executes one of them via the test expression. Each elif part has its own test expression.

```

1  /* example of if-then-elif-else */
2
3  //find_gpa : char -> float
4  let find_gpa (x : char) =
5      if x = 'A' then 4.0
6      elif x = 'B' then 3.0
7      elif x = 'C' then 2.0
8      elif x = 'F' then 0.0
9      else -1.0
10 printfn "GPA is = %.2f" (find_gpa 'A')
11 printfn "GPA is = %.2f" (find_gpa 'B')
12 printfn "GPA is = %.2f" (find_gpa 'C')
13 printfn "GPA is = %.2f" (find_gpa 'F')
14 printfn "GPA is = %.2f" (find_gpa 'Z')
15
16 // output:
17 // GPA is = 4.00
18 // GPA is = 3.00
19 // GPA is = 2.00
20 // GPA is = 0.00
21 // GPA is = -1.00

```

**Match expressions** Sometimes, it is hard to deal with multiple branches by using different types of if-elif-else statements. F# provides match expressions to compensate for programming requirements with many branches. The official F# documentation says, "The pattern matching expressions allow for complex branching based on the comparison of a test expression with a set of patterns. In the match expression, the test expression is compared with each pattern in turn, and when a match is found, the corresponding result expression is evaluated and the resulting value is returned as the value of the match expression." [5]. The syntax of match expressions and pattern matching are very similar to Haskell.

```

1  // from: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/match-
    // expressions#guards-on-patterns
2
3  // Match expression.
4  match test-expression with
5  | pattern1 [ when condition ] -> result-expression1
6  | pattern2 [ when condition ] -> result-expression2
7  | ...
8
9  // Pattern matching function.
10 function
11 | pattern1 [ when condition ] -> result-expression1

```

```

12 | pattern2 [ when condition ] -> result-expression2
13 | ...

```

### 4.3 Repetition mode

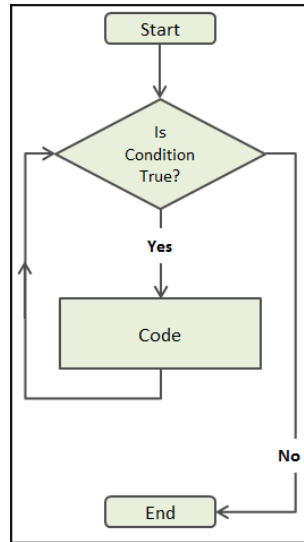


Figure 1: Repetition Control Structure

Repetition control structures allow us to efficiently repeat executing one or more statements up to a desired number of times or until a condition is met. Figure.1 demonstrates this process, which shows how repetition control structures work. First, before the repetition control structure starts, one or more control variables need to be initialized. Second, the control variable is tested via the test expression. If it evaluates to true, the statements in the repetition control structure are executed; otherwise, it will stop executing code within the repetition. In F#, there are two types of repetition control structures, the for-loop and the while-loop. These are very similar to loops in Python or Java.

Compared to Haskell, F# provides repetitive control structures. In Haskell, recursion must be used to perform repetition. Sometimes, the implementation of recursion is much simpler than the iterative solution, but it may be harder to understand which could increase the cost of maintenance. Since F# provides iterative solutions, we can convert some understandable recursion solutions to iterative solutions. Therefore, as a functional language, F# will be more friendly and more flexible compared to Haskell in creating repetitive control structures.

```

1 // for ... to
2 for var = start-expr to end-expr do
3     ... // loop body
4
5 //for ... downto
6 for var = start-expr downto end-expr do
7     ... // loop body
8
9 //for ... in
10 for pattern in enumerable-expression do
11     body-expression
12
13 //while-loop
14 while condition-expression do
15     body-expression

```

```

1 /* example of for ... to */
2 /* print number from 1 to 5 */
3
4 let for_to (x : int) =
5     for i = x to 5 do

```

```

6         printfn "i = %d" i
7
8 // call function for_to
9 for_to 1
10
11 /* example of for...downto */
12 /* print number from 5 to 1 */
13 let for_downto (x : int) =
14     for i = x downto 1 do
15         printfn "i = %d" i
16
17 // call function for_downto
18 for_downto 5
19
20 /* example of for...in */
21 /* traversing the list from the first element to the last element */
22 let list1 = [ 2; 4; 6; 8; 10 ]
23 for i in list1 do
24     printfn "%d" i
25
26 /* example of while-do loop */
27 /* find sum of 1,2,3,4,5 */
28 let mutable i = 1
29 let mutable sum = 0
30 while (i <= 5) do
31     sum <- sum + i
32     i <- i + 1
33 printfn "%d" sum

```

## 5 Subprograms

Subprograms are the essential building blocks of a programming language. They allow programmers to transform simple control structures and programming logic into code that can complete complex tasks. Skilled programmers can break complex tasks down into simple and easily manageable subprograms, such as *functions*, *modules*, and *classes*. Functions are the most basic way to create a set of instructions for a frequent operation that performs a task, returns a value, or modifies a value. Modules are typically larger than functions and are used for sectioning off parts of programs that are similar or perform related operation. Modules can hold variables, functions, and also classes. Just like modules, classes are used to hold variables and functions that are similar or are used in similar ways. Classes are very similar to modules, except in one key way. Classes can be instantiated into objects, or multiple instances of a class can exist that all hold different data. This makes classes useful for tracking groups of data that can be manipulated in similar ways. All of these subprograms allow for the modularity of the whole program, or breaking down the program into manageable bits.

### 5.1 Functions

Functions are the most common way to segment a program into reusable code. Functions can vary greatly in their uses. Functions can take any number of variables and can all return values, making them extremely versatile. The following example demonstrates a few basic functions:

```

1 /* example of a function */
2
3 // function declaration along with its parameter num1
4 // The return statement of the function is everything after the equal sign.
5 let func1 num1 =
6     num1 + 3
7
8 printfn "%d \n" (func1 4)
9
10 // output:
11 // 7

```

## 5.2 Modules

Modules in F# serve many different roles. Modules can be used to breakup sections of code within the same file and also can be used to define a whole file. This means modules in F# are nested and can be defined as either internal (within a file) or external (outside file). In order to define a module the keyword **module** can be used. Additionally, files will be implicitly defined if not explicitly defined [5]. Here is an example of an internal module:

```

1  /* example of a internal module */
2  /* All in file internal.fs */
3  module internal                //Whole file is module internal
4
5  module example1 =              //Module name declared as example1 (also nested within
    internal module)
6      let value1 = 25            //Module holds a value within it
7      let func1 num1 =          //Module also hold a function within it
          num1 + 3
8
9
10 /* A few ways to access the code from the internal module using qualifying names.*/
11 let modVal1 = example1.value1  //Using qualifying names, or the name of the module.
12 printfn "%d \n" modVal1        //Output: 25
13
14 open example1                  //Opening the module instead of using qualifying names
15
16 let modVal2 = func 3
17 printfn "%d \n" modVal2        //Output: 6

```

Additionally these modules can be used externally in files outside of the original file. Here is an example of an external module:

```

1  /* example of a external module */
2  /* All in file external.fs */
3
4  printfn "%d \n" internal.example1.value1  //Must call all of the nested modules to
    get to value
5
6  //Output: 25
7  open internal                          //Opened the file's module
8  printfn "%d \n" example1.value1        //Output: 25
9
10 open internal.example1                  //Opened the internal module in the
    external module
11 printfn "%d \n" value1                  //Output: 25

```

One other thing to note about F# is namespaces. Namespaces behave in almost the exact same way as modules, except they are not nested and namespaces are top-level. Namespaces are used for groups of modules that all do related, but not necessarily similar things. [5]

## 5.3 Classes

Classes in F# behave like classes and objects do in most languages. They act to form instantiatable objects that can all perform the same functions and hold the same data, but that can hold unique data for each individual object. Just like in other languages F# classes can have attributes and methods of their own. [8] Here is an example of a basic class definition:

```

1  /* example of a class */
2
3  type PersonClass(inputFirstName: string , inputLastName: string , inputAge: int) as self
4      =
5      //The basic constructor
6      //A list of attributes within the class
7      let mutable firstName = inputFirstName
8      let mutable lastName = inputLastName
9      let mutable age = inputAge

```



```

10 //A do method that is implemented when the constructor is called.
11 do
12     printfn "New Person Recorded: \n"
13     self.PrintPerson()
14
15 //Constructor for a PersonClass that takes no arguments.
16 new() = PersonClass("John", "Doe", 0)
17
18 //Class methods follow below.
19 member this.PrintPerson() =
20     printfn "Name: %s, %s Age: %d \n" lastName firstName age
21
22 member this.getName() =
23     lastName + ", " + firstName
24
25 member this.setName(inputFirstName: string, inputLastName: string) =
26     lastName <- inputLastName
27     firstName <- inputFirstName
28
29 member this.getAge() =
30     age
31
32 member this.hadBirthday() =
33     age <- age + 1
34

```

As can be seen above, there are many similarities with F# and other programming language classes. The keyword *self* is used to refer to the specific instance of the object. The keyword *member* is slightly different from other languages. *member* is used to attach the functions to the class. The above example shows how a class can have attributes and associated functions much like in other programming languages. [8]

## 6 Interesting lexical feature

F# provides a special data type which is called *option type*. The official documentation states, "the option type in F# is used when an actual value might not exist for a named value or variable." [9]. Option type is defined with two cases, *Some* and *None*. When the value exists, it will return the value with *Some* type. Otherwise, it will return *<null>*. Option type is able to reduce the work for dealing with exceptions. In some cases, the code may be short and clean when dealing with exception. For example, *List.choose* has two parts, given higher ordering function and the input list. It will iterate over every element in the given list and return a list that contains elements where the given function returns *Some*. In other words, *List.choose* is similar to *List.filter*, and it will filter out the elements where the given function returns *<null>*. Therefore, option type will reduce huge amounts of work for dealing with exceptions.

```

1 /* example of option type */
2
3 let div m n =
4     match n with
5     | 0 -> None
6     | _ -> Some(m/n)
7
8 printfn "%A" (div 10 0)
9 printfn "%A" (div 10 5)
10
11 // Output:
12 <null>
13 Some 4

```

## 7 Summary

F# has many similar characteristics to other functional and object-oriented languages. F#'s types include common types such as lists, tuples, integers, floats, and many other types found in a variety of languages. F#'s control structures include if-elif-else statements along with common looping structures. Where F# starts to differ is in its subprograms. F#'s subprograms are unique for a functional programming languages. F# supports objects, which most functional programming languages do not include. F# has characteristics of both an object-oriented language and a functional language, allowing it to integrate easily into Microsoft's .NET framework. Microsoft built a highly versatile language with F#.

## References

- [1] Dave Fancher. *Book of F#: Breaking Free with Managed Functional Programming*. San Francisco: No Starch Press, Incorporated, 1 edition, 2014.
- [2] Wikipedia contributors. F sharp (programming language). [https://en.wikipedia.org/w/index.php?title=F\\_Sharp\\_\(programming\\_language\)&oldid=939687212](https://en.wikipedia.org/w/index.php?title=F_Sharp_(programming_language)&oldid=939687212), February 2020.
- [3] Chris Smith. *Programming F# 3.0*. O'Reilly Media, Inc, 2 edition, October 2012.
- [4] tutorialspoint. F# - data types. [https://www.tutorialspoint.com/fsharp/fsharp\\_data\\_types.htm](https://www.tutorialspoint.com/fsharp/fsharp_data_types.htm).
- [5] Microsoft. F# documentation - match expressions. <https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/match-expressions>.
- [6] Bob Myers. Control structures - intro, selection. <https://www.cs.fsu.edu/~myers/c++/notes/control1.html>, 2010.
- [7] Jeffrey Elkner, Allen B. Downey, and Chris Meyers. How to think like a computer scientist. <http://ice-web.cc.gatech.edu/ce21/1/static/thinkcspy/toc.html>.
- [8] ScottW. Classes. <https://fsharpforfunandprofit.com/posts/classes/>.
- [9] Microsoft. F# documentation - options. <https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/options>.