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 21, 2020

ABSTRACT

This section will be abstract.

1 Introduction

This section will be introduction.

2 History

F# was originally developed at Microsoft Research [1],and the first version was released in May2005 [2]. F# is a functional-first, multi-paradigm language. T he syntax o f F# provides a friendly way to w rite 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. One of the most important features of the .NET Framework is to allow developers using one language to interact with other languages [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. 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. In the meanwhile, 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. it means that the type checking will be done at compile time. For instance, if we try to assign a String to a Integer type, we will receive a compile error.

```
// Compile Error
let a : int = 1.20F
printfn "a: %d" a

//error FS0001: This expression was expected to have type
//     'int'
//but here has type
//     'float32'
//compiler exit status 1
```

Table 1: Integer Data Types

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} 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"

Table 2: Arithmetic operators

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

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].

```
// Simple example of Data type and Arithmetic operators
// initialize variables
let a : float32 = 1.20F
let b : float32 = 2.21F
printfn "a: %f" a
printfn "b: %f" b
printfn "sum: %f" (a + b)
```

```

// output:
// a: 1.200000
// b: 2.210000
// sum: 3.410000
```

Immutability: When you read some F# documentations, it may be very hard to find the word, variable. In some programming languages, we are allowed to re-assign the new value to the previous variable. However, in F#, 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 key word ***"mutable"*** to change the state.

```
// example of key word mutable
let mutable x : int = 10
x <- 15
printfn "x: %d" x

// output:
// x: 15
```

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

```
// example of Tuple

let names = ("John", 23)
printfn "name: %s" (fst names)
printfn "age: %d" (snd names)

// output:
// name: John
// age: 23
```

List: A list in F is an ordered, immutable series of elements of the same type [4]. The syntax is using semicolon-delimited to separate each element in a closed brackets. The empty list is denoted as []. In addition, list in F# supports list comprehension syntax.

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

Table 3: Some built-in List functions

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.

```
// Examples:
// From: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/lists
let list1 = [ 1; 2; 3 ]

printfn "list1.IsEmpty is %b" (list1.IsEmpty)
printfn "list1.Length is %d" (list1.Length)
printfn "list1.Head is %d" (list1.Head)
```

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 discussions via test expressions. There are three types of discussion modes that can be performed via control structures, ***Sequential mode***, ***Selection mode***, and ***Repetition mode*** [5]. 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 [6]. Selection mode differs from sequential mode by choosing between two or more alternative paths [5]. The final discussion mode type, repetition, is used for repeating a piece of code multiple times [5].

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 source code and sequentially runs the next statement. The following example demonstrates the sequential execution:

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

// output:
// 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 given code 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 ture, so I ran."
 7
 8 printfn "Hello, I am here now."
 9
10 // output:
11 // conditional expression was ture, 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.

```
/* example of if-then-else statement */

let x : int = 3

if x % 2 = 1 then
    printfn "x is odd."
else
    printfn "x is even."

// output:
// 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 it own test expression.

```
/* example of if-then-elif-else */

//find_gpa : char -> float
let find_gpa (x : char) =
    if   x = 'A' then 4.0
    elif x = 'B' then 3.0
    elif x = 'C' then 2.0
    elif x = 'F' then 0.0
    else -1.0
printfn "GPA is = %.2f" (find_gpa 'A')
printfn "GPA is = %.2f" (find_gpa 'B')
printfn "GPA is = %.2f" (find_gpa 'C')
printfn "GPA is = %.2f" (find_gpa 'F')
printfn "GPA is = %.2f" (find_gpa 'Z')

// output:
// GPA is = 4.00
// GPA is = 3.00
// GPA is = 2.00
// GPA is = 0.00
// 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."[4]. The syntax of match expressions and pattern matching are very similar to Haskell.

```
// from: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/match-
    expressions#guards-on-patterns

// Match expression.
match test-expression with
| pattern1 [ when condition ] -> result-expression1
| pattern2 [ when condition ] -> result-expression2
| ...

// Pattern matching function.
function
| pattern1 [ when condition ] -> result-expression1
| pattern2 [ when condition ] -> result-expression2
| ...
```

4.3 Repetition mode

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. 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.

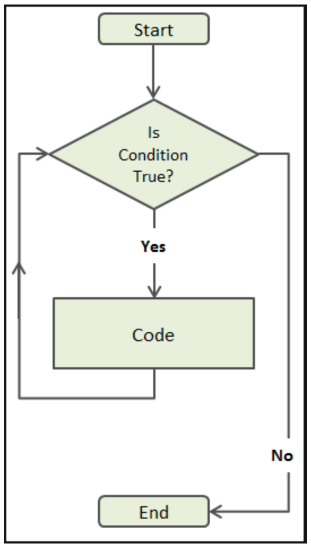

Figure 1: Repetition Control Structure

```
// for...to
for var = start-expr to end-expr do
    ... // loop body

//for...downto
for var = start-expr downto end-expr do
    ... // loop body

//for...in
for pattern in enumerable-expression do
    body-expression

//while-loop
while condition-expression do
    body-expression
```

```
/* example of for...to */
/* print number from 1 to 5 */

let for_to (x : int) =
     for i = x to 5 do
          printfn "i = %d" i

// call function for_to
for_to 1

/* example of for...downto */
/* print number from 5 to 1 */
let for_downto (x : int) =
     for i = x downto 1 do
          printfn "i = %d" i

// call function for_downto
for_downto 5

/* example of for...in */
/* traversing the list from the first element to the last element */
let list1 = [ 2; 4; 6; 8; 10 ]
for i in list1 do
```

```
    printfn "%d" i

/* example of while-do loop */
/* find sum of 1,2,3,4,5 */
let mutable i = 1
let mutable sum = 0
while (i <= 5) do
    sum <- sum + i
    i <- i + 1
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 that 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:

```
/* example of a function */
let func1 num1 =          // function declaration along with its parameter num1
    num1 + 3              // The return statement of the function is everything after
        the equal sign.

printfn "%d \n" (func1 4)

// output:
// 7
```

subsectionModules

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 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. [4] Here is an example of a internal module.

```
/* example of a internal module */
/* All in file internal.fs
module internal             //Whole file is module internal

module example1 =           //Module name declared as example1 (also nested within
    internal module)
     let value1  = 25       //Module holds a value within it
     let func1 num1 =       //Module also hold a function within it
         num1 + 3

/* A few ways to access the code from the internal module
Using qualifying names.
let modVal1 = example1.value1   //Using qualifying names, or the name of the module.
```

```
printfn "%d \n" modVal1          //Output: 25

open example1                    //Opening the module instead of using qualifying names
    .
let modVal2 = func 3
printfn "%d \n" modVal2          //Output: 6
```

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

```
/* example of a external module */
/* All in file external.fs

printfn "%d \n" internal.example1.value1     //Must call all of the nested modules to
    get to value
                                             //Output: 25
open internal                                //Opened the file's module
printfn "%d \n" example1.value1              //Output: 25

open internal.example1                       //Opened the internal module in the
    external module
printfn "%d \n" value1                       //Output: 25
```

One other thing to note about F# is namespace. 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. [4]

subsectionClasses

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. [#ForFunAndProfit] Here is an example of a basic class definition:

```
/* example of a class */

type PersonClass(inputFirstName: string, inputLastName: string, inputAge: int) as self
    =    //The basic constructor
    let mutable firstName = inputFirstName
                                                                   //A list of attributes within
    let mutable lastName = inputLastName
                                                                     //the class
    let mutable age = inputAge
    do
        printfn "New Person Recorded: \n"                      //A do method that is
            implemented when the contructor is
        self.PrintPerson()                                     //called.

    //Constructor for a PersonClass that takes no arguments.
    new() = PersonClass("John", "Doe", 0)

    //Class methods follow below.

    member this.PrintPerson() =
        printfn "Name: %s, %s  Age: %d \n" lastName firstName age

    member this.getName() =
        lastName + ", " + firstName

    member this.setName(inputFirstName: string, inputLastName: string) =
        lastName <- inputLastName
        firstName <- inputFirstName

    member this.getAge() =
        age

    member this.hadBirthday() =
        age <- age + 1
```

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. [#ForFunAndProfit]

6 Summary

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`, February 2020.

[3] Chris Smith. *Programming F# 3.0*. O'Reilly Media, Inc, 2 edition, October 2012.

[4] Microsoft. F# documentation. `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/match-expressions`.

[5] Bob Myers. Control structures - intro, selection. `https://www.cs.fsu.edu/~myers/c++/notes/control1.html`, 2010.

[6] 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`.

[7] ScottW. Classes. `https://fsharpforfunandprofit.com/posts/classes/`.