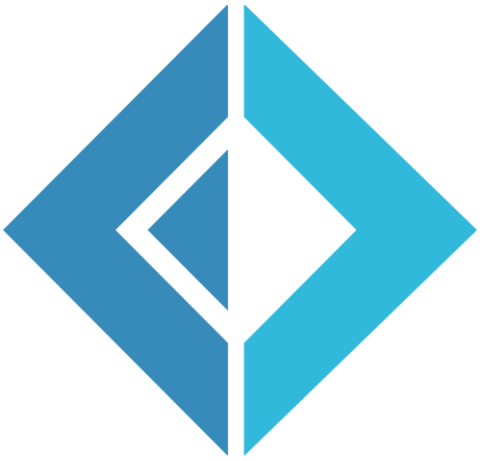

The History and Utility of the F# Language

Nathan Blankenship, Tucker Kane, Tatum Gray

Montana Technological University

CSCI 305: Concepts of Computer Programming Languages

October 01, 2021

Introduction

Functional Languages only make up a portion of all programming languages, however, these languages can be more practical to use. When compared to its contemporary languages, F# can execute the same logic in much less code. Our group decided to analyze F# for a few reasons. To get a better grasp of functional programming, in a framework that we are familiar with enough to point out the differences in structure. As F# uses the .NET framework, it shares a bit of its identity to C#, which, in turn, is similar in structure to programming in C or C++. This makes it familiar, but in such a way that differences are noticeable. Also, it is not an overly popular language, giving us an opportunity to learn a niche language. You never know when learning a functional language in the .NET framework will pay dividends, but you should be prepared for it when it happens.

History

Don Syme, a PHD Graduate from Cambridge, had been contacted by Microsoft Research in 1998 to develop a new Language for their .NET framework. Taking inspiration from his most preferable languages, Basic, Prolog, Scheme, and C as strong typed, he wished to use this opportunity to create a functional language (Syme, 2020, p. 75:10). Initially the project intended to port over the Haskell language to .NET, which was developed up until 2000. The team he was working with made advancement in developing this idea but ended up scraping the project. There was a problem with interweaving Haskell and .NET. To continue with the project, the Haskell language would need reworking to cooperate with the architecture. Types, new libraries, exceptions etc., to get it fully interoperable. Since the idea was to create an integrated language and fears stemming from the Haskell community about their language losing its integrity after being remade, the project was dropped (Syme, 2020, p. 75:13).

From the failed “Haskell.NET”, gave way to what we come to know as F#. Syme still wanted to achieve the goal of making a functional .NET language. This time he took inspiration from the OCaml language. Using its rule structure as a basis for development, Syme and his team started development in late 2001. With this, it would remain under exclusive Microsoft development until 2014.

Pros and Intended Uses

F#’s purpose in the programming world, is to provide a well-rounded functional language to expand the .NET framework with C# being in development, (software engineering stack exchange) that is strongly typed and prevents C++’s hierarchical classes. On a more personal scale, Don Syme, the lead developer of the F# project, started feeling nostalgic for strongly typed languages of Prolog, Scheme, C, and Basic after using C++ in grad school. He envisioned a new strongly typed language that uses OCaml as a base in design.

F# was originally only available on Windows, but later adapted to be available on several devices. F# was the first functional programming language that could also be used for scripting purposes. Though F# was developed for a wide variety of purposes, it is typically used for data and scientific analysis. The language being written in such an algorithmic way assists in computing their data driven equations. A large advantage of F# being a strongly typed language, is that it tends to catch more bugs at compile time rather than runtime, which is convenient for debugging and catching simple syntax errors. F# has many strengths, and strongly represents the ““Five Cs”- conciseness, convenience, correctness, concurrency, and completeness.” (Cartermp) as well as many other characteristics that define the language. The language has become increasingly popular because of its compatibility with all Microsoft products. F# is convenient because it does not have the clutter of semi colons and parentheses like many C languages, and it

often takes fewer lines of code to solve complex problem when compared to languages such as C#. Overall, F# code is simple and is not particularly picky about syntax.

Drawbacks

Despite the overall effectiveness of the language, F# still has its fair share of snags. For instance, during development of the language, because it was not open source until 2014, if a developer left the project, their efforts towards the language ceased (Syme, 2020, 75:51). They couldn't continue to work on F# as they were essentially locked out of development. In conjunction, the team also lost an experienced developer in the process (Syme, 2020, p. 75:51-52). Even with a full development team, they also need to account for what the C# team is doing. Because they both operate on the .NET framework, new features must be supported as they arrive. This has the side effect of having already implemented features being made redundant when C# decides to implement something similar. For instance, F# features the Async<T> keyword that allows a function to run asynchronously from the rest of the program. Later, C# would implement a Task<T>, which allows for multi- threading functions (Syme, 2020, p. 75:51-52). Both essentially do the same operation, but F# needed to account for it anyway. Just the same, the complexity of development increases when creating a new feature for F#. If it were to be added, it would have to be then added to C#, which isn't guaranteed to be simple (Syme, 2020, p. 75:51-52). Feature planning then needs to account for the broader scope of the framework of .NET rather than simply focusing on its own development.

Bugs are almost certain to pop up, even within the robustness of F#. For instance, the statically resolved type parameter, intended for operator overloading, when used for locking down types in functions can lead to 'corner case problems in code that are hard to resolve' (Syme, 2020, p. 75:51). F# also features generic comparisons that have performance issues and

can throw an error when comparing NaN with floating points (Syme, 2020, 75:51). F# may feature back-piping, but it does not allow for an iterative sequence of piping on one line. This makes code that does feature it more difficult to read but isn't much of a concern as conventions of regular piping in sequence prevent users from acknowledging this error (Syme, 2020, p. 75:52).

There is also some anecdotal evidence of problems from users. It may be easy to code, but after the logic of the code leaves short term memory, it can be hard for a human to parse what that logic was, especially without the need to define types. Even then, the compiler may not tell you exactly where the errors are in the code, making it a bit difficult to debug (DanManPanther, 2020). By using F#, you aren't getting much technical benefit (McCaffery, 2015). Which implies that the front end of coding the language might be preferable, but the backend doesn't make the operations any more efficient. Because its user base isn't that of more popular languages, issues during development aren't guaranteed to be provided a solution from internet crowd sourcing. F# does support a few dedicated communities, such as r/fsharp on reddit, but getting a unique answer for a unique solution will take time. This is a problem that languages with larger user bases rarely have. Granted the community has surely grown since McCaffery made his point. However, it's still useful in advising caution when learning a unique programming language. Especially as this problem may only present itself after already committing to the language (McCaffery, 2015).

Syntax

Comments

There are two ways to comment in F#. Like C++ you can do single line comments by simply putting "//" before the comment. Multiple line comments are started with "(*" and ended with "*)" and can encapsulate multiple lines of unwanted code or lengthy comments.

Assignment

Assignment in F# is delineated by a "let" statement followed by variable name then "=" and the value of the variable you are assigning (ex. let a = 2). By default, F# does not allow reassignment, but if you create a mutable variable then a value can be resigned to that variable (ex. Let mutable a = 2).

Operators

Most all arithmetic operators are the same from language to language, and F# is no exception in this category. The typical operators +, -, *, /, and % are all the same to typical languages in that they are used for addition, subtraction, multiplication, division, and calculating the remainder. The only arithmetic operator that differs from standard languages is the power or exponent operator, which is signified by ** rather than a ^ like in many programming languages.

F# comparison operators are very similar to many other programming languages. The operators < and > compute less than and greater than operations. The operator <= will return true if the right side of the equation is greater than or equal to the left side of the equation and will return false if not, and the operator >= will return true if the left side of the equation is greater than or equal to the right side and false if not. The = operator returns true if the left and right side of the equation is equal. The <> operator returns false if the left and right side are not equal and returns true if the two sides are equal.

There are two Boolean operations in F#, which are Boolean AND and Boolean OR. Boolean AND is signified by && and Boolean OR is signified by ||, these Boolean operations are the same in most programming languages.

All bitwise operators in F# are accompanied in pairs of three symbols. For example, the operation for bitwise AND is &&&. The operator for bit shift left is <<< and for bit shift right is >>>. The operator for bitwise OR is "^^^" and exclusive OR is |||. The operator for bitwise NOT is represented by ~~~.

Piping can be done in two ways. Using |> or, <|. These are utilized to allow functions, variable, etc. To operate on the value directionally piped into it. The back-piping character is not recommended as it can't be used in iteration where the forward-piping character can.

F# isn't a language that uses the off-side-rule to determine structure but spacing between arithmetic and comparisons is required to avoid errors.

Loops and Conditional Statements

In most languages the only loops used are "for" and "while" loops and the same stands for programming in F#. There is also the capability of using nested loops in F# to encompass loops inside of each other. All loop statements are ended with a "do" to express that if the condition is met, then do the operation until the condition is false (ex. for x in list do).

F# differs from most programming languages because "if else" is treated as an expression rather than a statement. The syntax of an "if" expression is typically an "if" followed by a then to delineate what should happen if the expression returns true, followed by an "else" if the if expression is found to be false or "else if" if there is another condition to be tested. Like some other programming languages, "elif" can be used as a substitute for an "else if".

Recurrence

The keyword “rec” is used to identify recursion within a functions body, this keyword allows a “let” function to be bound to the recursive methods. All functions that are recursive and bound by let are required to have the keyword “rec” in order to be recursive. Though recursion can still be done implicitly through classes without the need of the “rec” keyword.

Output

There are two simple functions for outputting to the terminal in F#. To print data to the terminal simply use a “printf” call followed by whatever string you would like to output. To start a new line after you have printed your desired statement, use “printfn” followed by whatever you would like to output. When printing desired values, you use a % followed by a specified character that is dependent on the return type of your desired output, and this statement should be encapsulated in the form of a string (ex. printfn(“This is the integer a %b”) boolvalue). Below is a table that contains the syntax for returning every desired return type.

Type	Description
%b	Formats a **bool,** formatted as **true** or **false**.
%c	Formats a character.
%s	Formats a **string,** formatted as its contents, without interpreting any escape characters.
%d, %i	Formats any basic integer type formatted as a decimal integer, signed if the basic integer type is signed.
%u	Formats any basic integer type formatted as an unsigned decimal integer.
%x	Formats any basic integer type formatted as an unsigned hexadecimal integer, using lowercase letters a through f.
%X	Formats any basic integer type formatted as an unsigned hexadecimal integer, using uppercase letters A through F.
%o	Formats any basic integer type formatted as an unsigned octal integer.
%e, %E, %f, %F, %g, %G	Formats any basic floating point type **(float, float32)** formatted using a C-style floating point format specifications.
%e, %E	Formats a signed value having the form [-]d.dddde[sign]ddd where d is a single decimal digit, dddd is one or more decimal digits, ddd is exactly three decimal digits, and sign is + or -.
%f	Formats a signed value having the form [-]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.

%g, %G	Formats a signed value printed in f or e format, whichever is more compact for the given value and precision.
%M	Formats a Decimal value.
%O	Formats any value, printed by boxing the object and using its **ToString** method.
%A, %+A	Formats any value, printed with the default layout settings. Use %+A to print the structure of discriminated unions with internal and private representations.
%a	A general format specifier, requires two arguments. The first argument is a function which accepts two arguments: first, a context parameter of the appropriate type for the given formatting function (for example, a TextWriter), and second, a value to print and which either outputs or returns appropriate text. The second argument is the particular value to print.
%t	A general format specifier, requires one argument: a function which accepts a context parameter of the appropriate type for the given formatting function (aTextWriter) and which either outputs or returns appropriate text. Basic integer types are **byte, sbyte, int16, uint16, int32, uint32, int64, uint64, nativeint,** and **unativeint.** Basic floating point types are **float** and **float32.**

Table from: F# - basics

In Summary

Even though F# may not be the most impressive language to program in, it offers a fluid workflow. Cutting down on lines of code speeds up the process and gets programs working much faster. It keeps a simple syntax that is easy to follow. Being non-null and strongly typed ensures good coding practices. As a language, it may not be suitable for someone's first experience programming, as it is not representative of the current landscape, but for a matured user it trims a lot of the fat that takes time away from finishing a project. Granted that project might have to be specialized toward scientific or data driven analysis to make full use of the language. Yet despite garnering a smaller coding community, a few technical bugs here and there, and personal taste in programming languages, it can be worthwhile language to learn. Given its influences in OCaml and a slew of strong typed languages, the next iteration upon F# should have a great chance at persuading the public about the utility of functional programming.

References

Cartermp. (n.d.). *Symbol and operator reference - F#*. Symbol and Operator Reference - F# | Microsoft Docs. Retrieved September 29, 2021, from https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/symbol-and-operator-reference/.

DanManPanther (2020, May 31). *Reasons NOT to Use F# for a Web App?* Online forum post. https://www.reddit.com/r/fsharp/comments/gu5ln8/reasons_not_to_use_f_for_a_web_app/

Don Syme. (June 2020). *The Early History of F#*. Proc. ACM Program. Lang. 4, HOPL, Article 75, 58 pages. https://doi.org/10.1145/3386325

F# - basic syntax. (n.d.). Retrieved September 29, 2021, from https://www.tutorialspoint.com/fsharp/fsharp_basic_syntax.htm.

Heller, M. (2018, April 23). *14 excellent reasons to use F#*. InfoWorld. Retrieved September 29, 2021, from https://www.infoworld.com/article/3269057/14-excellent-reasons-to-use-f-sharp.html#:~:text=F%23%20is%20for%20scripting.%20F%23%20supports%20functional%20programming,way%20of%20the%20partial%20application%20of%20function%20arguments.

McCaffery, J. D. (2015, March 1). *Why I Don't Like the F# Language*. WordPress. Retrieved September 29, 2021, from https://jamesmccaffrey.wordpress.com/2015/03/01/why-i-dont-like-the-f-language/.

Wikimedia Foundation. (2021, August 10). *F sharp (programming language)*. Wikipedia. Retrieved September 28, 2021, from https://en.wikipedia.org/wiki/F_Sharp_(programming_language)