Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Final Report 2019

Project Title:	**F# Style and "Probable Error" Analysers for Ionide**
Student:	**Louis Kueh Han Sheng**
CID:	**01078797**
Course:	**EIE4**
Project Supervisor:	**Dr Thomas Clarke**
Second Marker:	**Dr Edward Stott**

Contents

1. Abstract

There are many tools utilised by programmers to improve the quality and ease of writing code. However, few tools deal with tackling ambiguous errors that often confuse novice programmers. Short of building another compiler, the techniques utilised for tackling these ambiguous errors may be carried over to other programming languages. This project investigates ambiguous errors of the functional language F# and exploits heuristics drawn from the abstract syntax tree to clarify these ambiguous errors. To achieve this, real-time code analysers have been developed to evaluate code. These analysers are plugged into a popular extension for Visual Studio Code, Ionide, to run in conjunction with everyday work-flows. Through the usage of the analysers, users will have a clearer understanding of ambiguous errors that will help improve learning and efficiency. For example, by considering contextual information such as function names and the number of parameters, the analyser can prompt the user with an error message of greater insight. This project may serve as a new class of tools which improves error messages through determined heuristics.

2. Introduction

Programmers utilise many tools in their work. There are many projects that help the programmer improve their code, be it in readability or maintainability. However, there is a lack of tools that help the programmer tackle obscure errors, especially for novice programmers. Couple this together with learning a new way of thinking, such as learning a functional language, beginners have a steep road to climb in order to achieve mastery. This project aims to tackle these confusing errors and provide improved suggestions to better correct the code. This is achieved by writing code that takes in compiler information alongside the context of what is around the code and activates once certain criteria are met. How to best utilise this information from the compiler to provide better error messages is what this project investigates. While this project has a working tool developed to demonstrate potential corrections, it is not intended for the tool to be the main focus. Rather, this project investigates the potential of these improved corrections which can ideally be applied to other programming languages as well.

The background section will cover the reasons why this project was undertaken as well as other similar existing work. This will be followed by the design of the tool and the actual implementation, giving the reader an idea of how a similar project might be implemented. Next, the functional correctness of the tool will be demonstrated in the testing section and the results found will be presented. In the evaluation section, the features/ideas that the tool implements will be analysed according to how well it works. In this case, this will be whether the ideas that the tool implements actually help the programmer in the event of obscure errors.

3. Aim

The aim of this project is to develop analysers for F# language that have improved error messages over current tools and provide possible contextual corrections targeted at new programmers. These analysers will serve as plugins to the Ionide Extension for VS Code[1] and will offer suggestions in real time.

There are many tools targeted at improving code quality. Linters such as HLint[2] and code analysis tools such as sonarQube[3] come under this category. The majority of tools deal with stylistic errors and require a tool to be run or build first. Fewer use contextual information to suggest corrections specific to the user and even fewer are able to perform this in real time.

Real-time analysers for F# are made possible with the FSharp Compiler Service, exposing APIs for particular sections of the F# compiler. These include the syntax tree, hierarchical symbol table as well as the binding phase that exposes the compiler's semantic analysis of the code.

4. Background

This project will utilise the functional language F# to demonstrate these improved error correction methods. This will be achieved primarily through custom Analysers, which are real-time plugins that can produce hints and suggestions as the user writes their code. Currently, Ionide, the most popular F# plugin for Visual Studio Code, has support for custom analysers to be built and run alongside Ionide. Custom analysers have been developed to demonstrate improved error correction for F#.

4.1. What is F#

F# is a strongly typed, functional programming language designed by Don Syme, Microsoft Research and developed by Microsoft, The F# Software Foundation in 2005.[4] Built on the .NET Framework, F# offers the opportunity for programmers/companies used to traditional procedural languages to experiment with functional programming by allowing them to write parts of the program in the same ecosystem, the .NET language (e.g. C#), and a smaller section in F# to start with. This can be a low-risk method to encourage incremental usage of F# in .NET platforms, for example in production systems[5]. Big companies, such as Credit Suisse or Microsoft are advocates for F#.

4.2. Analysers

Analysers are plugins that can diagnose source code to produce custom error messages and suggested features. A similar project, Roslyn Analysers for the C# compiler Roslyn, already has multitudes of custom analysers made by the open source community[6]. The idea behind these analysers is that compilers have a wealth of information that can be used to improve the user's experience. This can be in the form of Intellisense, smart re-factoring and other tools to help the user write and understand the code. The core mission from Rosyln is as follows: "opening up the (compiler) black boxes and allowing tools and end users to share in the wealth of information compilers have about our code"[7].

A key component of analysers is that they work in real time. They take in the information the compiler currently has from the code, for example by walking through the Abstract Syntax Tree (AST) as the user types the code. This can be enabled through a compiler API, which exposes syntactic and semantic information through each stage of the compiler pipeline. Further details will be covered in the technical section later on.

The purpose of analysers is to give the user essential information as quickly as possible, enabling the user to correct the code based on their judgement. As analysers are project-based plugins, they can be modularized and distributed through package distribution networks such as NuGet or npm. [8]

Many languages have tools that make use of analysers, by far the most popular of these tools are named Linters. These generally help the programmer improve code readability. However, in some cases, they also provide some insight into why and how the code is structured the way it is [9].

4.3. Context

Many tools exist to help developers increase productivity [10]. Source control (git) enables developers to fall back to the old, working code in times of regression or bugs. The same source control can help developers work effectively in a team by introducing branches, whereby separate features can be developed simultaneously and combined later on. Project management tools (JIRA, Trello) help developers track issues that need to be resolved and can be ideal for agile teams. Quality assurance tools can come in many forms. For example, static code analysis tools such as sonar-

Qube help detect flaws in the code to improve its quality [3]. Travis CI is a continuous integration tool that works with many source control tools. Test suites are automatically run whenever a new push to a git branch is done [11]. This ensures that any regression in functionality will be detected as soon as possible and the developer can revert back to an older version.

Integrated Development Environments (IDE) are development environments for programmers, the primary area where the code is written. Examples include XCode, Eclipse or Visual Studio Code. These editors offer many features for the developer such as Intellisense, Autocompletion and git integration built right into the editor [12]. In the case of Visual Studio Code, developers can download extensions from the marketplace which offer additional functionality created by the open source community [13]. As such, there is a wide variety of tools and many of these extensions are tools that help increase productivity even more.

For example, Markdown All in One is a tool that introduces shortcuts, latex support and enables conversion of markdown files to HTML within the editor [14]. Other tools deal with improving the code quality of developers. Extensions such as Ionide for the F# language, enable usage of F# interactive to run code within the editor, display type signatures of functions and variables as well as error highlight for F# [1]. These help programmers easily identify where faults in the program might be. Similar to sonarQube, Ionide achieves error highlighting through the usage of Analysers, which will be covered in the section below.

While sonarQube requires the user to run the program against their code, Ionide enables the user to view error messages in real time without the user building the code. This is partly due to the available compiler APIs for F#. These APIs also enable Ionide to interact with the F# abstract syntax tree supporting custom made analysers which form the basis of this project [15]. Utilising this information allows investigation of a particular error and potential improvements to be made.

4.4. Motivation

There are many tools that help programmers write better code. Specifically, Linters are a class of analysers that are most similar to this project. They differ in the sense that they deal with formatting, how to best write your code to follow certain guidelines in terms of presentation and look. Instead, this project helps improve error messages but does not deal with the formatting of the code.

Users of F# (especially beginners), Open source community members or even the team behind the F# would appreciate such a tool. Several heuristics of error messages can prove to be useful not only to users of F# but may also carry weight across different languages. This can introduce a new class of tools to help distinguish itself from existing ones.

5. Similar Projects

Procedural languages are more popular than functional languages, with only one functional language (Scala) in the top 15 languages on GitHub [16]. Many projects [17] similar to analysers have already been done for these languages.

Functional languages are very different from procedural ones. Many of the features of analysers used for popular procedural languages do not apply to functional languages. Therefore, more emphasis will be placed on similar works for functional languages.

5.1. FSharpLint (F#)

FSharpLint is a linting tool that analyses F# code looking for any code that does not adhere to rules implemented. FSharpLint focuses primarily on improving code readability[18], with analysers/rules in areas such as naming conventions, length in sections of code, Typography (e.g. maximum number of characters on a line), number of nested statements and parameters to a function. These help to improve the code's readability but do not necessarily improve the user's understanding of the code. FSharpLint also has a FunctionReimplementation analyser that replaces lambda functions with their equivalent in Figure 1. This enables the user to understand when not to use simple lambda functions and their better alternatives.

```
fun x y -> x + y  is equivalent to  (+)
```

Figure 1. Example function implementation

Out of the nine analysers, only one of these can be considered to improve the user's understanding of the code. As such, there is a large scope for improvement in this area, especially for F#.

5.1.1 Ionide

Ionide is a visual studio code extension developed for F# that includes many features such as auto-complete, intellisense as well as F# Interactive to enable developers to run code quickly within the editor.[1]

Currently, Ionide has FSharpLint integrated together with it, enabling analysers from FSharpLint to be used with Ionide. Ionide has also added support for custom analysers, allowing analysers to be plugged into the extension [15]. This serves as the basis for the project, whereby custom analysers can be used in conjunction with Ionide.

5.2. HLint (Haskell)

HLint is a linting tool for the functional language Haskell that makes suggestions for possible improvements to Haskell code. These can include ideas such as using alternative functions, simplifying code and spotting redundancies. HLint aims to teach the programmer better style and improve the code this way. [2].

Similar to FSharpLint, HLint has a large section on improving code readability - bracket reduction, improved imports, naming conventions and structure of the code. An example can be found in Figure 2.

```
if a then b else if c then d else e
-- | a = b ; | c = d ; | otherwise = e
```

Figure 2. HLint readability example

Similar to FSharpLint's FunctionReimplementation, HLint also has an analyser that removes all unnecessary lambdas. Unlike FSharpLint, HLint has an analyser to simplify list operations as well. Some examples can be seen in Figure 3.

```
[x] ++ xs -- x : xs
yes = y :: [Char] -> a -- String -> a
1:2:[] -- [1,2]
```

Figure 3. HLint simplified list operations

Additionally, list operations can also be simplified when applying common functions in Haskell such as map, foldr or foldl.

HLint does not deal with obscure errors but instead aims to improve readability. Users are able to learn different approaches which can improve their code readability by systematically applying changes one by one.

5.3. Scala

Scala is one of the most popular functional languages on Github [16]. This can be attributed to its hybrid functional language and object-oriented properties, alongside the ability to run on the Java Virtual Machine. Approximately 85% of computers use Java[19], enabling Java developers to slowly take up Scala. As a result, Scala has many projects in the area of linters and analysers.

Scapegoat [20] is a static code analyser for Scala. Differing from analysers, static code analysers need to be built alongside the code they are analysing. For example, changing a section of the code will require it to be built again in order to view up to date errors. The main reason for this is that there is no compiler API for analysers to receive essential information needed to run the analysers without running the whole compiler.

Scapegoat currently has 117 rules, and many are similar simplifications of code as seen in FSharpLint and HLint. However, Scapegoat also has several rules that do not only deal with code readability but help call the users attention to what otherwise might be obscure error messages. For example, checking for toString calls to arrays, which do not work in Scala. Or try-catch blocks that catch null pointer exceptions. These can help the programmer identify actual errors that could occur as opposed to simply stylistic choices.

Scalastyle is a similar linter to scapegoat but only deals with styling. Another linter, WartRemover, deals entirely with compiler errors and the possible workarounds to avoid confusing error messages. For example when checking equality in arrays "==" checks for reference equality unlike other collections in Scala. This will simply throw a Boolean and in most cases is not what the programmer intended (checking that each element is equal). Other examples include the compiler referencing a generic type (Any) when there are multiple types in a list, as shown in Figure 4.

```scala
val any = List(1, true, "three")
```

Figure 4. Scala WartRemover generic type Example

This is usually incorrect and WartRemover prevents this from occuring by throwing an error.

5.4. C#

Though considered a primarily Object Oriented Language, C# shares many similarities with F#. This stems from shared eco-system that is the .NET framework. The C# compiler, Roslyn, enables programmers to take advantage of information from the compiler to produce real-time code warnings even before finishing the line [21]. Roslyn does not have the analysers itself, rather it serves as a platform for others to create analysers. The most popular of the .NET Analysers is the StyleCopAnalyzers [22].

Similar to other linters StyleCopAnalyzers provide readability, spacing and layout rules. Additionally, there are rules that aim to improve code maintainability as well as special rules such as displaying invalid settings file and disabling of XML comments [23]. There are no rules dealing with actual compile errors.

5.5. Conclusions

It is clear that there are many projects in the same area. Many describe themselves as linters, often focusing on coding style and improving readability. Some projects, such as Scala's WartRemover deal with areas where possible errors might occur. In this case, WartRemover deals with potential errors - the compiler does not throw any error and the code does indeed compile, but the code will be semantically different from what the programmer meant.

There is a large scope for analysers designed for beginners, especially for a functional language such as F#. This is an area that has not been covered by projects before. All projects target code styles for the general purpose audience.

Most programmers start out with a procedural language and some move on to pick up a functional language. Given the rise in popularity of functional languages, F#'s compatibility with the .NET framework, it is expected that there will be a large demand for F# in the future.

6. Requirements capture

For this project, the tool developed is required to interface with the popular visual studio code extension Ionide. This is desired in order to gather feedback from the community as well as serve as testing grounds for possible methodologies. The most important aspect is that Ionide already provides a platform to develop custom analysers, and allowing the interaction between user and code to be offloaded to Ionide and Visual Studio Code. This enables the project to focus on investigating possible heuristics rather than building a platform to demonstrate such heuristics.

6.1. Basic Requirements

The tool developed will interface with Ionide through its custom analysers features. Investigation of possible heuristics will primarily take place based on the information that Ionide passes on to the analysers. The final deliverable should tackle misleading or ambiguous error messages that do not conform with the error within the code. A very simple example would be the case of missing brackets, the analyser should therefore at the minimum mention something about the brackets in the error messages. This idea of a clear message is, of course, dependant on the user. However, simply stating the correct error (e.g. missing bracket here, too many arguments in this function) instead of a generic default error (e.g. segmentation fault, line does not parse etc) can serve as a good starting point as a requirement. The analyser should also highlight the area of code where the problem lies, to ensure that it is clearly visible to the user. This may be in the form of underlines or highlighting the code.

6.1.1 Performance

The tool should ideally be able to run in real time without any noticeable lag, as the user types the code. Ionide can be seen as a harness in which the analysers are run, thus, external factors will affect the performance. However, optimisation such as caching should be considered in the event of poor performance.

6.2. Desired Requirements

To improve upon this, pointing out contextual information may serve as an improved version to further bolster understanding of the problem. For instance, a simple case would be to specify the exact function that is causing the error by pointing out the function name and the expected number of parameters. Alternatively, pointing out that a particular declaration statement (including the name) can help speed up the process of diagnosing errors. By utilising contextual information such as names in addition to line numbers the programmer has an increased probability of realising which section of the code contains the errors. This may not have as much impact in small files, but in programs that are long or spread across multiple files this can speed up the process of identification of errors.

7. Analysis and Design

Due to the large scope of the project, there are many different tools and features that could have been implemented in order to tackle numerous error messages. Many different error messages can be interpreted as ambiguous, differing from person to person. A systematic approach based on three criteria was utilised to approach this problem.

- Effectiveness of correction - Does the error clearly state the exact cause of the error and at the correct location? Is the error always clear or only 60% of the time?
- Likelihood of usage - Does the error only occur on extreme fringe cases? Is it an error that is likely to be made by all users?
- Difficulty - How difficult is it to implement within the time-frame of the project?

7.1. Error Criteria

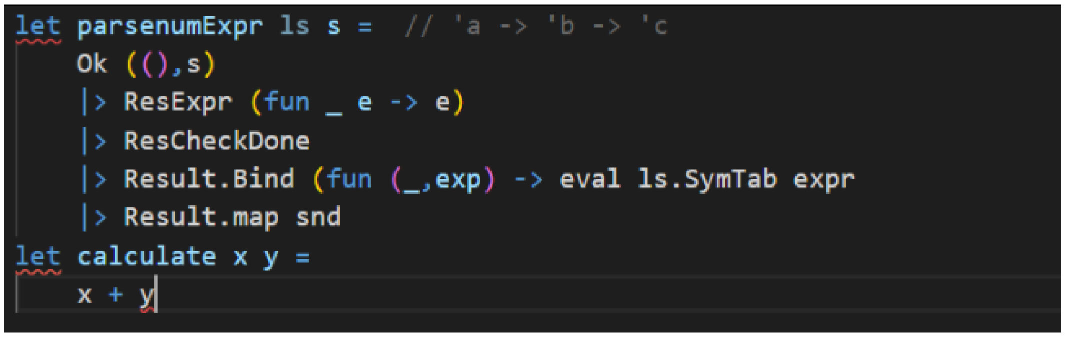
7.1.1 Effectiveness of correction

Error messages have varying effectiveness depending on the experience of the programmer. A very experienced programmer might not require any error messages at all, simply the correct location of the error might be enough to quickly fix the problem. On the other hand, novice programmers new to the language require more help in correcting the error. Having the exact cause of the error pinned down alongside the location can be the difference for the novice user to fix the code themselves. It is important to state the target audience for this experiment, novice programmers.

By targeting novice users, the analysers have a higher potential to be more effective. Ideally, with the usage of the analyser, the novice user will be able to correct code they would otherwise not be able to without the analyser. This has a higher impact on the user, as often experienced programmers know which error messages relate to which error based on experience, even if the message is obscure. As such effectiveness of correction may consider errors which may seem simple to the experienced user, but have a greater impact on novice users.

Within the effectiveness of correction, there are two criteria that are considered in determining whether the analyser built for the error would be worthwhile. Firstly, the clarity of the error message. This is important as there is no need to build an analyser if the error messages are clear and users can correct it themselves. Clarity of messages will be made by running code against the F# compiler without any analysers, and evaluating the error messages obtained. Though the clarity of an error message may differ from person to person, if the message states the exact cause of the error it is considered of good clarity. Secondly, the location of the error messages are the next criteria. Pointing out an error at the wrong location and the correct location can make a huge difference.

In order to demonstrate these criteria, an example will be evaluated according to the effectiveness of correction criteria.

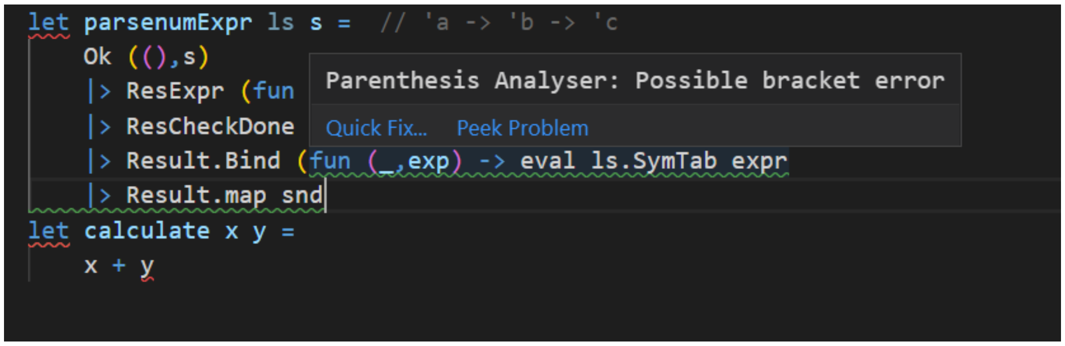


```
let parsenumExpr ls s =  // 'a -> 'b -> 'c
    Ok ((),s)
    |> ResExpr (fun _ e -> e)
    |> ResCheckDone
    |> Result.Bind (fun (_,exp) -> eval ls.SymTab expr
    |> Result.map snd
let calculate x y =
    x + y
```

Figure 5. Missing Bracket, No analyser

In Figure 5 there is an unmatched missing bracket on line 5 next to "(fun ..". Brackets have been highlighted in colour to easily identify the error. The error message from the F# compiler is "Unexpected keyword let or use in expression". Based on the effectiveness of correction criteria, the clarity of the error messages is unclear. The error stems from a missing bracket, not a let or use statement. Based on the message it may be difficult to correct the error. Note that the brackets are not colourised in default F# code, which makes it even harder to correct. Next, the error location is indicated by the red lines below the code. It is clear that the compiler has found the wrong location for the error. Combining both the clarity and location of the error messages, this example serves as an error which would have high impact when corrected.

In Figure 6 the Parenthesis Analyser is run, highlighting the exact location of the error and informing the user that there is a possible bracket error. This makes the error much clearer and improves the user's ability to correct the code when compared with the default "Unexpected keyword let or use in expression" error in the wrong location.



```
let parsenumExpr ls s =  // 'a -> 'b -> 'c
    Ok ((),s)
    |> ResExpr (fun
    |> ResCheckDone
    |> Result.Bind (fun (_,exp) -> eval ls.SymTab expr
    |> Result.map snd
let calculate x y =
    x + y
```

Parenthesis Analyser: Possible bracket error

Figure 6. Missing Bracket, With analyser

7.1.2 Likelihood of usage

The likelihood of usage considers the frequency of certain types of errors and can be quite difficult to consider. This criterion deals with the question of "If the error does not occur often, is it worth fixing?"

Given that the scope of usage encompasses all possible code, there are no statistics that demonstrate which functions that get used more than others. However, this criterion can be evaluated to a certain degree by considering the general case rather than specific cases. This means considering errors related to common paradigms found in F#, for example brackets, spacing, prefixes etc. in terms of syntax and let statements, if statements and function calls in terms of declarations. This can be contrasted with errors related to a specific usage of a library function. For example, a specific error arising from calling a function in a particular library right after another specific function call. While dealing with such errors does present some merit, it is more worthwhile to consider errors that have a higher frequency. These naturally lend themselves to errors relating to type errors often deriving from syntax errors.

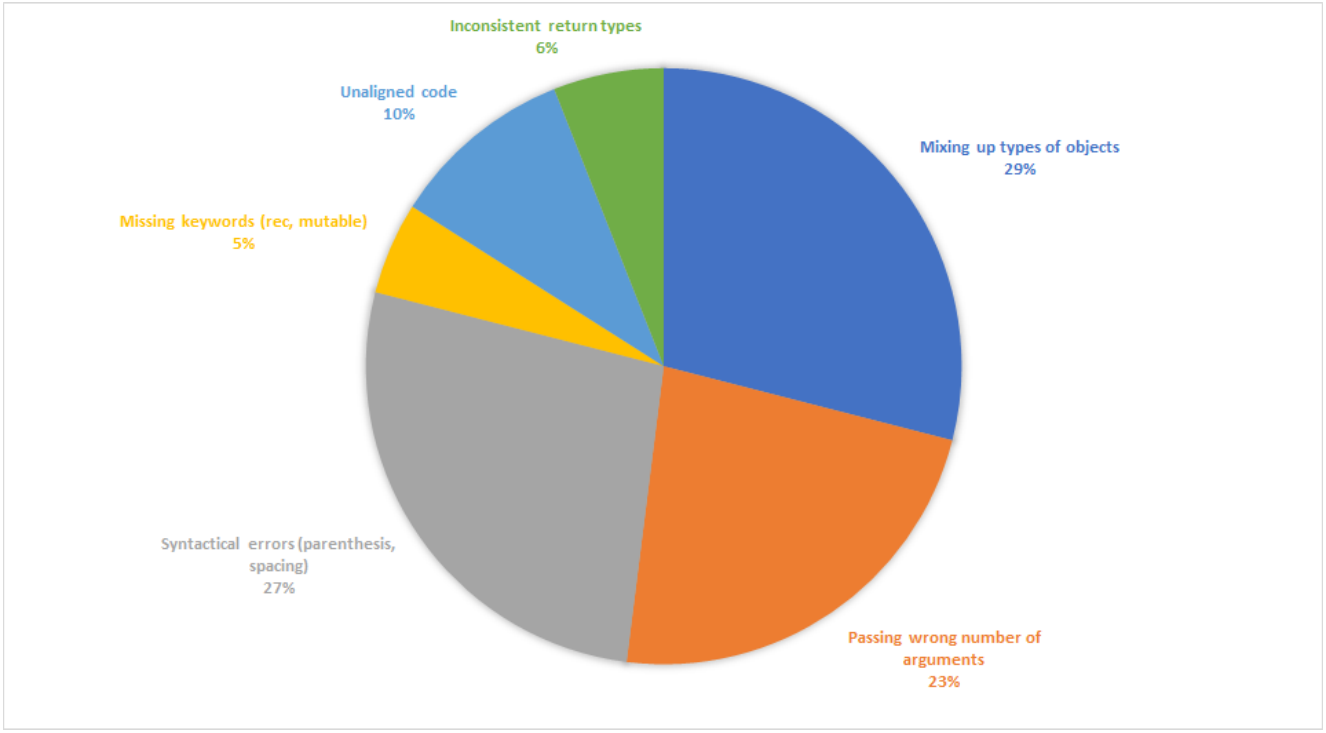


Figure 7. Personal recorded errors

Figure 7 shows the frequency of the type of errors recorded from self-learning F# as a novice user throughout the project; these are personal statistics that demonstrate roughly the frequency of certain types of errors. As the exact frequency of errors differs depending on the amount of code, percentages are used to demonstrate the most frequent error. These do not represent hard facts but serve as guidance to the general proportion of errors a novice user might expect to make. Data recorded is utilised by investigating the majority classes of errors encountered: types, arguments and syntactical errors. These form the basis of the three classes of analysers that were developed.

Further research includes the top 10 most common errors stated by popular website fsharpforfunandprofit.com [24] as well as community feedback on the most common errors [25]. These are compiled and discussed below in section 7.2.

7.1.3 Difficulty

The difficulty of implementing the analysers are an important factor to consider when choosing which feature to implement. The first step is to determine a feasible approach from available tools. As there is no set method of improving the error messages, an investigative path is taken. The first step is to evaluate current knowledge and available tools. As the project progressed, knowledge builds up to give insight into different methods to tackle the problem. From available tools, the syntax tree is obtained and compared with the error that is targeted. This is done by obtaining the syntax tree with source code containing the particular error, and comparing it with the syntax tree from source code without the error. This is often the hardest part in the process as the differences in syntax tree may not only be necessarily due to the error and requires trial and error to find key components that determine the error. If the error is determined to be caused by a particular part of the tree, the analyser can traverse the tree and pick out the relevant parts. If no suitable component is found, alternative approaches need to be undertaken. In these cases, parsing of characters can be utilised to generate another potential approach.

7.2. Examples of potential error fixes

In the previous section the three criteria for choosing which errors to tackle have been discussed. Examples of some of the most common errors novices experience FS0001[24] have been thoroughly investigated and evaluated. The examples below primarily consider effectiveness of correction and difficulty as all have a high likelihood of usage.

7.2.1 FS0001: The type 'X' does not match the type 'Y'

A thorough analysis of these errors will be investigated below to serve as an example of the decisions taken when deciding which rules will be included in the project.

This can be attributed to the strong type system of F#, which automatically infers types based on context. Yet F# does not require the programmer to specify types, for instance in a parameter list. While this reduces verbosity in F#, it does make it harder to navigate through compiler errors which can be seemingly confusing if the programmer does not fully understand the context behind F#'s types. Figure 1 below shows possible obscure errors and their fixes for FS0001.

At first glance, this compiler error can simply be attributed to mixing up object types. However, in reality, a number of factors can cause this error. For example, arguments can lead to types that do not match in cases where the compiler expects a function but a value is given. Or syntactical errors when using a tuple when none are expected.

A. Can't mix floats and ints

While the error message in Figure 8.

```
3 + 2.0 // wrong
// => error FS0001: The type
//'float' does not match the type

float 3 + 2.0 // corrrect
```

Figure 8. A. Clear Error message

seems to be quite clear, the issue can occur in more complicated cases (particularly library functions) where it is not clear why that is the case. This is shown in Figure 9.

Compiler Error	**Type of error**
The type 'float' does not match the type 'int'	A. Can't mix floats and ints
The type 'int' does not support any operators named 'DivideByInt'	A. Can't mix floats and ints
The type 'X' is not compatible with any of the types	B. Using the wrong numeric type.
This type (function type) does not match the type (simple type)	C. Passing too many arguments to a function.
This expression was expected to have (function type) but here has (simple type)	C. Passing too many arguments to a function.
This expression was expected to have (N part function) but here has (N-1 part function)	C. Passing too many arguments to a function.
This expression was expected to have (simple type) but here has (function type)	D. Passing too few arguments to a function.
This expression was expected to have (type) but here has (other type)	E. Straightforward type mismatch. F. Inconsistent returns in branches or matches. G. Watch out for type inference effects buried in a function.
Type mismatch. Expecting a (simple type) but given a (tuple type).	H. Have you used a comma instead of space or semicolon?
Type mismatch. Expecting a (tuple type) but given a (different tuple type).	I. Tuples must be the same type to be compared.
This expression was expected to have type 'a ref but here has type X	J. Don't use ! as the "not" operator.
The type (type) does not match the type (other type)	K. Operator precedence (especially functions and pipes).

Table 1. FS001 Errors and their fixes

```
[1..10] |> List.average   // wrong
   // => error FS0001: The type 'int'
   // does not support any operators
   // named 'DivideByInt'

[1..10] |> List.map float
|> List.average  //correct

[1..10] |> List.averageBy float
//correct (uses averageBy)
```

Figure 9. A. An obscure message via library fuction List.average

Though this error satisfies both effectiveness of correction and likelihood of usage, the difficulty criteria was not met. Simply detecting floats and integers is relatively easy. However, detecting exact function types and matching them to inputs proved challenging. Throughout investigation of the syntax trees no ideal solution was found and this error was not chosen for the analysers.

B. Using the wrong numeric type

When passing in the wrong type of object, the error message seems like it does not relate to the given code (Figure 10).

```
printfn "hello %i" 1.0
// should be a int not a float
// error FS0001: The type 'float'
// is not compatible with any of
// the types byte,int16,int32.
```

Figure 10. B. Relatively clear compiler message

A simple fix would be to cast the number 1.0 to an integer first, as seen in Figure 11.

```
printfn "hello %i" (int 1.0)
```

Figure 11. B. Fixed example

This error seems to be relatively straightforward especially if you are coming from a standard procedural programming background. It will not be included in the Analyser. This shows an example in which the errors would not be considered obscure.

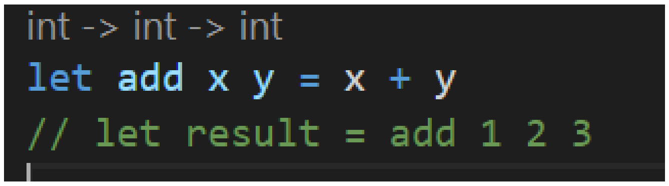
C. Passing too many arguments to a function

This error was very frequent during the personal experience. Running FSharp with Ionide: Notably the first function is being read as `int -> int -> (int -> obj)`due to the incorrect result line below.

```
// int -> int -> (int -> obj)
let add x y = x + y
let result = add 1 2 3
// ==> error FS0001: The type
//''a -> 'b' does not match the type 'int'
```

Figure 12. C. Error due to second line

Removing the second line gives Figure 13,



```
int -> int -> int
let add x y = x + y
// let result = add 1 2 3
```

Figure 13. C. Corrected after removing second line

which is the correct expression. Keeping the second line gives the error message shown in Figure 14.

```
// ==> error FS0001: The type
//''a -> 'b' does not match
// the type 'int'
```

Figure 14. C. Example of too many argument error

For beginner programmers, this is a particular error that does not seem to make much sense. From reading the error message, it seems as though one of the types of the argument is incorrect when in reality there are too many arguments. This is due to currying [26].

Breaking it down, the curried version is depicted in Figure 15.

```
let add x = // only 1 parameter
    let subFunction y =
    // x is in scope, no need to pass
        x + y
    // return the subfunction
    subFunction
// the `x` parameter is baked in
```

Figure 15. C. Example curried version of add x y

Passing in one argument to the function the ‘subFunction y‘ is returned (with x baked in). For the three parameter function `let add x y z = ...`, the curried function might be as follows below (Figure 16).

```
let add x = // 3 parameter version
    let subFunction y =
        let subFunction2 z =
            x + y + z
        // return the subfunction
        subFunction2
    subFunction
let result = add 1 2 3
```

Figure 16. C. Example three parameter curried function

Essentially, the compiler is expecting another function but only gets an integer in the function definition. The function is represented by the `'a -> 'b` signature in the error message. This error satisfies all three criteria: effectiveness of correction, likelihood of usage and difficulty.

D. Passing too few arguments to a function

Because of partial application in F#, functions with too few arguments will return a parameter baked in (Figure 17).

```
let reader = new -
- System.IO.StringReader("hello");
// no compiler error even
// though it is wrong!
let line = reader.ReadLine
// compiler error here
printfn "The line is %s" line

// ==> error FS0001: This expression
// was expected to have type string
// but here has type unit -> string
```

Figure 17. D. Missing unit parameter

This error is hard to correct as the error message is rather obscure. Many .NET library functions require the unit parameter `let line = reader.ReadLine()`.

While this error satisfies the effectiveness of correction and difficulty criteria, the likelihood of usage can be debated. The usage of .NET library functions does not apply to all projects, and while it is feasible to develop an analyser it may not have as much impact as other errors.

E. Straightforward type mismatch

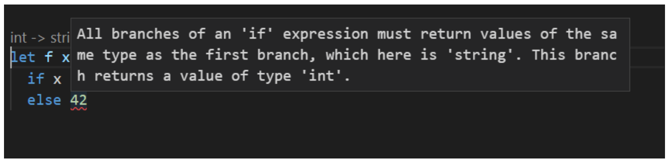
This error message is straightforward and novice programmers can see that a variable type is wrong. As such this will not be included in the analyser.

```
printfn "hello %s" 2
// => error FS0001: This expression
// was expected to have type string
// but here has type int
```

Figure 18. E. Straightforward type mismatch

F. Inconsistent return types in branches or matches

Every branch/match must return the same type. This can be particularly obscure. In this case, the Ionide extension for VS Code already does this well, so there is no need to include it.



```
int -> stri
let f x
  if x
  else 42
```

```
All branches of an 'if' expression must return values of the sa
me type as the first branch, which here is 'string'. This branc
h returns a value of type 'int'.
```

Figure 19. F. Ionide with a clear error message

G. Watch out for type inference effects buried in a function

Similar to 'E', a function may have a baked in type defined:

```
let do y =
   // do something
   printfn "y is %s" y

do 1
// => error FS0001: This expression
// was expected to have type string
// but here has type int
```

Figure 20. G. Baked in type

This error is quite clear. The programmer will look for something expecting a string and should find the '%s' relatively easily. As such, this will not be included in the analyser.

H. Comma instead of space/semicolon

As mentioned earlier, commas exists only in tuples, while in many other languages it is used to define multiple arguments to a functions:

```
// define a two parameter function
let add x y = x + 1

add(x,y)
// FS0001: This expression
// was expected to have type int
// but here has type 'a * 'b
```

Figure 21. H. Mixing up tuples and commas

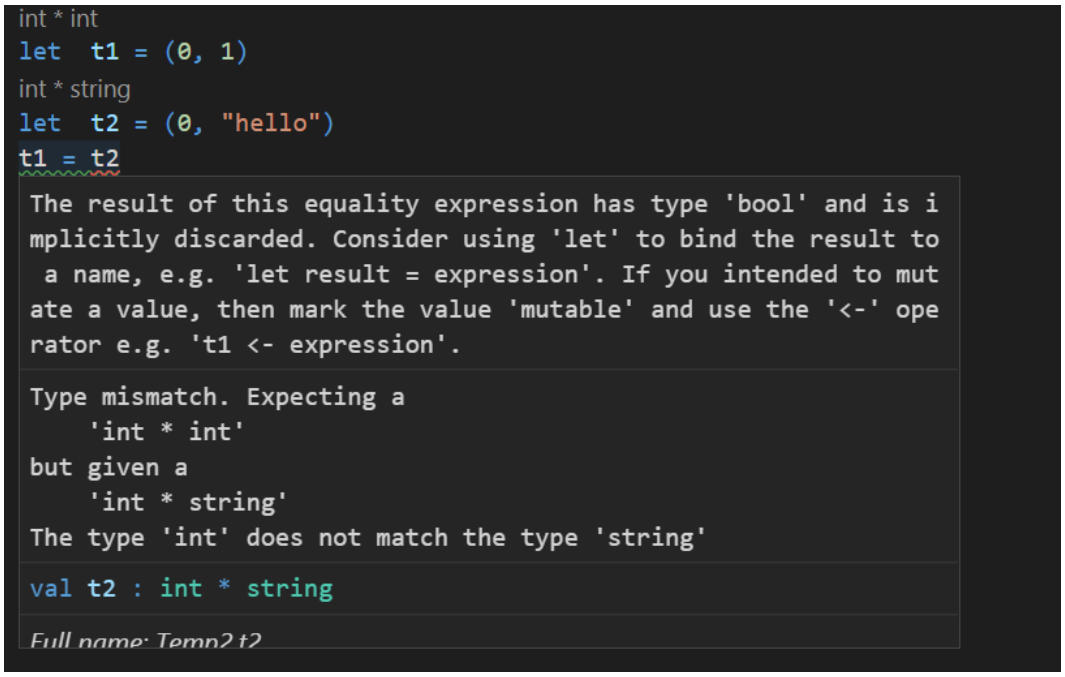
The error message can be obscure to novice programmers who have not encountered a tuple before.

I. Tuples must be the same type to be compared or pattern matched

Comparing tuples of the same type can lead to a rather obscure message as shown above. This comparison also includes calling a function argument with different lengths of tuple:

```
let f(x,y) = x - y
let temp = (1,2,3)
f temp
// Tuples have differing
// lengths 2 and 3
```

Figure 22. I. Incorrect length example



```
int * int
let  t1 = (0, 1)
int * string
let  t2 = (0, "hello")
t1 = t2
The result of this equality expression has type 'bool' and is i
mplicitly discarded. Consider using 'let' to bind the result to
 a name, e.g. 'let result = expression'. If you intended to mut
ate a value, then mark the value 'mutable' and use the '<-' ope
rator e.g. 't1 <- expression'.
Type mismatch. Expecting a
    'int * int'
but given a
    'int * string'
The type 'int' does not match the type 'string'
val t2 : int * string
Full name: Temp2.t2
```

Figure 23. I. Ionide states type error

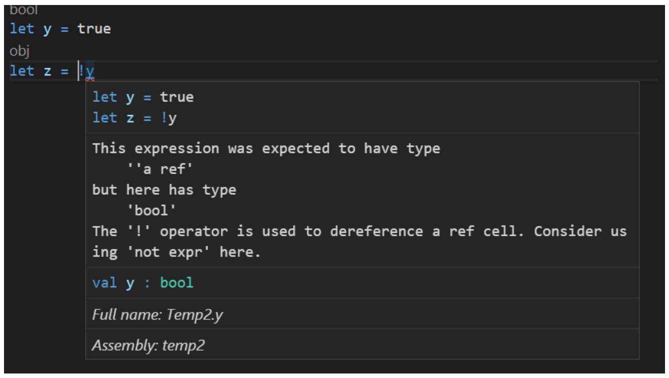
In this case, Ionide already highlights to the user that the tuples have differing lengths, so this particular feature will not be included.

J. ! is not the 'NOT' operator

```
let y = true
let z = !y      //wrong
// => error FS0001: This expression
// was expected to have type
// 'a ref but here has type bool
```

Figure 24. J. Incorrect usage of NOT operator

This is an obscure error. However, Ionide currently displays a clear error as shown below:



```
bool
let y = true
obj
let z = !y
```

```
let y = true
let z = !y
This expression was expected to have type
    ''a ref'
but here has type
    'bool'
The '!' operator is used to dereference a ref cell. Consider us
ing 'not expr' here.
val y : bool
Full name: Temp2.y
Assembly: temp2
```

Figure 25. J. Ionide clear error

To fix simply use the not keyword: `let z = not y` . This feature will not be added.

K. Operator precedence

In F#, operator precedence can cause ambiguous errors as there is no clear message explaining why the error has occurred.

```
String.length "hello" + "world"
// => error FS0001: The type
//'string' does not
// match the type 'int'

// what is really happening
(String.length "hello") + "world"
```

Figure 26. K.Precedence example

A simple fix is to use parentheses `String.length ("hello" + "world")`.

In contrast, the pipe operator has low precedence compared to other operators. In the example below the pipe takes the whole expression to its left, leading to an error.

```
let result = 42 + [1..10] |> List.sum
// => => error FS0001: The type
//''a list' does not
// match the type 'int'

// what is really happening
let result = (42 + [1..10]) |> List.sum

// fix with parenthesis
let result = 42 + ([1..10] |> List.sum)
```

Figure 27. K. Pipe operator precedence

This error satisfies all three criteria, effectiveness of correction, likelihood of usage and difficulty. Improved error messages will help users to realise that brackets are needed, and any usage of function calls with operators will have to take operator precedence into account. This will be the Operator Precedence Analyser.

7.3. Design of analyser

7.3.1 Parameter Analyser

The parameter analyser primarily deals with too many parameters when calling a given function. In F#, functions can be called with fewer parameters - also known as partial application. As such errors do not occur when there are too few parameters, unlike other languages. The goal of the analyser is to detect areas where a user defined function is called with too many parameters, and provide contextual information relating to the error. This contextual information includes the function name as well as the number of arguments to the function. The reason for choosing user defined functions is that library functions are harder to detect with contextual information. Furthermore, library functions have defined parameters which can be looked up through documentation whereas custom functions have to be checked in the code itself.

The goal of the Parameter analyser is to highlight the exact line of the code where too many arguments have been called, specify the exact name of the function call as well as the expected number of arguments. These additional information helps the user not only to pin-point the error but also realise the correct fix relatively quickly. This can be contrasted with the current error on too many arguments, "FS0001: Type 'a → 'b' does not match type 'int'". The original error does locate the section of code that needs to be corrected, but does not identify that the function has too many parameters nor does it specify which function call has the error or how many parameters to expect. This compiler error is due to curried parameters done by the compiler, explained in section C of 7.2.

7.3.2 Parenthesis Analyser

The parenthesis analyser tackles errors relating to missing parenthesis within the code. It is worth noting that the FSharp Compiler alongside Ionide does tackle some cases of missing parenthesis in certain cases. These cases are when the source code is relatively simple, with the error within a let statement or declaration. As additional operators such as pipes are utilised, cases where missing braces are not detected increase. The parenthesis analyser helps tackle such cases.

Note that the parenthesis analyser underwent several iterations, firstly utilising untyped syntax tree, followed by the typed tree and finally parsing of characters. These are explored in section 8. The reasoning for choosing to parse characters instead of traversing the syntax tree are that parsed errors due to mismatched brackets often corrupt the syntax tree and are the reason why the FSharpCompiler and Ionide do not detect these missing brackets in certain circumstances. As such, detection of brackets was achieved through parsing the source code, thus avoiding the corrupt tree. The trade off here is that is may be relatively inefficient to parse through the source code compared to traversing the syntax tree, especially for large code bases. However, this inefficiency is unlikely to have a noticeable effect during use.

The data structure chosen to check for correctness of brackets was a stack. Though F# does not have an inbuilt stack function, it is relatively easy to define a custom stack simply by mimicking push and pop operations on the stack. The stack was chosen because matching brackets means that only the previous element needs to be kept track of. For example, after encountering a ')' character, the previous bracket character should be a '('. This allows the checking of elements on the stack to be done in O(1) time, as only the top element needs to be checked. This can be compared to using a regular array, where it would take O(N) time to check for an element in the array (assuming elements are not indexed).

7.3.3 Operator Precedence Analyser

The operator precedence analyser deals with errors related to the priority of operations. This is due to the infix notation, X + Y, requiring rules stating operator precedence and associativity. For example, function calls in F# are left associative and have a higher precedence when compared to prefix operators [27]. On the other hand, pattern match statements are right associative. This results in obscure error messages relating to the incorrect type, when in reality the cause of the problem is due to operator precedence. This can be confusing on what exactly is occurring, especially for novice users. Though the error is located at the correct place, the type error does not indicate that brackets are missing.

In Figure 28 the function call String.Length is left associative and has higher precedence compared to the prefix '+', as such passes only the string "hello" into the function call. It is clear that the user has missed out on brackets.

```
String.length "hello" + "world"
// => error FS0001: The type
//'string' does not
// match the type 'int'

// what is really happening
(String.length "hello") + "world"
```

Figure 28. Operator Precedence example

A simple fix is to use parentheses `String.length ("hello" + "world")`.

Currently, the operator precedence analyser works by detecting function calls alongside the "+" operator. The majority of cases where operator precedence results in errors occur with a function call and pipes along side the '+' operator. Pipes are currently not detected as they are followed up with functions in almost all cases. The reasoning for the '+' operator is that it is both utilised to concatenate strings as well as add numbers together. When combined with function calls, this is the section where errors are most likely to occur.

The analyser also supports a small additional feature which tackles errors originating misuse of infix operators as prefix operators. These are often spacing errors and can be quite hard to catch in a large code base (Figure 29).

```
let result = 1 +3

// versus
let result = 1 + 3
```

Figure 29. Operator Precedence example

Currently the two most common operators that can be misused as either prefix or infix operators are "+" and "-". When used as prefix operators, these indicate positivity or negativity of a variable. On the other hand, when used as prefix operators they indicate addition, subtraction or concatenation of variables. Though this feature can be considered low-value as the frequency of the error may not be as high as other errors, the implementation workload was relatively low given the already existing operator precedence analyser.

8. Implementation

8.1. Analyser Introduction

In this project, analysers are functions that take in a context object from Ionide that contains the untyped and typed abstract tree, alongside the actual source code. The analysers then return a range object which will highlight the relevant areas. The high level design can be seen in Figure 30.

To traverse the abstract trees, custom functions are defined within the analyser that are passed into functions that traverse the abstract tree. These functions are modified and return values at different points of the traversal depending on the type of analyser. However, each of the functions will return a range object indicating the place in code where the analyser would like to highlight an area.

To perform traversal, recursive pattern matching is used to traverse the abstract syntax tree and call the defined function - in this case checking if an option has called them. Value function. Small snippets are shown in Figure 31.

Besides traversal of syntax trees, the analyser also has the option to manually parse characters of the source code itself. This might be desired in a situation where the syntax tree would not give an accurate depiction of the source code, for example a parse error due to a missing bracket. The error might be located in a different place entirely to the where the parse error was indicated, and therefore correct information of the location of the error is not known to the analyser from the syntax trees. In this case, manual traversal of the source code itself will indicate sections of missing brackets with good confidence.

Secondly, some work is needed to compile and build the analyser to work in conjunction with Ionide. There is little documentation regarding this section and so a simple tutorial could be of use to others who wish to create custom analysers for F#. This is already done on the Github page [28].

This is made possible primarily due to the F# compiler service [29], which provides APIs to access the abstract syntax tree, and is how Figure 31 performed pattern matching on it. However,

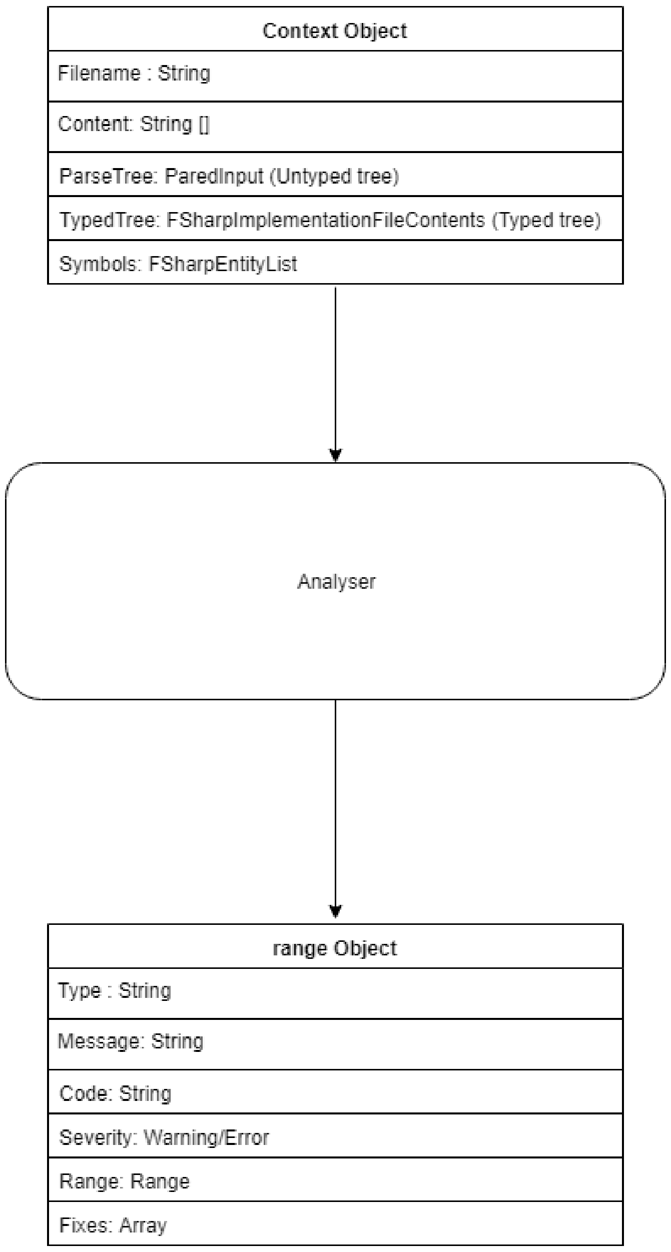


Figure 30. F. Ionide with a clear error message

there are problems with integrating with Visual Studio Code, which uses NodeJS processes for its extensions. This is because the F# Compiler Service itself is a .Net library and can't be used directly by NodeJS processes.

This is where Fsautocomplete comes in [30]. The project enables communication between the F# compiler service and the outside world. This is used by Ionide to communicate with the F# compiler service and is what enables custom analysers to be possible.

Lastly, several steps need to be undertaken in order to compile the analyser for usage. Currently,

```
// pattern match through abstract syntax tree
let rec visitExpr memberCallHandler (e:FSharpExpr) =
    match e with
    | BasicPatterns.AddressOf(lvalueExpr) ->
        visitExpr memberCallHandler lvalueExpr
    | BasicPatterns.AddressSet(lvalueExpr, rvalueExpr) ->
        visitExpr memberCallHandler lvalueExpr; visitExpr memberCallHandler rvalueExpr
    | BasicPatterns.Application(funcExpr, typeArgs, argExprs) ->
        visitExpr memberCallHandler funcExpr; visitExprs memberCallHandler argExprs
    ...
// our actual analyser which detects an option.Value is called
[<Analyzer>]
let optionValueAnalyzer : Analyzer =
    ...
        let name = String.Join(".", m.DeclaringEntity.Value.FullName, m.DisplayName)
        if name = "Microsoft.FSharp.Core.FSharpOption`1.Value" then
    ...
```

Figure 31. Sample Analyser Code Snippets

these are achieved in the form of a bash script in order to automate away manual labour. A snippet is shown in Figure 32

```
# setup things
...
dotnet pack --output nupkgs
echo "Compiled nupkg"
cp -fr nupkgs/${packageName} ../${usage_Dir}
echo "Copied nupkg to usage directory"
cd ..
cd FSharp.Analyzers.Sample.Usage/packages/analyzers/FSharp.Analyzers.Sample/
unzip -o ${packageName}
echo "Successfully unzipped and overwritten new files"
```

Figure 32. Bash script Snippet

8.1.1 Inputs and Outputs

Each analyser takes in a context object from Ionide. These are passed into the analysers as arguments, and are updated every time the user makes changes to the code.

At the top level, each analyser takes in a context object, which contains the information given to the Analyser from Ionide. The analyser processes this information in real time and outputs a range object back to Ionide that highlights relevant parts of the code.

Context Object: Ionide → analyser

```
FileName: string /* Full Path to file currently in view */
Content: string[] /* Contents of file separated by new lines */
ParseTree: ParsedInput /* Untyped syntax tree */
TypedTree: FSharpImplementationFileContents /* Typed syntax tree */
Symbols: FSharpEntity list /* Attributes of source code (e.g interface/ Module name) */
```

The output of each analyser is a message object containing information on the location of the error, severity and the message type.

Message Object: Result

```
Type: string /* Heading of error */
Message: string /* Detailed information on error */
Code: string /* CodeName for error e.g. P000 */
Severity: Severity /* Determines red/green underline */
Range: Range.range /* Location of code to be highlighted */
```

8.2. Enabling Analysers on Error

As some analysers only run once an error is detected, it is worth noting this in one separate section. The analysers should display improved error messages compared to the default Fsharp error messages under certain conditions. Conversely, under conditions where the code is optimal, the analysers should not output any messages. By running the analysers only when an error is detected, this reduces the chances of false positives and the error rate of the analysers.

Algorithm: Error detection key stages

```
/* Initialise FSharpChecker object to call API */
1 Create a mock file with file name and source code ;
2 Call FSharpChecker.ParseAndCheckProject to run file against F# compiler ;
3 Result will be an object with property Errors ;
4 Errors is an Array of FSharpErrorInfo, including the range, type and name of the error;
```

Errors are detected by utilising the F# Compiler API via the library FSharp.Compiler.SourceCodeServices. This library supports the creation of an FSharpChecker object, allowing calls to be made to the compiler given source code input. As such, after scrubbing and formatting the source code received from Ionide (Context.Content) it is passed into the FSharpChecker object. The compiler API is then called and run asynchronously and waits for the result, using F# command (Async.RunSynchronously). The API returns an array of errors, and by checking the emptiness of the array the analyser can be enabled to run only on detection of error.

It is important to note that the context object is returned quite frequently, at around once per second. This means that having the compiler API processes the source code multiple times is relatively inefficient. As such future work may include caching the result of the from the compiler API, or fix the frequency of the number of calls made by the function. Currently processing power is being utilised mostly by FSAC, FSharp Auto Complete, which forms the connection to the compiler API. The community has found FSAC to be inefficient and updates are on the way [31].

8.3. Function Name recording

In the parameter analyser and operator precedence analyser function names are recorded in order to provide further contextual information to the user. This is done by traversing the parse trees and recording function definitions and calls. An outline of the traversal can be seen in the Function Name Algorithm.

Function names are utilised in both the Parameter Analyser and Operator Precedence Analyser. Notably, each expression contains a body which is an expression as well. The source code is expressed as expressions within expressions, and the tree visits each of them looking for a particular type until the body is empty.

8.4. Parameter Analyser

The parameter analyser makes use of function by recording them down using the Function Name algorithm above. However, the parameter analyser also makes use of the number of arguments declared when defining a function. These are obtained through an additional traversal inside the SynPat data type, matching on a synVal data type. This can be seen in the Parameter Analyser Algortihm.

```
Algorithm: Function Name algorithm
/* Initialise data-structure for storage of function names */
FunctionNames ;
foreach NameSpace or Module do
    /* Traverse over declarations in a module */
    foreach Declaration do
        Pattern match SynPat data type to obtain information about declaration ;
        if SynPat contains a function declaration then
            /* Save the name of the function */
            FunctionNames ← Function Name
        end
        /* To check actual function names when used, traversal of the tree is needed.
           This is done by recursively iterating on a expression object for each
           declaration */
        foreach Expr do
            if Expr is of type App then
                if Expr has a function call then
                    Compare function call name with saved FunctionNames to determine if it
                     is a user defined function ;
                end
            end
            /* Recurse until body of expression is empty */
            if body of expression is not empty then
                Recurse on body of expression;
            end
        end
    end
end
```

The parameter analyser is firstly enabled if there is an error in the source code. Next, the untyped tree is traversed looking for declaration of user defined function calls. This is achieved through traversing the untyped abstract tree, looking for let declarations and matching a LongIdent pattern. This matches and returns the name of the function call, e.g. let add x y = ... returns the name of the function, "add". Next, within the same branch of the syntax tree the number of arguments of the function is recorded by matching on the SynValData type. The name, alongside the of the number of arguments are saved in a map with the key as the function name, and the value as the number of arguments. This enables a lookup of the number of arguments based on the name of the defined function.

It is worth to note that function overloading is not possible in F#, but is possible in other languages such as C++. In these cases this method of recording the function name will need to

Algorithm: Parameter Analyser algorithm

```
/* Initialise map for storage of function names (key:  function name, value:  number of
   arguments)                                                                          */
1  FunctionNameMap ;
2  foreach NameSpace or Module do
       /* Traverse over declarations in a module                                       */
3      foreach Declaration do
4          Pattern match SynPat data type to obtain information about declaration ;
5          if SynPat contains a function declaration then
               /* getNumArgs traverses a SynVal data type to obtained number of arguments for
                  a function                                                           */
6              Number of Arguments ← getNumArgs ;
               /* Save the name of the function with number of arguments in a map      */
7              FunctionNameMap ← Key: FunctionName, Value: Number of Arguments
8          end
           /* To check actual function names when used, traversal of the tree is needed.
              This is done by recursively iterating on a expression object for each
              declaration                                                              */
9          foreach Expr do
10             if Expr is of type App then
11                 if Expr has a function call then
12                     Compare function call name with saved FunctionNames to determine if it
                        is a user defined function ;
13                     If it is add function name and number of arguments to result
14                 end
15             end
               /* Recurse until body of expression is empty                            */
16             if body of expression is not empty then
17                 Recurse on body of expression;
18             end
19         end
20     end
21 end
```

be modified as function overloading enables different number of parameters for the same function name.

In the next part, traversal now looks for names of functions by matching on a SynExpr Ident pattern. This matches all function calls. For example, let result = add x y results in the name "add". Next, the name of the function is searched in the map that was built. If it exists in the map, then the number of arguments are compared. Next, if the arguments different the range is recorded and the user prompted for a possible recommendation.

Since traversal looks for names of functions using SynExpr Ident, it also matches inbuilt function calls by the compiler. For example, simple addition (x + y) results in a function called op_addition. As such in the unlikely event that the user defines a function with the same name as the compiler function it will result in a match. For example, if a user defines a function op_addition. The untyped tree currently cannot differentiate between user defined functions versus inbuilt compiler functions. However, using the typed tree it may be possible and can serve as an improved version of the analyser.

Furthermore, the parameter analyser only runs under when there is an error in the code. This reasoning is to ensure that user defined functions that are partially applied are not picked up by the analyser. This is a perfectly acceptable method of programming. For example partially applying only one of the arguments to a a function by setting it to a constant value will be seen as calling the function only with 1 argument, and can result in an in an incorrect number of arguments. This is idea is counteracted firstly by requiring an error to occur; with fewer arguments there will be no errors detected. Next, this is also counteracted by only checking if the arguments exceed the set number of arguments set out when the user defined the function. In the first iteration of the analyser arguments would only be checked for equality, and the analyser would activate in the event of an error and a different number of parameters to the function. However, only the case of an error and the case of too many parameters would there be an error. If there are too few parameters there would be no error regarding the number of parameters, only some other error. As such to reduce this possibility the analyser is only activated if the number of parameters exceed the number recorded in the map.

8.5. Parenthesis Analyser

The parenthesis analyser deals primarily with missing brackets in the source code. The first version of the parenthesis analyser worked by traversing over the untyped abstact tree. Through investigation of the tree, parse errors could be detected which indicated missing brackets in the area. However, these parse errors could sometimes be located in completely different locations to the actual error. Therefore making such an analyser might not be of as much use to the user. Next, the typed tree was investigated to determine if it could be used. However, no identifiable options were discovered. Instead, manual parsing of characters was investigated and an algorithm was developed to detect unmatched brackets. Each line of the source code is parsed and run against the algorithm. Though the algorithm was written using pattern matching in F#, the pseudo code displays the logic using normal if statements.

Algorithm: Parenthesis Analyser Bracket Matching Algorithm

```
1  Function BracketDetection(string, stack):
2      First character of string is matched and run against the conditions
3      if '(' found then
4          Add '(' to stack
5          Remove '(' from string
6          BracketDetection string stack
7      else if ')' found, '(' on top of stack then
           /* Found correct closing brackets                                 */
8          Pop '(' from stack
9          BracketDetection string stack
10     else if ')' found, '(' not on top of stack then
           /* If ')' found there must be '(' on top of stack to match        */
11         return error
12     else if Any other character, stack then
           /* If no brackets move on to next character                       */
13         Remove character from string
14         BracketDetection string stack
15     else if empty string, empty stack then
           /* This means the string has been parsed and brackets are matched */
16         return no error
17     else if empty string,stack non empty then
           /* There is an unmatched bracket                                  */
18         return error
19     else
           /* if both string and stack are empty                             */
20         return no error
```

8.6. Operator Precedence Analyser

This analyser deals with 2 issues. The first, simpler issue relates to error regarding prefixes and spacing issues. The next issue relates to errors with operator precedence regarding function calls.

The analyser combines them at a higher level through the following algorithm:

Algorithm: Operator Precedence

```
/* Analyser takes in Context object containing source code and syntax trees (ctx)      */
1  Input: ctx (Context object) ;
2  Output: Array of (Message object) ;
3  if Errors detected then
4      Save names of function calls throughout code;
5      if Error line contains any function call then
6          Create range object from error details ;
7          Add functionNames and prefix details to Result;
8      else if No function calls then
9          Check spacing and prefixes ;
10         Create range object from error details ;
11         Add prefix details ;
12 end
```

To check if the error contains any function calls, the each item in the set of recorded function names is checked for existence in the error line. In order to check spacing and prefix errors, a stack is used to keep track of characters. This is used to keep track of prefixes and spaces, and identify areas where there is a possibility of error. Most likely errors are spaces missing from prefixes, and the algorithm targets these specific issues. The function recursively iterates through the characters until the whole input has been parsed.

Algorithm: Prefix Checker

```
1  Input: Input String (String) /* Contains all characters on detected error line */
2  Output: Error detected (Boolean), Prefix (char);
3  if Found prefix character in string, stack is empty then
4      Add prefix to stack ;
5      Recurse with new stack;
6  else if Found space character, stack has prefix on top then
       /* There is a space between prefixes and the next character, unlikely to be wrong */
7      Pop prefix off the stack;
8      Recurse with reduced stack;
9  else if Found another character other than space, stack has prefix on top then
       /* There is no space between prefixes and the next character, likely be wrong */
10     Return error alongside character and prefix;
11 else if No more characters, stack empty then
12     Return no error;
```

Note that the analyser does not support multi-line errors at this time. Reasoning behind this is that errors due to spacing with prefixes are unlikely to be spread across two lines.

9. Testing

Testing was performed in two phases, user testing and automated testing. Both of these components help verify the robustness of the analysers, and determine how effective the analysers can be in a real world environment. User testing is utilised to evaluate effectiveness of error messages with the analysers compared. From user testing feedback on the impact of the analysers will be obtained, assuming full correctness of the analysers. Correctness of the analyser can be defined as only running under appropriate conditions and detecting all appropriate errors.

On the other hand, automated testing tests for correctness of analysers. Analysers should only be run when certain conditions are met, and not otherwise. Analysers should detect errors related to their specific use case. These two points are investigated for each analyser by constructing an automated test bench that runs the analyser against custom code.

9.1. User testing

Each volunteer has different backgrounds in terms of experience with F#, ranging from no experience, some functional language experience and prior experience with F# through a course. Tests were conducted in an interview format, with invalid code written prior to the interview. The volunteers were then asked to correct the code firstly without the analysers, then with the analyser activated. The error messages given by both the default F# compiler and the analyser were rated on a scale of 1 -5, with 1 being the least effective alongside any comments. The code snippets were designed to be simple so that a wide range of volunteers could participate regardless of previous experience with F#.

It is important to note that due to the small number of volunteers, feedback may contain biases. This is especially important when tackling errors, as prior experience in F# can drastically change the impact of the analysers. Nevertheless, steps have been taken to combat this bias. Firstly, each volunteer states their prior experience in order to ascertain the degree of familiarity with F#. Next, during attempts to correct code each volunteer was asked to state if the code was corrected solely on the error messages or any prior knowledge. These provide context for the comments the volunteers might make, in order to better evaluate the analysers.

9.2. Automated Testing

Currently, tests are setup to primarily test functionality through unit testing. Tests are setup with expecto, a testing library for F#. Tests performed by mocking a Context object, an exact replica of what Ionide gives to to the Analyser.

Each context object contains the ParseTree(untyped tree) and the Typed tree. To begin with, example code for a specific functionality is manually generated. The parseTree and TypedTree are then obtained via the FSharp Compiler API, FSharp.Compiler.SourceCodeServices. These are used to make a mock Context object, which then calls the Analyser function. This ensures that the tests utilises the exact analyser that is run. The results from the Analyser function are a range object that the Ionide extension takes in to display a tooltip and highlight the relevant code. Some properties of this range object are the message that is displayed upon hovering, the severity (warning/error) and the actual place in the code where the error is relevant.

Default tests mock some source code with an error, call the analyser function and expect the result to highlight the area where the error is. These are tested by mocking the output range and error message. Further tests include testing for false execution of the analyser, for example utilising code snippets that should not activate the analyser. To further test the extent of functionality the input code will be of increasing complexity to fully test the functionality of the analyser. Increasing complexity of testing can be placed in classes listed below.

- Basic functionality tests, single error on single line
- Multiple errors on a line
- Multiple errors on 2 different lines
- Correct code in between error lines
- Correct line using a function call
- Randomised tests

Randomised tests are tests that utilise random number generators in order to generate different source code each time. These are utilised to test varying parameters, such as different locations of brackets or different number of parameters.

9.3. Performance

Within the project, performance can be tested in numerous ways. Firstly, the most basic test of performance would be simply judge the latency of the the analyser by simply judging how long it takes for the error messages to appear under the correct conditions. While this method may not be the best for hard quantitative the benefits are that it is much easier to perform such testing in terms of timing and effort spent.

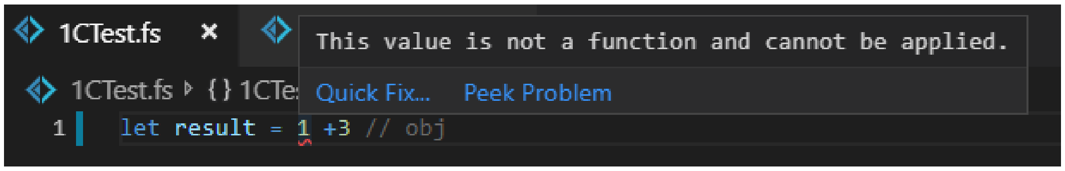
On the flip side, the relative benefits of performance needs to be clarified. While performance is important in any software project, it is worth noting that the goal of the project is to investigate how error messages can be improved, rather than a fully established tool to use. In this sense, performance is not as important as long as performance does not have a dire impact on the user experience. For example, a basic criteria would be for the analyser to respond within 1-2 seconds in the right situation. Furthermore, much of the performance is dependant on Ionide. The analysers can be seen as a program that utilises Ionide and conversely FSharp Auto Complete to achieve its goals. As such, while it is true that optimisation can be made to the code to improve performance to a degree, performance is largely dependant on the amount of source code and the efficiency of the FSharp Compiler API service. Therefore, performance in the context of the analysers itself is not as important as the effectiveness of the heuristics .

10. Results

10.1. User testing

- Volunteer 1: No prior experience with F#
- Volunteer 2: Some functional language experience but not with F#
- Volunteer 3: Prior experience with F#

10.1.1 Code snippet 1: Prefix Spacing

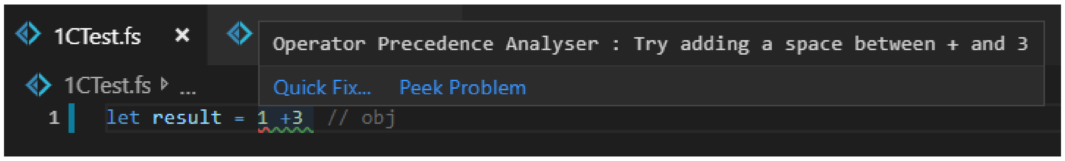


```
This value is not a function and cannot be applied.
```

```
let result = 1 +3 // obj
```

Figure 33. CodeSnippet 1 with error

Though the error might seem trivial the tests should start out with something relatively easy to smooth out the interview process. In this code snippet, there should be a spacing between the prefix '+' and the letter 3. This targets the operator precedence analyser and its prefix feature. The error messages with the analyser enabled are shown in Figure. 34.



```
Operator Precedence Analyser : Try adding a space between + and 3
```

```
let result = 1 +3 // obj
```

Figure 34. CodeSnippet 1 with error

Volunteer 1: Able to correct the code after approximately 3 minutes without the analyser. Rated the default error message as ambiguous, and commented that the compiler error does not give information about the source of the error. With the analyser message, volunteer 1 felt it was much easier to correct the code quickly. (Default Error rating: 2/5. Analyser error rating: 4/5)

Volunteer 2: Able to correct the code within a minute. Commented on not understanding why the error states that the integer 1 is not a function and cannot be applied, when it is clear that 1 is an integer. Volunteer 2 felt that the error message with the analyser identified the key problem of the error in a much clearer way. (Default error rating: 1/5. Analyser error rating: 5/5)

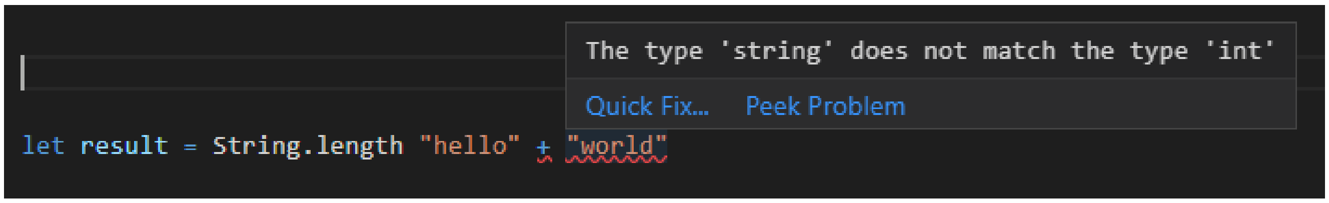
Volunteer 3: Able to correct the code within a minute. Commented that the error message could be because of the compiler interpreting the +3 as an argument for the function 1. Volunteer 3 felt that the error message is clearer with the Analyser. Volunteer 3 commented that he has not been frequently stuck in this type of error and questions the overall impact of the analyser as Volunteer 3 felt that it is a relatively easy fix. However Volunteer 3 also commented that he does see some usefulness especially for novice users. (Default error rating: 2/5. Analyser error rating: 3/5)

10.1.2 Code snippet 2: Operator Precedence

This code snippet deals with errors relating to the Operator Precedence Analyser. As the function call String.Length is right associative and has higher precedence than the + operator, the compiler is perceives the line with only the "hello" string as an argument.

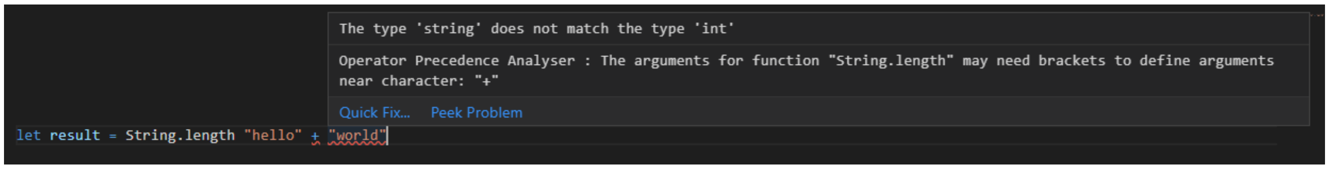
```
String.length "hello" + "world"
// => error FS0001: The type
//'string' does not
// match the type 'int'

// what is really happening
(String.length "hello") + "world"
```

Figure 35. Code snippet 2 correction



```
The type 'string' does not match the type 'int'
Quick Fix...   Peek Problem
let result = String.length "hello" + "world"
```

Figure 36. CodeSnippet 2 with error



```
The type 'string' does not match the type 'int'
Operator Precedence Analyser : The arguments for function "String.length" may need brackets to define arguments
near character: "+"
Quick Fix...   Peek Problem
let result = String.length "hello" + "world"
```

Figure 37. CodeSnippet 2 with error

Volunteer 1: Not able to correct error after 10 minutes using default error messages. Commented that it is unclear why the string type does not match an int when doing string concatenation.

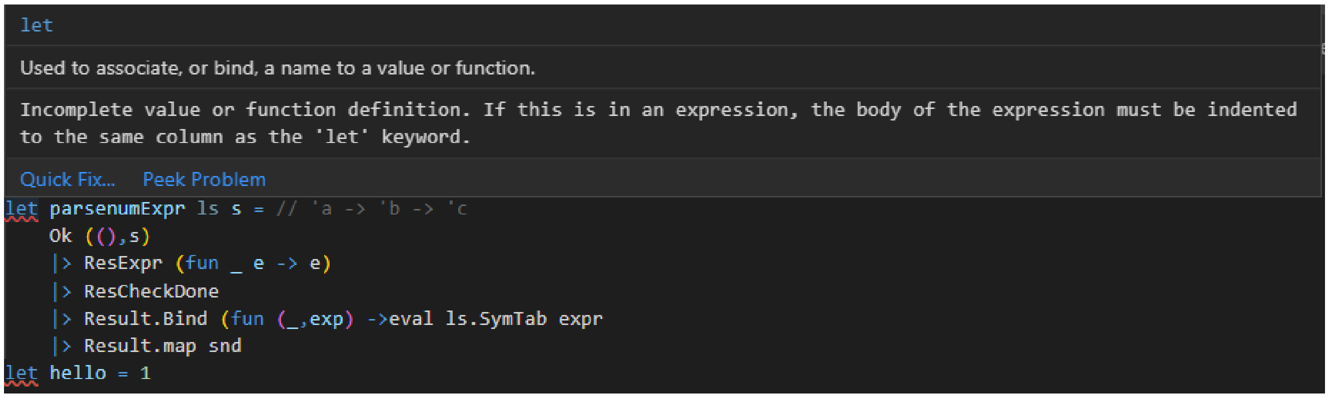
With the analyser, the volunteer realised that brackets were needed to group the arguments together.(Default Error rating: 1/5. Analyser error rating: 4/5)

Volunteer 2: Able to correct the code within a minute. Commented that previous experience in other languages helped fix the error. Felt that the default error message was better than the default error message in code snippet 1. With the analyser, Volunteer 2 felt that the error was more clearly defined in that brackets were required. (Default error rating: 2/5. Analyser error rating: 4/5)

Volunteer 3: Able to correct the code within a minute. Commented that he was familiar with these sorts of errors as they are quite frequent. With the analyser Volunteer 3 liked the specific function name mentioned for the error as it pointed at specific areas of the code. (Default Error rating: 1/5. Analyser error rating: 5/5)

10.1.3 Code snippet 3: Parenthesis Analyser

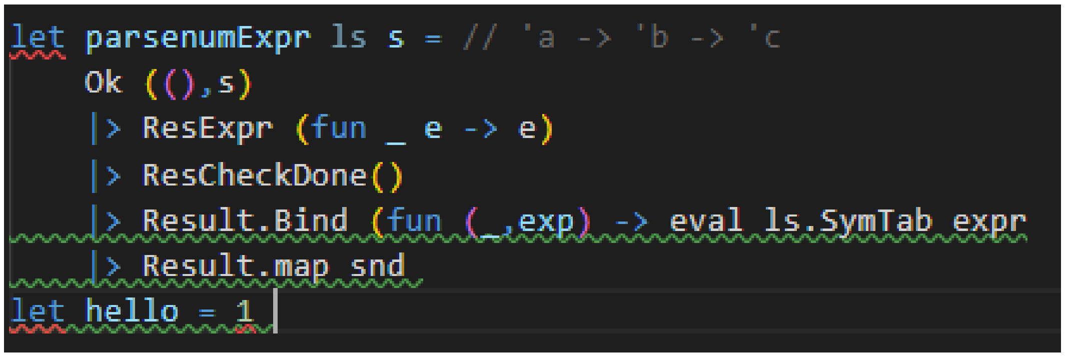
This code snippet deals with the parenthesis analyser. As parenthesis matching is a much easier problem in short code bases, the code snippet was extended to increase the difficulty of finding the error. Brackets are highlighted here to easily see the location of the missing bracket. Note that bracket highlighting was not present during user testing.



```
let
Used to associate, or bind, a name to a value or function.
Incomplete value or function definition. If this is in an expression, the body of the expression must be indented
to the same column as the 'let' keyword.
```

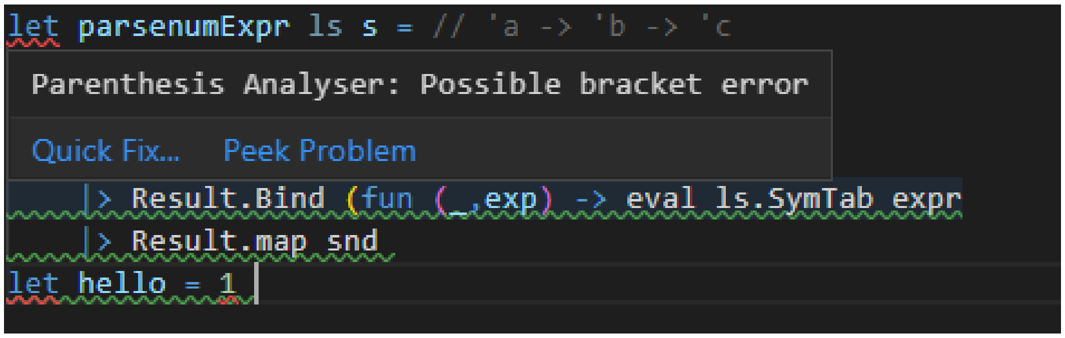
```
let parsenumExpr ls s = // 'a -> 'b -> 'c
    Ok ((),s)
    |> ResExpr (fun _ e -> e)
    |> ResCheckDone
    |> Result.Bind (fun (_,exp) ->eval ls.SymTab expr
    |> Result.map snd
let hello = 1
```

Figure 38. CodeSnippet 3 with error



```
let parsenumExpr ls s = // 'a -> 'b -> 'c
    Ok ((),s)
     |> ResExpr (fun _ e -> e)
     |> ResCheckDone()
     |> Result.Bind (fun (_,exp) -> eval ls.SymTab expr
     |> Result.map snd
let hello = 1
```

Figure 39. CodeSnippet 3 with analyser highlighted



```
let parsenumExpr ls s = // 'a -> 'b -> 'c
 Parenthesis Analyser: Possible bracket error
```

```
      |> Result.Bind (fun (_,exp) -> eval ls.SymTab expr
      |> Result.map snd
let hello = 1
```

Figure 40. CodeSnippet 3 with error message

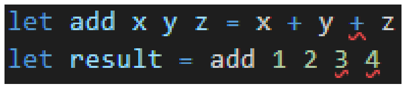
Volunteer 1: Not able to correct error after 10 minutes using default error messages. Commented that default error messages were extremely unclear as the actual correction has nothing to do with indentation. Found it hard to spot error message without analyser enabled. With the analyser enabled the error was corrected within a minute. Volunteer 1 found the underlining of sections right after the error to be incredibly useful in locating the error. (Default Error rating: 1/5. Analyser error rating: 5/5)

Volunteer 2: Able to correct the code within 10 minutes. Commented that the default error for this code snippet is the most obscure so far. With the analyser Volunteer 2 found that the underlining of lines helped locate the error much quicker. However, volunteer 2 felt that instead of underlining the whole line the underline should only be start at the exact parenthesis error, starting from the '(' character. (Default Error rating: 1/5. Analyser error rating: 4/5)

Volunteer 3: Able to correct the code within 3 minutes. Commented that the default error message is quite confusing. Also commented that there are tools that help combat these errors, such as bracket colouriser. With the analyser enabled, volunteer 3 felt that the underlines should start at the exact character error '(' to improve clarity. Found that highlighting all the lines after the error line helped in locating the error, compared to previous experience with bracket matching tools that only returned a Boolean. (Default Error rating: 1/5. Analyser error rating: 4/5)

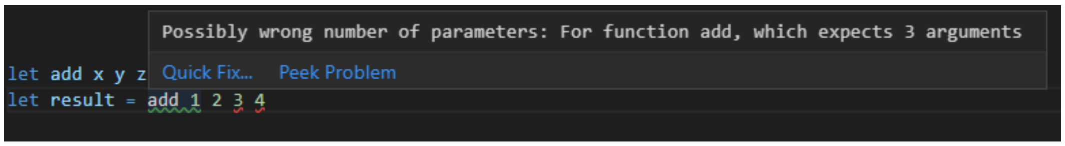
10.1.4 Code snippet 4: Parameter Analyser

This code snippet deals with the parameter analyser. The function add has been declared with three parameters but called with four.



```
let add x y z = x + y + z
let result = add 1 2 3 4
```

Figure 41. CodeSnippet 4 with error (The type ”a → 'b' does not match the type 'int')



```
Possibly wrong number of parameters: For function add, which expects 3 arguments
let add x y z
let result = add 1 2 3 4
```

Figure 42. CodeSnippet 4 with analyser

Volunteer 1:Able to correct error after 5 minutes using default error messages. Commented that the error message is confusing as the error message regarding the type seems to be out of context. With the analyser Volunteer 1 found the code much easier to correct, given the function name and number of arguments. (Default Error rating: 2/5. Analyser error rating: 4/5)

Volunteer 2: Able to correct the code within 3 minutes using default error messages. Commented that this code snippet was a relatively easy fix with or without the analyser. However also commented that the analyser could be useful in scenarios where the codebase is complex. (Default Error rating: 1/5. Analyser error rating: 2/5)

Volunteer 3: Able to correct the code within 3 minutes. Commented that the errors are due to currying performed by the FSharp compiler. Volunteer 3 felt that this was one of the more frequent errors encountered when coding in F#. Mentioned underlines for all parameters of function at the function call instead of only the first parameter. Found the specific function name inclusion alongside the number of arguments to be particularly useful. (Default Error rating: 3/5. Analyser error rating: 3/5)

10.1.5 Code snippet 5: Operator Precedence Analyser

This code snippet deals with the operator precedence analyser. The pipe operator takes in everything on the left side (42 + [1..10]) which causes the error. To fix the error, brackets are required

([1..10] | > List.sum) to define the precedence of operators

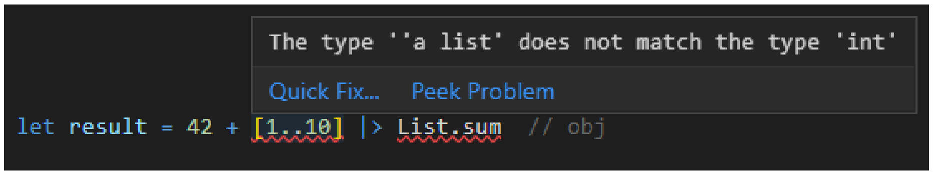

```
The type ''a list' does not match the type 'int'
Quick Fix...   Peek Problem
let result = 42 + [1..10] |> List.sum  // obj
```

Figure 43. CodeSnippet 5 default error message

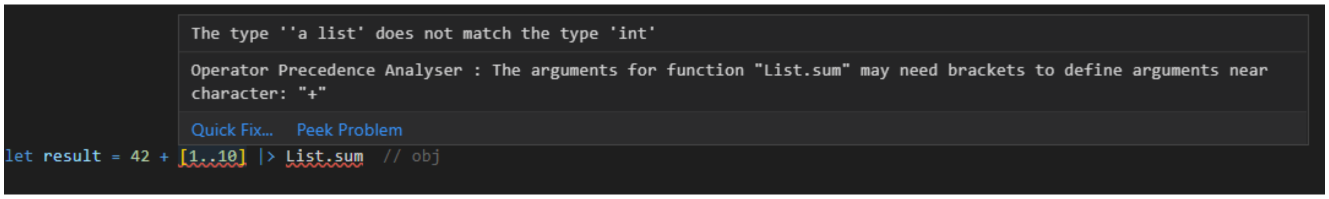

```
The type ''a list' does not match the type 'int'
Operator Precedence Analyser : The arguments for function "List.sum" may need brackets to define arguments near
character: "+"
Quick Fix...   Peek Problem
let result = 42 + [1..10] |> List.sum  // obj
```

Figure 44. CodeSnippet 5 with analyser

Volunteer 1: Unable to correct code with default error messages. Commented on the lack of knowledge of the pipe operator in F#. With the analyser, was still unable to correct the error. Found both default and analyser error messages confusing. (Default Error rating: 1/5. Analyser error rating: 2/5)

Volunteer 2: Able to correct the code within 1 minute using default error messages. Notes that previous experience with functional languages helped in this area, not neccesarily the error message. Found the error message only slightly helpful, as it is unclear where brackets need to be placed (Default Error rating: 1/5. Analyser error rating: 2/5)

Volunteer 3: Able to correct the code within 1 minute. Commented that this was a frequent error that was encountered. Found that the error message from the analyser was slightly obscure, as there is no indication of where exactly to place the brackets. However mentioned that the hint for bracket is quite useful in indicating the potential fix. Also found the specific names of functions and prefixes to be helpful in locating the error.(Default Error rating: 2/5. Analyser error rating: 3/5)

10.1.6 Miscellaneous comments

Volunteers 2 and 3 noted that the green underline provided by the analyser sometimes made it hard to notice the error, as the green underline is usually for warnings. They felt that a red underline would help the error messages stand out more.

Upon starting the Visual Studio Code, volunteers 1 and 2 found that the analysers were not run until some input was entered. This left them waiting before manually inputting a character.

10.1.7 Overall Results

Table 2 summarises the results found for the user testing explained before. From the results presented, it can be seen that the average default error rating corresponds to 1.4/5 while the analyser error rating to 3.6/5 resulting in more than double the rating when using the analysers.

Volunteer	**Error Correction Time**	**Default Error Rating**	**Analyser Error Rating**
Code snippet 1: Prefix Spacing			
1	3 minutes	2/5	4/5
2	1 minute	1/5	5/5
3	1 minute	2/5	3/5
Code snippet 2: Operator Precedence			
1	Unable to correct error	1/5	4/5
2	1 minute	2/5	4/5
3	1 minute	1/5	5/5
Code snippet 3: Parenthesis Analyser			
1	Unable to correct error	1/5	5/5
2	10 minutes	1/5	4/5
3	3 minutes	1/5	4/5
Code snippet 4: Parameter Analyser			
1	5 minutes	2/5	4/5
2	3 minutes	1/5	2/5
3	3 minutes	3/5	3/5
Code snippet 5: Operator Precedence			
1	Unable to correct error	1/5	2/5
2	1 minute	1/5	2/5
3	1 minute	2/5	3/5

Table 2. User testing results

10.2. Automated Testing

The results obtained for the automated testing are collected in Table 3. The number of tests that passed and failed for each different analyser used are recorded.

Analyser	Number of tests pass	Number of tests fail
Operator Precedence	6	3
Prefix Spacing	10	5
Parameter	4	2
Parenthesis	8	0

Table 3. Automated testing results

11. Evaluation

Evaluation will be done based on feedback obtained during user testing alongside results from automated testing. Key aspects will be highlighted and discussed in this section. Feedback obtained from user testing tests for the effectiveness of analysers, whether the usage of analysers improves the user's ability to solve error messages. Automated testing tests the functionality of the analysers, whether they are triggered at the correct location and under correct criteria.

11.1. Evaluation of Testing methodology

Firstly, it is important to note that the user testing has been performed with only 3 volunteers. It would be ideal to have a larger number of volunteers, in order to evaluate the analysers to the full extent through more rigorous feedback. However the feedback obtained can nevertheless be taken into account and form a basis of evaluation. Next, only 5 code snippets were tested. This was chosen in order to reduce the time spent testing in interviews. There was an adequate amount of feedback from all three volunteers, and it is unlikely and increased amount code snippets would produce further feedback.

Secondly, by utilising selected-code snippets bias may have been introduced as the code snippets may have errors that are less clear compared to average errors. In these cases the analyser may work well, however in another untested cases the result may differ. While it is unfeasible to test for every possible source code, consideration of the bias needs to taken into account when evaluating the test data.

It can also be noted that the 80% of the code snippets were able to be corrected without the analysers. While this number may seem high, this stems from the decision taken in testing to simplify code snippets in order to account for volunteers with less experience in F#. Furthermore, being able to correct the code within a minute differs from being able to correct the code within 10 minutes. This variance in time taken to fix the error is where additional information from analysers come into play. Lastly, an ideal test methodology would implement two separate sets of tests group and one given the analysers and the other without. These would enable verification of the difference in speed for correcting the errors. Currently, as the volunteers are correcting the error messages with and without analysers, once the error has been fixed the volunteers will already know what to look for. This could reduce the impact of the analysers.

11.2. Evaluation of user feedback

11.2.1 Code Snippet 1: Prefix Spacing

10.1.1 The volunteers overall felt that the default error messages were much worse when compared to the analyser messages (5/15 versus 12/15). By introducing specific characters and a potential fix, volunteers felt that they would be able to fix the error quicker with the analyser. This indicates an area where the operator precedence analyser could be put to good use, in this particular scenario.

One argument raised by the volunteers was that the code snippet was a relatively easy fix and may not be as frequent as other errors. All volunteers were able to correct the code within 3 minutes. This indicates that there might not be a need for the analyser, as the error is simple enough to be fixed without it. Furthermore, this class of analysers with prefix operators have a much higher chance of occurring in integer operations, which do not necessarily appear in all projects. In this sense, the prefix aspect of the analyser may not hold much impact as the situations where it will be useful are less frequent.

However, spotting a potential error in one line of code versus a thousand lines is a big difference. A small fix on one line can be very hard to spot in a complex project. In line with the requirements capture, the error is located appropriately and a message is displayed that helps the user correct it. However, given that the error is a relatively easy fix in most cases, coupled with the fact that the error may not appear as frequently as other errors, makes the prefix feature of the Operator Precedence analyser have minimal impact in terms of effectiveness.

11.2.2 Code Snippet 2: Operator Precedence

10.1.2 The volunteers overall felt that the default error messages were much worse when compared to the analyser messages (4/15 versus 13/15). Notably, volunteers felt that providing context of the error greatly increased the ability to find and locate the area, and transversely correct the error. Volunteers 2 and 3 were able to correct the error without the use of the analyser. However, Volunteer 1 was only able to correct the error with use of the analyser. Taking into context that volunteer 1 has the least F# experience, the analyser demonstrates a high impact case in this scenario.

Volunteer 3 also mentioned that these operator precedence errors are a frequent occurrence. Volunteers felt that the specific naming of the function and the prompt to add in parenthesis helped to speed up error correction. Given that without the analyser volunteer 1 would not be able to correct the error, and that the volunteers have frequently experienced similar errors, the Operator Precedence Analyser has great impact in terms of its effectiveness.

11.2.3 Code Snippet 3: Parenthesis Analyser

10.1.3 The volunteers overall felt that the default error messages for this particular test was much worse than normal (3/15 versus 13/15). Feedback for this user test was mostly positive, with volunteer 1 spotting the missing parenthesis only after running the analyser. It should be noted that the speed of fixing errors do play a role in the evaluation, and having a good error message should speed up the time taken to fix the error.

However feedback regarding the location of the error message was mentioned. In particular, the highlighting of the error underlines the entire line, instead of starting exactly at the error (Figure 40). Notably, this feature worked in a previous version of the Parenthesis Analyser. However the previous version worked by iterating over the tree and finding a parse error. In some cases the error would be located at incorrect lines throughout the code. As such the analyser was re-written to utilise a stack based approach. Currently, indicated error line helps the user locate the error especially in a large code base, as it is likely the user will simply scan the first line of the underlined error to find the error. However, pint-pointing the exact character where the error starts will remove the need for the user to scan the line in search of the error. This improved feature for the parenthesis analyser will be placed under future works.

At the current state the parenthesis has a good impact in terms of effectiveness. This is demonstrated by helping volunteer 1 discover the missing bracket, alongside high ratings from the volunteers.

11.2.4 Code Snippet 4: Parameter Analyser

10.1.4 The volunteers had mixed results in this section compared to previous user tests. Rating for default errors were 6/15 and the rating for analyser errors were 9/15. All volunteers were able to fix the error within 10 minutes without usage of the analyser. All volunteers agreed that the error messages from the analyser were better than that of the default errors, however volunteers 2 and 3 found that the number of parameters was a relatively easy spot and questioned the purpose of the analyser. This indicates that for the nature of the this problem, i.e. mismatched parameters the correction is relatively easy to spot even without a clear error message. This analyser provides minimal impact based on user feedback for a simple code base. However, it might find use in a more complex code base with many function calls scattered across.

11.2.5 Code Snippet 5: Operator Precedence Analyser

10.1.5 Volunteers found the error message relatively easy to fix, with only volunteer 1 unable to correct the code within 10 minutes. However this is likely attributed to the lack of knowledge from volunteer 1.

The volunteers found the analyser error more effective than the default error rating. (Default error rating: 4/15, analyser error rating: 8/15). However, the effectiveness of the analyser message is minimal as all volunteers found both the default and analyser error messages confusing to a certain extent. Volunteers found the error message on adding parenthesis unclear, as no areas are highlighted as to where the parenthesis should be placed. This may be contrasted to CodeSnippet 2, where the same error message was placed. In the case of CodeSnippet 2, the volunteers felt that error messages from the analyser had merit. In the case of CodeSnippet 5, the addition of the pipe operator made the error message confusing to the volunteers. This confusion can be attributed to the error message itself, which states that ”function List.sum may need brackets to define arguments near character +”. Observing the source code, it is clear that the pipe operator (Figure 44) makes it unclear which side the brackets should be.

This impacts the effectiveness of the Operator Precedence analyser. In this scenario, the impact of the analyser is minimal even though the error message from the analyser offer better clarity. A possible solution to the problem would be to identify which characters are most likely to be grouped together. For example, since List.sum takes in a List, it should be grouped together with the list [1..10].

11.3. Evaluation of Automated testing

More focus was spent in evaluating the effectiveness of analysers through user testing compared to functionality of analysers. This is because the analysers developed for this project are not intended to be released as fully fledged software, rather they serve as a proof of concept on which more advanced analysers may be developed on. It is worth noting that the analysers serve as a suggestion tool and are not intended to be correct a 100% of the time. In an ideal scenario many tests containing source code targeted at a specific error should be run against the analyser to obtain ascertain the functionality of the system. Current functionality can be evaluated, and it is intended that the analysers be improved upon by the presentation date.

11.3.1 Parameter Analyser Tests

Currently, the parameter analyser has a test containing a randomised number of parameters. The parameter analyser is able to handle varying amounts of arguments as a result of correct reading of the the syntax tree. This ensures that functionality of detecting the correct number of arguments works in cases of a simple function call. However, when passing in function calls to a print statement, the analyser does not detect any errors even if the the function has been called with multiple parameters. The analyser also works with brackets inserted around the expression.

This could likely be because traversal is stopped due to an unmatched pattern in the syntax tree. Next, relatively complex expressions also fail to correctly identify parameters.

```
let add x y = x + y
let boo = (([1..10]|> List.sum) + (add 1 2 3) + 5)
```

Figure 45. Example of relatively complex expression

11.3.2 Operator Precedence Tests

Currently, the Operator Precedence operator precedence analyser fails these classes of tests: Incorrect functions names during multiple function calls and multiple errors on a line.

Names of function calls are stored in an array correctly. When an error is detected on a line, the line is parsed looking for the function call. However, if there are multiple function calls on a single line then the detected function name results in a concatenation of all the detected function calls. The result is that if a single line has multiple operators with errors, the names of all the function calls with be combined. Though this case is unlikely, it serves as a breaking point for the analyser. This is also the case for the prefix feature of the analyser.

```
let result = 42 + [1..10]|> List.sum
let result2 = result + 2
```

Figure 46. Example of repeated function call

In this example the returned error on line 1 would result in the function "List.sum.result" being returned on the error on line 1.

11.3.3 Parenthesis Tests

Parenthesis tests had a 100% success rate. Missing brackets were able to be detected on the correct line in both simple and complex code. Brackets across multiple lines were also able to be detected. However, this does not mean that the parenthesis is fool-proof and there are potential tests that could make it fail. Unlike the other analysers, the parenthesis analyser was built using manual parsing of the code rather than traversal of the tree. This makes it more resilient to errors such as parse errors or missing branches of the syntax tree.

As mentioned in user testing, a key aspect that could be improved is that the location of error is identified not only on the correct line but also the correct character. Currently the parenthesis analyser can be said to be the most functional at handling complex code.

11.4. Scope of project

The scope for the project differs from the majority related work such as code formatters or linters which only deal with the appearance of code. This project pushes the boundary of what tools can do by analysing potential fixes for ambiguous errors. Part of the reason that the majority of tools deal with visual appearance of the code can be attributed to the fact that improving compiler errors is quite difficult to do, and dealing with error messages falls in the realm of the compiler itself. In this sense, the project can be seen as improving compiler's error messages from an external perspective, utilising information to further improve the user's understanding without touching the compiler itself. Due to this nature, this project will most likely not be of great use to established programming languages who already have great error messages as there is nothing to improve upon. However, newer or less common languages such as F# can see use as there are often obscure error messages that can be improved.

In the case of this project, scaffolding for building analysers were already in place through Ionide and the FSharp Compiler API. Other languages may not have the same privilege. In these cases information such as abstract syntax tree may be incredibly difficult to obtain and conversely an analyser may be impossible to build as there is not enough information. Lastly, having an analyser be part of a coding suite (through extensions) such as Visual Studio Code can be critical to its usefulness. An analyser that runs automatically in a user's normal environment in real-time will be much more useful than as a standalone console application. This way, the user will get to work in a familiar environment and improves ease of use.

In the sections below specific analysers are compared in the context of other related work.

11.4.1 Operator Precedence Analyser

There are close to no community built tools that aim to tackle operator precedence errors. These are likely due to the difficulty, complexity and impact of such tools. Different programming languages have dissimilar operator precedence, and in order to implement such a tools the easiest method would be to utilise a compiler API to traverse the compiled parse tree in order to identify which operators should come first. The closest tool can be said to be an Operator-precedence parser [32], a parser that interprets operator-precedence grammar. In most cases, the compiler will already perform the precedence calculations and the impact of such a tool may be negligible.

11.4.2 Parenthesis Analyser

There are several solutions available to tackle mismatched brackets. One direct competitor to the parenthesis analyser would be the BracketColorizer extension [33] available in Visual Studio Code. This extension enables brackets in the user's code to be colourised in order clearly see matching pairs. This allows the user to clearly see which brackets match, reducing the chances of a mismatched bracket. BracketColorizer is also language agnostic, meaning that it takes in source code from Visual Studio Code and analyses it regardless of the programming language. This allows BracketColorizer to have large impact as its user base applies to everyone. On the other hand, without additional tools, BracketColorizer is limited to only highlighting whereas the parenthesis analyser is able to underline and produce error messages within the code itself. Furthermore, the analyser has access to additional information through the syntax trees, allowing for improvements such as information regarding code such as function calls and names to be identified and presented. The parenthesis analyser may be seen as a tool specific to F# with a higher potential ceiling of improvement whereas the BracketColorizer may be seen as a more generalised tool that applies to multiple languages.

11.4.3 Parameter Analyser

There are close to no tools that measure the number of parameters expected from a function. Based on feedback from user testing, the impact of such a tool may be limited due to the the ease of correction of such errors. However, this is a feature integrated into most IDEs today. Examples include Eclipse, Visual Studio, IntelliJ IDEA as well as Ionide itself. These integrated development environments display function parameters often in a separate window. To consider the case of Ionide, hovering over function parameters indicates the type and expected arguments. However, during development mistakes relating to parameters can be easily made and finding the solution to correcting the code may be difficult even with the indicated parameters. The analyser can be seen as a tool that investigates and provides some context to the possibility of errors, whereas Ionide and the other IDEs can be viewed as having an integrated documentation tool look-up for parameters.

12. Conclusions and Further Work

Multiple analysers have been successfully developed that integrate with Ionide and provide better error messages compared to default error messages. These analysers can serve as a new class of tools that improve current obscure error messages, without having to modify compilers. Various errors were researched as potential candidates for different classes of analysers. Errors were investigated and evaluated in terms of their effectiveness of correction, likelihood of usage and difficulty of implementation. Large quantities of data related to the F# syntax tree were analysed in order to locate the properties of the errors. The analysers are based on different approaches depending on the information provided by the syntax tree. Consequently, analysers were developed for Parameter, Operator Precedence and Parenthesis class errors. Where appropriate, contextual information relating to the error such as function names or the number of arguments were gathered to boost the clarity of the error message.

Based on feedback from user-testing, error messages from the analysers helped provided a clearer image of what needs to be fixed in order to correct the error messages. With the exception of certain hints from the Operator Precedence Analyser and the Parameter analyser, volunteers appreciated the error messages that the analysers provided. In terms of functionality, most of the analysers fail to accurately deliver correct error messages especially in complex code conditions. This makes them unsuited for use in a real coding environment. However, the analysers have proven that it is possible to improve upon obscure error messages without having to rewrite the compiler messages themselves. This is achieved through utilising contextual information gathered from the syntax trees.

The analysers utilises the popular F# extension Ionide for use in Visual Studio Code in order to run analysers. This limits the applications of the analysers as they are designed for Ionide in mind, though modifications can be made to apply the analysers to other languages and platforms. Given the scope of work, analysers can always be refined and improved upon. Additional analysers can be built targeting other potential errors listed in the report and existing analysers can be improved upon, for example by increasing functionality under complex code conditions. New approaches may be also taken to provide extra contextual information to further bolster error correction messages of the analyser.

13. Appendix

13.1. User Guide

The source code for the project can be found on `https://github.com/jovanhan2/F-Sharp-Analyser`.

To install and test the custom analysers, the following application requisites need to be met

- Visual Studio Code 1.35.1
- Ionide-fsharp 3.33.0. (Note that downgrading is possible through VS code's extension interface)

After installing Ionide, downloaded the packed .nupkg file from the root folder of the Github. Now, several options need to be enabled within Visual Studio Code.

- Open the settings page in Visual Studio Code (File → Preferences → Settings)
- Search for FSharpėnableAnalyzers in the search box and ensure the FSharpėnableAnalyzers option is selected
- Restart Visual Studio Code

An optional scaffolding repository can be cloned from github at this link `https://github.com/jovanhan2/FSharp-Custom-Analysers-Usage`. The repository contains the .nupkg file placed in the correct directory. Ensure that FSharp.analyzersPath is not set. Instructions below.

- Next, check if the FSharp.analyzersPath option is set. If it is, the settings.json file will contain a property.
- By default the FSharp.analyzersPath should not be set. This property determines where to place the downloaded nupkg in a new FSharp project.
- If the FSharp.analyzersPath is not set, place the downloaded .nupkg into packages/Analyzers directory. If using the scaffolding repository, this is already done for you

The analysers should now be activated. Further instructions can be found at the scaffolding repository or at the main repository of the project.

References

[1] "Ionide." [Online]. Available: http://ionide.io/

[2] Ndmitchell, "ndmitchell/hlint," Jan 2019. [Online]. Available: https://github.com/ndmitchell/hlint

[3] "Continuous code quality." [Online]. Available: https://www.sonarqube.org/

[4] "**f sharp (programming language)," Dec 2018. [Online]. Available: https://en.wikipedia.org/wiki/F_Sharp_(programming_language)

[5] "Twenty six low-risk ways to use f# at work." [Online]. Available: https://fsharpforfunandprofit.com/posts/low-risk-ways-to-use-fsharp-at-work/

[6] Cybermaxs, "Cybermaxs/awesome-analyzers," Nov 2018. [Online]. Available: https://github.com/Cybermaxs/awesome-analyzers

[7] Gewarren, "Code analysis using roslyn analyzers - visual studio." [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/code-quality/roslyn-analyzers-overview

[8] J. Robichaud, "dotnet/roslyn." [Online]. Available: https://github.com/dotnet/roslyn/wiki/RoslynOverview

[9] ***K.F.Tomasdottir, M. Aniche, and A. van Deursen, "Why and how javascript developers use linters." IEEE, 2017, pp. 578–589.

[10] *K.Liibert, "12 best tools for software developers - dzone agile," Oct 2018. [Online]. Available: https://dzone.com/articles/12-best-tools-for-software-developers

[11] "Travis ci - test and deploy your code with confidence." [Online]. Available: https://travis-ci.org/

[12] "Visual studio code - code editing. redefined," Apr 2016. [Online]. Available: https://code.visualstudio.com/

[13] "Visual studio marketplace." [Online]. Available: https://marketplace.visualstudio.com/

[14] "Markdown all in one - visual studio marketplace." [Online]. Available: https://marketplace.visualstudio.com/items?itemName=yzhang.markdown-all-in-one

[15] K. Cie, "Introducing f# analyzers – lambda factory – medium," Sep 2018. [Online]. Available: https://medium.com/lambda-factory/introducing-f-analyzers-772487889429

[16] "Github." [Online]. Available: https://octoverse.github.com/2017/

[17] Pocke, "pocke/awesome-lint," Feb 2016. [Online]. Available: https://github.com/pocke/awesome-lint

[18] M. Mcveigh, "Fsharplint." [Online]. Available: http://fsprojects.github.io/FSharpLint/

[19] **M.Kranc, "The rise and fall of scala - dzone java," Oct 2016. [Online]. Available: https://dzone.com/articles/the-rise-and-fall-of-scala

[20] Sksamuel, "sksamuel/scapegoat," Jan 2019. [Online]. Available: https://github.com/sksamuel/scapegoat

[21] Gregvanl, "Getting started with roslyn analyzers - visual studio." [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/extensibility/getting-started-with-roslyn-analyzers?view=vs-2017

[22] DotNetAnalyzers, "Dotnetanalyzers/stylecopanalyzers," Jan 2019. [Online]. Available: https://github.com/DotNetAnalyzers/StyleCopAnalyzers

[23] ——, "Stylecopanalyzers special rules." [Online]. Available: https://github.com/DotNetAnalyzers/StyleCopAnalyzers/blob/master/documentation/SpecialRules.md

[24] "Troubleshooting f#." [Online]. Available: https://fsharpforfunandprofit.com/troubleshooting-fsharp/#FS0001A

[25] "Common programming mistakes for f# developers to avoid." [Online]. Available: http://lonelypad.blogspot.com/2012/09/common-programming-mistakes-for-f.html

[26] “Currying.” [Online]. Available: https://fsharpforfunandprofit.com/posts/currying/

[27] Cartermp, “Symbol and operator reference - f#.” [Online]. Available: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/symbol-and-operator-reference/index

[28] L. Kueh, “jovanhan2/f-sharp-analyser,” Jan 2019. [Online]. Available: https://github.com/jovanhan2/F-Sharp-Analyser

[29] D. Thomas, A.-D. Phan, T. Petricek, and M. Corporation. [Online]. Available: https://fsharp.github.io/FSharp.Compiler.Service/

[30] Fsharp, “fsharp/fsautocomplete,” Jan 2019. [Online]. Available: https://github.com/fsharp/FsAutoComplete

[31] Ionide, “100% cpu usage caused by fsharp projectcrackertools · issue #152 · ionide/ionide-vscode-fsharp.” [Online]. Available: https://github.com/ionide/ionide-vscode-fsharp/issues/152

[32] “Operator-precedence grammar,” Jan 2019. [Online]. Available: https://en.wikipedia.org/wiki/Operator-precedence_grammar

[33] “Bracket pair colorizer.” [Online]. Available: https://marketplace.visualstudio.com/items?itemName=CoenraadS.bracket-pair-colorizer