FearnLang

Developing a Custom Learner Programming Language

Centre Number: 61425

Candidate Number:

Table of Contents

Introduction

The objective of the project is to build a programming language. Specifically, the *Fearn* Programming Language (a portmanteau of *Facos* and *Learn*), hereafter referred to as FearnLang or Fearn, is designed to be a transitional language for beginners, shifting from beginner languages to more intensive systems languages.

Consider a beginner like myself. I started out learning Python in early 2019, at a robotics club. Spending a few years building more and more complex programs, and familiarising myself with the environment within software development, I quickly became aware of the discourse surrounding programming languages, their utility, and their weaknesses. Python and JavaScript are considered good starting points for those 'getting their feet wet' with programming, due to their simple syntax and capabilities to make impressive and visual solutions with relatively minimal effort, from command-line apps to websites and games. However, like me, once they immerse themselves with programming terms and culture, and are more familiar with the basics (control-structures, functions, simple OOP, etc), they'll want the next language they learn to be one that provides them with greater performance. This is often when beginners, including me, meet systems languages like C, C++, and Java. However, this is also where many (other-wise capable) developers give up, due to the change in syntax and difficulty they encounter, from memory-management, to strong and explicit type systems, fixed-length arrays, and the more complex process of trying to debug code in a compiled language.

The ultimate aim of this project is to encourage learners by giving them a simple, C-like language with which they can become more familiar with these concepts, while also being in a more forgiving environment. A broad structure for this project would be something like:

- Define the syntax and features of the MVP of the language
- Build a Compiler, which will read in a text source file, and produce some sort of objective code (e.g. assembly, machine code, bytecode).
- Define/Develop a runtime to execute generated code

Analysis

Background

A programming language is a broad term for dynamic notations used to create human-readable instructions that a computer can interpret and execute. There can be split into 2 levels of abstraction, and both of interest to this project: High-level and Low-level.

Low-level languages are those that a comprised of a series of restricted, simple instructions that can be executed in one or a few cycles by a computer processor; they are difficult to solve complex tasks with, due to how simple and atomic the instruction sets are, and the instructions useable changes depending on CPU architecture, making them difficult to port between devices.

High-level languages are more human-readable and easy to use to complete complex tasks. These come in a far greater variety of forms than low-level languages, such as visual block-based languages (like MIT's Scratch), graph-based (such as Unity's Visual Scripting), and logic languages like Prolog. The language I intend to create is a text-based imperative language, by far the most popular category in software development. This includes scripting languages like Python, JavaScript, and Lua; additionally, there are more intensive, 'mid-level' languages used to develop systems in industry, games, and other applications that require a large amount of resources to run. Examples of these are C++, Java, Rust, Go, Dart, and many others.

The primary differences between scripting languages and systems languages are that:

- Systems languages provide greater performance and efficiency, as they are compiled and often allow for precise memory management
- Scripting languages are easier to work with for beginners, as they are interpreted (making development faster), and rely on larger programs to do the 'heavy lifting'. An example of this is JavaScript, which allows for very rewarding and visual development of websites and games, while relying on browsers to execute these instructions efficiently

These two categories aren't isolated, and complement each other. Often, scripting languages, as well as query and markdown languages, are used to give large, complex systems (written in systems languages) instructions on what to do to complete a task. A good example of this is Game Engines, like Unity, Unreal, or Godot. These all allow for scripting in custom languages, common scripting languages, and sometimes graph-based visual programming, yet they are implemented using lower-level languages that can run much faster, and implement these instructions efficiently, while this nuance and complexity is abstracted from the developer at build-time.

The Fearn Programming Language

The C programming language was created in the 1970s by Dennis Ritchie of Bell Labs. The purpose of this project was a high-level programming language that developers could use to implement large-scale systems, like Bell Labs' Unix Operating System. Many languages draw their primary syntax, and basic control structures, to C, including every single example I've named above, and so this will be a large inspiration for my own language.

To create a programming language, I need to develop a translator program, which will validate and reduce source code down into simple instructions, such as bytecode or Assembly. The two types are an interpreter and a compiler; as most systems languages opt for the latter, to improve execution times, this is the route I am following. A compiler is a program which reads in a text file containing source code following a specific language syntax (such as Java, C, or Fearn), and produces object code that a computer can understand and execute. This is done using a process that can be broadly split into front-end and back-end tasks, as seen in the below table.

Front-End *(Takes in the program, and processes it into an intermediate representation, raising errors when anything is invalid)*

- **Lexing** (splitting the source into token)
- **Parsing** (Composing those tokens into an Abstract Syntax Tree)
- **Semantic Analysis** (Validating the program)

Back-End *(Generates code that can be executed, and optimises it for maximum speed at runtime and minimal file size)*

- **Optimisation** (Generating an intermediate code and applying a ruleset to improve
- **Code Gen** (writing executable code to a binary file)

Research

Stack Overflow Developer Survey 2023 – Market Research

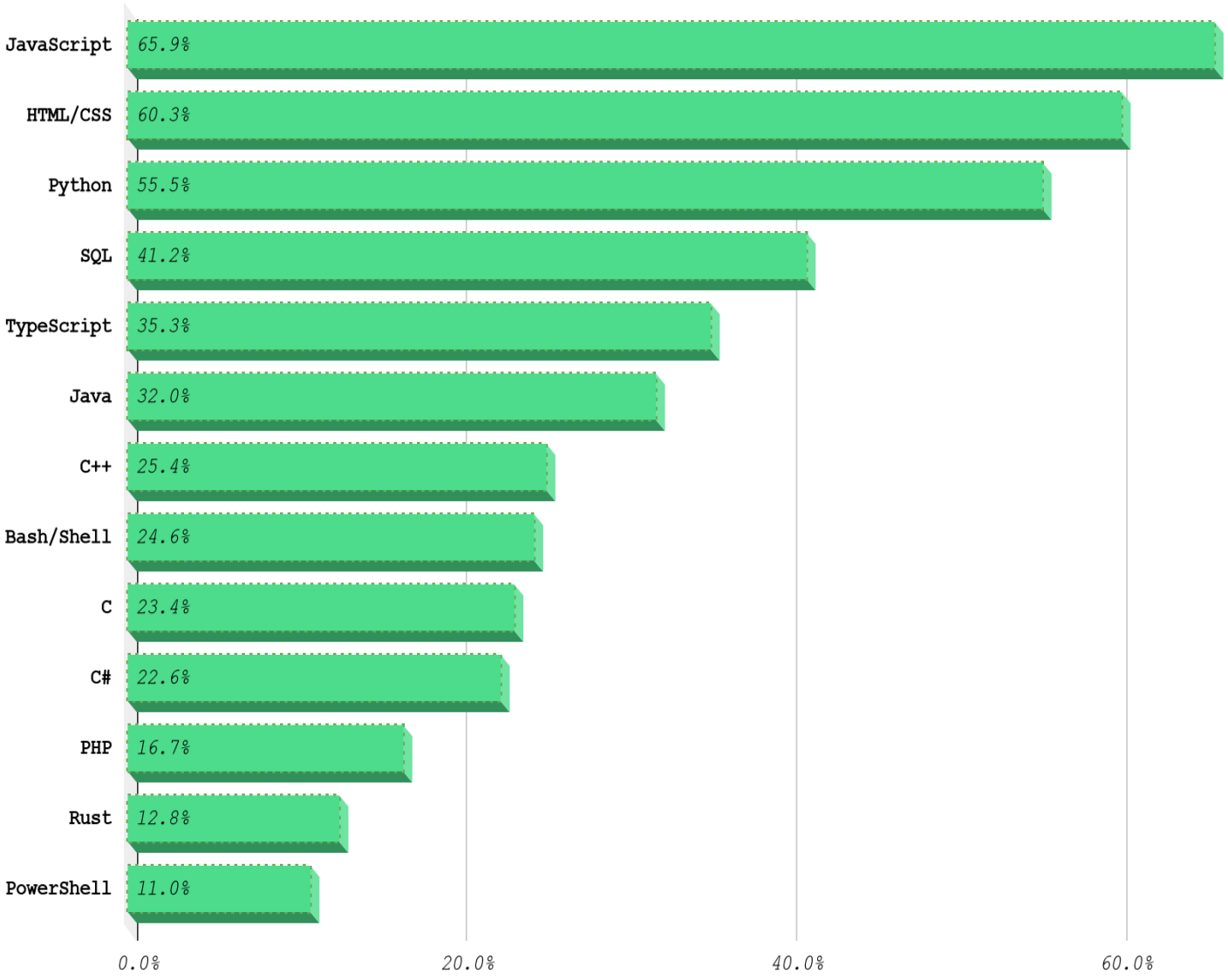
At the beginning of the project, I needed to confirm exactly which languages are used by beginners the most. This was useful as it allowed me to examine the syntax used by these languages that could be adapted for Fearn, to make the language more comfortable to transition into. To do this, I used Google Colaboratory to examine the results of the 2023 Stack Overflow Developer Survey, which surveyed over 90,000 developers to uncover developer demographics, the technologies they use and enjoy, technology they want to use in the future, etc.[1] Using Pandas and Matplotlib in Python, I imported the CSV file of all public responses to the survey, cleaned them, and then filtered them by the experience of the programmer, defining a beginner as someone who has been programming for 5 years or less. I then calculated the percentage of programmers who mentioned using a particular language, for each of the 51 languages that Stack Overflow surveyed.[2]

My results show that the most common languages used were JavaScript (which 66% of respondents had used), HTML/CSS (60%), Python (56%), SQL (41%), and TypeScript (36%). HTML and CSS are both markdown/styling languages, and SQL is used for queries; however, this leaves me to take inspiration from languages like JavaScript, TypeScript, and Python when designing my syntax, to ensure the user's comfort.

[1] Stack Exchange Inc. 2023. "Stack Overflow Developer Survey 2023." Stack Overflow. 2023. https://survey.stackoverflow.co/2023/.

[2] Stack Overflow Beginner Language Analysis: https://github.com/tjfacos/FearnLang/blob/main/Analysis/StackOverflowSurvey/StackOverflow_Survey_Analysis.ipynb

Figure 1: Languages Favoured by Beginners



Third Party

In order to clarify the essential aspects of language design to focus on, I interviewed Leo Spratt, a Computer Science student from Canterbury Christ Church University, as my third party. He has multiple years of programming experience, having worked in both scripting and markdown languages (Python, JavaScript, HTML, CSS, etc), as well as lower-level systems languages, particularly C/C++, and C#.

His suitability as a third party on this project stems from two main factors:

- **Programming Experience**: Spratt isn't a beginner to programming, and is accomplished as a developer. However, still being a student, he is able to draw on his own experiences learning to program to inform me what aspects of my project are the most important, and how best to go about them.
- **Language Building Experience**: Spratt has developed his own, BASIC-like language, LeoLang[3]. This is a bare-bones language, supporting variables, input/output, and mathematical operations, and which is transpiled to C - meaning the program is translated from LeoLang to the C language, and then a C compiler like GCC or clang can be used to convert it into an executable.

Question 1

What are the most important aspects of a programming language for a beginner, compared to a more advanced programmer?

Response

- Manual memory management is not required
- Redeclaring variables is required when changing its type
- Reduced amount of syntax needed for simple programming tasks such as defining a variable or performing mathematical operations

Analysis

- The importance of manual memory management, such as the use of pointers and memory allocation/deallocation in C, becomes more prevalent over time, as developers want to create programs that value performance and (both space and time) efficiency, over the speed and ease of the development process
    - From this, I should put consideration into how I could include memory features in my language. This could include explicitly-types pointers, explicit memory allocation and deallocation (like the malloc and free commands in C), or references to variables
- Spratt's second point refers to strongly-typed variables, and that the data type a variable stores can't change in those languages favoured by advanced programmers

[3]The repository for LeoLang can be found on GitHub: https://github.com/enchant97/leo-lang/

    - This feature of strongly typed variables will be essential to my language, as it is to a vast majority of C-like systems languages
- The third point concerns the amount of boilerplate code required for simple programs. Systems languages aren't designed for the smaller scale explorations and tasks that scripting languages may be better for
    - On one hand, I shouldn't eliminate all the boilerplate code, as that's unrealistic for a systems language; on the other hand, I shouldn't allow the language to become too bloated, and unpleasant to use. A good compromise may be similar to C or Go, where execution starts at a named main procedure.

Hello World in 3 Languages

Python (scripting)

```python
print("Hello, World!")
```

Java (systems)

```java
class HelloWorld
{
    public static void main(String args[])
    {
        System.out.println("Hello World");
    }
}
```

Go (systems), the 'goldilocks' zone

```go
func main() {
    fmt.Println("Hello, World!")
}
```

Question 2

For a beginner with experience with a high-level language (Python, JavaScript, etc), which languages would you recommend for them to expand their skills (particularly Systems Languages like Java, C#, C++, Rust etc), and why?

Response

- Go (errors are returned instead of raised, static typing, no OOP, has a GC *[Garbage Collector]*)
- Kotlin (less boilerplate code compared to standard Java)
- C (learn about how memory actually works)

Analysis

- The Go programming language is used to create industry systems, and is developed/maintained by Google. It has several attractive features, that I should consider implementing at some stage of my program's compilation or runtime processes:
    - **Informative Error Raising;** this will be vital to explain to the user what's wrong with their code, so they can learn from the experience
    - **Static and Explicit Typing**, which I've already addressed as vital
    - **Memory Handling**, through pointers or explicit allocation methods. These features would be best if I wanted my language to provide more academic, theoretical learning on computer architecture, as opposed to purely practical experience (making them a better programmer)

Question 3

For learners transitioning to more traditional harder languages, what features do you think it's important for them to familiarise themselves with (examples would be pointers, fixed-length arrays, binary operations, etc.)?

Response

- Pointers
- Not using OOP
- Reference vs owned memory

Analysis

- These points are quite straight forward, and link well into points I've already raised, particularly pointers and memory management
- In order to limit the scope of my project, I intend to restrict my language to be purely procedural, and not to include any Object-Oriented features
    - Object-Oriented Programming has become more controversial in recent years, due to the complexity it can introduce into programs.[4] Thus, not allowing users to use classes may allow them to get use to other methods, that produce safer, more maintainable code in the long run

Question 4

What is your favourite programming language at the moment, and why? Also, what is your least favourite language and why?

Response

- Rust is currently my favourite as it allows for system access whilst having great memory management through a borrow checker and the way that errors are returned.
- JavaScript/TypeScript as certain operations have unexpected results a good place listing them all is: https://github.com/denysdovhan/wtfjs. It also utilises null, undefined and NaN.

[4] Talin. 2018. "The Rise and Fall of Object Oriented Programming." Machine Words. Medium. November 23, 2018. https://medium.com/machine-words/the-rise-and-fall-of-object-oriented-programming-d67078f970e2

Analysis

- An important aspect of my language should be predictability, so that it's obvious what will happen beforehand
    - One idea to help this may to restrict the operations one symbol can do, to just one purpose (for example, not allowing `'+'` for string concatenations)
- Rust is a good example of memory safety and error handling
    - It utilises a memory ownership system, to better manage heap memory and prevent redundancy or duplication of data in memory[5]
    - The error messages Rust provides are verbose and informative, providing messages in the context of the program, often using the same identifiers you (the developer use)

Question 5

Are there any programming practices that you believe my language like should encourage or prevent, in order to improve the coding practices of the user?

Response

- Encourage: Return errors
- Prevent: Multiple nullable types (like JS)
- Encourage: Starting indexes from 0
- Prevent: OOP features like inheritance (can cause confusion when methods are overridden)
- Encourage: Use indentation instead of brackets
- Encourage: Restrict to only allow one operation on a single line e.g. don't allow declaring two variables on one line

Analysis

- The first few points link into the strong type system I want for this language
    - Return errors refer to a function, with an explicit return type, returning data of a different type, or no data at all
    - FearnLang must restrict itself to, at most, one null type
- Again, Spratt expressed that Object-oriented design leads to confusion - one of the reasons it will not be included in FearnLang
- Spratt recommends that use of indentation for code nesting, as opposed to curly braces. This is an interesting point as…
    - On one hand, most C-like systems languages are 'curly brace' languages,

[5] Klabnik, Steve, and Carol Nichols. 2022. "What Is Ownership? - the Rust Programming Language." Doc.rust-Lang.org. 2022. https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

where indented code blocks are closed off with {} ○ On the other hand, languages like Python, which use a tabbed structure enforce code cleanliness, which is a useful style worth enforcing, to make programs easier to read and enforce ● Restricting the user to one operation to one line will likely make parsing the code easier, so I'm more than happy to comply with this point
Question 6 If you have any ideas for objectives, or any other comments, please add them here
Response Focus first on features that are core for a programming language such as: variables, mathematical operations and input/output. You may find it interesting to have a look at my experimental language I tried creating a few years ago: https://github.com/enchant97/leo-lang (it was based on the BASIC language).
Analysis ● This point will be helpful in setting objectives to produce a Minimum Viable Product (MVP) ○ One option may be to start by just restricting the user to strings and integers as data types, before implementing more later

Key Takeaways

The key points from Spratt's feedback are:

- FearnLang's Design should encourage understanding of memory management, through pointers, as well as possibly C-style explicit memory allocation and deallocation.
- Enforce strict and explicit memory control, both for variables and functions (return types, parameters and arguments, etc).
- Encourage the beginner developer to produce clean, terse code, that's easy to read and maintain. Particular examples of this would be
  - Mandate discipline when using tabs and whitespace
  - One operation to a line
- Encourage predictable programs, by...
  - Using, at most, one null type
  - Performing rigorous type checking
  - Not supporting OOP
- Make sure the language supports the developer well, through Garbage Collection and Informative Error Raising

Analysis of Existing Languages

JavaScript

Sample Program

```
/*
FizzBuzz Program
Iterate over each number 1 to n.
*   If n is a multiple of 3, print Fizz
*   If n is a multiple of 5, print Buzz
*   If both, print FizzBuzz
*   Otherwise, print the number
*/
function FizzBuzz (n)
{
    for (let i = 1; i <= n; i++)
    {
        let output = "";

        if (i % 3 == 0) {
            output += "Fizz"
        }

        if (i % 5 == 0) {
            output += "Buzz"
        }

        if (output) {
            console.log(output)
        } else {
            console.log(i)
        }

    }
}
// Call FizzBuzz from 1 to 100
FizzBuzz(100)
```

Introduction

- JavaScript is a language which is the standard for web scripting, and for writing programs to execute within a web browser or using a web interface, such as apps and games.
- It's also used to develop server-side applications, such as REST APIs, through a run-time like Node.js.[6]

Features

- Multi-paradigm language, supporting imperative, object-oriented, functional features (such as first class functions, taking other functions as arguments).
- Dynamically typed - meaning that variables can change type during execution, and no errors are raised due to type issues unless the program explicitly checks.
- Includes many built-in libraries and functions

Implementation

- All facilities of JavaScript are provided internally by objects, created dynamically at runtime.[7]
- Managed by the garbage collector (part of the runtime or JIT compiler), which tracks when memory needs to be allocated, and when it can be released.
    - The core concepts of this design are references and reachability; all values determined to be reachable are guaranteed to be in

[6] MDN Contributors. 2019. "JavaScript." MDN Web Docs. July 19, 2019. https://developer.mozilla.org/en-US/docs/Web/javascript.

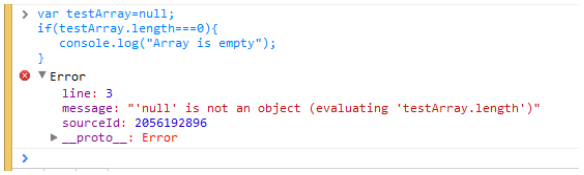
[7] ECMA International. 2023. "ECMAScript® 2021 Language Specification." Tc39.Es. September 22, 2023. https://tc39.es/ecma262/.

memory, such as the local function and its local variables; the most essential reachable values (often reached directly by the global or local scope in a program) are called roots. If an object isn't reachable (not connected by a chain of references to a root), then it is disposed of, so the memory is made available for use by other data.[8]

Syntax

- JavaScript is a curly braces language, so nested code blocks don't need to be indented
  - This has long been a standard in procedural languages, but it can encourage code to be messy and hard to read, as it doesn't enforce how to present a program
- The different versions and standards of JavaScript allow for multiple different ways of writing out the same thing
  - For example, to define a function, you can use `function MyFunc () {}`, or `let MyFunc = () => {}`
  - This creates ambiguity and confusion, especially if a team of developers approach the same problem differently
- `let`, `var`, and `const` are all used to declare variables, creating further inconsistencies

Error Raising



- Varies by runtime/browser
- Refers to the line number, type of error, and what was being evaluated
  - This is helpful for identifying where the problem exists, for further investigation, but may not be enough to fix it on its own

Takeaways

- JavaScript is a scripting language, which many of my potential users could be moving from
  - Many of the basic features of the language should be included in FearnLang, to help the developer feel comfortable using the language
- Error raising should be as informative as possible
  - Line and column number
  - In-context messages, using the user-defined identifiers if possible
  - Provide as much detail as possible

[8] Kantor, Ilya. 2022. "Garbage Collection." Javascript.info. October 14, 2022. https://javascript.info/garbage-collection.

- The syntax must be unambiguous, and consistent throughout
- A garbage collector, or some other automatic memory management, to detect and resolve memory leaks, is vital to ensure an inexperienced developer doesn't create dangerous problems with their code
    - This could be done during Semantic Analysis, rather than at runtime, by traversing the Abstract Syntax Tree and making sure all heap memory will be released

Kotlin

Sample Program

```
// Euclid's Algorithm to find the
// Greatest Common Divisor (HCF)
fun gcd(A: Int, B: Int): Int {
  if (A == 0) {
       return B
  } else if (B == 0) {
       return A
  }

  return gcd(B, A % B)
}

fun main() {
  print("Enter A:")
  val A: Int =
Integer.valueOf(readLine())

  print("Enter B:")
  val B: Int =
Integer.valueOf(readLine())

  val X: Int = gcd(A, B)
  println("GCD is $X")
}
```

Introduction

- Kotlin is a cross-platform, general-purpose programming language, designed to interoperate with Java - a long-standing tool for industry software
- It is used for a variety of tasks, but is most notable for being the primary language used to write Android apps

Features

- Multi-Paradigm (Procedural, Functional, Object-oriented)
- Large standard library, built-in modules, and facilities to import user created modules
- Kotlin is statically-typed, meaning a full type-check is run during compilation. The language is strongly typed, but not explicitly (variable types can be inferred by the compiler, but cannot change)

Implementation

- Kotlin can be used as a system or a scripting language, and has a JIT compiler to support both
- It targets the Java Virtual Machine, a process virtual machine that simulates traditional computer architecture
    - That means the bytecode Kotlin's compiler produces can be run on any device which has the JVM, making it highly portable
    - It can also target WebAssembly (allowing it to run in the browser), and native assembly[9]

[9] JetBrains. 2018. "Kotlin/Docs/Contributing.md at Master · JetBrains/Kotlin." GitHub. 2018. https://github.com/JetBrains/kotlin/blob/master/docs/contributing.md.

Syntax

- As in many C-like languages, execution of Kotlin programs starts at a named main procedure
- Variables are declared using the `'val'` keyword, followed by an identifier
    - Specifying a data type is optional
- Kotlin features null-safety
    - The programmer needs to specifically state if a variable can have a null value
- Kotlin was designed to be more concise and easy to use compared to Java[10]

Error Raising

```
main.kt:11:1: error: a 'return' expression required in a function with a block body ('{...}')
}
^
```

- Identifies the row and column number at which the error occurs
- States the rule that was broken, but doesn't include context (for example, the function name)

Takeaways

- Kotlin is a good example of a light-weight language, which has clear and concise syntax
- It doesn't feature a lot of the memory features that I want to include in Fearn, such as pointers

[10] JetBrains. 2023. "FAQ - Help | Kotlin." Kotlin Help. 2023. https://kotlinlang.org/docs/faq.html#what-advantages-does-kotlin-give-me-over-the-java-programming-language.

C

Sample Program

```c
#include <stdio.h>

// Bubble Sort in C, using pointers
void bubbleSort(int *arr, int n) {
  int i, j, temp;
  for (i = 0; i < n - 1; i++)
  {
       for (j = 0; j < n - i - 1; j++)
       {
       // Bubble largest item to the
end
       if (*(arr + j) > *(arr + j + 1))
       {
       temp = *(arr + j);
       *(arr + j) = *(arr + j + 1);
       *(arr + j + 1) = temp;
       }
       }
  }
}

int main(void) {
  int Arr[] = {10, 22, 3, 41, 103, 7};
  bubbleSort(Arr, 6);

  for (int i = 0; i < 6; i++)
  {
       printf("%d ", Arr[i]);
  }

  return 0;
}
```

Introduction

- C is a programming language developed in the 1970s by Dennis Ritchie, at Bell Labs
    - Bell Labs also created the UNIX operating system, which forms the basis of many modern OS programs. It was written in C to make it portable across processor architectures
- The problem it solved was to give programmers a higher level language to create software in
    - At the time, a lot of work was done manually, in Assembly, with few firm standards between architectures

Features

- C is a purely procedural language
- It provides a standard library, but not many built-in functions that don't have to be explicitly included by the preprocessor
- C is a language that uses preprocessor directives, such as includes and macros
    - These can sometimes cause confusion as to their purpose, leading to mistakes
- C has a strong, explicit type system
- Supports pointers and pointer arithmetic, but doesn't perform any checks
    - This can lead to OS errors far later down the line, that may not be caught in development
- Allows manual memory allocation using malloc and free
    - This can create extra complexity and lead to errors down the line
    - Improved performance and space efficiency

Implementation

- As C is an ANSI standard, anyone can create a C compiler, and there are many in use
- GCC is the GNU C Compiler, which comes with GNU/Linux distributions. It's a traditional command line compiler, which simply compiles the program to a native binary, in an output file
- Clang is a compiler that's built on the LLVM toolset, used to create compilers, and which could be helpful to my project.
    - LLVM allows compiler developers to generate a cross-platform intermediate code, LLVM IR
    - LLVM will then perform optimisations and generate a native binary, eliminating the need for me to develop a dedicated runtime, though memory safety will need to be ensured at compile time
    - This toolset is also employed by the compiler for Rust
- C also has compilers in existence to target WebAssembly

Syntax

- C has a concise syntax, with a main function acting as the start point
- Low number of reserved words
- Simple syntax that forms the basis of most mainstream programming languages

Error Raising

- Simple, giving line and column numbers
- Don't provide context
- Varies by compiler
- Runtime errors are often hard to decipher

Takeaways

- When considering memory management, I must consider what practices could be dangerous, and either stop them or monitor how they are used at runtime
- LLVM could be a useful tool for my project, but may also add complexities when working around it and making sure the programs produced are memory safe

Objectives

1. Define a syntax for FearnLang
    1.1. Must allow the user to create variables, of an explicit data type
    1.2. Allow for Mathematical operations (Addition, Subtraction, Multiplication, Division, MOD, DIV)
    1.3. Allow for text Input/Output in the terminal
    1.4. The language must support string, integer, and boolean data types
    1.5. Fearn must be able to cast between the 2 primitive data types
    1.6. Must be able to create fixed-length arrays
    1.7. Allow for the use of C-style pointers
    1.8. Must allow for the use of Boolean Logic
2. Develop a Fearn Compiler
    2.1. Allows the user to call it from the command line, passing in a file name with a *.fearn* extension
    2.2. Read the file out, and parse it with into an Abstract Syntax Tree
        - Since I will be working in C++, I can use GNU Bison and Flex to parse the file
    2.3. Represent the AST as interconnected objects in C++, which point to each other
    2.4. Perform a Semantic Analysis on the AST
        - This will involve enforcing a strong type system, and that expressions have the correct type in the given context
        - A symbol table will be used to make sure all identifiers have been declared at an appropriate point in the program, considering the order of instructions and scope
    2.5. Generate and Optimise Intermediate Code
        - This will involve constructing a control flow graph, and working iteratively to improve it until no further changes are made
        - This will include dead code elimination and constant propagation
        - This process can be done in memory, modelling each instruction as an object
    2.6. Write the Intermediate Code to a binary file
    2.7. Any errors raised must be *as informative as possible,* giving a column and line number, and ideally using in context information using variable and function names
3. Develop a Fearn Runtime
    3.1. Must take from the command line the name of the Fearn binary

3.2. Must follow the instructions contained in the file
3.3. Raise Errors if certain rules are broken, for example...
    - Data cannot be cast
    - List index out of the range
    - Pointers attempt to access data not allocated to the program

Extension Tasks

- Include support for floating point numbers
- Expand the standard library to include additional functions, such as common mathematical functions, or improved I/O
- Develop a VS Code Extension, for syntax highlighting and to compile and run Fearn programs quickly

Rationale for A-Level Standard

- The lexing and parsing processes will require me to define BNF and Regular Expressions, both of which are on the A-level syllabus
- The project relies heavily on Object-Oriented Programming, to model the Abstract Syntax Tree, and Intermediate Code
- The specifics of compilation are beyond the scope of AQA A-level Computer Science, though they appear on some other specifications, and the techniques used (programming in C++, compiler design and construction, parsing and lexical analysis) are mostly left to university Computer Science
- Semantic Analysis will require me to implement Post-Order Tree Traversal to verify data types, scopes, and the definitions of identifiers
- Optimisation over a Control Flow Graph will require graph traversal

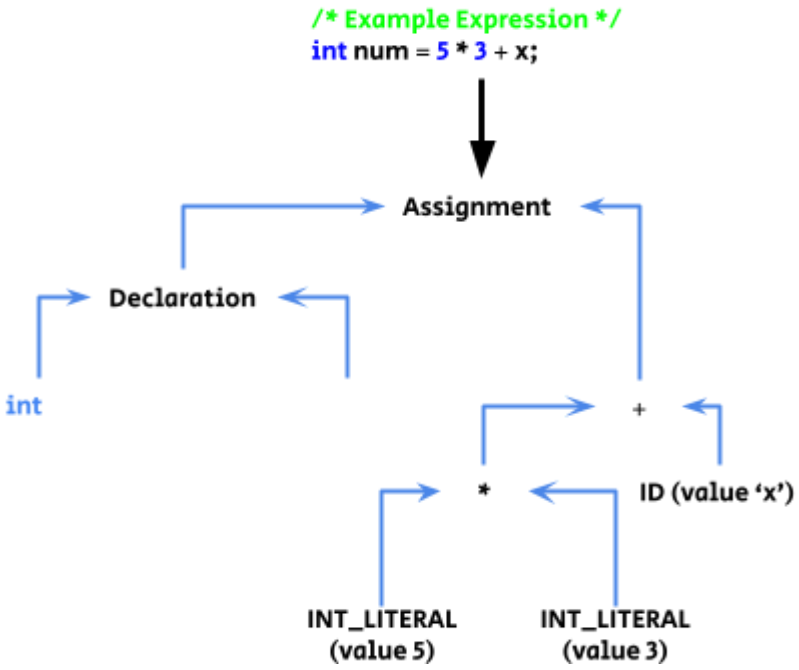
Modelling

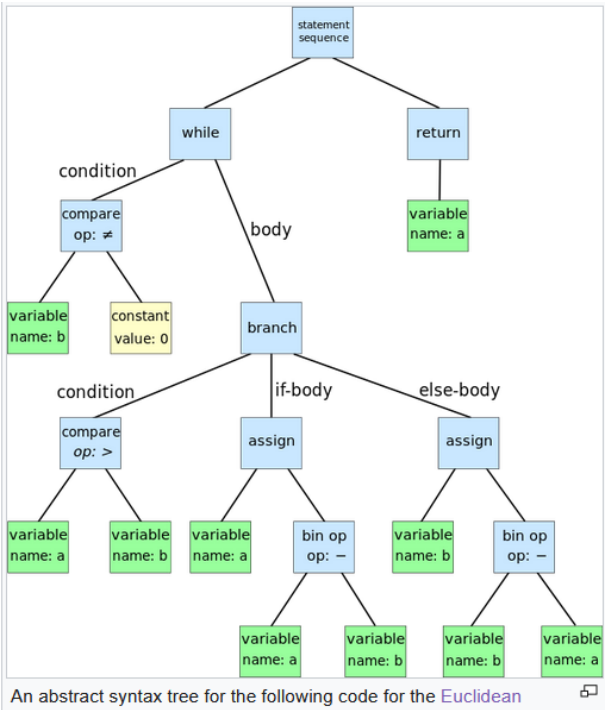
Prototype Lexing and Parsing Files

For my project, I plan to use GNU Bison, a parser generator, and Flex, a lexer generator.[11] As a result, I prototyped by writing .y and .l files that specify a grammar close to the one I plan to use for FearnLang, based on similar files for ANSI C.[12,13] These prototype files can be found in Appendix α.

Logo for GNU Bison

Abstract Syntax Tree (AST) Models





An abstract syntax tree for the following code for the Euclidean algorithm:

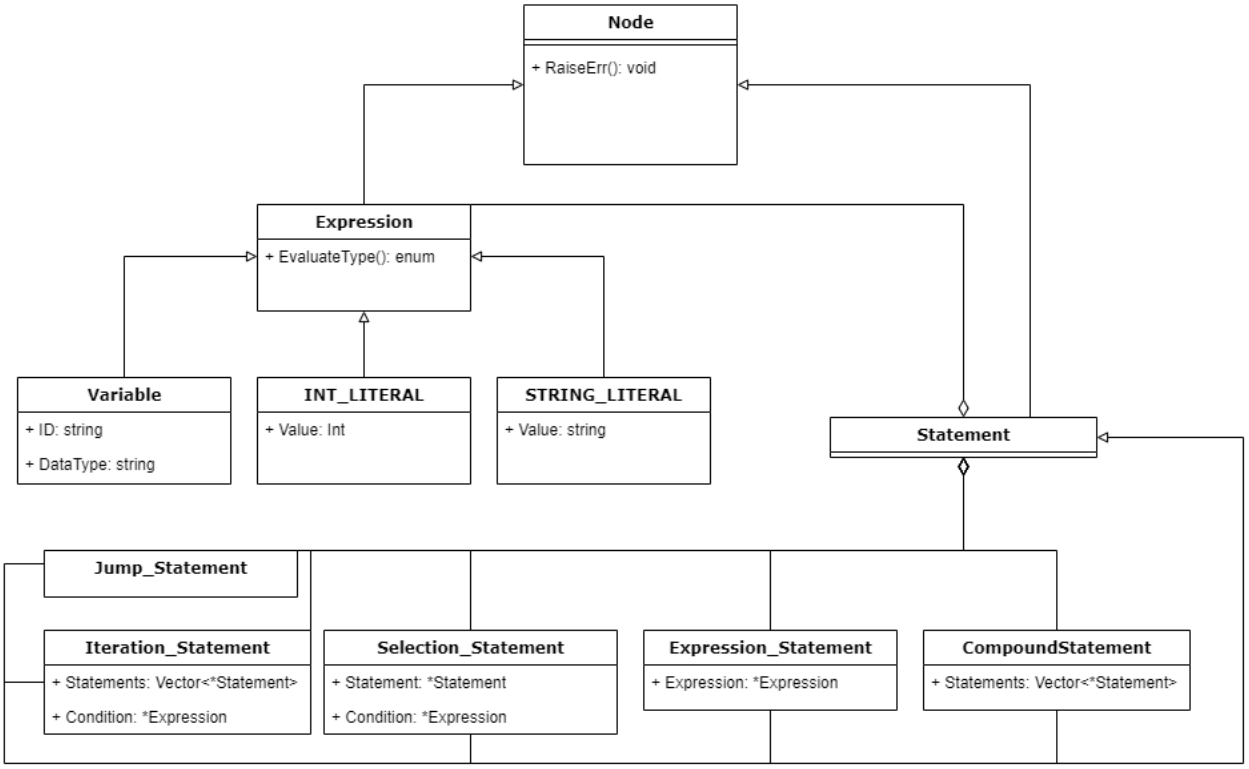
```
while b ≠ 0:
    if a > b:
        a := a - b
    else:
        b := b - a
return a
```

[11] These applications are sometimes referred to as 'compiler compilers'

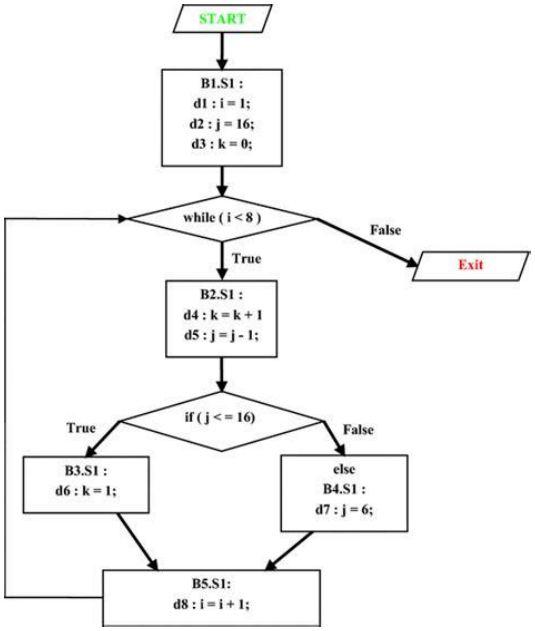
[12] Lee, Jeff. 1985. "ANSI C Grammar (Lex)." www.lysator.liu.se. 1995. https://www.lysator.liu.se/c/ANSI-C-grammar-l.html.

[13] Lee, Jeff. 1985. "ANSI C Grammar (Yacc)." www.lysator.liu.se. 1995. https://www.lysator.liu.se/c/ANSI-C-grammar-y.html.

AST Class Diagram[14]



Control Flow Graph (with example Intermediate Code)[15]

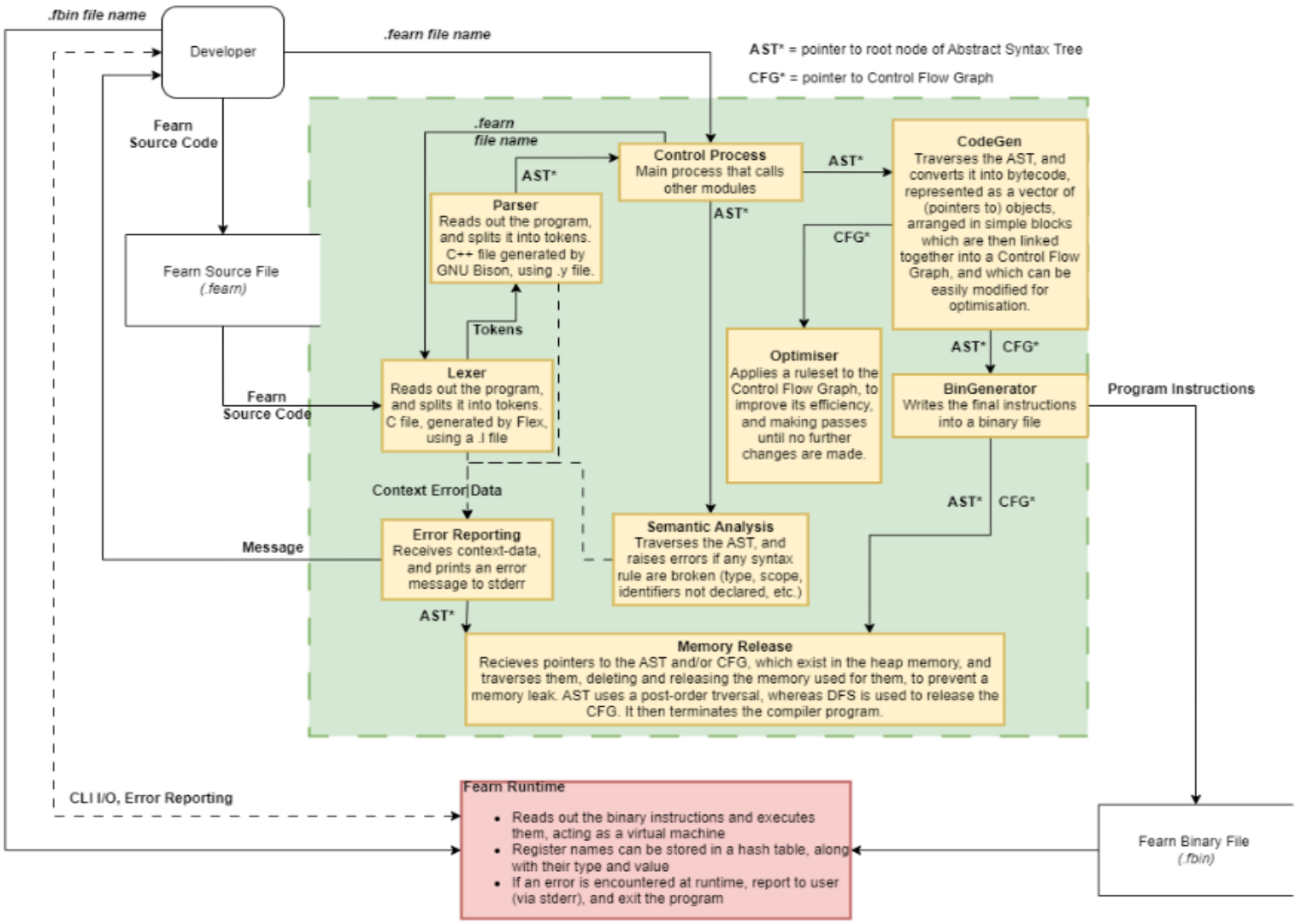


[14] This is only a sample of the classes that will be present, as representing all of them (such as the subclasses of expressions) is infeasible in a diagram

[15] A computational model that represents sequential blocks, and how control is transferred between them. It will be useful when performing optimisations.

System Overview Flowchart

Green is processes contained within the Fearn Compiler (FearnC)
Red denotes the Fearn Runtime (FearnRun)



Appendix α – Model Grammar

Prototype GNU Bison and Flex Files

Fearn_Parser_Model.y

```
%{

    #include <stdio.h>
    #include <stdbool.h>

    FILE *yyin;

    int yyerror();
    extern int yylex();

%}

%union {
    int intval;
    double floatval;
    int boolval;
    char* strval;
}


/* Token Definitions (only compound symbols need tokens, e.g. +=, ++, etc.
) */
%token IDENTIFIER STRING_LITERAL INT_LITERAL FLOAT_LITERAL BOOL_LITERAL
%token INCREMENT "++" DECREMENT "--"
%token LESS_OR_EQUAL "<=" GREATER_OR_EQUAL ">=" EQUIVALENT "=="
NOT_EQUIVALENT "!="
%token AND "&&" OR "||"
%token MULT_ASSIGN "*=" DIV_ASSIGN "/=" MOD_ASSIGN "%=" ADD_ASSIGN "+="
SUB_ASSIGN "-="

%token STRING "string" INT "int" FLOAT "float" BOOL "bool" VOID "void"

%token IF "if" ELSE "else"
%token FOR "for" CONTINUE "continue" BREAK "break"
%token RETURN "return" LET "let"

/* Precedences */
```

```
%left "<" "<=" ">" ">="

%left "+" "-"
%left "*" "/" "%"

%start program

/* Rules - Reference: https://www.lysator.liu.se/c/ANSI-C-grammar-y.html
*/
%%

/* Names of the basic data types */
type_name
    : VOID
    | INT
    | STRING
    | FLOAT
    | BOOL
    ;

/* Specifier for return types, declarations etc.
Can either be a basic type, or list of elements of a basic type */
type_specifier
    : type_name
    | type_name '[' expression ']'
    | type_name '[' ']'
    ;

/* Whole Program is comprised of functions and global declarations */
program
    :
    | function program
    | declaration program
    ;

/* A Function is a return_type, identifier, parameters, and a compound
statement. */
function
    : type_specifier IDENTIFIER '(' ')' compound_statement
    | type_specifier IDENTIFIER '(' parameters_list ')' compound_statement
    ;

/* A list of parameters */
parameters_list
```

```
    : parameter
    | parameters_list ',' parameter
    ;

/* The written parameter to a function, in the form <type, identifier> */
parameter
    : type_specifier IDENTIFIER
    ;

/* The declaration of a new variable */
declaration
    : LET type_specifier IDENTIFIER ';'
    | LET type_specifier IDENTIFIER '=' expression ';'
    ;

/* Statements used to build up programs */
statement
    : compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    ;

/* If and If-Else statements */
selection_statement
    : IF '(' expression ')' compound_statement
    | IF '(' expression ')' compound_statement ELSE compound_statement
    | IF '(' expression ')' compound_statement ELSE selection_statement
    ;

/* Expression Statements evaluate to something */
expression_statement
    : ';'
    | expression ';'
    ;

/* For Loops */
iteration_statement
    : FOR '(' declaration expression_statement ')' compound_statement
    | FOR '(' declaration expression_statement expression ')'
compound_statement
    | FOR '(' ';' ';' ')' compound_statement
    ;
```

```
/* Flow - control statements */
jump_statement
    : CONTINUE ';'
    | BREAK ';'
    | RETURN ';'
    | RETURN expression ';'
    ;

/* Code Blocks (variable declarations must come before statements) */
compound_statement
    : '{' '}'
    | '{' statement_list '}'
    | '{' declaration_list '}'
    | '{' declaration_list statement_list '}'
    ;

/* List of statements (used inside code blocks) */
statement_list
    : statement
    | statement_list statement
    ;

/* List of variable declarations */
declaration_list
    : declaration
    | declaration_list declaration
    ;


/* Fundamental expressions */
primary_expression
    : IDENTIFIER
    | STRING_LITERAL
    | BOOL_LITERAL
    | INT_LITERAL
    | FLOAT_LITERAL
    | '(' expression ')'
    | '[' expression ']' // For lists
    ;

/* Expressions with some sort of suffix (e.g. function calls, indexing) */
postfix_expression
    : primary_expression
```

```
    | postfix_expression '[' expression ']'
    | postfix_expression '(' ')'
    | postfix_expression '(' argument_expression_list ')'
    | postfix_expression INCREMENT
    | postfix_expression DECREMENT
    ;

/* List of arguements */
argument_expression_list
    : expression
    | argument_expression_list ',' expression
    ;

/* Operators on a single operand */
unary_operator
    : '+'
    | '-'
    ;
/* Assignment Operators */
assignment_operator
    : '='
    | MULT_ASSIGN
    | DIV_ASSIGN
    | MOD_ASSIGN
    | ADD_ASSIGN
    | SUB_ASSIGN
    ;

/* Expressions with unary operators */
unary_expression
    : postfix_expression
    | INCREMENT unary_expression
    | DECREMENT unary_expression
    | unary_operator cast_expression
    ;

cast_expression
    : unary_expression
    | '(' type_name ')' unary_expression
    ;

exponential_expression
    : cast_expression
    | exponential_expression '^' cast_expression
```

```
    ;

multiplicative_expression
    : exponential_expression
    | multiplicative_expression '*' exponential_expression
    | multiplicative_expression '/' exponential_expression
    | multiplicative_expression '%' exponential_expression
    ;

additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;

relational_expression
    : additive_expression
    | relational_expression '<' additive_expression
    | relational_expression '>' additive_expression
    | relational_expression GREATER_OR_EQUAL additive_expression
    | relational_expression LESS_OR_EQUAL additive_expression
    ;

equality_expression
    : relational_expression
    | equality_expression EQUIVALENT relational_expression
    | equality_expression NOT_EQUIVALENT relational_expression
    ;

and_expression
    : equality_expression
    | and_expression AND equality_expression
    ;

or_expression
    : and_expression
    | or_expression OR and_expression
    ;

assignment_expression
    : or_expression
    | unary_expression assignment_operator assignment_expression
    ;
```

```
expression
    : assignment_expression
    | expression ',' assignment_expression
    ;


/* Driver Program */
%%

int parser_main(int argc, char* argv[])
{

    if (argc == 0)
    {
        printf("ERROR: No file");
        exit(1);
    }

    FILE *file;
    fopen_s(&file, argv[0], "r");
    if (file) {
        yyin = file;
    } else {
        perror("Failed to open file.");
        exit(-1);
    }

    yyparse();

    if (file != NULL)
    {
        fclose(file);
    }

    return 0;
}

yyerror(char* s)
{
    fprintf(stderr, "ERROR: %s\n", s);
    exit(-1);
    return 0;
}
```

Fearn_Lexer_Model.l

```
%{
    /* Definitions */
    #include "Parser.tab.h"

    #include <stdlib.h>
    #include <stdio.h>
    #include <string.h>

    #define fileno _fileno

    int lineNumber = 1;
    int columnNumber = 1;

    void count();

%}

%option noyywrap
%option nounistd

digit       [0-9]
letter      [a-zA-Z_]
whitespace  [ \t\v\n\f]+


%x c_comment

/* Rules */

%%

{whitespace}                { count();                          }
"/*"                        { count(); BEGIN(c_comment);        }
<c_comment>\n               { count();                          }
<c_comment>.*"*/"           { count(); BEGIN(INITIAL);          }
<c_comment>.*               { count();                          }
"//".*                      { count();                          }

"for"                       { count(); return(FOR);             }
"break"                     { count(); return(BREAK);           }
"continue"                  { count(); return(CONTINUE);        }
```

```
"string"                    { count(); return(STRING);              }
"int"                       { count(); return(INT);                 }
"float"                     { count(); return(FLOAT);               }
"bool"                      { count(); return(BOOL);                }
"void"                      { count(); return(VOID);                }

"if"                        { count(); return(IF);                  }
"else"                      { count(); return(ELSE);                }

"return"                    { count(); return(RETURN);              }
"let"                       { count(); return(LET);                 }

"true"                      { count(); return(BOOL_LITERAL);        }
"false"                     { count(); return(BOOL_LITERAL);        }

{letter}({letter}|{digit})* { count(); return(IDENTIFIER);          }
{digit}+                    { count(); return(INT_LITERAL);         }
{digit}+\.{digit}+          { count(); return(FLOAT_LITERAL);       }
\".*\"                      { count(); return(STRING_LITERAL);      }


"<="                        { count(); return(LESS_OR_EQUAL);       }
">="                        { count(); return(GREATER_OR_EQUAL);    }
"=="                        { count(); return(EQUIVALENT);          }
"!="                        { count(); return(NOT_EQUIVALENT);      }

"&&"                        { count(); return(AND);                 }
"||"                        { count(); return(OR);                  }

"++"                        { count(); return(INCREMENT);           }
"--"                        { count(); return(DECREMENT);           }

"+="                        { count(); return(ADD_ASSIGN);          }
"-="                        { count(); return(SUB_ASSIGN);          }
"*="                        { count(); return(MULT_ASSIGN);         }
"/="                        { count(); return(DIV_ASSIGN);          }
"%="                        { count(); return(MOD_ASSIGN);          }

";"                         { count(); return(';');                 }
"{"                         { count(); return('{');                 }
"}"                         { count(); return('}');                 }
"["                         { count(); return('[');                 }
"]"                         { count(); return(']');                 }
","                         { count(); return(',');                 }
```

```
":"                     { count(); return(':'); }
"="                     { count(); return('='); }
"("                     { count(); return('('); }
")"                     { count(); return(')'); }
"&"                     { count(); return('&'); }
"!"                     { count(); return('!'); }
"-"                     { count(); return('-'); }
"+"                     { count(); return('+'); }
"*"                     { count(); return('*'); }
"/"                     { count(); return('/'); }
"%"                     { count(); return('%'); }
"<"                     { count(); return('<'); }
">"                     { count(); return('>'); }
"^"                     { count(); return('^'); }


%%


void count()
{
    int i;

    for (i = 0; yytext[i] != '\0'; i++)
        if (yytext[i] == '\n')
        {
            columnNumber = 1;
            lineNumber++;
        }
        else if (yytext[i] == '\t')
        {
            columnNumber += 8 - (columnNumber % 8);
        }
        else
        {
            columnNumber++;
        }

    ECHO;
}
```