

# FearnLang



Developing a Learner  
Programming Language

---

Centre Number	61425
---------------	-------

Candidate Number	7071
------------------	------

---

# Table of Contents

<b>Introduction</b>	<b>6</b>
<b>Analysis</b>	<b>7</b>
Background	7
Research	9
Stack Overflow Developer Survey 2023 – Market Research	9
Third Party	11
Key Takeaways	17
Analysis of Existing Languages	18
JavaScript	18
Kotlin	21
C	23
Stages of Compilation	26
Objectives	28
Rationale for A-Level Standard	29
Modelling	29
Prototype Lexing and Parsing Files	29
Abstract Syntax Tree (AST) Models	30
Example Parse Tree	30
Example AST Class Diagram	31
Example Object Code	32
<b>Documented Design</b>	<b>35</b>
Systems Overview Diagram	35
Potential Solutions	36
C/C++	36
Java	36
Chosen Solution	37
FearnLang Syntax	37
General Notes	37
Ease of Use	38
Lexical Analysis and Parsing	39
Example Regular Expressions	39
Example Productions	40
Constructing the AST	42
AST UML Diagrams	45
Semantic Analysis	48
Symbol Analysis and the Symbol Table	48
Type Analysis	50
Code Generation	51
The Java Virtual Machine (JVM) and ASM	52

# The Fearn Programming Language

Descriptors	52
ASM	52
Struct Generation	53
Global Generation	55
Function Generation	55
Generating Expressions	56
Generating Statements	56
Optimisations	57
Import Generation	57
Status Reporting & CLI	57
Examples	58
<b>Technical Solution</b>	<b>59</b>
Project Structure	59
Package Structure	61
Objectives Achieved	62
1.x: Define a Syntax for FearnLang	62
1.1: Must allow the user to create and modify variables	70
1.2: Allow for Basic Mathematical operations	81
1.3: Allow for text Input/Output, through the console	92
1.4: The language must support string, integer, float, and boolean types	95
1.5: Fearn must be able to cast data to primitive types	96
1.6: Must be able to create fixed-length arrays	101
1.7: Allow for the definition, and calling, of functions and procedures	110
1.8: Allow for the definition of user-defined data types	121
1.9: Must allow for the use of Boolean Logic	130
1.10: Allow importing of global Fearn variables, structs, and functions	130
2.1: Allows the user to call it from the command line	134
2.2: Read the file out, and parse it with into an Abstract Syntax Tree	138
2.3: Represent the AST as interconnected objects	138
2.4: Perform a Semantic Analysis on the AST	138
2.5: Optimise AST	139
2.6: Generate target code	142
2.7: Any errors raised must be as informative as possible	142
Technical Skills	144
Model	144
Algorithms	145
Example Scripts: Additional Skills	146
Dependencies	146
<b>Testing</b>	<b>147</b>
Simple Testing	147
Complex Testing	150
Erroneous Testing	155
Parsing Test	156

# The Fearn Programming Language

<b>Evaluation</b>	<b>157</b>
General Evaluation	157
Objective 1: Define the FearnLang Syntax	157
Objective 2: Develop the FearnLang Compiler	157
Potential Improvements	157
Independent Feedback	158
<b>Appendix A – Model Grammar</b>	<b>159</b>
Fearn_Parser_Model.y	159
Fearn_Lexer_Model.l	168
<b>Appendix B – Example Fearn Programs</b>	<b>172</b>
HelloWorld.fearn	172
StructExample.fearn	172
BubbleSort.fearn	173
MergeSort.fearn	174
BinarySearch.fearn	177
Dijkstra.fearn	179
graph.fearn	181
<b>Appendix C – Full Source Code</b>	<b>183</b>
main.java	183
parser	187
FearnGrammar.g4	187
parser.ASTConstructor	194
ast	216
ast.ASTNode	216
ast.Declaration	217
ast.Program	219
ast.Struct	220
ast.expression	221
<i>ast.expression.Expression</i>	221
<i>ast.expression.ArrayBody</i>	223
<i>ast.expression.ArrayInitExpression</i>	226
<i>ast.expression.AssignExpression</i>	230
<i>ast.expression.BinaryExpression</i>	236
<i>ast.expression.CastExpression</i>	244
<i>ast.expression.FnCallExpression</i>	248
<i>ast.expression.IncrExpression</i>	254
<i>ast.expression.IndexExpression</i>	257
<i>ast.expression.PrimaryExpression</i>	260
<i>ast.expression.StructAttrExpression</i>	264
<i>ast.expression.StructInitExpression</i>	266
<i>ast.expression.UnaryExpression</i>	268
ast.statement	271
<i>ast.statement.Statement</i>	271

# The Fearn Programming Language

<i>ast.statement.CompoundStatement</i>	273
<i>ast.statement.ExpressionStatement</i>	275
<i>ast.statement.IterationStatement</i>	277
<i>ast.statement.JumpStatement</i>	282
<i>ast.statement.ReturnStatement</i>	284
<i>ast.statement.SelectionStatement</i>	286
ast.function	289
<i>ast.function.Function</i>	289
<i>ast.function.Parameter</i>	291
ast.type	292
<i>ast.type.TypeSpecifier</i>	292
<i>ast.type.PrimitiveSpecifier</i>	293
<i>ast.type.StructInstanceSpecifier</i>	294
<i>ast.type.ArraySpecifier</i>	295
<i>ast.type.ArrayBodySpecifier</i>	296
semantics.table	297
semantics.table.SymbolTable	297
semantics.table.Row	306
semantics.table.VariableRow	306
semantics.table.FunctionRow	307
semantics.table.StructRow	308
codegen	309
codegen.CodeGenerator	309
codegen.CastOptimiser	318
codegen.CastOptimiser	321
codegen.ImportCompiler	323
util	328
util.Reporter	328
util.FearnErrorListener	330
FearnRuntime.java	331
FearnStdLib	334
FearnStdLib.io	334
FearnStdLib.maths	334
FearnStdLib.RandomNumbers	334
CMD Files	335
FearnC.cmd	335
FearnRun.cmd	335

# 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, encounter systems languages like C/C++, Kotlin, Rust, or 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).
- If necessary, 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 are 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.

<p><b>Front-End</b> (<i>Takes in the program, and processes it into an intermediate representation, raising errors when anything is invalid</i>)</p> <ul style="list-style-type: none"><li>• <b>Lexing</b> (splitting the source into token)</li><li>• <b>Parsing</b> (Composing those tokens into an Abstract Syntax Tree)</li><li>• <b>Semantic Analysis</b> (Validating the program)</li></ul>	<p><b>Back-End</b> (<i>Generates code that can be executed, and optimises it for maximum speed at runtime and minimal file size</i>)</p> <ul style="list-style-type: none"><li>• <b>Optimisation</b> (Generating an intermediate code and applying a ruleset to improve)</li><li>• <b>Code Gen</b> (writing executable code to a binary file)</li></ul>
---	---



# The Fearn Programming Language

## 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.<sup>1</sup> 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.<sup>2</sup>

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.

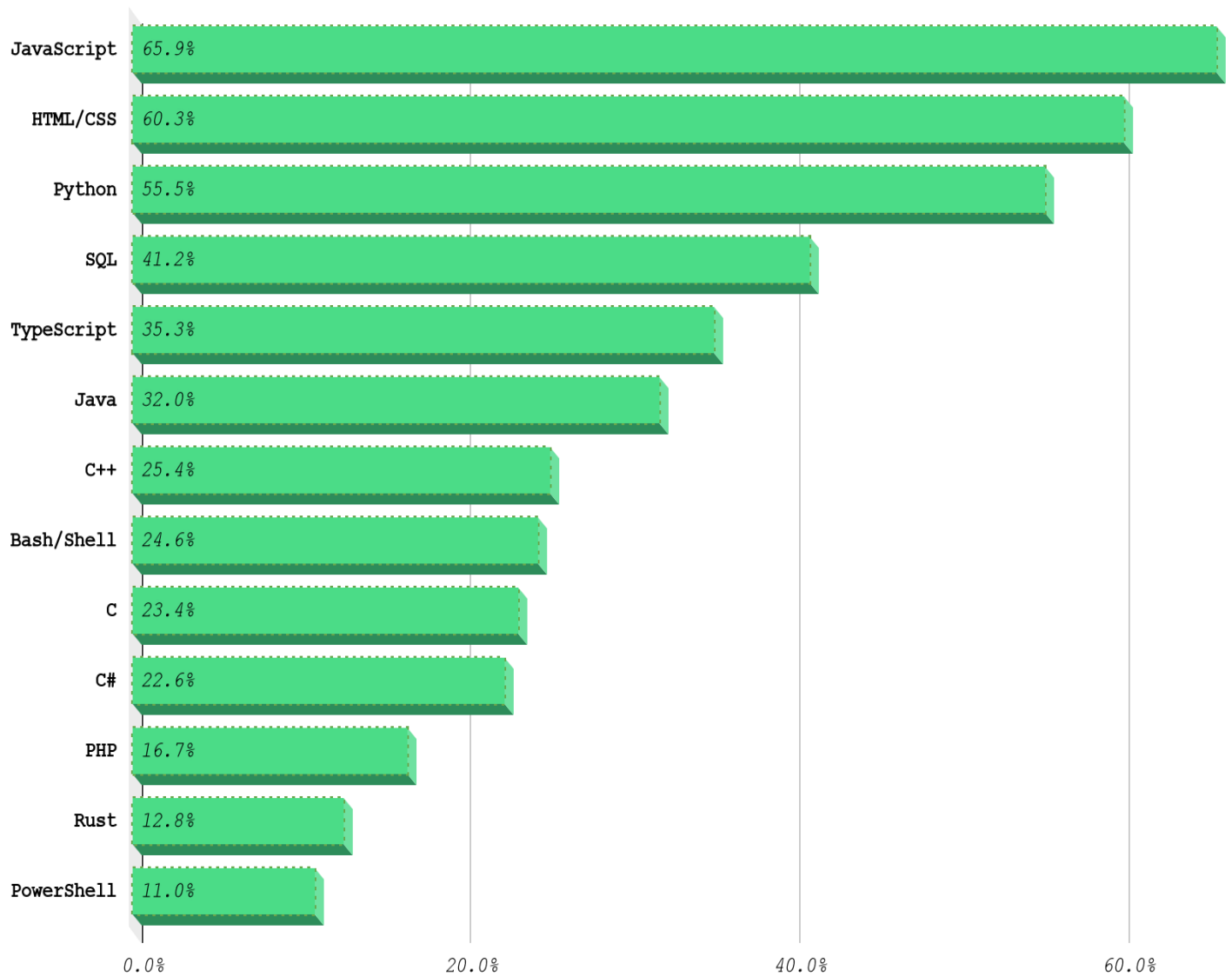
---

<sup>1</sup> Stack Exchange Inc. 2023. "Stack Overflow Developer Survey 2023." Stack Overflow. 2023.  
<https://survey.stackoverflow.co/2023/>.

<sup>2</sup> Stack Overflow Beginner Language Analysis:  
<https://colab.research.google.com/drive/1kKt66qAhXb2yfIiDZh8t2HIvTRwBBSkj?usp=sharing>

# The Fearn Programming Language

*Figure 1: Languages Favoured by Beginners*



# The Fearn Programming Language

## 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<sup>3</sup>. 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.

---

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

## The Fearn Programming Language

### **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
- Redefining 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-typed 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
  - 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.

## The Fearn Programming Language

### 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)

## The Fearn Programming Language

### 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 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.<sup>4</sup> 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

---

<sup>4</sup> 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>

## The Fearn Programming Language

### 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.

### 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 be 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<sup>5</sup>
  - 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)

---

<sup>5</sup> 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>

## The Fearn Programming Language

### 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, 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)



# The Fearn Programming Language

## 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.
- 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, strict, and explicit type checking, both for variables and functions (return types, parameters and arguments, etc).
  - Not supporting OOP
- Make sure the language supports the developer well, through Garbage Collection and Informative Error Raising
  - Garbage Collection is the concept of releasing memory allocated to a program back to the computer after the program has finished executing
  - Error Raising must be as informative as possible, both at compile time (when the Fearn Compiler is called) and runtime (when the compiled program is run)

# The Fearn Programming Language

## 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.<sup>6</sup>

#### Features

- Multi-paradigm language, supporting imperative, object-oriented, functional features (such as first class functions, taking other functions as arguments).

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

# The Fearn Programming Language

- 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.<sup>7</sup>
- 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 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.<sup>8</sup>

## 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

```
> var testArray=null;
  if(testArray.length===0){
    console.log("Array is empty");
  }
  >
  Error
    line: 3
    message: "'null' is not an object (evaluating 'testArray.length')"
    sourceId: 2056192896
    __proto__: Error
  >
```

- Varies by runtime/browser
- Refers to the line number, type of error, and what was being evaluated

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

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

## The Fearn Programming Language

- 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
- 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
  - Alternatively, I could use libraries or tools that automate this for me at runtime

# The Fearn Programming Language

## 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)

# The Fearn Programming Language

## 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<sup>9</sup>

## 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<sup>10</sup>

## 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

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

<sup>10</sup> 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>.

# The Fearn Programming Language

## C

Sample Program

```
#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;
            }
        }
    }
}
// Start Point
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

# The Fearn Programming Language

## 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



## The Fearn Programming Language

### 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

# The Fearn Programming Language

## Stages of Compilation

In preparation for this project, I performed research into the theory of compilation and language design, as well as the practical tools used in implementing a compiler. The following breakdown expands on my prior work, completing the online Compilers course from Stanford University<sup>11</sup>, developing a standard compiler structure to fit my specific needs.

### *I. Lexical Analysis / Lexing*

The first stage of compilation is lexical analysis, performed by a module / program called a lexical analyser, lexer, or tokenizer. The problem this solves is to break down the text-based words of the language (lexemes), and convert them into meaningful lexical tokens that the program can then understand in context, and in relation to one another.

### *II. Parsing*

A parser receives the lexical tokens, and converts them into an Abstract Syntax Tree (AST). This is based on a series of productions, often written in BNF or extended BNF, a formal language which specifies patterns which match a particular production, and that can include tokens and other productions.

### *III. Semantic Analysis*

This is a module dedicated to enforcing the rules of the programming language. Some of its common tasks include:

- Type Analysis of expressions to ensure the right data types are used in context (for example, an integer can't be passed to a function that only accepts a boolean value)
- Scope Checking, using a symbol table to ensure variables have been declared before their used

---

<sup>11</sup> Aiken, Alex. 2020. "Compilers." EdX. March 17, 2020.  
<https://www.edx.org/learn/computer-science/stanford-university-compilers>.

# The Fearn Programming Language

## IV. Optimisation

This stage performs iterative improvements on the program, to reduce its complexity and increase its efficiency on the processor.

- *Constant Propagation*: Variables which have a constant value can be propagated, so each instance of it is replaced by its constant value. The challenge with this is ensuring that there is no way that variable is ever changed dynamically
- *Constant Folding*: Evaluating mathematical expressions that have a constant value (doesn't change at run time)
- *Dead-Code Elimination*: Removing any instructions that could never be reached, or that are redundant (e.g. defining variables that are never accessed).

When optimising a complex program with control structures (if statements, loops, etc), sophisticated compilers construct a Control Flow Graph, which represents different blocks of code and how control can change between them. These optimisations can then be applied across the graph.

## V. Code Generation

Finally, object code is generated, which represents a program's behaviour in short, simple instructions that can be understood, either directly by the processor or by a virtual machine, which can then run the program.

# The Fearn Programming Language

## Objectives

### 1. Define a Syntax for FearnLang

- 1.1. Must allow the user to create variables, of an explicit data type, and to modify their values
- 1.2. Allow for Mathematical operations, such as Addition, Subtraction, Multiplication, Division, and Modulo
- 1.3. Allow for text Input/Output, through the console
- 1.4. The language must support string, integer, float, and boolean data types
- 1.5. Fearn must be able to cast data to primitive types
- 1.6. Must be able to create fixed-length arrays
- 1.7. Allow for the definition, and calling, of functions and procedures.
- 1.8. Allow for the definition of user-defined data types (e.g. C-style structs)
- 1.9. Must allow for the use of Boolean Logic
- 1.10. Must allow for the importing of global variables, structs, and functions from other Fearn programs

### 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
  - I will use some external tool to generate a lexer and parser for my language, specifying the grammar with regular expression and BNF
- 2.3. Represent the AST as interconnected objects, using composition to relate each node to those it contains
  - An example of this would be a binary operation, which would be a node that contained the operator, as well as bath operands, which are themselves child nodes of the operation
- 2.4. Perform a Semantic Analysis on the AST
  - This will involve enforcing a strong type system, verifying 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. *(Optional)* Optimise AST
  - This will work iteratively to improve the program until no further changes are made
  - Examples of this would be constant folding, constant propagation, or dead code elimination
  - I've made this objective optional as it doesn't affect the core functionality of Fearn, but would improve its performance.

## The Fearn Programming Language

- 2.6. Generate target code, that can be then executed, either directly on the processor (by generating assembly) or using a virtual machine like the JVM
- 2.7. Any errors raised must be as informative as possible, using contextual information - such variable and function names

## 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; the techniques used (compiler design and construction, parsing and lexical analysis) are mostly left to university Computer Science
- Semantic Analysis will require me to implement Depth-First Tree Traversal to verify data types, scopes, and the definitions of identifiers. It is also a substantial example of defensive design.
- Code Generation is also a recursive process, which requires me to traverse the Abstract Syntax Tree, and have a good grasp of assembly-like language.

## Modelling

### Prototype Lexing and Parsing Files

For my project, I plan to use lexer/parser generators. One option was using GNU Bison, a parser generator, and Flex, a lexer generator - should I decide to work in C++.<sup>12</sup> 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.<sup>13,14</sup> These prototype files can be found in [Appendix A](#).

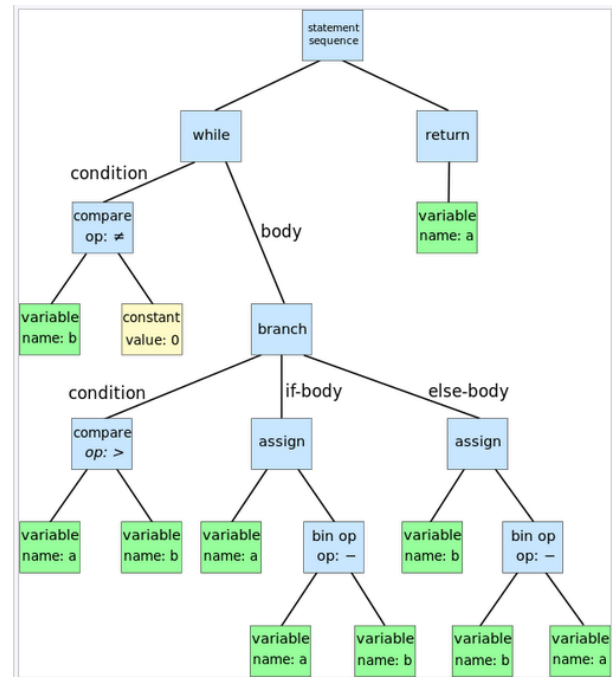
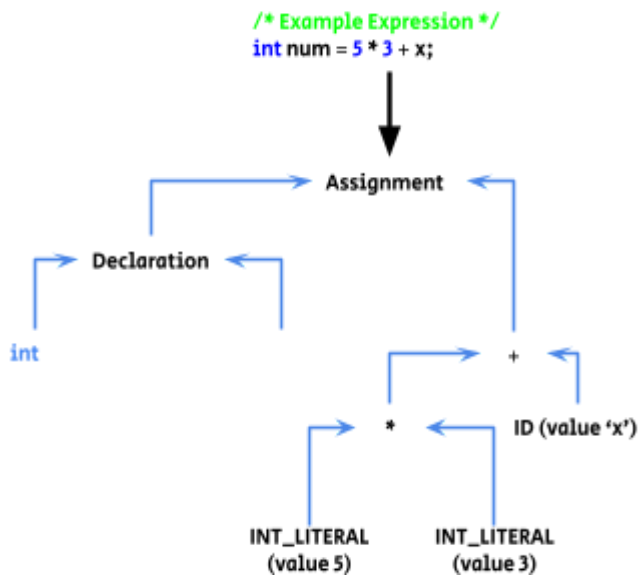
---

<sup>12</sup> These applications are sometimes referred to as 'compiler compilers'

<sup>13</sup> Lee, Jeff. 1985. "ANSI C Grammar (Lex)." [www.lysator.liu.se. 1995.](http://www.lysator.liu.se/1995)  
<https://www.lysator.liu.se/c/ANSI-C-grammar-l.html>.

<sup>14</sup> Lee, Jeff. 1985. "ANSI C Grammar (Yacc)." [www.lysator.liu.se. 1995.](http://www.lysator.liu.se/1995)  
<https://www.lysator.liu.se/c/ANSI-C-grammar-y.html>.

## Abstract Syntax Tree (AST) Models

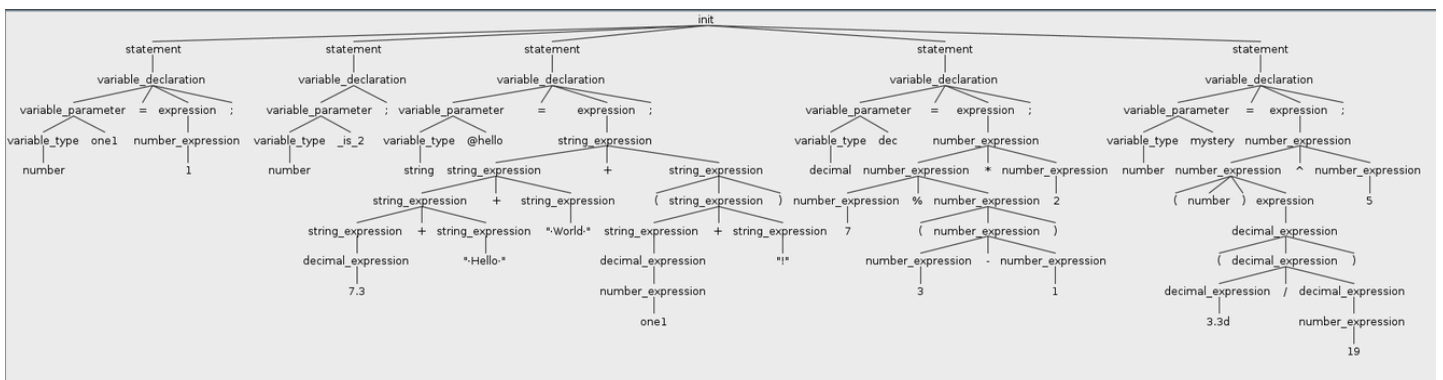


An abstract syntax tree for the following code for the Euclidean algorithm:

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

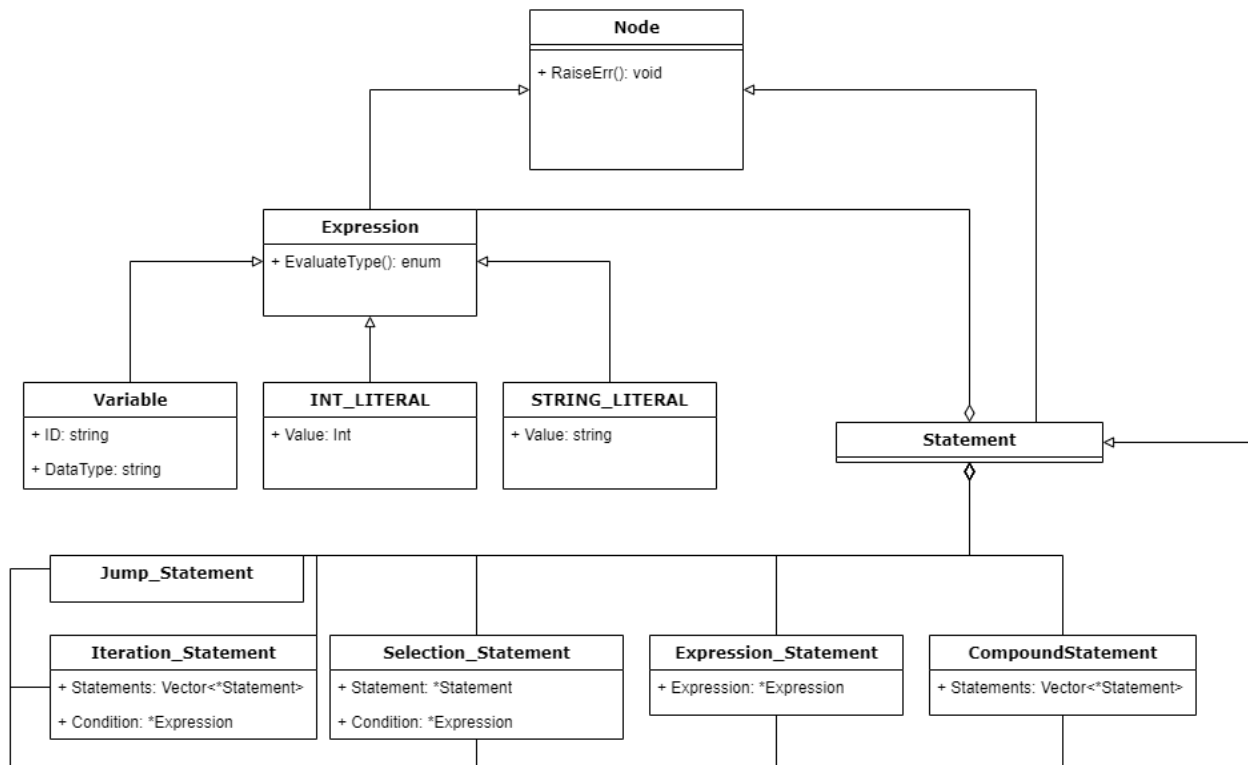
## Example Parse Tree

Below is an example of a Parse Tree produced by ANTLR, another lexer/parser generator I considered using for the project. This shows not just how each node links to the others, but also displays syntactical symbols used in the language grammar, such as semi-colons or brackets, that aren't helpful when generating code. Thus, these parse trees would need to be simplified into an Abstract Syntax Tree, by traversing it and extracting only necessary information.



# The Fearn Programming Language

## Example AST Class Diagram<sup>15</sup>



<sup>15</sup> This is only an example of classes that will be present, their fields, and inheritance.

# The Fearn Programming Language

## Example Object Code

BinarySearch.java

```
public class BinarySearch {

    public static int Search(int[] arr, int key, int low, int high) {
        int index = -1;

        while (low <= high) {
            int mid = low + ((high - low) / 2);

            if (arr[mid] < key) {
                return Search(arr, key, mid + 1, high);
            } else if (arr[mid] > key) {
                return Search(arr, key, low, mid - 1);
            } else if (arr[mid] == key) {
                index = mid;
                break;
            }
        }

        return index;
    }

    public static void main(String[] var0) {
        System.out.println(Search(
            new int[] {0, 4, 7, 16, 54, 129},
            54,
            0,
            5
        ));
    }
}
```



# The Fearn Programming Language

BinarySearch.class (Equivalent Bytecode)

```
public class BinarySearch {
    public BinarySearch();
        Code:
        0: aload_0
        1: invokespecial #1          // Method java/lang/Object."<init>":()V16
        4: return

    public static int Search(int[], int, int, int);
        Code:
        0: iconst_m1
        1: istore      4
        3: iload_2
        4: iload_3
        5: if_icmpgt 7317
        8: iload_2
        9: iload_3
        10: iload_2
        11: isub
        12: iconst_2
        13: idiv
        14: iadd
        15: istore      5
        17: aload_018
        18: iload      5
        20: iaload
        21: iload_1
        22: if_icmpge 36
        25: aload_0
        26: iload_1
        27: iload      5
        29: iconst_1
        30: iadd
        31: iload_3
        32: invokestatic #7          // Method Search:([IIII)I19
        35: ireturn
        36: aload_0
        37: iload      5
        39: iaload
        40: iload_1
        41: if_icmple 55
        44: aload_0
        45: iload_1
        46: iload_2
        47: iload      5
        49: iconst_1
```

---

<sup>16</sup> This is the generic Java Object Constructor. Every object in a Java program must have a constructor, which will first call a generic constructor for all objects. If no constructor is defined, this generic constructor is all that is called.

<sup>17</sup> This shows the program jumping on a condition. In the JVM, this jump will pop the last two values from the stack. If value1 > value2, it will jump to line 73.

<sup>18</sup> ALOAD differs for ILOAD, as ALOAD loads an object from local variables section of memory, whereas ILOAD loads a value of type int.

<sup>19</sup> This demonstrates that methods and variables in the JVM have a signature. In this case, the method takes a 1-D int array ([I], 3 int values (I), and returns an int.

## The Fearn Programming Language

```
50: isub
51: invokestatic #7          // Method Search:([IIII)I
54: ireturn
55: aload_0
56: iload          5
58: iaload
59: iload_1
60: if_icmpne      70
63: iload          5
65: istore         4
67: goto           73
70: goto           3
73: iload          4
75: ireturn
```

```
public static void main(java.lang.String[]);
```

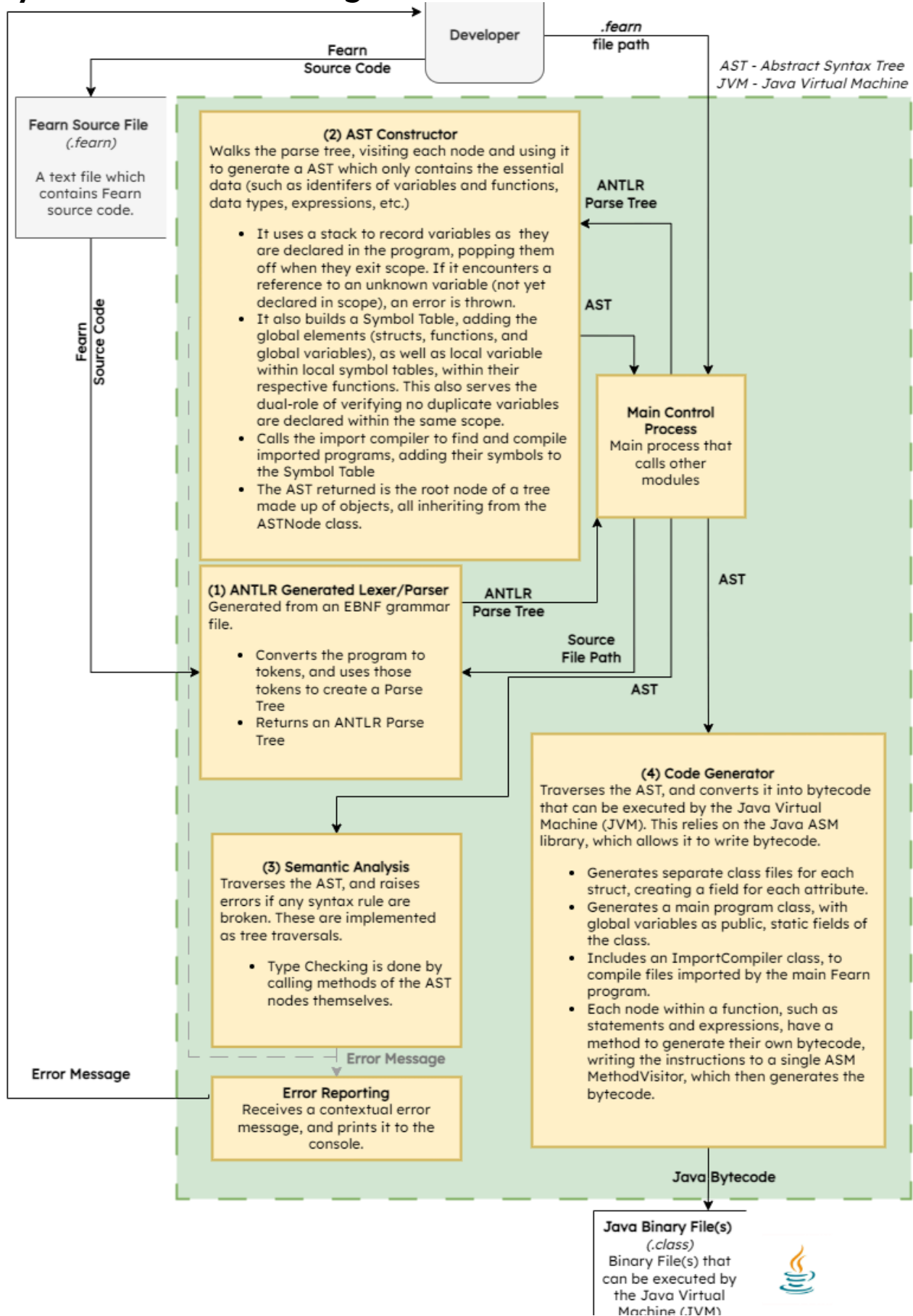
```
Code:
```

```
0: getstatic #13          // Field java/lang/System.out:Ljava/io/PrintStream;
3: bipush      6
5: newarray      int
7: dup
8: iconst_0
9: iconst_0
10: iastore
11: dup
12: iconst_1
13: iconst_4
14: iastore
15: dup
16: iconst_2
17: bipush      7
19: iastore
20: dup
21: iconst_3
22: bipush      16
24: iastore
25: dup
26: iconst_4
27: bipush      54
29: iastore
30: dup
31: iconst_5
32: sipush      129
35: iastore
36: bipush      54
38: iconst_0
39: iconst_5
40: invokestatic #7          // Method Search:([IIII)I
43: invokevirtual #19        // Method java/io/PrintStream.println:(I)V
46: return
```

```
}
```

# Documented Design

## Systems Overview Diagram



# The Fearn Programming Language

## Potential Solutions

I had multiple options for which technologies to use, however two were the most rational.

### C/C++

- The legacy option for constructing compilers and other performance and efficiency-reliant software, such as Operating Systems and Virtual Machines, as it allows for low-level control
- The Visual C/C++ Compiler (built into Visual Studio) compiles to native assembly. Thus, there is no garbage collection that would slow down the program.
- It allows me to aggregate objects using memory pointers
- I can also use BNF and RegEx in my lexer/Parser, as I can use Flex and GNU Bison, two generators that allow me to tokenize programs and parse said tokens into a parse tree.
- On the other hand, using C++ would require a number of functions to be defined to release memory to prevent a memory leak
- It's lower abstraction syntax would also make it harder to manage large complex systems, such as a compiler
- Moreover, bringing libraries into my project is far harder in C++, as there's no standard way to group or import external projects

### Java

- Object-oriented systems language
- Compiled .jar files are more portable across Operating Systems thanks to the JVM, as opposed to processor specific binaries produced using C++
- This language allows me to easily import libraries in .jar files.
- ANTLR4 is a combination lexer/parser generator that generates Java code. It uses extended BNF / EBNF, which allows me to use RegEx-style meta-characters, and allows me to not have to define as many tokens as I can define RegEx patterns within productions.
- ANTLR4 can also generate walkers, Classes that allow me to control a traversal of the tree, and which I can override with program-specific code to execute when it encounters each node.
- ANTLR4 is also far easier to test and debug than Flex or GNU Bison, as it can produce graphical parse trees, given a grammar and text input
- Java is an easier language to work in, and promotes natural usage of Object-Oriented Programming, which I plan to use to model my AST
- There are also multiple libraries for code generation in Java, such as ASM and Cglib, which allow developers to procedurally generate bytecode instructions

## The Fearn Programming Language

to run on the Java Virtual Machine, a cross-platform Stack Machine which executes bytecode produced by the Java compiler.

- This technique of targeting the JVM is the same used by languages like Kotlin and Scala, which also have compilers built using Java
- It allows me to take advantage of the features built-into Java, including Objects, arithmetic and boolean operations, I/O, and Java's garbage collector (which releases memory used for variables once they exit scope)
- The main downside is that Java isn't as efficient as C/C++, as it uses a garbage collector and is interpreted at runtime, although this is done as bytecode running on the JVM.

## Chosen Solution

Since I need to work to a strict time limit to complete my NEA, I decided to select Java, ANTLR, and ASM as the technologies to build the Fearn Compiler. This is because they're easier to work with and test as opposed to C/C++.

## FearnLang Syntax

My third party's advice emphasised the importance of producing clean and simple code, without unnecessary bloating. This has led me to design Fearn's syntax to be as easy to read as possible, while still enforcing strict type rules. Example scripts, demonstrating Fearn's syntax, can be found in [Appendix B](#).

## General Notes

- A Fearn Program is composed of four elements
  - 1) Imports:** Added at the top of the file. These can import standard library modules (e.g `io` for the `print()` and `input()` functions), or other fearn programs - imported using the path to the program from the importing script.
  - 2) Struct Definitions:** Each struct has an identifier (used to instantiate it), and a body containing declarations, that are the struct's attributes.
  - 3) Global Variables:** Variables declared outside functions, that are accessible at any point in the program, or in programs that import the script.
  - 4) Function:** Standard C-style functions, declaration with an identifier, typed arguments, a return type, and a compound statement body.
- The language is designed so that developers can write easy to read and maintain code, and do so quickly. This includes C-style comments, and disregarding whitespace.

## The Fearn Programming Language

- After consideration, I decided against his notion of enforcing python-style tabulation. This is because:
  - Implementing the emitting of indent and dedent tokens was overly-complicated
  - A vast majority of systems languages are 'curly braces' languages, where compound statements are surrounded by curly braces. This is the syntax a learner language should encourage.
- The global scope can contain declarations, initialising global variables, strict definitions, and function definitions
- Structs in Fearn allow the developer to create their own data structures, containing primitive or array fields
- Functions are defined using the keyword 'fn' followed by an identifier and typed arguments. One function must be the main function, which should have the signature '`fn main(args : str[]) => void`'.
  - This notation is easy to read and write, and is a lightweight style, similar to the Go and Rust languages
- Fearn features a standard library, which allows the user to import basic functions into their project. An example module is io, which contains the print and input functions.

## Ease of Use

- Strings in Fearn allow for single or double quotes, to allow flexibility
- Fearn supports **Universal Function Notation**, also known as *Universal Function Call Syntax*, which allows functions to be called in a similar style to methods in an Object-Oriented language

```
// The following are all correct
a = myFunction(x, y);
b = x.myFunction(y);
c = x.myFunction(y).myFunction(z);
```

- The syntactic similarity of the language syntax to other C-like languages is meant to get learners more comfortable with this highly common approach
- The notation is also designed to be lightweight and clear.

# The Fearn Programming Language

## Lexical Analysis and Parsing

I've used ANTLRv4 for this stage. The reason for using a parser generator is to speed up development and make it easy to test and implement changes to the Fearn grammar.<sup>20</sup>

### Example Regular Expressions

Regular Expressions are used by ANTLR's lexer to match each lexeme to a token, and then feed these tokens to the parser.

Expression	Matches
<code>D [0-9]</code>	<i>A digit (used in later patterns)</i>
<code>L [a-zA-Z_]</code>	<i>A letter</i>
<code>{L}({L} {D})*</code>	<i>An Identifier (for variables, functions, and structs)</i>
<code>{D}+</code>	<i>Integer Literal</i>
<code>{D}+\.{D}+</code>	<i>Float Literal</i>
<code>".*"   '.*'</code>	<i>String Literal</i>
<code>'true'   'false'</code>	<i>Boolean Literal</i>
<code>\\/\\.*</code>	<i>A single-line C-style comment</i>
<code>\\/\\*(. \\n)*\\*\\/</code>	<i>A multi-line C-style comment</i>
<code>while return == &amp;&amp; += etc.</code>	<i>Exact patterns like these exist for every keyword and operator in FearnLang.</i>

---

<sup>20</sup> To learn about ANTLR's full functionality, please see The Definitive ANTLR 4 Reference (<https://dl.icdst.org/pdfs/files3/a91ace57a8c4c8cdd9f1663e1051bf93.pdf>)

# The Fearn Programming Language

## Example Productions

Productions are used to define non-terminal symbols in ANTLR, to build a parse tree. These are written in extended BNF, a superset of standard Backus-Naur that allows for additional metacharacters, like those used in regular expressions, to make grammars more straightforward by reducing the need for superfluous production rules and recursive productions.

<pre>type_name: 'int'   'float'   'bool'   'str';</pre>
Matches the literal names of primitive types, when the developer specifies a data type.
<pre>type_spec_struct: '\$' IDENTIFIER;</pre>
Matches the use of struct names as a type, e.g. '\$MyStruct x = new MyStruct()'
<pre>type_spec_arr: (type_spec_primitive   type_spec_struct) ('[]')+ ;</pre>
Matches the type specifier for a list, e.g. 'int[]'
<pre>program: (function   declaration   struct_def)+ ;</pre>
This defines the root type for a program, which is made up of global variables, struct definitions, and functions.
<pre>function: 'fn' IDENTIFIER '(' (( parameter ',')* parameter )? ')' '='&gt;' ( type_specifier   'void' ) compound_statement;</pre>
This matches a function, defined as the keyword fn, followed by its name, parameters, and a return type. The compound statement represents the function body.
<pre>parameter: IDENTIFIER ':' type_specifier ;</pre>
Matches a parameter, which has an identifier and a data type.
<pre>struct_def: 'struct' IDENTIFIER '{' declaration* '}' ;</pre>
Matches a struct definition, which is named and contains declarations of its attributes. <sup>21</sup>

---

<sup>21</sup> **FearnLang's structs don't allow methods.** However, the language supports **universal function notation**, meaning a function `doSomething(x: $MyStruct) => void` could be called as `x.doSomething` or `doSomething(x)`



# The Fearn Programming Language

```
compound_statement: '{' (declaration | statement)* '}' ;
```

Matches a compound statement, used as the bodies of functions and other statements.

```
selection_statement  
    : 'if' '(' expression ')' compound_statement  
    | 'if' '(' expression ')' compound_statement 'else' compound_statement  
    | 'if' '(' expression ')' compound_statement 'else' selection_statement  
    ;
```

Matches an if statement.

```
iteration_statement:  
'for' '(' init_expression? ';' continue_condition ';' iteration_expression? ')'   
compound_statement ;
```

Matches a loop. While loops aren't supported, however the syntax for iterations is loose enough that it can be used for either purpose.

```
jump_statement: 'continue' ';' | 'break' ';' | 'return' expression? ';' ;
```

Matches a jump statement.

```
declaration: 'let' IDENTIFIER ':' type_specifier ( '=' expression )? ';' ;
```

Matches declarations of variables.

```
array_init  
    : 'new' (type_specifier_primitive | type_specifier_struct)  
    ('[' expression ']') +  
    | 'new' ( type_specifier_primitive | type_specifier_struct ) ('[]') +  
    array_body;
```

Matches the initialisation of an array, which are declared in a similar way to arrays in Java, where either dimensions or a body can be provided.

```
array_body  
    : '{' (array_body ',') * array_body '}' | '{' (expression ',') * expression '}'  
    | '{' '}' ;
```

Matches an array body, e.g. {1, 2, 3} , {"this", "is"}, {"2", "dimensional"}}

```
struct_init : 'new' IDENTIFIER '(' ( ( expression ',') * expression )? ')';
```

Matches the instantiation of a struct, accepting arguments for each attribute.

# The Fearn Programming Language

## Constructing the AST

The AST Constructor is an object that inherits from a grammar-specific tree walker generated by ANTLR. It uses the visit method recursively, to perform a Depth-First Traversal of the parse tree ANTLR has generated, and return AST nodes that compose to represent the program in an abstract way that is far easier to process.

Initially, the imports at the top of the file are visited, and said programs are compiled, their binaries written to the build directory (where all compiled files are written), and the Symbol Tables produced during their compilation added to the current Symbol Table, to ensure all the elements of imported programs are present. If the program imports modules from Fearn's standard library, the signatures of the functions/structs included in those modules are added to the SymbolTable. Imported rows have an 'owner' property, which is used during Code Generation to ensure the compiled programs access elements from the correct compiled class.

The below pseudocode demonstrates examples of some of the methods that are called during the traversal of the parse tree. Each method follows a similar procedure, differing slightly for the needs of that type of node, and of the greater program.

1. Visit the children of the node recursively, receiving the AST Node objects they return.
2. Perform Semantic Analysis tasks if necessary
  - a. If the node is a declaration, add the variable to the symbol stack and the Symbol Table
    - i. There are multiple symbol tables, stored in a stack, as a function adds a new one for its local variables to the top of the stack. Once the sub-tree is traversed, that is popped off and stored in a FunctionRow, along with other data about the function, in the Global Symbol Table (always at the bottom of the stack). This means that declarations, for example, always add their variables to the Symbol Table for the current scope (be that a function scope or the global scope)
  - b. If the node is a variable reference (a type of primary expression with no children), throw an exception if the variable doesn't appear in the symbol stack
  - c. If the node is a compound statement, remove any new symbols in the symbolstack that were added by nested statements, as those symbols will now be out of scope and inaccessible.
3. Create a new AST Node object to represent this node, and return it to its parent

Below are some example pseudocode methods that construct the Abstract Syntax Tree.

## The Fearn Programming Language

```
# Example for visiting a variable reference
visitVar_ref_expr(ctx)
    id = ctx.getText()vbghmnj

    if symAnalysisStack does not contain id
        Reporter.ReportErrorAndExit(
            "Variable Identifier Unknown in Scope: " + id
        )

    return new PrimaryExpression<String>(
        id,
        ExprType.VariableReference
    )

# Example for visiting a literal value
visitLiteral(ctx)
    switch ctx.getStart().getType()
    case FearnGrammarLexer.INT_LIT:
        return new PrimaryExpression<Integer>(
            Integer.valueOf(ctx.getText()),
            ExprType.IntLiteral
        )
    case FearnGrammarLexer.BOOL_LIT:
        return new PrimaryExpression<Boolean>(
            Boolean.valueOf(ctx.getText()),
            ExprType.BoolLiteral
        )
    case FearnGrammarLexer.FLOAT_LIT:
        # FearnLang Floats are represented by Java doubles
        # for additional precision
        return new PrimaryExpression<Double>(
            Double.valueOf(ctx.getText()),
            ExprType.FloatLiteral
        )
    case FearnGrammarLexer.STR_LIT:
        return new PrimaryExpression<String>(
            ctx.getText().substring(1, ctx.getText().length() - 1),
            ExprType.StrLiteral
        )
    default:
        Reporter.ReportErrorAndExit(
            "Parse Error: Unable to Parse literal value"
        )

# Example for visiting a Declaration
visitDeclaration(ctx)
    identifier = ctx.IDENTIFIER().getText()
    # Add to symbol stack
    symAnalysisStack.push(identifier)
    # Traverse TypeSpecifier to get TypeSpecifier object
    type_spec = (TypeSpecifier)visit(ctx.type_specifier)
    init_expression = null
    # Get initialisation expression, if present
    if ctx.getChildCount() > 5
        init_expression = (Expression)visit(ctx.expression)
```

## The Fearn Programming Language

```
# Add variable to the current SymbolTable (could be for a function or the
global scope)
symTabStack.peek().addSymbol(
    identifier,
    new VariableRow(
        identifier,
        type_spec
    )
)
# Return Declaration object (derived class of AST Node, see UML Diagram
below)
return new Declaration(identifier, type_spec, init_expression)

# Example for visiting a Compound Statement
visitCompound_statement(ctx)
# Get initial number of variable symbols
numOfSyms = size of symAnalysisStack
# Create arrays for local declarations and statements
local_decls = new ArrayList<Declaration>
local_stmts = new ArrayList<Statement>
# Visit Declarations and Statements, adding the returned objects
for i = 0 to size of ctx.declaration() - 1
    local_decls.add(visit(ctx.declaration(i)))

for i = 0 to size of ctx.statement() - 1
    local_stmts.add(visit(ctx.statement(i)))
# Pop the newly created variable symbol off the stack (out of scope)
for i = 0 to size of symAnalysisStack - numOfSyms - 1
    symAnalysisStack.pop()

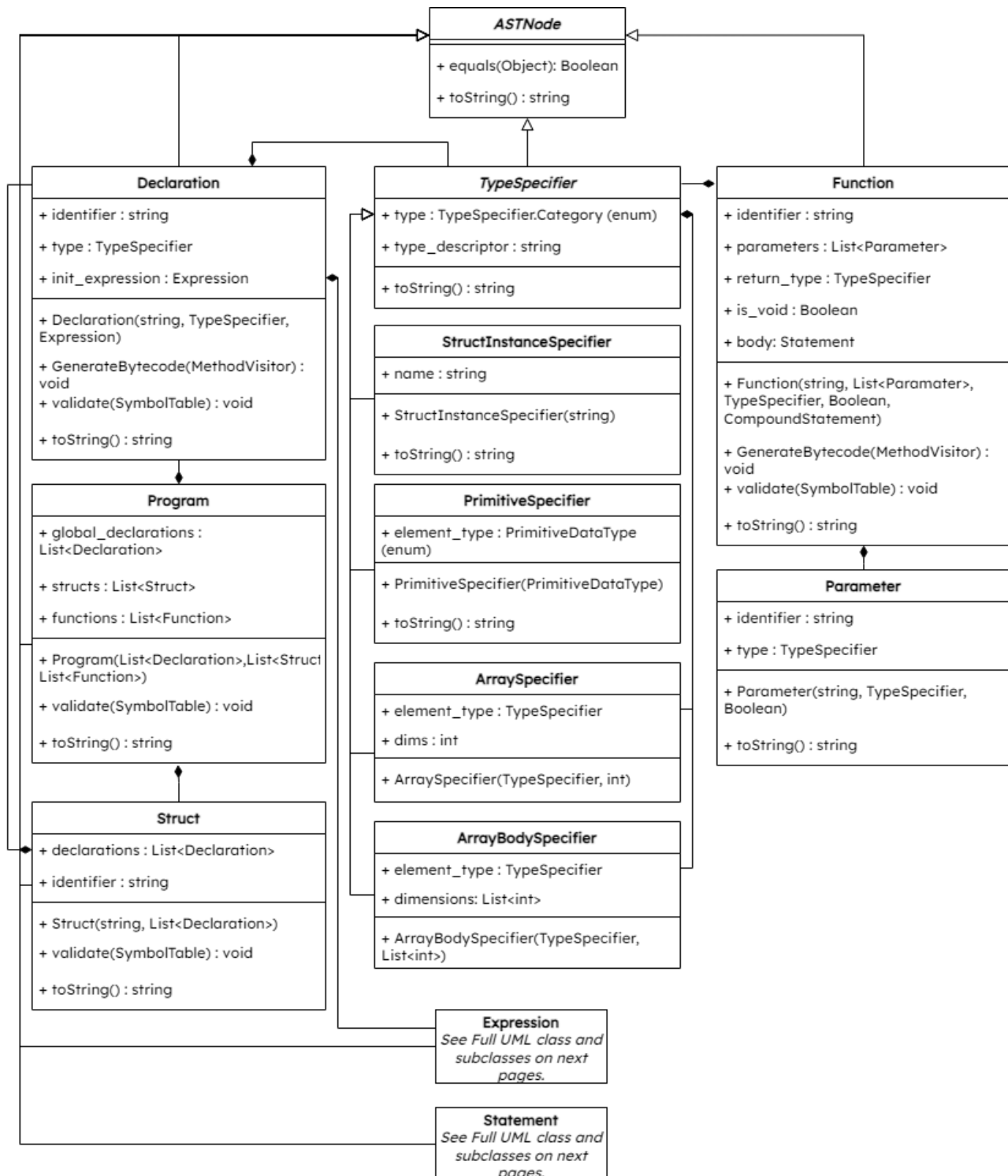
return new CompoundStatement(local_decls, local_stmts)
```

The AST Converter class also performs a few initial semantic checks that can't easily be done by the AST objects themselves. This includes symbol analysis. This is done by adding new symbols to a stack as they are declared, and popping them off once they exit scope. This ensures no variable is used outside the scope it's declared in, or before it has been declared.

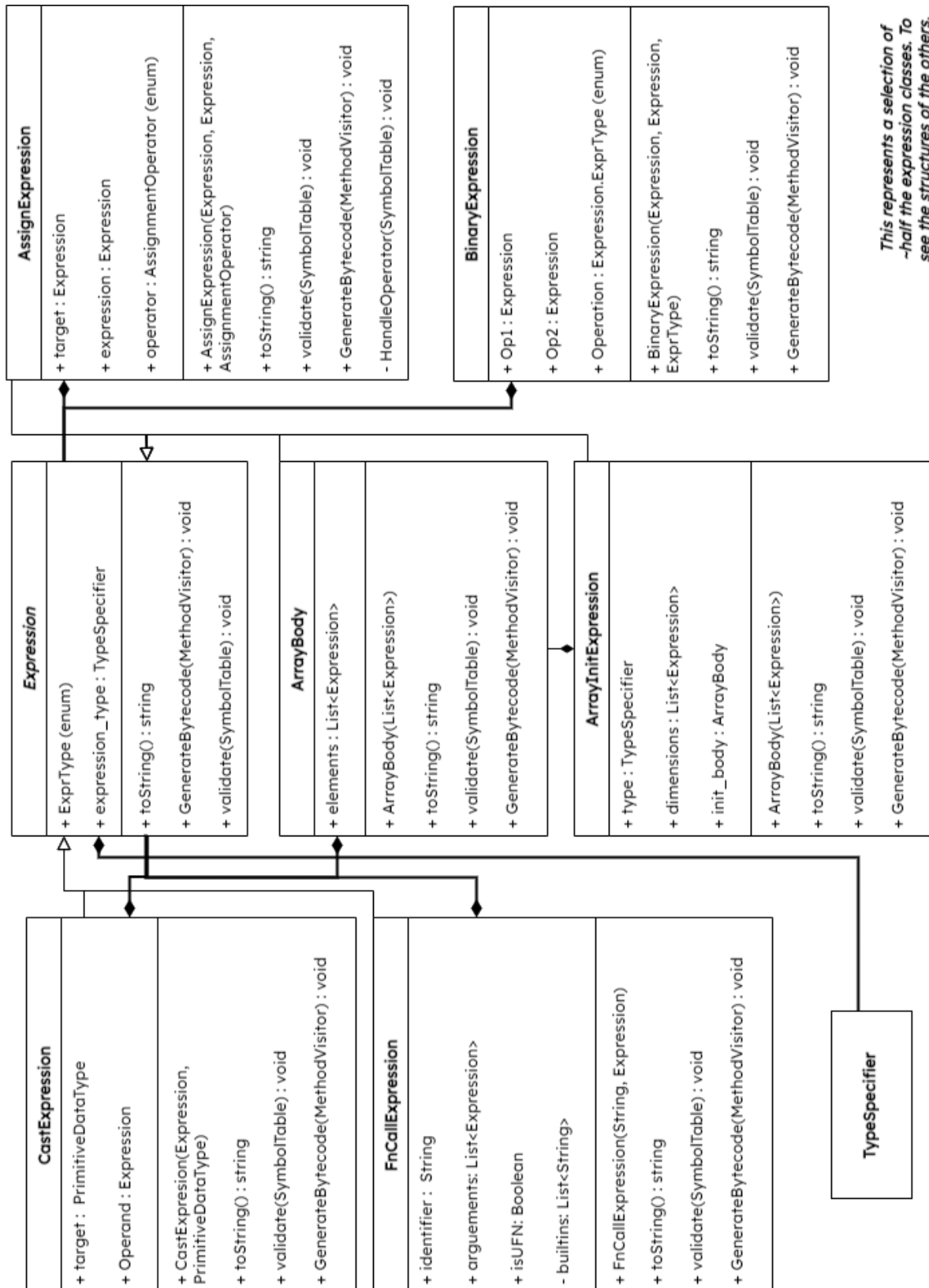
The AST Converter also constructs a Symbol Table. This involves adding new rows to an array for each variable declared. Structs and Functions have local symbol tables that represent their local symbols, including function arguments. This both ensures no variable identifiers in the same function or struct have the same identifier, and provides important data on the number of variables, and the data types associated with symbols, when specific Semantic Analysis and Code Generation is performed later.

# The Fearn Programming Language

## AST UML Diagrams

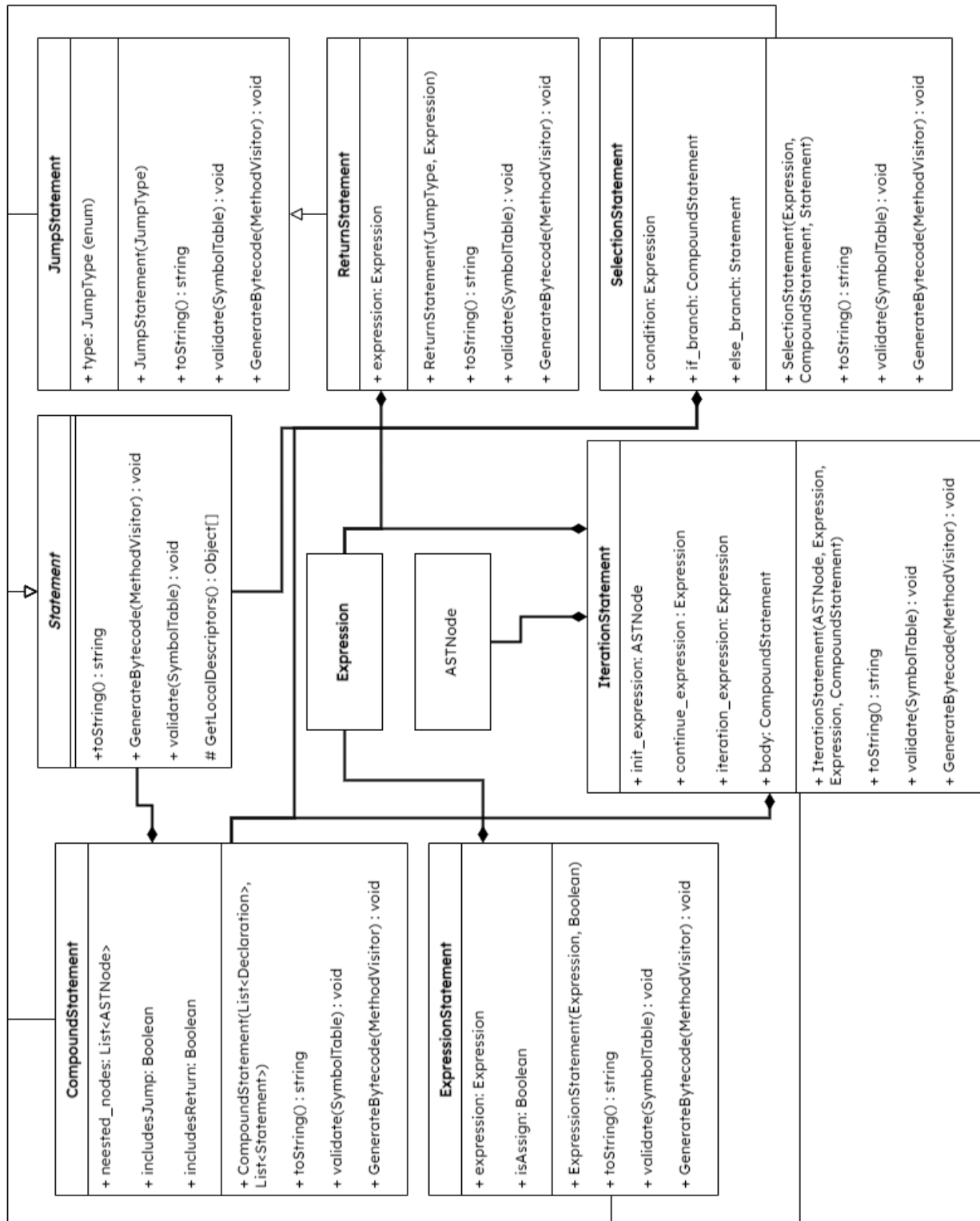


# The Fearn Programming Language



This represents a selection of  
-half the expression classes. To  
see the structures of the others,  
please see the full source code in  
Appendix 1.

# The Fearn Programming Language



# The Fearn Programming Language

## Semantic Analysis

The Semantics classes perform two major checks:

- **Symbol Analysis:** While some of this is done during the construction of the Abstract Syntax Tree (specifically, validating variables are all declared before use), remaining issues (such as ensuring that functions and structs referenced in the program have actually been defined), are performed at this stage, by the Symbol Table.
- **Type Analysis:** Performing a post-order, depth-first traversal of the abstract syntax tree, using a common method defined for each node in the AST, to validate the types of its subtree, before setting its own type of returning it if necessary.

## Symbol Analysis and the Symbol Table

The compiler builds a Symbol Table during the construction of the AST, modelled using composition of objects. It contains a row for every symbol in the program.

There are three types of row:

- **VariableRow:** Stores the identifier and type specifier of variables (within function scope or the global scope). It also stores the type descriptor of the variable, a text-based expression of the data type used during code generation.
- **FunctionRow:** Stores the identifier, return type specifier, and parameters of each function. It also stores its type descriptor. Additionally, it stores a local symbol table of variables declared within the scope.
- **StructRow:** Stores the identifier and its attributes in a local symbol table.

The Symbol Table is used during the validation traversal to retrieve type specifiers of variables (as well as argument and return types of functions), as part of expression validation. For example, in the declaration `let y : int = MyFunc(x, 5)`, the Symbol Table would be used to ...

- Validate that `MyFunc` is defined in the program
- Validate `x` is of the right type to be the first argument of `MyFunc` (an error would be raised at this stage if `x` was the wrong type, or `MyFunc` wasn't defined)
- Validate the second argument of `MyFunc` is an integer
- Validate that `MyFunc` returns an integer value

The Symbol Table methods also perform validation tasks on the program. For example, an error for a function not being defined is raised if the main semantic analysis process (*the Post-Order Depth-First Traversal detailed in the next section*) tries to access the functions signature or return type, to ensure type validity (*e.g. asserting the arguments to a function or of the right type*).

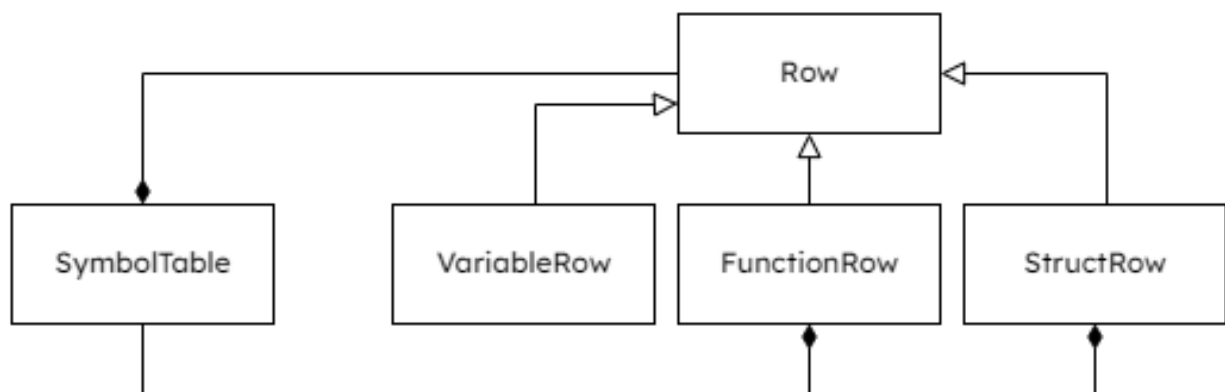


## The Fearn Programming Language

Additionally, the Symbol Table performs some Code Generation tasks. As rows are created, they generate and store their own type descriptor - a string version of the data type or function signature associated with the symbol - which are essential for specifying the signatures of class fields and methods. A full explanation of descriptors is included in the Code Generation section. Moreover, the index of a variable within the symbol table is used as its index in the Stack Frame (also elaborated on in Code Generation).

The rows in the Symbol Table are stored in a list, sequentially, in the order that they're first encountered - which can then be searched linearly. I decided against using some form of ordered list or hash table, as the positions of the variables in a Symbol Table are vital for getting their indexes for Code Generation (*i.e. if a parameter was to be moved to a different index, the program would break as it accidentally accesses the wrong data*). Thus, the approach I decided on was to keep the rows in a precise order for this purpose. It was also appropriate as it's unlikely a program written in FearnLang is particularly complex or uses a large number of variables; this means a simple list is efficient enough for the task at hand.

The below diagram shows the the inheritance and composition relationships between semantic classes (SymbolTable and the Rows)



# The Fearn Programming Language

## Type Analysis

Every Expression and Statement Node implements a common `validate()` method. This is used to perform a DFT to validate the data types of every node in the tree, as well as performing other semantic tasks. The structure and requirements for semantic validation differ by the type and state of each node, and so each class that can be instantiated must have its own implementation. This traversal passes a symbol table down to every node, reflecting the symbol table of the scope it exists within. Each node's `validate` method verifies the correctness of the entire subtree, for which it is the root. For example, expressions validate the types of its own operands, as well as returning the type specifier for the expression. For instance, the binary expression  $e_1 + e_2 \dots$

- Both expressions must evaluate to the same data type
- That data type must be `int`, `float`, or `string` (as the expression can also represent concatenation)
- If the above two conditions are satisfied, the method will return the common data type of the two expressions, otherwise an error is raised
  - The error message will include the `toString()` version of the node, to assist the developer in finding the location of the error by effectively showing them the offending code.

```
// Validate method for an index expression (e.g. x[1] )
function validate(symTable):
    seq_type = sequence.validate(symTable)
    index_type = index.validate(symTable)

    if seq_type.getClass() != ArraySpecifier.class and not
seq_type.equals(new PrimitiveSpecifier(PrimitiveDataType.STR)):
        Reporter.ReportErrorAndExit(
            toString() + ": Can only take index of Arrays and Strings."
        )

    if not index_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT)):
        Reporter.ReportErrorAndExit(
            toString() + ": Index can only be an int."
        )

    if seq_type.equals(new PrimitiveSpecifier(PrimitiveDataType.STR)):
        expression_type = seq_type
        return expression_type

    seq_arr_spec = cast(ArraySpecifier, seq_type)
    if seq_arr_spec.dimensions == 1:
        expression_type = seq_arr_spec.element_type
    else:
        expression_type = new ArraySpecifier(
            seq_arr_spec.element_type,
            seq_arr_spec.dimensions - 1
        )
    return expression_type
```

## The Fearn Programming Language

Above is a pseudocode example of the common `validate()` method. This example is specifically for index expressions, for which the method checks that...

- A) The expression being indexed is of a type that can be indexed, either a string or an array (of any type)
- B) The index is an integer

Static members of the Code Generator class are used in order to store important data about the state of the traversal. This includes the Global Symbol Table (containing the global variables, as well as rows for each struct and function), the return type specifier of the function currently being validated, to ensure return statements return the correct sort of data, and the loop depth to validate the appearance of jump statements (break, continue, etc).

If a node is found to be invalid, an error is printed, using the `toString()` method to show the offending statement as to help the user find the bug in their program. Otherwise, the type of the element at the index, either a primitive data type or the type of another, nested array is returned to the next node up in the AST.

To see how type validation works for other expressions and classes of the AST, please see the full source code in [Appendix C](#).

## Code Generation

The Code Generation is performed in three stages, with the majority of processing taking place in the third stage.

- 1) Struct Generation:** Class files are generated for each struct, with the fields of the struct being created as public fields of the class. During Function Generation, when structs are instantiated, new
- 2) Global Generation:** A main program class file is generated. Global variables in the Fearn program are added to the class as a public static field, so it is available in all methods.
- 3) Function Generation:** Each function is generated as a public, static method of the class. The body of the function is generated by calling the body's `GenerateBytecode()` method.

A `CodeGenerator` object is responsible for generating the program and struct class files, and producing class files in a new build directory, in the same directory as the source code, to promote well-structured projects. The `GenerateBytecode()` method is a common method of every AST Node that is an executable structure in the program: Declarations, Statements, and Expressions.

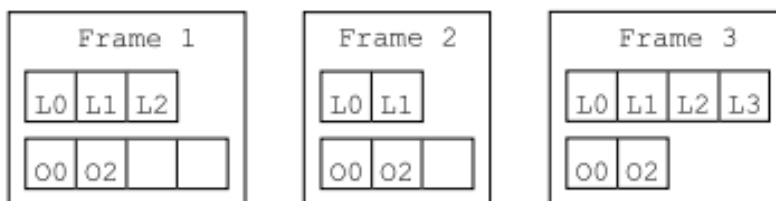
# The Fearn Programming Language

## The Java Virtual Machine (JVM) and ASM

The Java Virtual Machine is the process Virtual Machine developed alongside the Java language. It interprets and executes Bytecode stored in compiled .class files. Each thread running in the JVM has a call stack, containing the stack frames for each function executing in the program. Each stack frame is made up of two major components.

- **Local Variables:** These are stored at indexes in an array. In static methods (which I'm generating), the first variables in the stack frame are the parameters of the method, in order. During generation, my program uses the index of a variable in the symbol table as its index in the frame.
- **Operand Stack:** Stack which holds the objects/values that are operated upon. Variables are loaded into the stack, and can be assigned using the value at the top of the stack.

The diagram shows example stack frames, with local variables (Lx) and Operands (Ox) are stored separately.



### Descriptors

Type Descriptors are used to describe the type of data and variables textually within the JVM. Some examples from the ASM documentation are included below.

Method declaration in source file	Method descriptor
<code>void m(int i, float f)</code>	<code>(IF)V</code>
<code>int m(Object o)</code>	<code>(Ljava/lang/Object;)I</code>
<code>int[] m(int i, String s)</code>	<code>(ILjava/lang/String;) [I</code>
<code>Object m(int[] i)</code>	<code>([I)Ljava/lang/Object;</code>

Java type	Type descriptor
boolean	Z
char	C
byte	B
short	S
int	I
float	F
long	J
double	D
Object	Ljava/lang/Object;
int []	[I
Object [] []	[[Ljava/lang/Object;

### ASM

My program generates bytecode instructions using the ASM library, which provides Visitor classes to generate class files and write instructions.<sup>22</sup> Specifically, my program uses ClassWriters to create .class files for structs and the main class in the build directory. Moreover, it uses MethodVisitors to visit each instruction within a function. Another class I utilise is the MethodNode class, a derived class of the method visitor that allows me to perform optimisations on the bytecode to eliminate redundant operations.

<sup>22</sup> To learn more about ASM's full functionality, and how its API works for generating classes and methods, please see the ASM guide (<https://asm.ow2.io/asm4-guide.pdf>)

# The Fearn Programming Language

## Struct Generation

Structs are generated in separate class files in the build directory, prefixed with a '\$' to differentiate them from standard program files. The procedure for generations is detailed in the pseudocode below.

```
function GenerateStructs(structs: List<Struct>):  
    for each struct in structs:  
        // Initialize a ClassWriter to dynamically create a Java  
        // class for the struct  
        classWriter = new ClassWriter(COMPUTE_MAXS)  
        // Start defining the class structure  
        classWriter.visit(  
            V19,  
            ACC_PUBLIC | ACC_SUPER,  
            "$" + struct.identifier,  
            null,  
            "java/lang/Object",  
            null  
        )  
  
        // Define the constructor method, getting the method  
        // signature from the symbol table  
        cv = classWriter.visitMethod(  
            ACC_PUBLIC,  
            "<init>",  
            GlobalSymbolTable.GetGlobalStructDescriptor(struct.identifier),  
            null,  
            null  
        )  
  
        // Begin generating constructor code  
        cv.visitCode()  
        cv.visitVarInsn(ALOAD, 0)  
        cv.visitMethodInsn(INVOKE_SPECIAL, "java/lang/Object",  
            "<init>", "()V", false)  
  
        i = 0  
  
        // Generate fields and constructor instructions for each  
        // declaration in the struct  
        for each decl in struct.declarations:  
            descriptor =  
            SymbolTable.GenBasicDescriptor(decl.type)
```

## The Fearn Programming Language

```
// Define a field for the declaration in the class
classWriter.visitField(
    ACC_PUBLIC,
    decl.identifier,
    descriptor,
    null,
    null
)

// Load 'this' to the operand stack
cv.visitVarInsn(ALOAD, 0)

// Load argument to the operand stack
cv.visitVarInsn(ALOAD, ++i)

// Assign argument to field in the constructor
cv.visitFieldInsn(
    PUTFIELD,
    "$" + struct.identifier,
    decl.identifier,
    SymbolTable.GenBasicDescriptor(decl.type)
)

// Finish defining the constructor
cv.visitInsn(RETURN)
cv.visitMaxs(0, 0)
cv.visitEnd()

// Finish defining the class
classWriter.visitEnd()

// Specify the destination path for the generated class
file
    destination = Paths.get(buildPath.toString(),
String.format("%s.class", struct.identifier))

    try:
        // Write the generated class file to the specified
destination
        Files.write(destination, classWriter.toByteArray())
    catch IOException as e:
        // Report an error if writing fails
        Reporter.ReportErrorAndExit("Struct Gen Error: " +
e.toString())
```

# The Fearn Programming Language

```
// Report success with the generated class file path
Reporter.ReportSuccess(
    "GENERATED Struct File: " +
destination.toAbsolutePath() + ";",
    false
)
```

## Global Generation

At this point, the Code Generator starts to generate the main program class. This includes a standard constructor, as every class in Java requires one, as well as static fields and variables.

Global Variables are added as public, static fields of the main class. They are generated using their identifier, as well as their type specifier from the global symbol table. A static block is then used to assign these fields their initial values, if present.

## Function Generation

For each function, the ClassWriter visits a method of that name, using its method descriptor from the global symbol table. The symbol table for the function is stored in the static LocalSymbolTable attribute, so it can be accessed during generation (for retrieving descriptors and other data). The body's GenerateBytecode() method is then called.

The GenerateBytecode() method, included in all Statement and Expression classes, accepts a MethodNode (essentially a list of instructions) as an argument, and adds to that list the instructions needed to implement that node. For example, taking our binary expression  $e_1 + e_2$  from earlier in this chapter, the expressions GenerateBytecode method would ...

1. Call  $e_1$ .GenerateBytecode(MethodNode) &  $e_2$ .GenerateBytecode(MethodNode)
  - The implementation of these expression (whatever they are) leaves the value they evaluate to on the top of the operand stack
2. Determine and perform the appropriate operation
  - a. If the expression\_type of this expression (a property of every expression object, the TypeSpecifier of the data type they evaluate to) is an integer, cast both expression from Integer Objects to primitive integers, add them, and cast the result to an integer object
  - b. If the expression is a float, cast the expressions to primitive doubles, add them, and cast the result to a Double object
  - c. If the expression results in a string, call the concatenation method from the Fearn Runtime (this is explained later).

# The Fearn Programming Language

## Generating Expressions

For expression nodes, the bytecode generation methods are implemented to follow two rules:

- The bytecode generated by any one Expression must always leave the value the expression evaluates at runtime at the top of the Operand Stack
  - AssignExpression objects bend this rule, as no value is left on top of the stack after the value has been loaded into a local variable.
- This resulting value must be an object

This second rule is designed to restrict the amount of processing necessary to determine the instruction necessary at any one point in the program. This is because JVM instructions are often typed when working with primitive values (for instance, `ILOAD` and `ISTORE` are used to load from and store to primitive integer variables in the method's stack frame). However, knowing that the value being produced by an expression is always an object allows me to use Object instructions (such as `ASTORE` and `ALOAD`) in most scenarios, casting to primitive types when necessary (such as the addition example from above).

## Generating Statements

The procedure for generating statements varies wildly by class. Expression Statements, for instance, simply generate the bytecode for their expression object, then include a `POP` instruction if the expression has left a value on the stack.

More complicated are statements involving jumps, such as `IterationStatement`, `SelectionStatement`, and `JumpStatement`. These take advantage of ASM's `Label` class to create labels the program can jump to. For example, A `SelectionStatement` uses two labels - one to go to the else branch (if the condition evaluates to false), and another to go to the end of the statement - included after the else branch, so if the initial condition is true, the program jumps to the end of the statement rather than falling through to the else branch. `Iteration Statements` use three labels - at the front, before the iteration statement (which runs every loop), and at the end. These are added to a label stack - a static member of the Code Generator - so jump statements can refer to the label they need to go to. Once the code for the iteration has been generated, the labels are popped off the stack to avoid future jump statements referencing the wrong labels.



# The Fearn Programming Language

## Optimisations

The Fearn Compiler performs a small number of optimisations to eliminate redundant instructions in the program.

- **Cast Optimisations:** Generating bytecode using the AST results in many instances where a value at the top of the stack is cast from a primitive value to an object, then immediately back to a primitive type, due to the instructions being generated by different nodes with no context as to how they are being used in the program. To improve the efficiency and reduce the size of binaries the Fearn Compiler generates, after functions are generated, the program loops through every instruction in the MethodNode object, deleting any instance of these redundant casts occurring.
- **Dead-Code Elimination:** Instructions within a compound statement after a jump statement are unreachable. Thus, the program stops generating bytecode for the expression after an instance of this statement (break, continue, or return) has been encountered.

## Import Generation

The Code generation package includes an ImportCompiler, called during the AST Construction, which recursively calls a Compile method to build the other program, storing it in the same build directory.

Code Generator objects for each script file are added to a generator stack, and popped off when they are no longer needed. This is so the correct program names for the code currently being processed is always available in the object at the top of the stack.

## Status Reporting & CLI

The Fearn Compiler, like the vast majority of compilers, is a command-line application. The compiler includes two unique commands

1. **FearnC** : the Fearn Compile command, called along with the relative path to a *.fearn* source file. It calls the compiler (stored in a .jar file), and reports the status of the compilation - including any errors that the compiler has encountered, including bold and informative error messages, and the files it generates
2. **FearnRun** : This is an abstraction on the java command, which runs the compiled program.

Errors are designed to be as informative as possible, so the learner developer can easily fix it.

- A custom error listener is used alongside ANTLR, which prints out the exact line and column at which the error occurs, along with the error message ANTLR automatically generates.

## The Fearn Programming Language

- When a semantic error occurs, the offending code is printed to the terminal. The code itself is not preserved at this stage; however, all AST Node classes have a `toString()` method, allowing the compiler to print exactly how they would look in the source code.
  - This should enable the programmer to see exactly where the error was made in their program.
- All compilation errors include the name of the file in which they occurred, to ensure the developer knows where to look

## Examples

### *Successful Compilation*

```
>> FearnC main.fearn
FearnC: GENERATED Program      : C:\Users\thoma\Desktop\NEA\Example
Scripts\build\main.class;
FearnC: Compilation Successful!
      -> Run `cd build ; FearnRun main [args...]\` to run Program
```

### *Successful Execution*

```
>> FearnRun main
Hello, World!
```

### *Successful Compilation (multiple files - imports and structs)*

```
>> FearnC Imports\main.fearn
FearnC: GENERATED Struct File : C:\Users\thoma\Desktop\NEA\Example
Scripts\Imports\build\Person.class;
FearnC: GENERATED Program      : C:\Users\thoma\Desktop\NEA\Example
Scripts\Imports\build\Person.class;
FearnC: GENERATED Program      : C:\Users\thoma\Desktop\NEA\Example
Scripts\Imports\build\main.class;
FearnC: Compilation Successful!
      -> Run `cd Imports\build ; FearnRun main [args...]\` to run Program
```

### *Type Error*

```
>> FearnC main.fearn
FearnC (main): ERROR: let x : int = "Howdy!"; - Cannot assign str to int
```

### *Function Definition Missing*

```
>> FearnC main.fearn
FearnC (main): ERROR: Function func is not defined.
```

### *Variable Definition Missing*

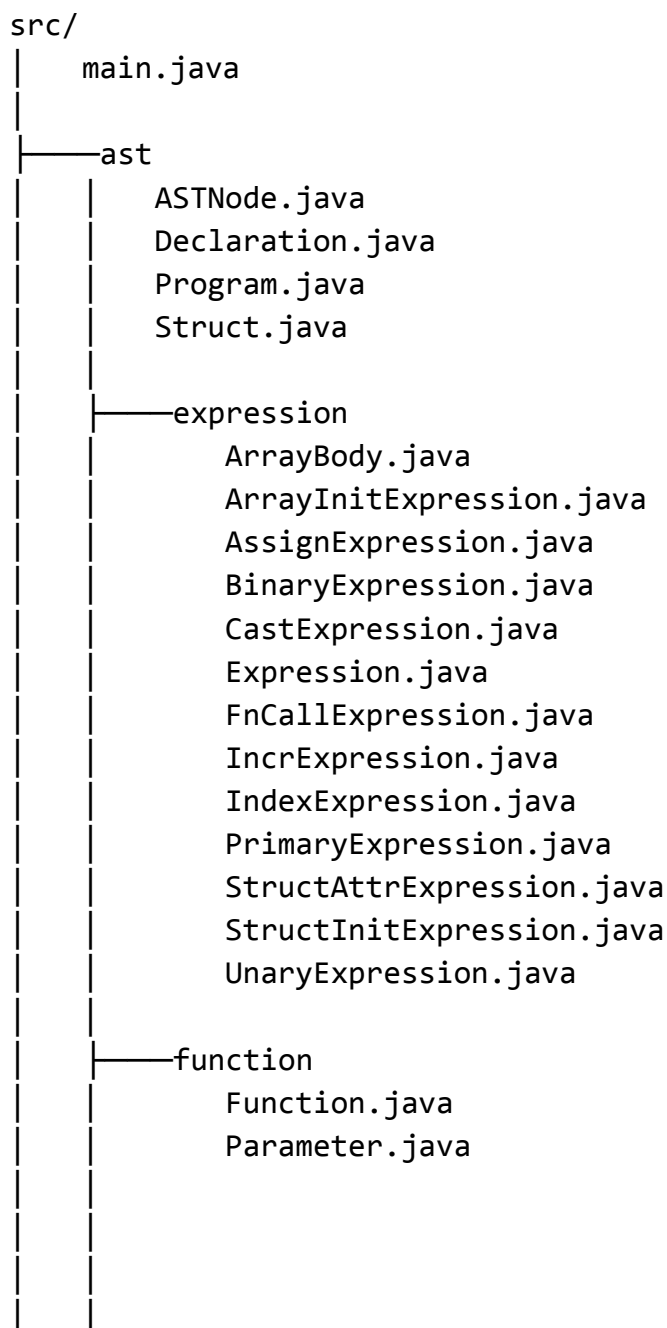
```
>> FearnC main.fearn
FearnC (main): ERROR: Line 8 : Variable Identifier Unknown in Scope: x
```

# Technical Solution

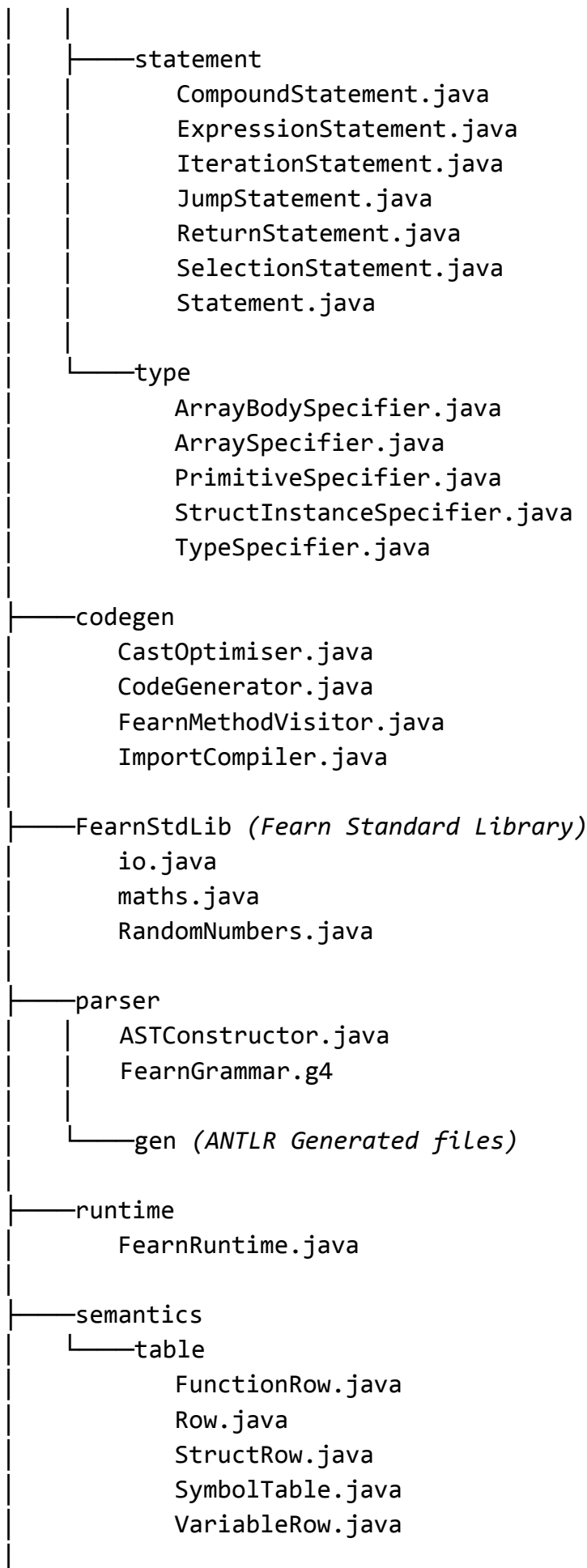
## Project Structure

This diagram shows the file structure of the `src` directory, containing my project's source code. To see the purpose and functionality of each file, please refer to [Appendix C](#).

**NB:** `parser/gen` is the directory containing the grammar-specific lexer/parser files, generated automatically by ANTLR. I have chosen not to include the contents of this directory in this document, as the code within is not created by me.



# The Fearn Programming Language



# The Fearn Programming Language

```
|
|
|_util
    FearnErrorListener.java
    Helper.java
    Reporter.java
```

## Package Structure

Java programs are arranged into packages. The packages included in FearnC are included below

<b>ast</b> : Contains the classes that represent nodes of the Abstract Syntax Tree <ul style="list-style-type: none"><li>• <b><i>expression</i></b> : Classes to represent expressions</li><li>• <b><i>statement</i></b> : Classes to represent statements</li><li>• <b><i>type</i></b> : Classes to represent data types (TypeSpecifiers)</li><li>• <b><i>function</i></b> : Classes for the high-level representation of functions</li></ul>
<b>codegen</b> : Classes for Code Generation tasks, including program/struct class generation, and bytecode optimisation
<b>parser</b> : Classes for Lexical Analysis and Parsing of Fearn Source Code, into an AST <ul style="list-style-type: none"><li>• <b><i>gen</i></b> : The ANTLR-generated Lexer/Parser for the Fearn grammar</li></ul>
<b>semantics.table</b> : Classes related to the Symbol Table, such as the main SymbolTable class, and Row classes
<b>util</b> : Utility classes, that provide simple functionality, such as error reporting
<b>FearnStdLib</b> : The Fearn Standard Library, containing the implementations, in Java, or built-in functions - that can be referenced by the bytecode FearnC generates. <ul style="list-style-type: none"><li>→ <i>This is not directly referenced by any other part of the compiler. Instead, it exists to give compiled programs access to the functions at runtime. It is packaged in the FearnC.jar file the project compiles to, and so is available on the Java CLASSPATH at runtime.</i></li></ul>

In addition to Java source code, there are also several release files that support the operation of the Fearn language

- **FearnC.cmd** : Windows Command File that implements the FearnC command, by calling the FearnC.jar file, which the compiler is packaged to
- **FearnRun.cmd** : Windows Command File that implements the FearnRun command (effectively a wrapper for the java command), that calls compiled Fearn programs (the generated class files within the generated build directory)

# The Fearn Programming Language

## Objectives Achieved

This section details, point-by-point, the places within my project where I have achieved my objectives, as set out in the Analysis section - specifically the features and syntax included in the language, and how they are internally represented and implemented (covering objectives 1 and 2). The Fearn Compiler consists of ~6,500 lines of code (including comments), which can be viewed - in full - in [Appendix C](#). Due to the size of the project, only a small selection of examples can be featured below, but I will make reference to other files where examples can be found - where relevant.

Please note, some of the below refer to the traversals performed by the ASTConstructor class on the ANTLR-generated parse tree of a Fearn program. These are included in order to demonstrate how these traversals are done, in order to convert the ANTLR Parse Tree into a more meaningful (and useful) Abstract Syntax Tree. The file ASTConstructor.java is over 1000 lines long, however, so not every section below includes the specific method that is used to derive that construct. If you wish to learn more about these traversals, they are documented in full in Appendix C - Full Source Code.

### 1.x: Define a Syntax for FearnLang

Examples of the syntax for the Fearn Language can be found in Appendix B. The syntax itself is defined using extended BNF, in the grammar file FearnGrammar.g4 - which is interpreted by the ANTLR parser generator to create a specific lexer/parser (in the parser.gen package) that can interpret FearnLang.

#### FearnGrammar.g4

/\*

FearnGrammar.g4

Unlike every other file that forms the compiler, this is not a java source file. This is an ANTLR grammar file, written using EBNF. The difference between EBNF and regular BNF is that it allows the use of regex-style metacharacters, allowing me to write more concise production, and less of them. This allows me to write a far neater grammar.

Each rule has one or more patterns associated with it, containing the rules or tokens it is made up of.

The rule at the bottom of file (in UPPER CASE) are token rules - used by the lexer to match patterns for tokens

## The Fearn Programming Language

in the source code, which have a large language that couldn't be explicitly specified (such as keywords, e.g. 'new', 'fn', 'int', etc.)

Some rules have labels next to them (indicated with #). These instruct ANTLR to generate the custom walker class with separate visit methods and contexts for each

```
*/

grammar FearnGrammar;

/* DATA TYPES AND TYPE SPECIFIERS */

// type_name matches the literal names of the 4 primitive types
type_name
    : 'int'
    | 'float'
    | 'bool'
    | 'str'
    ;

// type_specifiers can describe a type which is primitive, an array,
// or a struct
type_specifier
    : type_specifier_primitive
    | type_specifier_struct
    | type_specifier_arr
    ;

// primitive specifiers are just the type name (e.g. let n : int; )
type_specifier_primitive
    : type_name
    ;

// To use a struct as a data type, it can be done in the form $[MyStruct]
// A $ prefix is used to differentiate a struct instance specifier from
// other source code references to the struct
type_specifier_struct
    : '$' IDENTIFIER
    ;

// An array specifier will be the type specifier of the elements contained,
```

# The Fearn Programming Language

```
// and the N '[]'s, to indicate an N-dimensional array
type_specifier_arr
    : (type_specifier_primitive | type_specifier_struct ) ('[]')+
    ;

/* HIGH LEVEL STRUCTURES */
// Imports start with the 'import' keyword, followed with either
// -> An identifier, indicating the import of a standard library module
//      (e.g. `io` or `maths`)
// -> The 'from' keyword, followed by a string literal, of the relative
//      path, from the main program file, of the file being imported
module_import
    : 'import' (IDENTIFIER | 'from' STR_LIT)
    ;

// Fearn Programs, at their root, are 0 or more imports, followed by one
// or more instances of a function, global variable, or struct definition
// -> EOF is a special token for the end of a file, included to ensure
//      this production as consumed the entire source file, and would raise
//      an error if it had, due to poor syntax, not consumed it all.
program
    : (module_import)* (function | declaration | struct_def )+ EOF
    ;

// Functions are defined using the 'fn' keyword, a list of parameters, a
// return type (a type specifier or 'void'), and then a compound statement
// for the function body.
// -> The '=>' symbol is simply to indicate the data type returned.
function
    : 'fn' IDENTIFIER '(' (( parameter ',' )* parameter)? ')' '=>' ( type_specifier | 'void' )
compound_statement
    ;

// Function parameters have an identifier, and a data type specifier
parameter
    : IDENTIFIER ':' type_specifier
    ;

// Structs are defined using the 'struct' keyword, an identifier, as well as a
// list of 0 or more declarations, used to declare variables. These declared
// variables become the attributes of the struct, with each instance of a struct
// having a value for each attribute.
struct_def
    : 'struct' IDENTIFIER '{' declaration* '}'
    ;
```



# The Fearn Programming Language

```
// Declarations represent the creation of new variables within a defined scope.
// It uses the 'let' keyword, an identifier for the variable, and a type specifier.
// An expression (to initialise the variable) can also be added.
declaration
    : 'let' IDENTIFIER ':' type_specifier ( '=' expression)? ';'
    ;

/* STATEMENTS */

// Statements can be one of the five below types
statement
    : compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    ;

// A compound statement, syntactically, is a collection of declarations and statements,
// between '{}'.
compound_statement
    : '{' (declaration | statement)* '}'
    ;

// An expression statement is an expression (e.g. a function call), or an assignment
// (changing the value of a variable)
expression_statement
    : expression ';'          # simple_expr_stmt
    | assign_expression ';'    # assign_expr_stmt
    ;

// A selection statement is a standard 'if-else' statement. It can be either:
// -> A single if, where the body is only run if the condition is met
// -> An if-else chain, where another body/selection is run if the first doesn't
selection_statement
    : 'if' '(' expression ')' compound_statement
# single_if
    | 'if' '(' expression ')' compound_statement 'else' (compound_statement |
selection_statement)      # if_else
    ;

// Fearn doesn't support while loops, but does feature a highly permissive for loop,
// which can perform the same purpose. The following few rules all the user to create
// a loop, in the form `for (int i = 0; i < 10; i++) {...}`
```

# The Fearn Programming Language

```
iteration_statement
    : 'for' '(' init_expression continue_condition iteration_expression ')' compound_statement
    ;

// The initialisation can be a declaration, expression, or assignment

// The declaration requires no ';', as it's definition already matches an ending ';'
init_expression
    : declaration
    | expression ';'
    | assign_expression ';'
    | ';'
    ;

// The continue condition must be a boolean expression
continue_condition
    : expression ';'
    ;

// The iteration expression (run at the end of every loop) can be either an expression,
// assignment, or nothing at all
iteration_expression
    : expression
    | assign_expression
    |
    ;

// A jump statement is one of the below, which function as you'd expect in any C-like language
// The return expression is optional, as void functions don't return a value
jump_statement
    : 'continue' ';'           # cont_stmt
    | 'break' ';'             # break_stmt
    | 'return' expression? ';' # return_stmt
    ;

/* EXPRESSIONS */

// This is a recursive rule for all non-assignment expressions in Fearn (i.e. all code that
// evaluates to a data value). They are defined recursively, as they're no no way to tell what
// type of expression should be used in many contexts. For example, the unary not expression can
// operate on any expression that results a boolean value (e.g. a boolean function, variable,
// struct
// instance attribute, other boolean expressions in brackets, etc). Most of the below rules are
// self-explanatory.
// -> The dot expression can be used either to access a struct attribute, or to call a
```

# The Fearn Programming Language

```
//      function using Universal Function Notation / UFN ( a.f(x) == f(a, x) )
// -> The order of rules defines their precedence, and what the parse tree looks like.
//      For example, the expression `a + b * c` is parsed as `(a + (b * c))`, because
//      multiplicative expressions are matched before additive expressions
// -> The arithmetic expressions use `op=(...)`, because the operators must be of the same
//      precedence
```

expression

```
  : IDENTIFIER                                # id_expr
  | literal                                    # lit_expr
  | array_init                                # arr_init_expr
  | struct_init                               # struct_init_expr
  | expression '.' expression                 # dot_expr
  | IDENTIFIER '(' ( ( expression ',' )* expression )? ')' # fn_call_expr
  | '(' expression ')'                       # brac_expr
  | expression '[' expression ']'            # index_expr

  | '+' expression                           # u_plus_expr
  | '-' expression                           # u_minus_expr
  | '!' expression                           # u_not_expr

  | op=('++' | '--') expression              # pre_inc_expr
  | expression op=('++' | '--')              # post_inc_expr
  | '(' type_name ')' expression             # cast_expr

  | expression '^' expression               # exp_expr
  | expression op=('*' | '/' | '%') expression # mult_expr
  | expression op=('-' | '+') expression      # add_expr

  | expression '<' expression                 # less_expr
  | expression '>' expression                 # greater_expr
  | expression '<=' expression                # less_eq_expr
  | expression '>=' expression                # greater_eq_expr
  | expression '==' expression               # eq_expr
  | expression '!=' expression               # not_eq_expr
  | expression '&&' expression                 # and_expr
  | expression '||' expression               # or_expr
  ;
```

```
// Literal matches any literal token
```

literal

```
  : STR_LIT | BOOL_LIT | INT_LIT | FLOAT_LIT ;
```

# The Fearn Programming Language

```
// array_init matches an array initialisation, which requires a type specifier for the element
types,
// and either...
// A) One or more dimensions (in square brackets)
// B) A body that represents its initial value
// This syntax follows Java's notation for initialising fixed-length arrays
array_init
    : 'new' ( type_specifier_primitive | type_specifier_struct ) ('[' expression '']')+
    | 'new' ( type_specifier_primitive | type_specifier_struct ) ('[]')+ array_body
    ;

// An array body is 0 or more elements, which may themselves be array bodies (for
multi-dimensional arrays)
array_body
    : '{' (array_body ',')* array_body '}'
    | '{' (expression ',')* expression '}'
    | '{}''
    ;

// A struct is initialised with the identifier of the struct (which we're making an instance
of), and arguments
// to set that struct's initial state
struct_init
    : 'new' IDENTIFIER '(' ( ( expression ',')* expression )? ')'
    ;

// Assignment expressions are used to change the value of a variable
assign_expression
    : expression assignment_operator expression
    ;

assignment_operator
    : '=' | '+=' | '-=' | '*=' | '/=' | '%=' ;

/*
LEXER RULES

The following defines language-specific tokens, at aren't just simple words or punctuation, and
that have to match
a certain pattern, defined using regular expressions.

*/
```

# The Fearn Programming Language

```
// Define Fragments, used in the below token rules

// Digits
fragment D : [0-9] ;

// Letters
fragment L : [a-z]
           | [A-Z]
           | '_'
           ;

/* Define Tokens for Literals */
// Integers are one or more digits
INT_LIT    :  D+                      ;
// Floating-point numbers are two sets of 1 or more digits, separated by a dot
FLOAT_LIT  :  D+'.'D+                  ;
// A string is any sequence of characters, in between quotes, "" or ''
STR_LIT    :  '"'(.)*?'"'| '\''(.)*?\'' ;
// A boolean value is a true or false value
BOOL_LIT   :  'true' | 'false'        ;

// Identifiers (defined by the programmer) must be a letter, followed by 0 or
// more letters or digits
IDENTIFIER :  L(L|D)*                  ;

/* Ignore (certain) whitespace */
WS         :  ( ' ' | '\t' | '\n' | '\r' )+ -> skip ;

/* Ignore C-Style Comments */
BLOCKCOMMENT :  '/' '*' .*? '/' -> skip ;
LINECOMMENT  :  '/' '/' ~[\r\n]* -> skip ;
```

# The Fearn Programming Language

## 1.1: Must allow the user to create and modify variables

Variables in Fearn are created using declarations, in the form described by the declaration rule in `FearnGrammar.g4`, where the `type_specifier` describes the data type (e.g. `int[]`, `str`, `$MyStruct`), and the `expression` is used to initialise the new variable to a value.

When the file is parsed, the parse tree is traversed by the `ASTConstructor` class, to convert it into a more meaningful Abstract Syntax Tree. The method to visit a declaration node in the `ASTConstructor` is below.

### `ASTConstructor.java` (*visitDeclaration method*)

```
/* DECLARATIONS
 *
 * New variables are declared at the start of the compound statement in
 * which they're in scope. This function...
 * 1) Gets the identifier for the variable, its type specifier, and the
 *    initialisation expression, if present.
 * 2) Adds the identifier to the stack
 * 3) Adds a row for the variable to the local/global symbol table (this
 *    will throw an error if another variable of the same name exists in
 *    scope), by adding the row to the SymbolTable at the top of the
 *    symbolTableStack
 * 4) Returns a Declaration object
 *
 */
public Declaration visitDeclaration(FearnGrammarParser.DeclarationContext ctx)
{
    String identifier = ctx.IDENTIFIER().getText();
    symbolAnalysisStack.push(identifier);
    TypeSpecifier type_spec = (TypeSpecifier)visit(ctx.type_specifier());
    Expression init_expression = null;
    if (ctx.getChildCount() > 5)
    {
        init_expression = (Expression)visit(ctx.expression());
    }
    symbolTableStack.peek().addRow(
        new VariableRow(
            identifier,
            type_spec
        )
    );
    return new Declaration(identifier, type_spec, init_expression);
}
```

## The Fearn Programming Language

This method returns (to the method constructing the AST node above the declaration in the tree) an object of class Declaration. The Fearn Compiler uses AST Node classes to represent constructs in the program, as well as to validate and generate bytecode for that construct. For Declarations, bytecode is generated in two places (both included below).

- Declarations within a local scope (a Compound Statement in a Function body) are generated with the Declaration class
- Global Declarations have their bytecode generated by the CodeGenerator class (which is the class responsible for taking the bytecode of structs, functions, and global variables - and converting them into Java .class files)

### Declaration.java

```
package ast;

import static org.objectweb.asm.Opcodes.ASTORE;

import org.objectweb.asm.MethodVisitor;

import ast.expression.Expression;
import ast.type.TypeSpecifier;
import codegen.CodeGenerator;
import semantics.table.SymbolTable;
import util.Reporter;

/* Declaration.java
 *
 * Represents a Variable Declaration in the AST.
 *
 * Fields:
 * -> identifier: the string identifier of the variable, in the source code
 * -> type: TypeSpecifier for the datatype of the variable
 * -> init_expression: The expression used to initialise the variable (may be null)
 */

public class Declaration extends ASTNode {

    public String identifier;
    public TypeSpecifier type;
    public Expression init_expression;

    // Standard Constructor
```

## The Fearn Programming Language

```
public Declaration(String id, TypeSpecifier t, Expression e)
{
    identifier = id;
    type = t;
    init_expression = e;
}

// String representation takes the same form as a declaration in source code
// It reclusively calls the toString methods for the type, and init expression
// The ternary expression only includes the init_expression if it is not null
@Override public String toString()
{
    return (init_expression == null) ?
        "let " + identifier + " : " + type.toString() + ";"
        : "let " + identifier + " : " + type.toString() + " = " +
init_expression.toString() + ";"
    ;
}

// GenerateBytecode() generates the init_expression bytecode (leaving its value
// at the top of the operand stack), and then stores it at the variable index
// indicated by the LocalSymbolTable.
// -> This is ONLY called to generate local variables within functions (globals
//      are handled in CodeGenerator.java), so no need to handle global variables
public void GenerateBytecode(MethodVisitor mv)
{
    if (init_expression != null) {
        init_expression.GenerateBytecode(mv);
        mv.visitVarInsn(ASTORE, CodeGenerator.LocalSymbolTable.GetIndex(identifier));
    }
}

// validate() compares the expression_type of the init_expression to the type of the
// variable. If they don't match, an error is raised. Otherwise, declaration is valid
// and the method returns. The method will return immediately if init_expression is
// null, as a declaration without initialisation cannot be invalid (at this stage,
// errors such as two variables in the same function having the same name are caught
// by the SymbolTable during AST Construction)
public void validate(SymbolTable symbolTable) {
    if (init_expression == null) return;
    TypeSpecifier exprType = init_expression.validate(symbolTable);
    if (!type.equals(exprType))
```



## The Fearn Programming Language

```
{
    Reporter.ReportErrorAndExit(
        "Cannot assign " + exprType.toString() + " to " + type.toString(),
        this
    );
}
}
```

### CodeGenerator.java (*GenerateProgram*)

```
private void GenerateProgram(
    ArrayList<Function> functions,
    ArrayList<Declaration> global_declarations,
    Path finalProgramPath
) {
    // Create new class writer, to write program class
    ClassWriter classWriter = new ClassWriter(ClassWriter.COMPUTE_MAXS);

    // Class is a public, static class, with its name being the programName
    classWriter.visit(
        V19,
        ACC_PUBLIC | ACC_SUPER,
        programName,
        null,
        "java/lang/Object",
        null
    );

    // Generate Initial State of Global Variables (sv = StateVisitor)
    MethodVisitor sv = classWriter.visitMethod(
        ACC_STATIC,
        "<clinit>", // Indicates static block
        "()V", // Takes no arguments, and has void return type
        null,
        null
    );

    // Begin Defining Code
    sv.visitCode();

    // Add Global Declarations as public fields
    global_declarations.forEach(
```

## The Fearn Programming Language

```
(decl) -> {
    classWriter.visitField(
        ACC_PUBLIC | ACC_STATIC,
        decl.identifier,
        SymbolTable.GenBasicDescriptor(decl.type),
        null,
        null
    );

    if (decl.init_expression == null) { return; }

    decl.init_expression.GenerateBytecode(sv);
    sv.visitFieldInsn(
        PUTSTATIC,
        GeneratorStack.peek().programName,
        decl.identifier,
        SymbolTable.GenBasicDescriptor(decl.type)
    );
}
);

// Add return instruction
sv.visitInsn(RETURN);

// End Static Block Generation
sv.visitMaxs(0, 0);
sv.visitEnd();
...
```

FearnLang's strict type system is achieved using `validate()` methods in all Statement and Expression classes - that verify that the program follows the semantic rules of the language - for all possible constructs that can appear (for example, the `IterationStatement` class validates that the condition for a loop to continue will always evaluate to some boolean value). The types are represented by `TypeSpecifier` objects - which are included in the `ast.type` package in Appendix C.

The modification of variable values is done through assignments. The method for traversing these in the parse tree is trivial, and can be found in the `ASTConstructor` class in Appendix C. The AST Node for representing assignments, however, can be found directly below.

# The Fearn Programming Language

## AssignExpression.java

```
package ast.expression;

import static org.objectweb.asm.Opcodes.*;

import org.objectweb.asm.MethodVisitor;

import ast.type.TypeSpecifier;
import codegen.CodeGenerator;
import semantics.table.SymbolTable;
import util.Reporter;

/* AssignExpression.java
 *
 * Represents an Assignment in the AST.
 *
 * Fields:
 * -> AssignmentOperator (& operator): Enum that indicates the sort of assignment
 * -> target: The expression to be assigned to
 * -> expression: The value to assign to the target
 */

public class AssignExpression extends Expression {

    public static enum AssignmentOperator {
        Equals,
        AddEquals,
        SubEquals,
        MultEquals,
        DivEquals,
        ModEquals
    }

    public AssignmentOperator operator;

    public Expression target;
    public Expression expression;

    public AssignExpression(Expression t, Expression e, AssignmentOperator op)
    {
        target = t;
        expression = e;
        operator = op;
    }
}
```

# The Fearn Programming Language

```
@Override
public String toString()
{
    String opString = null;

    switch (operator) {
        case Equals      : opString = "="      ; break;
        case AddEquals   : opString = "+="     ; break;
        case SubEquals   : opString = "-="     ; break;
        case MultEquals  : opString = "*="     ; break;
        case DivEquals   : opString = "/="     ; break;
        case ModEquals   : opString = "%="     ; break;
    }

    return target.toString() + " " + opString + " " + expression.toString();
}

/* The Target Expression can be one of the following
 * -> A variable reference          (ASTORE at <INDEX> / PUTSTATIC if Global)
 * -> An Index Expression          (Iterative Loading of the array, then AASTORE)
 * -> A Struct Attribute Expression (PUTFIELD)
 */

/* To generate bytecode for an assignment...
 * 1) If assigning to a variable, generate the expression, then...
 *     a) If the variable is local, use ASTORE, with the index of the variable from
 *         the LocalSymbolTable
 *     b) If global, use PUTSTATIC, with the program name (the identifier for the
 *         program class) for the current generator, and the descriptor from the
 *         GlobalSymbolTable
 * 2) If assigning to an indexed location in an array...
 *     a) Generate the array (target.sequence)
 *     b) Generate the index (target.index) (casting it to primitive I)
 *     c) Generate expression (to be saved at index)
 *     d) Call AASTORE, storing the expression at the index, into the array
 * 3) Otherwise, the expression is to be assigned to an attribute of a struct
 *     instance ...
 *     a) Generate the struct instance
 *     b) Generate expression
 *     c) Use PUTFIELD to put the expression's value into the struct object's
 *         attribute
 */
```

# The Fearn Programming Language

```
@SuppressWarnings("unchecked")
@Override
public void GenerateBytecode(MethodVisitor mv) {

    if (target.getClass() == PrimaryExpression.class) // Variable Reference
    {
        expression.GenerateBytecode(mv);
        PrimaryExpression<String> t = (PrimaryExpression<String>)target;
        String identifier = t.value.toString();

        if (CodeGenerator.LocalSymbolTable.Contains(identifier)) {
            // Local Variable => ASTORE
            mv.visitVarInsn(ASTORE, CodeGenerator.LocalSymbolTable.GetIndex(identifier));
        } else {
            // Target is a Global Variable => PUTSTATIC
            mv.visitFieldInsn(
                PUTSTATIC,
                CodeGenerator.GeneratorStack.peek().programName,
                identifier,
                CodeGenerator.GlobalSymbolTable.GetVarDescriptor(identifier)
            );
        }
    }

    } else if (target.getClass() == IndexExpression.class) { // Index Expression

        IndexExpression targ = (IndexExpression)target;

        /*
         * To get an assign to an index expression, first I need to load the array by
         * generating the target. Then, I need to generate the index. Finally,
         * I need to generate the expression I wish to store and call AASTORE.
         */

        targ.sequence.GenerateBytecode(mv);
        targ.index.GenerateBytecode(mv);
        mv.visitMethodInsn(
            INVOKEVIRTUAL, "java/lang/Integer",
            "intValue", "()I", false
        );
        expression.GenerateBytecode(mv);
        mv.visitInsn(AASTORE);

    } else { // Struct Attribute Expression
```

## The Fearn Programming Language

```
StructAttrExpression targ = (StructAttrExpression)target;

/*
 * To assign to a struct attribute, I need to load the object,
 * then generate the expression. Then, I can use PUTFIELD to set
 * the attribute.
 */

targ.instance.GenerateBytecode(mv);
expression.GenerateBytecode(mv);

mv.visitFieldInsn(
    PUTFIELD,
    "$" + targ.struct_name,
    targ.attribute,
    targ.attr_descriptor
);
}
}

/* To validate an assignment...
 * 1) Ensure the target is an assignable expression (VariableReference,
 *    IndexExpression, or StructAttrExpression)
 * 2) Call HandleOperators (to handle operators like +=, %= etc).
 * 3) Check the TypeSpecifiers of the target and expression are the same
 * 4) Set expression_type to null and return it, as Assignments don't
 *    evaluate to a value
 */

@SuppressWarnings("rawtypes")
public TypeSpecifier validate(SymbolTable symTable) {

    if (target instanceof PrimaryExpression && ( (PrimaryExpression)target ).type ==
ExprType.VariableReference) {}
    else if (target.getClass() == IndexExpression.class) {}
    else if (target.getClass() == StructAttrExpression.class) {}
    else {
        Reporter.ReportErrorAndExit("Cannot assign value to " + target.toString(), this);
    }

    HandleOperators();

    // Check the TypeSpecifiers of the target and expression are equal
    TypeSpecifier targetType = target.validate(symTable);
```

# The Fearn Programming Language

```
    TypeSpecifier exprType =    expression.validate(symTable);

    if (!targetType.equals(exprType))
    {
        Reporter.ReportErrorAndExit("Cannot assign " + exprType.toString() + " to " +
targetType.toString(), this);
    }

    expression_type = null; // Assign Expression perform a job, they don't evaluate to
anything
    return expression_type;

}

/* HandleOperators handles the different operation assignments.
 *
 * It converts the expression to an instance of the class OpEqualsExpr, a derived
 * class of BinaryExpression. The purpose of doing this, over just using
 * BinaryExpression, is to override toString, to show the correct operator in case
 * of an error.
 *
 * Depending on the operation, an OpEqualsExpr is set. This means the
 * GenerateBytecode method doesn't need to handle each case.
 *
 * For example, the assignment 'x += 5' is simplified to 'x = x + 5', with the
 * 'x + 5' modelled as an OpEqualsExpr instance.
 *
 */

private void HandleOperators()
{

    class OpEqualsExpr extends BinaryExpression {

        public OpEqualsExpr(Expression op1, Expression op2, ExprType op) { super(op1, op2,
op ); }

        @Override
        public String toString() {

            String opString = null;

            switch (Operation) {
                case ExprType.Add    : opString = "+="; break;
                case ExprType.Sub    : opString = "-="; break;
            }
        }
    }
}
```

## The Fearn Programming Language

```
        case ExprType.Mult : opString = "*="; break;
        case ExprType.Div  : opString = "/="; break;
        case ExprType.Mod   : opString = "%="; break;
        default: break;
    }
    return String.format("%s %s %s", Op1.toString(), opString, Op2.toString());
}
}

switch (operator) {
    case Equals:
        return;
    case AddEquals:
        expression = new OpEqualsExpr(target, expression, ExprType.Add);
        return;
    case SubEquals:
        expression = new OpEqualsExpr(target, expression, ExprType.Sub);
        return;
    case MultEquals:
        expression = new OpEqualsExpr(target, expression, ExprType.Mult);
        return;
    case DivEquals:
        expression = new OpEqualsExpr(target, expression, ExprType.Div);
        return;
    case ModEquals:
        expression = new OpEqualsExpr(target, expression, ExprType.Mod);
        return;
}
}
}
```



# The Fearn Programming Language

## 1.2: Allow for Basic Mathematical operations

These operations, as well as other binary expressions such as logical comparisons, logical operations (AND, OR, and NOT), and exponentiation, are included in the expression rule of `FearnGrammar.g4`. These nodes in the parse tree are traversed by methods of `ASTConstructor`, some of which are below. All binary operations are represented by the `BinaryExpression` class, responsible for validating them, as well as generating bytecode that performs their functionality.

### `ASTConstructor.java` (*Traversals for multiplicative and additive expressions*)

```
/* BINARY OPERATIONS
 *
 * These all have two operands, and all function (roughly) in the same way
 * 1) Visit both operands
 * 2) Perform any additional processing (e.g. setting the ExprType)
 * 3) Return an Binary Expression object
 *
 * ExprType is again used to specify the operation
 */
// Multiplicative Expression (a (*|/|%) b )
@Override
public BinaryExpression visitMult_expr(FearnGrammarParser.Mult_exprContext ctx)
{
    Expression op1 = (Expression)visit(ctx.expression(0));
    Expression op2 = (Expression)visit(ctx.expression(1));
    ExprType type = null;

    switch (ctx.op.getText()) {
        case "*":
            type = ExprType.Mult;
            break;
        case "/":
            type = ExprType.Div;
            break;
        case "%":
            type = ExprType.Mod;
            break;
        default:
            break;
    }
    assert(type != null);
}
```

## The Fearn Programming Language

```
        return new BinaryExpression(op1, op2, type);
    }

    // Additive Expression (a (+|-) b )
    @Override
    public BinaryExpression visitAdd_expr(FearnGrammarParser.Add_exprContext ctx)
    {
        Expression op1 = (Expression)visit(ctx.expression(0));
        Expression op2 = (Expression)visit(ctx.expression(1));
        ExprType type = null;

        switch (ctx.op.getText()) {
            case "+":
                type = ExprType.Add;
                break;
            case "-":
                type = ExprType.Sub;
                break;
            default:
                break;
        }

        assert(type != null);
        return new BinaryExpression(op1, op2, type);
    }
}
```

### BinaryExpression.java

```
package ast.expression;

import org.objectweb.asm.MethodVisitor;

import static org.objectweb.asm.Opcodes.*;

import ast.type.PrimitiveSpecifier.PrimitiveDataType;
import ast.type.PrimitiveSpecifier;
import ast.type.TypeSpecifier;
import semantics.table.SymbolTable;
import util.Reporter;

/* BinaryExpression.java
```

# The Fearn Programming Language

```
*
* Represents a Binary Expression in the AST.
*
* Fields:
* -> Op1: Left operand
* -> Op2: Right operand
* -> Operation: The operation to be performed
*/

public class BinaryExpression extends Expression {

    public Expression Op1;
    public Expression Op2;
    public ExprType Operation;

    public BinaryExpression(Expression op1, Expression op2, ExprType op)
    {
        Op1 = op1;
        Op2 = op2;
        Operation = op;
    }

    @Override
    public String toString()
    {

        String opString = null;

        switch (Operation) {

            case Add      : opString = "+" ; break;
            case Sub      : opString = "-" ; break;
            case Mult     : opString = "*" ; break;
            case Div      : opString = "/" ; break;
            case Mod      : opString = "%" ; break;
            case Exponent : opString = "^" ; break;
            case Less     : opString = "<" ; break;
            case Greater  : opString = ">" ; break;
            case LessEq   : opString = "<="; break;
            case GreaterEq: opString = ">="; break;
            case LogicalAnd: opString = "&&" ; break;
            case LogicalOr : opString = "||" ; break;
```

## The Fearn Programming Language

```
        case Eq          : opString = "=="; break;
        case NotEq       : opString = "!="; break;

        default: break;
    }

    return String.format("%s %s %s", Op1.toString(), opString, Op2.toString());
}

/* To generate bytecode, in general...
 * 1) Generate both Operands (casting them if necessary)
 * 2) For each type of operation, call the corresponding instruction (or FearnRuntime
 *    method)
 * 3) Cast the result back to an object if necessary
 */

@Override
public void GenerateBytecode(MethodVisitor mv) {
    String t = "";

    if (Operation == ExprType.Eq || Operation == ExprType.NotEq)
    {
        // For these operations, the values on the stacks cannot be primitive
        Op1.GenerateBytecode(mv);
        Op2.GenerateBytecode(mv);
    }

    else if (Op1.expression_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT)))
    {
        // Generate operands, and cast to int
        t = "int";
        Op1.GenerateBytecode(mv);
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Integer", "intValue", "()I",
false);
        Op2.GenerateBytecode(mv);
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Integer", "intValue", "()I",
false);
    }

    else if (Op1.expression_type.equals(new
PrimitiveSpecifier(PrimitiveDataType.FLOAT)))
    {

```

## The Fearn Programming Language

```
t = "double";
// Generate operands, and cast to double
Op1.GenerateBytecode(mv);
mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Double", "doubleValue", "()D",
false);

Op2.GenerateBytecode(mv);
mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Double", "doubleValue", "()D",
false);
}
else if (Op1.expression_type.equals(new PrimitiveSpecifier(PrimitiveDataType.STR)))
{
    t = "str";
    Op1.GenerateBytecode(mv);
    Op2.GenerateBytecode(mv);
}
else if (Op1.expression_type.equals(new
PrimitiveSpecifier(PrimitiveDataType.BOOL)))
{
    t = "bool";
    Op1.GenerateBytecode(mv);
    Op2.GenerateBytecode(mv);
}
switch (Operation) {
    case Add:
        if (t == "int") {
            // Add
            mv.visitInsn(IADD);
            // Cast result to Integer
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf",
"(I)Ljava/lang/Integer;", false);
            return;
        } else if (t == "double") {
            // Add
            mv.visitInsn(DADD);
            // Cast result to Double
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Double", "valueOf",
"(D)Ljava/lang/Double;", false);
            return;
        } else { // String Concatenation
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "concat",
"(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;", false);
            return;
        }
}
```

# The Fearn Programming Language

```
    case Sub:
        if (t == "int") {
            // Add
            mv.visitInsn(ISUB);
            // Cast result to Integer
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf",
"(I)Ljava/lang/Integer;", false);
            return;
        } else { // Floats
            mv.visitInsn(DSUB);
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Double", "valueOf",
"(D)Ljava/lang/Double;", false);
            return;
        }

    case Mult:
        if (t == "int") {
            mv.visitInsn(IMUL);
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf",
"(I)Ljava/lang/Integer;", false);
            return;
        } else { // Floats
            mv.visitInsn(DMUL);
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Double", "valueOf",
"(D)Ljava/lang/Double;", false);
            return;
        }

    case Div:
        if (t == "int") {
            mv.visitInsn(IDIV);
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf",
"(I)Ljava/lang/Integer;", false);
            return;
        } else { // Floats
            mv.visitInsn(DDIV);
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Double", "valueOf",
"(D)Ljava/lang/Double;", false);
            return;
        }

    case Exponent:
```

# The Fearn Programming Language

```
        if (t == "int") {
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "exp",
"(II)Ljava/lang/Integer;", false);
            return;
        } else { // Floats
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "exp",
"(DD)Ljava/lang/Double;", false);
            return;
        }
    case Less:
        if (t == "int") {
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "less",
"(II)Ljava/lang/Boolean;", false);
            return;
        } else { // Floats
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "less",
"(DD)Ljava/lang/Boolean;", false);
            return;
        }
    case LessEq:
        if (t == "int") {
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "less_eq",
"(II)Ljava/lang/Boolean;", false);
            return;
        } else { // Floats
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "less_eq",
"(DD)Ljava/lang/Boolean;", false);
            return;
        }
    case Greater:
        if (t == "int") {
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "greater",
"(II)Ljava/lang/Boolean;", false);
            return;
        } else { // Floats
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "greater",
"(DD)Ljava/lang/Boolean;", false);
            return;
        }
    case GreaterEq:
        if (t == "int") {
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "greater_eq",
```

## The Fearn Programming Language

```
"(II)Ljava/lang/Boolean;", false);
    return;
} else { // Floats
    mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "greater_eq",
"(DD)Ljava/lang/Boolean;", false);
    return;
}
case Mod:
    mv.visitInsn(IREM);
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf",
"(I)Ljava/lang/Integer;", false);
    return;
case LogicalAnd:
    mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "and",
"(Ljava/lang/Boolean;Ljava/lang/Boolean;)Ljava/lang/Boolean;", false);
    return;
case LogicalOr:
    mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "or",
"(Ljava/lang/Boolean;Ljava/lang/Boolean;)Ljava/lang/Boolean;", false);
    return;
case Eq:
    mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "equals",
"(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Boolean;", false);
    return;
case NotEq:
    mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "equals",
"(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Boolean;", false);
    mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "not",
"(Ljava/lang/Boolean;)Ljava/lang/Boolean;", false);
    return;
default:
    Reporter.ReportErrorAndExit("Error in Generating Binary Expression.",
null);
    break;
}
}
```

/\* To validate...



## The Fearn Programming Language

```
* 1) Get types for both operands
* 2) Check they are valid types for the operation, and the same
*    -> Raise errors otherwise
* 3) Set expression_type to an appropriate TypeSpecifier, and
*    return it
*/
@Override
public TypeSpecifier validate(SymbolTable symTable) {

    TypeSpecifier op1_type = Op1.validate(symTable);
    TypeSpecifier op2_type = Op2.validate(symTable);

    switch (Operation) {

        // All cases where the operands must
        // both be numeric.
        case Mult:
        case Div:
        case Sub:
        case Exponent:
        case Less:
        case LessEq:
        case Greater:
        case GreaterEq:
            if (
                op1_type.equals(op2_type) &&
                (
                    op1_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT)) ||
                    op1_type.equals(new PrimitiveSpecifier(PrimitiveDataType.FLOAT))
                )
            ) {
                switch (Operation) {
                    case Mult:
                    case Div:
                    case Sub:
                    case Exponent:
                        expression_type = op1_type;
                        break;
                    default:
                        expression_type = new
PrimitiveSpecifier(PrimitiveDataType.BOOL);
                        break;
                }
            }
        }
    }
```

## The Fearn Programming Language

```
    }
    } else {
        Reporter.ReportErrorAndExit("Operands must be either (a) both ints, or
(b) both floats.", this);
    }
    break;

// Can work on numbers or strings
case Add:
    if (
        op1_type.equals(op2_type) &&
        (
            op1_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT)) ||
            op1_type.equals(new PrimitiveSpecifier(PrimitiveDataType.FLOAT)) ||
            op1_type.equals(new PrimitiveSpecifier(PrimitiveDataType.STR))
        )
    ) {
        expression_type = op1_type;
    } else {
        Reporter.ReportErrorAndExit("Operands must be either (a) both ints, (b)
both floats, or (c) both strings.", this);
    }
    break;

// Modulo only works on integers
case Mod:
    if ( op1_type.equals(op2_type) && op1_type.equals(new
PrimitiveSpecifier(PrimitiveDataType.INT)) ) {
        expression_type = op1_type;
    } else {
        Reporter.ReportErrorAndExit("Operands must both be ints.", this);
    }
    break;

// Both operands must be boolean
case LogicalAnd:
case LogicalOr:
    if ( op1_type.equals(op2_type) && op1_type.equals(new
PrimitiveSpecifier(PrimitiveDataType.BOOL)) ) {
        expression_type = op1_type;
    } else {
        Reporter.ReportErrorAndExit("Operands must both be boolean values.",
```

## The Fearn Programming Language

```
this);  
    }  
    break;  
  
    // Both must be of the same type  
    case Eq:  
    case NotEq:  
        if ( op1_type.equals(op2_type)) {  
            expression_type = new PrimitiveSpecifier(PrimitiveDataType.BOOL);  
        } else {  
            Reporter.ReportErrorAndExit("Operands must both be of the same type.",  
this);  
        }  
        break;  
  
    default:  
        Reporter.ReportErrorAndExit("An Error has occurred.", this);  
        break;  
  
    }  
  
    return expression_type;  
}  
}
```

# The Fearn Programming Language

## 1.3: Allow for text Input/Output, through the console

This is done through the `io` module, part of Fearn's Standard Library, along with the `maths` (mathematical functions and constants) and `random` (random numbers) modules. When imported, the `ASTConstructor` calls the `ImportCompiler` - a class responsible for importing elements of other programs into the primary program the user called to be compiled. In the case of Standard Library modules, the `GetStdLib` method is called, which creates a `SymbolTable` with the functions that module contains inside. These rows all have an owner (used when generating bytecode), which indicates the class on the user's Java CLASSPATH that the method/field belongs to.

Below are the `io` module implementation, as well as `ImportCompiler.GetStdLib`.

### io.java

```
package FearnStdLib;

import java.util.Scanner;

public class io {

    static Scanner scan = new Scanner(System.in);

    public static void print(String a) { System.out.println(a); }

    public static String input(String prompt)
    {
        String in;
        System.out.print(prompt);
        in = scan.nextLine();
        return in;
    }
}
```

# The Fearn Programming Language

## ImportCompiler.java (*GetStdLib*)

```
// This method handles the importing of standard library modules
// Each module has a case in the below statement, and these cases
// build and return a SymbolTable for the functions that module contains
public SymbolTable GetStdLib(String id) {
    CodeGenerator.GeneratorStack.push(new CodeGenerator());
    SymbolTable table = new SymbolTable();
    ArrayList<Parameter> params;
    // Switch makes standard library easy to expand
    switch (id) {
        case "io":
            // Set Program Name
            // This sets the row's owner, ensuring the bytecode for calling these
            // function calls refer the the correct package and class
            CodeGenerator.GeneratorStack.peek().programName = "FearnStdLib/io";
            // Add Print Function
            // Create new parameter list
            params = new ArrayList<>();
            // Add string parameter
            params.add(new Parameter(
                "", new PrimitiveSpecifier(PrimitiveDataType.STR)
            ));
            // Add to symbol table, with identifier print, and null return
            // type and local symbol table (irrelevant as this function has
            // no Fearn implementation)
            table.addRow(
                new FunctionRow(
                    "print",
                    params,
                    null,
                    null
                )
            );
            // Add Input Function
            // params remains the same (as both print and input take a single,
            // string argument)
            table.addRow(
                new FunctionRow(
                    "input",
                    params,
                    new PrimitiveSpecifier(PrimitiveDataType.STR),
                    null
                )
            );
            break;
```

## The Fearn Programming Language

```
        case "maths":
            ...
        case "random":
            ...
        default:
            Reporter.ReportErrorAndExit(
                "Standard library " + id + " does not exist.", null
            );
            break;
    }
    // Pop Generator, to return to primary program
    CodeGenerator.GeneratorStack.pop();
    // Return Symbol Table to primary compilation process
    return table;
}
```

# The Fearn Programming Language

## 1.4: The language must support string, integer, float, and boolean types

These four primitive types, as well as arrays of them, are supported in FearnLang. The Type Specifier rules of FearnGrammar.g4 specify the type names, as well as the syntax for using a struct as a type (i.e. a function can return an instance of a struct), as well as types that are arrays of primitive data types or struct instances. Data types are modelled within the compiler and AST using TypeSpecifier classes. The class for primitive data types (like the ones in the objective) is included below.

### PrimitiveSpecifier.java

```
package ast.type;

/* PrimitiveSpecifier.java
 *
 * TypeSpecifier to describe primitive data types. It uses a PrimitiveDataType
 * enum, which describes the data type as an INT, FLOAT, STR, or BOOL.
 *
 */

public class PrimitiveSpecifier extends TypeSpecifier {

    public enum PrimitiveDataType {
        INT,
        FLOAT,
        BOOL,
        STR
    }

    public PrimitiveDataType element_type;
    public PrimitiveSpecifier(PrimitiveDataType eleT)
    {
        type = Category.Primitive;
        element_type = eleT;
    }

    @Override
    public String toString()
    {
        return element_type.name().toLowerCase();
    }
}
```

# The Fearn Programming Language

## 1.5: Fearn must be able to cast data to primitive types

Casting in Fearn is done using cast expressions (described syntactically within the expression grammar rule), and represented by the CastExpression class (derived from the abstract Expression class). This class is included below.

### CastExpression.java

```
package ast.expression;

import org.objectweb.asm.MethodVisitor;
import static org.objectweb.asm.Opcodes.*;

import ast.type.PrimitiveSpecifier.PrimitiveDataType;
import ast.type.PrimitiveSpecifier;
import ast.type.TypeSpecifier;
import semantics.table.SymbolTable;
import util.Reporter;

/* CastExpression.java
 *
 * Represents a Type Cast in the AST.
 *
 * Fields:
 * -> target: The PrimitiveDataType being cast to
 * -> Operand: The expression to be cast
 */

public class CastExpression extends Expression {

    public PrimitiveDataType target;
    public Expression Operand;

    public CastExpression(Expression operand, PrimitiveDataType targetType)
    {
        target = targetType;
        Operand = operand;
    }

    @Override
    public String toString()
    {
        return "(" + target.name().toLowerCase() + ")" + Operand.toString();
    }

    /* To generate bytecode, generate the operand, then...
```



## The Fearn Programming Language

```
* -> If targeting int, call the appropriate method based on operands
*      expression_type, casting to Integer after
* -> If targeting float, follow a similar procedure
* -> If targeting str, call the Obj2Str method of the FearnRuntime
*      (remember, generating the Operand's bytecode leaves an object
*      of its value at runtime on top of the JVM's operand stack)
* -> If targeting bool, call the Obj2B method of the FearnRuntime
*/
@Override
public void GenerateBytecode(MethodVisitor mv) {

    Operand.GenerateBytecode(mv);

    switch (target) {
        case INT:
            if (Operand.expression_type.equals(new
PrimitiveSpecifier(PrimitiveDataType.FLOAT)))
            {
                mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Double", "intValue", "()I",
false);
            }

            else if (Operand.expression_type.equals(new
PrimitiveSpecifier(PrimitiveDataType.STR)))
            {
                mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf",
"(Ljava/lang/String;)Ljava/lang/Integer;", false);
                return;
            }

            else if (Operand.expression_type.equals(new
PrimitiveSpecifier(PrimitiveDataType.BOOL)))
            {
                mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Boolean", "booleanValue",
"()Z", false);
                mv.visitInsn(ICONST_0);
                mv.visitMethodInsn(INVOKESTATIC, "java/lang/Boolean", "compare", "(ZZ)I",
false);
            } else { return; }

            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf",
"(I)Ljava/lang/Integer;", false);
            return;

        case FLOAT:
```

# The Fearn Programming Language

```
        if (Operand.expression_type.equals(new
PrimitiveSpecifier(PrimitiveDataType.INT)))
        {
            mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Double", "intValue", "()I",
false);

            mv.visitInsn(I2D);
        }

        else if (Operand.expression_type.equals(new
PrimitiveSpecifier(PrimitiveDataType.STR)))
        {
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Double", "valueOf",
"(Ljava/lang/String;)Ljava/lang/Double;", false);
            return;
        } else { return; }

        mv.visitMethodInsn(INVOKESTATIC, "java/lang/Double", "valueOf",
"(D)Ljava/lang/Double;", false);
        return;

        case STR:
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "Obj2Str",
"(Ljava/lang/Object;)Ljava/lang/String;", false);
            return;
        case BOOL:
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "Obj2B",
"(Ljava/lang/Object;)Ljava/lang/Boolean;", false);
            return;
    }
}
```

```
/* To validate, validate the operand, and check that, for the target type, the
 * operand.expression_type is one where that operation is valid.
 */
```

```
@Override
```

```
public TypeSpecifier validate(SymbolTable symTable) {
    // For Each target type, ensure the operand can be cast
    TypeSpecifier op_type = Operand.validate(symTable);
```

# The Fearn Programming Language

```
switch (target) {
    case PrimitiveDataType.INT: // You can cast strings, floats, and bools
(Boolean.compare) to integers
        if (
            op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT    ))
            || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.FLOAT ))
            || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.STR   ))
            || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.BOOL  ))
        ) {
            expression_type = new PrimitiveSpecifier(PrimitiveDataType.INT);
        } else {
            Reporter.ReportErrorAndExit("Cannot perform cast from " +
op_type.toString() + " to int.", this);
        }
        break;

    case PrimitiveDataType.FLOAT: // You can cast strings and ints to floats
        if (
            op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT    ))
            || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.FLOAT ))
            || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.STR   ))
        ) {
            expression_type = new PrimitiveSpecifier(PrimitiveDataType.FLOAT);
        } else {
            Reporter.ReportErrorAndExit("Cannot perform cast from " +
op_type.toString() + " to float.", this);
        }
        break;

    case PrimitiveDataType.STR: // You can cast anything, including arrays and structs,
to strings
        expression_type = new PrimitiveSpecifier(PrimitiveDataType.STR);
        break;

    case PrimitiveDataType.BOOL: // You can cast strings, ints, and floats to bools
        if (
            op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT    ))
            || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.STR   ))
            || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.FLOAT ))
            || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.BOOL  ))
        ) {
            expression_type = new PrimitiveSpecifier(PrimitiveDataType.BOOL);
        } else {
            Reporter.ReportErrorAndExit("Cannot perform cast from " +
op_type.toString() + " to bool.", this);
        }
}
```

## The Fearn Programming Language

```
        }  
        break;  
    default: break;  
}  
return expression_type;  
}  
}
```

# The Fearn Programming Language

## 1.6: Must be able to create fixed-length arrays

FearnLang does allow the developer to create fixed-length arrays with Java-style syntax, for example:

- `new int[5];` // An array of 5 ints
- `new int[] {1, 2, 7, 9, -13};` // An array with an initial body).

The syntax for these are included in the `array_init` and `array_body` grammar rules, and their representation in the AST relies on two classes: `ArrayInitExpression` and `ArrayBody` (both included below). Moreover, the `IndexExpression` class is used to represent indexed references to the elements of an array (as well as the characters of a string). The logic to allow data at specific indexes to be modified is present in the `AssignExpression` class (Appendix C).

### ArrayInitExpression.java

```
package ast.expression;

import java.util.ArrayList;

import org.objectweb.asm.MethodVisitor;
import static org.objectweb.asm.Opcodes.*;

import ast.type.ArrayBodySpecifier;
import ast.type.ArraySpecifier;
import ast.type.PrimitiveSpecifier.PrimitiveDataType;
import ast.type.PrimitiveSpecifier;
import ast.type.TypeSpecifier;
import semantics.table.SymbolTable;
import util.Reporter;

/* ArrayInitExpression.java
 *
 * Represents an Array Initialisation in the AST (e.g. new int[5]).
 *
 * Fields:
 * -> type : TypeSpecifier for element types
 * -> dimensions: The expressions for the array dimensions (can be null if body provided)
 * -> init_body: The body used to initialise the array (can be null if dimensions provided)
 */

public class ArrayInitExpression extends Expression {
    public TypeSpecifier type;
    public ArrayList<Expression> dimensions;
```

## The Fearn Programming Language

```
public ArrayBody init_body;

public ArrayInitExpression(TypeSpecifier t, ArrayList<Expression> dims, ArrayBody ele)
{
    type = t;
    dimensions = dims;
    init_body = ele;
}

@Override
public String toString()
{
    String s = type.toString();
    if (dimensions.get(0) == null)
    {
        s += "[".repeat(dimensions.size());
    } else {
        for (Expression dim : dimensions)
        {
            s += '[' + dim.toString() + ']';
        }
    }

    if (init_body != null) s += init_body.toString();

    return s;
}

/* To generate bytecode for an Array Initialisation, the method...
 * 1) If a body has been provided, just generate that
 * 2) Otherwise, Generate the bytecode for each dimension, casting the values to
 *    primitive 'I'
 * 3) If the array is multidimensional, generate an array descriptor using
 *    SymbolTable, and use visitMultiANewArray, with the number of dimensions
 * 4) Otherwise, Use a new array, with the descriptor for an element
 */

@Override
public void GenerateBytecode(MethodVisitor mv) {

    // Just Generate the Body, if provided
    if (init_body != null)
    {
```

## The Fearn Programming Language

```
        init_body.GenerateBytecode(mv);
        return;
    }

    // Otherwise
    else
    {

        for (Expression dim : dimensions)
        {
            dim.GenerateBytecode(mv);
            mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Integer", "intValue", "()I",
false);
        }

        if (dimensions.size() > 1)
        {
            String desc = SymbolTable.GenBasicDescriptor(new ArraySpecifier(type,
dimensions.size()));
            mv.visitMultiANewArrayInsn(desc, dimensions.size());
            return;
        } else {
            String desc = SymbolTable.GenBasicDescriptor(type);
            desc = desc.substring(1, desc.length() - 1 );
            mv.visitTypeInsn(ANEWARRAY, desc);
        }
    }
}

/* To validate an Array Initialisation, the method...
 * 1) If a body has been provided...
 *     a) Validate the body (this ensures the types of elements in the body
 *         is consistent)
 *     b) Check the type of elements in the body matches that of the array
 *         -> This is done by repeatedly accessing the element type of the body
 *            specifier, repeating for each dimension (e.g. for a 2D array, this
 *            runs twice), until the type of the actual elements is reached
 *         -> Raise error otherwise
 *     c) Check the dimensions are the same (a 3D body is used for a 3D array)
 *     d) Set expression_type to an ArraySpecifier
 * 2) Otherwise...
 *     a) For each dimension, validate that they're of type int (raise error
 *         otherwise)
 *     b) Set expression_type to an ArraySpecifier
 * 3) Return expression_type
```

## The Fearn Programming Language

```
*/
@Override
public TypeSpecifier validate(SymbolTable symTable) {

    if (init_body != null)
    {
        // Check the Elements are of the same type
        ArrayBodySpecifier bodySpecifier =
(ArrayBodySpecifier)init_body.validate(symTable);

        TypeSpecifier typeOfElements = bodySpecifier;
        for (int i = 0; i < bodySpecifier.dimensions.size(); i++)
        {
            typeOfElements = ((ArrayBodySpecifier)typeOfElements).element_type;
        }

        if (!typeOfElements.equals(type))
        {
            Reporter.ReportErrorAndExit("Type of Elements in Array Body don't match the
element type of the Array.", this);
        }

        if (bodySpecifier.dimensions.size() != dimensions.size())
        {
            Reporter.ReportErrorAndExit("Dimensions of Array Body don't match dimensions of
Array Initialisation.", this);
        }

        expression_type = new ArraySpecifier(type, bodySpecifier.dimensions.size());
    } else {
        for (Expression e: dimensions)
        {
            TypeSpecifier dim_type = e.validate(symTable);
            if (!dim_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT)))
            {
                Reporter.ReportErrorAndExit("Dimensions of arrays must be of type int.",
this);
            }
        }
        expression_type = new ArraySpecifier(type, dimensions.size());
    }
    // Return ArraySpecifier
    return expression_type;
}
}
```



# The Fearn Programming Language

## ArrayBody.java

```
package ast.expression;

import org.objectweb.asm.MethodVisitor;
import static org.objectweb.asm.Opcodes.*;

import java.util.ArrayList;

import ast.type.ArrayBodySpecifier;
import ast.type.TypeSpecifier;
import semantics.table.SymbolTable;
import util.Reporter;

/* ArrayBody.java
 *
 * Represents an ArrayBody in the AST (e.g. {1, 2, 3}).
 *
 * Fields:
 * -> elements : The body's elements, modelled as a list of expressions.
 *
 */

public class ArrayBody extends Expression {

    public ArrayList<Expression> elements;

    public ArrayBody(ArrayList<Expression> ele)
    {
        elements = ele;
    }

    @Override
    public String toString()
    {
        return elements.toString().replace("[", "{").replace("]", "}");
    }

    /* To generate bytecode for an ArrayBody, the method...
     * 1) Gets a descriptor of the body's elements from the SymbolTable
     * 2) If the body is 1-D, the "L" and ";" ate the front and end of the descriptor
     */
}
```

## The Fearn Programming Language

```
*      are removed, to make it a class name
* 3) Push the array length to the stack, and use ANEWARRAY to create an empty
*     array, of that size, on the top of the stack
* 4) For each element in the body ...
*     4.1) Duplicate the array (as the AASTORE instruction will pop the array
*           from the stack, and we still have more element to assign)
*     4.2) Push the index to store the element (starting at 0 and incrementing
*           with each element) to the stack
*     4.3) Generate the expression to store at that index (GenerateBytecode()
*           on expressions leaves its value on top of the stack)
*     4.4) Use AASTORE to take the expression value, index, and array, and store
*           the element - at the index - in the array
*/
@Override
public void GenerateBytecode(MethodVisitor mv) {

    String desc = SymbolTable.GenBasicDescriptor(elements.get(0).expression_type);

    if (elements.get(0).getClass() != ArrayBody.class) // Array is 1-D
    {
        desc = desc.substring(1, desc.length() - 1);
        // This would transform "Ljava\lang\Integer;" to "java\lang\Integer" (a class name)
    }

    mv.visitIntInsn(SIPUSH, elements.size());
    mv.visitTypeInsn(ANEWARRAY, desc);

    int i = 0;
    for (Expression e : elements)
    {
        mv.visitInsn(DUP);
        mv.visitIntInsn(SIPUSH, i++);
        e.GenerateBytecode(mv);
        mv.visitInsn(AASTORE);
    }
}

/* To validate an ArrayBody, the method...
* 1) Gets the TypeSpecifier of the first element
* 2) Check the rest of the elements have the same TypeSpecifier
*    -> Raise Error otherwise
* 3) If the body contains other array bodies (N-D array), add the dimensions of the
child
*     bodies to the end of this body's dimensions
```

## The Fearn Programming Language

```
* 4) Set expression_type to an ArrayBodySpecifier, and return it
*/
@Override
public TypeSpecifier validate(SymbolTable symTable) {
    TypeSpecifier element_type = elements.get(0).validate(symTable);
    for (Expression e : elements.subList(1, elements.size() ))
    {
        TypeSpecifier e_type = e.validate(symTable);
        if (!element_type.equals(e_type)) Reporter.ReportErrorAndExit(
            "ArrayBody has inconsistent element type.",
            this
        );
    }
    ArrayList<Integer> dimensions = new ArrayList<Integer>();
    dimensions.add(elements.size());
    if (element_type.getClass() == ArrayBodySpecifier.class)
    {
        dimensions.addAll(((ArrayBodySpecifier)element_type).dimensions);
    }
    expression_type = new ArrayBodySpecifier(element_type, dimensions);
    return expression_type;
}
```

### IndexExpression.java

```
package ast.expression;

import org.objectweb.asm.MethodVisitor;
import static org.objectweb.asm.Opcodes.*;

import ast.type.ArraySpecifier;
import ast.type.PrimitiveSpecifier.PrimitiveDataType;
import ast.type.PrimitiveSpecifier;
import ast.type.TypeSpecifier;
import semantics.table.SymbolTable;
import util.Reporter;

/* IndexExpression.java
 *
 * Represents an Index Expression in the AST (e.g list[0] ).
 *
 * Fields:
 * -> sequence: An expression that is indexable (an array or string)
```

# The Fearn Programming Language

```
* -> index: An integer expression, hat represents the index to access
*/

public class IndexExpression extends Expression {

    public Expression sequence;
    public Expression index;

    public IndexExpression(Expression id, Expression i)
    {
        sequence = id;
        index = i;
    }

    @Override
    public String toString()
    {
        return sequence.toString() + '[' + index.toString() + ']';
    }

    /* To generate bytecode, first generate the sequence and index bytecode, to prepare
    * the stack.
    *
    * If the sequence is an array, use AALOAD instruction. Otherwise (sequence is a
    * string), use charAt method, then cast result to string
    *
    */
    @Override
    public void GenerateBytecode(MethodVisitor mv) {

        sequence.GenerateBytecode(mv);
        index.GenerateBytecode(mv);
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Integer", "intValue", "()I", false);

        if (sequence.expression_type instanceof ArraySpecifier)
        {
            mv.visitInsn(AALOAD);
        } else {
            mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/String", "charAt", "(I)C", false);
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/String", "valueOf",
"(C)Ljava/lang/String;", false);
        }
    }
}
```

## The Fearn Programming Language

```
/* To validate, validate the sequence and index.
 *
 * Raise an error is the sequence is not an array or string, or if the index is not
 * an int.
 *
 * Set expression_type to string if the sequence is a string. For arrays, take 1
 * dimension from the specifier of a multidimensional array. If the array is only
 * 1D, set the expression_type to the TypeSpecifier of an element.
 *
 * Return expression_type.
 */
@Override
public TypeSpecifier validate(SymbolTable symTable) {

    TypeSpecifier seq_type      = sequence.validate(symTable);
    TypeSpecifier index_type    = index.validate(symTable);

    if (seq_type.getClass() != ArraySpecifier.class && !seq_type.equals(new
PrimitiveSpecifier(PrimitiveDataType.STR)))
    {
        Reporter.ReportErrorAndExit("Can only take index of Arrays and Strings.", this);
    }
    if (!index_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT)))
    {
        Reporter.ReportErrorAndExit("Index can only be an int.", this);
    }
    // Use seq_type to set expression type
    if (seq_type.equals(new PrimitiveSpecifier(PrimitiveDataType.STR)))
    {
        expression_type = seq_type;
        return expression_type;
    }
    ArraySpecifier seq_arr_spec = (ArraySpecifier)seq_type;
    if (seq_arr_spec.dimensionCount == 1) {
        expression_type = seq_arr_spec.element_type;
    } else {
        expression_type = new ArraySpecifier(seq_arr_spec.element_type,
seq_arr_spec.dimensionCount - 1);
    }
    return expression_type;
}
```

# The Fearn Programming Language

## 1.7: Allow for the definition, and calling, of functions and procedures

A function definition is traversed by the ASTConstructor, and represented by a Function object within the AST. Functions are generated in the CodeGenerator class, which invokes a depth-first traversal of the body of the function, using the GenerateBytecode() method. The bytecode is then written into the program class for the Fearn program that contains it. Function calls are represented by the FnCallExpression class.

### Function.java

```
package ast.function;

import java.util.ArrayList;

import ast.ASTNode;
import ast.statement.CompoundStatement;
import ast.type.TypeSpecifier;
import codegen.CodeGenerator;
import semantics.table.SymbolTable;
import util.Reporter;

/* Function.java
 *
 * Represents a function in the AST
 *
 * Fields:
 * -> identifier: string name of function (used as method name in the generated
 *     program class)
 * -> parameters: A list of Parameter object, representing the values the
 *     function takes in, and their local identifier
 * -> return_type: TypeSpecifier of the data value returned by the function (is
 *     null for a void function)
 * -> is_void: A boolean flag, indicating if the function is void (returns no data)
 */

public class Function extends ASTNode {

    public String identifier;
    public ArrayList<Parameter> parameters;

    public TypeSpecifier return_type;
    public Boolean is_void;
```

## The Fearn Programming Language

```
public CompoundStatement body;

public Function(
    String id,
    ArrayList<Parameter> params,
    TypeSpecifier rt,
    Boolean _void,
    CompoundStatement bod
) {
    identifier = id;
    parameters = params;
    return_type = rt;
    is_void = _void;
    body = bod;
}

@Override public String toString()
{
    String ret_type_str = is_void ? "void" : return_type.toString();
    return String.format(
        "fn %s%s => %s {...}",
        identifier,
        parameters.toString().replace("[", "(").replace("]", ")"),
        ret_type_str
    );
}

/* No Function.GenerateBytecode() is provided, as the generation of
 * functions using the method visitor is performed by CodeGenerator,
 * as the program's Class Writer is needed to add the method to the
 * main program class (see CodeGenerator.java).
 */

public void validate(SymbolTable symbolTable) {
    /* Sets Current Return Type to the return type specifier
     * for this function. This is used during the traversal of
     * the body, as return statements must return an expression
     * the evaluates to the right type.
     *
     * The body is validated, and (assuming function is not void),
     * must include a return statement (the type of expression
     * returned is validated by the return statement itself,
```

## The Fearn Programming Language

```
        * using CurrentReturnType).
        *
        */

    CodeGenerator.CurrentReturnType = return_type;
    body.validate(symbolTable);
    if (!body.includesReturn && return_type != null)
        Reporter.ReportErrorAndExit(
            "Function " + identifier + " must include a return statement in its main body.",
            null
        );
    }
}
```

### CodeGenerator.java (*GenerateProgram*)

```
/* GenerateProgram
 *
 * Method to generate program class files
 * 1) Use a class writer to create a public, static class, representing the program
 * 2) Create a static block (<clinit>), to define the default states of global variables
 * 3) For each global variable in the program ...
 *     -> Add a public, static field to the class
 *     -> If an initialisation expression has been provided, generate the expression's
 *          bytecode (putting the value of the expression at the top of the operand stack),
 *          and put the value into the field of the program class
 * 4) Add a default constructor, which simply invokes the default object constructor
 * 5) For each function in the program ...
 *     -> Add a method representing it to the program class (uses SymbolTable to get
method
 *          descriptor), and create a FearnMethodVisitor to generate code for it
 *     -> Create a MethodNode for the function (effectively a list of bytecode
instructions)
 *     -> Set LocalSymbolTable to the symbol table for this function
 *     -> Loop through all local variable indexes, initialising them all to null
 *     -> Generate function body bytecode, writing the instructions to the MethodNode
 *     -> Eliminate Redundant Casting in the MethodNode (see CastOptimiser.java)
 *     -> Use the FearnMethodVisitor to visit every instruction in the MethodNode, adding
 *          them to the actual class method
 * 6) Write program class to .class file
 * 7) Report Success
 */
```



## The Fearn Programming Language

```
private void GenerateProgram(
    ArrayList<Function> functions,
    ArrayList<Declaration> global_declarations,
    Path finalProgramPath
) {

    // Create new class writer, to write program class
    ClassWriter classWriter = new ClassWriter(ClassWriter.COMPUTE_MAXS);

    // Class is a public, static class, with its name being the programName
    classWriter.visit(
        V19,
        ACC_PUBLIC | ACC_SUPER,
        programName,
        null,
        "java/lang/Object",
        null
    );

    ...

    // Write Functions as Methods
    for (Function function : functions)
    {
        // Create new FearnMethodVisitor (see FearnMethodVisitor.java)
        // Also, visit new public static method with class writer
        // Gets Function method descriptor from symbol table to do this
        MethodVisitor function_visitor = new FearnMethodVisitor(ASM9,
            classWriter.visitMethod(
                ACC_PUBLIC | ACC_STATIC,
                function.identifier,
                SymbolTable.GenFuncDescriptor(function.parameters, function.return_type),
                null,
                null
            )
        );

        // Create MethodNode (a derived class of a method visitor, that's
        // effectively a list of instruction nodes, that can be iterated
        // through and optimised)
        MethodNode function_node = new MethodNode(ASM9);

        // Set Local Symbol Table
        LocalSymbolTable = GlobalSymbolTable.GetFuncSymbolTable(function.identifier);
    }
}
```

# The Fearn Programming Language

```
function_node.visitCode();

// Initialise all local variables to null
// Needs to start at function.parameters.size() to prevent nullifying the values of
parameters
for (int i = function.parameters.size(); i <
LocalSymbolTable.GetAllVarTypeSpecifiers().size(); i++)
{
    // Load null value
    function_node.visitInsn(ACONST_NULL);
    // Assign value to variable at index i
    function_node.visitVarInsn(ASTORE, i);
}

// Generate body of bytecode
function.body.GenerateBytecode(function_node);

// Add a return instruction if void, and no return statement already included
if (function.is_void && !function.body.includesReturn)
function_node.visitInsn(RETURN);

// End Function Generation
function_node.visitMaxs(0, 0);
function_node.visitEnd();

// Eliminate redundant casting
CastOptimiser.EliminateRedundantCasts(function_node);

// Write all instructions to method visitor
function_node.accept(function_visitor);
}
classWriter.visitEnd();
// Write program .class file
try {
    Files.write(finalProgramPath, classWriter.toByteArray());
} catch (IOException e) {
    Reporter.ReportErrorAndExit("Program Gen Error :- " + e.toString(), null);
}
// Report Success
Reporter.ReportSuccess(
    "GENERATED Program      : "+ finalProgramPath.toAbsolutePath() + ";",
    false
);
}
```

# The Fearn Programming Language

## FnCallExpression.java

```
package ast.expression;

import static org.objectweb.asm.Opcodes.*;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import org.objectweb.asm.MethodVisitor;

import ast.type.ArraySpecifier;
import ast.type.PrimitiveSpecifier.PrimitiveDataType;
import ast.type.PrimitiveSpecifier;
import ast.type.TypeSpecifier;
import codegen.CodeGenerator;
import semantics.table.SymbolTable;
import util.Reporter;

/* FnCallExpression.java
 *
 * Represents a Function Call in the AST.
 *
 * Fields:
 * -> identifier: Function identifier
 * -> arguments: The expressions used as arguments in the function call
 */

public class FnCallExpression extends Expression {

    public String identifier;
    public ArrayList<Expression> arguments;

    // Flag used to indicate if a function is written in Universal Function Notation
    // e.g. x.myFunction()
    // It is only used in toString()
    public Boolean isUFN = false;

    // List of built-in functions
    // These aren't added to the SymbolTable simply because they take
    // multiple data types as input (e.g. length() take 1 argument,
    // which is either an array or a string)
    static List<String> builtins = Arrays.asList("length", "slice");
```

## The Fearn Programming Language

```
public FnCallExpression(String fn_name, ArrayList<Expression> args)
{
    identifier = fn_name;
    arguments = args;
}

@Override
public String toString()
{
    if (isUFN)
    {
        String str_rep = arguments.get(0).toString() + "." + identifier;
        arguments.remove(0);
        str_rep += arguments.toString()
            .replace("[", "(")
            .replace("]", ")");
        return str_rep;
    }
    else
        return identifier + "(" + arguments.toString().substring(1,
arguments.toString().length() - 1) + ")";
}

/* To generate bytecode, generate all arguments.
 *
 * Then, if the function call is one of the built-ins, set the descriptor to the
 * correct value, and call the method, using INVOKESTATIC and the FearnRuntime.
 * -> If slice is used, returned array must be cast back, using the array's type
 * descriptor
 *
 * Otherwise, use INVOKESTATIC, using the identifier, descriptor, and owner from
 * the SymbolTable.
 */
@Override
public void GenerateBytecode(MethodVisitor mv) {

    // Gen args, then INVOKESTATIC
    for (Expression arg : arguments) arg.GenerateBytecode(mv);

    String desc;

    switch (identifier){
```

## The Fearn Programming Language

```
        case "length":
            desc = "(Ljava/lang/Object;)Ljava/lang/Integer;";
            break;

        case "slice":
            String t = null;

            if (arguments.get(0).expression_type.equals(new
PrimitiveSpecifier(PrimitiveDataType.STR))) t = "Ljava/lang/String;";
            else t = "[Ljava/lang/Object;";

            desc = String.format( "(%sLjava/lang/Integer;Ljava/lang/Integer;)s", t, t );
            break;

        default:
            desc = CodeGenerator.GlobalSymbolTable.GetGlobalFuncDescriptor(identifier);
            break;
    }

    if (builtins.contains(identifier))
    {
        mv.visitMethodInsn(
            INVOKESTATIC,
            "FearnRuntime",
            identifier,
            desc,
            false
        );

        if (identifier.equals("slice") && desc.contains("Object"))
        {
            mv.visitTypeInsn(CHECKCAST,
SymbolTable.GenBasicDescriptor(arguments.get(0).expression_type));
        }
        return;
    }

    mv.visitMethodInsn(
        INVOKESTATIC,
        CodeGenerator.GlobalSymbolTable.GetOwner(identifier, true),
        identifier,
        desc,
        false
    );
}
```

## The Fearn Programming Language

```
}

/* To validate...
 * -> If the function is one of the built-ins, check the number of arguments and
 *     their types, raising an error if necessary
 * -> Otherwise, using the parameters from the SymbolTable to validate the
 *     arguments (number and types)
 * -> Set expression_type to return type of function, and return this
 *
 */
@Override
public TypeSpecifier validate(SymbolTable symTable) {

    // Check the function's signature (Parameters from Symbol Table) against to types of
    each argument

    ArrayList<TypeSpecifier> arg_types = new ArrayList<TypeSpecifier>();

    switch (identifier) {

        // length(<str|arr>)
        case "length":
            if (arguments.size() != 1)
            {
                Reporter.ReportErrorAndExit("Wrong number of arguments for " + identifier +
" , expected 1.", this);
            }

            if (!(
                arguments.get(0).validate(symTable).equals(new
PrimitiveSpecifier(PrimitiveDataType.STR)) ||
                arguments.get(0).validate(symTable) instanceof ArraySpecifier
            ))
            {
                Reporter.ReportErrorAndExit("Wrong argument data type, expected string or
array.", this);
            }

            expression_type = new PrimitiveSpecifier(PrimitiveDataType.INT);
            return expression_type;
    }
}
```

## The Fearn Programming Language

```
// slice(<str|arr>, int, int);
case "slice":
    if (arguments.size() != 3)
    {
        Reporter.ReportErrorAndExit("Wrong number of arguments for " + identifier +
" , expected 3.", this);
    }

    if (!(
        arguments.get(0).validate(symTable).equals(new
PrimitiveSpecifier(PrimitiveDataType.STR)) ||
        arguments.get(0).validate(symTable) instanceof ArraySpecifier
    ))
    {
        Reporter.ReportErrorAndExit("Wrong argument data type, expected string or
array.", this);
    }

    if (!(
        arguments.get(1).validate(symTable).equals(new
PrimitiveSpecifier(PrimitiveDataType.INT)) ||
        arguments.get(2).validate(symTable).equals(new
PrimitiveSpecifier(PrimitiveDataType.INT))
    ))
    {
        Reporter.ReportErrorAndExit("Wrong argument data type, expected int.",
this);
    }

    expression_type = arguments.get(0).expression_type;
    return expression_type;
default:
    break;
}

ArrayList<TypeSpecifier> param_types =
CodeGenerator.GlobalSymbolTable.GetFuncParameterSpecifiers(identifier);

for (Expression arg : arguments)
{
    arg_types.add(arg.validate(symTable));
}

if (arguments.size() != param_types.size())
{
```

## The Fearn Programming Language

```
        Reporter.ReportErrorAndExit(
            "Wrong number of arguments for " + identifier + " , expected " +
param_types.size(),
            this
        );
    }

    for (int i = 0; i < param_types.size(); i++)
    {
        if (!param_types.get(i).equals(arg_types.get(i)))
        {
            Reporter.ReportErrorAndExit(
                arguments.get(i).toString() + " is of the wrong type, expected " +
param_types.get(i).toString(),
                this
            );
        }
    }

    expression_type = CodeGenerator.GlobalSymbolTable.GetTypeSpecifier(identifier, true);
    return expression_type;
}
}
```



# The Fearn Programming Language

## 1.8: Allow for the definition of user-defined data types

Fearn supports structs that can be instantiated with the syntax...

```
let instance : $MyStruct = new myStruct(arg1, arg2);
```

The definitions are made using the struct keyword, followed by a body of multiple declarations that represent the struct's attributes, such as ...

```
struct MyStruct {  
    let property1 : int;  
    let property2 : str[];  
}
```

The values of the attributes can be accessed and modified using a dot expression, such as `instance.property1`.

These are handled by the `Struct`, `StructInitExpression`, and `StructAttrExpression`. The code to generate struct classes (the java classes that represent user-defined structs) is in the `CodeGenerator` class. The logic for assigning to a struct attribute is found in `AssignmentExpression` (Appendix C).

### Struct.java

```
package ast;  
  
import java.util.ArrayList;  
  
import semantics.table.SymbolTable;  
import util.Reporter;  
  
/* Struct.java  
 *  
 * Represents struct definitions in the AST.  
 * -> It contains the struct's identifier, and attributes (as declarations)  
 * -> It's validate method checks the the user hasn't attempted to initialise  
 *     an attribute to a default value  
 * -> This is raised as invalid, simply because, when a struct is  
 *     instantiated, the user must define the initial value of every  
 *     attribute of that instance  
 */  
  
public class Struct extends ASTNode {  
    public String identifier;  
    public ArrayList<Declaration> declarations;
```

## The Fearn Programming Language

```
public Struct(String id, ArrayList<Declaration> decl)
{
    identifier = id;
    declarations = decl;
}

@Override
public String toString()
{
    return String.format("struct %s {...}", identifier);
}

public void validate(SymbolTable symbolTable) {
    for (Declaration decl : declarations)
    {
        if (decl.init_expression != null)
        {
            Reporter.ReportErrorAndExit(decl.toString() + ": Cannot assign default values
to struct attributes.", null);
        }
    }
}
}
```

### StructInitExpression.java

```
package ast.expression;

import java.util.ArrayList;

import org.objectweb.asm.MethodVisitor;
import static org.objectweb.asm.Opcodes.*;

import ast.type.StructInstanceSpecifier;
import ast.type.TypeSpecifier;
import codegen.CodeGenerator;
import semantics.table.SymbolTable;
import util.Reporter;

/* StructInitExpression.java
 *
 * This represents an initialisation of a struct instance
 *
 * Fields:
```

# The Fearn Programming Language

```
* -> name: The name of the struct being instantiated
* -> arguments: The expressions used for the struct's initial state
*/

public class StructInitExpression extends Expression {

    public String name;
    public ArrayList<Expression> arguments;

    public StructInitExpression(String n, ArrayList<Expression> args)
    {
        name = n;
        arguments = args;
    }

    @Override
    public String toString()
    {
        return "new " + name + "(" + arguments.toString().substring(1,
arguments.toString().length() - 1) + ")";
    }

    /* To generate bytecode, the NEW instruction is used to create an instance of the
    * struct class ($name). This instance is duplicated, and the arguments are generated,
    * leaving their values on top of the stack. Finally, the struct class's constructor is
    * invoked, to set the state of the object. This requires the descriptor from the Global
    * Symbol Table.
    */
    @Override
    public void GenerateBytecode(MethodVisitor mv) {
        mv.visitTypeInsn(NEW, "$" + name);
        mv.visitInsn(DUP);
        for (Expression arg : arguments)
        {
            arg.GenerateBytecode(mv);
        }
        mv.visitMethodInsn(
            INVOKESPECIAL,
            "$" + name,
            "<init>",
            CodeGenerator.GlobalSymbolTable.GetGlobalStructDescriptor(name),
            false
        );
    }
}
```

## The Fearn Programming Language

```
/* To validate, the arguments are validated, to ensure they are the right type,
 * and that there are the correct number. If so, a StructInstanceSpecifier is set
 * to expression_type and returned.
 */
@Override
public TypeSpecifier validate(SymbolTable symTable) {

    // Checks args, then return type of struct

    ArrayList<TypeSpecifier> attr_types =
CodeGenerator.GlobalSymbolTable.GetStructAttributeSpecifiers(name);

    if (attr_types.size() != arguments.size())
    {
        Reporter.ReportErrorAndExit("Wrong number of arguments, expected " +
attr_types.size(), this);
    }

    for (int i = 0; i < attr_types.size(); i++)
    {
        if (!arguments.get(i).validate(symTable).equals(attr_types.get(i)))
        {
            Reporter.ReportErrorAndExit("Wrong argument type for " +
arguments.get(i).toString() + ", expected " + attr_types.get(i).toString(), this);
        }
    }

    expression_type = new StructInstanceSpecifier(name);

    return expression_type;
}
}
```

### StructAttrExpression.java

```
package ast.expression;

import org.objectweb.asm.MethodVisitor;
import static org.objectweb.asm.Opcodes.*;

import ast.type.StructInstanceSpecifier;
import ast.type.TypeSpecifier;
```

## The Fearn Programming Language

```
import codegen.CodeGenerator;
import semantics.table.SymbolTable;
import util.Reporter;

/* StructAttrExpression.java
 *
 * This represents an access to a named attribute of a struct instance.
 *
 * Fields:
 * -> instance: Expression that resolves to an instance of a struct
 * -> attribute: The string name of the attribute being accessed
 * -> struct_name: The name of the struct, that instance is an instance of
 * -> attr_descriptor: The JVM descriptor of the attribute's type
 */

public class StructAttrExpression extends Expression {

    public Expression instance;
    public String attribute;

    protected String struct_name;
    protected String attr_descriptor;

    public StructAttrExpression(Expression n, String attr)
    {
        instance = n;
        attribute = attr;
    }

    @Override
    public String toString()
    {
        return instance.toString() + "." + attribute.toString();
    }

    /* To generate bytecode, simply generate the instance (putting it on top of the stack),
     * then use the GETFIELD instruction, with the name of the struct class ($name), and
the
     * attribute's identifier and descriptor.
     */

    @Override
    public void GenerateBytecode(MethodVisitor mv) {
```

## The Fearn Programming Language

```
// Gen instance, then GETFIELD
instance.GenerateBytecode(mv);

mv.visitFieldInsn(
    GETFIELD,
    "$" + struct_name,
    attribute,
    attr_descriptor
);
}

/* To validate,
 * -> Get the instance type, and ensure it is a struct
 * -> Get the struct's Symbol Table from the Global Symbol Table
 * -> Verify the attribute exists in the table
 * -> Set attr_descriptor, using the struct's Symbol Table
 * -> Set expression_type to the TypeSpecifier associated with that attribute, and
 *     return this
 */
@Override
public TypeSpecifier validate(SymbolTable symTable) {
    // Get from StructRow in SymbolTable
    TypeSpecifier inst_type = instance.validate(symTable);
    if (inst_type.getClass() != StructInstanceSpecifier.class)
    {
        Reporter.ReportErrorAndExit(inst_type.toString() + " is not a struct.", this);
    }
    struct_name = ((StructInstanceSpecifier)inst_type).name;
    SymbolTable structTable =
CodeGenerator.GlobalSymbolTable.GetStructSymbolTable(struct_name);
    if (!structTable.Contains(attribute))
    {
        Reporter.ReportErrorAndExit(struct_name + " has no attribute " + attribute,
this);
    }
    attr_descriptor = structTable.GetVarDescriptor(attribute);
    expression_type = structTable.GetTypeSpecifier(attribute, false);
    return expression_type;
}
}
```

## The Fearn Programming Language

### CodeGenerator.java (*GenerateStructs*)

```
/* GenerateStructs
 * Method to generate struct class files, for each struct in a program.
 * The method iterates thorough each struct (in a provided ArrayList) ...
 * 1) A new class writer object to created, to write the struct class
 * 2) A method visitor for the constructor (cv) is created, to write
 *    the constructor method. This is created using the descriptor
 *    from the global symbol table
 * 3) The constructor invokes Java's default object constructor first
 * 4) For each attribute (declaration) in the struct, add a public field
 *    to the struct class, and assigns the correct parameter of the
 *    constructor to that property
 * 5) Add RETURN instruction to end of constructor
 * 6) Output byte array, representing struct class, into a new .class file
 *    in the buildPath directory
 * 7) Report successful generation of struct class file
 */
private void GenerateStructs(ArrayList<Struct> structs)
{
    structs.forEach(
        (struct) -> {
            ClassWriter classWriter = new ClassWriter(ClassWriter.COMPUTE_MAXS);
            classWriter.visit(
                V19,
                // Defines access
                ACC_PUBLIC | ACC_SUPER,
                // Define class name
                // (Fearn struct class names have a $ prefix, to distinguish them from
program files)
                "$"+struct.identifier,
                null,
                // All Java objects derive from the base Object class
                "java/lang/Object",
                null
            );
            MethodVisitor cv = classWriter.visitMethod(
                ACC_PUBLIC,
                "<init>", // `<init>` indicates the constructor
                GlobalSymbolTable.GetGlobalStructDescriptor(struct.identifier),
                null,
                null
            );
        }
    );
}
```

## The Fearn Programming Language

```
cv.visitCode();
// Invoke default constructor
cv.visitVarInsn(ALOAD, 0);
cv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Object", "<init>", "()V",
false);

Integer i = 0;
for (Declaration decl : struct.declarations)
{
    // Create public field for attribute
    String descriptor = SymbolTable.GenBasicDescriptor(decl.type);
    classWriter.visitField(
        ACC_PUBLIC,
        decl.identifier,
        descriptor,
        null,
        null
    );

    // Generate Constructor Instructions that take in an argument,
    // and load it into the attribute, specified by decl
    // Load 'this' (the struct instance itself) to operand stack
    cv.visitVarInsn(ALOAD, 0);
    // Load argument to operand stack
    cv.visitVarInsn(ALOAD, ++i);
    // Assign argument to Field
    cv.visitFieldInsn(
        PUTFIELD,
        "$"+struct.identifier,
        decl.identifier,
        SymbolTable.GenBasicDescriptor(decl.type)
    );
}

cv.visitInsn(RETURN);
cv.visitMaxs(0, 0);
cv.visitEnd();

classWriter.visitEnd();

Path destination = Paths.get(buildPath.toString(),
String.format("%s.class", struct.identifier));
```



## The Fearn Programming Language

```
        try {
            Files.write(destination, classWriter.toByteArray());
        } catch (IOException e) {
            Reporter.ReportErrorAndExit("Struct Gen Error :- " + e.toString(),
null);;
        }
        Reporter.ReportSuccess(
            "GENERATED Struct File : " + destination.getAbsolutePath() + ";",
            false
        );
    }
};
}
```

# The Fearn Programming Language

## 1.9: Must allow for the use of Boolean Logic

Fearn supports the use of boolean operations as unary and binary expressions. As I've already included `BinaryExpression` in this section, I won't include it below.

The logical not operation is an example of a unary expression, represented in the `UnaryExpression` class (Appendix C). Since these are all rather trivial, I've elected not to include any code for this objective - but rather to simply point out their presence in the two classes mentioned above - both of which feature in Appendix C - Full Source Code.

## 1.10: Allow importing of global Fearn variables, structs, and functions

Importing from other files can be done using an import statement in the form...

```
import from "path/to/fearn/program"
```

When these are traversed in the `ASTConstructor`, it calls the method `ImportCompiler.Compile()`. This follows the same process as the main compiler process, for reading the imported file and compiling it into .class files in the build directory. The difference is that it also returns the `SymbolTable` for the imported class, which then has its rows loaded into the `SymbolTable` for the importing program; this means that using imported elements becomes valid, as they are found in the `SymbolTable` (*they now exist in the program*). Since rows have owners (reflecting the name of the program class they belong to), when these global elements are used later in the program, the `SymbolTable` returns the correct owner class, ensuring that the generated bytecode is referencing the right classes - allowing the program to work properly.

### ASTConstructor.java (*visitModule\_import*)

```
/* IMPORTS
 *
 * An Import Compiler is used to compile other Fearn programs,
 * imported by the current program, and returns their symbol table,
 * the rows of which are added to the current global symbol table.
 *
 * If an identifier is used, the import is from a standard library
 * module (e.g. io). The Import Compiler will construct a symbol table
 * for the functions contained within, and add them to the global symbol
 * table, so they can be used anywhere within the program.
 *
 */
@Override
public ASTNode visitModule_import(FearnGrammarParser.Module_importContext ctx)
{
    ImportCompiler comp = new ImportCompiler();
```

## The Fearn Programming Language

```
if (ctx.IDENTIFIER() == null)
{
    symbolTableStack.peek().addRowsFromTable(
        comp.Compile(ctx.STR_LIT().toString())
    );
} else {
    symbolTableStack.peek().addRowsFromTable(
        comp.GetStdLib(ctx.IDENTIFIER().toString())
    );
}
// All new symbols from the import are added to the
// symbol stack
for (Row row : symbolTableStack.peek().GetAllRows())
{
    if (row instanceof VariableRow)
    {
        symbolAnalysisStack.push(row.identifier);
    } else if (row instanceof StructRow)
    {
        for (Row var_row : ((StructRow)row).localSymbolTable.GetAllRows())
            symbolAnalysisStack.push(var_row.identifier);
    }
}
return null;
}
```

### ImportCompiler.java (*Compile*)

```
// This performs an identical process as Compile in main.java, with the
// difference of also returning the symbol table.
public SymbolTable Compile(String path) {

    path = CodeGenerator.buildPath.getParent().resolve(path.replaceAll("(\\'|\\")",
    ""))).toString();

    CharStream input = null;

    try {
        input = CharStreams.fromStream(new FileInputStream(path));
    } catch (Exception e) {
        Reporter.ReportErrorAndExit("IMPORTED FILE " + path + " NOT FOUND", null);
    }
}
```

# The Fearn Programming Language

```
if (path.endsWith("FearnRuntime.fearn") )
    Reporter.ReportErrorAndExit("FILENAME FearnRuntime.fearn IS FORBIDDEN.", null);

CodeGenerator cg = new CodeGenerator();

cg.SetProgramName(path);

CodeGenerator.GeneratorStack.push(cg);

FearnGrammarLexer lexer = new FearnGrammarLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
FearnGrammarParser parser = new FearnGrammarParser(tokens);

parser.removeErrorListeners();
parser.addErrorListener(new FearnErrorListener());

ParseTree parseTree = parser.program();

ASTConstructor astConstructor = new ASTConstructor();
ASTNode AST = astConstructor.visit(parseTree);

Program root = (Program)AST;
SymbolTable symTable = astConstructor.symbolTableStack.pop();

CodeGenerator.GlobalSymbolTable = symTable;

// Perform Type Analysis
root.validate(symTable);

cg.Generate(root, symTable);

CodeGenerator.GeneratorStack.pop();

return CodeGenerator.GlobalSymbolTable;
}
```

## SymbolTable.java (*addRow* and *addRowsFromTable*)

```
public void addRow(Row new_row)
{
    // Add a Single Row object, checking there's no clash
    // with rows already in the table
}
```

## The Fearn Programming Language

```
// Raise error if two symbols (of the same type) in the same
// scope (table) have the same identifier

for (Row r : Rows)
{
    if (
        new_row.getClass() == r.getClass()
        && r.identifier.equals(new_row.identifier)
    ) {
        Reporter.ReportErrorAndExit(
            "Symbol " + new_row.identifier + " can only exist once within scope.",
            null
        );
    }
}

// Set owner
// The owner represents the generated class the row belongs to
// E.g. The owner of a function, defined in lib.test, will become
// a method in the 'lib' class, and so its owner is 'lib'

new_row.owner = CodeGenerator.GeneratorStack.peek().programName;

Rows.add(new_row);
}

// Add Rows from a Symbol Table (used to add rows symbols
// from imported files/modules)
public void addRowsFromTable(SymbolTable table) {
    for (Row r : table.GetAllRows()) Rows.add(r);
}
```

# The Fearn Programming Language

## 2.1: Allows the user to call it from the command line

The FearnC command is implemented using a Windows Command (.cmd) file. This calls the compiler, which is packaged within FearnC.jar. The handling of the file argument, and orchestration of the compilation of the program, is done by the principal FearnC class.

### FearnC.cmd

```
@echo off
set jarPath=%~dp0\FearnC.jar
java -jar --enable-preview "%jarPath%" %1
```

### main.java

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

import parser.*;
import parser.gen.*;
import semantics.table.SymbolTable;

import java.io.FileInputStream;
import java.nio.file.Path;
import java.nio.file.Paths;

import util.*;
import ast.ASTNode;
import ast.Program;
import codegen.*;

/*
 * Main.java
 *
 * This is the compiler's main class, from which compilation begins.
 */

class FearnC
{
    public static FearnGrammarLexer lexer;
    public static CommonTokenStream tokens;
    public static FearnGrammarParser parser;
```

# The Fearn Programming Language

```
static CodeGenerator cg = new CodeGenerator();
static String sourceFileArgument;

/* The main function is the first to be called.
 *
 * The function...
 * 1) Adds the initial CodeGenerator (used to compile the script the
 *    user first passes) to the GeneratorStack
 * 2) Raising an error if a source file has not been provided, storing its
 *    path otherwise
 * 3) Sets the global build path (a static property of the
 *    CodeGenerator class - as all files must be generated in the
 *    same place), and the program name of the initial script
 * 4) Calls the Compile method
 * 5) Assuming no errors have been raised during Compilation, prints a
 *    success message, including a command-line instruction to run the
 *    compiled program
 */

public static void main(String[] args)
{
    // Add initial code generator to stack
    CodeGenerator.GeneratorStack.push(cg);
    // Raise error if no source file has been passed
    if ( args.length == 0 ) {
        Reporter.ReportErrorAndExit("NO SOURCE FILE", null);
    }
    // Set Build Path and Program name for generator

    // Build Path describes where to put all generated class file, and
    // is the same for all generators
    sourceFileArgument = args[0];
    cg.SetBuildPath(sourceFileArgument);
    cg.SetProgramName(sourceFileArgument);

    // Compile source file
    Compile(sourceFileArgument);
    // Pop generator from stack, as it's no longer in use
    CodeGenerator.GeneratorStack.pop();
    // Print green Success Message
    Path parent = Paths.get(sourceFileArgument).getParent();
    parent = parent == null ? Paths.get("build") : parent.resolve("build");
    Reporter.ReportSuccess(
        String.format(
```

# The Fearn Programming Language

```
        "Compilation Successful! \n\t -> Run `cd %s ; FearnRun %s [args...]\` to run
Program",

        parent.toString(),
        Paths.get(sourceFileArgument).getFileName().toString().replace(".fearn", "")
    ),
    true
);
}

/* Compile
 *
 * The method responsible for conducting the compilation of the
 * file the user has passed.
 *
 * 1) Loads the source file, raising an error if not found
 * 2) Calls the ANTLR-Generated lexer/parser, to convert the program to
 *    a walkable parse tree
 * 3) Calls the AST Constructor to walk the parse tree, generating the
 *    Abstract Syntax Tree
 * 4) Retrieves the AST's root, and the program's Global Symbol Table.
 *    The Symbol Table is stored in the CodeGenerator class, to be globally
 *    accessible
 * 5) The AST is validated, to ensure it follows the rule of the language
 *    (e.g. type rules, definitions of all functions/structs are present, etc.)
 * 6) Given it is valid, calls the CodeGenerator to generate the program binary,
 *    writing it into the build directory.
 * 7) Pops the initial CodeGenerator off the stack.
 *
 */
static void Compile(String path)
{
    // Read source file
    CharStream input = null;
    try {
        input = CharStreams.fromStream(new FileInputStream(sourceFileArgument));
    } catch (Exception e) {
        Reporter.ReportErrorAndExit("FILE " + sourceFileArgument + " NOT FOUND", null);
    }
    // FearnRuntime.fearn is prohibited, as the resulting class would conflict with the
    // runtime
    if (sourceFileArgument.endsWith("FearnRuntime.fearn") )
    {
        Reporter.ReportErrorAndExit("FILENAME FearnRuntime.fearn IS FORBIDDEN.", null);
    }
    // Create lexer and parser objects
```



## The Fearn Programming Language

```
lexer = new FearnGrammarLexer(input);
tokens = new CommonTokenStream(lexer);
parser = new FearnGrammarParser(tokens);
// Add custom error listener (to stylise error messages)
parser.removeErrorListeners();
parser.addErrorListener(new FearnErrorListener());
// Parse program from the root 'program' production
ParseTree parseTree = parser.program();
// Construct AST
ASTConstructor astConstructor = new ASTConstructor();
ASTNode AST = astConstructor.visit(parseTree);
// Retrieve AST root, and Symbol Table
Program root = (Program)AST;
SymbolTable symTable = astConstructor.symbolTableStack.pop();
CodeGenerator.GlobalSymbolTable = symTable;
// Perform semantic analysis
root.validate(symTable);
// Generate program and struct class files
cg.Generate(root, symTable);
}
};
```

## The Fearn Programming Language

### 2.2: Read the file out, and parse it with into an Abstract Syntax Tree

This is done by the main process (above), the ANTLR-generated parser, and the ASTConstructor (Appendix C) - some of which has been featured in this section. Since the relevant code has already been included in the *Technical Solution* section, I won't include it again here, but I direct your attention to `main.java` and `parser.ASTConstructor` in Appendix C - Full Source Code.

### 2.3: Represent the AST as interconnected objects

This is done using the classes contained within the `ast` package. Some of these classes have already been featured in the *Technical Solution* section. Since there are so many, it would be ludicrous to directly include them here, however I again direct you to the code listing for the `ast` package in Appendix C.

### 2.4: Perform a Semantic Analysis on the AST

Semantic Analysis is performed by each node in the AST, on itself and its entire subtree. This is done with `validate()` methods, specified in each Expression and Statement class. The validation of Expressions differs from that of Statements in that expressions return a type specifier - which describes the data type that they evaluate to - which is vital to validating the nodes above it in the tree. Some examples of these will be in such classes already included in the *Technical Solution*. To include generic examples below would be asinine; to see the logic and implementation of this type validation for each type of AST Node, please see Appendix C.

- Symbol Analysis (verifying that all functions, structs, and variables referenced have actually been defined) is also done by the `SymbolTable` (raising an error if the data for an element of the program cannot be found) and the `ASTConstructor` (using a stack to ensure that all variable identifiers have been declared - in scope - before they are referenced). You can see both classes for more detail in Appendix C.

# The Fearn Programming Language

## 2.5: Optimise AST

Although optimisations aren't directly applied to the Abstract Syntax Tree, it feels worth mentioning that some optimisations are applied during code generation - to produce higher quality bytecode with less instructions and that executes faster.

- In the `CompoundStatement` class, code generation stops once a `JumpStatement` is encountered in the body. This is due to the fact that any code that comes after a jump statement (`return`, `break`, or `continue`) is dead code, that could never be reached in executing
- Bytecode for functions is optimised by the method `CastOptimiser.EliminateRedundantCasts()`. This iterates through the instructions, removing instances of a value at the top of the stack being cast from primitive to object, and back to primitive (a product of my model of having each node generate bytecode independent of context, and expressions always evaluating to leave an object on top of the stack). This optimisation increases the compile time somewhat, but reduces the time and resources needed to run compiled Fearn programs.

### **CompoundStatement.java (*GenerateBytecode*)**

```
public void GenerateBytecode(MethodVisitor mv)
{
    /* Iterates through nested nodes, generating their bytecode
     * by calling the common GenerateBytecode method
     *
     * A small optimisation is performed here: if a JumpStatement is encountered,
     * the remaining code beyond is unreachable, and so is not generated.
     *
     */
    for (ASTNode node : nested_nodes)
    {
        if (node instanceof Declaration) ((Declaration)node).GenerateBytecode(mv);
        else ((Statement)node).GenerateBytecode(mv);

        if (node instanceof JumpStatement) return;
    }
}
```

# The Fearn Programming Language

## CastOptimiser.java

```
package codegen;

import static org.objectweb.asm.Opcodes.*;
import org.objectweb.asm.tree.*;

/* CastOptimiser.java
 *
 * This class exists to eliminate redundant bytecode.
 *
 * During development, I decided to enforce the rule that
 * expressions should return the object version of their
 * values, rather than primitives, as they are (in general)
 * easier to work with - as it requires less decision cross-node
 * decision making on which instruction to use (JVM instructions
 * are often typed).
 *
 * The downside of this was the bytecode produced was often
 * inefficient, since a child node would cast its primitive return
 * value to an object, and the parent node would immediately cast
 * it back.
 *
 * My solution to this is below. The method EliminateRedundantCasts
 * iterates through the generated bytecode instructions, in the
 * MethodNode used to generate a function. It then detects when
 * two sequential casts have occurred, and eliminates both. Since
 * these instructions were casting from primitive, to object, and
 * back again, eliminating them has no effect on program functionality,
 * but does make execution faster, more efficient, and makes the binaries
 * smaller.
 */

public class CastOptimiser {

    public static void EliminateRedundantCasts(MethodNode node)
    {
        // Iterate through each instruction in function's MethodNode
        for (int i = 0; i < node.instructions.size(); i++)
        {
            AbstractInsnNode current_insn = node.instructions.get(i);

            // Check for redundant int casting (check for the descriptor
            // for casting from primitive int to Integer object)
        }
    }
}
```

## The Fearn Programming Language

```
    if (
        current_insn.getOpcode() == INVOKESTATIC
        && ((MethodInsnNode)current_insn).desc.equals("(I)Ljava/lang/Integer;")
    ) {
        // Check if next instruction is casting straight back (by checking descriptor)
        AbstractInsnNode next_insn = node.instructions.get(i + 1);
        if (
            next_insn.getOpcode() == INVOKEVIRTUAL
            && ((MethodInsnNode)next_insn).desc.equals("(I)I")
        ) {
            // Remove both instructions
            node.instructions.remove(current_insn);
            node.instructions.remove(next_insn);

            // Decrement index to stay at same location next iteration
            i--;
            continue;
        }
    }
    // Check for redundant bool casting (same procedure as int, refactored for
primitive Z and Boolean object)
    if (
        current_insn.getOpcode() == INVOKESTATIC
        && ((MethodInsnNode)current_insn).desc.equals("(Z)Ljava/lang/Boolean;")
    ) {
        AbstractInsnNode next_insn = node.instructions.get(i + 1);
        if (
            next_insn.getOpcode() == INVOKEVIRTUAL
            && ((MethodInsnNode)next_insn).desc.equals("(Z)Z")
        ) {
            // Remove both instructions
            node.instructions.remove(current_insn);
            node.instructions.remove(next_insn);
            // Decrement index to stay at same location
            i--;
            continue;
        }
    }
    // Check for redundant double casting (same procedure as int, refactored for
primitive D and Double object)
    if (
        current_insn.getOpcode() == INVOKESTATIC
        && ((MethodInsnNode)current_insn).desc.equals("(D)Ljava/lang/Double;")
    ) {
        AbstractInsnNode next_insn = node.instructions.get(i + 1);
```

# The Fearn Programming Language

```
        if (
            next_insn.getOpcode() == INVOKEVIRTUAL
            && ((MethodInsnNode)next_insn).desc.equals("()D")
        ) {
            // Remove both instructions
            node.instructions.remove(current_insn);
            node.instructions.remove(next_insn);
            // Decrement index to stay at same location
            i--;
            continue;
        }
    }
}
```

## 2.6: Generate target code

Java bytecode (executed by the Java Virtual Machine) is generated by the CodeGenerator and the AST Nodes (which generate bytecode to implement their own logic, using basic instructions, labels, and bytecode generated by nodes in their subtree). Since the methods to generate bytecode and store it in class files already feature in the *Technical Solution* section (as do multiple examples of GenerateBytecode() implementations), I will only (again) direct the reader to Appendix C for more detail.

## 2.7: Any errors raised must be as informative as possible

This is achieved using the toString() method, implemented in every node within the ast package. This can be invoked on any node, and returns a string representation of the node - following Fearn syntax. The purpose of this is to allow the compiler (in case of an error) to reproduce the offending source code, so the user can easily find and fix it. Examples of toString() implementations can be found in Appendix C, in any non-abstract class in the ast package.

- Errors during AST Construction substitute the string representation with the exact line and column where the error occurred. See ASTConstructor (Appendix C) for more detail.

The errors themselves are reported by the Reporter class, which prints them in bold red in order to be eye-catching to the user. The error-reporting method is included below.

## The Fearn Programming Language

### Reporter.java (*ReportErrorAndExit*)

```
public static void ReportErrorAndExit(String err, ASTNode offendingNode)
{
    if (offendingNode == null)
        System.out.println(
            String.format(
                "FearnC (%s): %s%sERROR: %s %s",
                CodeGenerator.GeneratorStack.peek().programName,
                ANSI_BOLD,
                ANSI_RED,
                err,
                ANSI_RESET
            )
        );
    else
        System.out.println(
            String.format(
                "FearnC (%s): %s%sERROR: %s - %s %s",
                CodeGenerator.GeneratorStack.peek().programName,
                ANSI_BOLD,
                ANSI_RED,
                offendingNode.toString(),
                err,
                ANSI_RESET
            )
        );
    System.exit(1);
}
```

# The Fearn Programming Language

## Technical Skills

This section is dedicated to pointing out high-level technical skills used in the project, referencing the source code. All the below mentioned classes can be found in Appendix C, and so are also found in the subsection above (*Objectives Achieved*).

### Model

- **Trees:** The programs are represented using an Abstract Syntax Tree, modelled using objects in the `ast` package, linked together with composition (a node will contain nodes representing its subtree, e.g. an `ExpressionStatement` containing an `Expression`)
- **Stacks:** Stacks are used in multiple places around the program
  - In `ASTConstructor`, a stack is used to represent the variable identifiers currently in scope. This is then used to verify that, when variables are referenced, they exist within the scope at that point in the program.
  - In `CodeGenerator`, there are two stacks that are static attributes of the class
    - `GeneratorStack`: Stores instances of `CodeGenerator`, in order to access the name of the program being compiled (which varies as programs can be imported). This is then used to set the owner of rows and refer to the output program class.
    - `LabelStack`: Stores `Label` objects used during code generation, when generating loops. They are used to allow `JumpStatements` to reference the correct labels for the program to go to, when executed, by ensuring the correct labels for the most recent loop are always at the top of the stack.
- **Complex Object-Oriented Programming:** The Abstract Syntax Tree is represented using objects, which are composed of other objects that represent the state and structure of abstract structures within the Fearn source code. All AST Nodes inherit from the base `ASTNode` class, which is abstract. There are also other abstract superclasses: `Statement`, `Expression`, and `TypeSpecifier`. `JumpStatement` is also used as the base class of `ReturnStatement`, as they both result in a jump in the control of the program. Additionally, abstract statements in the abstract classes are overridden by the derived classes (*polymorphism*). The program leverages public, private, protected, and static class members (*protected members in Java can only be accessed by subclasses and other members of the same package*); it also implements a common interface for `Statement` and `Expression` classes - all of which feature a `validate()` and `GenerateBytecode()` method so traversals of the AST are easy to perform and objects remain independent of each other.



## The Fearn Programming Language

- **Defensive Programming and Exception Handling:** Although classed under Coding Style, I felt it was worth pointing out that the semantic analysis tasks performed by the program (Symbol Analysis and Type Analysis), as well as Error Reporting, are examples of the program screening user input to ensure it's valid.

## Algorithms

- **Stack Operations:** *See above*
- **Tree Traversals:** Multiple depth-first tree traversals are performed in the program. The `validate()` and `GenerateBytecode()` methods are defined and implemented in the AST Node classes themselves.
  1. `ASTConstructor` defines the logic for visiting each node in the ANTLR parse tree. It uses the in-built `visit()` method of the ANTLR Walker to do this.
  2. A semantic traversal is done on the Abstract Syntax Tree. This is done using a common `validate()` method, which is implemented in every class of the `ast` package. Each node invokes the method on its children, so all nodes are traversed - to ensure the program is valid.
    - a. The philosophy behind this is that most nodes validate themselves in isolation, independent of the context in which they exist in the AST. Some exceptions exist to this rule, however, such as `JumpStatements` (`break`, and `continue`); these use a static property of the `CodeGenerator` (`LoopDepth`) to ensure the traversal is within a loop, making them valid (`depth > 0`).
  3. Code Generation for functions is done with a `GenerateBytecode()` method, implemented for every node representable in a function body (statements, expressions, and declarations). Each node invokes the method for its children, when appropriate (e.g. a binary addition expression, between two integer values, will generate the bytecode for its two operands, then append an `iADD` operation to add them together)
    - a. Nodes, again, generate bytecode for their logic independently.
    - b. All nodes visit their instructions using a `MethodVisitor` (a class provided by the Java ASM library), which is used by ASM to then create the method in the compiled program class.
- **Recursive Algorithms**
  1. The static method `SymbolTable.GenBasicDescriptor()` uses recursion for generating descriptors for arrays.
  2. The static method `FearnRuntime.equals()` (used when compiled programs are run to compare the top two objects on the stack, and return true if they're equal) uses recursion to compare arrays, by invoking itself on each element of each array.

## The Fearn Programming Language

3. Several programs in the Example Scripts (Appendix B), as well as the Testing section, use recursion. These include the `GCD()` example, `BinarySearch()`, and the `GetPath()` function of the Dijkstra's implementation.
  4. The traversals performed on the AST could be considered 'recursive-like', in that each method implementation calls the method again on its child nodes - until it reaches the leaves of the tree (the base case)
- **Dynamic Generation of Objects:** This is self-evident in the fact that the Abstract Syntax Tree (modelled with objects) is built by the `ASTConstructor` to represent the program the user has specified within the inputted `.fearn` source file.

## Example Scripts: Additional Skills

As part of the testing process, I wrote a number of example Fearn programs, which demonstrate a number of A-level standard algorithms. These can be viewed in Appendix B and the Testing section, and include **Merge Sort**, **Bubble Sort**, **Binary Search**, and **Dijkstra's Algorithm**. These demonstrate the implementation of complex algorithms, recursive design, and OOP-like programming (through the use of structs and Universal Function Notation / Universal Function Call Syntax), among other skills considered high-level by the specification.

## Dependencies



Below are the external libraries my project relies on, with URLs to the projects themselves.

- **ANTLRv4:** Parser generator, using to generate a Java parser program for the Fearn grammar, tokenizing the source code, and parsing the tokens into an ANTLR Parse Tree, which could be walked using a class derived from a custom-generated ANTLR Tree Walker.
  - Website: <https://www.antlr.org/>
- **ASM:** Bytecode engineering library used to generate Java classes (*representing Fearn programs [program class files] and Fearn structs [struct class files]*), by visiting individual JVM instructions.
  - Website: <https://asm.ow2.io/>
- **AssertJ:** Library to perform assertions in Java, which throw Exceptions if a condition is not true. This is used to compare `ASTNode` objects, by comparing their class and recursively checking their attributes. This was vital as Java does not natively have any way to compare objects for equality, except checking if they are *literally* the same object (at the same memory address).
  - Website: <https://joel-costigliola.github.io/assertj/>

## Testing

When approaching testing for a program as complex as a compiler, I've chosen an approach that tests it against scripts written in the Fearn language that demonstrate that the points of my objectives were achieved, and which are divided into four categories: simple, complex, and erroneous. The below test cases are primarily for showing specific features required in Objective 1. The fact that they work implies Objective 2 functioning (some more explicit testing is included later).<sup>23</sup>

### Simple Testing

Script	Recorded Output
<pre>// HelloWorld.fearn : Simple Hello World program // Demonstrates Objective 1.3, 1.7  // Import IO functions import io // Main procedure (where execution starts) fn main(args : str[]) =&gt; void { print("Hello, World!"); }</pre>	
<pre>/* HelloName.fearn : Print name to terminal  * Tests print(), input(), and length()  * Demonstrates objectives 1.1, 1.3, 1.4, 1.9  */ import io fn main(args : str[]) =&gt; void {     // Declare string variable name     let name: str;     if (args.length() &gt; 0)     {         // If argument has been provided, assign name         name = args[0];     } else {         // Otherwise, get name from input         name = input("What's your name : ");     }     print("Hello, " + name + "!"); }</pre>	

<sup>23</sup> These QR Codes will lead you to the testing videos. They are somewhat difficult to watch on phones, so if you have trouble, this URL will take you to the folder where the videos are hosted **(they are numbered in order, as they appear - so the HelloWorld test is test1)**:  
[https://drive.google.com/drive/folders/1CKI61lFe-O\\_9beuJIxkCI9r94ksAIPM5](https://drive.google.com/drive/folders/1CKI61lFe-O_9beuJIxkCI9r94ksAIPM5)

## The Fearn Programming Language

```
/*  
Fibonacci.fearn : Prints the first n numbers of the  
Fibonacci sequence, where n is passed in at run time.
```

```
Demonstrates Objectives 1.2 (arithmetic), 1.5 (casting),  
1.6 (arrays)
```

```
*/
```

```
import io
```

```
fn main(args : str[]) => void
```

```
{  
    let n : int = (int)input("Enter n : ");  
    print((str)fib(n));  
}
```

```
fn fib(n : int) => int[]
```

```
{  
    let arr : int[] = new int[n];  
    for (let i : int = 0; i < n; i++)  
    {  
        if (i < 2) { arr[i] = 1; }  
        else {  
            arr[i] = arr[i-1] + arr[i-2];  
        }  
    }  
    return arr;  
}
```



```
/* main.fearn : An example of importing global elements (structs,  
 * functions, etc.) from a separate Fearn program.  
 * Demonstrates Objective 1.10 (imports)  
 */
```

```
import io
```

```
import from 'Person.fearn'
```

```
fn main(args : str[]) => void
```

```
{  
    let user : $Person = new Person(args[0], (int)args[1]);  
    print(user.toString());  
    print(user.name + "'s name is " + (str)user.name.length()  
        + " characters long.");  
    print("Global Variable value: " + GLOBAL);  
}
```



## The Fearn Programming Language

```
/* Person.fearn : A file that is imported by main.fearn.
 * As no main function is included, it cannot be used as
 * a program on its own.
 *
 * Demonstrates 1.8 (structs)
 */

struct Person {
    let name : str;
    let age : int;
}

let GLOBAL : str = "This is a global variable, defined in Person.fearn"

fn toString(person : $Person) => str
{
    return person.name + ", a " + (str)person.age + " year old.";
}
```

```
/* arithmetic.fearn : Demonstrates simple mathematical operations. */

import io

fn main() => void
{
    let a : float = (float)input("Enter A: ");
    let b : float = (float)input("Enter B: ");


    print("a + b : " + (str)(a + b));
    print("a - b : " + (str)(a - b));
    print("a * b : " + (str)(a * b));
    print("a / b : " + (str)(a / b));
    print("a ^ b : " + (str)(a ^ b));

    print("a % b : " + (str)((int)a % (int)b));
}
```



# The Fearn Programming Language

## Complex Testing

Script	Recorded Output
<pre>/* BinarySearch.fearn : Implements a Binary Search on a  * list provided on the command line.  * Also demonstrates use of the slice() function.  */ import io fn main(args : str[]) =&gt; void {     if (args.length() == 0) { print("Must provide list"); return; }      let term : int = (int)input("ENTER SEARCH TERM : ");      let list : int[] = new int[ args.length() ];     for (let i : int = 0; i &lt; args.length(); i++)     { list[i] = (int)args[i]; }      let index : int = list.BinarySearch(term);     if (index &gt; -1) { print("INDEX IS : " + (str)index); }     else { print("NOT FOUND"); } } fn BinarySearch(list : int[], term : int) =&gt; int {     let m : int = list.length() / 2;     let index : int;      if (list.length() &lt; 1) { return -1; }     else if (list[m] == term) { index = m; }     else if (list[m] &gt; term) {         index = BinarySearch(list.slice(0, m), term);     } else {         index = BinarySearch(             list.slice(m + 1, list.length()),             term         );         if (index &gt; -1 ) { index += m + 1; }     }     return index; }</pre>	

## The Fearn Programming Language

```
/* GCD.fearn : Implements Euclid's Algorithm
 * recursively, to find the Greatest Common Divisor
 * of two numbers, A and B.
 */
```

```
import io
```

```
fn gcd(A : int, B : int) => int
{
    if (A == 0) {
        return B;
    } else if (B == 0) {
        return A;
    }

    return gcd(B, A % B);
}
```

```
fn main(args: str[]) => void
{
    print((str)gcd((int)args[0], (int)args[1]));
}
```



```
/* dijkstra.fearn : Implements Dijkstra's Algorithm (shortest
 * path) on the graph specified in graph.fearn.
 */
```

```
import io
```

```
import from "graph.fearn"
```

```
fn GetNode(id : str) => $Node
{
    for (let i : int = 0; i < nodes.length(); i++)
    {
        if (nodes[i].ID == id) { return nodes[i]; }
    }

    // This would cause an error if the node is not
    // found. It should never execute.
    return nodes[i];
}
```

```
let frontier : $Node[] = new $Node[nodes.length()];
let lengthOfFrontier : int = 0;
```



## The Fearn Programming Language

```
fn AddToFrontier(node : $Node) => void
{
    for (let i : int = 0; i < lengthOfFrontier; i++) {
        if (frontier[i].index == node.index) { return; }
    }
    frontier[lengthOfFrontier++] = node;
}

fn RemoveFromFrontier(node : $Node) => void
{
    let new_frontier : $Node[] = new $Node[nodes.length()];
    let x : int = 0;
    for (let i : int = 0; i < lengthOfFrontier; i++) {
        if (frontier[i].index == node.index) { continue; }
        new_frontier[x++] = frontier[i];
    }
    lengthOfFrontier--;
    frontier = new_frontier;
}

fn GetNextNode() => $Node
{
    let next_node : $Node = frontier[0];
    for (let i : int = 1; i < lengthOfFrontier; i++)
    {
        if (frontier[i].label < next_node.label)
        {
            next_node = frontier[i];
        }
    }
    return next_node;
}

fn Dijkstra(start: str, target : str) => void
{
    let current : $Node = start.GetNode();
    current.label = 0;
    current.isFinal = true;
    for (;current.ID != target;)
    {
        let neighbours : int[] = graph[current.index];
        // For each neighbour of the current node, update
```



## The Fearn Programming Language

```
// labels and add to frontier
for (let i : int = 0; i < neighbours.length(); i++)
{

    if (neighbours[i] < 0) { continue; }

    let neighbour : $Node = nodes[i];
    if (neighbour.isFinal) { continue; }

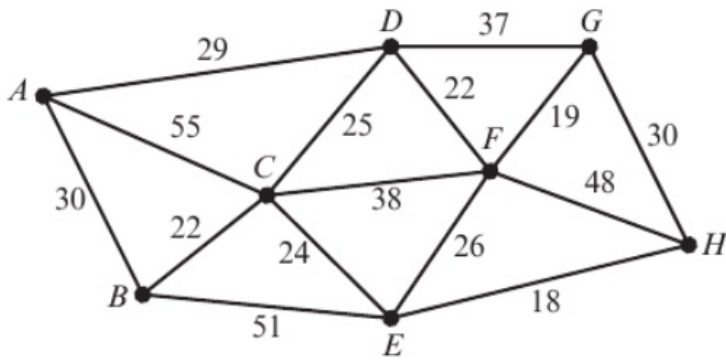
    let new_label : int = current.label + neighbours[i];
    if (neighbour.label < 0 || new_label < neighbour.label)
    {
        neighbour.label = new_label;
        neighbour.previous = current.ID;
    }
    neighbour.AddToFrontier();
}

if (lengthOfFrontier == 0) { return; }
let next_node : $Node = GetNextNode();
next_node.isFinal = true;
next_node.RemoveFromFrontier();
current = next_node;
}
}

fn GetPath(start : str, end : str) => str
{
    let path : str = end;
    if (start != end)
    {
        path = GetPath(start, end.GetNode().previous)+" -> "+path;
    }
    return path;
}

fn main() => void
{
    let start : str = input("Enter Start : ");
    let end : str = input("Enter End : ");
    Dijkstra(start, end);
    let path : str = GetPath(start, end);
    path += " (Distance " + (str)end.GetNode().label + ")";
    print(path);
}
```

## The Fearn Programming Language



// graph.fearn: Represents the graph above

```
struct Node {
    let ID : str;
    let index : int;
    let label : int;
    let isFinal : bool;
    let previous : str;
}

// Distance matrix
let graph : int[][] = new int[][] {
    {-1, 30, 55, 29, -1, -1, -1, -1},
    {30, -1, 22, -1, 51, -1, -1, -1},
    {55, 22, -1, 25, 24, 38, -1, -1},
    {29, -1, 25, -1, -1, 22, 37, -1},
    {-1, 51, 24, -1, -1, 26, -1, 18},
    {-1, -1, 38, 22, 26, -1, 19, 48},
    {-1, -1, -1, 37, -1, 19, -1, 30},
    {-1, -1, -1, -1, 18, 48, 30, -1}
};

// Array of nodes
let nodes : $Node[] = new $Node[] {
    new Node("A", 0, -1, false, ""),
    new Node("B", 1, -1, false, ""),
    new Node("C", 2, -1, false, ""),
    new Node("D", 3, -1, false, ""),
    new Node("E", 4, -1, false, ""),
    new Node("F", 5, -1, false, ""),
    new Node("G", 6, -1, false, ""),
    new Node("H", 7, -1, false, "")
};
```

# The Fearn Programming Language

## Erroneous Testing

Erroneous Code	Output
<code>let num : int = "Hello";</code>	FearnC (testing): <b>ERROR: let num : int = "Hello"; - Cannot assign str to int</b>
<code>23 + true</code>	FearnC (testing): <b>ERROR: 23 + true - Operands must be either (a) both ints, (b) both floats, or (c) both strings.</b>
<code>(int)(new MyStruct())</code>	FearnC (testing): <b>ERROR: (int)new MyStruct() - Cannot perform cast from \$MyStruct to int.</b>
<code>new int[] {1, 2, true, 4, "2"}</code>	FearnC (testing): <b>ERROR: {1, 2, true, 4, "2"} - ArrayBody has inconsistent element type.</b>
<code>list["0"]</code>	<b>ERROR: list["0"] - Index can only be an int.<sup>24</sup></b>
<code>6++</code>	<b>ERROR: 6++ - Cannot increment 6. Only int variables can be incremented/decremented.</b>
<code>fn f(x : int) =&gt; int {     return 5; }  f()</code>	<b>ERROR: f() - Wrong number of arguments for f , expected 1</b>
<code>f("string")</code>	<b>ERROR: f("string") - "string" is of the wrong type, expected int</b>
<code>q("string")</code>	<b>ERROR: Function q is not defined.</b>
<code>fn f(x : int) =&gt; int {     return "Hello"; }</code>	<b>ERROR: return "Hello"; - Incorrect return type, expected int</b>
<code>fn f(x : int) =&gt; int {}</code>	<b>ERROR: Function f must include a return statement in its main body.</b>
<code>1 + 2;</code>	<b>ERROR: 1 + 2; - Invalid Statement.</b>
<i>No Source File Provided</i>	<b>FearnC (null): ERROR: NO SOURCE FILE</b>
<i>Source File not found</i>	FearnC (non_existant): <b>ERROR: FILE non_existant.fearn NOT FOUND</b>

These are all examples of errors raised by the Fearn compiler when run with erroneous code. However, this is a small selection. To see the contexts of other errors, that can be found in Appendix C, in most source files calling `Reporter.ReportErrorAndExit()`, and well as `Reporter.java` itself.

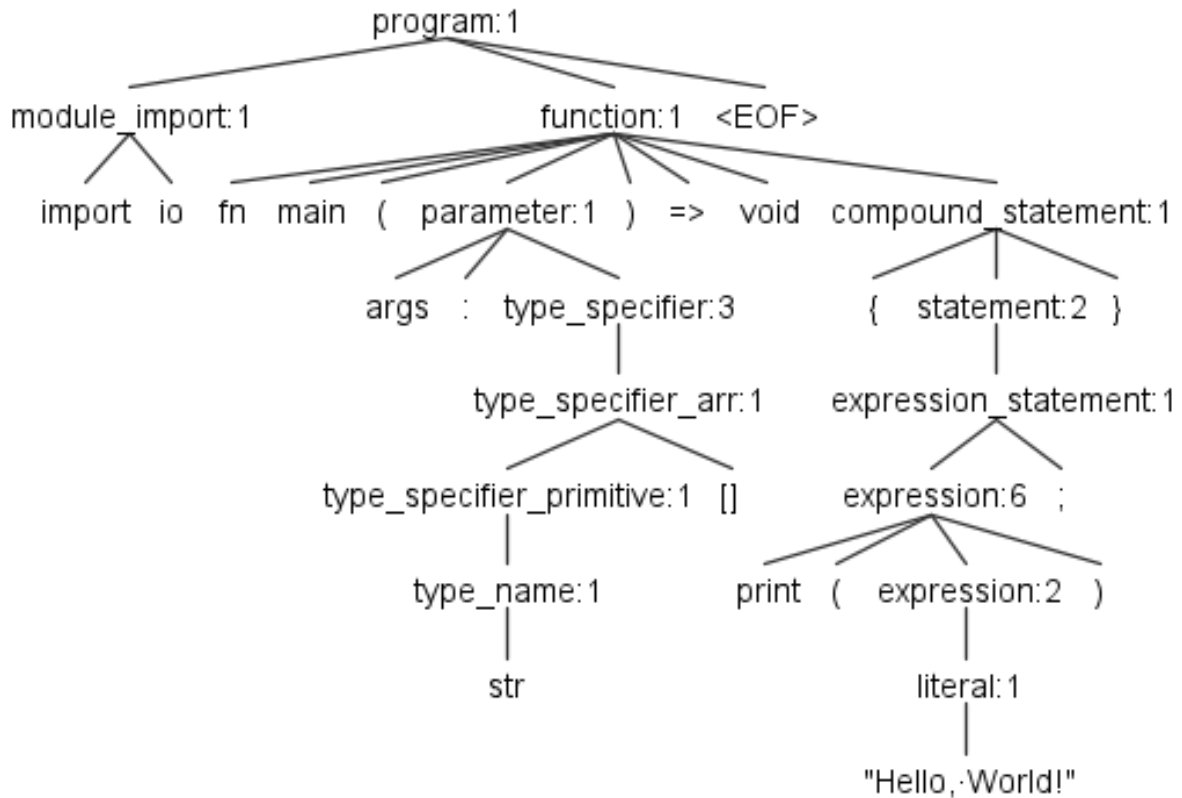
---

<sup>24</sup> I'm hereafter not including 'FearnC (filename):' to save space, and show the error itself better

# The Fearn Programming Language

## Parsing Test

This is a parse tree, produced by ANTLR, which tests the FearnGrammar.g4 file. This one in particular is produced by the HelloWorld.fearn example script (Appendix C), a short and simple example - chosen only so the parse tree can comfortably fit on the page.



# Evaluation

## General Evaluation

Overall, I'm incredibly pleased with how my project has turned out, especially considering the daunting scope of the idea that it was conceived in the first place: creating a compiler for a general-purpose, C-like programming language, from scratch. The project, as shown in the [Technical Solution](#) and [Testing](#) sections, meets all objectives set for it - not considering the number of other features I included in order to make Fearn a true C-style general purpose language - including C-style comments, iteration statements (for loops), selection (if-else) statements, increment/decrement expressions, and an expanded standard library.

## Objective 1: Define the FearnLang Syntax

I believe this objective was met extremely well. The Technical Solution section details how each sub-objective was implemented within the compiler, and the Testing section shows each required feature operating successfully, both in isolation and when composed into more complex systems, like the Dijkstra's and Merge Sort implementations.

## Objective 2: Develop the FearnLang Compiler

Once again, the proof of this objective being met is that the Fearn compiler functions correctly, and produces code that has shown to be fast and effective. The compiler itself, as a program, is laid out and compartmentalised rationally into packages, sub-packages, and classes - and data is hidden where necessary to improve the security and reliability of an Object-Oriented project on this scale.

## Potential Improvements

The language has been designed with future development in mind, being rigorously documented and compartmentalised. Improvements I'd like to make include an expanded standard library - perhaps for making HTTP requests or to allow a developer to design a simple web server, and a module for creating/manipulating GUIs.

One large regret I have is not including any sort of null value, however this could be rectified (relatively) easily by modifying the grammar, and including null as a type (perhaps inheriting from all other types to ensure a null type can be accepted where necessary, while not requiring a full restructuring of the type system). However, the inclusion of such a feature comes with drawbacks and potential risks of an

## The Fearn Programming Language

erroneous program being built without error, a valid program being rejected, or errors in the compiler itself.

### Independent Feedback

In order to evaluate my program, I installed the language on a friend's computer, and asked him to experiment with the language - doing his best to find errors if he could. His comments are below:

`"I've tested out Fearn for some basic programs such as defining a user and displaying the contents of the user on the terminal. I also found that the error messages received were very clear and helpful in finding the mistakes in my coding. I also tried to find bugs, by entering illegal values into functions and using predefined functions and variables and all attempts resulted in the program functioning perfectly or returning an error directing me to the illegal value I had entered."`

Considering that he was intentionally attempting to break the program, the fact that it has stood up to field testing demonstrates the robustness of my compiler's design and implementation - considering a large number of edge cases and potential misuses of the language. The comment about error messages being informative and helpful in locating bugs also demonstrates the language's credentials as a *learner* language, showing the programmer the mistakes made during development (showing the name of the file and offending source code), and the exact reason the code is rejected.

# Appendix A – 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;  
}
```

# The Fearn Programming Language

```
/* 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 */
```



# The Fearn Programming Language

```
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 */
```

# The Fearn Programming Language

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
```

# The Fearn Programming Language

```
;
```

```
/* 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
    ;
```

# The Fearn Programming Language

*/\* 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 arguments \*/*

```
argument_expression_list
    : expression
    | argument_expression_list ',' expression
    ;
```

*/\* Operators on a single operand \*/*

```
unary_operator
    : '+'
    | '-'
    ;
```

*/\* Assignment Operators \*/*

```
assignment_operator
    : '='
    | MULT_ASSIGN
    | DIV_ASSIGN
```

## The Fearn Programming Language

```
| 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
```

## The Fearn Programming Language

```
| 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
```

# The Fearn Programming Language

```

;

/* 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;
}
```

# The Fearn Programming Language

## *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
```



# The Fearn Programming Language

*/\* 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);	}

## The Fearn Programming Language

"=="	{ 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('^'); }

## The Fearn Programming Language

```
%%  
  
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;  
}
```

# Appendix B – Example Fearn Programs

All of these are included in the Example Scripts directory in this project's GitHub repository (*see front cover*). They are included here for convenience.

## HelloWorld.fearn

```
import io

fn main(args : str[]) => void
{
    print("Hello, World!");
}
```

## StructExample.fearn

```
import io

struct Person {
    let name : str;
    let age : int;
}

fn main(args : str[]) => void
{
    let user : $Person = new Person(
        args[0],
        (int)args[1]
    );

    print("The user is " + user.name + ", a " + (str)user.age + " year old.");
}
```

# The Fearn Programming Language

## BubbleSort.fearn

```
/*
This is a Fearn Program that implements a bubble sort
on the arguments passed through in the command line.
E.g. FearnRun BubbleSort 1 3 7 2 8 -> [1, 2, 3, 7, 8]
*/

import io

fn main(args : str[]) => void
{
    let list : int[] = new int[args.length()];
    for (let i : int = 0; i < args.length(); i++)
    {
        list[i] = (int)args[i];
    }
    print(bubble(list));
}

fn bubble(l : int[]) => int[]
{
    let swapped : bool = true;
    let temp : int;
    for (;swapped;)
    {
        swapped = false;
        for ( let i : int = 1; i < l.length(); i += 1 )
        {
            if ( l[i - 1] > l[i] )
            {
                temp = l[i - 1];
                l[i - 1] = l[i];
                l[i] = temp;
                swapped = true;
            }
        }
    }
    return l;
}
```

# The Fearn Programming Language

## MergeSort.fearn

```
/*
MergeSort.fearn

Implements a Merge Sort.

*/

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
fn merge(arr : int[], l : int, m : int, r : int) => void
{
    // Find sizes of two subarrays to be merged
    let n1 : int = m - l + 1;
    let n2 : int = r - m;

    // Create temp arrays
    let L : int[] = new int[n1];
    let R : int[] = new int[n2];

    let i : int;
    let j : int;
    let k : int;

    // Copy data to temp arrays
    for (i = 0; i < n1; i += 1)
    {
        L[i] = arr[l + i];
    }

    for (j = 0; j < n2; j += 1)
    {
        R[j] = arr[m + 1 + j];
    }

    // Merge the temp arrays

    // Initial indices of first and second subarrays
```

## The Fearn Programming Language

```
i = 0;
j = 0;

// Initial index of merged subarray array
k = 1;

for (; i < n1 && j < n2; ) {

    if (L[i] <= R[j])
    {
        arr[k] = L[i];
        i += 1;
    }

    else
    {
        arr[k] = R[j];
        j += 1;
    }

    k += 1;

}

// Copy remaining elements of L[] if any
for (; i < n1 ;) {
    arr[k] = L[i];
    i += 1;
    k += 1;
}

// Copy remaining elements of R[] if any
for (; j < n2 ;) {
    arr[k] = R[j];
    j += 1;
    k += 1;
}

}

// Main function that sorts arr[l..r] using
// merge()
fn sort(arr : int[], l : int, r : int) => void
```

## The Fearn Programming Language

```
{
    if (l < r) {

        // Find the middle point
        let m : int = l + (r - l) / 2;

        // Sort first and second halves
        sort(arr, l, m);
        sort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}

// Driver code
fn main(args : str[]) => void
{
    let arr : int[] = new int[args.length()];

    for (let i : int = 0; i < args.length(); i++)
    {
        arr[i] = (int)args[i];
    }

    print("Given array is");
    print(arr);

    sort(arr, 0, arr.length() - 1);

    print("Sorted array is");
    print(arr);
}
```



# The Fearn Programming Language

## BinarySearch.fearn

```
fn main(args : str[]) => void
{
    let index : int;
    let list : int[];
    let term : int;

    if (args.length() < 2)
    {
        print("Must provide list and search terms (min. 2 args)");
        return;
    }

    list = new int[ args.length() - 1 ];
    term = (int)args[ args.length() - 1 ];

    for (let i : int = 0; i < args.length() - 1; i++)
    {
        list[i] = (int)args[i];
    }

    print("SEARCH TERM IS : " + (str)term);

    index = BinarySearch(list, term);

    if (index > -1) {
        print("INDEX IS : " + (str)index);
    } else {
        print("ITEM NOT FOUND");
    }
}
```

## The Fearn Programming Language

```
fn BinarySearch(list : int[], term : int) => int
{
    let m : int = list.length() / 2;
    let index : int;

    if (list[m] == term)
    {
        index = m;
    } else if (list.length() <= 1)
    {
        index = -1;
    } else if (list[m] > term)
    {
        index = BinarySearch(list.slice(0, m), term);
        return index;
    } else {
        index = BinarySearch( list.slice(m + 1, list.length()) , term);

        if (index > -1 )
        {
            index += m + 1;
        }
    }
    return index;
}
```

# The Fearn Programming Language

## Dijkstra.fearn

```
/* dijkstra.fearn : Implements Dijkstra's Algorithm (shortest
 * path) on the graph specified in graph.fearn (as a node list
 * and distance matrix).
 */

import io
import from "graph.fearn"

fn GetNode(id : str) => $Node
{
    for (let i : int = 0; i < nodes.length(); i++)
    {
        if (nodes[i].ID == id) { return nodes[i]; }
    }

    // This would cause an error if the node is not
    // found. It should never execute.
    return nodes[i];
}

let frontier : $Node[] = new $Node[nodes.length()];
let lengthOfFrontier : int = 0;

fn AddToFrontier(node : $Node) => void
{
    for (let i : int = 0; i < lengthOfFrontier; i++) {
        if (frontier[i].index == node.index) { return; }
    }
    frontier[lengthOfFrontier++] = node;
}

fn RemoveFromFrontier(node : $Node) => void
{
    let new_frontier : $Node[] = new $Node[nodes.length()];
    let x : int = 0;

    for (let i : int = 0; i < lengthOfFrontier; i++) {
        if (frontier[i].index == node.index) { continue; }
        new_frontier[x++] = frontier[i];
    }

    lengthOfFrontier--;
    frontier = new_frontier;
}
```

## The Fearn Programming Language

```
fn GetNextNode() => $Node
{
    let next_node : $Node = frontier[0];
    for (let i : int = 1; i < lengthOfFrontier; i++)
    {
        if (frontier[i].label < next_node.label)
        {
            next_node = frontier[i];
        }
    }
    return next_node;
}

fn Dijkstra(start: str, target : str) => void
{
    let current : $Node = start.GetNode();
    current.label = 0;
    current.isFinal = true;

    for (;current.ID != target;)
    {
        let neighbours : int[] = graph[current.index];

        // For each neighbour of the current node, update
        // labels and add to frontier
        for (let i : int = 0; i < neighbours.length(); i++)
        {

            if (neighbours[i] < 0) { continue; }

            let neighbour : $Node = nodes[i];
            if (neighbour.isFinal) { continue; }

            let new_label : int = current.label + neighbours[i];

            if (neighbour.label < 0 || new_label < neighbour.label)
            {
                neighbour.label = new_label;
                neighbour.previous = current.ID;
            }

            neighbour.AddToFrontier();
        }

        if (lengthOfFrontier == 0) { return; }
    }
}
```

## The Fearn Programming Language

```
    let next_node : $Node = GetNextNode();
    next_node.isFinal = true;
    next_node.RemoveFromFrontier();
    current = next_node;
}
}

fn GetPath(start : str, end : str) => str
{
    let path : str = end;
    if (start != end)
    {
        path = GetPath(start, end.GetNode().previous) + " -> " + path;
    }

    return path;
}

fn main() => void
{
    let start : str = input("Enter Start : ");
    let end : str = input("Enter End : ");

    Dijkstra(start, end);

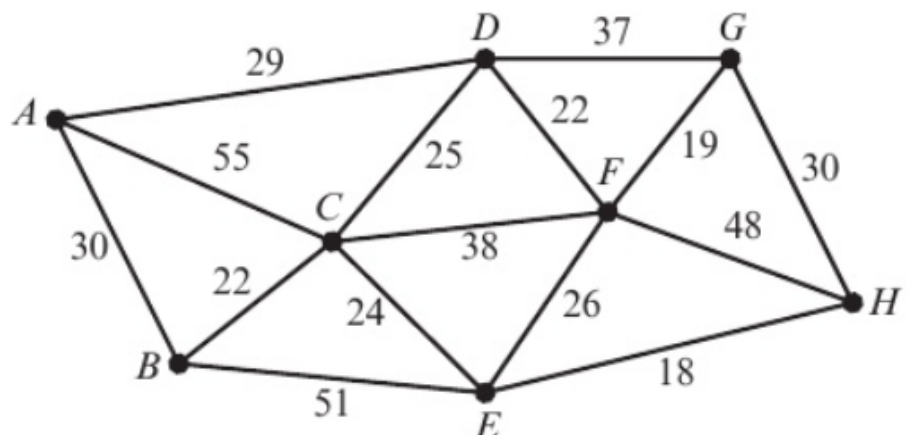
    let path : str = GetPath(start, end);
    path += " (Distance " + (str)end.GetNode().label + ")";

    print(path);
}
```

### graph.fearn

// graph.fearn: Represents the graph pictured

```
struct Node {
    let ID : str;
    let index : int;
    let label : int;
    let isFinal : bool;
    let previous : str;
}
```



## The Fearn Programming Language

```
// Distance matrix (-1 indicates no connection)
```

```
let graph : int[][] = new int[][] {  
    {-1, 30, 55, 29, -1, -1, -1, -1},  
    {30, -1, 22, -1, 51, -1, -1, -1},  
    {55, 22, -1, 25, 24, 38, -1, -1},  
    {29, -1, 25, -1, -1, 22, 37, -1},  
    {-1, 51, 24, -1, -1, 26, -1, 18},  
    {-1, -1, 38, 22, 26, -1, 19, 48},  
    {-1, -1, -1, 37, -1, 19, -1, 30},  
    {-1, -1, -1, -1, 18, 48, 30, -1}  
};
```

```
// Array of nodes
```

```
let nodes : $Node[] = new $Node[] {  
    new Node("A", 0, -1, false, ""),  
    new Node("B", 1, -1, false, ""),  
    new Node("C", 2, -1, false, ""),  
    new Node("D", 3, -1, false, ""),  
    new Node("E", 4, -1, false, ""),  
    new Node("F", 5, -1, false, ""),  
    new Node("G", 6, -1, false, ""),  
    new Node("H", 7, -1, false, "")  
};
```

# Appendix C – Full Source Code

## main.java

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

import parser.*;
import parser.gen.*;
import semantics.table.SymbolTable;

import java.io.FileInputStream;
import java.nio.file.Path;
import java.nio.file.Paths;

import util.*;
import ast.ASTNode;
import ast.Program;
import codegen.*;

/*
 * Main.java
 *
 * This is the compiler's main class, from which compilation begins.
 */

class FearnC
{
    public static FearnGrammarLexer lexer;
    public static CommonTokenStream tokens;
    public static FearnGrammarParser parser;

    static CodeGenerator cg = new CodeGenerator();
    static String sourceFileArgument;

    /* The main function is the first to be called.
     *
     * The function...
     * 1) Adds the initial CodeGenerator (used to compile the script the
     *    user first passes) to the GeneratorStack
     */
}
```

## The Fearn Programming Language

```
* 2)   Raising an error if a source file has not been provided, storing its
*       path otherwise
* 3)   Sets the global build path (a static property of the
*       CodeGenerator class - as all files must be generated in the
*       same place), and the program name of the initial script
* 4)   Calls the Compile method
* 5)   Assuming no errors have been raised during Compilation, prints a
*       success message, including a command-line instruction to run the
*       compiled program
*/
public static void main(String[] args)
{
    // Add initial code generator to stack
    CodeGenerator.GeneratorStack.push(cg);

    // Raise error if no source file has been passed
    if ( args.length == 0 ) {
        Reporter.ReportErrorAndExit("NO SOURCE FILE", null);
    }

    // Set Build Path and Program name for generator
    // Build Path describes where to put all generated class file, and
    // is the same for all generators

    sourceFileArgument = args[0];
    cg.SetBuildPath(sourceFileArgument);
    cg.SetProgramName(sourceFileArgument);

    // Compile source file
    Compile(sourceFileArgument);

    // Pop generator from stack, as it's no longer in use
    CodeGenerator.GeneratorStack.pop();

    // Print green Success Message
    Path parent = Paths.get(sourceFileArgument).getParent();
    parent = parent == null ? Paths.get("build") : parent.resolve("build");
    Reporter.ReportSuccess(
        String.format(
            "Compilation Successful! \n\t -> Run `cd %s ; FearnRun %s [args...]` to run Program",
            parent.toString(),
            Paths.get(sourceFileArgument).getFileName().toString().replace(".fearn", "")
        ),
        true
    );
}
```



# The Fearn Programming Language

```
/* Compile
 *
 * The method responsible for conducting the compilation of the
 * file the user has passed.
 *
 * 1) Loads the source file, raising an error if not found
 * 2) Calls the ANTLR-Generated lexer/parser, to convert the program to
 *    a walkable parse tree
 * 3) Calls the AST Constructor to walk the parse tree, generating the
 *    Abstract Syntax Tree
 * 4) Retrieves the AST's root, and the program's Global Symbol Table.
 *    The Symbol Table is stored in the CodeGenerator class, to be globally
 *    accessible
 * 5) The AST is validated, to ensure it follows the rule of the language
 *    (e.g. type rules, definitions of all functions/structs are present, etc.)
 * 6) Given it is valid, calls the CodeGenerator to generate the program binary,
 *    writing it into the build directory.
 * 7) Pops the initial CodeGenerator off the stack.
 *
 */

static void Compile(String path)
{
    // Read source file
    CharStream input = null;

    try {
        input = CharStreams.fromStream(new FileInputStream(sourceFileArgument));
    } catch (Exception e) {
        Reporter.ReportErrorAndExit("FILE " + sourceFileArgument + " NOT FOUND", null);
    }

    // FearnRuntime.fearn is prohibited, as the resulting class would conflict with the runtime
    if (sourceFileArgument.endsWith("FearnRuntime.fearn") )
    {
        Reporter.ReportErrorAndExit("FILENAME FearnRuntime.fearn IS FORBIDDEN.", null);
    }

    // Create lexer and parser objects
    lexer = new FearnGrammarLexer(input);
    tokens = new CommonTokenStream(lexer);
    parser = new FearnGrammarParser(tokens);
}
```

## The Fearn Programming Language

```
// Add custom error listener (to stylise error messages)
parser.removeErrorListeners();
parser.addErrorListener(new FearnErrorListener());

// Parse program from the root 'program' production
ParseTree parseTree = parser.program();

// Construct AST
ASTConstructor astConstructor = new ASTConstructor();
ASTNode AST = astConstructor.visit(parseTree);

// Retrieve AST root, and Symbol Table
Program root = (Program)AST;
SymbolTable symTable = astConstructor.symbolTableStack.pop();

CodeGenerator.GlobalSymbolTable = symTable;

// Perform semantic analysis
root.validate(symTable);

// Generate program and struct class files
cg.Generate(root, symTable);
}
};
```

# The Fearn Programming Language

## *parser*

### FearnGrammar.g4

```
/*
```

```
FearnGrammar.g4
```

Unlike every other file that forms the compiler, this is not a java source file. This is an ANTLR grammar file, written using EBNF. The difference between EBNF and regular BNF is that it allows the use of regex-style metacharacters, allowing me to write more concise production, and less of them. This allows me to write a far neater grammar.

Each rule has one or more patterns associated with it, containing the rules or tokens it is made up of.

The rule at the bottom of file (in UPPER CASE) are token rules - used by the lexer to match patterns for tokens in the source code, which have a large language that couldn't be explicitly specified (such as keywords, e.g. 'new', 'fn', 'int', etc.)

Some rules have labels next to them ( indicted with #). These instruct ANTLR to generate the custom walker class with separate visit methods and contexts for each

```
*/
```

```
grammar FearnGrammar;
```

```
/* DATA TYPES AND TYPE SPECIFIERS */
```

```
// type_name matches the literal names of the 4 primitive types
```

```
type_name
    : 'int'
    | 'float'
    | 'bool'
    | 'str'
    ;
```

# The Fearn Programming Language

```
// type_specifiers can describe a type which is primitive, an array,
// or a struct
type_specifier
    : type_specifier_primitive
    | type_specifier_struct
    | type_specifier_arr
    ;

// primitive specifiers are just the type name (e.g. let n : int; )
type_specifier_primitive
    : type_name
    ;

// To use a struct as a data type, it can be done in the form ${MyStruct}
// A $ prefix is used to differentiate a struct instance specifier from
// other source code references to the struct
type_specifier_struct
    : '$' IDENTIFIER
    ;

// An array specifier will be the type specifier of the elements contained,
// and the N '[]'s, to indicate an N-dimensional array
type_specifier_arr
    : (type_specifier_primitive | type_specifier_struct ) ('[]')+
    ;

/* HIGH LEVEL STRUCTURES */

// Imports start with the 'import' keyword, followed with either
// -> An identifier, indicating the import of a standard library module
//      (e.g. `io` or `maths`)
// -> The 'from' keyword, followed by a string literal, of the relative
//      path, from the main program file, of the file being imported
module_import
    : 'import' (IDENTIFIER | 'from' STR_LIT)
    ;

// Fearn Programs, at their root, are 0 or more imports, followed by one
// or more instances of a function, global variable, or struct definition
// -> EOF is a special token for the end of a file, included to ensure
//      this production as consumed the entire source file, and would raise
//      an error if it had, due to poor syntax, not consumed it all.
program
    : (module_import)* (function | declaration | struct_def )+ EOF
    ;
```

# The Fearn Programming Language

```
// Functions are defined using the 'fn' keyword, a list of parameters, a
// return type (a type specifier or 'void'), and then a compound statement
// for the function body.
// -> The '=>' symbol is simply to indicate the data type returned.
function
    : 'fn' IDENTIFIER '(' (( parameter ',' )* parameter)? ')' '=>' ( type_specifier | 'void' )
compound_statement
    ;

// Function parameters have an identifier, and a data type specifier
parameter
    : IDENTIFIER ':' type_specifier
    ;

// Structs are defined using the 'struct' keyword, an identifier, as well as a
// list of 0 or more declarations, used to declare variables. These declared
// variables become the attributes of the struct, with each instance of a struct
// having a value for each attribute.
struct_def
    : 'struct' IDENTIFIER '{' declaration* '}'
    ;

// Declarations represent the creation of new variables within a defined scope.
// It uses the 'let' keyword, an identifier for the variable, and a type specifier.
// An expression (to initialise the variable) can also be added.
declaration
    : 'let' IDENTIFIER ':' type_specifier ( '=' expression)? ';'
    ;

/* STATEMENTS */

// Statements can be one of the five below types
statement
    : compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    ;

// A compound statement, syntactically, is a collection of declarations and statements,
// between '{}'.
compound_statement
    : '{' (declaration | statement)* '}'
    ;
```

# The Fearn Programming Language

```
// An expression statement is an expression (e.g. a function call), or an assignment
// (changing the value of a variable)
expression_statement
: expression ';'          # simple_expr_stmt
| assign_expression ';'    # assign_expr_stmt
;

// A selection statement is a standard 'if-else' statement. It can be either:
// -> A single if, where the body is only run if the condition is met
// -> An if-else chain, where another body/selection is run if the first doesn't
selection_statement
: 'if' '(' expression ')' compound_statement          # single_if
| 'if' '(' expression ')'
    compound_statement
    'else' (compound_statement | selection_statement) # if_else
;

// Fearn doesn't support while loops, but does feature a highly permissive for loop,
// which can perform the same purpose. The following few rules allow the user to create
// a loop, in the form `for (int i = 0; i < 10; i++) {...}`
iteration_statement
: 'for' '(' init_expression continue_condition iteration_expression ')' compound_statement
;

// The initialisation can be a declaration, expression, or assignment

// The declaration requires no ';', as its definition already matches an ending ';'
init_expression
: declaration
| expression ';'
| assign_expression ';'
| ';'
;

// The continue condition must be a boolean expression
continue_condition
: expression ';'
;

// The iteration expression (run at the end of every loop) can be either an expression,
// assignment, or nothing at all
iteration_expression
: expression
| assign_expression
|
;
;
```

# The Fearn Programming Language

```
// A jump statement is one of the below, which function as you'd expect in any C-like language
// The return expression is optional, as void functions don't return a value
jump_statement
: 'continue' ';'          # cont_stmt
| 'break' ';'             # break_stmt
| 'return' expression? ';' # return_stmt
;

/* EXPRESSIONS */

// This is a recursive rule for all non-assignment expressions in Fearn (i.e. all code that
// evaluates to a data value). They are defined recursively, as there's no way to tell what
// type of expression should be used in many contexts. For example, the unary not expression can
// operate on any expression that results in a boolean value (e.g. a boolean function, variable,
// struct
// instance attribute, other boolean expressions in brackets, etc). Most of the below rules are
// self-explanatory.
// -> The dot expression can be used either to access a struct attribute, or to call a
//      function using Universal Function Notation / UFN ( a.f(x) == f(a, x) )
// -> The order of rules defines their precedence, and what the parse tree looks like.
//      For example, the expression `a + b * c` is parsed as `(a + (b * c))`, because
//      multiplicative expressions are matched before additive expressions
// -> The arithmetic expressions use `op=(...)`, because the operators must be of the same
//      precedence

expression
: IDENTIFIER                # id_expr
| literal                   # lit_expr
| array_init                # arr_init_expr
| struct_init               # struct_init_expr
| expression '.' expression # dot_expr
| IDENTIFIER '(' ( ( expression ',' )* expression )? ')' # fn_call_expr
| '(' expression ')'        # brac_expr
| expression '[' expression ']' # index_expr
| '+' expression            # u_plus_expr
| '-' expression            # u_minus_expr
| '!' expression            # u_not_expr
| op=('++' | '--') expression # pre_inc_expr
| expression op=('++' | '--') # post_inc_expr
| '(' type_name ')' expression # cast_expr
| expression '^' expression  # exp_expr
| expression op=('*' | '/' | '%') expression # mult_expr
| expression op=('-' | '+') expression # add_expr
```

# The Fearn Programming Language

```
| expression '<' expression                                # less_expr
| expression '>' expression                                # greater_expr
| expression '<=' expression                               # less_eq_expr
| expression '>=' expression                               # greater_eq_expr
| expression '==' expression                              # eq_expr
| expression '!=' expression                              # not_eq_expr
| expression '&&' expression                               # and_expr
| expression '||' expression                              # or_expr
;

// Literal matches any literal token
literal
    : STR_LIT | BOOL_LIT | INT_LIT | FLOAT_LIT ;

// array_init matches an array initialisation, which requires a type specifier for the element
// types,
// and either...
// A) One or more dimensions (in square brackets)
// B) A body that represents its initial value
// This syntax follows Java's notation for initialising fixed-length arrays
array_init
    : 'new' ( type_specifier_primitive | type_specifier_struct ) ('[' expression ']')+
    | 'new' ( type_specifier_primitive | type_specifier_struct ) ('[]')+ array_body
    ;

// An array body is 0 or more elements, which may themselves be array bodies (for
// multi-dimensional arrays)
array_body
    : '{' (array_body ',')* array_body '}'
    | '{' (expression ',')* expression '}'
    | '{}''
    ;

// A struct is initialised with the identifier of the struct (which we're making an instance of),
// and arguments
// to set that struct's initial state
struct_init
    : 'new' IDENTIFIER '(' ( ( expression ',')* expression )? ')'
    ;

// Assignment expressions are used to change the value of a variable
assign_expression
    : expression assignment_operator expression
    ;
assignment_operator
    : '=' | '+=' | '-=' | '*=' | '/=' | '%=' ;
```



# The Fearn Programming Language

```
/*
LEXER RULES

The following defines language-specific tokens, at aren't just simple words or punctuation, and
that have to match
a certain pattern, defined using regular expressions.
*/

// Define Fragments, used in the below token rules

// Digits
fragment D : [0-9] ;

// Letters
fragment L : [a-z]
            | [A-Z]
            | '_'
            ;

/* Define Tokens for Literals */
// Integers are one or more digits
INT_LIT    :  D+
            ;
// Floating-point numbers are two sets of 1 or more digits, separated by a dot
FLOAT_LIT  :  D+'.'D+
            ;
// A string is any sequence of characters, in between quotes, "" or ''
STR_LIT    :  '""(.)*?'"" | '\''(.)*?'\''
            ;
// A boolean value is a true or false value
BOOL_LIT   :  'true' | 'false'
            ;

// Identifiers (defined by the programmer) must be a letter, followed by 0 or
// more letters or digits
IDENTIFIER :  L(L|D)*
            ;

/* Ignore (certain) whitespace */
WS         :  ( ' ' | '\t' | '\n' | '\r' )+ -> skip
            ;

/* Ignore Comments */
BLOCKCOMMENT :  '/*' .*? '*/' -> skip
              ;
LINECOMMENT  :  '//' ~[\r\n]* -> skip
              ;
```

# The Fearn Programming Language

## parser.ASTConstructor

```
package parser;

import java.util.ArrayList;
import java.util.Stack;

import parser.gen.*;
import semantics.table.*;
import util.*;

import ast.*;
import ast.expression.*;
import ast.expression.AssignExpression.AssignmentOperator;
import ast.expression.Expression.ExprType;
import ast.function.Function;
import ast.function.Parameter;
import ast.type.ArraySpecifier;
import ast.type.PrimitiveSpecifier.PrimitiveDataType;
import codegen.ImportCompiler;
import ast.type.PrimitiveSpecifier;
import ast.type.StructInstanceSpecifier;
import ast.type.TypeSpecifier;
import ast.statement.*;
import ast.statement.JumpStatement.JumpType;

/* ASTConstructor.java
 *
 * This class is derived from the FearnGrammarBaseVisitor class,
 * generated by ANTLR from the grammar specified in FearnGrammar.g4
 *
 * The base class 'walks' (DFT) the parse tree recursively, at returns a
 * new AST Node at each parse tree node, after having walked its entire
 * subtree.
 *
 * The point of doing this, as opposed to just operating directly on the
 * ANTLR Parse Tree is:
 *
 * A) To eliminate redundant semantic information (e.g. punctuation, where
 *    brackets are used, etc.)
 *
 * B) To identify the structures used where it may be ambiguous (e.g. a dot expression
 *    could be used to access a struct's attribute, or could be a function call using
 *    universal function notation / UFN)
 *
 * C) To perform symbolic analysis, ensuring that every variable used has been declared.
 *    -> This doesn't perform all symbol analysis tasks (e.g. ensuring a
 *        function/struct is defined, but this is handled during the general
 *        validation traversal)
```

## The Fearn Programming Language

```
* D) To construct a symbol table, used to perform a full validation on the program and to
*     generate its implementation bytecode
*
* Each method defined below specifies actions taken when the traversal meets a particular
* production. The visit() function is used to traverse a child node of the current node.
*
* ANTLR Walkers use contexts (ctx) to give information about the form of the node. These can
* be used to
*
* A) Access a child node by name (e.g. ctx.expression() gives the child ctx of the child
*     expression node of the ctx)
*
* B) Access children by index
*
* C) Retrieve the number of children, the text representing the node (which matches the grammar
*     production) in the source file, or get the line number (by getting the line number of the
*     first token in the production rule match (the subtree of the production) )
*
*/
```

```
public class ASTConstructor extends FearnGrammarBaseVisitor<ASTNode> {
```

```
    /* SYMBOL ANALYSIS
```

```
    *
    * This is done using a stack
    * -> New identifiers for variables are added to the stack
    * -> They are popped of when they exit scope
    * -> Any variable references using names not in the stack
    *     raises an error
    */
```

```
    Stack<String> symbolAnalysisStack = new Stack<String>();
```

```
    /* PRIMARY EXPRESSIONS
```

```
    *
    * Fundamental, atomic expression
    * -> visitId_expr : visits references to variables
    *     -> Raises an error if the variable referenced isn't in the stack
    *         (hence, out of scope when it is used)
    * -> visitLiteral : visit literal values
    *
    * Both return PrimaryExpression objects, which are generic objects
    * when hold the data value they represent, or the string ID of the
    * referenced variable.
    *
    * The ExprType enum (part of the Expression class) is used to specify the
    * type of expression. Enums are used throughout the compiler for this
    * purpose (differentiating the purpose/function of AST Nodes).
    */
```

# The Fearn Programming Language

```
@Override
public Expression visitId_expr(FearnGrammarParser.Id_exprContext ctx)
{
    String id = ctx.getText();

    if (!symbolAnalysisStack.contains(id))
    {
        Reporter.ReportErrorAndExit(String.format(
            " Line %s : Variable identifier Unknown in Scope: %s .",
            ctx.getStart().getLine(),
            id
        ), null);
    }

    return new PrimaryExpression<String>(id, ExprType.VariableReference);
}

@Override
public Expression visitLiteral(FearnGrammarParser.LiteralContext ctx)
{
    switch (ctx.getStart().getType())
    {
        case FearnGrammarLexer.INT_LIT:
            return new PrimaryExpression<Integer>( Integer.valueOf(ctx.getText()),
ExprType.IntLiteral);
        case FearnGrammarLexer.BOOL_LIT:
            return new PrimaryExpression<Boolean>( Boolean.valueOf(ctx.getText()),
ExprType.BoolLiteral);

        case FearnGrammarLexer.FLOAT_LIT:
            // FearnLang Floats are represented by Java doubles
            return new PrimaryExpression<Double>(Double.valueOf(ctx.getText()),
ExprType.FloatLiteral);

        case FearnGrammarLexer.STR_LIT:
            return new PrimaryExpression<String>(ctx.getText().substring( 1,
ctx.getText().length() - 1), ExprType.StrLiteral);

        default:
            Reporter.ReportErrorAndExit("Parse Error: Unable to Parse literal value", null);
    }

    return null;
}
```

# The Fearn Programming Language

```
// Bracket expressions, ( expression ), are basically used to clarify precedence.
// Since this is natural as part of a tree, bracket expressions simply return
// whatever's in the brackets
@Override
public Expression visitBrac_expr(FearnGrammarParser.Brac_exprContext ctx)
    { return (Expression)visit(ctx.expression()); }

/* BINARY OPERATIONS
 *
 * These all have two operands, and all function (roughly) in the same way
 * 1) Visit both operands
 * 2) Perform any additional processing (e.g. setting the ExprType)
 * 3) Return an Binary Expression object
 *
 * ExprType is again used to specify the operation
 *
 */

// Multiplicative Expression (a (*|/|%) b )
@Override
public BinaryExpression visitMult_expr(FearnGrammarParser.Mult_exprContext ctx)
{
    Expression op1 = (Expression)visit(ctx.expression(0));
    Expression op2 = (Expression)visit(ctx.expression(1));

    ExprType type = null;

    switch (ctx.op.getText()) {
        case "*":
            type = ExprType.Mult;
            break;
        case "/":
            type = ExprType.Div;
            break;
        case "%":
            type = ExprType.Mod;
            break;
        default:
            break;
    }

    assert(type != null);

    return new BinaryExpression(op1, op2, type);
}
```

# The Fearn Programming Language

```
// Additive Expression (a (+|-) b )
@Override
public BinaryExpression visitAdd_expr(FearnGrammarParser.Add_exprContext ctx)
{
    Expression op1 = (Expression)visit(ctx.expression(0));
    Expression op2 = (Expression)visit(ctx.expression(1));

    ExprType type = null;

    switch (ctx.op.getText()) {
        case "+":
            type = ExprType.Add;
            break;
        case "-":
            type = ExprType.Sub;
            break;
        default:
            break;
    }

    assert(type != null);

    return new BinaryExpression(op1, op2, type);
}

// Exponential Expression (a ^ b)
@Override
public BinaryExpression visitExp_expr(FearnGrammarParser.Exp_exprContext ctx)
{
    Expression op1 = (Expression)visit(ctx.expression(0));
    Expression op2 = (Expression)visit(ctx.expression(1));

    return new BinaryExpression(op1, op2, ExprType.Exponent);
}

// Boolean Logical Expression
// <
@Override
public BinaryExpression visitLess_expr(FearnGrammarParser.Less_exprContext ctx)
{
    Expression op1 = (Expression)visit(ctx.expression(0));
    Expression op2 = (Expression)visit(ctx.expression(1));

    return new BinaryExpression(op1, op2, ExprType.Less);
}
```

## The Fearn Programming Language

```
// >
@Override
public BinaryExpression visitGreater_expr(FearnGrammarParser.Greater_exprContext ctx)
{
    Expression op1 = (Expression)visit(ctx.expression(0));
    Expression op2 = (Expression)visit(ctx.expression(1));

    return new BinaryExpression(op1, op2, ExprType.Greater);
}

// <=
@Override
public BinaryExpression visitLess_eq_expr(FearnGrammarParser.Less_eq_exprContext ctx)
{
    Expression op1 = (Expression)visit(ctx.expression(0));
    Expression op2 = (Expression)visit(ctx.expression(1));

    return new BinaryExpression(op1, op2, ExprType.LessEq);
}

// >=
@Override
public BinaryExpression visitGreater_eq_expr(FearnGrammarParser.Greater_eq_exprContext ctx)
{
    Expression op1 = (Expression)visit(ctx.expression(0));
    Expression op2 = (Expression)visit(ctx.expression(1));

    return new BinaryExpression(op1, op2, ExprType.GreaterEq);
}

// ==
@Override
public BinaryExpression visitEq_expr(FearnGrammarParser.Eq_exprContext ctx)
{
    Expression op1 = (Expression)visit(ctx.expression(0));
    Expression op2 = (Expression)visit(ctx.expression(1));

    return new BinaryExpression(op1, op2, ExprType.Eq);
}

// !=
@Override
public BinaryExpression visitNot_eq_expr(FearnGrammarParser.Not_eq_exprContext ctx)
{
    Expression op1 = (Expression)visit(ctx.expression(0));
    Expression op2 = (Expression)visit(ctx.expression(1));

    return new BinaryExpression(op1, op2, ExprType.NotEq);
}
```

# The Fearn Programming Language

```
// &&
@Override
public BinaryExpression visitAnd_expr(FearnGrammarParser.And_exprContext ctx)
{
    Expression op1 = (Expression)visit(ctx.expression(0));
    Expression op2 = (Expression)visit(ctx.expression(1));

    return new BinaryExpression(op1, op2, ExprType.LogicalAnd);
}

// ||
@Override
public BinaryExpression visitOr_expr(FearnGrammarParser.Or_exprContext ctx)
{
    Expression op1 = (Expression)visit(ctx.expression(0));
    Expression op2 = (Expression)visit(ctx.expression(1));

    return new BinaryExpression(op1, op2, ExprType.LogicalOr);
}

/* UNARY OPERATIONS (1 operand) */
// -a
@Override
public UnaryExpression visitU_minus_expr(FearnGrammarParser.U_minus_exprContext ctx)
{
    Expression op = (Expression)visit(ctx.expression());

    return new UnaryExpression(op, ExprType.Negate);
}

// !a
@Override
public UnaryExpression visitU_not_expr(FearnGrammarParser.U_not_exprContext ctx)
{
    Expression op = (Expression)visit(ctx.expression());

    return new UnaryExpression(op, ExprType.LogicalNot);
}

// Postfix increment : a-- | a++
@Override
public IncrExpression visitPost_inc_expr(FearnGrammarParser.Post_inc_exprContext ctx)
{
    // Uses the text representation of the operator to discern if the operation
    // in an increment or a decrement
    Boolean isDecrement = false;
    if(ctx.op.getText().equals("--"))
    {
```



## The Fearn Programming Language

```
        isDecrement = true;
    }

    // Last argument indicates if the expression is prefix (important for code generation
    // to perform operations in the right order)
    return new IncrExpression((Expression)visit(ctx.expression()), isDecrement, false);
}

@Override
public IncrExpression visitPre_inc_expr(FearnGrammarParser.Pre_inc_exprContext ctx)
{
    Boolean isDecrement = false;
    if(ctx.op.getText().equals("--"))
    {
        isDecrement = true;
    }

    return new IncrExpression((Expression)visit(ctx.expression()), isDecrement, true);
}

// Cast Expressions (Casting data to a primitive data type)
// Utilises the PrimitiveDataType enum (see PrimitiveExpression)
@Override
public CastExpression visitCast_expr(FearnGrammarParser.Cast_exprContext ctx)
{
    PrimitiveDataType targetType = null;

    switch (ctx.type_name().getText()) {
        case "int" : targetType = PrimitiveDataType.INT; break;
        case "float": targetType = PrimitiveDataType.FLOAT; break;
        case "str" : targetType = PrimitiveDataType.STR; break;
        case "bool" : targetType = PrimitiveDataType.BOOL; break;
        default: break;
    }

    return new CastExpression((Expression)visit(ctx.expression()), targetType);
}

// Index Expression ( e.g. myArray[0] )
@Override
public IndexExpression visitIndex_expr(FearnGrammarParser.Index_exprContext ctx)
{
    return new IndexExpression(
        (Expression)visit(ctx.expression(0)),
        (Expression)visit(ctx.expression(1))
    );
}
```

# The Fearn Programming Language

```
/* MISCELLANEOUS EXPRESSIONS */

// Function Calls
// -> Returns a FnCallExpression object, with the functions string identifier,
//      and expression arguments
@Override
public FnCallExpression visitFn_call_expr(FearnGrammarParser.Fn_call_exprContext ctx)
{
    String function_name = ctx.IDENTIFIER().getText();
    ArrayList<Expression> args = new ArrayList<Expression>();

    // Iterate through the child expressions (the arguments), visit them, and
    // add the returned Expression objects to args
    for (int i = 0; i < ctx.expression().size(); i++)
    {
        args.add((Expression)visit(ctx.expression(i)));
    }

    return new FnCallExpression(function_name, args);
}

// Dot Expressions (e.g. x.y ) /
@SuppressWarnings("rawtypes")
@Override
public Expression visitDot_expr(FearnGrammarParser.Dot_exprContext ctx)
{
    // Get the expression before and after the dot
    Expression predot = (Expression)visit(ctx.expression(0));
    Expression postdot = (Expression)visit(ctx.expression(1));
    /* EITHER:
    * A) postdot is a function call
    * B) postdot is an identifier (a variable reference)
    * C) Input in erroneous
    */
    if (postdot instanceof FnCallExpression)
    {
        // This matches a function call using universal function notation
        // e.g. a.equals(b) == equals(a, b)
        FnCallExpression UFN_call = (FnCallExpression)postdot;
        // Add the predot expression to the arguments
        UFN_call.arguments.add(0, predot);
        // Sets the is_UFN flag, which is used to print the function call
        // correctly, in case of error
        UFN_call.isUFN = true;

        return UFN_call;
    }
}
```

# The Fearn Programming Language

```
        else if (postdot instanceof PrimaryExpression && ((PrimaryExpression)postdot).type ==
ExprType.VariableReference)
        {
            // This matches an expression to access an attribute of a struct instance
            return new StructAttrExpression(predot, ctx.expression(1).getText());
        }
        else {
            // If neither above cases matched, the expression is illegal, and an error is raised
            Reporter.ReportErrorAndExit(String.format(
                "Line %s : %s.%s is not an attribute expression or a function call.",
                ctx.getStart().getLine(),
                predot.toString(),
                postdot.toString()
            ), null);
            return null;
        }
    }
}

// Array Initialisation (e.g. new int[] {1, 2, 3, 7, 123 } / new str[5])
@Override
public ArrayInitExpression visitArray_init(FearnGrammarParser.Array_initContext ctx)
{
    // Get type of the elements (either primitive or struct instances)
    TypeSpecifier type = (TypeSpecifier)visit(ctx.getChild(1));

    // Get numerical dimensions (if specified)
    ArrayList<Expression> dims = new ArrayList<Expression>();

    for (int i = 0; i < ctx.expression().size(); i++)
    {
        dims.add((Expression)visit(ctx.expression(i)));
    }

    // If an array body ( e.g. {1, 2, 3} ) is provided, visit the body
    ArrayBody init = null;

    if (ctx.array_body() != null)
    {
        init = (ArrayBody)visit(ctx.array_body());

        // If an array body is provided, dimensions are not present
        // Thus, dims is populated with the correct number of null values
        // `ctx.getChildCount() - 3` is needed, to remove from the count
        // the `new` keyword, type specifier, and arraybody - leaving the number
        // of `[]` used (the N of the N-dimensional array)
```

## The Fearn Programming Language

```
        for (int i = 0; i < ctx.getChildCount() - 3; i++) dims.add(null);
    }

    return new ArrayInitExpression(type, dims, init);
}

// ArrayBody (e.g. { 1 , 2 })
@Override
public Expression visitArray_body(FearnGrammarParser.Array_bodyContext ctx)
{
    ArrayList<Expression> elements = new ArrayList<Expression>();

    // Visit each element in the array, and add it to the elements
    for (int i = 1; i < ctx.getChildCount() - 1; i += 2)
    {

        elements.add((Expression)visit(ctx.getChild(i)));
    }

    return new ArrayBody(elements);
}

// Struct Initialisation ( e.g. new Person('Kenneth', 23) )
// Get struct ID, visit arguments (expressions), and return the StructInitExpression
@Override
public StructInitExpression visitStruct_init(FearnGrammarParser.Struct_initContext ctx)
{
    String struct_id = ctx.IDENTIFIER().getText();

    ArrayList<Expression> args = new ArrayList<Expression>();

    for (int i = 0; i < ctx.expression().size(); i++)
    {
        args.add((Expression)visit(ctx.expression(i)));
    }

    return new StructInitExpression(struct_id, args);
}

/* Assignment Expressions (e.g. x = y; x += 3; y %= 2) */
@Override
public AssignExpression visitAssign_expression(FearnGrammarParser.Assign_expressionContext
ctx)
{

```

## The Fearn Programming Language

```
// Visit target (what you're assigning a value to), and expression (the new value)
Expression target = (Expression)visit(ctx.expression(0));
Expression expression = (Expression)visit(ctx.expression(1));

// Set operator (an enum value - see AssignmentExpression)
AssignmentOperator op = null;

switch (ctx.assignment_operator().getText()) {
    case "="      : op = AssignmentOperator.Equals      ; break;
    case "+="     : op = AssignmentOperator.AddEquals   ; break;
    case "-="     : op = AssignmentOperator.SubEquals   ; break;
    case "*="     : op = AssignmentOperator.MultEquals  ; break;
    case "/="     : op = AssignmentOperator.DivEquals   ; break;
    case "%="     : op = AssignmentOperator.ModEquals   ; break;
    default: break;
}

return new AssignExpression(target, expression, op);
}

/* TYPE SPECIFIERS
 *
 * These, syntactically, are used to indicate type.
 *
 * There are three sorts of type specifiers:
 * -> Primitive: One of the 4 simple data types: int, float, str, bool
 * -> Struct: A struct instance, shown as `IDENTIFIER`, where IDENTIFIER
 * indicates the struct used
 * -> Array: An N-dimensional array, that stores primitives or struct instances
 *
 */

public PrimitiveSpecifier
visitType_specifier_primitive(FearnGrammarParser.Type_specifier_primitiveContext ctx)
{
    PrimitiveDataType type = null;

    switch (ctx.getText()) {
        case "int"   : type = PrimitiveDataType.INT;   break;
        case "float": type = PrimitiveDataType.FLOAT; break;
        case "str"   : type = PrimitiveDataType.STR;   break;
        case "bool"  : type = PrimitiveDataType.BOOL;  break;
    }

    return new PrimitiveSpecifier(type);
}
```

## The Fearn Programming Language

```
}

public ArraySpecifier visitType_specifier_arr(FearnGrammarParser.Type_specifier_arrContext
ctx)
{
    TypeSpecifier type = (TypeSpecifier)visit(ctx.getChild(0));
    Integer dims = ctx.getChildCount() - 1;
    return new ArraySpecifier(type, dims);
}

public StructInstanceSpecifier
visitType_specifier_struct(FearnGrammarParser.Type_specifier_structContext ctx)
{
    return new StructInstanceSpecifier(ctx.IDENTIFIER().getText());
}

/* DECLARATIONS
 *
 * New variables are declared at the start of the compound statement in
 * which they're in scope. This function...
 * 1) Gets the identifier for the variable, its type specifier, and the
 *    initialisation expression, if present.
 * 2) Adds the identifier to the stack
 * 3) Adds a row for the variable to the local/global symbol table (this
 *    will throw an error if another variable of the same name exists in
 *    scope), by adding the row to the SymbolTable at the top of the
 *    symbolTableStack
 * 4) Returns a Declaration object
 *
 */

public Declaration visitDeclaration(FearnGrammarParser.DeclarationContext ctx)
{
    String identifier = ctx.IDENTIFIER().getText();
    symbolAnalysisStack.push(identifier);
    TypeSpecifier type_spec = (TypeSpecifier)visit(ctx.type_specifier());

    Expression init_expression = null;

    if (ctx.getChildCount() > 5)
    {
        init_expression = (Expression)visit(ctx.expression());
    }
}
```

## The Fearn Programming Language

```
symbolTableStack.peek().addRow(
    new VariableRow(
        identifier,
        type_spec
    )
);

return new Declaration(identifier, type_spec, init_expression);

}

/* STATEMENTS */

/* COMPOUND STATEMENTS
 *
 * A collection of statements, surrounded by curly braces.
 *
 * This function:
 * 1) Records the number of symbols in the stack initially
 * 2) Visits all nested declarations and statements
 * 4) Removes any new symbols added to the stack during the traversal
 *    of its subtree (by popping the stack for the difference between
 *    the new size of the stack, and the initial number). This is done
 *    because said variables declared within the statement are out of
 *    scope beyond it.
 */
@Override
public CompoundStatement visitCompound_statement(FearnGrammarParser.Compound_statementContext
ctx)
{
    int numOfSyms = symbolAnalysisStack.size();
    ArrayList<ASTNode> local_nodes = new ArrayList<ASTNode>();

    for (int i = 1; i < ctx.getChildCount() - 1; i++)
        local_nodes.add(visit(ctx.getChild(i)));

    // Remove Symbols added within the compound statement
    for (int i = 0; i < symbolAnalysisStack.size() - numOfSyms; i++)
        symbolAnalysisStack.pop();

    return new CompoundStatement(local_nodes);
}
```

## The Fearn Programming Language

```
/* EXPRESSION STATEMENTS
 *
 * Simply a simple expression or assignment expression. The expressions are visited, and
 * the resulting Expression object is used to return an ExpressionStatement.
 *
 * The boolean flag at the end indicates the ExpressionStatement is an assignment expression
 * if true.
 */
@Override
public ExpressionStatement visitSimple_expr_stmt(FearnGrammarParser.Simple_expr_stmtContext
ctx)
{
    return new ExpressionStatement((Expression)visit(ctx.expression()), false);
}

@Override
public ExpressionStatement visitAssign_expr_stmt(FearnGrammarParser.Assign_expr_stmtContext
ctx)
{
    return new ExpressionStatement((Expression)visit(ctx.assign_expression()), true);
}

/* SELECTION STATEMENTS
 *
 * Statements used for branching (if and if-else)
 *
 * Each of the below returns a SelectionStatement, which takes three arguments:
 * the condition expression, the if branch body (a compound statement), and
 * the else branch statement (either a compound statement, or another selection
 * statement - to build a if-else chain)
 *
 */
@Override
public SelectionStatement visitSingle_if(FearnGrammarParser.Single_ifContext ctx)
{
    return new SelectionStatement(
        (Expression)visit(ctx.expression()),
        (CompoundStatement)visit(ctx.compound_statement()),
        null
    );
}
```



# The Fearn Programming Language

```
@Override
public SelectionStatement visitIf_else(FearnGrammarParser.If_elseContext ctx)
{
    return new SelectionStatement(
        (Expression)visit(ctx.expression()),
        (CompoundStatement)visit(ctx.compound_statement(0)),
        (Statement)visit(ctx.getChild(ctx.getChildCount() - 1))
    );
}

/* ITERATION STATEMENTS
 *
 * Statements used for loops (for loops, to be precise - Fearn doesn't
 * support while loop, but the syntax for for loops is permissive enough
 * for it to be used as a while loop)
 *
 * This function:
 * 1) Visits the initialisation expression/declaration (run before the loop),
 *    if present
 * 2) Visits the continue expression (a boolean expression that must evaluate
 *    to true for the loop body to run)
 * 3) Visits the iteration expression, run at the end of each iteration, if
 *    present
 * 4) Visits the compound statement, that is the body of the loop
 * 5) Returns an IterationStatement
 *
 */
@Override
public IterationStatement
visitIteration_statement(FearnGrammarParser.Iteration_statementContext ctx)
{
    ASTNode init = null;
    Expression cont_expr = null;
    Expression iter_expr = null;
    CompoundStatement body = null;
    if (!ctx.init_expression().getText().equals(";"))
        // Either an Expression or a Declaration
        init = visit(ctx.init_expression().getChild(0));
    if (!ctx.continue_condition().getText().equals(";"))
        cont_expr = (Expression)visit(ctx.continue_condition().getChild(0));
    if (ctx.iteration_expression().getChildCount() > 0)
        iter_expr = (Expression)visit(ctx.iteration_expression().getChild(0));
    body = (CompoundStatement)visit(ctx.compound_statement());
    return new IterationStatement(init, cont_expr, iter_expr, body);
}
```

# The Fearn Programming Language

```
/* JUMP STATEMENTS
 *
 * Statements used to jump around a program (return, break, and continue)
 *
 * Each one returns a JumpStatement, with return statements returning the
 * subclass ReturnStatement, which allows an expression for the statement to
 * return to be passed.
 *
 * Jump Statements use the enum JumpType to differentiate themselves.
 */

@Override
public JumpStatement visitCont_stmt(FearnGrammarParser.Cont_stmtContext ctx)
{ return new JumpStatement(JumpType.Continue); }

@Override
public JumpStatement visitBreak_stmt(FearnGrammarParser.Break_stmtContext ctx)
{ return new JumpStatement(JumpType.Break); }

@Override
public ReturnStatement visitReturn_stmt(FearnGrammarParser.Return_stmtContext ctx)
{
    return new ReturnStatement(
        JumpType.Return,
        (ctx.expression() == null) ? null : (Expression)visit(ctx.expression())
    );
}

/* HIGH-LEVEL STRUCTURES */
/* FUNCTION DEFINITIONS
 *
 * In the form `fn IDENTIFIER(parameter*) => return_type_specifier`
 *
 * This function:
 * 1) Determines if the function returns void, by checking for a type_specifier
 * 2) Visits the return type specifier if present
 * 3) Visits the parameters, adding them to a list. These are added to the
 *    symbol table before any local variables within the function, as the JVM
 *    adds arguments in the first indexes of the local variable list. Thus, since
 *    this table is used to get variable indexes during code generation, arguments
 *    have to be first.
 * 4) Visits the body
 * 5) Pops the arguments from the symbol stack (they are added when the parameters
 *    are visited)
```

## The Fearn Programming Language

```
* 6) Returns a Function object, retrieving the string identifier from the IDENTIFIER
* token
*/
@Override
public Function visitFunction(FearnGrammarParser.FunctionContext ctx)
{
    ArrayList<Parameter> parameters = new ArrayList<Parameter>();
    TypeSpecifier return_type = null;
    Boolean is_void = false;

    if ( ctx.type_specifier() == null ) {
        is_void = true;
    } else {
        return_type = (TypeSpecifier)visit(ctx.type_specifier());
    }

    for (int i = 0; i < ctx.parameter().size(); i++)
    {
        // Visit Parameter
        Parameter param = (Parameter)visit(ctx.parameter(i));

        // Add parameter to symbol table
        symbolTableStack.peek().addRow(
            new VariableRow(
                param.identifier,
                param.type
            )
        );

        // Add to parameters
        parameters.add(param);
    }

    CompoundStatement body = (CompoundStatement)visit(ctx.compound_statement());

    for (int i = 0; i < parameters.size(); i++) symbolAnalysisStack.pop();

    return new Function(ctx.IDENTIFIER().getText(), parameters, return_type, is_void, body);
}
```

## The Fearn Programming Language

```
/*  
Parameter, in the form `IDENTIFIER : type_specifier`.
```

The function visits the `type_specifier`, gets the string identifier, and adds the parameter to the symbol stack, to ensure references to the parameters are detected as valid during traversal of the function body.

```
*/  
  
@Override  
public Parameter visitParameter(FearnGrammarParser.ParameterContext ctx)  
{  
    symbolAnalysisStack.push(ctx.IDENTIFIER().getText());  
    return new Parameter(  
        ctx.IDENTIFIER().getText(),  
        (TypeSpecifier)visit(ctx.type_specifier())  
    );  
}
```

```
/* STRUCT DEFINITIONS  
*  
* In the form `struct IDENTIFIER {declaration*}`  
*  
* This function:  
* 1) Visits all the declarations within the struct (its attributes)  
* 2) Gets the identifier string  
* 3) Returns a Struct object  
*/
```

```
@Override  
public Struct visitStruct_def(FearnGrammarParser.Struct_defContext ctx)  
{  
    ArrayList<Declaration> decl = new ArrayList<Declaration>();  
    for (int i = 0; i < ctx.declaration().size(); i++)  
    {  
        decl.add((Declaration)visit(ctx.declaration(i)));  
    }  
  
    return new Struct(ctx.IDENTIFIER().getText(), decl);  
}
```

## The Fearn Programming Language

```
/* IMPORTS
 *
 * An Import Compiler is used to compile other Fearn programs,
 * imported by the current program, and returns their symbol table,
 * the rows of which are added to the current global symbol table.
 *
 * If an identifier is used, the import is from a standard library
 * module (e.g. io). The Import Compiler will construct a symbol table
 * for the functions contained within, and add them to the global symbol
 * table, so they can be used anywhere within the program.
 */
@Override
public ASTNode visitModule_import(FearnGrammarParser.Module_importContext ctx)
{
    ImportCompiler comp = new ImportCompiler();

    if (ctx.IDENTIFIER() == null)
    {
        symbolTableStack.peek().addRowsFromTable(
            comp.Compile(ctx.STR_LIT().toString())
        );
    } else {
        symbolTableStack.peek().addRowsFromTable(
            comp.GetStdLib(ctx.IDENTIFIER().toString())
        );
    }

    // All new symbols from the import are added to the
    // symbol stack
    for (Row row : symbolTableStack.peek().GetAllRows())
    {
        if (row instanceof VariableRow)
        {
            symbolAnalysisStack.push(row.identifier);
        } else if (row instanceof StructRow)
        {
            for (Row var_row : ((StructRow)row).localSymbolTable.GetAllRows())
            {
                symbolAnalysisStack.push(var_row.identifier);
            }
        }
    }

    return null;
}
```

## The Fearn Programming Language

```
/* The symbol table stack is used to store symbol tables for
 * functions, structs, and the global scope. When new rows are
 * created, they are added to the table at the top of the stack.
 * These can then be popped off to be used, say within a FunctionRow
 * or StructRow, stored within the global symbol table. This means rows
 * are always added to the symbol table for the part of the program
 * currently being traversed.
 */

public Stack<SymbolTable> symbolTableStack = new Stack<SymbolTable>();

/* Program (root of AST)
 *
 * This visits all imports, global declarations, structs, and functions.
 *
 * For structs and functions, a new symbol table is added to the stack.
 * This means newly-declared symbols are added to this symbol table. It
 * is then popped off the stack, and stored within a row for the larger
 * structure, which is finally stored in the global symbol table (always
 * at the bottom of the stack).
 *
 * The function returns a Program object, representing the root of the tree.
 */

@Override
public Program visitProgram(FearnGrammarParser.ProgramContext ctx)
{
    ArrayList<Declaration> global_declarations = new ArrayList<Declaration>();
    ArrayList<Struct> structs = new ArrayList<Struct>();
    ArrayList<Function> functions = new ArrayList<Function>();

    symbolTableStack.add(new SymbolTable());

    for (int i = 0; i < ctx.module_import().size(); i++) visit(ctx.module_import(i));

    for (int i = 0; i < ctx.declaration().size(); i++)
    {
        Declaration decl = visitDeclaration(ctx.declaration(i));
        global_declarations.add(decl);
    }
}
```

## The Fearn Programming Language

```
for (int i = 0; i < ctx.struct_def().size(); i++)
{
    symbolTableStack.add(new SymbolTable());

    Struct struct = visitStruct_def(ctx.struct_def(i));
    structs.add(struct);

    SymbolTable local_syms = symbolTableStack.pop();

    symbolTableStack.peak().addRow(
        new StructRow(struct.identifier, local_syms)
    );
}

for (int i = 0; i < ctx.function().size(); i++)
{
    symbolTableStack.add(new SymbolTable());

    Function func = visitFunction(ctx.function(i));
    functions.add(func);

    SymbolTable local_syms = symbolTableStack.pop();

    symbolTableStack.peak().addRow(
        new FunctionRow(func.identifier, func.parameters, func.return_type, local_syms)
    );
}

return new Program(global_declarations, functions, structs);
}

}
```

# The Fearn Programming Language

## ast

### ast.ASTNode

```
package ast;

import static org.assertj.core.api.Assertions.*;

/* ASTNode.java
 *
 * This class is the super class for every AST class, for nodes which
 * appear in the Abstract Syntax Tree.
 *
 * These are the classes responsible for representing a program,
 * validating it as as following Fearn's semantic rules, and generating
 * bytecode for their logic when they're used.
 *
 * This is an abstract class, and includes
 * -> equals(): A method to compare ASTNode classes, often used to compare
 *     TypeSpecifier object to validate data types. It uses AssertJ's
 *     recursive assertion function, which compares an object by comparing
 *     their attributes, rather than addresses in memory. If the assertion
 *     that two ASTNodes are equal fails, the exception is caught and false
 *     is returned. Otherwise, method returns true.
 * -> toString(): An abstract method to return a string representation of a
 *     node. This is used when reporting errors, to show the offending source
 *     code.
 *     -> It was also heavily used during debugging to ensure programs
 *         where parse correctly, and the AST was being properly constructed.
 */

public abstract class ASTNode {
    @Override
    public boolean equals(Object o) {
        try {
            assertThat(this).usingRecursiveComparison().isEqualTo(o);
        } catch (AssertionError e) {
            return false;
        }
        return true;
    }
    public abstract String toString();
}
```



# The Fearn Programming Language

## ast.Declaration

```
package ast;

import static org.objectweb.asm.Opcodes.ASTORE;

import org.objectweb.asm.MethodVisitor;

import ast.expression.Expression;
import ast.type.TypeSpecifier;
import codegen.CodeGenerator;
import semantics.table.SymbolTable;
import util.Reporter;

/* Declaration.java
 *
 * Represents a Variable Declaration in the AST.
 *
 * Fields:
 * -> identifier: the string identifier of the variable, in the source code
 * -> type: TypeSpecifier for the datatype of the variable
 * -> init_expression: The expression used to initialise the variable (may be null)
 */

public class Declaration extends ASTNode {

    public String identifier;
    public TypeSpecifier type;
    public Expression init_expression;

    // Standard Constructor
    public Declaration(String id, TypeSpecifier t, Expression e)
    {
        identifier = id;
        type = t;
        init_expression = e;
    }
}
```

## The Fearn Programming Language

```
// String representation takes the same form as a declaration in source code
// It reclusively calls the toString methods for the type, and init expression
// The ternary expression only includes the init_expression if it is not null
@Override public String toString()
{
    return (init_expression == null) ?
        "let " + identifier + " : "+ type.toString() + ";"
        : "let " + identifier + " : "+type.toString()+" = "+init_expression.toString()+";" ;
}

// GenerateBytecode() generates the init_expression bytecode (leaving its value
// at the top of the operand stack), and then stores it at the variable index
// indicated by the LocalSymbolTable.
// -> This is ONLY called to generate local variables within functions (globals
//      are handled in CodeGenerator.java), so no need to handle global variables
public void GenerateBytecode(MethodVisitor mv)
{
    if (init_expression != null) {
        init_expression.GenerateBytecode(mv);
        mv.visitVarInsn(ASTORE, CodeGenerator.LocalSymbolTable.GetIndex(identifier));
    }
}

// validate() compares the expression_type of the init_expression to the type of the variable
// If they don't match, an error is raised. Otherwise, declaration is valid and the method
// returns. The method will return immediately if init_expression is null, as a declaration
// without initialisation cannot be invalid (at this stage, errors such as two variables in
// the same function having the same name are caught by the SymbolTable during AST
// Construction)
public void validate(SymbolTable symbolTable) {
    if (init_expression == null) return;
    TypeSpecifier exprType = init_expression.validate(symbolTable);
    if (!type.equals(exprType))
        Reporter.ReportErrorAndExit(
            "Cannot assign " + exprType.toString() + " to " + type.toString(),
            this
        );
}
}
```

# The Fearn Programming Language

## ast.Program

```
package ast;

import java.util.ArrayList;
import ast.function.Function;
import semantics.table.SymbolTable;
/* Program.java
 *
 * The root node of the AST.
 * -> It contains global declarations, structs, and functions.
 * -> Its validate() method simply calls the validate() methods for
 *     its children
 * -> Function validations are have their local symbol table
 *     passed to them, from the global symbol table
 */
public class Program extends ASTNode {
    public ArrayList<Declaration> global_declarations;
    public ArrayList<Struct> structs;
    public ArrayList<Function> functions;

    public Program(
        ArrayList<Declaration> global_decl,
        ArrayList<Function> funcs,
        ArrayList<Struct> s
    ) {
        global_declarations = global_decl;
        functions = funcs;
        structs = s;
    }

    @Override
    public String toString()
    {
        return String.format("%s%s%s", global_declarations, structs, functions);
    }

    public void validate(SymbolTable symbolTable) {
        for (Declaration d : global_declarations) d.validate(symbolTable);
        for (Struct s : structs) s.validate(symbolTable);
        for (Function f : functions) f.validate(
            symbolTable.GetFuncSymbolTable(f.identifier)
        );
    }
}
```

# The Fearn Programming Language

## ast.Struct

```
package ast;

import java.util.ArrayList;
import semantics.table.SymbolTable;
import util.Reporter;

/* Struct.java
 *
 * Represents struct definitions in the AST.
 * -> It contains the struct's identifier, and attributes (as declarations)
 * -> It's validate method checks the the user hasn't attempted to initialise
 *     an attribute to a default value
 *     -> This is raised as invalid, simply because, when a struct is
 *         instantiated, the user must define the initial value of every
 *         attribute of that instance
 */

public class Struct extends ASTNode {
    public String identifier;
    public ArrayList<Declaration> declarations;
    public Struct(String id, ArrayList<Declaration> decl)
    {
        identifier = id;
        declarations = decl;
    }
    @Override
    public String toString()
    {
        return String.format("struct %s {...}", identifier);
    }
    public void validate(SymbolTable symbolTable) {
        for (Declaration decl : declarations)
        {
            if (decl.init_expression != null)
            {
                Reporter.ReportErrorAndExit(
                    decl.toString() + ": Cannot assign default values to struct attributes.",
                    null
                );
            }
        }
    }
}
```

# The Fearn Programming Language

## ast.expression

ast.expression.Expression

```
package ast.expression;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import ast.ASTNode;
```

```
import ast.type.*;
```

```
import semantics.table.SymbolTable;
```

```
/* Expression.java
```

```
*
```

```
* The abstract class that represents all expressions that exist in Fearn.
```

```
*
```

```
* Fields:
```

```
* -> ExprType: Enum of expression types, used to tell certain expression objects apart.
```

```
* -> toString(): Abstract method, that returns a string representation of the node, in  
* the syntax of Fearn
```

```
* -> validate(): Method that validates an expression, and its sub-expressions, as being  
* syntactically in and of themselves (not considering the context in which they are  
* used)
```

```
* -> Unlike Statement.validate(), this returns a TypeSpecifier, representing  
* the data type of the data the expression evaluates to (e.g. 1 + 2 evaluates  
* to an int)
```

```
* -> This is vital to ensure the types of sub-expressions are valid  
* (e.g. "Hello" + 1 is an invalid expression, because one operand is a str,  
* and the other an int)
```

```
* -> GenerateBytecode(): Method that generates the bytecode that will leave the evaluated  
* value of the expression on the top of the operand stack.
```

```
* -> This is always left as the Object version of the basic primitive type, to  
* restrict the number of unique instructions the compiler needs to use (JVM instruction  
* are often typed, but objects only use one set)
```

```
* -> expression_type : The TypeSpecifier of the data type an expression evaluates to. It  
* is set during validation, so nodes don't need to be repeatedly re-validated - and  
* the type can be accessed during code generation.
```

```
*/
```

# The Fearn Programming Language

```
public abstract class Expression extends ASTNode {

    public static enum ExprType
    {
        // MISC
        FuncCall, StructAttribute, TypeCast, StructInit, ArrayInit, Index,

        // PRIMARY
        VariableReference, IntLiteral, FloatLiteral, BoolLiteral, StrLiteral,

        // UNARY
        Negate, LogicalNot, Increment,

        // BINARY
        Eq, NotEq,
        Less, Greater,
        LessEq, GreaterEq,
        LogicalAnd, LogicalOr,
        Exponent, Mult, Div, Mod, Add, Sub
    }

    @Override public abstract String toString();

    public abstract void GenerateBytecode(MethodVisitor mv);
    public abstract TypeSpecifier validate(SymbolTable symTable);

    public TypeSpecifier expression_type;
}
```

# The Fearn Programming Language

ast.expression.ArrayBody

```
package ast.expression;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import static org.objectweb.asm.Opcodes.*;
```

```
import java.util.ArrayList;
```

```
import ast.type.ArrayBodySpecifier;
```

```
import ast.type.TypeSpecifier;
```

```
import semantics.table.SymbolTable;
```

```
import util.Reporter;
```

```
/* ArrayBody.java
```

```
 *
```

```
 * Represents an ArrayBody in the AST (e.g. {1, 2, 3}).
```

```
 *
```

```
 * Fields:
```

```
 * -> elements : The body's elements, modelled as a list of expressions.
```

```
 *
```

```
 */
```

```
public class ArrayBody extends Expression {
```

```
    public ArrayList<Expression> elements;
```

```
    public ArrayBody(ArrayList<Expression> ele)
```

```
    {
```

```
        elements = ele;
```

```
    }
```

```
    @Override
```

```
    public String toString()
```

```
    {
```

```
        return elements.toString().replace("[", "{").replace("]", "}");
```

```
    }
```

```
    /* To generate bytecode for an ArrayBody, the method...
```

```
    * 1) Gets a descriptor of the body's elements from the SymbolTable
```

```
    * 2) If the body is 1-D, the "L" and ";" ate the front and end of the descriptor  
    *     are removed, to make it a class name
```

```
    * 3) Push the array length to the stack, and use ANEWARRAY to create an empty  
    *     array, of that size, on the top of the stack
```

## The Fearn Programming Language

```
* 4) For each element in the body ...
*     4.1) Duplicate the array (as the AASTORE instruction will pop the array
*           from the stack, and we still have more element to assign)
*     4.2) Push the index to store the element (starting at 0 and incrementing
*           with each element) to the stack
*     4.3) Generate the expression to store at that index (GenerateBytecode()
*           on expressions leaves its value on top of the stack)
*     4.4) Use AASTORE to take the expression value, index, and array, and store
*           the element - at the index - in the array
*/
@Override
public void GenerateBytecode(MethodVisitor mv) {

    String desc = SymbolTable.GenBasicDescriptor(elements.get(0).expression_type);

    if (elements.get(0).getClass() != ArrayBody.class) // Array is 1-D
    {
        desc = desc.substring(1, desc.length() - 1);
        // This would transform "Ljava\lang\Integer;" to "java\lang\Integer"
        // (a class name)
    }

    mv.visitIntInsn(SIPUSH, elements.size());
    mv.visitTypeInsn(ANEWARRAY, desc);

    int i = 0;
    for (Expression e : elements)
    {
        mv.visitInsn(DUP);
        mv.visitIntInsn(SIPUSH, i++);
        e.GenerateBytecode(mv);
        mv.visitInsn(AASTORE);
    }

}

/* To validate an ArrayBody, the method...
* 1) Gets the TypeSpecifier of the first element
* 2) Check the rest of the elements have the same TypeSpecifier
*    -> Raise Error otherwise
* 3) If the body contains other array bodies (N-D array), add the dimensions of the
*     child
*     bodies to the end of this body's dimensions
* 4) Set expression_type to an ArrayBodySpecifier, and return it
*/
```



# The Fearn Programming Language

```
@Override
public TypeSpecifier validate(SymbolTable symTable) {

    TypeSpecifier element_type = elements.get(0).validate(symTable);

    for (Expression e : elements.subList(1, elements.size() ))
    {
        TypeSpecifier e_type = e.validate(symTable);

        if (!element_type.equals(e_type)) Reporter.ReportErrorAndExit(
            "ArrayBody has inconsistent element type.",
            this
        );
    }

    ArrayList<Integer> dimensions = new ArrayList<Integer>();
    dimensions.add(elements.size());

    if (element_type.getClass() == ArrayBodySpecifier.class)
    {
        dimensions.addAll(((ArrayBodySpecifier)element_type).dimensions);
    }

    expression_type = new ArrayBodySpecifier(element_type, dimensions);
    return expression_type;
}
}
```

# The Fearn Programming Language

ast.expression.ArrayInitExpression

```
package ast.expression;
```

```
import java.util.ArrayList;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import static org.objectweb.asm.Opcodes.*;
```

```
import ast.type.ArrayBodySpecifier;
```

```
import ast.type.ArraySpecifier;
```

```
import ast.type.PrimitiveSpecifier.PrimitiveDataType;
```

```
import ast.type.PrimitiveSpecifier;
```

```
import ast.type.TypeSpecifier;
```

```
import semantics.table.SymbolTable;
```

```
import util.Reporter;
```

```
/* ArrayInitExpression.java
```

```
 *
 * Represents an Array Initialisation in the AST (e.g. new int[5]).
 *
```

```
 * Fields:
```

```
 * -> type : TypeSpecifier for element types
```

```
 * -> dimensions: The expressions for the array dimensions (can be null if body provided)
```

```
 * -> init_body: The body used to initialise the array (can be null if dimensions provided)
```

```
 */
```

```
public class ArrayInitExpression extends Expression {
```

```
    public TypeSpecifier type;
```

```
    public ArrayList<Expression> dimensions;
```

```
    public ArrayBody init_body;
```

```
    public ArrayInitExpression(TypeSpecifier t, ArrayList<Expression> dims, ArrayBody ele)
```

```
    {
```

```
        type = t;
```

```
        dimensions = dims;
```

```
        init_body = ele;
```

```
    }
```

## The Fearn Programming Language

```
@Override
public String toString()
{
    String s = type.toString();
    if (dimensions.get(0) == null)
    {
        s += "[".repeat(dimensions.size());
    } else {
        for (Expression dim : dimensions)
        {
            s += '[' + dim.toString() + ']';
        }
    }

    if (init_body != null) s += init_body.toString();

    return s;
}

/* To generate bytecode for an Array Initialisation, the method...
 * 1) If a body has been provided, just generate that
 * 2) Otherwise, Generate the bytecode for each dimension, casting the values to
 *    primitive 'I'
 * 3) If the array is multidimensional, generate an array descriptor using
 *    SymbolTable, and use visitMultiANewArray, with the number of dimensions
 * 4) Otherwise, Use a new array, with the descriptor for an element
 */
@Override
public void GenerateBytecode(MethodVisitor mv) {

    // Just Generate the Body, if provided
    if (init_body != null)
    {
        init_body.GenerateBytecode(mv);
        return;
    }
}
```

## The Fearn Programming Language

```
// Otherwise
else
{
    for (Expression dim : dimensions)
    {
        dim.GenerateBytecode(mv);
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Integer", "intValue", "()I", false);
    }

    if (dimensions.size() > 1)
    {
        String desc = SymbolTable.GenBasicDescriptor(
            new ArraySpecifier(type, dimensions.size())
        );
        mv.visitMultiANewArrayInsn(desc, dimensions.size());
        return;
    } else {
        String desc = SymbolTable.GenBasicDescriptor(type);
        desc = desc.substring(1, desc.length() - 1 );
        mv.visitTypeInsn(ANEWARRAY, desc);
    }
}
}

/* To validate an Array Initialisation, the method...
* 1) If a body has been provided...
*     a) Validate the body (this ensures the types of elements in the body
*         is consistent)
*     b) Check the type of elements in the body matches that of the array
*         -> This is done by repeatedly accessing the element type of the body
*             specifier, repeating for each dimension (e.g. for a 2D array, this
*             runs twice), until the type of the actual elements is reached
*         -> Raise error otherwise
*     c) Check the dimensions are the same (a 3D body is used for a 3D array)
*     d) Set expression_type to an ArraySpecifier
* 2) Otherwise...
*     a) For each dimension, validate that they're of type int (raise error
*         otherwise)
*     b) Set expression_type to an ArraySpecifier
* 3) Return expression_type
*/
```

# The Fearn Programming Language

```
@Override
public TypeSpecifier validate(SymbolTable symTable) {

    if (init_body != null)
    {
        // Check the Elements are of the same type
        ArrayBodySpecifier bodySpecifier = (ArrayBodySpecifier)init_body.validate(symTable);

        TypeSpecifier typeOfElements = bodySpecifier;
        for (int i = 0; i < bodySpecifier.dimensions.size(); i++)
        {
            typeOfElements = ((ArrayBodySpecifier)typeOfElements).element_type;
        }

        if (!typeOfElements.equals(type))
            Reporter.ReportErrorAndExit(
                "Type of Elements in Array Body don't match the element type of the Array.",
                this
            );

        if (bodySpecifier.dimensions.size() != dimensions.size())
            Reporter.ReportErrorAndExit(
                "Dimensions of Array Body don't match dimensions of Array Initialisation.",
                this
            );

        expression_type = new ArraySpecifier(type, bodySpecifier.dimensions.size());
    } else {

        for (Expression e: dimensions)
        {
            TypeSpecifier dim_type = e.validate(symTable);
            if (!dim_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT)))
            {
                Reporter.ReportErrorAndExit("Dimensions of arrays must be of type int.",
                this);
            }
        }

        expression_type = new ArraySpecifier(type, dimensions.size());
    }
    // Return ArraySpecifier
    return expression_type;
}
```

# The Fearn Programming Language

```
ast.expression.AssignExpression
```

```
package ast.expression;
```

```
import static org.objectweb.asm.Opcodes.*;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import ast.type.TypeSpecifier;
```

```
import codegen.CodeGenerator;
```

```
import semantics.table.SymbolTable;
```

```
import util.Reporter;
```

```
/* AssignExpression.java
```

```
 *
```

```
 * Represents an Assignment in the AST.
```

```
 *
```

```
 * Fields:
```

```
 * -> AssignmentOperator (& operator): Enum that indicates the sort of assignment
```

```
 * -> target: The expression to be assigned to
```

```
 * -> expression: The value to assign to the target
```

```
 */
```

```
public class AssignExpression extends Expression {
```

```
    public static enum AssignmentOperator {
```

```
        Equals,
```

```
        AddEquals,
```

```
        SubEquals,
```

```
        MultEquals,
```

```
        DivEquals,
```

```
        ModEquals
```

```
    }
```

```
    public AssignmentOperator operator;
```

```
    public Expression target;
```

```
    public Expression expression;
```

```
    public AssignExpression(Expression t, Expression e, AssignmentOperator op)
```

```
    {
```

## The Fearn Programming Language

```
target = t;
expression = e;
operator = op;
}

@Override
public String toString()
{
    String opString = null;

    switch (operator) {
        case Equals      : opString = "="      ; break;
        case AddEquals   : opString = "+="     ; break;
        case SubEquals   : opString = "-="     ; break;
        case MultEquals  : opString = "*="     ; break;
        case DivEquals   : opString = "/="     ; break;
        case ModEquals   : opString = "%="     ; break;
    }

    return target.toString() + " " + opString + " " + expression.toString();
}

/* The Target Expression can be one of the following
 * -> A variable reference          (ASTORE at <INDEX> / PUTSTATIC if Global)
 * -> An Index Expression          (Iterative Loading of the array, then AASTORE)
 * -> A Struct Attribute Expression (PUTFIELD)
 */

/* To generate bytecode for an assignment...
 * 1) If assigning to a variable, generate the expression, then...
 *     a) If the variable is local, use ASTORE, with the index of the variable from
 *         the LocalSymbolTable
 *     b) If global, use PUTSTATIC, with the program name (the identifier for the
 *         program class) for the current generator, and the descriptor from the
 *         GlobalSymbolTable
 * 2) If assigning to an indexed location in an array...
 *     a) Generate the array (target.sequence)
 *     b) Generate the index (target.index) (casting it to primitive I)
 *     c) Generate expression (to be saved at index)
 *     d) Call AASTORE, storing the expression at the index, into the array
 * 3) Otherwise, the expression is to be assigned to an attribute of a struct
 *     instance ...
 *     a) Generate the struct instance
 *     b) Generate expression
 *     c) Use PUTFIELD to put the expression's value into the struct object's
```

# The Fearn Programming Language

```
*      attribute
*
*/
@SuppressWarnings("unchecked")
@Override
public void GenerateBytecode(MethodVisitor mv) {

    if (target.getClass() == PrimaryExpression.class) // Variable Reference
    {
        expression.GenerateBytecode(mv);
        PrimaryExpression<String> t = (PrimaryExpression<String>)target;
        String identifier = t.value.toString();

        if (CodeGenerator.LocalSymbolTable.Contains(identifier)) {
            // Local Variable => ASTORE
            mv.visitVarInsn(ASTORE, CodeGenerator.LocalSymbolTable.GetIndex(identifier));
        } else {
            // Target is a Global Variable => PUTSTATIC
            mv.visitFieldInsn(
                PUTSTATIC,
                CodeGenerator.GeneratorStack.peek().programName,
                identifier,
                CodeGenerator.GlobalSymbolTable.GetVarDescriptor(identifier)
            );
        }
    }

    } else if (target.getClass() == IndexExpression.class) { // Index Expression

        IndexExpression targ = (IndexExpression)target;

        /*
        * To get an assign to an index expression, first I need to load the array by
        * generating the target. Then, I need to generate the index. Finally,
        * I need to generate the expression I wish to store and call AASTORE.
        */

        targ.sequence.GenerateBytecode(mv);
        targ.index.GenerateBytecode(mv);
        mv.visitMethodInsn(
            INVOKEVIRTUAL, "java/lang/Integer",
            "intValue", "()I", false
        );
        expression.GenerateBytecode(mv);
        mv.visitInsn(AASTORE);

    } else { // Struct Attribute Expression
```



## The Fearn Programming Language

```
StructAttrExpression targ = (StructAttrExpression)target;

/*
 * To assign to a struct attribute, I need to load the object,
 * then generate the expression. Then, I can use PUTFIELD to set
 * the attribute.
 */

targ.instance.GenerateBytecode(mv);
expression.GenerateBytecode(mv);

mv.visitFieldInsn(
    PUTFIELD,
    "$" + targ.struct_name,
    targ.attribute,
    targ.attr_descriptor
);
}
}

/* To validate an assignment...
 * 1) Ensure the target is an assignable expression (VariableReference,
 *    IndexExpression, or StructAttrExpression)
 * 2) Call HandleOperators (to handle operators like +=, %= etc).
 * 3) Check the TypeSpecifiers of the target and expression are the same
 * 4) Set expression_type to null and return it, as Assignments don't
 *    evaluate to a value
 */

@SuppressWarnings("rawtypes")
public TypeSpecifier validate(SymbolTable symTable) {

    if (target instanceof PrimaryExpression && (
        (PrimaryExpression)target ).type == ExprType.VariableReference) {}
    else if (target.getClass() == IndexExpression.class) {}
    else if (target.getClass() == StructAttrExpression.class) {}
    else {
        Reporter.ReportErrorAndExit("Cannot assign value to " + target.toString(), this);
    }
}
```

# The Fearn Programming Language

```
HandleOperators();

// Check the TypeSpecifiers of the target and expression are equal
TypeSpecifier targetType = target.validate(symTable);
TypeSpecifier exprType = expression.validate(symTable);

if (!targetType.equals(exprType))
{
    Reporter.ReportErrorAndExit(
        "Cannot assign " + exprType.toString() + " to " + targetType.toString(),
        this
    );
}
// Assign Expression perform a job, they don't evaluate to anything
expression_type = null;
return expression_type;
}

/* HandleOperators handles the different operation assignments.
 *
 * It converts the expression to an instance of the class OpEqualsExpr, a derived
 * class of BinaryExpression. The purpose of doing this, over just using
 * BinaryExpression, is to override toString, to show the correct operator in case
 * of an error.
 *
 * Depending on the operation, an OpEqualsExpr is set. This means the
 * GenerateBytecode method doesn't need to handle each case.
 *
 * For example, the assignment 'x += 5' is simplified to 'x = x + 5', with the
 * 'x + 5' modelled as an OpEqualsExpr instance.
 */
private void HandleOperators()
{
    class OpEqualsExpr extends BinaryExpression {
        public OpEqualsExpr(Expression op1, Expression op2, ExprType op)
        { super(op1, op2, op ); }

        @Override
        public String toString() {
            String opString = null;
            switch (Operation) {
                case ExprType.Add : opString = "+="; break;
                case ExprType.Sub : opString = "-="; break;
                case ExprType.Mult : opString = "*="; break;
                case ExprType.Div : opString = "/="; break;
            }
        }
    }
}
```

## The Fearn Programming Language

```
        case ExprType.Mod      : opString = "%="; break;
        default: break;
    }

    return String.format("%s %s %s", Op1.toString(), opString, Op2.toString());
}
}
switch (operator) {
    case Equals:
        return;
    case AddEquals:
        expression = new OpEqualsExpr(target, expression, ExprType.Add);
        return;
    case SubEquals:
        expression = new OpEqualsExpr(target, expression, ExprType.Sub);
        return;
    case MultEquals:
        expression = new OpEqualsExpr(target, expression, ExprType.Mult);
        return;
    case DivEquals:
        expression = new OpEqualsExpr(target, expression, ExprType.Div);
        return;
    case ModEquals:
        expression = new OpEqualsExpr(target, expression, ExprType.Mod);
        return;
}
}
}
```

# The Fearn Programming Language

```
ast.expression.BinaryExpression

package ast.expression;

import org.objectweb.asm.MethodVisitor;

import static org.objectweb.asm.Opcodes.*;

import ast.type.PrimitiveSpecifier.PrimitiveDataType;
import ast.type.PrimitiveSpecifier;
import ast.type.TypeSpecifier;
import semantics.table.SymbolTable;
import util.Reporter;

/* BinaryExpression.java
 *
 * Represents a Binary Expression in the AST.
 *
 * Fields:
 * -> Op1: Left operand
 * -> Op2: Right operand
 * -> Operation: The operation to be performed
 */

public class BinaryExpression extends Expression {

    public Expression Op1;
    public Expression Op2;
    public ExprType Operation;

    public BinaryExpression(Expression op1, Expression op2, ExprType op)
    {
        Op1 = op1;
        Op2 = op2;
        Operation = op;
    }

    @Override
    public String toString()
    {
        String opString = null;
        switch (Operation) {
            case Add      : opString = "+" ; break;
            case Sub      : opString = "-" ; break;
            case Mult     : opString = "*" ; break;
            case Div      : opString = "/" ; break;
        }
    }
}
```

## The Fearn Programming Language

```
        case Mod          : opString = "%" ; break;
        case Exponent     : opString = "^" ; break;
        case Less         : opString = "<" ; break;
        case Greater      : opString = ">" ; break;
        case LessEq       : opString = "<="; break;
        case GreaterEq    : opString = ">="; break;
        case LogicalAnd   : opString = "&&"; break;
        case LogicalOr    : opString = "||"; break;
        case Eq           : opString = "=="; break;
        case NotEq        : opString = "!="; break;

        default: break;
    }

    return String.format("%s %s %s", Op1.toString(), opString, Op2.toString());
}

/* To generate bytecode, in general...
 * 1) Generate both Operands (casting them if necessary)
 * 2) For each type of operation, call the corresponding instruction (or FearnRuntime
 *    method)
 * 3) Cast the result back to an object if necessary
 */

@Override
public void GenerateBytecode(MethodVisitor mv) {
    String t = "";

    if (Operation == ExprType.Eq || Operation == ExprType.NotEq)
    {
        // For these operations, the values on the stacks cannot be primitive
        Op1.GenerateBytecode(mv);
        Op2.GenerateBytecode(mv);
    }

    else if (Op1.expression_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT)))
    {
        // Generate operands, and cast to int
        t = "int";
        Op1.GenerateBytecode(mv);
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Integer", "intValue", "()I", false);
        Op2.GenerateBytecode(mv);
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Integer", "intValue", "()I", false);
    }

    else if (Op1.expression_type.equals(new PrimitiveSpecifier(PrimitiveDataType.FLOAT)))
```

## The Fearn Programming Language

```
{
    t = "double";
    // Generate operands, and cast to double
    Op1.GenerateBytecode(mv);
    mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Double", "doubleValue", "()D", false);
    Op2.GenerateBytecode(mv);
    mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Double", "doubleValue", "()D", false);
}

else if (Op1.expression_type.equals(new PrimitiveSpecifier(PrimitiveDataType.STR)))
{
    t = "str";
    Op1.GenerateBytecode(mv);
    Op2.GenerateBytecode(mv);
}

else if (Op1.expression_type.equals(new PrimitiveSpecifier(PrimitiveDataType.BOOL)))
{
    t = "bool";
    Op1.GenerateBytecode(mv);
    Op2.GenerateBytecode(mv);
}

switch (Operation) {
    case Add:
        if (t == "int") {
            // Add
            mv.visitInsn(IADD);
            // Cast result to Integer
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf",
"(I)Ljava/lang/Integer;", false);
            return;
        } else if (t == "double") {
            // Add
            mv.visitInsn(DADD);
            // Cast result to Double
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Double", "valueOf",
"(D)Ljava/lang/Double;", false);
            return;
        } else { // String Concatenation
```

# The Fearn Programming Language

```
        mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "concat",
"(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;", false);
        return;
    }

    case Sub:
        if (t == "int") {
            // Add
            mv.visitInsn(ISUB);
            // Cast result to Integer
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf",
"(I)Ljava/lang/Integer;", false);
            return;
        } else { // Floats
            mv.visitInsn(DSUB);
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Double", "valueOf",
"(D)Ljava/lang/Double;", false);
            return;
        }

    case Mult:
        if (t == "int") {
            mv.visitInsn(IMUL);
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf",
"(I)Ljava/lang/Integer;", false);
            return;
        } else { // Floats
            mv.visitInsn(DMUL);
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Double", "valueOf",
"(D)Ljava/lang/Double;", false);
            return;
        }

    case Div:
        if (t == "int") {
            mv.visitInsn(IDIV);
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf",
"(I)Ljava/lang/Integer;", false);
            return;
        } else { // Floats
            mv.visitInsn(DDIV);
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Double", "valueOf",
"(D)Ljava/lang/Double;", false);
            return;
        }

    case Exponent:
```

# The Fearn Programming Language

```
        if (t == "int") {
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "exp",
"(II)Ljava/lang/Integer;", false);
            return;
        } else { // Floats
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "exp",
"(DD)Ljava/lang/Double;", false);
            return;
        }
    case Less:
        if (t == "int") {
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "less",
"(II)Ljava/lang/Boolean;", false);
            return;
        } else { // Floats
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "less",
"(DD)Ljava/lang/Boolean;", false);
            return;
        }
    case LessEq:
        if (t == "int") {
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "less_eq",
"(II)Ljava/lang/Boolean;", false);
            return;
        } else { // Floats
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "less_eq",
"(DD)Ljava/lang/Boolean;", false);
            return;
        }
    case Greater:
        if (t == "int") {
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "greater",
"(II)Ljava/lang/Boolean;", false);
            return;
        } else { // Floats
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "greater",
"(DD)Ljava/lang/Boolean;", false);
            return;
        }
    case GreaterEq:
        if (t == "int") {
            mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "greater_eq",
"(II)Ljava/lang/Boolean;", false);
            return;
        } else { // Floats
```



# The Fearn Programming Language

```
        mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "greater_eq",
"(DD)Ljava/lang/Boolean;", false);
        return;
    }

    case Mod:
        mv.visitInsn(IREM);
        mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf",
"(I)Ljava/lang/Integer;", false);
        return;
    case LogicalAnd:
        mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "and",
"(Ljava/lang/Boolean;Ljava/lang/Boolean;)Ljava/lang/Boolean;", false);
        return;
    case LogicalOr:
        mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "or",
"(Ljava/lang/Boolean;Ljava/lang/Boolean;)Ljava/lang/Boolean;", false);
        return;
    case Eq:
        mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "equals",
"(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Boolean;", false);
        return;
    case NotEq:
        mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "equals",
"(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Boolean;", false);
        mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "not",
"(Ljava/lang/Boolean;)Ljava/lang/Boolean;", false);
        return;
    default:
        Reporter.ReportErrorAndExit("Error in Generating Binary Expression.", null);
        break;
}
}

/* To validate...
 * 1) Get types for both operands
 * 2) Check they are valid types for the operation, and the same
 *    -> Raise errors otherwise
 * 3) Set expression_type to an appropriate TypeSpecifier, and
 *    return it
 */
@Override
public TypeSpecifier validate(SymbolTable symTable) {

    TypeSpecifier op1_type = Op1.validate(symTable);
```

# The Fearn Programming Language

```
TypeSpecifier op2_type = Op2.validate(symTable);

switch (Operation) {

    // All cases where the operands must
    // both be numeric.
    case Mult:
    case Div:
    case Sub:
    case Exponent:
    case Less:
    case LessEq:
    case Greater:
    case GreaterEq:
        if (
            op1_type.equals(op2_type) &&
            (
                op1_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT)) ||
                op1_type.equals(new PrimitiveSpecifier(PrimitiveDataType.FLOAT))
            )
        ) {
            switch (Operation) {
                case Mult:
                case Div:
                case Sub:
                case Exponent:
                    expression_type = op1_type;
                    break;
                default:
                    expression_type = new PrimitiveSpecifier(PrimitiveDataType.BOOL);
                    break;
            }
        } else {
            Reporter.ReportErrorAndExit("Operands must be either (a) both ints, or (b)
both floats.", this);
        }
        break;

    // Can work on numbers or strings
    case Add:
        if (
            op1_type.equals(op2_type) &&
            (
                op1_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT)) ||
                op1_type.equals(new PrimitiveSpecifier(PrimitiveDataType.FLOAT)) ||
                op1_type.equals(new PrimitiveSpecifier(PrimitiveDataType.STR))
            )
        ) {
            switch (Operation) {
                case Add:
                    expression_type = op1_type;
                    break;
                default:
                    expression_type = new PrimitiveSpecifier(PrimitiveDataType.BOOL);
                    break;
            }
        } else {
            Reporter.ReportErrorAndExit("Operands must be either (a) both ints, or (b)
both floats.", this);
        }
        break;
}
```

## The Fearn Programming Language

```
        )
    ) {
        expression_type = op1_type;
    } else {
        Reporter.ReportErrorAndExit("Operands must be either (a) both ints, (b) both
floats, or (c) both strings.", this);
    }
    break;

    // Modulo only works on integers
    case Mod:
        if ( op1_type.equals(op2_type) && op1_type.equals(new
PrimitiveSpecifier(PrimitiveDataType.INT)) ) {
            expression_type = op1_type;
        } else {
            Reporter.ReportErrorAndExit("Operands must both be ints.", this);
        }
        break;

    // Both operands must be boolean
    case LogicalAnd:
    case LogicalOr:
        if ( op1_type.equals(op2_type) && op1_type.equals(new
PrimitiveSpecifier(PrimitiveDataType.BOOL)) ) {
            expression_type = op1_type;
        } else {
            Reporter.ReportErrorAndExit("Operands must both be boolean values.", this);
        }
        break;

    // Both must be of the same type
    case Eq:
    case NotEq:
        if ( op1_type.equals(op2_type)) {
            expression_type = new PrimitiveSpecifier(PrimitiveDataType.BOOL);
        } else {
            Reporter.ReportErrorAndExit("Operands must both be of the same type.", this);
        }
        break;
    default:
        Reporter.ReportErrorAndExit("An Error has occurred.", this);
        break;
}
return expression_type;
}
```

# The Fearn Programming Language

```
ast.expression.CastExpression
package ast.expression;

import org.objectweb.asm.MethodVisitor;
import static org.objectweb.asm.Opcodes.*;

import ast.type.PrimitiveSpecifier.PrimitiveDataType;
import ast.type.PrimitiveSpecifier;
import ast.type.TypeSpecifier;
import semantics.table.SymbolTable;
import util.Reporter;

/* CastExpression.java
 *
 * Represents a Type Cast in the AST.
 *
 * Fields:
 * -> target: The PrimitiveDataType being cast to
 * -> Operand: The expression to be cast
 */

public class CastExpression extends Expression {
    public PrimitiveDataType target;
    public Expression Operand;
    public CastExpression(Expression operand, PrimitiveDataType targetType)
    {
        target = targetType;
        Operand = operand;
    }
    @Override
    public String toString()
    {
        return "(" + target.name().toLowerCase() + ")" + Operand.toString();
    }
    /* To generate bytecode, generate the operand, then...
     * -> If targeting int, call the appropriate method based on operands
     *      expression_type, casting to Integer after
     * -> If targeting float, follow a similar procedure
     * -> If targeting str, call the Obj2Str method of the FearnRuntime
     * -> If targeting bool, call the Obj2B method of the FearnRuntime
     */

    @Override
    public void GenerateBytecode(MethodVisitor mv) {
```

# The Fearn Programming Language

```
Operand.GenerateBytecode(mv);

switch (target) {
    case INT:
        if (Operand.expression_type.equals(new
PrimitiveSpecifler(PrimitiveDataType.FLOAT)))
        {
            mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Double", "intValue", "()I",
false);
        }

        else if (Operand.expression_type.equals(new
PrimitiveSpecifler(PrimitiveDataType.STR)))
        {
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf",
"(Ljava/lang/String;)Ljava/lang/Integer;", false);
            return;
        }

        else if (Operand.expression_type.equals(new
PrimitiveSpecifler(PrimitiveDataType.BOOL)))
        {
            mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Boolean", "booleanValue", "()Z",
false);

            mv.visitInsn(ICONST_0);
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Boolean", "compare", "(ZZ)I",
false);
        } else { return; }

            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf",
"(I)Ljava/lang/Integer;", false);
            return;

    case FLOAT:
        if (Operand.expression_type.equals(new PrimitiveSpecifler(PrimitiveDataType.INT)))
        {
            mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Double", "intValue", "()I",
false);

            mv.visitInsn(I2D);
        }

        else if (Operand.expression_type.equals(new
PrimitiveSpecifler(PrimitiveDataType.STR)))
        {
```

# The Fearn Programming Language

```
        mv.visitMethodInsn(INVOKESTATIC, "java/lang/Double", "valueOf",
"(Ljava/lang/String;)Ljava/lang/Double;", false);
        return;
    } else { return; }

    mv.visitMethodInsn(INVOKESTATIC, "java/lang/Double", "valueOf",
"(D)Ljava/lang/Double;", false);
    return;

    case STR:
        mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "Obj2Str",
"(Ljava/lang/Object;)Ljava/lang/String;", false);
        return;
    case BOOL:
        mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "Obj2B",
"(Ljava/lang/Object;)Ljava/lang/Boolean;", false);
        return;
    }

}

/* To validate, validate the operand, and check that, for the target type, the
 * operand.expression_type is one where that operation is valid.
 */
@Override
public TypeSpecifier validate(SymbolTable symTable) {
    // For Each target type, ensure the operand can be cast
    TypeSpecifier op_type = Operand.validate(symTable);

    switch (target) {
        case PrimitiveDataType.INT: // You can cast strings, floats, and bools
(Boolean.compare) to integers
            if (
                op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT    ))
                || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.FLOAT ))
                || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.STR   ))
                || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.BOOL  ))
            ) {
                expression_type = new PrimitiveSpecifier(PrimitiveDataType.INT);
            } else {
                Reporter.ReportErrorAndExit("Cannot perform cast from " + op_type.toString() +
" to int.", this);
            }
            break;

        case PrimitiveDataType.FLOAT: // You can cast strings and ints to floats
            if (
```

## The Fearn Programming Language

```
        op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT    ))
    || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.FLOAT  ))
    || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.STR    ))
) {
    expression_type = new PrimitiveSpecifier(PrimitiveDataType.FLOAT);
} else {
    Reporter.ReportErrorAndExit("Cannot perform cast from " + op_type.toString() +
" to float.", this);
}
break;

case PrimitiveDataType.STR: // You can cast anything, including arrays and structs, to
strings
    expression_type = new PrimitiveSpecifier(PrimitiveDataType.STR);
    break;

case PrimitiveDataType.BOOL: // You can cast strings, ints, and floats to bools
    if (
        op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT    ))
    || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.STR    ))
    || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.FLOAT  ))
    || op_type.equals(new PrimitiveSpecifier(PrimitiveDataType.BOOL    ))
    ) {
        expression_type = new PrimitiveSpecifier(PrimitiveDataType.BOOL);
    } else {
        Reporter.ReportErrorAndExit(
            "Cannot perform cast from " + op_type.toString() + " to bool.", this);
    }
    break;

default: break;
}
return expression_type;
}
```

# The Fearn Programming Language

```
ast.expression.FnCallExpression
```

```
package ast.expression;
```

```
import static org.objectweb.asm.Opcodes.*;
```

```
import java.util.ArrayList;
```

```
import java.util.Arrays;
```

```
import java.util.List;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import ast.type.ArraySpecifier;
```

```
import ast.type.PrimitiveSpecifier.PrimitiveDataType;
```

```
import ast.type.PrimitiveSpecifier;
```

```
import ast.type.TypeSpecifier;
```

```
import codegen.CodeGenerator;
```

```
import semantics.table.SymbolTable;
```

```
import util.Reporter;
```

```
/* FnCallExpression.java
```

```
 *
```

```
 * Represents a Function Call in the AST.
```

```
 *
```

```
 * Fields:
```

```
 * -> identifier: Function identifier
```

```
 * -> arguments: The expressions used as arguments in the function call
```

```
 */
```

```
public class FnCallExpression extends Expression {
```

```
    public String identifier;
```

```
    public ArrayList<Expression> arguments;
```

```
    // Flag used to indicate if a function is written in Universal Function Notation
```

```
    // e.g. x.myFunction()
```

```
    // It is only used in toString()
```

```
    public Boolean isUFN = false;
```

```
    // List of built-in functions
```

```
    // These aren't added to the SymbolTable simply because they take
```

```
    // multiple data types as input (e.g. length() take 1 argument,
```

```
    // which is either an array or a string)
```

```
    static List<String> builtins = Arrays.asList("length", "slice");
```



## The Fearn Programming Language

```
public FnCallExpression(String fn_name, ArrayList<Expression> args)
{
    identifier = fn_name;
    arguments = args;
}

@Override
public String toString()
{
    if (isUFN)
    {
        String str_rep = arguments.get(0).toString() + "." + identifier;
        arguments.remove(0);
        str_rep += arguments.toString()
            .replace("[", "(")
            .replace("]", ")");
        return str_rep;
    } else
    {
        return identifier+"("+arguments.toString().substring(1, arguments.toString().length()-1)+")";
    }
}

/* To generate bytecode, generate all arguments.
 *
 * Then, if the function call is one of the built-ins, set the descriptor to the
 * correct value, and call the method, using INVOKESTATIC and the FearnRuntime.
 * -> If slice is used, returned array must be cast back, using the array's type
 * descriptor
 *
 * Otherwise, use INVOKESTATIC, using the identifier, descriptor, and owner from
 * the SymbolTable.
 */
@Override
public void GenerateBytecode(MethodVisitor mv) {

    // Gen args, then INVOKESTATIC
    for (Expression arg : arguments) arg.GenerateBytecode(mv);

    String desc;

    switch (identifier){

        case "length":
            desc = "(Ljava/lang/Object;)Ljava/lang/Integer;";
            break;
    }
}
```

# The Fearn Programming Language

```
        case "slice":
            String t = null;

            if (arguments.get(0).expression_type.equals(new
PrimitiveSpecifier(PrimitiveDataType.STR))) t = "Ljava/lang/String;";
            else t = "[Ljava/lang/Object;";

            desc = String.format( "(%sLjava/lang/Integer;Ljava/lang/Integer;)s", t, t );
            break;

        default:
            desc = CodeGenerator.GlobalSymbolTable.GetGlobalFuncDescriptor(identifier);
            break;
    }

    if (builtins.contains(identifier))
    {
        mv.visitMethodInsn(
            INVOKESTATIC,
            "FearnRuntime",
            identifier,
            desc,
            false
        );

        if (identifier.equals("slice") && desc.contains("Object"))
        {
            mv.visitTypeInsn(CHECKCAST,
SymbolTable.GenBasicDescriptor(arguments.get(0).expression_type));
        }

        return;
    }

    mv.visitMethodInsn(
        INVOKESTATIC,
        CodeGenerator.GlobalSymbolTable.GetOwner(identifier, true),
        identifier,
        desc,
        false
    );
};
```

# The Fearn Programming Language

```
}

/* To validate...
 * -> If the function is one of the built-ins, check the number of arguments and
 *     their types, raising an error if necessary
 * -> Otherwise, using the parameters from the SymbolTable to validate the
 *     arguments (number and types)
 * -> Set expression_type to return type of function, and return this
 *
 */
@Override
public TypeSpecifier validate(SymbolTable symTable) {

    // Check the function's signature (Parameters from Symbol Table) against to types of each
argument

    ArrayList<TypeSpecifier> arg_types = new ArrayList<TypeSpecifier>();

    switch (identifier) {

        // length(<str|arr>)
        case "length":
            if (arguments.size() != 1)
            {
                Reporter.ReportErrorAndExit("Wrong number of arguments for " + identifier + "
, expected 1.", this);
            }

            if (!(
                arguments.get(0).validate(symTable).equals(new
PrimitiveSpecifier(PrimitiveDataType.STR)) ||
                arguments.get(0).validate(symTable) instanceof ArraySpecifier
            ))
            {
                Reporter.ReportErrorAndExit("Wrong argument data type, expected string or
array.", this);
            }

            expression_type = new PrimitiveSpecifier(PrimitiveDataType.INT);
            return expression_type;

        // slice(<str|arr>, int, int);
```

# The Fearn Programming Language

```
        case "slice":
            if (arguments.size() != 3)
            {
                Reporter.ReportErrorAndExit("Wrong number of arguments for " + identifier + "
, expected 3.", this);
            }

            if (!(
                arguments.get(0).validate(symTable).equals(new
PrimitiveSpecifier(PrimitiveDataType.STR)) ||
                arguments.get(0).validate(symTable) instanceof ArraySpecifier
            ))
            {
                Reporter.ReportErrorAndExit("Wrong argument data type, expected string or
array.", this);
            }

            if (!(
                arguments.get(1).validate(symTable).equals(new
PrimitiveSpecifier(PrimitiveDataType.INT)) ||
                arguments.get(2).validate(symTable).equals(new
PrimitiveSpecifier(PrimitiveDataType.INT))
            ))
            {
                Reporter.ReportErrorAndExit("Wrong argument data type, expected int.", this);
            }

            expression_type = arguments.get(0).expression_type;
            return expression_type;

        default:
            break;
    }

    ArrayList<TypeSpecifier> param_types =
CodeGenerator.GlobalSymbolTable.GetFuncParameterSpecifiers(identifier);

    for (Expression arg : arguments)
    {
        arg_types.add(arg.validate(symTable));
    }

    if (arguments.size() != param_types.size())
```

## The Fearn Programming Language

```
{
    Reporter.ReportErrorAndExit(
        "Wrong number of arguments for " + identifier + " , expected " +
param_types.size(),
        this
    );
}

for (int i = 0; i < param_types.size(); i++)
{
    if (!param_types.get(i).equals(arg_types.get(i)))
    {
        Reporter.ReportErrorAndExit(
            arguments.get(i).toString() + " is of the wrong type, expected " +
param_types.get(i).toString(),
            this
        );
    }
}

expression_type = CodeGenerator.GlobalSymbolTable.GetTypeSpecifier(identifier, true);
return expression_type;
}

}
```

# The Fearn Programming Language

ast.expression.IncrExpression

```
package ast.expression;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import ast.expression.AssignExpression.AssignmentOperator;
```

```
import ast.type.PrimitiveSpecifier.PrimitiveDataType;
```

```
import ast.type.PrimitiveSpecifier;
```

```
import ast.type.TypeSpecifier;
```

```
import semantics.table.SymbolTable;
```

```
import util.Reporter;
```

```
/* IncrExpression.java
```

```
 * Represents an Increment/Decrement Expression in the AST.
```

```
 * Fields:
```

```
 * -> expression: An assignable integer expression, to increment/decrement
```

```
 * -> isPrefix: Boolean flag, true if the expression is a prefix increment/decrement
```

```
 * -> isDecrement: Boolean flag, true if expression is a Decrement
```

```
 *
```

```
 * -> increment: private AssignExpression, used to generate the bytecode to
```

```
 * increment/decrement the target expression
```

```
 */
```

```
public class IncrExpression extends Expression {
```

```
    public Expression expression;
```

```
    public Boolean isPrefix;
```

```
    public Boolean isDecrement;
```

```
    private AssignExpression increment;
```

```
    public IncrExpression(Expression expr, Boolean isDec, Boolean isPre) {
```

```
        expression = expr;
```

```
        isPrefix = isPre;
```

```
        isDecrement = isDec;
```

```
    }
```

```
    @Override public String toString()
```

```
    {
```

```
        String symbol = isDecrement ? "--" : "++";
```

```
        if (isPrefix) return symbol + expression.toString();
```

```
        return expression.toString() + symbol;
```

```
    }
```

```
    /* To generate bytecode, the bytecode to load the expression and perform the increment
```

## The Fearn Programming Language

```
* both need to be generated. The order in which this is done depends on whether the
* expression is prefix.
*/
public void GenerateBytecode(MethodVisitor mv)
{
    if (isPrefix)
    {
        // Increment first, then load value to stack
        increment.GenerateBytecode(mv);
        expression.GenerateBytecode(mv);
        return;
    }

    // Load value to stack, then increment
    expression.GenerateBytecode(mv);
    increment.GenerateBytecode(mv);
    return;
}

/* To validate, ensure the expression is assignable. This means it is either
* -> A variable reference (a sort of Primary expression)
* -> An index expression (e.g. x[0])
* -> A reference to an attribute of a struct instance
*
* Then, the expression is validated, and an error is raised if it's not of type int.
*
* If the expression is valid, increment is set to a new AssignExpression, where the
* operator is determined using the isDecrement property. If so, the expression uses -=,
* otherwise +=. This expression is validated (no errors should be raised at this point,
* but the nodes below need to configure themselves before generation). expression_type is
* set to int, and this is returned.
*
*/

@SuppressWarnings("rawtypes")
public TypeSpecifier validate(SymbolTable symTable)
{
    if (expression instanceof PrimaryExpression
        && ( (PrimaryExpression)expression ).type == ExprType.VariableReference) {}
    else if (expression instanceof IndexExpression) {}
    else if (expression instanceof StructAttrExpression) {}
    else Reporter.ReportErrorAndExit(
        "Cannot increment " + expression.toString() + ". Only int variables can be
incremented/decremented.",
        this
    );
}
```

## The Fearn Programming Language

```
TypeSpecifier expr_type = expression.validate(symTable);

if ((!expr_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT))))
{
    Reporter.ReportErrorAndExit("Can only increment/decrement ints.", this);
}

increment = new AssignExpression(
    expression,
    new PrimaryExpression<Integer>(1, ExprType.IntLiteral),
    isDecrement ? AssignmentOperator.SubEquals : AssignmentOperator.AddEquals
);

increment.validate(symTable);

expression_type = expr_type;
return expression_type;
}
}
```



# The Fearn Programming Language

ast.expression.IndexExpression

```
package ast.expression;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import static org.objectweb.asm.Opcodes.*;
```

```
import ast.type.ArraySpecifier;
```

```
import ast.type.PrimitiveSpecifier.PrimitiveDataType;
```

```
import ast.type.PrimitiveSpecifier;
```

```
import ast.type.TypeSpecifier;
```

```
import semantics.table.SymbolTable;
```

```
import util.Reporter;
```

```
/* IndexExpression.java
```

```
 *
```

```
 * Represents an Index Expression in the AST (e.g list[0] ).
```

```
 *
```

```
 * Fields:
```

```
 * -> sequence: An expression that is indexable (an array or string)
```

```
 * -> index: An integer expression, hat represents the index to access
```

```
 */
```

```
public class IndexExpression extends Expression {
```

```
    public Expression sequence;
```

```
    public Expression index;
```

```
    public IndexExpression(Expression id, Expression i)
```

```
    {
```

```
        sequence = id;
```

```
        index = i;
```

```
    }
```

```
    @Override
```

```
    public String toString()
```

```
    {
```

```
        return sequence.toString() + '[' + index.toString() + '];
```

```
    }
```

```
/* To generate bytecode, first generate the sequence and index bytecode, to prepare
```

```
 * the stack.
```

## The Fearn Programming Language

```
* If the sequence is an array, use AALOAD instruction. Otherwise (sequence is a
* string), use charAt method, then cast result to string
*/
@Override
public void GenerateBytecode(MethodVisitor mv) {

    sequence.GenerateBytecode(mv);
    index.GenerateBytecode(mv);
    mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Integer", "intValue", "()I", false);

    if (sequence.expression_type instanceof ArraySpecifier)
    {
        mv.visitInsn(AALOAD);
    } else {
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/String", "charAt", "(I)C", false);
        mv.visitMethodInsn(INVOKESTATIC, "java/lang/String", "valueOf",
"(C)Ljava/lang/String;", false);
    }
}

/* To validate, validate the sequence and index.
*
* Raise an error is the sequence is not an array or string, or if the index is not
* an int.
*
* Set expression_type to string if the sequence is a string. For arrays, take 1
* dimension from the specifier of a multidimensional array. If the array is only
* 1D, set the expression_type to the TypeSpecifier of an element.
*
* Return expression_type.
*
*/

@Override
public TypeSpecifier validate(SymbolTable symTable) {

    TypeSpecifier seq_type      = sequence.validate(symTable);
    TypeSpecifier index_type    = index.validate(symTable);

    if (seq_type.getClass() != ArraySpecifier.class && !seq_type.equals(new
PrimitiveSpecifier(PrimitiveDataType.STR)))
    {
        Reporter.ReportErrorAndExit("Can only take index of Arrays and Strings.", this);
    }

    if (!index_type.equals(new PrimitiveSpecifier(PrimitiveDataType.INT)))
```

## The Fearn Programming Language

```
{
    Reporter.ReportErrorAndExit("Index can only be an int.", this);
}

// Use seq_type to set expression type
if (seq_type.equals(new PrimitiveSpecifier(PrimitiveDataType.STR)))
{
    expression_type = seq_type;
    return expression_type;
}

ArraySpecifier seq_arr_spec = (ArraySpecifier)seq_type;
if (seq_arr_spec.dimensionCount == 1) {
    expression_type = seq_arr_spec.element_type;
} else {
    expression_type = new ArraySpecifier(seq_arr_spec.element_type,
seq_arr_spec.dimensionCount - 1);
}

return expression_type;
}
}
```

# The Fearn Programming Language

```
ast.expression.PrimaryExpression
```

```
package ast.expression;
```

```
import static org.objectweb.asm.Opcodes.*;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import ast.type.PrimitiveSpecifier.PrimitiveDataType;
```

```
import ast.type.PrimitiveSpecifier;
```

```
import ast.type.TypeSpecifier;
```

```
import codegen.CodeGenerator;
```

```
import semantics.table.SymbolTable;
```

```
/* PrimaryExpression.java
```

```
*
```

```
* This represents literals and variable references in the AST.
```

```
*
```

```
* This is a generic class, which allows the object to store the value of a literal
```

```
* using the correct data type.
```

```
*
```

```
* Fields:
```

```
* -> value: The value of the expression. This has a generic type to store literals
```

```
* as the right type (e.g. a bool literal 'true' will be stored in a
```

```
* PrimaryExpression<Boolean> object)
```

```
* -> type: Indicates the type of primary expression (e.g. VariableReference,
```

```
* FloatLiteral, etc.)
```

```
*/
```

```
public class PrimaryExpression<T> extends Expression {
```

```
    public T value;
```

```
    public ExprType type;
```

```
    public PrimaryExpression(T val, ExprType t)
```

```
    {
```

```
        this.value = val;
```

```
        this.type = t;
```

```
    }
```

```
    @Override
```

```
    public String toString()
```

```
    {
```

```
        if (type == ExprType.StrLiteral) return "\"" + value.toString() + "\"";
```

```
        else return value.toString();
```

```
    }
```

## The Fearn Programming Language

```
/* To generate bytecode, the procedure depends on the type of expression
 * -> For variable references, ALOAD is used to load local variables,
 *     and GETSTATIC is used to retrieve global variables
 * -> The LocalSymbolTable is checked first, as if a local and global variable
 *     have the same identifier, the local value takes priority.
 * -> For integers, SIPUSH is used to push the value to the stack, which is cast to
 *     an Integer object
 * -> For float, LDC loads the value to the stack, and then it's cast to a Double
 *     Object
 * -> For boolean, a 1 or a 0 is loaded to the stack, and then cast to Boolean
 * -> For string, LDC is used to load the value
 */
public void GenerateBytecode(MethodVisitor mv)
{
    if (type == ExprType.VariableReference)
    {
        // Find index of variable (in THIS scope)
        // ALOAD <index>
        if (CodeGenerator.LocalSymbolTable.Contains(this.value.toString()))
        {
            mv.visitVarInsn(ALOAD,
CodeGenerator.LocalSymbolTable.GetIndex(this.value.toString()));
        }

        // Otherwise, variable is global, GETSTATIC
        else
        {
            mv.visitFieldInsn(
                GETSTATIC,
                CodeGenerator.GlobalSymbolTable.GetOwner(this.value.toString(), false),
                this.value.toString(),
                CodeGenerator.GlobalSymbolTable.GetVarDescriptor(this.value.toString())
            );
        }

        // Handle Literals
    } else {
```

# The Fearn Programming Language

```
switch (this.type) {
    case IntLiteral:
        mv.visitIntInsn(SIPUSH, (int)value);
        mv.visitMethodInsn(
            INVOKESTATIC,
            "java/lang/Integer",
            "valueOf",
            "(I)Ljava/lang/Integer;",
            false
        );
        return;
    case FloatLiteral:
        mv.visitLdcInsn((Double)value);
        mv.visitMethodInsn(
            INVOKESTATIC,
            "java/lang/Double",
            "valueOf",
            "(D)Ljava/lang/Double;",
            false
        );
        return;
    case BoolLiteral:
        if ((Boolean)this.value) { mv.visitInsn(ICONST_1); }
        else { mv.visitInsn(ICONST_0); }
        mv.visitMethodInsn(
            INVOKESTATIC,
            "java/lang/Boolean",
            "valueOf",
            "(Z)Ljava/lang/Boolean;",
            false
        );
        return;
    case StrLiteral:
        mv.visitLdcInsn(value);
        return;
    default:
        break;
}
}
```

/\* This validate method doesn't do much, as a primary expression, isolated from context,  
\* cannot be invalid. The method simply returns the relevant TypeSpecifier. For variable

## The Fearn Programming Language

```
* references, this is retrieved from the Symbol Table.
*/
@Override
public TypeSpecifier validate(SymbolTable symTable) {
    if ( type == ExprType.VariableReference && symTable.Contains(value.toString()))
    {
        expression_type = symTable.GetTypeSpecifier(value.toString(), false);
    }
    else if ( type == ExprType.VariableReference ) {
        expression_type = CodeGenerator.GlobalSymbolTable.GetTypeSpecifier(value.toString(),
false);
    } else {
        switch (this.type) {
            case IntLiteral: expression_type = new PrimitiveSpecifier(PrimitiveDataType.INT);
                break;
            case FloatLiteral: expression_type = new
PrimitiveSpecifier(PrimitiveDataType.FLOAT); break;
            case StrLiteral: expression_type = new PrimitiveSpecifier(PrimitiveDataType.STR);
                break;
            case BoolLiteral: expression_type = new
PrimitiveSpecifier(PrimitiveDataType.BOOL); break;
            default: break;
        }
    }
    return expression_type;
}
```

# The Fearn Programming Language

ast.expression.StructAttrExpression

```
package ast.expression;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import static org.objectweb.asm.Opcodes.*;
```

```
import ast.type.StructInstanceSpecifier;
```

```
import ast.type.TypeSpecifier;
```

```
import codegen.CodeGenerator;
```

```
import semantics.table.SymbolTable;
```

```
import util.Reporter;
```

```
/* StructAttrExpression.java
```

```
 *
```

```
 * This represents an access to a named attribute of a struct instance.
```

```
 *
```

```
 * Fields:
```

```
 * -> instance: Expression that resolves to an instance of a struct
```

```
 * -> attribute: The string name of the attribute being accessed
```

```
 * -> struct_name: The name of the struct, that instance is an instance of
```

```
 * -> attr_descriptor: The JVM descriptor of the attribute's type
```

```
 */
```

```
public class StructAttrExpression extends Expression {
```

```
    public Expression instance;
```

```
    public String attribute;
```

```
    protected String struct_name;
```

```
    protected String attr_descriptor;
```

```
    public StructAttrExpression(Expression n, String attr)
```

```
    {
```

```
        instance = n;
```

```
        attribute = attr;
```

```
    }
```

```
    @Override
```

```
    public String toString()
```

```
    {
```

```
        return instance.toString() + "." + attribute.toString();
```

```
    }
```



## The Fearn Programming Language

```
/* To generate bytecode, simply generate the instance (putting it on top of the stack),
 * then use the GETFIELD instruction, with the name of the struct class ($name), and the
 * attribute's identifier and descriptor.
 */
@Override
public void GenerateBytecode(MethodVisitor mv) {
    // Gen instance, then GETFIELD
    instance.GenerateBytecode(mv);

    mv.visitFieldInsn(
        GETFIELD,
        "$" + struct_name,
        attribute,
        attr_descriptor
    );
}

/* To validate,
 * -> Get the instance type, and ensure it is a struct
 * -> Get the struct's Symbol Table from the Global Symbol Table
 * -> Verify the attribute exists in the table
 * -> Set attr_descriptor, using the struct's Symbol Table
 * -> Set expression_type to the TypeSpecifier associated with that attribute, and
 *     return this
 */
@Override
public TypeSpecifier validate(SymbolTable symTable) {
    // Get from StructRow in SymbolTable
    TypeSpecifier inst_type = instance.validate(symTable);
    if (inst_type.getClass() != StructInstanceSpecifier.class)
    {
        Reporter.ReportErrorAndExit(inst_type.toString() + " is not a struct.", this);
    }
    struct_name = ((StructInstanceSpecifier)inst_type).name;
    SymbolTable structTable =
CodeGenerator.GlobalSymbolTable.GetStructSymbolTable(struct_name);
    if (!structTable.Contains(attribute))
    {
        Reporter.ReportErrorAndExit(struct_name + " has no attribute " + attribute, this);
    }
    attr_descriptor = structTable.GetVarDescriptor(attribute);
    expression_type = structTable.GetTypeSpecifier(attribute, false);
    return expression_type;
}
}
```

# The Fearn Programming Language

ast.expression.StructInitExpression

```
package ast.expression;
```

```
import java.util.ArrayList;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import static org.objectweb.asm.Opcodes.*;
```

```
import ast.type.StructInstanceSpecifier;
```

```
import ast.type.TypeSpecifier;
```

```
import codegen.CodeGenerator;
```

```
import semantics.table.SymbolTable;
```

```
import util.Reporter;
```

```
/* StructInitExpression.java
```

```
 *
 * This represents an initialisation of a struct instance
 *
 * Fields:
 * -> name: The name of the struct being instantiated
 * -> arguments: The expressions used for the struct's initial state
 */
```

```
public class StructInitExpression extends Expression {
```

```
    public String name;
```

```
    public ArrayList<Expression> arguments;
```

```
    public StructInitExpression(String n, ArrayList<Expression> args)
```

```
    {
        name = n;
        arguments = args;
    }
```

```
    @Override
```

```
    public String toString()
```

```
    {
        return "new " + name + "(" + arguments.toString().substring(1,
arguments.toString().length() - 1) + ")";
    }
```

```
    /* To generate bytecode, the NEW instruction is used to create an instance of the
    * struct class ($name). This instance is duplicated, and the arguments are generated,
    * leaving their values on top of the stack. Finally, the struct class's constructor is
    * invoked, to set the state of the object. This requires the descriptor from the Global
    * Symbol Table.
    */
```

```
    @Override
```

## The Fearn Programming Language

```
public void GenerateBytecode(MethodVisitor mv) {
    mv.visitTypeInsn(NEW, "$" + name);
    mv.visitInsn(DUP);
    for (Expression arg : arguments)
    {
        arg.GenerateBytecode(mv);
    }
    mv.visitMethodInsn(
        INVOKESPECIAL,
        "$" + name,
        "<init>",
        CodeGenerator.GlobalSymbolTable.GetGlobalStructDescriptor(name),
        false
    );
}

/* To validate, the arguments are validated, to ensure they are the right type,
 * and that there are the correct number. If so, a StructInstanceSpecifier is set
 * to expression_type and returned.
 */
@Override
public TypeSpecifier validate(SymbolTable symTable) {
    // Checks args, then return type of struct
    ArrayList<TypeSpecifier> attr_types =
CodeGenerator.GlobalSymbolTable.GetStructAttributeSpecifiers(name);

    if (attr_types.size() != arguments.size())
    {
        Reporter.ReportErrorAndExit("Wrong number of arguments, expected " +
attr_types.size(), this);
    }

    for (int i = 0; i < attr_types.size(); i++)
    {
        if (!arguments.get(i).validate(symTable).equals(attr_types.get(i)))
        {
            Reporter.ReportErrorAndExit("Wrong argument type for " +
arguments.get(i).toString() + ", expected " + attr_types.get(i).toString(), this);
        }
    }
    expression_type = new StructInstanceSpecifier(name);
    return expression_type;
}
}
```

# The Fearn Programming Language

```
ast.expression.UnaryExpression
package ast.expression;

import org.objectweb.asm.MethodVisitor;

import static org.objectweb.asm.Opcodes.*;

import ast.type.PrimitiveSpecifier.PrimitiveDataType;
import ast.type.PrimitiveSpecifier;
import ast.type.TypeSpecifier;
import semantics.table.SymbolTable;
import util.Reporter;
/* UnaryExpression.java
 *
 * This represents a unary expression (e.g. -a, !a)
 *
 * Fields:
 * -> operand: The expression to operate on
 * -> operator: Indicates the operation to be done
 */
public class UnaryExpression extends Expression {

    public Expression operand;
    public ExprType operator;

    public UnaryExpression(Expression op, ExprType type)
    {
        operand = op;
        operator = type;
    }
    @Override
    public String toString()
    {
        String opString = null;
        switch (operator) {
            case Negate:
                opString = "-";
                break;
            case LogicalNot:
                opString = "!";
                break;
            default:
                break;
        }
        return opString + operand.toString();
    }
}
```

## The Fearn Programming Language

```
/* To generate bytecode, the instructions used depend on the operation, and type of
 * operand. First, the operand must be generated (value now at top of stack)
 * For negation...
 * -> IF the operand is an int, Convert the Integer object to a primitive I, use the
 *      INEG instruction to negate the value, and cast the result back to Integer
 * -> IF the operand is an float, Convert the Double object to a primitive D, use the
 *      DNEG instruction to negate the value, and cast the result back to Double
 * For the not operation, call the FearnRuntime.not() method.
 */
@Override
public void GenerateBytecode(MethodVisitor mv) {
    // Generate Instructions to place operand expression on top of stack
    operand.GenerateBytecode(mv);
    if (operator == ExprType.Negate) {
        // Cast to I or D (as applicable), and Negate
        if (((PrimitiveSpecififier)operand.expression_type).element_type ==
PrimitiveDataType.INT)
        {
            mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Integer", "intValue", "()I", false);
            mv.visitInsn(INEG);
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf",
"(I)Ljava/lang/Integer;", false);
        } else {
            mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Double", "doubleValue", "()D",
false);
            mv.visitInsn(DNEG);
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/Double", "valueOf",
"(D)Ljava/lang/Double;", false);
        }

    } else { // operator == ExprType.LogicalNot
        mv.visitMethodInsn(INVOKESTATIC, "FearnRuntime", "not",
"(Ljava/lang/Boolean;)Ljava/lang/Boolean;", false);
    }
}

/* To validate
 * -> For negation, the operand must be an int or float, otherwise an error is raised
 * -> For logical not, the operand must evaluate to a boolean value
 */
@Override
public TypeSpecififier validate(SymbolTable symTable) {

    TypeSpecififier operandType = operand.validate(symTable);
```

## The Fearn Programming Language

```
if (operator == ExprType.Negate) {
    if (operandType instanceof PrimitiveSpecifier)
    {
        switch (((PrimitiveSpecifier)operand.expression_type).element_type) {
            case INT: expression_type = new PrimitiveSpecifier(PrimitiveDataType.INT );
break;

            case FLOAT: expression_type = new PrimitiveSpecifier(PrimitiveDataType.FLOAT);
break;

            default: Reporter.ReportErrorAndExit(
                "Type Error: " + operand.toString() + " must be an INT or FLOAT value.",
                this
            );
        }
    } else Reporter.ReportErrorAndExit(
        "Type Error: " + operand.toString() + " must be an INT or FLOAT value.",
        this
    );
} else { // Logical Not
    if (
        operandType instanceof PrimitiveSpecifier
        && ((PrimitiveSpecifier)operand.expression_type).element_type ==
PrimitiveDataType.BOOL
    ) expression_type = new PrimitiveSpecifier(PrimitiveDataType.BOOL);
    else Reporter.ReportErrorAndExit(
        "Type Error: " + operand.toString() + " must be a Boolean value.", this);
}
return expression_type;
}
}
```

# The Fearn Programming Language

## ast.statement

ast.statement.Statement

```
package ast.statement;
```

```
import java.util.ArrayList;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import ast.ASTNode;
```

```
import ast.type.TypeSpecifier;
```

```
import codegen.CodeGenerator;
```

```
import semantics.table.SymbolTable;
```

```
/* Statement.java
```

```
*
```

```
* Abstract class representing a Statement in Fearn.
```

```
*
```

```
* Fields:
```

```
* -> toString(): Abstract method, that returns a string representation of the node, in  
* the syntax of Fearn
```

```
* -> GenerateBytecode(): Method that generates the bytecode that will perform the  
* functionality of the statement.
```

```
* -> validate(): Method that checks that a Statement is syntactically valid, raising  
* an error if not
```

```
* -> GetLocalDescriptors(): Protected method that iterates through the TypeSpecifiers  
* for all local variables, converting them to descriptors and returning them as  
* an Object array.
```

```
* -> This is used for when the visitField method is used, after jumps in the bytecode,  
* to specify the state of the stack frame, at runtime.
```

```
*/
```

```
public abstract class Statement extends ASTNode {
```

```
    @Override public abstract String toString();
```

```
    public abstract void GenerateBytecode(MethodVisitor mv);
```

```
    public abstract void validate(SymbolTable symbolTable);
```

## The Fearn Programming Language

```
protected Object[] GetLocalDescriptors()
{
    ArrayList<String> descriptors = new ArrayList<String>();

    for (TypeSpecifier spec : CodeGenerator.LocalSymbolTable.GetAllVarTypeSpecifiers())
    {
        String desc = SymbolTable.GenBasicDescriptor(spec);
        // The descriptors for objects (e.g. Ljava/lang/Integer;) need to be
        // converted into just the class name (java/lang/Integer)
        if (!desc.startsWith "["))
            desc = desc.substring(1, desc.length() - 1);
        descriptors.add(desc);
    }

    Object[] locals = descriptors.toArray();

    return locals;
}
```



# The Fearn Programming Language

```
ast.statement.CompoundStatement
```

```
package ast.statement;
```

```
import java.util.ArrayList;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import ast.ASTNode;
```

```
import ast.Declaration;
```

```
import semantics.table.SymbolTable;
```

```
/* CompoundStatement.java
```

```
 *
```

```
 * Represents a compound statement, a collection of statements within curly brackets.
```

```
 *
```

```
 * Contains nested_nodes, Statements and Declarations that exist within the scope of the
```

```
 * block. It also has boolean values includesJump and includesReturn.
```

```
 */
```

```
public class CompoundStatement extends Statement {
```

```
    public ArrayList<ASTNode> nested_nodes;
```

```
    public Boolean includesJump = false;
```

```
    public Boolean includesReturn = false;
```

```
    public CompoundStatement(ArrayList<ASTNode> nodes)
```

```
    {
```

```
        nested_nodes = nodes;
```

```
    }
```

```
    @Override public String toString()
```

```
    {
```

```
        return "{" + nested_nodes.toString() + "}";
```

```
    }
```

```
    public void GenerateBytecode(MethodVisitor mv)
```

```
    {
```

```
        /* Iterates through nested nodes, generating their bytecode
```

```
        * by calling the common GenerateBytecode method
```

```
        *
```

```
        * A small optimisation is performed here: if a JumpStatement is encountered,
```

```
        * the code beyond is unreachable, and so is not generated.
```

```
        *
```

```
        */
```

## The Fearn Programming Language

```
for (ASTNode node : nested_nodes)
{
    if (node instanceof Declaration) ((Declaration)node).GenerateBytecode(mv);
    else ((Statement)node).GenerateBytecode(mv);

    if (node instanceof JumpStatement) return;
}
}

public void validate(SymbolTable symbolTable) {
    /* Iterate nested_nodes, and validates them
    *
    * includesJump and includesReturn are also set at this point.
    *
    * These exist both for validation and code generation reasons:
    * -> If this CompoundStatement is the body of a function, includesReturn
    *      must be true.
    * -> If this CompoundStatement is part of an if-else statement, includesJump
    *      indicates no more code should be generated to finish off the block.
    */
    for (ASTNode node : nested_nodes)
    {
        if (node instanceof Declaration) ((Declaration)node).validate(symbolTable);
        else ((Statement)node).validate(symbolTable);

        if (node instanceof JumpStatement) includesJump = true;
        if (node instanceof ReturnStatement) includesReturn = true;
    }
}
}
```

# The Fearn Programming Language

```
ast.statement.ExpressionStatement
```

```
package ast.statement;
```

```
import static org.objectweb.asm.Opcodes.*;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import ast.expression.Expression;
```

```
import ast.expression.FnCallExpression;
```

```
import ast.expression.IncrExpression;
```

```
import semantics.table.SymbolTable;
```

```
import util.Reporter;
```

```
/* ExpressionStatement.java
```

```
*
```

```
* Represents an Expression Statement in the AST. It contains
```

```
* -> The expression
```

```
* -> isAssign: a boolean value indicating if the expression
```

```
* is an assignment
```

```
*/
```

```
public class ExpressionStatement extends Statement {
```

```
    public Expression expression;
```

```
    public Boolean isAssign;
```

```
    public ExpressionStatement(Expression expr, Boolean assign)
```

```
    {
```

```
        expression = expr;
```

```
        isAssign = assign;
```

```
    }
```

```
    @Override public String toString()
```

```
    {
```

```
        return expression.toString() + ";;";
```

```
    }
```

```
/* The GenerateBytecode method simply generates the bytecode for the expression.
```

```
*
```

```
* If the expression doesn't evaluate to null (hence evaluates to some data, still on  
* the stack), a POP instruction is added on the end to remove the value.
```

```
*
```

```
*/
```

## The Fearn Programming Language

```
@Override
public void GenerateBytecode(MethodVisitor mv) {
    expression.GenerateBytecode(mv);
    // If the expression evaluates to a value, pop from operand stack
    if (expression.expression_type != null) mv.visitInsn(POP);
}

/* To validate, the expression is validated. The expression is also
 * checked to ensure it's valid as a statement, on its own. This means
 * it is either
 * -> An Assignment
 * -> A Function Call
 * -> An Increment Expression
 */
public void validate(SymbolTable symbolTable) {

    if (!isAssign
        && !(expression instanceof FnCallExpression || expression instanceof IncrExpression)
    ) Reporter.ReportErrorAndExit("Invalid Statement.", this);

    expression.validate(symbolTable);
}
}
```

# The Fearn Programming Language

ast.statement.IterationStatement

```
package ast.statement;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import static org.objectweb.asm.Opcodes.*;
```

```
import org.objectweb.asm.Label;
```

```
import ast.ASTNode;
```

```
import ast.Declaration;
```

```
import ast.expression.*;
```

```
import ast.type.PrimitiveSpecifier.PrimitiveDataType;
```

```
import ast.type.PrimitiveSpecifier;
```

```
import codegen.CodeGenerator;
```

```
import semantics.table.SymbolTable;
```

```
import util.Reporter;
```

```
/* IterationStatement.java
```

```
*
```

```
* Represents a for loop in the AST.
```

```
*
```

```
* The class contains an init_expression (an Expression or Declaration), a
```

```
* continue_expression (A boolean expression, determining whether the loop
```

```
* should run), and an iteration_expression (running at the end of each loop).
```

```
*
```

```
* It also contains the loop body, which runs every iteration.
```

```
*
```

```
*/
```

```
public class IterationStatement extends Statement {
```

```
    public ASTNode init_expression;
```

```
    public Expression continue_expression;
```

```
    public Expression iteration_expression;
```

```
    public CompoundStatement body;
```

```
    public IterationStatement(
```

```
        ASTNode init, Expression c_expr, Expression i_expr, CompoundStatement bod
```

```
) {
```

```
    init_expression = init;
```

```
    continue_expression = c_expr;
```

```
    iteration_expression = i_expr;
```

```
    body = bod;
```

```
}
```

## The Fearn Programming Language

```
@Override public String toString()
{
    String d, c, i;
    if (init_expression == null) d = ""; else d = init_expression.toString();
    if (continue_expression == null) c = ""; else c = continue_expression.toString();
    if (iteration_expression == null) i = ""; else i = iteration_expression.toString();

    return String.format("for ( %s ; %s ; %s )", d, c, i);
}
```

```
public void GenerateBytecode(MethodVisitor mv) {

    /* This requires three labels, one at the top, one at the end of the body (before
    * the iteration statement), and another at the bottom. The second label is used
    * as a target for continue statements.
    *
    * These labels are added to a stack, in the order (START, ITERATE, END), so they
    * can be referenced by jump statements `continue` and `return`.
    *
    * First, generate the init_expression, and visit the start label.
    *
    * Then, generate the continue expression, and IFEQ (if value at top of stack
    * equals 0 [false]) skip to the end label (don't run the body). Then,
    * Generate the body. Visit the second label, and generate iteration statement.
    * Finally, GOTO start.
    *
    * Visit the end label.
    *
    * After the start label, mv.visitFrame(Opcodes.F_FULL);
    * -> This common procedure is a protected method of the Statement class.
    * -> After the other labels, mv.visitFrame(Opcodes.F_SAME, 0, null, 0, null)
    *     is needed (frame remains the same)
    *
    * Remember, all variables are initialised null at the start of any function, so
    * the number of locals should never really change.
    */

    Label startLoopLabel = new Label();
    Label endBodyLabel = new Label();
    Label endLoopLabel = new Label();

    CodeGenerator.LabelStack.add(startLoopLabel);
    CodeGenerator.LabelStack.add(endBodyLabel);
    CodeGenerator.LabelStack.add(endLoopLabel);
}
```

## The Fearn Programming Language

```
Object[] locals = GetLocalDescriptors();
int numLocals = locals.length;

if (init_expression != null) {
    if (init_expression instanceof Expression)
    {
        ((Expression)init_expression).GenerateBytecode(mv);
        if (((Expression)init_expression).expression_type != null) mv.visitInsn(POP);
    } else ((Declaration)init_expression).GenerateBytecode(mv);
}

// Visit start of loop
mv.visitLabel(startLoopLabel);

// Verify Frame State
mv.visitFrame(F_FULL, numLocals, locals, 0, new Object[] {});

// Generate continue expression, and cast to primitive boolean (Z)
continue_expression.GenerateBytecode(mv);
mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Boolean", "booleanValue", "()Z", false);

// If the condition is false, skip the body, to the end label
mv.visitJumpInsn(IFEQ, endLoopLabel);

// Generate the body
body.GenerateBytecode(mv);

// Visit end of body
mv.visitLabel(endBodyLabel);

// Verify Frame State (Same Frame)
mv.visitFrame(F_SAME, 0, null, 0, null);

// Generate iteration_expression (which runs at the end of every loop)
if (iteration_expression != null) {
    iteration_expression.GenerateBytecode(mv);
    if (iteration_expression.expression_type != null) mv.visitInsn(POP);
}

// Return to the start of the loop
mv.visitJumpInsn(GOTO, startLoopLabel);

// Visit end of loop
mv.visitLabel(endLoopLabel);
mv.visitFrame(F_SAME, 0, null, 0, null);
```

## The Fearn Programming Language

```
// Remove the labels from the global LabelStack
CodeGenerator.LabelStack.pop();
CodeGenerator.LabelStack.pop();
CodeGenerator.LabelStack.pop();

}

public void validate(SymbolTable symbolTable) {
    /* -> The init expression can be either an expression, or a declaration
    * -> The continue expression must be a boolean value
    * -> The iteration expression and body must also be visited
    *
    * It must be noted that any one of these, except the continue expression
    * and body, can be null
    */

    // Increment Loop Depth, so jump statements like
    // break and continue nested within the body register as
    // valid when they are validated.
    CodeGenerator.loopDepth++;

    // If an initialisation expression is present, validate it
    if (init_expression != null)
    {
        if (init_expression instanceof Expression)
        {
            ((Expression)init_expression).validate(symbolTable);
        } else {
            ((Declaration)init_expression).validate(symbolTable);
        }
    }

    // If the continue condition (loop continues until condition evaluates false)
    // is null, raise an error
    if (continue_expression == null) Reporter.ReportErrorAndExit(
        "Iteration condition missing.", this
    );

    // Raise Error if the continue expression doesn't evaluate to a boolean value
    if (!continue_expression.validate(symbolTable).equals(
        new PrimitiveSpecifier(PrimitiveDataType.BOOL))
    ) {
        Reporter.ReportErrorAndExit("Iteration condition must be a boolean value.", this);
    }
}
```



## The Fearn Programming Language

```
// If an iteration expression is present (runs at the end of every loop, e.g. i++),
// validate it
if (iteration_expression != null) iteration_expression.validate(symbolTable);

// Validate the body of the loop
body.validate(symbolTable);

// Decrement loop depth back to original value
// (outside a loop, jump statements like break and continue
// are invalid).
CodeGenerator.loopDepth--;
}
}
```

# The Fearn Programming Language

```
ast.statement.JumpStatement
```

```
package ast.statement;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import codegen.CodeGenerator;
```

```
import static org.objectweb.asm.Opcodes.*;
```

```
import semantics.table.SymbolTable;
```

```
import util.Reporter;
```

```
/* JumpStatement.java
```

```
*
```

```
* Represents a jump statement (break, continue, return) in the AST. It is also  
* the superclass for the ReturnStatement class.
```

```
*
```

```
* It contains the JumpType enum, and a type property, which indicates the  
* functionality the jump performs.
```

```
*
```

```
*/
```

```
public class JumpStatement extends Statement {
```

```
    // Represents the type of jump being performed
```

```
    public enum JumpType
```

```
    {
```

```
        Continue,
```

```
        Break,
```

```
        Return
```

```
    }
```

```
    public JumpType type;
```

```
    public JumpStatement (JumpType t)
```

```
    {
```

```
        type = t;
```

```
    }
```

```
    @Override public String toString()
```

```
    {
```

```
        return type.name().toLowerCase() + ";;";
```

```
    }
```

## The Fearn Programming Language

```
public void GenerateBytecode(MethodVisitor mv) {
    /* This is the reason CodeGenerator.LabelStack exists
    *
    * If JumpType is Break, GOTO the last label (the end of the current loop)
    * on the stack. This should be peeked, to prevent it being removed.
    *
    * If JumpType is Continue, GOTO the second-to-last label.
    * This requires me to treat the stack like an array (something
    * that Java's flexible Stack type lets me do), getting the
    * label from index size() - 2.
    */

    try {
        switch (type) {
            case Break: mv.visitJumpInsn(GOTO, CodeGenerator.LabelStack.peek()); return;
            case Continue: mv.visitJumpInsn(GOTO, CodeGenerator.LabelStack.get(
                CodeGenerator.LabelStack.size() - 2
            )); return;
            default:
                break;
        }
    } catch (Exception e) {
        Reporter.ReportErrorAndExit(type.name() + " must be contained with a loop.", null);
    }
}

public void validate(SymbolTable symbolTable) {
    // If loop depth (the number of loops the traversal is currently in)
    // is > 0, statement is valid.
    if (CodeGenerator.loopDepth > 0) return;
    Reporter.ReportErrorAndExit(
        "Jump Statement " + type.name() + " is invalid outside a loop.",
        null
    );
}
}
```

# The Fearn Programming Language

```
ast.statement.ReturnStatement
```

```
package ast.statement;
```

```
import static org.objectweb.asm.Opcodes.*;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import ast.expression.Expression;
```

```
import codegen.CodeGenerator;
```

```
import semantics.table.SymbolTable;
```

```
import util.Reporter;
```

```
/* ReturnStatement.java
```

```
 *
```

```
 * Represents a return statement
```

```
 *
```

```
 * It contains the expression to return, which may be null.
```

```
 *
```

```
 */
```

```
public class ReturnStatement extends JumpStatement {
```

```
    Expression expression;
```

```
    public ReturnStatement(JumpType t, Expression e)
```

```
    {
```

```
        super(t);
```

```
        expression = e;
```

```
    }
```

```
    @Override public String toString()
```

```
    {
```

```
        return "return " + (expression == null ? "" : expression.toString()) + ";;";
```

```
    }
```

```
    /* To generate bytecode, use RETURN instruction if there is no expression to return.
```

```
     * Otherwise, evaluate the expression (generate its bytecode), and return the value
```

```
     * with ARETURN.
```

```
     */
```

```
    @Override
```

```
    public void GenerateBytecode(MethodVisitor mv) {
```

```
        if (expression == null) mv.visitInsn(RETURN);
```

```
        else {
```

```
            expression.GenerateBytecode(mv);
```

```
            mv.visitInsn(ARETURN);
```

```
        }
```

```
    }
```

## The Fearn Programming Language

```
public void validate(SymbolTable symbolTable) {
    // Get return type from Code Generator. Then, ensure the return is of the correct type

    /* This section is complicated by the fact that both expression
     * and CurrentReturnType can be null, causing the .equals() method
     * to throw an error.
     */

    if (CodeGenerator.CurrentReturnType == null)
    {
        // Expect expression to be null, error otherwise
        if (expression == null) return;
        else Reporter.ReportErrorAndExit("Incorrect return type, expected void", this);
    } else {
        // Expect type of expression to match current return type
        if (
            expression == null
            || !expression.validate(symbolTable).equals(CodeGenerator.CurrentReturnType)
        ) {
            Reporter.ReportErrorAndExit(
                "Incorrect return type, expected " + CodeGenerator.CurrentReturnType.toString(),
                this
            );
        }
        else return;
    }
}
```

# The Fearn Programming Language

```
ast.statement.SelectionStatement
```

```
package ast.statement;
```

```
import static org.objectweb.asm.Opcodes.*;
```

```
import org.objectweb.asm.Label;
```

```
import org.objectweb.asm.MethodVisitor;
```

```
import ast.expression.Expression;
```

```
import ast.type.PrimitiveSpecifier.PrimitiveDataType;
```

```
import ast.type.PrimitiveSpecifier;
```

```
import semantics.table.SymbolTable;
```

```
import util.Reporter;
```

```
/* SelectionStatement.java
```

```
 *
```

```
 * Represents an if(-else) statement.
```

```
 *
```

```
 * The class contains an expression condition, a boolean expression which
```

```
 * determines if the if_branch executes. The if_branch is a compound statement -
```

```
 * executing only if the expression is true. else_branch is a statement, either a
```

```
 * compound statement or another selection statement (for if-else chains), which
```

```
 * only executes if the condition evaluates to false.
```

```
 *
```

```
 */
```

```
public class SelectionStatement extends Statement {
```

```
    public Expression condition;
```

```
    public CompoundStatement if_branch;
```

```
    public Statement else_branch;
```

```
    public SelectionStatement(Expression cond, CompoundStatement if_body, Statement else_body)
```

```
    {
```

```
        condition = cond;
```

```
        if_branch = if_body;
```

```
        else_branch = else_body;
```

```
    }
```

## The Fearn Programming Language

```
@Override public String toString()
{
    if (else_branch == null)
    {
        return String.format(
            "if (%s) {...}",
            condition.toString()
        );
    }

    else if (else_branch instanceof SelectionStatement)
    {
        return String.format(
            "if (%s) {...} else %s",
            condition.toString(),
            else_branch.toString()
        );
    }
    else {
        return String.format(
            "if (%s) {...} else {...}",
            condition.toString()
        );
    }
}

@Override
public void GenerateBytecode(MethodVisitor mv) {

    /* This requires labels and GOTO statements.
     * First, I generate the condition, leaving a Boolean
     * object at the top of the stack. Then, I cast to
     * the primitive boolean type. If that equals 0, then
     * GOTO the label after the if_branch (either else or end,
     * depending on whether an else branch exists). Otherwise, fall
     * through to the if_branch body.
     */

    // Create array with type descriptors for all variables within stack

    // This is vital for verifying the state of the stack frame after any
    // unconditional jump statement

    Object[] locals = GetLocalDescriptors();
```

## The Fearn Programming Language

```
int numLocals = locals.length;

// Create labels
Label else_label = new Label();
Label end_label = new Label();

// Get Condition
condition.GenerateBytecode(mv);
mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Boolean", "booleanValue", "()Z", false);

// If equal to 0 (false), skip if_branch (jump to else - if defined - or end)
if (else_branch != null) mv.visitJumpInsn(IFEQ, else_label);
else mv.visitJumpInsn(IFEQ, end_label);

// Generate if branch (program falls through to this if expression evaluates to true)
if_branch.GenerateBytecode(mv);

// Only GOTO end if no jump statement exists within the if_branch body
if (!if_branch.includesJump) mv.visitJumpInsn(GOTO, end_label);

// If an else branch exists, Generate its bytecode here
if (else_branch != null)
{
    mv.visitLabel(else_label);
    mv.visitFrame(F_FULL, numLocals, locals, 0, new Object[] {});
    else_branch.GenerateBytecode(mv);
}
mv.visitLabel(end_label);
mv.visitFrame(F_FULL, numLocals, locals, 0, new Object[] {});
}

/* To validate, confirm the condition evaluates to a boolean value.
 * Validate if and else branches also.
 */
public void validate(SymbolTable symbolTable) {
    if(!condition.validate(symbolTable).equals(new
PrimitiveSpecifier(PrimitiveDataType.BOOL)))
        Reporter.ReportErrorAndExit(
            condition.toString() + ": Condition must be boolean.",
            null
        );
    if_branch.validate(symbolTable);
    if (else_branch != null) else_branch.validate(symbolTable);
}
}
```



# The Fearn Programming Language

## ast.function

ast.function.Function

```
package ast.function;
```

```
import java.util.ArrayList;
```

```
import ast.ASTNode;
```

```
import ast.statement.CompoundStatement;
```

```
import ast.type.TypeSpecifier;
```

```
import codegen.CodeGenerator;
```

```
import semantics.table.SymbolTable;
```

```
import util.Reporter;
```

```
/* Function.java
```

```
 *
```

```
 * Represents a function in the AST
```

```
 *
```

```
 * Fields:
```

```
 * -> identifier: string name of function (used as method name in the generated  
 *      program class)
```

```
 * -> parameters: A list of Parameter object, representing the values the  
 *      function takes in, and their local identifier
```

```
 * -> return_type: TypeSpecifier of the data value returned by the function (is  
 *      null for a void function)
```

```
 * -> is_void: A boolean flag, indicating if the function is void (returns no data)
```

```
 */
```

```
public class Function extends ASTNode {
```

```
    public String identifier;
```

```
    public ArrayList<Parameter> parameters;
```

```
    public TypeSpecifier return_type;
```

```
    public Boolean is_void;
```

```
    public CompoundStatement body;
```

```
    public Function(  
        String id,
```

```
        String id,
```

```
        ArrayList<Parameter> params,
```

```
        TypeSpecifier rt,
```

```
        Boolean _void,
```

```
        CompoundStatement bod
```

```
    )
```

## The Fearn Programming Language

```
{
    identifier = id;
    parameters = params;
    return_type = rt;
    is_void = _void;
    body = bod;

}

@Override public String toString()
{
    String ret_type_str = is_void ? "void" : return_type.toString();
    return String.format(
        "fn %s%s => %s {...}",
        identifier,
        parameters.toString().replace("[", "(").replace("]", ")"),
        ret_type_str
    );
}

/* No Function.GenerateBytecode() is provided, as the generation of
 * functions using the method visitor is performed by CodeGenerator,
 * as the program's Class Writer is needed to add the method to the
 * main program class (see CodeGenerator.java).
 */
public void validate(SymbolTable symbolTable) {
    /* Sets Current Return Type to the return type specifier
     * for this function. This is used during the traversal of
     * the body, as return statements must return an expression
     * that evaluates to the right type.
     *
     * The body is validated, and (assuming function is not void),
     * must include a return statement (the type of expression
     * returned is validated by the return statement itself,
     * using CurrentReturnType).
     */
    CodeGenerator.CurrentReturnType = return_type;
    body.validate(symbolTable);
    if (!body.includesReturn && return_type != null)
        Reporter.ReportErrorAndExit(
            "Function " + identifier + " must include a return statement in its main body.",
            null
        );
}
}
```

# The Fearn Programming Language

```
ast.function.Parameter
```

```
package ast.function;
```

```
import ast.ASTNode;
```

```
import ast.type.TypeSpecifier;
```

```
/* Parameter.java
```

```
 *
```

```
 * Represents a function parameter in the AST
```

```
 *
```

```
 * Fields:
```

```
 * -> identifier: string name of parameter, which can be used as a variable  
 *       within the function body
```

```
 * -> type: TypeSpecifier of the data value of the parameter (just like any  
 *       normal variable). Used to validate the type of arguments when function  
 *       is called
```

```
 */
```

```
public class Parameter extends ASTNode {
```

```
    public String identifier;
```

```
    public TypeSpecifier type;
```

```
    public Parameter(String id, TypeSpecifier t)
```

```
    {
```

```
        identifier = id;
```

```
        type = t;
```

```
    }
```

```
    @Override public String toString()
```

```
    {
```

```
        return String.format("%s : %s", type.toString(), identifier);
```

```
    }
```

```
}
```

# The Fearn Programming Language

## ast.type

ast.type.TypeSpecifier

```
package ast.type;
```

```
import ast.ASTNode;
```

```
/* TypeSpecifier.java
```

```
*
```

```
* The abstract superclass of all TypeSpecifier classes.
```

```
*
```

```
* TypeSpecifiers represent data types, associated with variables, function
```

```
* return types, etc. These can be compared to ensure a structure in the program
```

```
* is valid.
```

```
*
```

```
* It has a `type`, which is a Category from the below enum.
```

```
* -> Primitive Types are indivisible, such as ints, floats, strings, and
```

```
*      Booleans (bools)
```

```
* -> Array types are for arrays of primitive values or struct instances
```

```
* -> StructInstance represents the data is an instance of a user-defined struct
```

```
*
```

```
*/
```

```
public abstract class TypeSpecifier extends ASTNode {
```

```
    public enum Category
```

```
    {
```

```
        Primitive,
```

```
        Array,
```

```
        StructInstance
```

```
    }
```

```
    public Category type;
```

```
}
```

# The Fearn Programming Language

ast.type.PrimitiveSpecifier

```
package ast.type;
```

```
/* PrimitiveSpecifier.java
```

```
*
```

```
* TypeSpecifier to describe primitive data types. It uses a PrimitiveDataType
```

```
* enum, which describes the data type as an INT, FLOAT, STR, or BOOL.
```

```
*
```

```
*/
```

```
public class PrimitiveSpecifier extends TypeSpecifier {
```

```
    public enum PrimitiveDataType {
```

```
        INT,
```

```
        FLOAT,
```

```
        BOOL,
```

```
        STR
```

```
    }
```

```
    public PrimitiveDataType element_type;
```

```
    public PrimitiveSpecifier(PrimitiveDataType eleT)
```

```
    {
```

```
        type = Category.Primitive;
```

```
        element_type = eleT;
```

```
    }
```

```
    @Override
```

```
    public String toString()
```

```
    {
```

```
        return element_type.name().toLowerCase();
```

```
    }
```

```
}
```

# The Fearn Programming Language

ast.type.StructInstanceSpecifier

```
package ast.type;
```

```
/* StructInstanceSpecifier.java
```

```
 *
```

```
 * Represents the data type for the instance of a struct. It contains the name  
 * of the struct.
```

```
 *
```

```
 */
```

```
public class StructInstanceSpecifier extends TypeSpecifier {
```

```
    public String name;
```

```
    public StructInstanceSpecifier(String structName)
```

```
    {
```

```
        type = Category.StructInstance;
```

```
        name = structName;
```

```
    }
```

```
    @Override
```

```
    public String toString()
```

```
    {
```

```
        return String.format("%s", name);
```

```
    }
```

```
}
```

## The Fearn Programming Language

ast.type.ArraySpecifier

```
package ast.type;
```

```
/* ArraySpecifier.java
```

```
 *
 * Used to specify an array data type (e.g. int[]). This includes the
 * TypeSpecifier of the elements, and the number of dimensions.
 *
 */
```

```
public class ArraySpecifier extends TypeSpecifier {
```

```
    public TypeSpecifier element_type;
```

```
    public Integer dimensionCount;
```

```
    public ArraySpecifier(TypeSpecifier eleT, Integer dims)
```

```
    {
        type = Category.Array;
        element_type = eleT;
        dimensionCount = dims;
    }
```

```
    @Override
```

```
    public String toString()
```

```
    {
        return element_type.toString() + "[".repeat(dimensionCount);
    }
```

```
}
```

# The Fearn Programming Language

```
ast.type.ArrayBodySpecifier
```

```
package ast.type;
```

```
import java.util.ArrayList;
```

```
/* ArrayBodySpecifier.java
```

```
 *
```

```
 * This describes the type of an arraybody, and is used to ensure  
 * multi-dimensional arrays have consistent dimensions.
```

```
 *
```

```
 * It contains a TypeSpecifier for the elements, and a list of dimensions.  
 * (e.g. for a 2-D array body with 2 arrays, each containing 3 elements,  
 * the body would have a specifier [2, 3])
```

```
 */
```

```
public class ArrayBodySpecifier extends TypeSpecifier {
```

```
    public TypeSpecifier element_type;
```

```
    public ArrayList<Integer> dimensions;
```

```
    public ArrayBodySpecifier(TypeSpecifier eleT, ArrayList<Integer> dims)
```

```
    {
```

```
        element_type = eleT;
```

```
        dimensions = dims;
```

```
    }
```

```
    @Override
```

```
    public String toString()
```

```
    {
```

```
        return element_type.toString() + dimensions.toString();
```

```
    }
```

```
}
```



# The Fearn Programming Language

## semantics.table

### semantics.table.SymbolTable

```
package semantics.table;

import java.util.ArrayList;
import java.util.NoSuchElementException;

import ast.function.Parameter;
import ast.type.*;
import codegen.CodeGenerator;
import util.Reporter;

/*
 * SymbolTable.java
 *
 * This class represents a Symbol Table, used to keep
 * track of variables, functions, and structs used in a
 * program.
 *
 * SymbolTable objects are composed of rows. The methods
 * are used to add rows, and query their types, owners, etc.
 *
 * Its methods also serve the role of detecting when a symbol
 * has not been declared, raising a relevant error. A majority
 * of the below methods will raise an error if their function
 * fails, using the ReportErrorAndExit() function.
 */

public class SymbolTable {

    private ArrayList<Row> Rows = new ArrayList<Row>();

    /* General Methods */

    public void addRow(Row new_row)
    {
        // Add a Single Row object, checking there's no clash
        // with rows already in the table
        // Raise error if two symbols (of the same type) in the same
        // scope (table) have the same identifier
    }
}
```

## The Fearn Programming Language

```
for (Row r : Rows)
{
    if (
        new_row.getClass() == r.getClass()
        && r.identifier.equals(new_row.identifier)
    ) {
        Reporter.ReportErrorAndExit(
            "Symbol " + new_row.identifier + " can only exist once within scope.",
            null
        );
    }
}

// Set owner
// The owner represents the generated class the row belongs to
// E.g. The owner of a function, defined in lib.test, will become
// a method in the 'lib' class, and so its owner is 'lib'
new_row.owner = CodeGenerator.GeneratorStack.peek().programName;
Rows.add(new_row);
}

// Add Rows from a Symbol Table (used to add rows symbols
// from imported files/modules)
public void addRowsFromTable(SymbolTable table) {
    for (Row r : table.GetAllRows()) Rows.add(r);
}

// Return all rows
public ArrayList<Row> GetAllRows()
{
    return Rows;
}

/* GenBasicDescriptor
 *
 * This generates the type descriptors for int, float, bool, and str
 * types in Fearn, which are translated to Integer, Double, Boolean,
 * and String Java objects.
 *
 * It also recursively build type descriptors for arrays, and generate
 * struct descriptors using their class name ($IDENTIFIER).
 *
 * These are generated from TypeSpecifier objects - which are used by FearnC
 * to describe data types.
 *
 * These type descriptor strings are used to specify the type of
 * elements to the JVM.
 */
```

## The Fearn Programming Language

```
static public String GenBasicDescriptor(TypeSpecifier typeSpecifier)
{
    String type_descriptor = "";

    if (typeSpecifier.getClass() == PrimitiveSpecifier.class)
    {
        switch ( ((PrimitiveSpecifier)typeSpecifier).element_type ) {
            case INT : type_descriptor += "Ljava/lang/Integer;" ; break;
            case FLOAT: type_descriptor += "Ljava/lang/Double;" ; break;
            case STR : type_descriptor += "Ljava/lang/String;" ; break;
            case BOOL : type_descriptor += "Ljava/lang/Boolean;" ; break;
            default: break;
        }
    }

    else if (typeSpecifier.getClass() == ArraySpecifier.class)
    {
        type_descriptor += type_descriptor += "[".repeat(
            ((ArraySpecifier)typeSpecifier).dimensionCount
        );
        type_descriptor += GenBasicDescriptor(((ArraySpecifier)typeSpecifier).element_type);
    }

    else if (typeSpecifier.getClass() == ArrayBodySpecifier.class)
    {
        type_descriptor += "[";
        type_descriptor += GenBasicDescriptor(
            ((ArrayBodySpecifier)typeSpecifier).element_type);
    }

    else // Struct Instance
    {
        type_descriptor += "L$" + ( (StructInstanceSpecifier)typeSpecifier ).name + ";";
    }

    return type_descriptor;
}
```

## The Fearn Programming Language

```
// Private method to retrieve a row in the table, by performing a linear search
// Throws an exception if it is not found
private Row GetRow(String id, Boolean isFunction, Boolean isStruct)
    throws NoSuchElementException
{
    for (Row r : Rows)
    {

        if (r.identifier.equals(id) && r instanceof FunctionRow && isFunction)
            return r;

        else if (r.identifier.equals(id) && r instanceof StructRow && isStruct)
            return r;

        else if (r.identifier.equals(id) && r instanceof VariableRow
            && !isFunction && !isStruct
        )
            return r;
    }
    throw new NoSuchElementException(id);
}

// Retrieves the Type Specifier associated with a row in the table (e.g. variable
// data type, function return type).
public TypeSpecifier GetTypeSpecifier(String id, Boolean isFunction) {
    try {
        Row row = GetRow(id, isFunction, false);
        if (isFunction) return ((FunctionRow)row).return_type;
        else return ((VariableRow)row).dataType;
    } catch (Exception e) {
        Reporter.ReportErrorAndExit(
            "Definition for " + id + " not provided in scope.",
            null
        );
    }
    return null;
}

// Returns true if identifier is contained within the table
public Boolean Contains(String id) {
    for (int i = 0; i < Rows.size(); i++)
    {
        if (Rows.get(i).identifier.equals(id)) { return true; }
    }
    return false;
}
```

# The Fearn Programming Language

```
// Get the owner associated with an identifier
public String GetOwner(String id, Boolean isFunction)
{
    try {
        return GetRow(id, isFunction, false).owner;
    } catch (Exception e) {
        Reporter.ReportErrorAndExit("Owner for " + id + " not found.", null);
    }
    return null;
}

/* Variable Methods */

// Get the string type descriptor of a variable in the table
public String GetVarDescriptor(String id) {
    try {
        return GetRow(id, false, false).descriptor;
    } catch (Exception e) {
        Reporter.ReportErrorAndExit("Unknown Variable " + id, null);
    }

    return null;
}

/* Gets the index of a variable in the table
 *
 * These indexes are used in Code Generation, as the JVM
 * stores local variables at indexes, separate to the stack.
 *
 * Since each variable in a function's symbol table is stored
 * at a different index, starting from 0, the index in the table is
 * used as the index for the local variable in the stack frame, at
 * runtime.
 */

public Integer GetIndex(String id) {
    for (int i = 0; i < Rows.size(); i++)
    {
        if (Rows.get(i).identifier.equals(id)) { return i; }
    }

    Reporter.ReportErrorAndExit("Unknown Variable " + id, null);
    return null;
}
```

# The Fearn Programming Language

```
/* Function Methods */

/* Generates the method descriptor for a function.
 *
 * Functions in FearnLang are modelled as public static methods
 * of the class representing the module/script they are defined in
 * (e.g. a function f() => void, defined in program.fearn, would
 * become `public static void f()`, in the `program` class).
 *
 * In the JVM, as with any typed element, each method has a descriptor
 * to define the types of its arguments, and its return type.
 *
 * Below, The parameter's types are all generated and appended to the descriptor,
 * along with the descriptor for the return type. If a function is void (return_type
 * is null), `V` is used instead.
 */

static public String GenFuncDescriptor(ArrayList<Parameter> params, TypeSpecifier return_type)
{
    String desc = "(";
    for (Parameter p : params) desc += GenBasicDescriptor(p.type);

    desc += ")";

    if (return_type == null) desc += "V";
    else desc += GenBasicDescriptor(return_type);

    return desc;
}

// Get the function's local symbol table, containing its local variables.
public SymbolTable GetFuncSymbolTable(String id) {

    try {
        return ((FunctionRow)GetRow(id, true, false)).localSymbolTable;
    } catch (Exception e) {
        Reporter.ReportErrorAndExit("Symbol Table for function " + id + " not found.", null);
    }

    return null;
}
```

## The Fearn Programming Language

```
// Get a function's method descriptor
public String GetGlobalFuncDescriptor(String id) {

    try {
        return GetRow(id, true, false).descriptor;
    } catch (Exception e) {
        Reporter.ReportErrorAndExit("Descriptor for function " + id + " not found.", null);
    }

    return null;
}

/*
 * Gets the TypeSpecifier objects associated with the arguments of a function
 * This is called when the program has attempted to call a function, and raises
 * an error if the function has not been found (indicating the program has not
 * defined it)
 */
public ArrayList<TypeSpecifier> GetFuncParameterSpecifiers(String id)
{
    ArrayList<TypeSpecifier> t_list = new ArrayList<TypeSpecifier>();

    try {
        for (Parameter p : ((FunctionRow)GetRow(id, true, false)).parameters)
            t_list.add(p.type);
        return t_list;
    } catch (Exception e) {
        Reporter.ReportErrorAndExit("Function " + id + " is not defined.", null);
    }

    return t_list;
}

/* Struct Methods */
// Get the TypeSpecifiers for the attributes of a struct (used
// to check the types of the values used to initialise a struct)
public ArrayList<TypeSpecifier> GetStructAttributeSpecifiers(String id)
{
    try {
        return ((StructRow)GetRow(
            id, false, true
        )).localSymbolTable.GetAllVarTypeSpecifiers();
    } catch (Exception e) {
        Reporter.ReportErrorAndExit("Struct " + id + " not defined.", null);
    }

    return null;
}
```

## The Fearn Programming Language

```
/*
```

Generate method descriptor for constructor for the class  
that represents the struct

E.g. Given a struct ...

```
struct myStruct
{
    let x : int;
}
```

... a class called ``$myStruct`` is generated (a ``$`` is added to differentiate between program classes and struct classes), with its constructor being ...

```
public $myStruct(Integer var0)
```

```
*/
```

```
public static String GenStructDescriptor(SymbolTable localTable)
{
    String desc = "(";

    for (TypeSpecifier t : localTable.GetAllVarTypeSpecifiers())
        desc += GenBasicDescriptor(t);

    desc += ")V";
    return desc;
}
```

```
// Get the TypeSpecifiers of all variables in a table
```

```
public ArrayList<TypeSpecifier> GetAllVarTypeSpecifiers()
{
    ArrayList<TypeSpecifier> t_list = new ArrayList<TypeSpecifier>();

    for (Row r : Rows)
    {
        if (r.getClass() == VariableRow.class)
        {
            t_list.add(((VariableRow)r).dataType);
        }
    }

    return t_list;
}
```



## The Fearn Programming Language

```
// Get Struct class constructor method descriptor
public String GetGlobalStructDescriptor(String id) {

    try {
        return GetRow(id, false, true).descriptor;
    } catch (Exception e) {
        Reporter.ReportErrorAndExit("Descriptor for struct " + id + " not found.", null);
    }

    return null;
}

// Get Struct's local Symbol Table (containing its attributes)
public SymbolTable GetStructSymbolTable(String id) {

    try {
        return ((StructRow)GetRow(id, false, true)).localSymbolTable;
    } catch (Exception e) {
        Reporter.ReportErrorAndExit("Symbol Table for struct " + id + " not found.", null);
    }

    return null;
}
}
```

# The Fearn Programming Language

## semantics.table.Row

```
package semantics.table;

/*
 * Row.java
 *
 * Abstract super class of all rows that can exist with a symbol table
 * Includes:
 *   -> identifier - the symbol's name in the program
 *   -> descriptor - the symbol's type/method descriptor
 *   -> owner      - the program class the symbol belongs to
 */
public abstract class Row {

    public String identifier;
    public String descriptor;
    public String owner = null;

    public Row(String id)
    {
        identifier = id;
    }
}
```

## semantics.table.VariableRow

```
package semantics.table;
import ast.type.TypeSpecifier;

/*
 * Variable.java
 *
 * Represents a variable, within either the global symbol table,
 * or the local symbol table of a function/struct
 * Includes:
 *   -> dataType - the type specifier, representing the variable's data type
 */
public class VariableRow extends Row {
    TypeSpecifier dataType;
    public VariableRow(String id, TypeSpecifier type)
    {
        super(id);
        dataType = type;
        // Generate type descriptor, representing variable's type
        // to the JVM
        descriptor = SymbolTable.GenBasicDescriptor(dataType);
    }
}
```

# The Fearn Programming Language

semantics.table.FunctionRow

```
package semantics.table;

import java.util.ArrayList;
import ast.function.*;
import ast.type.TypeSpecifier;

/*
 * FunctionRow.java
 *
 * Represents a function, within the global symbol table
 * Includes:
 *   -> parameters - A list of Parameter objects, representing the function parameters
 *   -> return_type - the typeSpecifier of the value the function returns (null is the
 *       function is void)
 *   -> localSymbolTable - a table of the local variables that only exist within the
 *       scope of the function
 */

public class FunctionRow extends Row {
    ArrayList<Parameter> parameters;
    TypeSpecifier return_type;
    SymbolTable localSymbolTable = new SymbolTable();

    public FunctionRow(
        String identifier, ArrayList<Parameter> params,
        TypeSpecifier rt, SymbolTable local
    ) {
        super(identifier);
        parameters = params;
        return_type = rt;
        // Generate method descriptor
        descriptor = SymbolTable.GenFuncDescriptor(params, rt);
        localSymbolTable = local;
    }
}
```

## The Fearn Programming Language

semantics.table.StructRow

```
package semantics.table;
/*
 * StructRow.java
 *
 * Represents a struct, within the global symbol table
 * Includes:
 *   -> localSymbolTable - a table of the struct's attributes
 */
public class StructRow extends Row {

    public SymbolTable localSymbolTable = new SymbolTable();

    public StructRow(String id, SymbolTable symtab)
    {
        super(id);
        localSymbolTable = symtab;
        // Generate method descriptor for struct class's constructor
        descriptor = SymbolTable.GenStructDescriptor(symtab);
    }
}
```

# The Fearn Programming Language

## codegen

### codegen.CodeGenerator

```
package codegen;

import org.objectweb.asm.*;
import org.objectweb.asm.tree.MethodNode;

import static org.objectweb.asm.Opcodes.*;

import ast.Declaration;
import ast.Program;
import ast.Struct;
import ast.function.Function;
import ast.type.TypeSpecifier;
import semantics.table.SymbolTable;

import util.Reporter;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Stack;

/* CodeGenerator.java
 *
 * This class is the primary class used to generate object code.
 *
 * It has multiple public, static properties used during semantic
 * analysis and code generation, which is useful for ensuring data
 * is available globally, for any point in the program to access, or
 * modify if needs be.
 *
 * -> buildPath: The path to the build directory. This is set once,
 * at the start of compilation, and used by every CodeGenerator
 * object, to ensure all classes are generated in the same place
 *
 * -> GeneratorStack: A stack of generator objects, used to easily
 * reference the generator currently in use, specifically to ensure
 * the correct program name is being used (each import will have its
 * own name, which becomes the name of the class generated to represent
 * it; to ensure that the owners of class elements refer to the correct
 * class, it's vital the program name of the program currently being
 * generated is accessible at all times).
```

## The Fearn Programming Language

```
* -> GlobalSymbolTable: The symbol table for the global scope of the
*     program. Vital for locating details on global elements of a program
*     (e.g. functions, structs, global variables).
* -> LocalSymbolTable: The symbol table for the function currently being
*     generated. This is vital for getting variable descriptors and indexes.
* -> CurrentReturnType: Return type for function being generated. This is
*     used to verify the expression types for return statements.
* -> loopDepth: Used during semantic analysis to ensure break/continue
*     statements are used within loops (when loopDepth > 0)
* -> LabelStack: A stack used during code generation, to provide break/continue
*     statements with the most recent loop labels, to reference where in the
*     program to jump
*/
```

```
public class CodeGenerator {

    public static Path buildPath;
    public static Stack<CodeGenerator> GeneratorStack = new Stack<>();
    public static SymbolTable GlobalSymbolTable;
    public static SymbolTable LocalSymbolTable;
    public static TypeSpecifier CurrentReturnType;
    public static Integer loopDepth = 0;
    public static Stack<Label> LabelStack = new Stack<Label>();

    // Name of Program being generated by this object, which is used as the identifier
    // for the program class (the .class file for the program, generated using ASM)
    public String programName;

    /* GenerateStructs
    *
    * Method to generate struct class files, for each struct in a program.
    * The method iterates through each struct (in a provided ArrayList) ...
    * 1) A new class writer object to created, to write the struct class
    * 2) A method visitor for the constructor (cv) is created, to write
    *     the constructor method. This is created using the descriptor
    *     from the global symbol table
    * 3) The constructor invokes Java's default object constructor first
    * 4) For each attribute (declaration) in the struct, add a public field
    *     to the struct class, and assigns the correct parameter of the
    *     constructor to that property
    * 5) Add RETURN instruction to end of constructor
    * 6) Output byte array, representing struct class, into a new .class file
    *     in the buildPath directory
    * 7) Report successful generation of struct class file
    */
}
```

# The Fearn Programming Language

```
private void GenerateStructs(ArrayList<Struct> structs)
{
    structs.forEach(
        (struct) -> {

            ClassWriter classWriter = new ClassWriter(ClassWriter.COMPUTE_MAXS);

            classWriter.visit(
                V19,
                // Defines access
                ACC_PUBLIC | ACC_SUPER,
                // Define class name
                // (Fearn struct class names have a $ prefix, to distinguish them from program
                // files)
                "$"+struct.identifier,
                null,
                // All Java objects derive from the base Object class
                "java/lang/Object",
                null
            );

            MethodVisitor cv = classWriter.visitMethod(
                ACC_PUBLIC,
                "<init>", // `<init>` indicates the constructor
                GlobalSymbolTable.GetGlobalStructDescriptor(struct.identifier),
                null,
                null
            );
            cv.visitCode();
            // Invoke default constructor
            cv.visitVarInsn(ALOAD, 0);
            cv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Object", "<init>", "()V", false);

            Integer i = 0;
            for (Declaration decl : struct.declarations)
            {
                // Create public field for attribute
                String descriptor = SymbolTable.GenBasicDescriptor(decl.type);
                classWriter.visitField(
                    ACC_PUBLIC,
                    decl.identifier,
                    descriptor,
                    null,
                    null
                );
            }
        }
    );
}
```

## The Fearn Programming Language

```
// Generate Constructor Instructions that take in an argument,
// and load it into the attribute, specified by decl

// Load 'this' (the struct instance itself) to operand stack
cv.visitVarInsn(ALOAD, 0);

// Load argument to operand stack
cv.visitVarInsn(ALOAD, ++i);

// Assign argument to Field
cv.visitFieldInsn(
    PUTFIELD,
    "$"+struct.identifier,
    decl.identifier,
    SymbolTable.GenBasicDescriptor(decl.type)
);
}

cv.visitInsn(RETURN);
cv.visitMaxs(0, 0);
cv.visitEnd();

classWriter.visitEnd();

Path destination = Paths.get(
    buildPath.toString(),
    String.format("%s.class",
        struct.identifier)
);

try {
    Files.write(destination, classWriter.toByteArray());
} catch (IOException e) {
    Reporter.ReportErrorAndExit("Struct Gen Error :- " + e.toString(), null);
}

Reporter.ReportSuccess(
    "GENERATED Struct File : " + destination.toAbsolutePath() + ";",
    false
);
}

);
}
```



# The Fearn Programming Language

```
/* GenerateProgram
 *
 * Method to generate program class files
 * 1) Use a class writer to create a public, static class, representing the program
 * 2) Create a static block (<clinit>), to define the default states of global variables
 * 3) For each global variable in the program ...
 *    -> Add a public, static field to the class
 *    -> If an initialisation expression has been provided, generate the expression's
 *        bytecode (putting the value of the expression at the top of the operand stack),
 *        and put the value into the field of the program class
 * 4) Add a default constructor, which simply invokes the default object constructor
 * 5) For each function in the program ...
 *    -> Add a method representing it to the program class (uses SymbolTable to get method
 *        descriptor), and create a FearnMethodVisitor to generate code for it
 *    -> Create a MethodNode for the function (effectively a list of bytecode instructions)
 *    -> Set LocalSymbolTable to the symbol table for this function
 *    -> Loop through all local variable indexes, initialising them all to null
 *    -> Generate function body bytecode, writing the instructions to the MethodNode
 *    -> Eliminate Redundant Casting in the MethodNode (see CastOptimiser.java)
 *    -> Use the FearnMethodVisitor to visit every instruction in the MethodNode, adding
 *        them to the actual class method
 * 6) Write program class to .class file
 * 7) Report Success
 */
```

```
private void GenerateProgram(
    ArrayList<Function> functions, ArrayList<Declaration> global_declarations,
    Path finalProgramPath)
{
    // Create new class writer, to write program class
    ClassWriter classWriter = new ClassWriter(ClassWriter.COMPUTE_MAXS);

    // Class is a public, static class, with its name being the programName
    classWriter.visit(
        V19,
        ACC_PUBLIC | ACC_SUPER,
        programName,
        null,
        "java/lang/Object",
        null
    );
};
```

## The Fearn Programming Language

```
// Generate Initial State of Global Variables (sv = StateVisitor)
MethodVisitor sv = classWriter.visitMethod(
    ACC_STATIC,
    "<clinit>", // Indicates static block
    "()V", // Takes no arguments, and has void return type
    null,
    null
);

// Begin Defining Code
sv.visitCode();

// Add Global Declarations as public fields
global_declarations.forEach(
    (decl) -> {
        classWriter.visitField(
            ACC_PUBLIC | ACC_STATIC,
            decl.identifier,
            SymbolTable.GenBasicDescriptor(decl.type),
            null,
            null
        );

        if (decl.init_expression == null) { return; }

        decl.init_expression.GenerateBytecode(sv);
        sv.visitFieldInsn(
            PUTSTATIC,
            GeneratorStack.peek().programName,
            decl.identifier,
            SymbolTable.GenBasicDescriptor(decl.type)
        );
    }
);

// Add return instruction
sv.visitInsn(RETURN);

// End Static Block Generation
sv.visitMaxs(0, 0);
sv.visitEnd();
```

# The Fearn Programming Language

```
// Generate Default Constructor
MethodVisitor cv = classWriter.visitMethod(
    ACC_PUBLIC,
    "<init>",
    "()V",
    null,
    null
);

cv.visitVarInsn(ALOAD, 0);
cv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Object", "<init>", "()V", false);
cv.visitInsn(RETURN);
cv.visitMaxs(0, 0);
cv.visitEnd();

// Write Functions as Methods
for (Function function : functions)
{
    // Create new FearnMethodVisitor (see FearnMethodVisitor.java)
    // Also, visit new public static method with class writer
    // Gets Function method descriptor from symbol table to do this
    MethodVisitor function_visitor = new FearnMethodVisitor(ASM9,
        classWriter.visitMethod(
            ACC_PUBLIC | ACC_STATIC,
            function.identifier,
            SymbolTable.GenFuncDescriptor(function.parameters, function.return_type),
            null,
            null
        )
    );

    // Create MethodNode (a derived class of a method visitor, that's
    // effectively a list of instruction nodes, that can be iterated
    // through and optimised)
    MethodNode function_node = new MethodNode(ASM9);

    // Set Local Symbol Table
    LocalSymbolTable = GlobalSymbolTable.GetFuncSymbolTable(function.identifier);

    function_node.visitCode();
}
```

## The Fearn Programming Language

```
// Initialise all local variables to null
// Needs to start at function.parameters.size() to prevent nullifying
// the values of parameters
for (int i = function.parameters.size();
    i < LocalSymbolTable.GetAllVarTypeSpecifiers().size(); i++) {
    // Load null value
    function_node.visitInsn(ACONST_NULL);
    // Assign value to variable at index i
    function_node.visitVarInsn(ASTORE, i);
}

// Generate body of bytecode
function.body.GenerateBytecode(function_node);

// Add a return instruction if void, and no return statement already included
if (function.is_void && !function.body.includesReturn)
    function_node.visitInsn(RETURN);

// End Function Generation
function_node.visitMaxs(0, 0);
function_node.visitEnd();

// Eliminate redundant casting
CastOptimiser.EliminateRedundantCasts(function_node);

// Write all instructions to method visitor
function_node.accept(function_visitor);
}

classWriter.visitEnd();

// Write program .class file
try {
    Files.write(finalProgramPath, classWriter.toByteArray());
} catch (IOException e) {
    Reporter.ReportErrorAndExit("Program Gen Error :- " + e.toString(), null);
}

// Report Success
Reporter.ReportSuccess(
    "GENERATED Program      : " + finalProgramPath.toAbsolutePath() + ";",
    false
);
}
```

## The Fearn Programming Language

```
// Generate Program, from AST root and SymbolTable
public void Generate(Program root, SymbolTable symTab)
{
    GlobalSymbolTable = symTab;

    // Initialise LocalSymbolTable to an empty table
    LocalSymbolTable = new SymbolTable();

    // Get path to program class file
    Path finalProgramPath = buildPath.resolve(
        GeneratorStack.peek().programName + ".class").toAbsolutePath();

    // Create new build directory
    File dir = new File(buildPath.toString());
    dir.mkdir();

    // Generate Class files to represent structs
    GenerateStructs(root.structs);

    // Generate Program Class (defines globals and functions)
    GenerateProgram(
        root.functions,
        root.global_declarations,
        finalProgramPath
    );
}

// Set Build Path, from String path to source file
public void SetBuildPath(String path) {
    buildPath = Paths.get(path).toAbsolutePath().getParent().resolve("build");
}

// Set Program Name, from String path to source file
public void SetProgramName(String path)
{
    programName = Paths.get(path).getFileName().toString().replace(".fearn", "");
}
}
```

# The Fearn Programming Language

## codegen.CastOptimiser

```
package codegen;

import static org.objectweb.asm.Opcodes.*;
import org.objectweb.asm.tree.*;

/* CastOptimiser.java
 *
 * This class exists to eliminate redundant bytecode.
 *
 * During development, I decided to enforce the rule that
 * expressions should return the object version of their
 * values, rather than primitives, as they are (in general)
 * easier to work with - as it requires less decision cross-node
 * decision making on which instruction to use (JVM instructions
 * are often typed).
 *
 * The downside of this was the bytecode produced was often
 * inefficient, since a child node would cast its primitive return
 * value to an object, and the parent node would immediately cast
 * it back.
 *
 * My solution to this is below. The method EliminateRedundantCasts
 * iterates through the generated bytecode instructions, in the
 * MethodNode used to generate a function. It then detects when
 * two sequential casts have occurred, and eliminates both. Since
 * these instructions were casting from primitive, to object, and
 * back again, eliminating them has no effect on program functionality,
 * but does make execution faster, more efficient, and makes the binaries
 * smaller.
 */

public class CastOptimiser {

    public static void EliminateRedundantCasts(MethodNode node)
    {
        // Iterate through each instruction in function's MethodNode
        for (int i = 0; i < node.instructions.size(); i++)
        {
            AbstractInsnNode current_insn = node.instructions.get(i);
```

## The Fearn Programming Language

```
// Check for redundant int casting (check for the descriptor
// for casting from primitive int to Integer object)
if (
    current_insn.getOpcode() == INVOKESTATIC
    && ((MethodInsnNode)current_insn).desc.equals("(I)Ljava/lang/Integer;")
) {
    // Check if next instruction is casting straight back
    // (by checking descriptor)
    AbstractInsnNode next_insn = node.instructions.get(i + 1);
    if (
        next_insn.getOpcode() == INVOKEVIRTUAL
        && ((MethodInsnNode)next_insn).desc.equals("(I)I")
    ) {
        // Remove both instructions
        node.instructions.remove(current_insn);
        node.instructions.remove(next_insn);

        // Decrement index to stay at same location next iteration
        i--;
        continue;
    }
}

// Check for redundant bool casting (same procedure as int, refactored
// for primitive Z and Boolean object)
if (
    current_insn.getOpcode() == INVOKESTATIC
    && ((MethodInsnNode)current_insn).desc.equals("(Z)Ljava/lang/Boolean;")
) {
    AbstractInsnNode next_insn = node.instructions.get(i + 1);
    if (
        next_insn.getOpcode() == INVOKEVIRTUAL
        && ((MethodInsnNode)next_insn).desc.equals("(Z)Z")
    ) {
        // Remove both instructions
        node.instructions.remove(current_insn);
        node.instructions.remove(next_insn);

        // Decrement index to stay at same location
        i--;
        continue;
    }
}
```

## The Fearn Programming Language

```
// Check for redundant double casting (same procedure as int, refactored for
// primitive D and Double object)
if (
    current_insn.getOpcode() == INVOKESTATIC
    && ((MethodInsnNode)current_insn).desc.equals("(D)Ljava/lang/Double;")
) {
    AbstractInsnNode next_insn = node.instructions.get(i + 1);
    if (
        next_insn.getOpcode() == INVOKEVIRTUAL
        && ((MethodInsnNode)next_insn).desc.equals "()D")
    ) {
        // Remove both instructions
        node.instructions.remove(current_insn);
        node.instructions.remove(next_insn);

        // Decrement index to stay at same location
        i--;
        continue;
    }
}
}
```



# The Fearn Programming Language

## codegen.CastOptimiser

```
package codegen;

import static org.objectweb.asm.Opcodes.*;

import org.objectweb.asm.MethodVisitor;

/* FearnMethodVisitor.java
 *
 * This is a derived class of ASM's MethodVisitor. It's purpose (beyond that of a normal
 * MethodVisitor) is to catch and correct errors when calling the visitFrame() method.
 *
 * visitFrame() is called whenever a label is visited, and defines the state of the frame
 * at that jump location. To ensure no confusion or loss of data, I often use 'F_FULL' by
 * default, defining the exact state of local variable in the frame, using the local
 * variables from the function's symbol table. The issue is that, if the method is called
 * twice in a row (by different AST nodes), an IllegalStateException is raised. This visitor
 * catches that exception, and replaces the call with an F_SAME call, instructing the JVM
 * that the state of the frame has not changed - this suppresses the error.
 *
 * You can read more about this here:
https://asm.ow2.io/javadoc/org/objectweb/asm/MethodVisitor.html#visitFrame\(int,int,java.lang.Object\[\],int,java.lang.Object\[\]\)
 */

public class FearnMethodVisitor extends MethodVisitor{
    public FearnMethodVisitor(int api, MethodVisitor mv)
    {
        super(api, mv);
    }

    @Override
    public void visitFrame(
        final int type, final int numLocal, final Object[] local, final int numStack,
        final Object[] stack
    ) {
        try {
            super.visitFrame(type, numLocal, local, numStack, stack);
        } catch (IllegalStateException e) {
            super.visitFrame(F_SAME, 0, null, 0, null);
        }
    }
}
```

## The Fearn Programming Language

```
// Java un-escapes escape characters read in by the parser,  
// so this implementation of visitLdc re-escapes them  
@Override  
public void visitLdcInsn(Object obj)  
{  
    if (obj instanceof String)  
    {  
        String input_str = String.valueOf(obj);  
        input_str = input_str  
            .replace("\\n", "\n")  
            .replace("\\t", "\t")  
            .replace("\\b", "\b")  
        ;  
        super.visitLdcInsn(input_str);  
        return;  
    } else super.visitLdcInsn(obj);  
}
```

# The Fearn Programming Language

## codegen.ImportCompiler

```
package codegen;

import java.io.FileInputStream;
import java.util.ArrayList;

import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

import ast.ASTNode;
import ast.Program;
import ast.function.Parameter;
import ast.type.PrimitiveSpecifier;
import ast.type.PrimitiveSpecifier.PrimitiveDataType;
import parser.ASTConstructor;
import parser.gen.*;
import semantics.table.*;
import util.FearnErrorListener;
import util.Reporter;

/* ImportCompiler.java
 *
 * This contains methods to handle the importing of other Fearn Programs,
 * and standard library modules.
 */

public class ImportCompiler {

    // This performs an identical process as Compile in main.java, with the
    // difference of also returning the symbol table.
    public SymbolTable Compile(String path) {

        path = CodeGenerator.buildPath.getParent().resolve(path.replaceAll("(\\'|\\\"")",
        "").toString());

        CharStream input = null;

        try {
            input = CharStreams.fromStream(new FileInputStream(path));
        } catch (Exception e) {
            Reporter.ReportErrorAndExit("IMPORTED FILE " + path + " NOT FOUND", null);
        }
    }
}
```

## The Fearn Programming Language

```
if (path.endsWith("FearnRuntime.fearn") )
{
    Reporter.ReportErrorAndExit("FILENAME FearnRuntime.fearn IS FORBIDDEN.", null);
}

CodeGenerator cg = new CodeGenerator();

cg.SetProgramName(path);

CodeGenerator.GeneratorStack.push(cg);

FearnGrammarLexer lexer = new FearnGrammarLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
FearnGrammarParser parser = new FearnGrammarParser(tokens);

parser.removeErrorListeners();
parser.addErrorListener(new FearnErrorListener());

ParseTree parseTree = parser.program();

ASTConstructor astConstructor = new ASTConstructor();
ASTNode AST = astConstructor.visit(parseTree);

Program root = (Program)AST;
SymbolTable symTable = astConstructor.symbolTableStack.pop();

CodeGenerator.GlobalSymbolTable = symTable;

// Perform Type Analysis
root.validate(symTable);

cg.Generate(root, symTable);

CodeGenerator.GeneratorStack.pop();

return CodeGenerator.GlobalSymbolTable;
}

// This method handles the importing of standard library modules
// Each module has a case in the below statement, and these cases
// build and return a SymbolTable for the functions that module contains
public SymbolTable GetStdLib(String id) {
```

```
    CodeGenerator.GeneratorStack.push(new CodeGenerator());
```

```
    SymbolTable table = new SymbolTable();
```

# The Fearn Programming Language

```
ArrayList<Parameter> params;

// Switch makes standard library easy to expand
switch (id) {
    case "io":
        // Set Program Name
        // This sets the row's owner, ensuring the bytecode for calling these
        // function calls refer the the correct package and class
        CodeGenerator.GeneratorStack.peek().programName = "FearnStdLib/io";

        // Add Print Function

        // Create new parameter list
        params = new ArrayList<>();

        // Add string parameter
        params.add(new Parameter(
            "", new PrimitiveSpecifier(PrimitiveDataType.STR)
        ));

        // Add to symbol table, with identifier print, and null return
        // type and local symbol table (irrelevant as this function has
        // no Fearn implementation)
        table.addRow(
            new FunctionRow(
                "print",
                params,
                null,
                null
            )
        );

        // Add Input Function
        // params remains the same (as both print and input take a single,
        // string argument)
        table.addRow(
            new FunctionRow(
                "input",
                params,
                new PrimitiveSpecifier(PrimitiveDataType.STR),
                null
            )
        );
        break;
```

# The Fearn Programming Language

```
case "maths":

    CodeGenerator.GeneratorStack.peek().programName = "FearnStdLib/maths";

    // Add PI() -> value of PI
    params = new ArrayList<>();
    table.addRow(
        new FunctionRow(
            "PI",
            params,
            new PrimitiveSpecifier(PrimitiveDataType.FLOAT),
            null
        )
    );

    // Add Eulers() -> value of Euler's number
    table.addRow(
        new FunctionRow(
            "Eulers",
            params,
            new PrimitiveSpecifier(PrimitiveDataType.FLOAT),
            null
        )
    );

    // Add sin, cos, and tan functions
    params = new ArrayList<>();
    params.add(new Parameter("", new PrimitiveSpecifier(PrimitiveDataType.FLOAT)));
    table.addRow(
        new FunctionRow(
            "sin",
            params,
            new PrimitiveSpecifier(PrimitiveDataType.FLOAT),
            null
        )
    );

    table.addRow(
        new FunctionRow(
            "cos",
            params,
            new PrimitiveSpecifier(PrimitiveDataType.FLOAT),
            null
        )
    );
```

## The Fearn Programming Language

```
        table.addRow(  
            new FunctionRow(  
                "tan",  
                params,  
                new PrimitiveSpecifier(PrimitiveDataType.FLOAT),  
                null  
            )  
        );  
  
        break;  
  
    case "random":  
        CodeGenerator.GeneratorStack.peek().programName = "FearnStdLib/RandomNumbers";  
  
        // Add random -> Random double between 0 and 1  
        params = new ArrayList<>();  
        table.addRow(  
            new FunctionRow("random", params,  
                new PrimitiveSpecifier(PrimitiveDataType.FLOAT), null)  
        );  
  
        // Add randomFromRange -> Random integer from range  
        params = new ArrayList<>();  
        params.add(new Parameter("", new PrimitiveSpecifier(PrimitiveDataType.INT)));  
        params.add(new Parameter("", new PrimitiveSpecifier(PrimitiveDataType.INT)));  
        table.addRow(  
            new FunctionRow("randomInRange", params,  
                new PrimitiveSpecifier(PrimitiveDataType.INT), null)  
        );  
  
        break;  
  
    default:  
        Reporter.ReportErrorAndExit("Standard library " + id + " does not exist.", null);  
        break;  
}  
  
// Pop Generator, to return to primary program  
CodeGenerator.GeneratorStack.pop();  
  
// Return Symbol Table to primary compilation process  
return table;  
}  
}
```

# The Fearn Programming Language

## util

### util.Reporter

```
package util;

import ast.ASTNode;
import codegen.CodeGenerator;

/*
 * Reporter.java
 *
 * This is a container class for two functions:
 *
 * -> ReportErrorAndExit : Prints a red error message to
 * the terminal, then exits the program. If a node has
 * been passed, it also prints the node of the AST causing
 * the error, reconstructing the corresponding source code
 * using ASTNode.toString() (DFT, adding to a string representing
 * the offending source code)
 *
 * -> ReportSuccess: Used to report when binaries have been built,
 * and where. These messages are printed in purple, or green if
 * the option to exit the program has been set to true. If so,
 * function also exits FearnC.
 */

public class Reporter {
    private static String ANSI_RESET = (char)27 + "[0m";
    private static String ANSI_RED = (char)27 + "[31m";
    private static String ANSI_GREEN = (char)27 + "[32m";
    private static String ANSI_PURPLE = (char)27 + "[35m";
    private static String ANSI_BOLD = (char)27 + "[1m";

    public static void ReportErrorAndExit(String err, ASTNode offendingNode)
    {
        if (offendingNode == null)
            System.out.println(
                String.format(
                    "FearnC (%s): %s%sERROR: %s %s",
                    CodeGenerator.GeneratorStack.peek().programName,
                    ANSI_BOLD, ANSI_RED, err, ANSI_RESET
                )
            );
    }
}
```



# The Fearn Programming Language

```
        else System.out.println(
            String.format(
                "FearnC (%s): %s%sERROR: %s - %s %s",
                CodeGenerator.GeneratorStack.peek().programName,
                ANSI_BOLD, ANSI_RED, offendingNode.toString(), err, ANSI_RESET
            )
        );
        System.exit(1);
    }

    public static void ReportSuccess(String message, boolean exit)
    {
        String colour = ANSI_PURPLE;
        if (exit) { colour = ANSI_GREEN; }

        System.out.println(
            String.format(
                "FearnC: %s%s%s%s",
                ANSI_BOLD,
                colour,
                message,
                ANSI_RESET
            )
        );

        if (exit) System.exit(0);
    }
}
```

# The Fearn Programming Language

util.FearnErrorListener

```
package util;

import org.antlr.v4.runtime.*;

/*
 * FearnErrorLiseter.java
 *
 * Utility class to catch ANTLR-generated parse errors,
 * and passes them through to ReportErrorAndExit(), along
 * with the line and column numbers, to help the developer
 * fix the error.
 *
 */

public class FearnErrorListener extends BaseErrorListener{

    @Override
    public void syntaxError(
        Recognizer<?, ?> recognizer,
        Object offendingSymbol,
        int line, int charPositionInLine,
        String msg,
        RecognitionException e
    )
    {

        String message = msg;

        if (message.startsWith("no viable alternative at input"))
        {
            message = "Unable to parse line. Check for missing characters, like semicolons (;).";
        }

        Reporter.ReportErrorAndExit(
            "Parse Error : line "+line+"; col "+charPositionInLine+": "+message, null
        );
    }
}
```

# The Fearn Programming Language

## FearnRuntime.java

```
import java.util.Arrays;

/*
 * FearnRuntime.java
 *
 * This file is part of the compiled .jar file
 * that the compiler is distributed in.
 *
 * Its purpose is to implement functions that are
 * too unwieldy to implement manually in bytecode,
 * at compile time.
 *
 * This includes:
 * -> The implementation for mathematical exponentiation
 * -> Boolean functions to compare values
 * -> Concatenation
 * -> Casting functions
 * -> The implementation for the two built-in functions,
 *     length() and slice()
 */

public class FearnRuntime {

    /* Exponentiation
     *
     * Integer and floating-point implementations of
     *  $x^y$ 
     */

    public static Integer exp(int op1, int op2)
    {
        Double d = Math.pow((double)op1, (double)op2);
        return d.intValue();
    }

    public static Double exp(double op1, double op2)
    {
        Double d = Math.pow((double)op1, (double)op2);
        return d;
    }

    // Boolean functions
    /*
```

# The Fearn Programming Language

The `equals` function is used to evaluate the equality of two objects. it includes a case to compare arrays (a facility many languages don't offer), by recursively comparing the equality of every item in the array, as well as the arrays length, returning the AND of these comparisons.

```
*/
```

```
public static Boolean equals(Object op1, Object op2)
{
    if (op1 == null && op2 == null)
    {
        return true;
    }

    if (op1.getClass().isArray())
    {
        Object[] arr1 = (Object[])op1;
        Object[] arr2 = (Object[])op2;

        Boolean areEqual = arr1.length == arr2.length;

        for (int i = 0; i < arr1.length; i++)
        {
            areEqual = areEqual && equals(arr1[i], arr2[i]);
        }

        return areEqual;
    }

    return op1.equals(op2);
}

public static Boolean less(int op1, int op2) { return op1 < op2; }
public static Boolean less(double op1, double op2) { return op1 < op2; }
public static Boolean less_eq(int op1, int op2) { return op1 <= op2; }
public static Boolean less_eq(double op1, double op2) { return op1 <= op2; }
public static Boolean greater(int op1, int op2) { return op1 > op2; }
public static Boolean greater(double op1, double op2) { return op1 > op2; }
public static Boolean greater_eq(int op1, int op2) { return op1 >= op2; }
public static Boolean greater_eq(double op1, double op2) { return op1 >= op2; }
public static Boolean not(Boolean op) { return !op; }
public static Boolean and(Boolean a, Boolean b) { return a && b; }
public static Boolean or(Boolean a, Boolean b) { return a || b; }
```

# The Fearn Programming Language

```
public static String concat(String a, String b) { return a + b; }

// Casts an Java object to boolean, by comparing it to zero
public static Boolean Obj2B(Object o)
{
    return !(o.equals(0) || o.equals(0.0));
}

// Casts objects to strings
public static String Obj2Str(Object o)
{
    if (o.getClass().isArray())
    {
        return Arrays.deepToString((Object[])o);
    } else {
        return o.toString();
    }
}

/*
Sequence functions
-> These implement the two built-in
    functions, length() and slice()
*/
public static Integer length(Object o)
{
    if (o instanceof String)
    {
        return String.valueOf(o).length();
    }

    return ((Object[])o).length;
}
public static String slice(String seq, Integer start, Integer end)
{
    return seq.substring(start, end);
}
public static <T> T[] slice(T[] seq, Integer start, Integer end)
{
    return Arrays.copyOfRange(seq, start, end);
}
}
```

# The Fearn Programming Language

## FearnStdLib

### FearnStdLib.io

```
package FearnStdLib;

import java.util.Scanner;

public class io {
    static Scanner scan = new Scanner(System.in);
    public static void print(String a)
    {
        System.out.println(a);
    }
    public static String input(String prompt)
    {
        String in;
        System.out.print(prompt);
        in = scan.nextLine();
        return in;
    }
}
```

### FearnStdLib.maths

```
package FearnStdLib;

public class maths {
    public static Double PI() { return Math.PI; }
    public static Double Eulers() { return Math.exp(1.0); }
    public static Double sin(Double theta) { return Math.sin(theta); }
    public static Double cos(Double theta) { return Math.cos(theta); }
    public static Double tan(Double theta) { return Math.tan(theta); }
}
```

### FearnStdLib.RandomNumbers

```
package FearnStdLib;

import java.util.Random;

public class RandomNumbers {
    public static Double random() { return Math.random(); }
    public static Integer randomInRange(Integer LB, Integer UB)
    { return LB + new Random().nextInt(UB-LB); }
}
```

# The Fearn Programming Language

## CMD Files

These files are not part of the compiler itself. Instead, they are Windows Command Files which, when added to the user's PATH, implement the *FearnC* and *FearnRun* Commands. The files are written to be included in a single 'release' directory, alongside the FearnC.jar file, which the compiler itself builds to.

### FearnC.cmd

```
@echo off
set jarPath=%~dp0\FearnC.jar
java -jar --enable-preview "%jarPath%" %1
```

### FearnRun.cmd

```
@echo off
if "%~1"==" " (
    echo Usage: FearnRun "<program-name>" "<args>" ...
    exit /b 1
)
java --enable-preview %*
```