Creating an Interpreter

Full Name: ************
Candidate Number: ****
Centre Number: ****
Date: 01/10/2020

Contents

Analysis

Introduction

Overview

I am going to write a **basic interpreted programming language** that is aimed at beginners who are learning how to program.

The Problem

While there is a plethora of available programming languages for beginners to learn, such as Python or Java, they often end up either hiding programming paradigms or showing too many at once – both traits are not good for beginners. **There are very few programming languages that are *built for beginners only*** – languages such as Python hide a lot of work going on and encourage bad habits, whereas Java forces Object Orientated Programming at all times.

My Solution

I intend to **write an interpreted language that is designed solely for learning** – it will not be aimed at industry or even hobbyist programmers. This language will not be overly complex due to needing to stay in A-Level scope and being beginner-focused. I am writing this for people who have never written program code before (at any age) and I hope that this language will provide a simple entry point to programming along with some good habits.

Existing Languages

There are thousands of programming languages available to anyone who wishes to try them – they all have a wide range of features both simple and complex, and therefore they all have their own idiosyncrasies that can be confusing to a beginner and ultimately create bad habits. For those wishing to get into Computer Science, basic programming theory is one of the most important foundations of knowledge that they will need for any form of problem solving. Below are some examples of languages that beginners are often taught first.

[1]*Python 3*

Overview

Python is one of the most popular programming languages in the world, being recorded as the second most loved language in the Stack Overflow 2019 Developer Survey[2]. It has also been regarded as the **easiest programming language to get into**, adopting a more **English-based syntax style** and coming with IDLE (Integrated Development & Learning Environment). As it is an **interpreted language**, Python can have an intuitive shell that can be used for rapid prototyping and testing. Python's popularity could be attributed to its ease of use, or because it has hundreds of thousands of libraries that make use of its parent language, C++.

[1] https://python.org (picture) accessed 26/10/2020
[2] https://insights.stackoverflow.com/survey/2019 accessed 28/10/2020

Being written in C++, Python is automatically given advantages in terms of creating and importing features from other more complex libraries, allowing it to 'simplify' C++ libraries and make them more accessible to beginners and professionals alike. While this is a great advantage in real-world programming and industry, it is unnecessary for learning and assessment of programming concepts. **Python has inherited (and created) many complex additions that are irrelevant to beginners** and can often lead to gaps in their knowledge when moving to more complex languages.

Syntax

```
>>> print("Hello World")
Hello World
```

This picture shows an example of a simple print statement being input. Unlike some other languages, Python's print statements and other key built-in subroutines are simplistic words such as **`print`** and **`input`**. In simplifying key functions that are often the first ones used by beginners, Python makes the process of first learning programming a lot more understandable by not overcomplicating these features.

As well as this, **Python does not require you to interact with Object Orientated Programming (OOP)**. This makes it stand out to some other languages that are often used to teach beginners, such as Java or C#. If beginners are learning these languages, they must either learn object orientation first and then go onto programming paradigms or they will just have to 'ignore' the class definitions at the start of Java or C# programs.

Python is a lot better for teaching beginners, as it does not force either – **the beginner can just go straight into a text file and write a Python statement,** without having to copy down OOP statements that they do not yet understand.

```
x = 0
x = "String"
x = []
```

The syntax on the left is an example of how Python uses **dynamic typing**, meaning that you can change a variable's type to whatever you'd like without explicitly saying so – the variable in this syntax takes on the types Integer, String and Array all in three lines and will stay an array at the end if unchanged. While this may seem like a useful feature at first, it is often looked on to be a **bad habit to do this**, as it leads to uncertainty on what type a variable is anywhere in the program – in one line, **x** could be an Integer and in fifty lines down it could be an array, leaving uncertainty as to what functions you can apply to it without causing errors. When someone is learning to program, **good habits should be forced** from the beginning to ultimately make a better programmer – they should learn that they cannot do this and often it is better to have a language that will **not** let them do it to prevent it.

Furthermore, the syntax above shows that Python is **weakly** typed, it does not require the programmer to declare the type of the variable upon first appearance. This may also be another convenient and quick way of writing things for more experienced programmers, but ultimately it **does not help a beginner learn how variables are of specific types** and what those types are. A language that made them think of what type of variable they require when they are writing the program without letting them just rely on the interpreter to work it out is therefore better for their learning – this would be a **strongly** typed language.

```
x = 20

if x != 0 and x % 2 == 1:
    print("X is odd")
```

This program contains a simple **`if`** statement with two conditions, joined by the **and** statement. It is specifically important that Python uses keywords a lot more than key characters, as in most other languages this 'and' would be represented by **&&.**

As well as this, **Python uses indentation to mark out code blocks** instead of the more common '{' and '}'. While at first this may make the programs seem less intimidating and more English looking, it can also end up with students being confused as to why the programming is not running as they have used a combination of spaces and tabs – Python can interpret a program with **only** tabs or spaces at the beginning of each line in a block, but not both.

The use of more English-based syntax is a theme throughout Python, allowing simple loops to be more readable and simplistic. This is an example of a program that a beginner may be asked to create with a while loop – output all numbers from 1 to 10.

```
count = 1
while count <= 10:
    print(count)
    count = count + 1
```

Summary of Python

Python is:

- An interpreter
- Very popular for both beginners and professionals alike
- Written with simplistic English keywords meaning it is easier to read
- Indentation-based, meaning code blocks are denoted by how many tabs or spaces they have at the beginning of each line
- Written with lightweight object-orientation

Advantages (for beginners)

- More word-based syntax makes it more readable and less intimidating to a beginner
- Being an interpreter means the program will still run up to the point of an error – this is more intuitive to a beginner as the program is seems to be running 'line-by-line'
- Object orientation is not forced – a beginner does not need to be told to ignore object definitions at the start of the program as they are not present. A program can be written entirely without including OOP

Disadvantages (for beginners)

- Indentation can be confusing as errors can be caused by using a mix of tabs and spaces, a problem which is invisible in most text editors
- There are a lot of complex features that can overshadow learning basic programming concepts

Java 14[3]

Java is an object-orientated programming language that is designed for general purpose. **Compiled** Java code can run on all platforms that support it without the requirement to re-compile – it does this by running on a Java virtual machine (JVM). As of 2019, there were 45 billion active JVMs[4] around the world.

[3] https://blogs.oracle.com/documakertech/do-i-need-to-subscribe-to-java-se-to-run-oracle-documaker (picture) accessed 28/10/2020
[4] https://www.oracle.com/in/a/ocom/docs/java-strength-in-numbers.pdf accessed 28/10/2020

Due to the intended 'general-purpose' design of Java, it can be used in multiple different industries around the world and yet still act as a beginner language for individuals. Java has given rise to some of the most used technologies and games such as today, with the popular game Minecraft selling over 200 million copies[5] and therefore being the best-selling video game of all time.

Java is also a great way to introduce beginners to object-orientated programming concepts, though unless the intention is to teach them that to begin with, they will have to ignore certain parts of every program until they reach it. **Object orientation is forced in Java** – it is unlike Python in that programs that are meant to be run must have a class declaration.

Syntax

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

This a hello world program in Java. It is important to note that before the print statement can even be written, two more lines must be added to first declare the class **`Main`** and then the function **`main`**. This is an example of something that might be confusing or intimidating to a beginner first learning Java – either they have to be taught the entirety of object-orientated programming and how it works, or they have to be told to simply ignore these lines for now.

As well as this, the print statement **`System.out.println("Hello World");`** can be confusing to a beginner – and this is one of the first things that they will be taught. Unlike in Python, where the print function can be used by simply writing **`print()`**, Java requires a few more additions that also have the same problem as the first two lines – either explain what **`System.out`** is referring to (more OOP) or **tell the beginner to ignore it**. Both options can **cause confusion among beginners**, and the more things that need to be included in a simple program often means more mistakes.

```
public class Main {
    public static void main(String[] args) {
        int x = 0;
        String y = "Zero";
        int[] z = {};
    }
}
```

This is an example of how variables are declared in Java. Java is **statically** typed, meaning that the type of a variable is always known. As well as this, it is **strongly** typed – this means that the type of a variable must be declared.

This is a **much better approach for beginners** than simply declaring a variable that can be any type at any time, as it **forces them to think** about what they are going to use the variable for and allows them to have guaranteed knowledge of what type the variable is at any point in the program so that they do not make as many mistakes when doing operations with it.

As well as this, Java **does not allow variables in the same scope (code block) to be 'reused'** (unlike Python). This means that beginners will not be comforted with bad habits of reusing variables like 'count' for several different purposes, ultimately leading to less confusion and better programming habits.

[5] https://www.theverge.com/2020/5/18/21262045/minecraft-sales-monthly-players-statistics-youtube 28/10/2020

Java **`if`** statements are also very different to Python. The condition of an **`if`** statement, or the contents of any flow control feature, must be enclosed in brackets. As well as this, the statements inside code blocks must be enclosed with curly brackets. This is **instead** of using tabs and spaces (indentation) to denote code blocks' starting and ending points, and thus avoids any errors attributed to improper indentation.

```java
public class Main {
    public static void main(String[] args) {
        int x = 20;
        if (x != 0 && x % 2 == 1) {
            System.out.println("X is odd.");
        }
    }
}
```

It is also important to note that instead of using the keyword **and** inside the condition, the characters **&&** are used instead. This can be intimidating at first to a beginner, especially since all the conditions are characters and not keywords – these will have to be learnt. A statement like this will not easily be readable to the untrained eye, whereas a similar statement in Python will be at least slightly more understandable due to the use of basic English keywords and removal of brackets.

Summary of Java

Java is:

- Compiled and then run on a Java virtual machine
- A general-purpose language that can be run on many different architectures
- One of the most popular languages in industry
- Heavily object-orientated, strongly typed

Advantages (for beginners)

- Java can be used in almost any area after learning: game development, desktop applications, back-end services, and more
- Java is a great steppingstone for object-orientated languages and follows more common syntax rules used in languages such as C++, C#
- Strong typing means that a beginner needs to think through what they are going to do with a variable and prevents future confusion or errors

Disadvantages (for beginners)

- Forced object-orientation can initially confuse beginners and forces them to either learn or ignore it
- Syntax is a lot less friendly and more symbol-based; ultimately means statements are harder to read and understand by an inexperienced developer
- Programs need to be compiled and then run by a JVM, adding another layer of things to confuse or go wrong

Interview

Interviewee

The person I have interviewed is Eddie, who has just recently learnt how to program with Java after beginning on Python 3.

Questions

Q1: *What programming language did you start programming with?*
A1: I first began learning how to program with Python and now I've moved onto Java.

Q1.1: *Was there anything initially intimidating or confusing?*
A1.1: When we did variable assignment, it was confusing at first because it looks kind of mathematical but doesn't work the same way – the equals sign is assigning a value and not stating that the variable is already equal. Also, installing Python and Java was kind of confusing as you couldn't run any program without the right versions installed.
Summary:

- Eddie started learning with Python and then moved onto Java
  - It is important to note he moved onto Java, a strongly typed language with forced OOP
- Eddie found **variable assignment confusing** as it was not as intuitive
  - I cannot really do much about this to help as it will be a common part of any programming language
- Eddie thought that **installing Python or Java was confusing for a beginner**
  - I will need to make sure that I pick **a simplistic platform** for this interpreter so that it doesn't require much (if any) extra installation

Q2: *What kind of programs did you first learn to write?*
A2: We started with a simple 'Hello World' print statement program and then worked our way to variables and outputting them. After that we moved onto if-else statements.
Q2.1: *How did you find outputs?*
A2.1: It was kind of confusing that the word you had to write to output was called 'print' as I originally thought that it meant I was actually printing something. Other than that though, Python made it quite easy – I only had to write a one-word long function and whatever I wanted to output and then it was done. Looking at Java, it seems a lot more complex to write "System.out.println" than just "print".
Q2.2: *How did you find taking input?*
A2.2: Just like with the output, Python made it really simple to take input and assign a variable to input. All I had to do was declare a variable and set it equal to "input()", which I could do in one line and run instantly. It was a little confusing that the program would just stop and wait for something to be entered without you knowing an input was needed. With Java though, it is much more confusing because I have to import a scanner library and much more – Python made it easy with one line.
Q2.3: *What would you change about input/output for a beginner language?*
A2.3: Well I think I would make it so that a 'print' statement would actually be something like "output()" or "say()". This would make it a bit less confusing for someone who is beginning because they do not understand why outputting is known as 'printing' and therefore won't immediately know what the function is doing or how to recognise it. The input system in Python seems fine, where the function just returns a value entered into the command line – though it'd be good if it automatically wrote "Input: " or something similar to that when requiring input, otherwise the beginner might not know they are supposed to type something.
Summary:

- Eddie's first program was a **simple print statement**
  - He then moved onto **variables** and **if statements**
  - I think that I will only implement up to if statements because that is significant ground to cover for learning to make basic programs
- Eddie found **print statement's identifiers confusing** at first

  - The word 'print' isn't descriptive to a beginner, and Java makes it even more confusing with "System.out.println"
- Overall, Eddie's requirements for a beginner-oriented input/output system
  - **Printing function should be named better**, such as "output()"
  - Input functions should automatically show that they require input for the program to move forward.

Q3: *How did you find variable assignment? What was confusing?*
A3: In Python, it is really easy to declare variables and I didn't have too many problems at first – this was the problem though in the end. Because Python didn't really make me choose what type I wanted a variable to be, I never really had to think about it and didn't pay much attention to variable types overall. This meant later on when I moved to Java that I was a bit surprised because I suddenly had to think of a type for every single variable, and I couldn't lazily reassign them like in Python.
Q3.1: *What would you change for a variable assignment system in a beginner's language?*
A3.1: I'd probably force them to do it like Java first – they have to say exactly what type a variable is when declaring it and they can't reassign it. This will force good habits and will make them think a bit more about what type they are going to need.
Summary:

- Eddie thinks that this interpreter should force **strongly typed variable** declarations to promote awareness and thought over what type of variables the beginner is using
- Eddie also thinks the interpreter **should not allow you to reassign a variable** like Python does, as it promotes bad habits

Q4: *How did you find learning if statements in Python?*
A4: It was quite confusing for a while because of the indentation. I was not sure why I kept get errors in my program until I realised that I had been using a mixture of tabs and spaces and forgot to indent some parts of the program. This was confusing to me, because the tabs and spaces were invisible, and I'd assumed that meant they were not important to Python at all. As for the actual if statements, we began by learning simple conditions like logic operations and arithmetic ones, such as if A > B. We used a combination of variables and hard-written values to make comparisons.
Q4.1: *What kind of things would you like for a beginner language if statement?*
A4.1: I would probably get rid of the indentation as a whole – it is a really confusing thing and it is quite hard to spot or fix if you don't already know how. After moving onto Java, it makes a lot more sense to use brackets to show where a code block begins and ends and leads to a lot less syntax errors. Also, I think only simple comparisons are needed with 2 values on either side. You don't need complex and long comparisons when you are first starting, and super long if statement conditions are a bad habit for disorganised programs.
Summary:

- Eddie found **indentation** quite **confusing** in Python
  - He could not see where errors were coming from
  - He did not understand why they were being caused
  - He thought that **brackets** should be used **instead** of tabs and spaces, as they were less confusing and didn't have 'hidden' errors
- Eddie first learnt simple comparisons **between two arguments**
  - He thought that the language should only need to compare between two

- I will likely only include to this level of comparison (meaning no AND/OR operations) as I do not feel it is necessary to go any further – it would also likely range outside of the scope

Summary of Interview & Objectives

- The first program Eddie learnt was how to output
    - He found it confusing that it is called 'print' and not something like output
      → I will call the print function 'output' with a usage to be something roughly like "output(string);"
      → I will keep in mind that any built-in functions I include need to have understandable identifiers that are friendly to beginners
      SYNTAX OBJECTIVE 1: **Working output** functionality with descriptive identifier
- Eddie thought the input system should be simplistic and indicate when expecting input
    - He did not like how Java had to have a scanner library imported and object initiated just to take input, and preferred the ability to do it in one line like Python
      → I will not have an 'input' or 'scanner' library, it will be automatically built into the language like in Python
      → The input system will have an understandable identifier and usage
      SYNTAX OBJECTIVE 2: **Working input** functionality with descriptive identifier
- Eddie did not like how Python took care of most variable functionality
    - He found it set him up for bad habits when Python didn't require strong typing and allowed you to reassign variable types
      → I will implement and force strongly typed variables (e.g "**int** x = 0;")
      → I will not implement the functionality to reassign variable types at all, it will cause an error when attempted
      SYNTAX OBJECTIVE 3: Strongly typed variable naming and no reassignment
- Indentation was a source of initial confusion and errors for Eddie
    - He did not understand where errors were coming from due to using a mix of tabs and spaces
    - It was hard to spot these errors as both are whitespace and thus not visible in a lot of text editors, he suggested using bracket code blocks instead

      → Code block notation will not be based off indentation (though indentation can still be used for nice formatting)
      → I will make use of brackets to show where a code block begins and ends, making it a lot clearer for someone reading
      SYNTAX OBJECTIVE 4: Code blocks should be denoted by '{' and '}' with functional nesting of statements
- Eddie learnt simple comparisons between two variables with if statements
    - Eddie did not think that a beginner would need to use any more complex comparisons at first other than simple logic and arithmetic ones
      → I will include basic if statement functionality that can evaluate a two-operand comparison
      → As an **extended** objective I will include 'else' functionality as this is would require a more complex system
      → Nesting if statements should be possible

SYNTAX OBJECTIVE 5: Functional two-operand comparison if statements with the following code block being run if true
EXTENDED SYNTAX OBJECTIVE 5.1: Else statements that are linked to the above if statement. This does not include ‘else if’.

Rough Syntax Definition

This interview and my previous research of existing languages allows me to get a rough idea for syntax examples of my interpreter, though these may be subject to change.

Hello World Output

```
output("Hello World");
```

Variable Manipulation

```
int x = 2 + 3*(2-1);
x = 2;
x = x + 1;
```

Input (subject to change)

```
string x = "";
input(x);
```

If Statement

```
if (condition) {
    ...
}
```

Prototyping & Modules

Languages

There are several potential languages that I can use to write the project in:

Python 3

- Python is interpreted, so it will likely be slower for large-scale programs
- Python’s object orientation is not conventionally implemented, it has thicker syntax and multi-class programs are not easy to organise
- Cross-platform as it’s an interpreter
- Users would be required to install Python, something which a beginner will likely not know how to do properly

Java

- Java is compiled but still requires a Java virtual machine to run
- Java has fully implemented object orientation with mostly standard conventions
- Users will have to install a JVM, therefore making it a more confusing process to be able to run the interpreter as it will be stored in a .jar file
- Cross-platform due to JVM

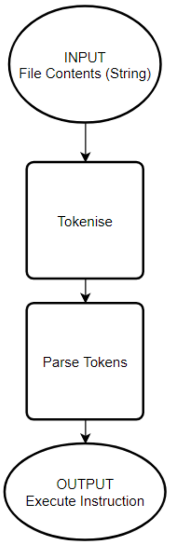
C#

- C# is compiled to an executable file (.exe) or platform-specific executable
- C# has fully implemented object orientation and follows convention
- Users will not have to install any additional programs to run a C# .exe file
- Cross-platform due to .NET Core functionalities

Weighing up the options with creating the language in mind, I feel that C# is the best. This is due to it not requiring the user, who will be a beginner, to install a Java virtual machine or the latest version of Python just to run an interpreter to learn programming. If they are only just learning to program with my interpreter, then these installations will potentially be confusing tasks for them and can go wrong – preventing them even running the interpreter.

Prototyping

Interpreters can be split up into different smaller modules that each feed into each other to eventually produce a result:



Input

A file containing plaintext represents the program – this file can be written to by any IDE or text editor and does not need to be 'compiled'.

Tokenisation

Tokenisation groups parts of a string into tokens, removing any unnecessary whitespace. A 'lexer' uses tokenisation but also gives context to each token, stating useful information about them to give to the parser. Tokenisation can be done on a large scale, such as lexing, or on a small-scale, such as evaluating expressions.

Parsing Tokens

To parse tokens is to assemble them into a more ordered structure for them to be executed. Tokens will be assembled into an abstract syntax tree (AST) which represents the overall structure of the program.

Output

Output is the product of the parser's results being executed. It can take the form of an actual console output or any system action and is done in the order denoted by the syntax trees.

While I cannot easily prototype a large-scale tokenisation algorithm (lexer) without writing one completely, I can prototype tokenising methods and parsing methods on a smaller scale: **evaluating mathematical expressions**.

One of the features that will already be required of this language is to understand basic mathematical expressions and work out their result considering order of operations. There are two different ways I could approach this:

1. **I can use a language's built-in expression evaluator** or an imported library to do so
2. **I can write my own expression evaluator**. This will make use of trees, recursion (in traversal) and Reverse Polish Notation.

While simply importing a library that will evaluate the expression from a function would work, it will **not allow me to prototype the actual process of tokenising, tree creation and traversal** – all of which I will have to do myself on a larger-scale to write an Interpreter.

Prototype Program

These are the requirements and how I have created/tested them:

1. Program should be able to handle single and multi-digit **integers** in an expression
2. Program should understand what the basic mathematical operators do
3. The program should be able to correctly order the execution of operations, and take into account any sections nested in brackets (and).
4. Program should output the postfix version of the expression, along with the correct result

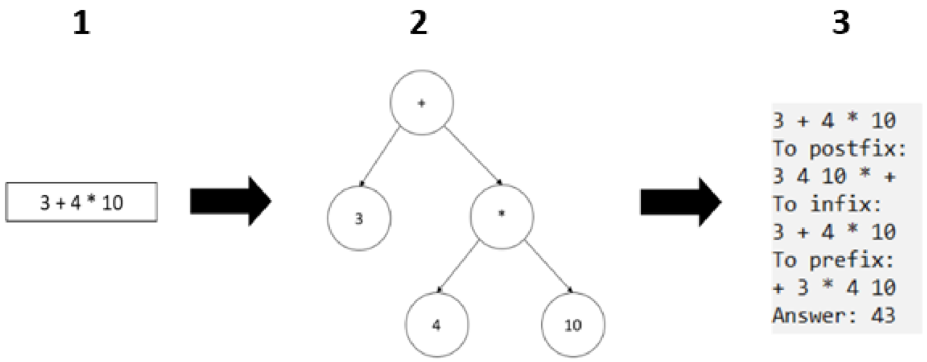
I have created a small-scale tokenisation function which recognises the following definitions:

<digit> : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<integer> : <digit> | <digit><integer>

<operator> : + | - | / | * | ^ | (|)

For now, it works based on a simplistic loop and digit cache, and only outputs tokens as two types – 'Num' (Integer) and 'BinOp' (Binary Operator).

This gives the tokens over to the small-scale parser. I could have written my own mathematical expression parser, but I have chosen to implement Dijkstra's Shunting-Yard algorithm[6] to create the tree as it is much more efficient and reliable. This algorithm makes use of **stacks** and builds an **abstract syntax tree** as shown in the diagram. After building this tree, all that needs to be done is post-order traversal to convert it to postfix (Reverse Polish Notation) and then use an algorithm to calculate the result. Different traversal orders (post, pre, in) produce different formats of output from the AST.



Program – *SB = Shown Below*

As we are using trees, we need to define a basic abstract class for a tree node (**SB**). This abstract class simply states we have a left node, a right node and a string value – all of which can be (and start off as) null. You will then see the two definitions for 'Num' and 'BinOp', **which both inherit from** TreeNode– they are making use of the abstract class. Note that the order of precedence for the operators is defined as a dictionary for ease of use when referencing and getting precedence, along with also just using it as a list of all the operators. Moving to the small-scale tokeniser, we use the

[6] https://brilliant.org/wiki/shunting-yard-algorithm/, 03/11/2020

```
abstract class TreeNode
{
    public TreeNode left = null;
    public TreeNode right = null;
    public string value = null;

    public TreeNode() // parameterless base for ease of use if you do not want to create a
    node with a left and right value
    {

    }

    public TreeNode(TreeNode inputLeft, TreeNode inputRight) // Simply exists for ease of use
    when creating a new TreeNode
    {
        this.left = inputLeft; this.right = inputRight;
    }
}
```

```
class Num : TreeNode
{
    public int intValue { get; set; } // this.value still exists, but is in string form - it
    is useful to have a string value and int value instead of problematic ToInt conversions
    later

    public Num(int inputValue)
    {
        this.intValue = inputValue;
        this.value = inputValue.ToString(); // Both values equal to the same integer, just
        different types.
    }
}
```

```
class BinOp : TreeNode
{
    public static Dictionary<string, int> precedences = new Dictionary<string, int>()
    {
        {"^", 4 },
        {"*", 3 },
        {"/", 3 },
        {"+", 2 },
        {"-", 2 },
        {")", 2 },
        {"(", 1 }
    }; // Static dictionary of precedence levels represented by ints for ease of comparison
    later on - used in Algorithms/Postfix.cs

    public BinOp(string operationValue)
    {
        this.value = operationValue;
    }

    public BinOp() // parameterless option
    {

    }
}
```

```
    public static List<ExpressionToken> TokeniseExpression(string expr) // Change a string to a list of tokens
    {
        List<ExpressionToken> exprTokens = new List<ExpressionToken>();
        string numCache = ""; // Cache of digits found, for numbers longer than one digit.

        foreach (char character in expr.Replace(" ", "")) // Guarantees no white space found or distributed in
        tokens.
        {
            if (BinOp.precedences.ContainsKey(character.ToString())) // If the character is an operator.
            {
                if (numCache.Length > 0) // If we still have digits in the cache then submit them to the token
                list as a single token (number) and clear it.
                {
                    exprTokens.Add(new ExpressionToken(numCache, true)); // Add full integer (currently in string
                    data type) to tokens, true meaning it is a number.
                    numCache = "";
                }
                exprTokens.Add(new ExpressionToken(character.ToString(), false)); // Add operator to tokens, false
                meaning it is not a number.
            }
            else if (Char.IsDigit(character)) // If it's a digit, add it to the numCache
            {
                numCache += character;
            }
        }
        if (numCache.Length > 0) exprTokens.Add(new ExpressionToken(numCache, true)); // If the expression ends
        and we still have cached digits, add the digits collected to the tokens list.

        return exprTokens;
    }
}
```

objects that just store a string value and a Boolean IsNumber, definition not shown in screenshots).

Now that we have defined our two tree node types and created a tokeniser, we need to implement an algorithm that takes in what our tokeniser has given us and creates an **AST** of them (**Stage 2 of diagram**). This will be our parser, specifically for mathematical expressions, and we can use the Dijkstra algorithm mentioned earlier (**SB**).

```
public static TreeNode InfixToPostfix(string infix)
{
    Stack<BinOp> operatorStack = new Stack<BinOp>();
    Stack<TreeNode> numStack = new Stack<TreeNode>();

    foreach (ExpressionToken token in TokeniseExpression(infix))
    {
        if (token.value == "(")
        {
            operatorStack.Push(new BinOp(token.value));
        }

        else if (token.isNumber) // If it is a number, add to numStack
        {
            numStack.Push(new Num(Int32.Parse(token.value)));
            // Simply create a new Num (inheritant from TreeNode) node with the number in the character.
        }

        else if (BinOp.precedences.ContainsKey(token.value) && token.value != ")")
        {
            while (operatorStack.Count > 0 && BinOp.precedences[operatorStack.Peek().value]
                >= BinOp.precedences[token.value])
            { // While the precedence of the top of the operatorStack is bigger than or equal to the precedence of the char
                BinOp binOperator = operatorStack.Pop();
                // Reversed as the second op was pushed at the end
                binOperator.right = numStack.Pop();
                binOperator.left = numStack.Pop();

                numStack.Push(binOperator);
            }
            // Now push operator at the end
            operatorStack.Push(new BinOp(token.value));
        }

        else if (token.value == ")")
        {
            while (operatorStack.Count > 0 && operatorStack.Peek().value != "(")
            {
                BinOp binOperator = operatorStack.Pop();
                // Reversed as the second op was pushed at the end
                binOperator.right = numStack.Pop();
                binOperator.left = numStack.Pop();

                numStack.Push(binOperator);
            }
            operatorStack.Pop();
        }

        else
        {
            throw new SyntaxErrorException();
        }
    }

    while (operatorStack.Count > 0)
    {
        BinOp binOperator = operatorStack.Pop();
        // Reversed as the second op was pushed at the end
        binOperator.right = numStack.Pop();
        binOperator.left = numStack.Pop();

        numStack.Push(binOperator);
    }

    return numStack.Pop();
}
```

Now that we can create an abstract syntax tree from our original infix expression, we need to be able to compute what the result is. This means we need to traverse the abstract syntax tree, form an output from post-order traversal, and then use a Reverse Polish Notation algorithm to calculate the result. We begin by defining some useful recursive tree traversal methods (**M**), and the RPN

```
        /*
         * All three of these are recursive - they create their own lists which are returned and added onto the list of the parent.
         */
        public static List<TreeNode> postOrder(TreeNode node)
        {
            List<TreeNode> nodes = new List<TreeNode>();
            if (node.left != null) nodes.AddRange(postOrder(node.left)); // Add recursive call onto the end (via AddRange) of the List
            if (node.right != null) nodes.AddRange(postOrder(node.right));
            nodes.Add(node); // As is post order, add the parent node to the end.
            return nodes;
        }

        public static List<TreeNode> inOrder(TreeNode node)
        {
            List<TreeNode> nodes = new List<TreeNode>();
            if (node.left != null) nodes.AddRange(inOrder(node.left));
            nodes.Add(node); // In order so add parent node in between.
            if (node.right != null) nodes.AddRange(inOrder(node.right));
            return nodes;
        }

        public static List<TreeNode> preOrder(TreeNode node)
        {
            List<TreeNode> nodes = new List<TreeNode>();
            nodes.Add(node); // Pre order so add parent node first.
            if (node.left != null) nodes.AddRange(preOrder(node.left));
            if (node.right != null) nodes.AddRange(preOrder(node.right));
            return nodes;
        }
    }
```

```
        public static int Evaluate(List<TreeNode> nodes)
        {
            Stack<TreeNode> nodeStack = new Stack<TreeNode>(); // Create stack to use for RPN, type TreeNode as it could be a BinOp or Num

            foreach (TreeNode treeNode in nodes)
            {
                if (treeNode is Num) nodeStack.Push(treeNode); // If it is a Num (operand) then just push onto stack
                else if (treeNode is BinOp) // If it is a BinOp (operator) then pop last two operands and calculate:
                {
                    // Pop last two
                    Num arg2 = (Num) nodeStack.Pop();
                    Num arg1 = (Num) nodeStack.Pop(); // Note they are reversed, the first one to be popped is the second argument in the expression.

                    nodeStack.Push(new Num(Calculate(arg1.intValue, arg2.intValue, treeNode.value))
                        ); // push result as a Num type
                }
            }
            Num result = (Num)nodeStack.Pop(); // Stack should be left with just one Num (operand) as the final result
            return result.intValue;
        }

        public static int Calculate(int arg1, int arg2, string operation)
        {
            int result = 0; // Default is 0
            switch (operation)
            {
                case "+":
                    result = arg1 + arg2;
                    break;
                case "-":
                    result = arg1 - arg2;
                    break;
                case "*":
                    result = arg1 * arg2;
                    break;
                case "/":
                    result = arg1 / arg2;
                    break;
                case "^":
                    result = (int)Math.Pow(arg1, arg2);
                    break;
                default:
                    break; // do nothing as result = 0 already
            }
            return result;
        }
    }
```

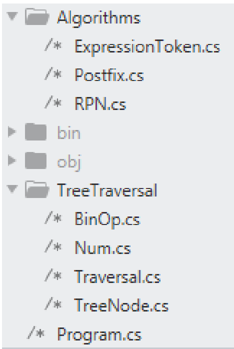
algorithm ('Evaluate') that we are going to use to compute the result – it takes an input of a list of TreeNodes in post-order (**M**).

Finally, we can combine all the functions we have created to achieve **Stage 3** of the diagram above and therefore form a program which will take input of a mathematical expression and output the postfix, infix, prefix form and result. (**SB**).

```
            TreeNode bin1 = Postfix.InfixToPostfix(Console.ReadLine()); // e.g "5 * 2 + 1" -> "5 2 * 1 +"
            // Using TreeNode type, not BinOp (Binary Operator) as we cannot guarantee the root node of the abstract syntax tree will be an operator.

            Console.WriteLine("To postfix:");
            foreach (TreeNode node in Traversal.postOrder(bin1))
            {
                Console.Write(node.value + " ");
            }
            Console.WriteLine("\nTo infix:");
            foreach (TreeNode node in Traversal.inOrder(bin1))
            {
                Console.Write(node.value + " ");
            }
            Console.WriteLine("\nTo prefix:");
            foreach (TreeNode node in Traversal.preOrder(bin1))
            {
                Console.Write(node.value + " ");
            }
            Console.WriteLine();

            // Now using reverse polish notation, calculate what the result is. This takes in a postfix-ordered list of TreeNodes.
            Console.WriteLine("Answer: " + RPN.Evaluate(Traversal.postOrder(bin1)));
        }
    }
```

```
3 + 4 * 10
To postfix:
3 4 10 * +
To infix:
3 + 4 * 10
To prefix:
+ 3 * 4 10
Answer: 43
```

The input: “3 + 4 * 10” is output in postfix, infix, and prefix. Then the result of the calculation is output using our InfixToPostfix to create a tree and our Evaluate function to calculate the answer with the RPN algorithm. These steps are confusing when strung together in a sentence but splitting them up into a modular structure with reusable and separated code blocks allows for a much cleaner and maintainable codebase.



Prototype Conclusion

Overall, the prototype has worked quite well – it is correctly able to understand the usage of brackets and nesting statements and creates an abstract syntax tree correctly based on it. I will likely use this prototype or something similar as an actual part of the project for evaluating mathematical expressions too. It has given me a better understand of how syntax is tokenised, parsed and computed. I have found a problem with this current algorithm implementation – **it doesn’t support unary minus** (to negate an operand, e.g “**-1** + 1”) as the ‘-‘ is treated as a binary operator, not unary. I need to design a modification for this.

Development Choices

Having built this prototype and gained an idea of the scale and complexity of the project and therefore I feel that it is best to take an **agile approach** – I am going to work on each objective (e.g tokeniser, parser, etc.) as sprints. This is almost required for a project like this as each module relies on the previous one’s output or result (tokeniser -> parser -> execution), separating tasks into smaller reachable objectives and considering user input throughout each stage.

Summary: How Is This A-Level Standard?

This is not intended to be an overly complex programming language – it will contain basic features and syntax. However, the foundation required to create an interpreter is complex itself. These are the complexities:

Expression Evaluation

As previously mentioned, evaluating mathematical expressions that require order of operations to be considered requires complex algorithms and data structures:

- Use of **abstract syntax trees (ASTs)** to map out expressions
- **Algorithm** making use of **multiple stacks** to convert infix expressions to ASTs
- Use of **recursive graph traversal** to convert ASTs into postfix ordered list
- Use of **Reverse Polish Notation algorithm** to calculate result of postfix expression

Tokenisation & Parsing

These are the key modules of an interpreter and are not inherently complex but quickly become so:

- **Complex algorithm(s)** required for the tokenisation of a file and for parsing
- Data structures such as **queues, stacks and dictionaries** required throughout
- A **complex inheritance system of objects (OOP)** required for parsing and evaluating

Object Orientation

OOP will be essential in every part of the project – this will include usage of polymorphism, inheritance, and abstract classes.

Summary: Objectives

Core Functionality

1. Functioning Lexer Module

   As previously explained, a lexer breaks a string down into groups of characters (tokens) and gives context with them where possible. This allows a file containing characters to be broken into smaller and more understandable chunks that can then be fed into the parser to be made sense of. This process will likely require **OOP** and a **complex algorithm**.

   The 'Lexer' should be able to isolate variable names, expressions, and operators, along with control flow statements like 'if'. It should be able to produce a list of tokens in the order of the program, with context for each.

2. Functioning Parser Module

   This module will need to be able to take what the 'Lexer' has output and make sense of it. A parser will extract the important information from the tokens, group it and create a program structure for the Evaluator to follow.

   This parser should be able to take input of the list of tokens from the 'Lexer' and build 'program steps' based on them. These represent individual program statements.

3. Functioning Evaluator Module

After the parser has collected the instructions that are required to be executed, along with their order, the interpreter should then work through them and carry out each one – this could be taking input, giving output, variable assignments, comparisons, etc. Taking input should directly interface with the console to require the Enter key to be pressed to move on, and output should be sent directly to the console too.

There should be an executing module which takes input from the parser and goes through each instruction in order. It needs to determine the directions and data of these instructions and execute them, making use of direct console engagement.

4. Error Handling

Error messages for this interpreter will only be caused by a small number of problems, such as a syntax error or invalid data types. These will need to stop the program running altogether without crashing the console.

When errors are encountered, the Interpreter must stop without 'crashing' or 'hanging'. This is not to be confused with the *interpreted program* crashing – it means that our Interpreter will not crash itself when finding errors in the interpreted program.

5. Syntax

The following syntax objectives need to be met, as defined after the **interview section**:

- SYNTAX OBJECTIVE 5.1: Working output functionality
    - A meaningful and understandable identifier that any beginner will be able to use
    - Example syntax (subject to change) should be valid:

```
output("Hello World");
```

    Following **output**(EXPR); where EXPR can be:

        - A maths expression, such as '2 + 3 * (36-6)'
        - A string expression, such as "Hello " + "World!"
        - An expression with variables, such as 'x + 1'
    - *addition from later thought*: **outputln** should be a variation of this that adds a newline to the end of the message.

- SYNTAX OBJECTIVE 5.2: Working input functionality
    - A meaningful and understandable identifier that any beginner will be able to use
    - When calling for input, there should be an automatic output indicating action is required – for example "INPUT: "

- Example syntax (subject to change) should be valid:

```
string x = "";
input(x);
```

  Following **input**(VARIABLE); where VARIABLE is a valid reference to a pre-existing variable declared earlier on in the program.
  This will take input, and when Enter is pressed it will store that input into the variable "x" and move on.
  *addition from later thought: **input** should actually be split into **inputStr** and **inputInt** to distinguish which is required instead of later conversion

- SYNTAX OBJECTIVE 5.3: Strongly typed variable naming and no reassignment
  - Variables should require a type to be stated when being declared, though after that they do not need to be referenced with their types
  - Reassigning a variable (or attempting to declare it twice) should not be allowed
  - Example syntax for declaration and usage (subject to change) should be valid:

```
int x = 2 + 3*(2-1);
x = 2;
x = x + 1;
```

  Following TYPE VARNAME = DATA; where:

  - TYPE can be:
    - INT
      - Any number of digits representing a whole number, can be negative. Any variable declared with an expression such as 2+2*(36) will automatically be represented as the integer result of the calculation
    - String
      - Collection of characters; can be empty. Denoted by quotation marks.
    - **EXTENDED SYNTAX OBJECTIVE 5.3.1**: Boolean
      - Can either be True or False, nothing else is valid. This is extended as it adds another layer of complexity to comparisons and expressions and requires logical expression analysis.
  - VARNAME can be any collection of letters, no digits allowed
  - DATA can be any valid expression or raw data such as a String or Int – **must match up with** TYPE **or will cause an error**.
- SYNTAX OBJECTIVE 5.4: Code blocks should be denoted by brackets, not indentation
  - An opening bracket '{' marks the beginning of a code block
  - A closing '}' marks the end
  - Optional indentation for easier reading, white space is allowed but not important
  - Example syntax for code block, using if statement example:

```
if (condition) {
    ...
}
```

  In theory, infinite nesting should be possible.

- **SYNTAX OBJECTIVE 5.5:** Functional two-operand comparison if statements
  - An if statement should be able to correctly evaluate a comparison between two operands, running the following code block if the comparison returns true
  - Example syntax for if statement (subject to change):

```
if (condition) {
    ...
}
```

    Following **if**(CONDITION) { CODEBLOCK } where:
    - CONDITION is a two-operand comparison in the form of: OPERAND COMPARATOR OPERAND
    - OPERAND can be any expression
      - E.g 2 + 2, “Hello”, etc.
    - COMPARATOR can be:
      - >
      - <
      - ==
      - !=
      - <=
      - >=
    - Example: 2 > 4, countVariable == 10
    - CODEBLOCK can represent any valid program statements

6. Overall Facilities & Features

Provided the prior listed objectives are complete, there should be enough features of the language overall for a beginner to write simple programs as solutions to tasks. For example, a beginner should be able to create programs that solve tasks such as below:

Task: *Write a simple I/O program that takes input of a name (string) and outputs ‘Hi, <name>’*

Example Solution:

```
string name = "";

inputStr(name);

outputln("Hello, " + name);
```

Task: *Output all numbers from 1 -> 99 inclusive.*

Example Solution:

```
int number = 1;

while (number <= 99) {
    outputln(number);
    number = number + 1;
}
```

Solutions such as these examples should be fully functioning programs and solve the tasks – all language features in them should work.

A general list of beginner programs that should be possible to write in our language:

- Simple **input/output with variables**

    - E.g output name, calculation result, etc.

- String **comparison and validation**
    - E.g input a guessed name, output whether it is the right one

- **Number manipulation and calculation**
    - E.g input two numbers and output the result of a multiplication

- Complex **if + else statements**
    - E.g input two numbers, if one is the square of the other then output the result, else output a different message

- **While loop usage (extended)**
    - E.g output all numbers from 1->99
    - E.g Guess password infinitely until correct
        - Extension: Guess password until out of guesses

As part of the **Testing** section I will write some of these programs in different variations and run them to see if they work.

7. Freedom of Solution

Leading on from Objective 6, there should be a high enough level of complexity in the interpreted language to solve solutions in many **different** ways.

For example, to solve the '**Output all numbers from 1-99 inclusive.**' task, it should be possible to write and run any of these programs:

```
int number = 0;

while (number < 99) {
    number = number + 1;
    outputln(number);
}
```

```
int number = 0;

while (number < 99) {
    outputln(number+1);

    number = number+1;
}
```

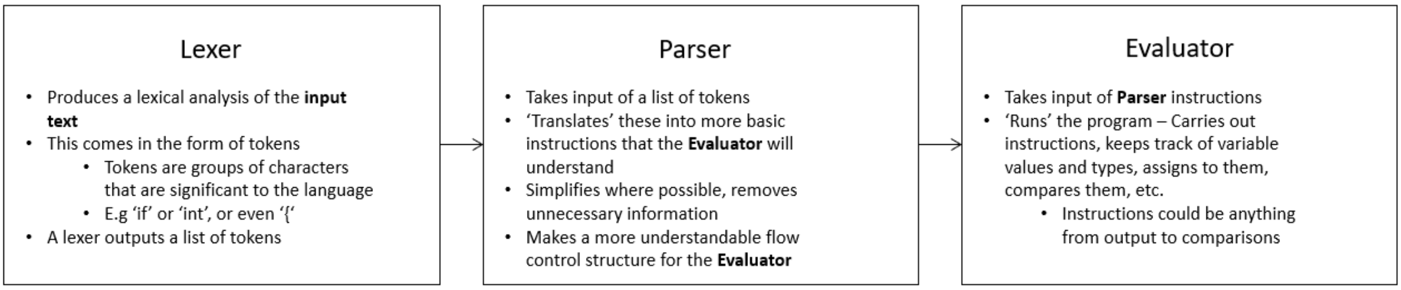
```
int number = 1;

while (number <= 99) {
    outputln(number);
    number = number + 1;
}
```

All of these should run and do the exact same thing – **the language is complex enough to write the same program in different ways.** This can only be measured through extensive testing.

Design

Overview

Writing an interpreter is like chaining separate modules together that each pass along a new piece of information. There are three modules: the Lexer, Parser and Evaluator. Data is passed 'along' each of these modules.



Input/Output

IO for this program will be extremely simple – it will be **console-based**. After the user has written a file with their code to run, they will be able to launch the interpreter executable:

```
Enter a valid file name:
```

This message is all that is required for an input prompt – there is no need for a complex GUI or syntax editor, as the programs being written for this interpreter will not be large or confusing.

As for output, other than 'output' statements themselves (which will print directly to the console), the only output will be the following:

Program Start & Finish

```
---- PROGRAM START: test.txt ----
```

```
---- PROGRAM FINISHED: test.txt ----
```

Error Output – likely to be revised

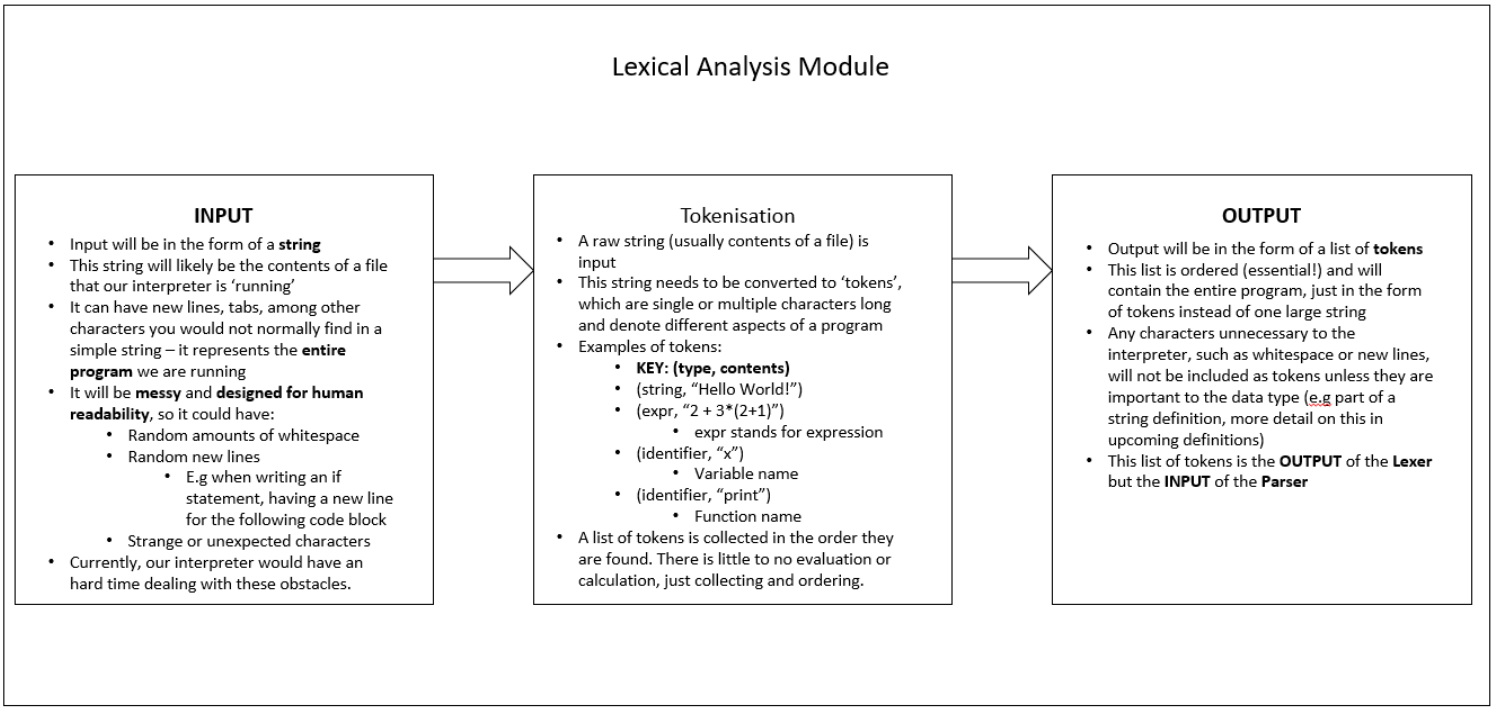
```
Error in SYNTAX, please check program code is valid.
```

```
Error in VARIABLE_REFERENCE, please check program code is valid.
```

The level of description of each error along with any other important information (e.g line, character number) are listed as **extended objectives**.

Lexer

Beneath is a model of what a Lexer does.



Before creating this module, we need to have a defined grammar for the language, so we know what we are going to be analysing.

```
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<integer> ::= <digit>|<digit><integer>

<char> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<identifier> ::= <char>|<char><identifier>|<identifier><digit>|<identifier><digit><identifier>

<grammar> ::= (|)|{|}|=|;

<operator> ::= +|-|/|*|^
```

These definitions are in Backus-Naur Form. The '**identifier'** definition could be the **identifier** of a function, keyword, or variable.

__revised addition to BNF definitions:__ <grammar> should also have '>' and '<' and '!'

Main Tokenisation Algorithm

```
TOKENS <- []
WHILE MORE_CHARACTERS() DO
    CHARACTER <- NEXT_CHAR()
    IF CHARACTER IS TAB OR SPACE OR NEW_LINE THEN
        CONTINUE                              // SKIP UNINMPORTANT CHARACTERS

    ELSE IF CHARACTER IS IN "+-*/^" THEN
        TOKENS.ADD(("OPERATOR", CHARACTER)) // ADD TOKEN WITH (TYPE, VALUE)

    ELSE IF CHARACTER IS IN "(){};=<>" THEN
        TOKENS.ADD(("GRAMMAR", CHARACTER))  // ADD GRAMMAR TOKENS

    ELSE IF CHARACTER IS DIGIT THEN
        ...                                   // CAPTURE NUMBER
    ELSE IF CHARACTER IS LETTER THEN
        ...                                   // CAPTURE IDENTIFIER
    ELSE IF CHARACTER IS '"' THEN
        ...                                   // CAPTURE STRING
    ELSE THROW EXCEPTION
ENDWHILE
```

This is a **pseudocode skeleton** for the **tokenising** algorithm. Tokens will be an object in themselves, but for the sake of simplification they are just tuples here.

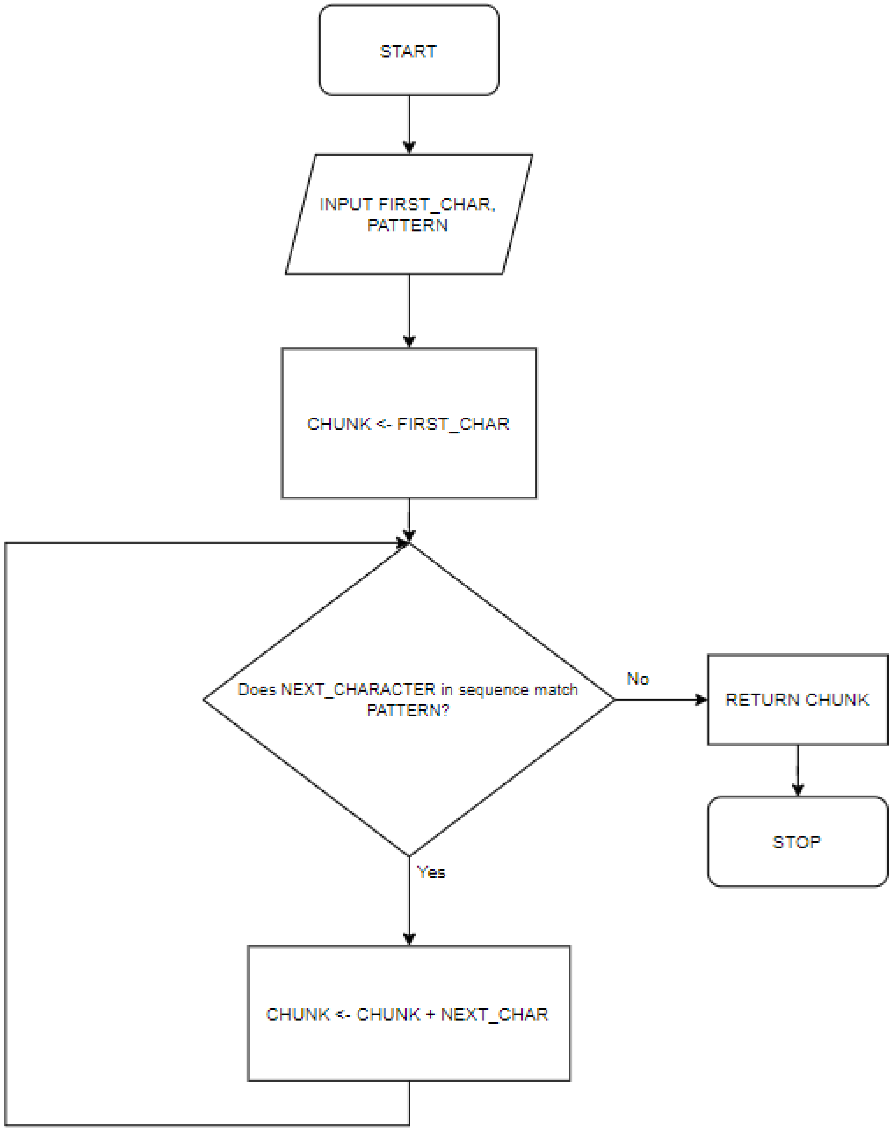
Capturing Numbers (Only Integers)

When encountering a digit **first**, we are expecting an integer token – as defined in our BNF rules:

```
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<integer> ::= <digit>|<digit><integer>
```

This means that when we find a digit, we need to search for any following digits and collect them all as one token. To do this, we need a 'grab' function.

Grab Algorithm using PATTERN

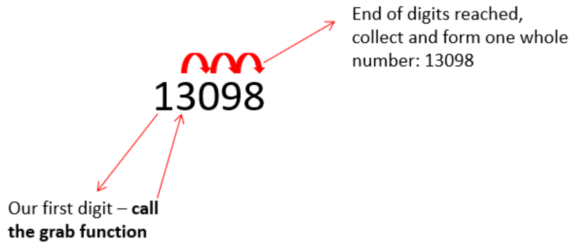


NEXT_CHARACTER in sequence refers to an outside object method that will return the next character in the original string being tokenised.

PATTERN refers to an input Regular Expressions pattern (for a single character) – for our specific case, we are just looking for a number, which consists of digits.

Therefore, for **CAPTURE NUMBER**, the regex pattern we will use will be `[0-9]`, which represents any numerical digit.

Here is an outline of how the number would be captured:



Each red arrow represents a single iteration in the above loop. At the end, the full number is returned as a **number token**.

CAPTURE IDENTIFIER – SMALL ALGORITHM

The same 'grab' algorithm is used for identifiers, but this time with a different pattern input: **`[a-zA-Z0-9]`**. Note that these patterns are for single characters, not whole strings – therefore they do not need any extra symbols to represent 1 or more etc.

An identifier can contain any digits or letters, but must start with a letter:

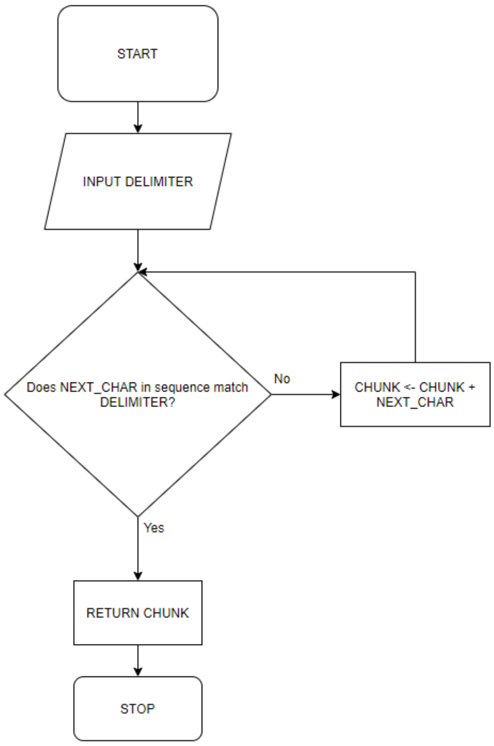
```
<identifier> ::= <char>|<char><identifier>|<identifier><digit>|<identifier><digit><identifier>
```

Therefore, we only begin to **CAPTURE IDENTIFIER** when we find a letter:

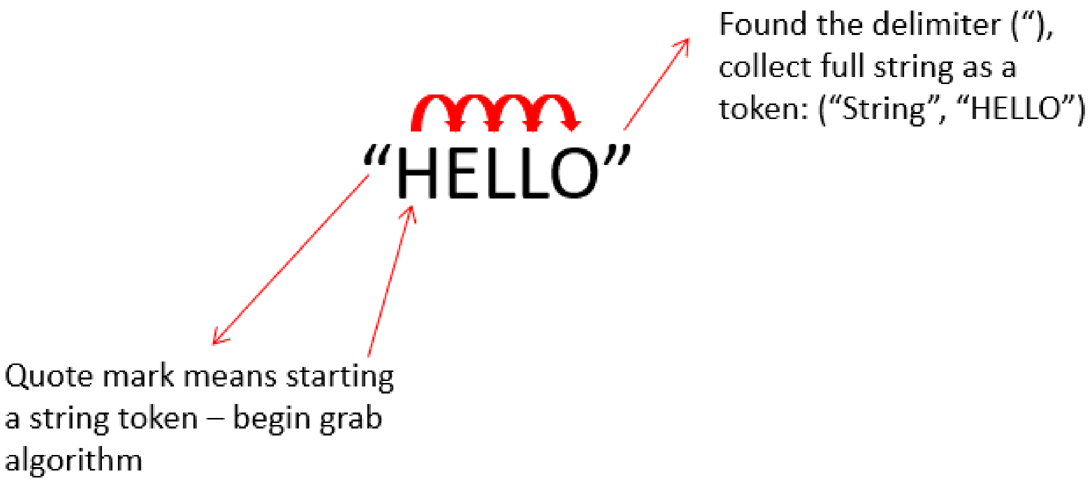
```
ELSE IF CHARACTER IS LETTER THEN
   ...                              // CAPTURE IDENTIFIER
```

CAPTURE STRING – SMALL ALGORITHM

Capturing a string requires a modified version of our 'grab' algorithm. Instead of using a pattern to collect characters, we need to collect **any** type of character until we hit a specific one (in the case of a string it will be "):



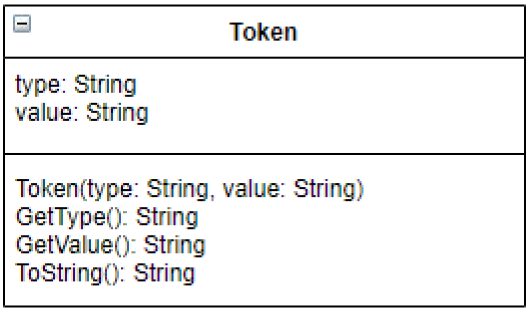
Using this algorithm to capture a string as one token works as follows:



Note that although the token is represented as ("String", "HELLO"), the quotation marks are NOT captured as the **contents** of the string.

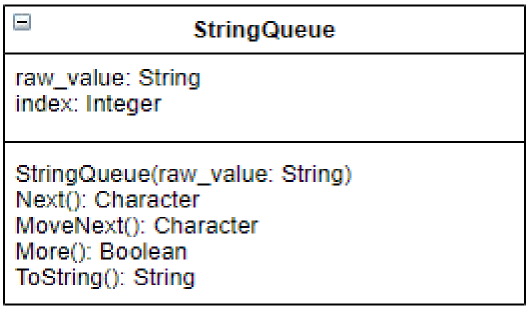
Tokenisation Algorithm Wrap-Up

When all these algorithms are used together as part of the main tokenisation algorithm, they should produce **tokens**:



The **`ToString()`** method returns the **token** in the format (type, value) – this is the format that we have been referring it to as in the previous diagrams, e.g ("String", "HELLO").

Also, to move through the characters as a **sequence** and not just loop through a string, we need to create a special type of string that is a modified **queue** structure:



post-analysis addition: Rename to CharQueue as it's more descriptive.

The **Next()** method returns the **next character in the sequence**, but does not move the index forward – **like the 'peek' method in a queue**.

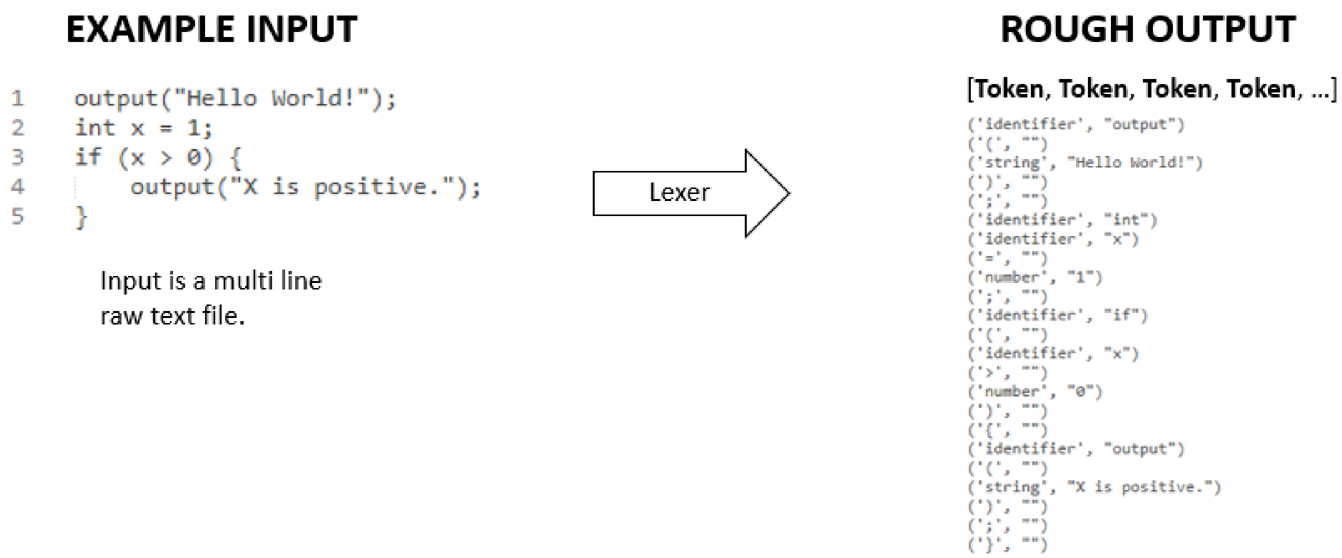
The **MoveNext()** method returns the **next character in the sequence**, but also **increments** the index – **like the 'pop' method in a queue**.

The **More()** method returns a **Boolean** representing whether or not there are more characters left in the sequence or if we have reached the end.

The **ToString()** method returns the **raw_string** variable – the original input string.

The reason for creating a queue-like structure is so that we can iterate through a 'list' of **characters** in different methods **simultaneously**, all at the same index. If our main algorithm calls MoveNext() to move along the index, then calls a 'capture' method which gets the next 5 characters in the queue, the main algorithm will be on the same index (5 characters ahead now). If we used a simple list implementation, we would have to pass by reference everywhere.

These algorithms should work together to produce the following result:



<u>revised design addition:</u> Grammar tokens such as "=" or "(" should be ("**grammar**", "=") instead of just ("=", "").

This output will be input into the **Parser** module.

Parser

Below is what the **parser** will do.

Parser Module

INPUT

- Input will be a **list of Token objects**
- This is the **output** of the **Lexer** module
- Each Token has a **type** and **value**
- This list contains unnecessary information for the **Evaluator**, so the **Parser** needs to remove it and create a new list of **useful information**

Build Evaluation Steps List

- As this is a small language, we can create a simple ordered list of 'steps' to evaluate for the **Evaluator** module to run
- Tokens are grouped together **and important information is extracted** and **checked syntactically**
- Once a pattern is recognised, a **Step object** is created and added to an ordered list
- A **Step** could be an **Event** or **If Statement:**
  - **Events**
    - Variable manipulation, function calls, etc.
  - **If Statement**
    - If statement AND code-block contained
- **NOTE:** The parser will parse EVERYTHING, it is **not running** the program – if there is a **false if statement** then its contents will **still be parsed**.

OUTPUT

- Output will be an ordered list of **Steps**
  - **Event** and **If Statement** classes both inherit from **Step** so both can be stored in the same list
- Each object has:
  - **Operands** – useful tokens that represent data being manipulated
  - **(Event only)** Event type – e.g VAR_DECLARE, FUNC_CALL
  - **(IfStatement only)** Comparison operands, code block operands
    - Comparison operands are the tokens in the condition part, e.g (x == 2)
    - Code block operands is another list of **Steps** which represent the **If Statement's** contents
- The **Evaluator** module will **evaluate** this list

Here is the process we are aiming to complete:

String is tokenised by **Lexer**

```
int x = 1;
if (x > 0) {
    output("X is positive.");
}
output("Comparison done.");
x = x + 1;
if (x > 0) {
    output("After increment, X is positive.");
}
output("New comparison done.");
```

Lexer

```
('identifier', "int")
('identifier', "x")
('grammar', "=")
('number', "1")
('grammar', ";")
('identifier', "if")
('grammar', "(")
('identifier', "x")
('grammar', ">")
('number', "0")
('grammar', ")")
('grammar', "{")
('identifier', "output")
('grammar', "(")
('string', "X is positive.")
('grammar', ")")
('grammar', ";")
('grammar', "}")
('identifier', "output")
('grammar', "(")
('string', "Comparison done.")
('grammar', ")")
('grammar', ";")
('identifier', "x")
('grammar', "=")
```

(This program would have more tokens than those shown here)

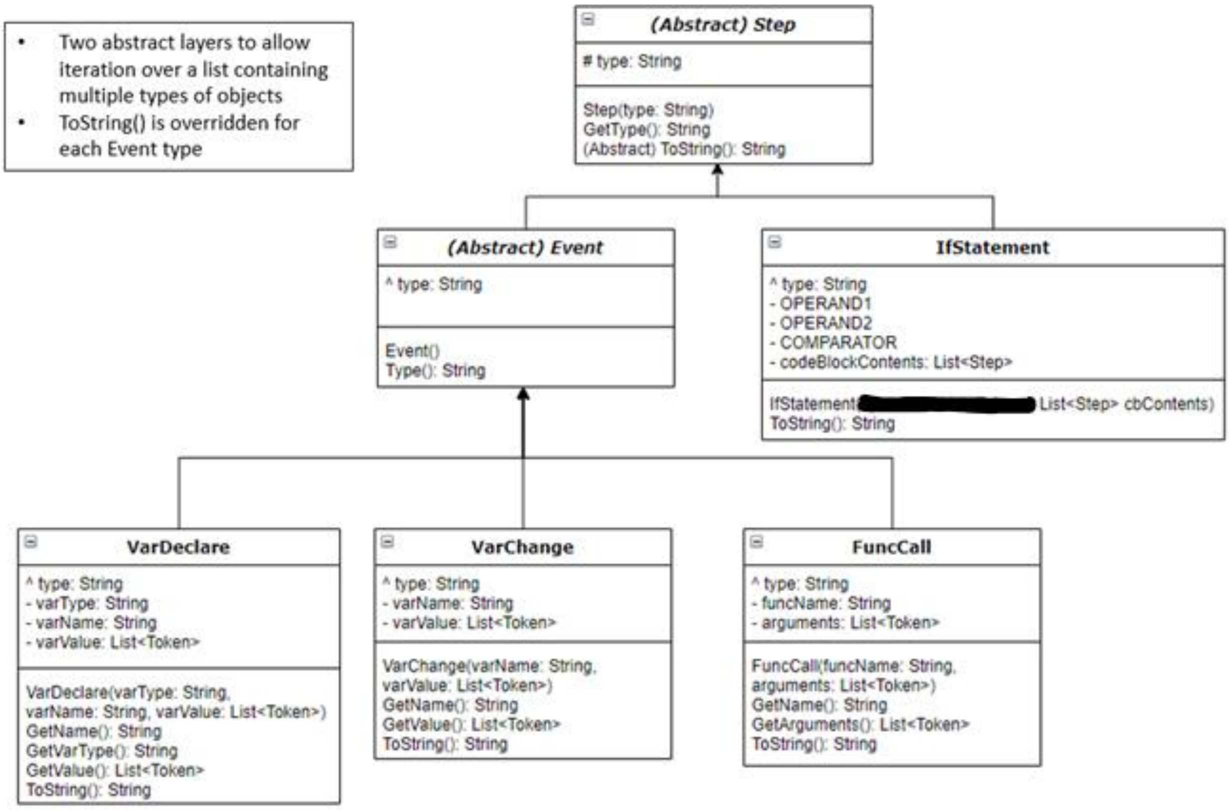
Tokens are grouped and **Step** list is built by **Parser**

Parser

```
[Event, IfStatement, Event, Event, IfStatement, Event]
          [Event]                    [Event]
```

List of Tokens -> List of Step objects

Classes & Inheritance



**post design decision to remove argument from IfStatement constructor.*

This is an **example output** for the Parser:

```
[Event, Event, IfStatement, Event]
```

- Each ***Step*** could be an:
  - **IfStatement**
  - ***Event***
- Each ***Event*** could be a:
  - **VarDeclare (Variable Declaration)**
  - **VarChange (Variable Change)**
  - **FuncCall (Function Call)**
- An **IfStatement** *has-a* **List of Step objects** (Composition)
  - This is the '**codeBlockContents**'
  - This could hold more **Events** or **IfStatements**
  - More explained on this design in the If statement 'capture' algorithm following
  - (extended) **ElseStatement** object – inherits from **Step**. Also has **codeBlockContents**.
- (extended) A **WhileLoop** inherits from an **IfStatement**
  - As they follow the same structure (While & If) I found I could reuse the **IfStatement** object's properties for a while loop (inherit from it)
  - A **WhileLoop** has a **codeBlockContents** just like an **IfStatement**

We need to design an overall 'parsing' algorithm that can detect the **beginning** of one of these programming statements, and then call functions that will **capture** all required data from the **Tokens** we have been given.

General Parsing Algorithm

This is a simple language; therefore, every programming statement begins with an **identifier** (see BNF definitions on **Page 26**).

```
1   WHILE MORE_TOKENS() DO
2       TOKEN <- NEXT_TOKEN()
3
4       IF TOKEN.TYPE IS 'identifier' THEN
5           // Could be any word outside a string
6           // SEE 'identifier' BNF DEFINITIONS
7
8           IF TOKEN.VALUE IS IN ['int', 'string', 'bool'] THEN
9               // Beginning of variable declaration
10              // We are here: int x = 0;
11              //             ^^^
12              EvaluationSteps.ADD(CAPTURE_VAR_DECLARE())
13              // Append Step list with new 'captured' Step
14
15          ELSE IF TOKEN.VALUE IS 'if' THEN
16              // If statement found
17              // We are here: if(x == 1) {
18              //             ^^
19              EvaluationSteps.ADD(CAPTURE_IF_STATEMENT())
20
21          ELSE IF PEEK_NEXT_TOKEN().VALUE() IS '(' THEN
22              // We have ruled out if statements
23              // Therefore this must be a function call
24              // We are here: output("Hello World");
25              //                    ^
26              EvaluationSteps.ADD(CAPTURE_FUNC_CALL())
27
28          ELSE IF PEEK_NEXT_TOKEN().VALUE() IS '=' THEN
29              // Change to pre-existing variable
30              // We must be here: x = x + 1;
31              //                    ^
32              EvaluationSteps.ADD(CAPTURE_VAR_CHANGE())
33
34          ELSE THROW SYNTAX_ERROR() // No pattern recognised
35      ELSE THROW SYNTAX_ERROR()
36      ENDIF
37  ENDWHILE
```

*Extended: **`ELSE IF TOKEN.VALUE IS 'while' THEN …`**

This will be added ~line 20. It will **reuse** the **`CAPTURE_IF_STATEMENT()`** as a while loop follows the same structure as an if statement – the data captured will just be added to a **WhileLoop** instead of an **IfStatement**.

*Extended (2): Checking after IfStatement capture if there is an ElseStatement next – if so this will be added right after IfStatement on the EvaluationSteps list.

- We can assume that if the first identifier is, for example, 'int', we are at the beginning of a variable declaration, e.g: **`int x = 0;`**
- We can assume that if the first identifier is "if", we are at an if statement, e.g: **`if(x==1){}`**
- We can assume if the **next** token (peek it, don't pop) is '(' and we **haven't** found an if statement, then it **must** be a function call, e.g: **`output(x);`**
- Finally, we can assume if the **next** token is '=' then we **must** be at a variable change (not declaration!), e.g: **`x = x + 1;`**

This routine is the foundation of the parsing process – when we find these we simply call a capture function to gather the necessary information, then move on.

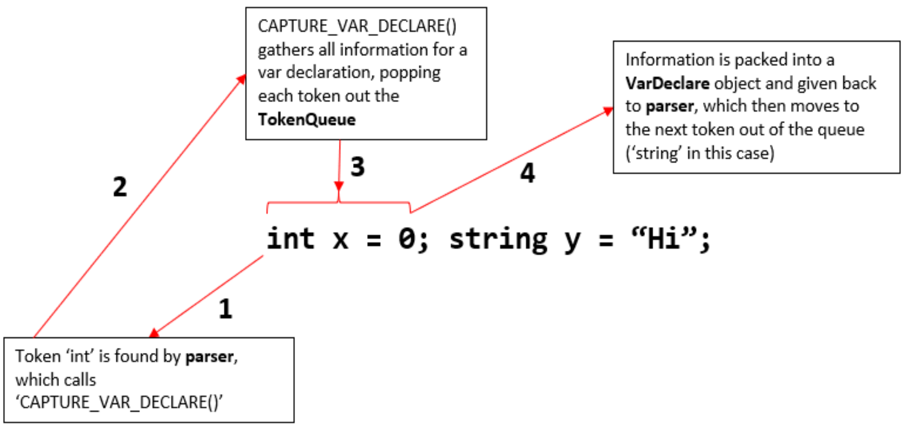
You will notice this looks similar to the **Tokenisation** algorithm (e.g MORE_TOKENS(), NEXT_TOKENS()). We will need a **TokenQueue** data structure just as we used the **StringQueue** in the **Tokeniser:**

```
TokenQueue
--------------------------------
tokens: List<Token>
index: int
--------------------------------
TokenQueue(tokens: List<Token>)
Next(): Token
MoveNext(): Token
More(): bool
ToString(): string
```

This allows us to go through a **global** 'queue' of tokens via different functions, all keeping up on the same index, so that **we don't parse tokens that have already been 'captured'** by other functions.

Here is an example of a **one iteration of the parsing algorithm, order of running is denoted by numbering**:

Statements are broken down to an '**atomic**' level to represent only information useful to the **parser**:




```
1   int x = 1+2;
```

(Represented as a string for demonstration – will actually be a list of tokens as given to the parser by the tokeniser.)

VarDeclare Object

```
VarType: 'int'
VarName: 'x'
VarValue: ['1', '+', '2']
```

*EXAMPLE FUNCTION: Capturing Variable Declarations

```
FUNC CAPTURE_VAR_DECLARE()
    VarType <- CURRENT_TOKEN() // This function has only been called because the parser has FOUND a VarType, so we must be on it in the queue

    NextTok <- NEXT_TOKEN() // Pop next out of queue

    IF NextTok.TYPE IS 'identifier' THEN
        VarName <- NextTok
    ELSE THROW_SYNTAX_ERROR()
    ENDIF

    IF NEXT_TOKEN() IS NOT '=' THEN // Simultaneously pop next token out the queue and check it follows the pattern
        THROW_SYNTAX_ERROR()
    ENDIF

    VarValue <- [] // Our 'value' could be an expression consisting of many tokens, so let's store it as a list of them

    WHILE (MORE_TOKENS() AND PEEK_NEXT_TOKEN() IS NOT ';') DO
        // Add all tokens until we reach ';' (the end)
        VarValue.ADD(NEXT_TOKEN())
    ENDWHILE

    NEXT_TOKEN() // To skip the ';' at the end

    // We have all the information, now return the object:
    RETURN NEW VarDeclare(VarType, VarName, VarValue)
ENDFUNC
```

This function **gathers** all information about the suspected **variable declaration** while also moving along the **global token queue**, then returns all the useful information in a single **VarDeclare** object.

All one-line long programming statements can be captured in essentially the same way – checking for grammar tokens (e.g '=', '{') and skipping them, collecting **important tokens that represent data** and returning objects with this data in **atomic** form:

CAPTURE_VAR_CHANGE()

```
x = x + 1;
```

VarChange Object

```
VarName: 'x'
VarValue: ['x', '+', '1']
```

CAPTURE_FUNC_CALL()

```
output(x + 1);
```

FuncCall Object

```
FuncName: 'output'
Arguments: ['x', '+', '1']
```

FUNCTION: Capturing 'If' statements

An **IfStatement** object has attribute **codeBlockContents**, which is a list of **Step** objects – this represents lines 6-8 in this 'If' statement:

Stored in **`IfStatement.codeBlockContents`** as List<Step>

`[Event, Event, Event]`

```
5  if(x == 1) {
6      x = 2;
7      output(x);
8      int y = 0;
9  }
```

Storing a list of steps inside each **IfStatement** allows us to have infinitely nested 'If' statements:

```
5   if(x == 1) {
6       x = x + 1;
7       if (x > 0) {
8           if (x < 3) {
9               output(x);
10          }
11      }
12  }
```

Line 5 If statement's **codeBlockContents**

`[Event, IfStatement]`

Line 7 If statement's **codeBlockContents**

`[IfStatement]`

Line 8 If statement's **codeBlockContents**

`[Event]`

Each **IfStatement** object stores all its contents – when an If statement is detected, its contents are packaged inside to form a single **IfStatement** object with a list of **Steps**

Due to the potential for nested If Statements, we will need a **semi-recursive** parsing process.

```
FUNC PARSE(TOKENS)
    ...
    IF TOKEN.VALUE = "if" THEN
        CAPTURE_IF_STATEMENT()
        ...
ENDFUNC

FUNC CAPTURE_IF_STATEMENT()
    COLLECT OPERAND1, OPERAND2, COMPARATOR // Collect contents of ()

    COLLECT CB_TOKENS // Collect tokens inside the {}

    IF_STATEMENT.CB_CONTENTS <- PARSE(CB_TOKENS) // Create Step list of tokens in code block
    IF_STATEMENT.OPERANDS <- IF_CONDITION
ENDFUNC
```

We will technically call the Parse function inside itself when we need to obtain a **Step** list of the contents of the If statement code block.

Also, as our language only supports 1 comparison in a condition (e.g x > 0) – we need to separate these into **OPERAND1**, **OPERAND2**, and **COMPARATOR**.

Here is how this function would be applied to an actual if statement:

```
if (x > 0) {
    output(x);
}
```

- **Line 9** of the pseudocode will collect the “X”, “>” and “0”
- **Line 11** of the pseudocode will collect the “output(x);” in token form.
- **Line 13** of the pseudocode will turn the “output(x);” tokens into a list of Steps: **[Event]** (only one event as there is only one statement in the code block)

Parser-Wrap Up

- The Parser is a big algorithm made up of smaller capturing routines
- These capture groups of **tokens** and pack them into specific **objects**
- These objects are either **IfStatement, WhileLoop, ElseStatement** or **Event** type
- They are packed into an **ordered** list called **EvaluationSteps**

After the **parser** has called all the relevant ‘capture’ functions, added new **Event** or **IfStatement** objects to a **List<Step> EvaluationSteps**, it is to be given as **input** to the **Evaluator**.

Evaluator

The Evaluator module is the final stage of interpreting the program. So far, we have gathered a list of **tokens**, grouped them into **Step** objects, and used the **Parser** to make a list of **Evaluation Steps** – now we need to go through each in order and decide which actions to take to **evaluate** them.

The structure of the Evaluator is like the previous modules – it is a main algorithm that calls multiple different smaller ones. Here is the main **Evaluator** algorithm:

```
FUNC EVALUATE(EVALUATION_STEPS)
    FOR EvalStep in EVALUATION_STEPS DO
        IF EvalStep.TYPE() IS "IF_STATEMENT" THEN
            IF COMPARE_EXPRESSIONS(EvalStep.OP1, EvalStep.OP2, EvalStep.COMPARATOR) THEN
                // If 'if' condition is true, 'run' the code block
                EVALUATE(EvalStep.GetCodeBlock())
            ENDIF

        ELSE IF EvalStep.TYPE() IS "VAR_DECLARE" THEN
            IF VARIABLES.CONTAINS(EvalStep.NAME()) THEN
                THROW DECLARE_ERROR()
                // Variable already exists
            ENDIF

            VarExpr <- RESOLVE_EXPRESSION(EvalStep.VALUE())
            // Resolve expression to one value token

            IF NOT EvalStep.VAR_TYPE() == VarExpr.TYPE() THEN
                // If the type declared does not match the expression type
                // e.g int x = "Hello World";
                THROW TYPE_ERROR()
            ENDIF

            VARIABLES.ADD(EvalStep.NAME(), VarExpr) // ADD variable and resolved value to dictionary

        ELSE IF EvalStep.TYPE() IS "VAR_CHANGE" THEN
            IF NOT VARIABLES.CONTAINS(EvalStep.NAME()) THEN
                // Variable does not exist, can't change it then
                THROW REFERENCE_ERROR()
            ENDIF

            VarType <- VARIABLES[EvalStep.NAME()].TYPE()
            // Get type of pre-existing variable, e.g 'int'

            NewValue <- RESOLVE_EXPRESSION(EvalStep.VALUE())
            // Resolve expression of new value to one single token

            IF NOT VarType == NewValue.TYPE() THEN
                // Type of new value does not equal pre-existing type
                // e.g assigning string to an int variable
                THROW TYPE_ERROR()
            ENDIF

            VARIABLES[EvalStep.NAME()] <- NewValue
            // CHANGE the variable's value in the dictionary

        ELSE IF EvalStep.TYPE() IS "FUNC_CALL" THEN
            IF NOT EvalStep.NAME() == "inputstr" or "inputint" THEN
                CALL_FUNCTION(EvalStep.NAME(), RESOLVE_EXPRESSION(EvalStep.ARGUMENTS()))

                // If not calling inputStr or inputInt, call function and resolve the arguments to one value
            ELSE
                // If you ARE calling inputStr/inputInt, we should not resolve the arguments as you're
                referencing a variable name, not value

                CALL_FUNCTION(EvalStep.Name(), EvalStep.ARGUMENTS()[0])
                // Arguments is a list but should only have one value for these functions: a variable name to
                input to
            ENDIF
        ELSE THROW SYNTAX_ERROR() // Unrecognised Step object.
        ENDIF
    ENDFOR
ENDFUNC
```

- **VARIABLES** represent a **dictionary** storing (VariableName, VariableValue)
    - **VariableValue** is a **single token** of type '**number**' or '**string**'
    - To get the type of a pre-existing variable, you just get the type the **VariableValue** token. E.g **`VARIABLES[EvalStep.NAME()].TYPE()`**

- **RESOLVE_EXPRESSION(List<Token> expr)** is a function that takes input of an expression in the form of a list of tokens, and **outputs a single token** which is the result of performing the expression.
- **CALL_FUNCTION(String Name, Token Argument)** is a function that enacts calling functions with a single argument. If the function name is 'inputStr' or 'inputInt', the argument to those functions is a **variable name** to store the input value in – therefore it should **not be resolved** (the values of the variable should not replace it in the argument, we need the variable **name** not **value**)

Resolving Expressions – Main Algorithm

```
FUNC RESOLVE_EXPRESSION(List<Token> Expr)
    toReturn <- Token("", "")
    // Declare for now as blank token but change later, always return this variable

    Expr <- VARS_TO_VALUE(Expr)
    // Replace all variable reference tokens with their actual values

    ExprType <- CHECK_TYPES(Expr)
    // This function checks the types of the tokens in the expression
    // If they are not ALL numbers or ALL strings, it will throw an error
    // If they are ALL numbers, it will return 'number'
    // If they are ALL strings, it will return 'string'
    // This represents the 'final' outcome of resolving the expr

    IF ExprType IS "string" THEN
        // String expression, must be concatenation
        IF (Expr.COUNT() == 1) THEN
            toReturn <- Token("string", Expr[0].VALUE())
            // If only one string in expression, final value is just that string
        ELSE
            IF Expr[0].TYPE() IS NOT "string" THEN
                // First value of expression is not string - invalid expression
                // e.g string x = + "Hello World";
                // Expr starts with '+' and not a string, INVALID
                THROW SYNTAX_ERROR()
            ELSE
                FinalResult <- Expr[0].Value() // First string in expression
                Index <- 1 // Already got first Token, start loop on next Token

                WHILE Index < Expr.COUNT() DO
                    IF Expr[index].TYPE() IS "operator" THEN
                        IF Expr[index].VALUE() IS "+" AND Index < Expr.COUNT() THEN

                            // Can concatenate next string
                            FinalResult <- FinalResult + Expr[Index + 1].VALUE()

                        ELSE THROW TYPE_MATCH_ERROR()
                        // Cannot do any operation other than "+" on strings.
                    ENDIF

                    Index <- Index + 1
                ENDWHILE

                toReturn <- Token("string", FinalResult)
                // Set toReturn to final result of string concatenation
            ENDIF
        ENDIF
    ELSE IF ExprType is "number" THEN
        // Maths expression

        RootNode <- CREATE_TREE(Expr) // Convert to Syntax Tree
        PostOrderList <- POST_ORDER_TRAVERSE(RootNode) // Post order list of visited nodes
        Result <- RPN_EVALUATE(PostOrderList) // Reverse Polish Evaluation of list for result

        toReturn <- Token("number", Result)
    ELSE THROW SYNTAX_ERROR() // Invalid expression type, must be 'number' or 'string'
    ENDIF

    RETURN toReturn
ENDFUNC
```

This algorithm needs to take **input** of an expression – a **list of tokens**. It then needs to 'resolve' these into a result and return this as a **single token**:

E.g for the expression **`1+4/2`**
`INPUT: ['1', '+', '4', '/', '2']`
`OUTPUT: '3'`

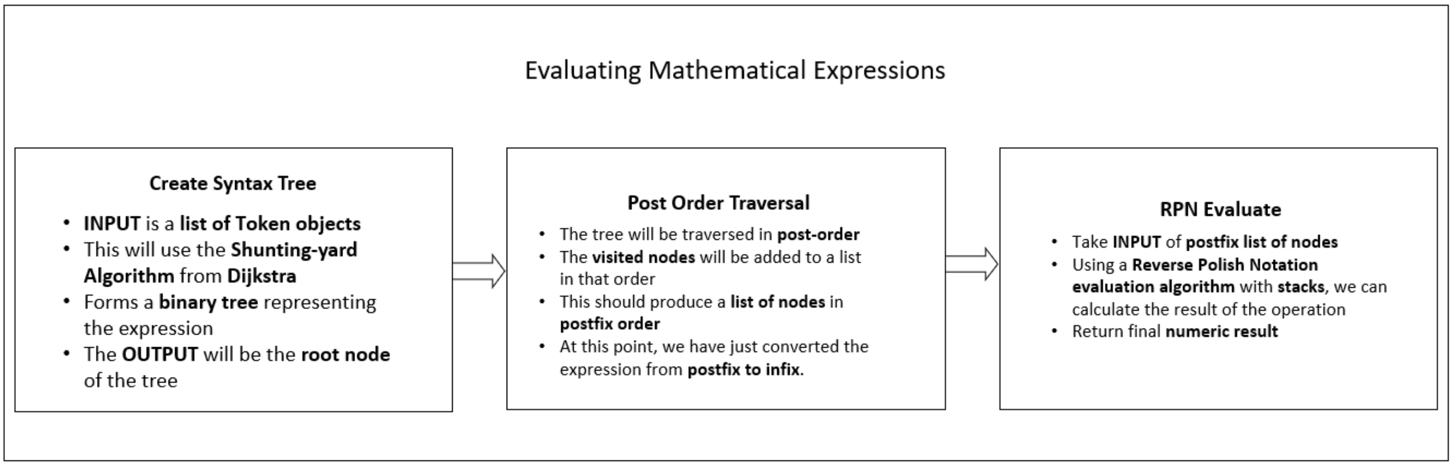
NOTE: If the expression contains a **VARIABLE** reference, then it will replace it with that value as long as it is a **number**:
E.g for the expression **`x+4/2`** where x exists with value '10'
`INPUT: ['x', '+', '4', '/', '2']`
`OUTPUT: '12'`

NOTE: If the expression contains a **STRING** reference, it will assume concatenation:
E.g for expression **`"Hello " + "World"`**
`INPUT: ['Hello ', '+', 'World']`
`OUTPUT: 'Hello World'`

To resolve **mathematical expressions**, we need to employ the same system that was used in the prototype.

Resolving Expressions – Mathematical

This is a more in-depth breakdown of the system I prototyped which is to be implemented.



Modified Shunting-yard Algorithm (Unary Support)

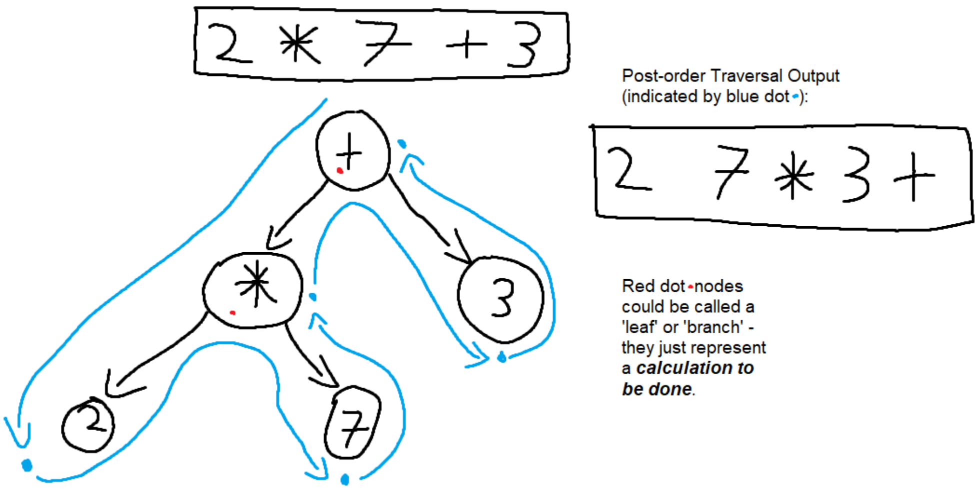
We need this algorithm to create a syntax tree – I have modified the original algorithm to create an AST and to support the unary minus, though this requires an extra routine to distinguish the unary minus from the binary, replacing it with "_":

```
func FIND_UNARY_MINUS(List<Tokens> tokens)

    // We are going to return the same list of Tokens but with unary minus changed from '-' char to '_' to distinguish it
    // e.g "-1 - 1" will return "_1 - 1" as the first '-' is unary.

    toReturn <- []

    index <- 0

    WHILE index < tokens.COUNT DO
        tok <- tokens[index]

        IF tok.TYPE() is "operator" AND tok.VALUE() is "-" THEN
            // token is a '-' operator

            IF index == 0 OR tokens[index-1].VALUE() is "(" OR tokens[index-1].TYPE() is "operator" THEN
                // It is an UNARY minus if:
                    - it is the first token in the expression
                    - it is right after a '('
                    - it is right after another operator

                tok <- Token("operator", "_") // reassign to unary
                // Instead of adding the '-' token, add '_' to signify UNARY
            ENDIF
        ENDIF

        toReturn <- toReturn + tok

        index <- index + 1
    ENDWHILE

    RETURN toReturn
ENDFUNC
```

Now that we have a small routine to replace all unary minus signs with a special "_" to distinguish them from regular "-" signs, we can implement the SY algorithm to handle the "_" separately.

Abstract Syntax Tree Building

This is the type of tree we are aiming to build:



We will build it with my modified Shunting-yard algorithm[7] (pseudocode following) and traverse it in post order to output the expression in **prefix**. This algorithm itself does **NOT** calculate anything – just **builds a map of what to calculate**.

[7] https://brilliant.org/wiki/shunting-yard-algorithm/ (accessed 25/02/2021)

```
func BUILD_TREE(List<Token> infix)
    infix <- FIND_UNARY_MINUS(infix) // replace unary minus with "_" to signify unary

    operatorStack <- Stack(TreeNode) // stacks both hold TreeNode object type
    numStack <- Stack(TreeNode)

    FOR token IN infix DO
        IF token.VALUE() is "(" THEN
            operatorStack.PUSH( TreeNode("(") )
            // beginning of a nested expression, just leave it until find closing ')'
        ENDIF

        ELSE IF token.TYPE() is "number" THEN
            numStack.PUSH( TreeNode(INT(token.VALUE())) )
            // Add string value converted to integer to the numStack
        ENDIF

        ELSE IF PRECEDENCES.CONTAINS_KEY(token.VALUE()) AND token.VALUE() is NOT ")" THEN
            // precedences is a dictionary of all operators and their precedence (int)
            // CONTAINS_KEY essentially checks if the VALUE() is an operator, e.g "+"

            WHILE operatorStack.COUNT > 0 AND PRECEDENCES[operatorStack.PEEK().VALUE()] >= PRECEDENCES[
            token.VALUE()] DO
                // This is a confusing while loop. In English:
                // 'While there are more items in the opstack AND the precedence of the item at the top
                of the opstack is bigger or equal to the precedence of the current operator we have
                found (token.VALUE())'

                // Simplified: 'while we continue to find operators with higher or equal precedence in
                the opstack'

                // We pop an operator off and create a LEAF calculation
                // a branch looks like this, e.g:
                //     +
                //  1    2
                // A branch represents a calculation to do.

                operator <- operatorStack.POP() // pop off next operator (to be root of branch)

                IF operator.VALUE() is "_" THEN
                    operator.LEFT <- numStack.POP()
                    // If it's an UNARY MINUS, only pop 1 operand (as it's unary)
                ELSE
                    // If NOT an unary minus, must be a binary operator, pop 2 operands
                    operator.RIGHT <- numStack.POP()
                    operator.LEFT <- numStack.POP()
                    // set right and left children of branch to numstack items
                ENDIF

                numStack.PUSH(operator) // now push the branch of calculation onto numStack

            ENDWHILE

            operatorStack.PUSH( TreeNode(token.VALUE()) )
            // Now that we've created branches for everything to the LEFT of our operator, we need to
            push the operator onto the stack.
        ELSE IF token.VALUE() is ")" THEN
            WHILE operatorStack.COUNT > 0 AND operatorStack.PEEK().VALUE() is NOT "(" DO
                // repeat of branch-making from above:

                operator <- operatorStack.POP()

                IF operator.VALUE() is "_" THEN
                    operator.LEFT <- numStack.POP()
                ELSE
                    operator.RIGHT <- numStack.POP()
                    operator.LEFT <- numStack.POP()
                ENDIF

                numStack.PUSH(operator)
            ENDWHILE

            operatorStack.POP() // DIFFERENT!!! Pop off the last operator as it will be '('
        ELSE THROW SYNTAX_ERROR()
        ENDIF
    ENDFOR

    WHILE operatorStack.COUNT > 0 DO
        // repeat of branch-making
        operator <- operatorStack.POP()

        IF operator.VALUE() is "_" THEN
            operator.LEFT <- numStack.POP()
        ELSE
            operator.RIGHT <- numStack.POP()
            operator.LEFT <- numStack.POP()
        ENDIF

        numStack.PUSH(operator)
    ENDWHILE

    return numStack.POP()
    // Return root node of AST (sitting at top of numStack)
    // NO CALCULATIONS HAVE BEEN DONE, JUST BUILT THE TREE.
```

- **PRECEDENCES** is a **dictionary** containing the precedence of the operators:

```
{"_", 5 }, // unary minus has a higher precedence than all below
{"^", 4 },
{"*", 3 },
{"/", 3 },
{"+", 2 },
{"-", 2 },
{")", 2 },
{"(", 1 }
```

Recursive Post Order Traversal

```
FUNC POST_ORDER_TRAVERSE(TreeNode Node)
    Nodes <- List<TreeNode>
    IF Node.LEFT THEN // If LEFT child exists
        Nodes.ADD(POST_ORDER_TRAVERSE(Node.LEFT))
    ENDIF
    IF Node.RIGHT THEN // If RIGHT child exists
        Nodes.ADD(POST_ORDER_TRAVERSE(Node.RIGHT))
    ENDIF
    Nodes.ADD(Node)
    // Root node at the end of list as post-order.

    RETURN Nodes
ENDFUNC
```

Reverse Polish Notation Algorithm

```
FUNC RPN_EVALUATE(List<TreeNode> nodes)
    nodeStack <- Stack(TreeNode)

    FOR treeNode in nodes DO
        IF treeNode is Number THEN
            nodeStack.PUSH(treeNode)
        ELSE IF treeNode is Operator THEN
            // treeNode.VALUE() is the operator in string form, e.g "+"

            IF treeNode.VALUE() is "_" THEN
                // unary, only pop off 1
                arg1 <- Number(nodeStack.POP())

                nodeStack.PUSH( Number(CALCULATE(arg1, -1, "*")) ) // unary minus so just multiply by -1
            ELSE
                arg2 <- Number(nodeStack.POP())
                arg1 <- Number(nodeStack.POP())

                // Note they are reversed order - first one popped off is 2nd arg
                nodeStack.PUSH( Number(CALCULATE(arg1, arg2, treeNode.VALUE())) )
            ENDIF
    ENDFOR

    RETURN nodeStack.POP() // return top of nodeStack
ENDFUNC


FUNC CALCULATE(arg1, arg2, operator)
    result <- 0

    SWITCH Operator
        CASE "+":
            result <- arg1 + arg2
            break
        CASE "-":
            result <- arg1 - arg2
        ...
        // have case for each operator: +, -, *, /, ^, etc.

        DEFAULT:
            break // result will be 0 already as default.
    ENDSWITCH

    RETURN result
ENDFUNC
```

This algorithm calculates the result of an input of a Postfix expression.

Evaluator Wrap-Up

- The **Evaluator** takes **input** of a **List of Step objects**
- If the **Step** is an **IfStatement** object it **checks the condition** and **runs the code block recursively** if it is **true**
    - If the **Step** is a **WhileLoop** it is treated as an **IfStatement** and continuously ran until the condition is **false**
- If the **Step** is a **VarDeclare** object it makes necessary checks and adds a new variable to a **dictionary**
- If the **Step** is a **VarChange** object it makes necessary checks then changes the value of that variable in the **dictionary**
- Finally, if the **Step** is a **FuncCall** then it will act out the actions of the called function with the argument given
- **Expressions** can be **resolved** into **strings** or **numbers**
    - When an expression is resolved, the **output** will be in the form of a single **Token** object.

Technical Solution

I have split the objectives into **core functionality** and **related**.

Core Program Objectives

- Functioning **Lexer** Module
- Functioning **Parser** Module
- Functioning **Evaluator** Module

Related / Broader Objectives

- All of the **syntax objectives** are broader – they cannot be 'reached' within a simple algorithm but are instead guides to follow throughout
    - I will consider these 'complete' in the **Parser (Core) module**, though they are 'reached' using all three modules
    - The **Parser** will recognise syntax, therefore reaching each of the objectives
- Custom **Error** Messages

Development Notes

➔ I have separated these so that I can focus on working on the modules while *including* the broader objectives throughout.

➔ I am going to **write each module separately** (agile sprints), but still make use of some classes I have written across multiple modules – these are classes such as the **Token**, which can be reused in all three modules.

➔ I have realised that there are easy ways to add meaningful feature additions at certain points in the development, such as ElseStatements or WhileLoops. These have **not** required significant design changes and instead just **reuse prior-designed code** – this demonstrates the potential for expansion and modularity of the system.

Error Handling & Messages

As any of these modules could come across an error in the interpreted program, it is probably best that the first thing I create is the ability to display these errors and stop the program. C# has a pre-existing **Exception** class – these can be used in C#'s **'throw'** keyword to produce an error manually.

➔ I will create custom classes for each error, all of which will inherit from the built in **Exception** class. These are the errors that the interpreter may find in the program:

- Invalid syntax
    - E.g 'int x !)()() = 10;'
- Invalid type for variable value in declaration
    - E.g 'int x = "Hello";'
- Mismatched variable types in the same expression
    - E.g "Hello" * 2
- Referencing a non-existent variable

    - Declaring an existing variable
    - Comparing mismatched types
        - E.g “1” == 1
- ➔ I also need a way of **handling** the error.
    - C#’s **Exception** class will close the console window when running into an error – I do not want this to happen right away as the user won’t see the error message.
    - I need a static **Error** class with static method **ShowError(string)**. This will be used to **pause the program** and **show the error message**. Once the enter key is pressed, the console window (and program) will close.

Static Error Class

```
static class Error
{
    public static void ShowError(string err) // Pause program (input prompt) then kill it.
    {
        Console.WriteLine(err);
        Console.ReadLine(); // Used to pause it, Enter required to move on
        Environment.Exit(1); // Exits process
    }
}
```

This static method will be used by each individual error class with pre-written message each. The **entire program** will quit when **Enter** is pressed.

Individual Errors

```
class SyntaxError : Exception
{
    public SyntaxError() : base()
    {
        Error.ShowError("SYNTAX error. Re-read your program and check for spelling/keyword mistakes.");
    }
}

class TypeError : Exception
{
    public TypeError() : base()
    {
        Error.ShowError("TYPE error. Declared type does not match value of expression type.");
    }
}

class TypeMatchError : Exception
{
    public TypeMatchError() : base()
    {
        Error.ShowError("TYPE_MATCH error. Types in expression are not compatible.");
    }
}

class ReferenceError : Exception
{
    public ReferenceError() : base()
    {
        Error.ShowError("REFERENCE error. Variable or function referenced does not exist.");
    }
}

class DeclareError : Exception
{
    public DeclareError() : base()
    {
        Error.ShowError("DECLARE_ERROR. Variable already exists.");
    }
}

class ComparisonError : Exception
{
    public ComparisonError() : base()
    {
        Error.ShowError("COMPARISON error. Cannot compare different types.");
    }
}
```

As we have inherited from the **Exception** class, we can use statements like this:

```
throw new SyntaxError();
```

No program code will be executed after one of these statements is reached – this is **important** as I will be using them **throughout** my program in this form:

```
...
if (found error) throw new SyntaxError(); // No code past this executed if true.
...
```

These are like customs checks at an airport – if a single error is found then the entire program must be stopped. There will almost always be code after these statements – I am not worried about base cases or the exit points caused by this as **any error is a critical error** and **nothing can be executed after**.

The usage of these throughout completes **Objective 4 – Error Handling**.

Lexer Module (Core)

This is the first core module that I have written in an Agile sprint. Before writing the **Main Tokenisation Algorithm**, I need to define the **Token** and **TokenQueue** classes.

Token Class

```
class Token
{
    private string type;
    private string value;

    public Token(string type, string value)
    {
        this.type = type; this.value = value;
    }

    public override string ToString()
    {
        return "('" + type + "', \"" + value + "\")"; // e.g ("grammar", "+")
    }

    public string Type()
    {
        return type;
    } // GetType() is a built-in C# method to get the type of variable, hence this is named Type() instead - no need to override.

    public string Value()
    {
        return value;
    }
}
```

I will not be naming my **getters** here as **GetX()** – a lot of built-in C# methods that refer to similar attributes start with **Get** and I want to avoid naming conflicts (it is also bad practice to override methods such as **GetType**).

TokenQueue Class

```
class TokenQueue
{
    private List<Token> tokens;
    private int index = 0;

    public TokenQueue(List<Token> inputTokens)
    {
        tokens = inputTokens;
    }


    public Token Next() // peek
    {
        if (!(tokens.Count > 0 && index < tokens.Count)) throw new SyntaxError();
        return tokens[index]; // Returns index
    }

    public Token MoveNext() // pop
    {
        if (!(tokens.Count > 0 && index < tokens.Count)) throw new SyntaxError();
        return tokens[index++]; // Returns index THEN increments it
    }

    public bool More() // check if able to peek or pop
    {
        return index < tokens.Count;
    }

    public List<Token> Contents() { return tokens; }
}
```

CharQueue Class

```
class CharQueue
{
    private string raw_value = "";
    private int index = 0;

    public CharQueue(string value)
    {
        raw_value = value;
    }

    public char Next() // peek
    {
        if (!(raw_value.Length > 0 && index < raw_value.Length)) throw new SyntaxError();
        else return raw_value[index]; // Returns index
    }

    public char MoveNext() // pop
    {
        if (!(raw_value.Length > 0 && index < raw_value.Length)) throw new SyntaxError();
        else return raw_value[index++]; // Returns index THEN increments it
    }

    public bool More() // check if can peek or pop
    {
        return index < raw_value.Length;
    }

    public string Contents() { return raw_value; }
}
```

Main Tokenisation Algorithm

Now that I've written the **Token** and **TokenQueue** classes, I can write the main algorithm used in the Lexer to **tokenise**. As stated in the design section, the **input** to the Lexer module will be a string (contents of a file) and the **output** should be a list of **Token** objects.

```
9      class Tokeniser
10     {
11         private CharQueue contents;
12
13         public Tokeniser(string input)
14         {
15             contents = new CharQueue(input);
16         }
17
18         // NOTE: The usage of RegEx here is STRICTLY for single character matching to make it quicker to write
19         // Instead of saying 'is character abcdefghijkmno....' we can just use [a-zA-Z] and check for match of the CHARACTER
20         // We are NOT using RegEx to find keywords or tokens themselves - that is inefficient for long programs and often gets complex.
21         public IEnumerable<Token> Tokenise()
22             // IEnumerable allows us to use foreach with yielding in C#
23             // Not REQUIRED, but means there is no need to explicitly create a list and return it, instead use 'yield return'
24             // 'yield return' allows you to return each element one at a time - this is just more memory efficient
25             //
26             // As we are always going to be dealing with a large amount of tokens (think each typed character in a program),
27             // we should be considerate of efficiency for memory and looping - we do not need to create a list in this method.
28             //
29             // In our case, we are returning each Token object in a list of Tokens
30         {
31             while (contents.More()) // While elements left in queue of chars
32             {
33                 char character = contents.MoveNext();
34
35                 // I'll be using the a shortcut to check what the character is: String.Contains(char)
36                 // This is faster to write than 'if character == "+" || character == "-" etc...'
37
38                 if (" \n\t\r".Contains(character)) continue; // We do not care about spaces or new lines, skip iteration
39
40                 // Operators
41                 else if ("+-*/^".Contains(character)) yield return new Token("operator", character.ToString());
42
43                 // General Guideline Grammar
44                 else if ("(){};=<>!".Contains(character))
45                 {
46                     // Check if more tokens past it then check if we've found a comparator operator like "==", ">=", "<="
47                     if (contents.More() && "=<>".Contains(character) && "=<>".Contains(contents.Next()))
48                         yield return new Token("grammar", character.ToString() + contents.MoveNext().ToString());
49                         // If the next token is ALSO "=" or "<" or ">", then add both tokens together as one to form "==", "!=", "<=", ">="
50
51                     else yield return new Token("grammar", character.ToString());
52                     // else just add single token
53                 }
54                 // Numbers (only supports integers)
55                 // These are not full RegEx statements - they are only to match a SINGLE CHARACTER AT A TIME.
56                 else if (Regex.IsMatch(character.ToString(), "[0-9]")) yield return new Token("number", Match_GrabChunk(character, "[0-9]"));
57
58                 // Identifiers - can have numbers, not at the beginning though
59                 // We look at first for a letter, then grab any following letters/numbers
60                 else if (Regex.IsMatch(character.ToString(), "[a-zA-Z]")) yield return new Token("identifier", Match_GrabChunk(character, "[a-zA-Z0-9]"));
61
62                 // Strings can be denoted by ' or " in our language. If we find one of those, grab the rest of the string:
63                 else if ("\"'".Contains(character)) yield return new Token("string", Char_GrabChunk(character));
64
65                 // Any strange or unrecognisable characters throw an error.
66                 else throw new SyntaxError();
67             }
68         }
```

edit: line 47 statement** *"=<>".Contains(character)* ***should be *"!=<>".Contains(character)*

This is an implementation of the *Main Tokenisation* algorithm that I have designed, along with the smaller 'grabbing' or 'capturing'. Once again remember that RegEx is only being used as a quick way of matching a **single character** – this lexer is **not** using RegEx to look for keywords.

IEnumerable usage: As explained in the comment block, this is a more efficient way of creating a function that uses a loop to build and return a list. Instead of declaring a new list at the top of the function, adding each element to it in the loop, then returning it, we can just use **yield return** to return each element one-by-one which forms a list. This output can then be assigned to a list later:

```
Tokeniser tokeniser = new Tokeniser(toRun);
List<Token> tokens = tokeniser.Tokenise().ToList();
```

Lexer Module Wrap-Up

The Lexer module is the least complex of the three, hence only consisting of the **Tokeniser.cs**, **Token.cs**, and **TokenQueue.cs** files. We can also write a quick method to take input/output for testing purposes:

```
public static void Run()
{
    //-------- MULTI-LINE INPUT --------
    string input = "";
    string newInput = "";
    do
    {
        newInput = Console.ReadLine();
        input += newInput;
    } while (newInput.Length > 0);
    //-------- END OF MULTI-LINE INPUT --------

    Tokeniser tokeniser = new Tokeniser(input);

    foreach (Token tok in tokeniser.Tokenise()) // Loop through each returned element
    {
        Console.WriteLine(tok.ToString()); // Output makes use of overridden Token.ToString() method
    }
}
```

This program takes input of multiple lines then outputs each Token it has found. Here is a 'dummy' program that I've input into it:

```
int testing123 = 1 + 20*(2/2)-1;
string helloWrld = "Hello World!    ";
if (test > test) {
        output(test);
}
```

And here is the (correct) output:

```
('identifier', "int")
('identifier', "testing123")
('grammar', "=")
('number', "1")
('operator', "+")
('number', "20")
('operator', "*")
('grammar', "(")
('number', "2")
('operator', "/")
('number', "2")
('grammar', ")")
('operator', "-")
('number', "1")
('grammar', ";")
('identifier', "string")
('identifier', "helloWrld")
('grammar', "=")
('string', "Hello World!   ")
('grammar', ";")
('identifier', "if")
('grammar', "(")
('identifier', "test")
('grammar', ">")
('identifier', "test")
('grammar', ")")
('grammar', "{")
('identifier', "output")
('grammar', "(")
('identifier', "test")
('grammar', ")")
('grammar', ";")
('grammar', "}")
```

- This was just a quick test to cover every ‘feature’ or ‘grouping’ our Lexer module should recognise and group into **Tokens**
- It is important to note that errors such as **TypeErrors** will **not** be recognised by our Lexer – it is **ONLY** grouping up words and symbols into **Tokens**
- Also note that the string “Hello World!   “ still has all the spaces preserved from input – it is important that we leave anything inside the quotations alone
- Grammar and operator symbols have been correctly recognised
- Identifiers have been distinguished from strings

The Lexer module has been finished, this test has completed **Core Objective 1 – Lexer Module**, though I am going to do more substantial testing later on.

Parser Module (Core & Related)

This module will be part of the **core** objectives (**Core Objective 2 – Parser Module**) but as it is recognising syntax, I will consider the **related** **Syntax Objectives** as complete if they are recognised by it.

We need to define the classes following our UML diagram in the **Design** section.

Step Class (Abstract, Top-level)

```
abstract class Step
{
    protected string type;

    public Step(string type) { this.type = type; }

    public Step() {} // Parameterless option for child classes like Event

    public string Type()
    {
        return this.type;
    }

    abstract public override string ToString();
}
```

Event (Abstract, Inherits Step)

```
abstract class Event : Step
{
    public Event() { }

}
```

IfStatement (Inherits Step)

```
class IfStatement : Step
{
    private List<Step> codeBlockContents; // Every Step inside the if statement code block
    // Example condition: "x + 10 == 13"
    private List<Token> operand1; // Would be 'x + 10'
    private List<Token> operand2; // Would be '13'
    private string comparator; // Would be '=='
    // If statements only support 1 comparison for now (pattern: OPERAND COMPARATOR OPERAND)
    // an operand can however be an expression (stored as a list of Tokens)

    public IfStatement(List<Step> cbContents, List<Token> op1, List<Token> op2, string comparator)
    {
        this.type = "IF_STATEMENT";
        this.codeBlockContents = cbContents;
        this.operand1 = op1;
        this.operand2 = op2;
        this.comparator = comparator;
    }

    // No naming conflicts with built-ins so we can use Get__();
    // GETTERS
    public List<Token> GetOp1() { return operand1; }
    public List<Token> GetOp2() { return operand2; }
    public string GetComparator() { return comparator; }
    public List<Step> GetCBContents() { return codeBlockContents; }
    // END GETTERS

    public override string ToString()
    {
        List<string> operand1String = new List<string>();
        List<string> operand2String = new List<string>();
        foreach (Token tok in operand1) operand1String.Add(tok.Value()); // Just get their values into a list of strings to print
        foreach (Token tok in operand2) operand2String.Add(tok.Value());

        string codeBlockString = "";
        foreach (Step step in this.codeBlockContents) // Collect each Step in the codeblock and add to string
        {
            codeBlockString += step.ToString() + "\n";
        }
        return this.type + " CONDITION: (" + String.Join("", operand1String) + comparator // note the 'this.type' at the start instead of "IF", as this could be a while loop too
          + String.Join("", operand2String) + ")\n CONTENTS: \n" + codeBlockString;
        // Display condition operands AND each Step in codeblock
    }
}
```

WhileLoop (Inherits IfStatement)

```
class WhileLoop : IfStatement
{
    public WhileLoop(List<Step> cbContents, List<Token> op1, List<Token> op2, string comparator) : base(cbContents, op1, op2, comparator)
    {
        this.type = "WHILE_LOOP";
    }
}
```

The **base** keyword refers to the parent class constructor – this reuses the **IfStatement** constructor method.

ElseStatement (Inherits Step)

```
class ElseStatement : Step
{
    private List<Step> codeBlockContents; // Every Step inside the else statement code block

    public ElseStatement(List<Step> cbContents)
    {
        this.type = "ELSE_STATEMENT";
        this.codeBlockContents = cbContents;
    }

    public List<Step> GetCBContents() { return codeBlockContents; }

    public override string ToString()
    {

        string codeBlockString = "";
        foreach (Step step in this.codeBlockContents) // Collect each Step in the codeblock and add to string
        {
            codeBlockString += step.ToString() + "\n";
        }
        return "(ELSE) CONTENTS: \n" + codeBlockString; // Display each Step in codeblock
    }
}
```

Function Call (Inherits Event)

```
class FuncCall : Event
{
    private string funcName;
    private List<Token> arguments;
    // Arguments could be an expression or a single element referencing a variable name.

    public FuncCall(string funcName, List<Token> arguments)
    {
        this.type = "FUNC_CALL";
        this.funcName = funcName.ToLower();
        this.arguments = arguments;
    }

    public string GetName() { return funcName; }

    public List<Token> GetArguments() { return arguments; }

    public override string ToString()
    {
        List<string> argumentTokens = new List<string>();
        foreach (Token tok in arguments) argumentTokens.Add(tok.Value());

        return "FUNC_CALL: {name: '" + funcName + "', argument tokens: [" + String.Join(", ", argumentTokens) + "]}";
        // Return function name and argument (in form of expression represented by Tokens)
    }
}
```

Variable Change (Inherits Event)

```
class VarChange : Event
{
    private string varName;
    private List<Token> varValue;
    // varValue is an expression represented as a list of Tokens

    public VarChange(string varName, List<Token> varValue)
    {
        this.type = "VAR_CHANGE";
        this.varName = varName;
        this.varValue = varValue;
    }

    public string GetName() { return varName; }

    public List<Token> Value() { return varValue; }
    // Value() and not GetValue() as name conflict with built-in

    public override string ToString()
    {
        List<string> valueTokens = new List<string>();
        foreach (Token tok in varValue) valueTokens.Add(tok.Value());

        return "VAR_CHANGE: {name: '" + varName + "', value tokens: [" + String.Join(", ", valueTokens) + "]}";
        // Return variable name and value expression (as a list of Tokens)
    }
}
```

Variable Declare (Inherits Event)

```
class VarDeclare : Event
{
    private string varType;
    private string varName;
    private List<Token> varValue;

    public VarDeclare(string varType, string varName, List<Token> varValue)
    {
        this.type = "VAR_DECLARE";
        this.varType = varType;
        this.varName = varName;
        this.varValue = varValue;
    }

    public string GetName() { return varName; }
    public string GetVarType()
    {
        if (varType.Equals("int")) return "number";
        else return varType;
        // Syntax physically written as int, but we generalise to 'number' as we only support Integers.
        // Programmatically, they are treated as 'number' type. They are just called 'int' in the interpretation
        // to give a similar syntax style to C++/Java/C# for learning.
    }
    public List<Token> Value() { return varValue; } // Naming conflict if called GetValue()

    public override string ToString()
    {
        List<string> valueTokens = new List<string>();
        foreach (Token tok in varValue) valueTokens.Add(tok.Value());

        return "VAR_DECLARE: {type: '" + varType + "', name: '" + varName + "', value tokens: [" + String.Join(", ", valueTokens) + "]}";
        // Type of variable, name, value
    }
}
```

Syntax Check

We need a quick way of checking some pre-defined valid syntax. This is a static class used as a way of checking some valid keywords:

```
static class Syntax
{
    private static List<string> types = new List<string>() { "int", "string"};
    private static List<string> comparators = new List<string>() { "==", "!=", ">", "<", ">=", "<=" };

    public static bool IsType(string text)
    {
        return types.Contains(text.ToLower());
    }

    public static bool IsComparator(string tokenValue)
    {
        return comparators.Contains(tokenValue);
    }
}
```

This allows us to quickly find syntax errors in the main parsing method.

Main Parsing Method

Due to the size of this, I have broken down the important parts of the algorithm into separate screenshots. A full view of the entire file is also available.

Attributes & Constructor

```
class Parser
{
    private TokenQueue tokQueue;
    public Parser(List<Token> tokens) { tokQueue = new TokenQueue(tokens); }
```

The constructor just creates a queue based on the List of Tokens – the **input** of which is the **output** from the **Tokeniser** method (**Lexer** module).

ParseTokens Method (Overview)

```
15      public List<Step> ParseTokens() // Due to the size of this method I am not using IEnumerable 'yield return' as it is hard to track nested return statements.
16      {
17          List<Step> EvaluationSteps = new List<Step>();
18
19          while (tokQueue.More()) // While more tokens in queue (returns bool)
20          {
21              Token nextTok = tokQueue.MoveNext(); // pop next out of TokenQueue
22
23              if (nextTok.Type().Equals("identifier"))
24                  // All statements in our language begin with identifiers.
25                  // We do not know what we have at this point, so let's check the identifier to see which tokens should follow after.
26 >            {...
118             } else throw new SyntaxError(); // Statement doesn't begin with identifier - throw error.
119         }
120
121         return EvaluationSteps;
122     }
```

I have hidden the contents of the main code block to give an outline of what is returned and highlight that any statement NOT starting with an identifier throws a syntax error (custom). We are also using the **TokenQueue** class from the **Lexer** module.

- Everything past this point is **inside the Line 23 if statement** until stated otherwise.

ParseTokens – Checking for variable declaration

```
        // All statements in our language begin with identifiers.
        // We do not know what we have at this point, so let's check the identifier to see which tokens should follow after.
    {
        if (Syntax.IsType(nextTok.Value()))
        // If it is a var type, e.g "int", "string" - if it is, this is a variable declaration ("int x = 0;")
        {
            /*
             * EXPECTED PATTERN: varType varName = expr;
             * e.g int x = 2 + y*10;
             * e.g string testing = "Hello World!";
             */

            Event varDeclare = CaptureVarDeclare(nextTok.Value()); // Call method with argument storing the type of var being declared, e.g 'string'

            EvaluationSteps.Add(varDeclare); // Add Event object to the overall list of 'Steps' for the Evaluator module
        }
```

- This is an example of context-free assumption
    - We can **guarantee** that our current Token is an **identifier** due to the large if statement this is enclosed in
    - We can therefore assume if the Token value is ‘int’ or ‘string’, we are at the beginning of a variable declaration
    - We can then call a capture function, assign it to an Event

ParseTokens – Checking for if statements

```
        else if (nextTok.Value().ToLower().Equals("if"))
        // Start of an if statement
        {
            /*
             * EXPECTED PATTERN: if(operands) { codeblock }
             * e.g if (x > 0) {
             *       output(x);
             *       x = 0;
             *     }
             */

            IfStatement ifState = CaptureIfStatement(); // Capture all useful information of the following if statements

            // We COULD have an else statement, so let's check the next token
            // First check there are still MORE tokens to check to avoid out of range errors
            // Then check it's an IDENTIFIER ('else')
            if (tokQueue.More() && tokQueue.Next().Type().Equals("identifier") && tokQueue.Next().Value().Equals("else"))
            {
                // If next token is 'else' and an identifier
                ElseStatement elseState = CaptureElseStatement();
                EvaluationSteps.Add(ifState);
                EvaluationSteps.Add(elseState);
                // Add if state then else directly after (ordered list!)
            }
            else EvaluationSteps.Add(ifState); // if no 'else' statement exists just add the if statement

        }
```

- If we find an identifier with value ‘if’ we can assume we are at the start of an ‘if’ statement
- We should first call the capture function for ‘if’ statements
- It is POSSIBLE (but not guaranteed) to have an ‘else’ statement directly after
    - Check if there is **at least** one more Token in the queue to check
    - **Peek** the TokenQueue and check the next Token type is an identifier with ‘else’ value
        - If true, call Else capture function and add both the if and else to the function
        - The order of adding them is **important**
    - This is the only time an **Else** is recognised, if it is found out of context it will throw a syntax error

ParseTokens – Checking for While Loops

```
        else if (nextTok.Value().ToLower().Equals("while"))
        {
            IfStatement template = CaptureIfStatement(); // Trick the program to think it's capturing an if statement
            WhileLoop whileLoop = new WhileLoop(template.GetCBContents(), template.GetOp1(), template.GetOp2(), template.GetComparator());
            // Reuse code from the if statement because while & if follow the exact same structure:
            // while (condition) { codeblock }
            // if (condition) { codeblock }
            // We just captured an if statement 'template' then used the information it collected to create a while loop instead

            EvaluationSteps.Add(whileLoop);
        }
```

- I discovered this post-design but have gone back and specified it
- The **CaptureIfStatement()** method will be explained on one of the following pages
    - To summarise – it captures the **condition** inside the normal brackets, then all **Tokens** inside the curley brackets (codeblock), this is demonstrated by the comments
    - I realised I could reuse this to capture information about a While Loop, which is easier to think about as just a repeating 'if' statement

ParseTokens – Checking for Function Calls

```
        else if (GrammarTokenCheck(tokQueue.Next(), "("))
          // This condition will also return true if it finds an if/while statement, so it is AFTER the check for those.
          // As we're using else if, if the program didn't recognise a 'while' or 'if' statement, we will reach this check
          // We can GUARANTEE now that this must be a function call as 'if(){}' and 'while(){}' have been ruled out
        {
            /*
             * EXPECTED PATTERN: funcName(expr); // Can take any expression!
             * e.g output("Testing");
             * e.g output(1 + 23);
             * e.g output(x);
             */

            tokQueue.MoveNext(); // Skip the '(' token
            // Remember, nextTok still holds the value of the token before '('
            // This is the name of our function ('funcName')

            FuncCall funcCall = CaptureFunctionCall(nextTok.Value()); // Pass the function name, e.g 'output'
            EvaluationSteps.Add(funcCall);
        }
```

- If we reach this point in the 'else if' chain, we know that we can **rule out** it being an 'if' or 'while'
    - We must rule this out because e.g **if(x==1)** could potentially be recognised as a function call and not an 'if' statement

ParseTokens – Checking for Variable Changes

```
                    else if (GrammarTokenCheck(tokQueue.Next(), "=")) // .Next() is PEEK not POP.
                        // Check if the token AFTER this one is "="
                    {
                        /*
                         * EXPECTED PATTERN: varName = expr;
                         * e.g x = 2 + y*10;
                         * e.g testing = "Hello World!";
                         */

                        tokQueue.MoveNext(); // Skip the '=' token
                        // Remember, nextTok still holds the value of the token before the '='
                        // This is the name of our variable to change ('varName')

                        VarChange varChan = CaptureVarChange(nextTok.Value());
                        EvaluationSteps.Add(varChan);
                    }
```

- Once more we are using contextual assumptions
    - If our **first Token** (nextTok) is an identifier, and our NEXT Token is a grammar token '=' then we must be on a variable change
- Skip unneeded grammar Token and call capture function, giving the name of the variable to change

ParseTokens – Unrecognised Identifier

```
                    else throw new SyntaxError();
                    // If there is a rogue 'else' statement it will be caught in this
                    // Else statements are not 'looked' for on there own, they are only recognised when an if statement is found
```

- Remember again that 'Else' statements are **only checked for when an 'if' statement is already found**
- Any unrecognised patterns will be caught in this error, including an 'else' statement that has not been found paired with an 'if' statement

Once all of these are added to **EvaluationSteps**, an ordered list of Step objects for the **Evaluator**, it is returned.

Capture Methods – Overview

A capture method is called for each pattern recognised to extract useful information from the Tokens and create objects with that information – this is to make it easier for our Evaluator to understand what data it has available. Note that **tokQueue** is a global variable throughout the Parser class, so all capture methods are popping/peeking from the SAME queue to make sure data is not parsed twice.

Before writing the capture method, I need some other methods that contain code I will be reusing a lot:

Utility Method – GrammarTokenCheck

```
246         public bool GrammarTokenCheck(Token tok, string toCheck)
247             // It is important to check not just the value of the grammar token, but that it is a GRAMMAR token
248             // If we do not do this, a string ";" would return true as it has the value ';', but it is not actually part of the grammar in the program
249             // e.g int x = ";"; would break the program if we did not check the token ";" type and realise it's a string, not grammar.
250         {
251             return tok.Type().Equals("grammar") && tok.Value().Equals(toCheck);
252         }
```

This is a quick method to check a Token is type 'grammar' and then the value to check. It just exists to avoid writing the long Boolean condition over and over (Line 251).

Utility Method – Collecting Tokens inside brackets

```
public List<Token> CollectInsideBrackets(string openBracket, string closeBracket)
  // open & close bracket can technically be anything, but it is generally used for ( ) and { }
  // This method allows us to collect everything inside these brackets and can handle nested brackets by balancing them
  // e.g 'output(1 + (2*(1+2)));' has nested () brackets inside
  // This method, applied when the first '(' is found, will collect '1 + (2*(1+2))'
{
    int bracket_depth = 0;
    bool keepCollectingTokens = true;
    List<Token> toCollect = new List<Token>();

    while (keepCollectingTokens)
    {
        Token collectedTok = tokQueue.MoveNext();

        if (GrammarTokenCheck(collectedTok, openBracket)) bracket_depth++;
        else if (GrammarTokenCheck(collectedTok, closeBracket))
        {
            if (bracket_depth == 0)
            // Brackets already balanced, we have found the closing bracket that marks the end of the condition operands
            {
                keepCollectingTokens = false; // Finish collecting, will cause the final closing bracket to NOT be added to operands and stop the loop.
            }
            else bracket_depth--;
        }

        if (keepCollectingTokens) toCollect.Add(collectedTok); // If statement prevents the closing bracket at the end being added to the tokens as it is not part of the expression inside

    }

    return toCollect;
}
```

- Brackets can have any number of nested brackets inside them
  - E.g (1-(2+30/(10)))
  - Could also be a codeblock with nested '{' brackets
- I needed this utility to collect all the Tokens (including nested brackets) inside a defined pair of brackets to package them as a separate list
  - For example, finding the 'condition' in **if(x == 1) {}** would collect the 'x == 1' (as a list of Tokens)

Utility Method – Capturing comparator Token in condition

```
public string CollectComparator(List<Token> condition)
{
    bool found = false;
    int index = 0;
    string toReturn = "";

    while (!found && index < condition.Count)
    {
        if (condition[index].Type().Equals("grammar"))
        {
            if (Syntax.IsComparator(condition[index].Value())) // If it is e.g "==" or "!=", etc.
            {
                toReturn = condition[index].Value(); // Return which comparator it is, e.g "=="
                found = true; // stop loop
            }
        }
        index++;
    }

    return toReturn;
}
```

- A condition, in the form of a list of Tokens, will be given to this method
- It will find the comparator operator, e.g "==" and return it

Utility Method – Capturing operands in condition

We need to capture the two operands we are comparing in a condition. These could be expressions and not single Tokens, so we need a more complex splitting method:

```
        public (List<Token> Operand1, List<Token> Operand2) CaptureOperands(List<Token> condition, string comparator)
            // Split expression (below in token form) e.g:
            // Split ['2', '+', '1', '>', '1', '*', '3'] by '>'
        {
            List<Token> Operand1 = new List<Token>();
            List<Token> Operand2 = new List<Token>();
            bool passedSplitPoint = false;

            foreach (Token tok in condition)
            {
                if (GrammarTokenCheck(tok, comparator)) // Make sure grammar token and not a string with value ">" or similar
                {
                    passedSplitPoint = true;
                }
                else if (passedSplitPoint) // Passed the point to split by, add to second list
                {
                    Operand2.Add(tok);
                }
                else
                {
                    Operand1.Add(tok);
                }
            }
            return (Operand1, Operand2); // Return two halves of the list of Tokens
        }
```

We are just splitting a list by a specific Token value (our comparator). We cannot use built-in split methods as we are dealing with Token objects and not strings.

Capture Method – Capturing If Statements

```
        public IfStatement CaptureIfStatement()
        {
            List<Step> codeBlockContents = new List<Step>();

            // Next token after 'if' should be '('
            if (!GrammarTokenCheck(tokQueue.MoveNext(), "(")) throw new SyntaxError(); // Check next token while simulatneously moving along queue

            // Next token(s) should be operands to form a 'condition'
            // These token(s) will be inside ( )
            // e.g if (x > 0) {} Capture the "x > 0"
            List<Token> condition = CollectInsideBrackets("(", ")");
            // We need to separate these into OPERAND1, OPERAND2, COMPARATOR to go into the 'operands' list
            // OPERANDs can be any expression, such as 9+1*x, hence we have to collect them carefully
            // We can split the list of tokens by the comparator, but we need to find it first
            string comparator = CollectComparator(condition);
            if (comparator.Equals("")) throw new SyntaxError(); // If we didn't find a comparator we have a syntax error

            // We now have a comparator to split by
            (List<Token> Operand1, List<Token> Operand2) = CaptureOperands(condition, comparator);

            // Next token after ')' should be '{'
            if (!GrammarTokenCheck(tokQueue.MoveNext(), "{")) throw new SyntaxError(); // Check next token while simulatneously moving along queue

            // Next token(s) should all be programming statements inside the code block { }
            // Once again, we need to collect these so we can parse them
            List<Token> codeBlockTokens = CollectInsideBrackets("{", "}");

            Parser parseTokens = new Parser(codeBlockTokens); // Create new parser object with only codeblock tokens
            codeBlockContents = parseTokens.ParseTokens(); // (semi-recursion) Call parse function to parse the codeblock tokens and output a list of Step

            return new IfStatement(codeBlockContents, Operand1, Operand2, comparator);
        }
```

- Remember this code is also reused for While Loop capture

Capture Method – Capturing Else Statements

```
public ElseStatement CaptureElseStatement()
{
    List<Step> codeBlockContents = new List<Step>();
    // Called when Next() token is 'else' so we need to skip it:
    tokQueue.MoveNext();
    // We should be at the beginning of the else codeblock now
    // e.g: else { output("Hi"); }
    //           ^ we are here

    // Check next tok is the '{':
    if (!GrammarTokenCheck(tokQueue.MoveNext(), "{")) throw new SyntaxError(); // Check next token while simulatneously moving along queue

    // Next token(s) should all be programming statements inside the code block { }
    // We need to collect these so we can parse them
    List<Token> codeBlockTokens = CollectInsideBrackets("{", "}");

    Parser parseTokens = new Parser(codeBlockTokens); // Create new parser object with only codeblock tokens
    codeBlockContents = parseTokens.ParseTokens(); // (semi-recursion) Call parse function to parse the codeblock tokens and output a list of Step

    return new ElseStatement(codeBlockContents);
}
```

Capture Method – Variable Declaration

```
public VarDeclare CaptureVarDeclare(string varType)
{
    // Before creating a VarDeclare object, we should collect all the data we need while checking we actually have it in the right format.
    // We have already been given the 'type' token of the variable (it is an argument of this function)

    Token nextTok = tokQueue.MoveNext(); // Move to next token in Q

    // We expect a variable name to be next, which will be token type 'identifier'. If it is not, we have a syntax error!
    string varName; // Cannot declare a variable inside the if statement scope, declare it here

    if (nextTok.Type().Equals("identifier")) varName = nextTok.Value(); // Collect Token with name of the variable
    else throw new SyntaxError(); // Throw error if variable name is not an 'identifier' token (invalid syntax)

    // Next token after varName should be an "="
    if (!GrammarTokenCheck(tokQueue.MoveNext(), "=")) throw new SyntaxError(); // Throw syntax error if this is not found, and also move along queue index

    // We have varType and varName, now we need the tokens that form an expression to represent varValue.
    // This can be any amount of tokens, so we must collect them all
    List<Token> varValue = new List<Token>();

    while (tokQueue.More() && !GrammarTokenCheck(tokQueue.Next(), ";")) // Capture all tokens up to the end of the declaration (indicated by ";")
    {
        varValue.Add(tokQueue.MoveNext()); // Move along queue index and add each element it returns
    }


    tokQueue.MoveNext(); // Done! Now skip over the ";" - in the while loop we stopped when we 'peeked' it, but it isn't 'popped' out the queue yet.

    // We have all the data we need - we can create a VarDeclare object now

    return new VarDeclare(varType, varName, varValue);
}
```

Capture Method – Variable Change

```
public VarChange CaptureVarChange(string varName)
{
    // We have already been given the name of the variable to change
    // We need to capture the value to change it to ('varValue') e.g '3 + x +1'

    // This can be any amount of tokens, so we must collect them all
    List<Token> varValue = new List<Token>();

    while (tokQueue.More() && !GrammarTokenCheck(tokQueue.Next(), ";")) // Capture all tokens up to the end of the declaration (indicated by ";")
    {
        varValue.Add(tokQueue.MoveNext()); // Move along queue index and add each element it returns
    }


    tokQueue.MoveNext(); // Done! Now skip over the ";" - in the while loop we stopped when we 'peeked' it, but it isn't 'popped' out the queue yet.

    return new VarChange(varName, varValue);
}
```

Capture Method – Function Calls

```
public FuncCall CaptureFunctionCall(string funcName)
{
    // This is a simple capture - we just need to get all tokens inside the ( )
    // e.g output(x + 2); -> capture the "x+2"
    List<Token> arguments = CollectInsideBrackets("(", ")");

    // After collection has ended we should be at token ')' in example 'output("testing");'
    //                                                           We need to skip this token ^
    tokQueue.MoveNext(); // Skip the ";"

    return new FuncCall(funcName, arguments);
}
```

All of these are linking back to the capture methods in the Design section. They each return specific objects which all inherit from **Step**, ultimately the **ParseTokens** method adds these to the **EvaluationSteps** list.

Parser Wrap-Up

We have created capture functions for each possible pattern and a parsing algorithm to make use of them.

We can write a quick test program to take in a list of Tokens and use the .ToString() method foreach Step in the **EvaluationSteps** to check the Parser has recognised the right patterns:

```
class TestProgram
{
    public static void Run()
    {
        //-------- MULTI-LINE INPUT --------
        string input = "";
        string newInput = "";
        do
        {
            newInput = Console.ReadLine();
            input += newInput;
        } while (newInput.Length > 0);
        //-------- END OF MULTI-LINE INPUT --------

        Tokeniser tokeniser = new Tokeniser(input); // Use our Tokeniser to obtain a list of Tokens

        Parser parse = new Parser(tokeniser.Tokenise().ToList()); // Init parser with Tokens
        foreach (Step stepObj in parse.ParseTokens())
        {
            Console.WriteLine(stepObj.ToString()); // Output each Step in EvaluationSteps
        }
    }
}
```

We can then do a quick ‘dummy’ program to input into the multi-line prompt (I will do more extensive and wide-ranging tests later):

```
int xNumber = 20*10+(4/2);
xNumber = 201;

string hiWorld = "Hello World!";

outputln(hiWorld);

if (xNumber > 20*10) {
        outputln("Bigger than 200.");
} else {
        outputln("Smaller than 200.");
}
```

And the EvaluationSteps built by the Parser shows:

```
VAR_DECLARE: {type: 'int', name: 'xNumber', value tokens: [20, *, 10, +, (, 4, /, 2, )]}

VAR_CHANGE: {name: 'xNumber', value tokens: [201]}

VAR_DECLARE: {type: 'string', name: 'hiWorld', value tokens: [Hello World!]}

FUNC_CALL: {name: 'outputln', argument tokens: [hiWorld]}

(IF) CONDITION: (xNumber>20*10)
 CONTENTS:
    FUNC_CALL: {name: 'outputln', argument tokens: [Bigger than 200.]}

(ELSE) CONTENTS:
    FUNC_CALL: {name: 'outputln', argument tokens: [Smaller than 200.]}
```

```
public override string ToString()
{
    List<string> operand1String = new List<string>();
    List<string> operand2String = new List<string>();
    foreach (Token tok in operand1) operand1String.Add(tok.Value()); // Just get their values into a list of strings to print
    foreach (Token tok in operand2) operand2String.Add(tok.Value());

    string codeBlockString = "";
    foreach (Step step in this.codeBlockContents) // Collect each Step in the codeblock and add to string
    {
        codeBlockString += step.ToString() + "\n";
    }
    return "(IF) CONDITION: (" + String.Join("", operand1String) + comparator
      + String.Join("", operand2String) + ")\n CONTENTS: \n" + codeBlockString;
    // Display condition operands AND each Step in codeblock
}
```

This is the **ToString()** method from **IfStatement.cs**.

These outputs show the Parser has correctly captured all required information from the program – the test has completed **Core Objective 2 – Parser Module** (more testing to be done later).

Evaluator Module (Core & Related)

Resolving Expressions

As most programming statements in our language make use of **expressions**, we need to create a method of resolving these **expressions** into more understandable ways – such as changing the expression 2 + 1 + 2 to a single Integer result, 5. Expressions can also consist of strings, such as "Hello" + " World!" – we need to be able to distinguish between these types of expressions and make sure we do not have an inconsistency in the types used, for example "Hi" + 1 should throw an error.

Computing Mathematical Expressions

We need to define some utility classes and methods first.

Trees

```
abstract class TreeNode
{
    // We are not going to regulate access to these as there are no conditions for them that we'd even be able to check
    // The type requirement is the only level of regulation we need for these attributes
    public TreeNode left = null;
    public TreeNode right = null;
    public string value = null;

    public TreeNode() { } // parameterless base for ease of use if you do not want to create a node with a left and right value

    public TreeNode(TreeNode inputLeft, TreeNode inputRight) // Simply exists for ease of use when creating a new TreeNode
    {
        this.left = inputLeft; this.right = inputRight;
    }
}
```

Inheriting from TreeNode:

note: 'Operator' was originally called 'BinOp' (Binary Operator) but I've renamed it. It might be referenced in comments as 'BinOp'.

```
class Operator : TreeNode
{
    public static Dictionary<string, int> precedences = new Dictionary<string, int>()
    {
        {"_", 5 }, // unary minus has a higher precedence than all below
        {"^", 4 },
        {"*", 3 },
        {"/", 3 },
        {"+", 2 },
        {"-", 2 },
        {")", 2 },
        {"(", 1 }
    }; // Static dictionary of precedence levels represented by ints for ease of comparison later on - used in TreeBuilder.cs

    public Operator(string operationValue)
    {
        this.value = operationValue;
    }

    public Operator() // parameterless option
    {

    }
}
```

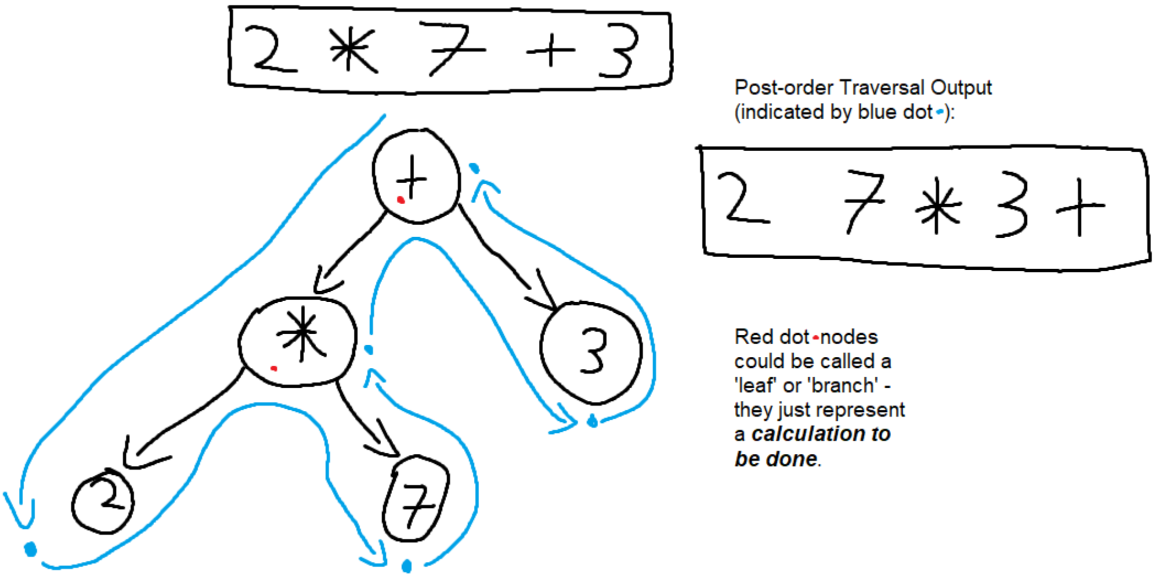
```
class Num : TreeNode
{
    private int intValue; // this.value still exists, but is in string form - it is useful to have a string value and int value instead of problematic ToInt conversions later

    public Num(int inputValue)
    {
        this.intValue = inputValue;
        this.value = inputValue.ToString(); // Both values equal to the same integer, just different types.
    }

    public int IntValue() { return intValue; }
}
```

Recursive Traversal Algorithms

```
class Traversal
{
    /*
     * All three of these are recursive - they create their own lists which are returned and added onto the list of the parent.
     * The only one *actually* used in our program is postOrder for the postfix expressions, but the other two are
     * useful for testing the Abstract Syntax Trees are correctly being made.
     *
     * Using inOrder traversal on an AST will output the expression in the original, human-readable mathematical form.
     */
    public static List<TreeNode> postOrder(TreeNode node)
    {
        List<TreeNode> nodes = new List<TreeNode>();
        if (node.left != null) nodes.AddRange(postOrder(node.left)); // Add recursive call onto the end (via AddRange) of the list
        if (node.right != null) nodes.AddRange(postOrder(node.right));
        nodes.Add(node); // As it's post order, add the parent node to the end.
        return nodes;
    }

    public static List<TreeNode> inOrder(TreeNode node)
    {
        List<TreeNode> nodes = new List<TreeNode>();
        if (node.left != null) nodes.AddRange(inOrder(node.left));
        nodes.Add(node); // In order so add parent node in between.
        if (node.right != null) nodes.AddRange(inOrder(node.right));
        return nodes;
    }

    public static List<TreeNode> preOrder(TreeNode node)
    {
        List<TreeNode> nodes = new List<TreeNode>();
        nodes.Add(node); // Pre order so add parent node first.
        if (node.left != null) nodes.AddRange(preOrder(node.left));
        if (node.right != null) nodes.AddRange(preOrder(node.right));
        return nodes;
    }
}
```

Prototype & Design Reminder

This is almost a replica of the prototype program expression evaluator. We have two classes inheriting from TreeNode – Operator (mathematical operations to be done or brackets) and Num (Number). These are the two types of nodes you can find in an abstract syntax tree:



Now that we have the tools to build a tree with, we need to write the algorithm to build one from a mathematical expression. This uses a modified version of Djikstra's Shunting-yard algorithm[8].

When I refer to 'leaf' in the comments on the following Shunting-yard algorithm code, I mean any TreeNode with two children that will altogether result in a calculation, e.g the red dot nodes in the diagram.

[8] https://en.wikipedia.org/wiki/Shunting-yard_algorithm [28/01/2021]

Shunting-yard Algorithm

Utility Method: Find Unary Minus

This method finds occurrences of unary minus signs and replaces them with the '_' (underscore) character.

```
private static List<Token> FindUnaryMinus(List<Token> exprTokens)
{
    List<Token> toReturn = new List<Token>();

    int index = 0;

    while (index < exprTokens.Count)
    {
        Token tok = exprTokens[index];

        if (tok.Type().Equals("operator") && tok.Value().Equals("-"))
        {
            // Unary minus rules (OR for each, not all 3 required):
            /* - First token in expression
             * - Right after '('
             * - Right after any operator
             */
            if (index == 0 || exprTokens[index - 1].Value().Equals("(") || exprTokens[index - 1].Type().Equals("operator"))
            {
                // found unary minus
                tok = new Token("operator", "_");
                // change token to unary minus: '_'
            }
        }

        toReturn.Add(tok);

        index++;
    }

    return toReturn;
}
```

Main AST Building Method

This is the Djikstra Shunting-yard Algorithm implementation with the modifications I've added for unary support. **FindUnaryMinus()** is used to change the unary '-' to '_'.

```
class TreeBuilder
{
    public static TreeNode BuildAST(List<Token> infix)
    /*
     * It is easier to think of this as a reversal of an RPN calculation algorithm using stacks.
     *
     * This is an implementation of Djikstra's Shunting-yard algorithm
     * It is not 100% true to the original; instead of resolving each expression in the stack it builds a tree
     * We use two stacks: one for operators (e.g +, -, /) and one for the operands (integer)
     * The numstack is not actually storing Integers, but more TreeNodes that represent Integers
     *       The numstack TreeNodes could just be of value '1' or could ALSO be a 'leaf' of the tree
     *       A 'leaf' is a small calculation that represents an Integer to be calculated
     *       An example leaf could be:
     *                   +
     *                 1   2
     *       Programatically, this will just be given as the '+' root node with .Left and .Right being nodes '1' & '2'
     *
     *
     *
     * INPUT: List of Token objects that represent a mathematical expression, e.g ['1', '+', '2']
     * (just showing Token.Value() as list elements for demo)
     *
     * OUTPUT: The ROOT node of the Abstract Syntax Tree (AST) as a TreeNode object.
     * The parents of any nodes in this AST could be a Num or BinOp (both inherit from TreeNode)
     * Num: Represents just an Integer itself
     * BinOp: Represents an Operator. Left & Right nodes of a BinOp are to be the operands.
     */
    {

        Stack<Operator> operatorStack = new Stack<Operator>();
        Stack<TreeNode> numStack = new Stack<TreeNode>();

        foreach (Token token in FindUnaryMinus(infix)) // Iterate over infix with unary minus signs found and changed to "_" from "-"
        {
            if (token.Value().Equals("(")) operatorStack.Push(new Operator("("));
            // If it is the opening of a nested expression, just add it to the opstack - precedences values will be dealt with later


            else if (token.Type().Equals("number")) // If it is a number (Integers only supported), add to numStack
            {
                numStack.Push(new Num(int.Parse(token.Value())));
                // Simply create a new Num (child class of TreeNode) node with the number in the character, converted to Integer type.
            }


            else if (Operator.precedences.ContainsKey(token.Value()) && !token.Value().Equals(")"))
                // BinOp.precedences is a DICTIONARY of all operators & their precedence (represented in Integers)
                // If token.Value() IS an OPERATOR and is NOT ")"
            {
                // We have found an operator like +
                // We need to resolve the operands into Leaves first
                while (operatorStack.Count > 0 && Operator.precedences[operatorStack.Peek().value] >= Operator.precedences[token.Value()])
                    // While the precedence of the top of the operatorStack is bigger than or equal to the precedence of the char
                    // Remember that precedences are stored as Integers, so we can compare them easily like this
                {
                    Operator binOperator = operatorStack.Pop();
                    // This will be the parent node of our 'leaf'
                    // the '+' in the example in the topmost comment

                    if (binOperator.value.Equals("_"))
                    {
                        // UNARY MINUS, only pop 1 operand
                        binOperator.left = numStack.Pop();
                    }
```

(cont. Next Page, same indent level)

```
                    else
                    {
                        // Reversed as the second op comes out first
                        binOperator.right = numStack.Pop();
                        binOperator.left = numStack.Pop();
                        // child nodes '1' and '2' added (following example in top comment)
                        // The numstack does not just contain raw Integer nodes, it could have another leaf (a leaf resolves to an Integer)
                    }

                    numStack.Push(binOperator); // Leaf created! Now push parent node of leaf back onto numStack
                }
                // Now that our loop has iteratively created leaves and connected them for our tree, we have finished

                // Now push operator at the end - we have not calculated anything with this one yet
                operatorStack.Push(new Operator(token.Value()));
            }

            else if (token.Value().Equals(")")) // End of nested () expression
            {
                while (operatorStack.Count > 0 && !operatorStack.Peek().value.Equals("("))
                {
                    Operator binOperator = operatorStack.Pop();

                    if (binOperator.value.Equals("_"))
                    {
                        // UNARY MINUS, only pop 1 operand
                        binOperator.left = numStack.Pop();
                    }
                    else
                    {
                        // Reversed as the second op comes out first
                        binOperator.right = numStack.Pop();
                        binOperator.left = numStack.Pop();
                        // child nodes '1' and '2' added (following example in top comment)
                        // The numstack does not just contain raw Integer nodes, it could have another leaf (a leaf resolves to an Integer)
                    }

                    numStack.Push(binOperator);
                }
                // Similar loop to previously used - this time to resolve everything that we have collected inside the brackets.
                // the minute we run into another nest, '(', we can leave it for later.

                operatorStack.Pop(); // We still have the '(' operator that started this expr in brackets left, let's get rid of it.
            }

            else
            {
                throw new SyntaxError(); // Don't recognise what kind of Token is in our expression.
            }
        }

        while (operatorStack.Count > 0) // Same leaf-making loop as before but with slightly different condition
        {
            Operator binOperator = operatorStack.Pop();

            if (binOperator.value.Equals("_"))
            {
                // UNARY MINUS, only pop 1 operand
                binOperator.left = numStack.Pop();
            }
            else
            {
                // Reversed as the second op comes out first
                binOperator.right = numStack.Pop();
                binOperator.left = numStack.Pop();
                // child nodes '1' and '2' added (following example in top comment)
                // The numstack does not just contain raw Integer nodes, it could have another leaf (a leaf resolves to an Integer)
            }

            numStack.Push(binOperator);
        } // While there are still operators left, make leaves of the remaining with their operands until no more to make

        // At this point, the root node should be left at the top of the numStack (and the only thing in it)
        return numStack.Pop();
        // we have NOT calculated anything - just built a tree of the expression and returned its root as a TreeNode.
    }
```

Reverse Polish Notation Algorithm

```
class RPN
{
    public static int Evaluate(List<TreeNode> nodes) // Input is a list of TreeNodes given in POSTFIX traversal order of the tree
    {
        Stack<TreeNode> nodeStack = new Stack<TreeNode>(); // Create stack to use for RPN, type TreeNode as it could be a BinOp or Num

        foreach (TreeNode treeNode in nodes)
        {
            if (treeNode is Num) nodeStack.Push(treeNode); // If it is a Num (Integer) then just push onto stack
            else if (treeNode is Operator) // If it is a BinOp (operator, root of leaf) then pop last two operands and calculate:
            {
                if (treeNode.value.Equals("_")) // unary minus only pops one operand
                {
                    Num arg1 = (Num)nodeStack.Pop();
                    nodeStack.Push(new Num(Calculate(arg1.IntValue(), -1, "*")));
                    // multiply by -1 to negate the operand
                }
                else
                {
                    // Pop last two
                    Num arg2 = (Num)nodeStack.Pop();
                    Num arg1 = (Num)nodeStack.Pop(); // Note they are reversed, the first one to be popped is the second argument in the expression.

                    nodeStack.Push(new Num(Calculate(arg1.IntValue(), arg2.IntValue(), treeNode.value))); // Create new Num (Integer) with result of calc
                }
            }
        }
        Num result = (Num)nodeStack.Pop(); // Stack should be left with just one Num (Integer) as the final result
        return result.IntValue();
    }

    public static int Calculate(int arg1, int arg2, string operation) // Simplest way to 'act out' operators that are in string-form
    {
        int result = 0; // Default is 0
        switch (operation)
        {
            case "+":
                result = arg1 + arg2;
                break;
            case "-":
                result = arg1 - arg2;
                break;
            case "*":
                result = arg1 * arg2;
                break;
            case "/":
                result = arg1 / arg2;
                break;
            case "^":
                result = (int)Math.Pow(arg1, arg2);
                break;
            default:
                break; // do nothing as result = 0 already
        }
        return result;
    }
}
```

- There is no need to add a specific '_' case in the **Calculate** function, as the unary minus just negates the operand (so we can multiply by '-1').

To test this I have written a program that shows all traversals of the tree (making use of **in order** and **pre order** traversal) to show it is being built correctly, then output the result of the RPN calculation:

```
class TestProgram
{
    public static void Run()
    {
        Tokeniser tokeniser = new Tokeniser(Console.ReadLine());
        List<Token> tokens = tokeniser.Tokenise().ToList();

        TreeNode bin1 = TreeBuilder.BuildAST(tokens); // e.g "5 * 2 + 1" -> "5 2 * 1 +"
        // Using TreeNode type, not BinOp (Binary Operator) as we cannot guarantee the root node of the abstract syntax tree will be an operator.

        Console.WriteLine("To postfix:");
        foreach (TreeNode node in Traversal.postOrder(bin1))
        {
            Console.Write(node.value + " ");
        }
        Console.WriteLine("\nTo infix:");
        foreach (TreeNode node in Traversal.inOrder(bin1))
        {
            Console.Write(node.value + " ");
        }
        Console.WriteLine("\nTo prefix:");
        foreach (TreeNode node in Traversal.preOrder(bin1))
        {
            Console.Write(node.value + " ");
        }
        Console.WriteLine();

        // Now using reverse polish notation, calculate what the result is. This takes in a postfix-ordered list of TreeNodes.
        Console.WriteLine("Answer: " + RPN.Evaluate(Traversal.postOrder(bin1)));

    }
}
```

Example input:

```
2 + 10*(50/2)
```

Given output:

```
To postfix:
2 10 50 2 / * +
To infix:
2 + 10 * 50 / 2
To prefix:
+ 2 * 10 / 50 2
Answer: 252
```

More extensive testing will be done on this in the **Testing** section, including unary minus tests.

Computing String Expressions

We can do this inside the main resolving function.

ResolveExpression Method

This method will take input of a list of Tokens (representing an expression) and output a single Token result. It may use the mathematical evaluation tools or may compute a string result.

```
        public Token ResolveExpression(List<Token> expr) // TODO
        {
            Token toReturn = new Token("", "");

            // First, replace variable name references with their values.
            expr = VariablesToValues(expr);

            // Now check tokens are all the same type in the expression (except grammar tokens)

            string exprResultType = CheckTypes(expr); // This func will throw error if they aren't
            // exprResultType now stores the final expected type for when expression is resolved to one token
            // e.g 1 + 1 => resolves to 'number'
            // e.g "1" + "1" => resolves to 'string'

            if (exprResultType.Equals("string"))
                // Indicates that we are dealing with a string expression
            {
                // The only operation that can be done to strings in an expression is '+' for concat
                if (expr.Count == 1) toReturn = new Token("string", expr[0].Value());
                // If there is only one token in the whole expression, it must just be a string
                // Therefore we can just return the string as it's 1 token
                else
                {
                    // We must be dealing with concatenation
                    if (!expr[0].Type().Equals("string")) throw new SyntaxError();
                    // Concatenation expressions MUST start with a string
                    // e.g string x = + "Hello World"; will cause ERROR as expr starts with '+'

                    string finalResult = expr[0].Value(); // First string in expression
                    int index = 1;

                    while (index < expr.Count)
                    {
                        if (expr[index].Type().Equals("operator"))
                        {
                            if (expr[index].Value().Equals("+") && index < expr.Count - 1)
                            {
                                finalResult += expr[index + 1].Value(); // Add NEXT string to final result
                            }
                            else throw new TypeMatchError(); // Cannot do any other operation than '+' on strings
                        }

                        index++;
                    }
                    toReturn = new Token("string", finalResult);
                }
            }
            else if (exprResultType.Equals("number"))
                // Indicates we are dealing with a mathematical expression
            {
                TreeNode root = TreeBuilder.BuildAST(expr); // Create abstract syntax tree of mathematical expression

                int result = RPN.Evaluate(Traversal.postOrder(root)); // Calculate result of RPN algorithm calculation

                toReturn = new Token("number", result.ToString());
            }
            else throw new SyntaxError(); // invalid expression type has somehow made it through, we cannot evaluate it so throw error.

            return toReturn;
        }
```

- If all elements of the expression are strings, it will check to make sure the expression only uses “+” operators (as that is all you can do to strings) and then concatenate them.
- If all the elements are numbers (integers), then it will use the mathematical evaluation tools to compute a result.
- Any result found is represented by a **single Token** object. A blank Token is returned by default.

To resolve expressions with variables in them, we need to have a method that replaces all variables in the expression with their raw values:

```
public List<Token> VariablesToValues(List<Token> expr)
{
    // Replace each variable in an expression with the value it references

    List<Token> newExpr = new List<Token>();
    Token toAdd;
    foreach (Token tok in expr)
    {
        if (tok.Type().Equals("identifier"))
        // Must be variable reference
        {
            if (variableScope.ContainsKey(tok.Value()))
            // If that variable exists
            {
                toAdd = variableScope[tok.Value()];
                // Add referenced variable's VALUE token to expression instead of the actual reference to the variable
            }
            else throw new ReferenceError(); // Referencing non-existent variable, throw error
        }
        else toAdd = tok;

        newExpr.Add(toAdd);
    }

    return newExpr;
}
```

This directly references the dictionary we are using to store variables.

We also need a quick checking method to make sure the types in an expression match up:

```
public string CheckTypes(List<Token> expr)
{
    // Check all types in an expression can work together
    // e.g 1 + 1 will work
    // e.g "1" + "1" will work
    // e.g "1" + 1 will cause an ERROR.
    bool foundNumber = false;
    bool foundString = false;

    foreach (Token tok in expr)
    {
        if (!tok.Type().Equals("grammar"))
        {
            if (tok.Type().Equals("number")) foundNumber = true;
            else if (tok.Type().Equals("string")) foundString = true;
        }
    }

    if (foundString == foundNumber) throw new TypeMatchError(); // Found either both or neither "string" and "number" types in same expression, causes error.
    else if (foundNumber) return "number";
    else return "string"; // Must be a string if not foundNumber and NOT(foundString == foundNumber == false).
}
```

Types of variables are gotten by getting the type of **Token** that is stored in the dictionary. For

```
string hello = "Hi";
```

example, if a string variable is declared:

```
{'hello': Token('string', "Hi")}
```

It will have the corresponding dictionary storage, with **Token type 'string'**:

Integer variables will have **Token type 'number'**.

Storing Variables & Constructor

```
private Dictionary<string, Token> variableScope;

public Evaluator()
{
    this.variableScope = new Dictionary<string, Token>();
}
```

- variableScope is accessible by any method in the evaluator
- Recursive evaluation of code blocks can take place without the need to create a new Evaluator object, allowing for the recursion to still be changing the same central variableScope object
- I will explain more about recursive evaluation in the main evaluator method

Utility Methods for Evaluator

These are just small pieces of code that are used a lot and therefore have been made as methods:

```
public bool TokenEqual(Token tok1, Token tok2) { return tok1.Type().Equals(tok2.Type()) && tok1.Value().Equals(tok2.Value()); }
// Checks both type and value of a token are equal
```

For checking conditions, we have a **CompareExpressions** method:

```
public bool CompareExpressions(List<Token> op1, List<Token> op2, string comparison)
{
    bool toReturn;
    Token resolvedOp1 = ResolveExpression(op1);
    Token resolvedOp2 = ResolveExpression(op2);

    if (resolvedOp1.Type().Equals("string") && resolvedOp2.Type().Equals("string"))
    {
        if (comparison.Equals("==")) toReturn = TokenEqual(resolvedOp1, resolvedOp2);
        else if (comparison.Equals("!=")) toReturn = !TokenEqual(resolvedOp1, resolvedOp2);
        else throw new ComparisonError(); // Cannot do any other comparison on strings.

    } else { // Any comparison that isn't == or != can ONLY be done on numbers
        int op1Integer; // We need to conver the number Tokens to actual Integers for comparison
        int op2Integer;
        if (resolvedOp1.Type().Equals("number") && resolvedOp2.Type().Equals("number"))
        {
            // Value is stored as a string, hence the conversion to integer.
            op1Integer = int.Parse(resolvedOp1.Value());
            op2Integer = int.Parse(resolvedOp2.Value());
        } else throw new ComparisonError(); // Invalid types to do these comparisons on

        switch (comparison)
        {
            case "==": // TokenEqual just checks type & value of Tokens are equal. No need to convert to C# Integer type.
                toReturn = TokenEqual(resolvedOp1, resolvedOp2);
                break;
            case "!=":
                toReturn = !TokenEqual(resolvedOp1, resolvedOp2);
                break;
            case "<=":
                // Check if equal or if less than.
                // Note TokenEqual takes in Tokens, OR we can compare their raw Integer values
                toReturn = TokenEqual(resolvedOp1, resolvedOp2) || (op1Integer < op2Integer);
                break;
            case ">=":
                // Check if equal or greater than.
                toReturn = TokenEqual(resolvedOp1, resolvedOp2) || (op1Integer > op2Integer);
                break;
            case "<":
                toReturn = (op1Integer < op2Integer);
                break;
            case ">":
                toReturn = (op1Integer > op2Integer);
                break;
            default:
                toReturn = false;
                break;
        }
    }

    return toReturn;
}
```

- This checks that the types of the two operands to compare are compatible (e.g string string or number number, nothing else allowed)
- TokenEqual, as mentioned above, checks the value and type are equal
    - This is used so that we do not accidentally resolve “1” == 1 as TRUE
- Only specific comparisons can be done for strings; == and !=

The other utility methods are the **VariablesToValues** & **CheckTypes**, which have already been mentioned.

Main Evaluator Method

```
        public void Evaluate(List<Step> evaluationSteps)
        {
            for (int index = 0; index < evaluationSteps.Count; index++)
            {
                Step evalStep = evaluationSteps[index];
                // .Type() can only be "VAR_DECLARE", "VAR_CHANGE", "FUNC_CALL", "IF_STATEMENT", "WHILE_LOOP"
                // It could also be "ELSE_STATEMENT", but we should only check for that DIRECTLY after an IF_STATEMENT
                if (evalStep.Type().Equals("IF_STATEMENT"))
                {…
                }
                else if (evalStep.Type().Equals("WHILE_LOOP"))
                {…

                }
                else if (evalStep.Type().Equals("VAR_DECLARE"))
                // Declare a variable in the variableScope
                {…
                }
                else if (evalStep.Type().Equals("VAR_CHANGE"))
                // Change a pre-existing variable
                {…

                }
                else if (evalStep.Type().Equals("FUNC_CALL"))
                // Call a function
                {…
                }
                else throw new SyntaxError(); // Unrecognised Step, crash program.
            }
        }
```

Evaluating If Statements

```
if (evalStep.Type().Equals("IF_STATEMENT"))
{
    // Evaluate if statement - contains OPERAND1, OPERAND2, COMPARISON, codeBlockContents
    IfStatement ifState = (IfStatement)evalStep; // Cast as we know it is now an IfStatement obj
    bool conditionResult = CompareExpressions(ifState.GetOp1(), ifState.GetOp2(), ifState.GetComparator());
    bool hasElse = index + 1 < evaluationSteps.Count && evaluationSteps[index + 1].Type().Equals("ELSE_STATEMENT"); // No chance of index out of range error as set to False before reaching it

    if (conditionResult)
    // If the 'IfStatement' condition is TRUE
    {
        Evaluate(ifState.GetCBContents()); // 'run' the contents of the if statement - this is RECURSIVE
        if (hasElse) evaluationSteps.RemoveAt(index + 1);
        // If we have an ELSE_STATEMENT after this, we need to remove it as the IF_STATEMENT has triggered (therefore the ELSE will not be triggered).
    }
    else if (hasElse)
    {
        // If the CONDITION is FALSE and the next Step obj is an ELSE_STATEMENT type

        ElseStatement elseState = (ElseStatement)evaluationSteps[index+1];
        // Cast to else
        Evaluate(elseState.GetCBContents()); // 'run' the contents of the else (RECURSION)

        evaluationSteps.RemoveAt(index + 1); // Remove ELSE_STATEMENT as we have used it and do not want to go over it again.
    }
}
```

Evaluating While Loops

```
else if (evalStep.Type().Equals("WHILE_LOOP"))
{
    WhileLoop whileLoop = (WhileLoop)evalStep;
    // Similar to if statement evaluation though no need to set a 'condition' variable because that condition may change
    // Basically just reusing the C# while loop with the template of the Interpreted one
    while (CompareExpressions(whileLoop.GetOp1(), whileLoop.GetOp2(), whileLoop.GetComparator()))
    {
        // While the condition is true, evaluate code inside
        Evaluate(whileLoop.GetCBContents());
    }

}
```

Evaluating Variable Declarations

```
else if (evalStep.Type().Equals("VAR_DECLARE"))
// Declare a variable in the variableScope
{
    VarDeclare varDecl = (VarDeclare)evalStep; // Cast as we know it's a VarDeclare obj
    if (variableScope.ContainsKey(varDecl.GetName())) throw new DeclareError();
    // If scope already has a variable that name, you cannot redeclare it as it already exists.
    // Potential endpoint if variable exists - entire program will stop (crash).
    Token varExpr = ResolveExpression(varDecl.Value());

    if (!varExpr.Type().Equals(varDecl.GetVarType())) throw new TypeError();
    // Value of variable does not match type with declared one. e.g 'int x = "Hello";'


    variableScope.Add(varDecl.GetName(), varExpr);
    // Type of variable can be found out by the .Type() of the key's Token.
    // e.g 'int x = 1 + 2;'
    // if we want to find variable 'x' type, we find variableScope[x].Type() which will return 'number', with variableScope[x].Value() being '3'
}
```

Evaluating Variable Changes

```
else if (evalStep.Type().Equals("VAR_CHANGE"))
// Change a pre-existing variable
{
    VarChange varChan = (VarChange)evalStep; // Cast as we know it is a VarChange obj

    if (!variableScope.ContainsKey(varChan.GetName())) throw new ReferenceError();
    // If variable is NOT in the variableScope then we cannot change it as it doesn't exist.
    // Potential endpoint for program crash
    string varType = variableScope[varChan.GetName()].Type();
    Token newValue = ResolveExpression(varChan.Value());

    if (!varType.Equals(newValue.Type())) throw new TypeError();
    // If the new value of the variable is not the right type, then crash.
    // Potential endpoint
    // e.g int x = 0; x = "hi"; will cause this error
    variableScope[varChan.GetName()] = newValue; // Assign new value (Token)

}
```

Evaluating Function Calls

```
else if (evalStep.Type().Equals("FUNC_CALL"))
// Call a function
{
    FuncCall functionCall = (FuncCall)evalStep; // Cast as we know it is a FuncCall obj now
    if (!functionCall.GetName().Equals("inputstr") && !functionCall.GetName().Equals("inputint")) // If NOT calling 'input' function
    {
        CallFunction(functionCall.GetName(), ResolveExpression(functionCall.GetArguments()));
        // Call function with name and *resolved* list of arguments
        // Resolve function always outputs a single token which is the result of an expression (list of tokens) being evaluated
    } else
        // SPECIAL CASE: Calling inputStr or inputInt functions indicates that the 'argument' is NOT an expression to be resolved, but rather a variable name to store input value in.
        // This means functionCall.Argumnets() will only have 1 token:
    {
        CallFunction(functionCall.GetName(), functionCall.GetArguments()[0]); // Pass in first value in Arguments as there only should be one - the variable to be input to
    }
}
```

This makes use of the **CallFunction** method – I made this solely to separate out the hard-coded actions for each function into a different place, as not to make the Evaluate method too long from it:

```
        public void CallFunction(string funcName, Token argument)
            // As this is a simple language, functions can only take one argument.
        {
            funcName = funcName.ToLower(); // We do not need to be case sensitive for our language.
            if (funcName.Equals("output")) Console.Write(argument.Value()); // Write to console with no new line
            else if (funcName.Equals("outputln")) Console.WriteLine(argument.Value()); // Write with new line
            else if (funcName.Equals("inputstr"))
            {
                string varName = argument.Value(); // Argument passed into input() function is a VARIABLE NAME REFERENCE

                if (!variableScope.ContainsKey(varName)) throw new ReferenceError();
                // If variable to input to doesn't exist there is an error

                if (!variableScope[varName].Type().Equals("string")) throw new TypeError();
                // inputStr() being done to a non-string variable causes an error

                Console.Write("> "); // Automatic input prompt
                string input = Console.ReadLine();

                variableScope[varName] = new Token("string", input); // Change value of variable to the input string
            }
            else if (funcName.Equals("inputint"))
            {
                string varName = argument.Value();

                if (!variableScope.ContainsKey(varName)) throw new ReferenceError();
                // If variable to input to doesn't exist there is an error

                if (!variableScope[varName].Type().Equals("number")) throw new TypeError();
                // inputInt() being done to a non-number variable causes an error

                Console.Write("> "); // Automatic input prompt
                int input;
                try
                {
                    input = int.Parse(Console.ReadLine());
                }
                catch
                {
                    // if input is not a number, cause error
                    throw new TypeError();
                }
                // It looks weird to catch and then throw an error anyway, but I've done it so I can use my custom TypeError() instead of C#'s one
                // My TypeError() will come up with a simple message and pause, the C# in-built error will kill the console window instead.

                variableScope[varName] = new Token("number", input.ToString());
                // The input is converted to an integer originally to check it is ACTUALLY a valid integer
                // Once we know that, we can store it as a 'number' token with string value and convert it without errors later
            }
            else throw new ReferenceError(); // No function name recognised, throw error.
        }
    }
```

As we only have a few built-in functions and no way to declare one in-language, we just must hardcode their actions.

Functions are also **not** case-sensitive.

Evaluator Wrap-Up

The Evaluation tools and the Evaluator itself have been written. Instead of writing a test program solely for the Evaluator, we can just link all three modules together now and use that to test instead.

By writing each of these Evaluator actions, I am counting the **Objective 5 – Syntax & Program** as complete, due to the syntax reaching the final stage of recognition; obvious syntax errors are caught in the **Lexer**, smaller syntax or formatting errors are caught in the **Parser**, and structural / logical syntax errors are caught in the **Evaluator** (refer to Testing section ahead).

Writing the Evaluator and seeing the final Interpreter run with all three modules linked together (below) will complete **Core Objective 3 – Evaluator Module**.

Linking Modules Together – Finish

To have a working Interpreter, the modules need to be linked together. We also need a way of inputting a file to run, and the prompts to say when the program starts and ends:

```
class Program
{
    static void Main(string[] args)
    {

        // Test programs to run in case of problems:
          // Parser_Module.TestProgram.Run();
          // Lexer_Module.TestProgram.Run();
          // Evaluator_Module.ExpressionEvaluation.TestProgram.Run();
          // Evaluator_Module.ExpressionEvaluation.TestProgram.Run();

        // FILE INPUT
        bool invalid = true;
        string toRun = "";
        while (invalid) {
            try
            {
                Console.Write("Enter a valid file name to run: ");

                toRun = System.IO.File.ReadAllText(Console.ReadLine());
                invalid = false;
            } catch
            {
                Console.WriteLine("Invalid file name.");
            }
        }
        // END OF FILE INPUT

        Console.WriteLine("-------------------- PROGRAM STARTED --------------------");

        Tokeniser tokeniser = new Tokeniser(toRun); // TOKENISE FILE CONTENTS
        List<Token> tokens = tokeniser.Tokenise().ToList(); // ToList() will put the output of the IEnumerable straight into 'tokens'

        Parser parseTok = new Parser(tokens); // PARSE TOKENISED PROGRAM
        List<Step> evalSteps = parseTok.ParseTokens(); // CREATE EvaluationSteps for EVALUATOR_MODULE

        Evaluator eval = new Evaluator(); // Init Evaluator
        eval.Evaluate(evalSteps); // EVALUATE EvaluationSteps

        Console.WriteLine("-------------------- PROGRAM ENDED --------------------");

    }
}
```

This is the entry point for the whole program, making use of all three modules. ***I have added a Console.ReadLine(); statement at the end so that the program does not instantly exit after running.**

Overall Technical Completeness (including Testing stage results)

Objective	Related Section(s)	Test Evidence	Complete?
1. Functioning Lexer Module *“**The ‘Lexer’ should be able to isolate variable names, expressions, and operators, along with** other identifiers like ‘if’. **It should be able to produce a list of tokens in the order of the program, with context for each token.**”*	**Design**: Lexer (p.24-29) **Technical Solution**: Lexer Module (p.45-48) **Testing:** **Source Code Folders:** `Lexer_Module`, `DataStructures`	**Technical Solution** (p.48) **Testing:** (Video: 1:19)	✓
2. Functioning Parser Module *“**This parser should be able to take input of the list of tokens from the ‘Lexer’ and build ‘program steps’ based off them.** These represent individual program statements.”*	**Design**: Parser (p.30-36) **Technical Solution**: Parser Module (p.49-61) **Testing:** **Source Code Folders:** `Parser_Module`	**Technical Solution** (p.61) **Testing:** (Video: 9:09)	✓
3. Functioning Evaluator Module *“**There should be an executing module which takes input from the parser** output* **and goes through each instruction in order. It needs to determine the directions and dat a of these instructions and execute them, making use of direct console engagement. After this has finished, the** execution **of the interpreted program will end.**”*	**Design**: Evaluator (p.36-42) **Technical Solution**: Parser Module (p.62-75) **Testing:** **Source Code Folders:** `Evaluator_Module`, `TreeTraversal`	**Testing:** (Video: 16:23)	✓
4. Error Handling *“**When errors are encountered, the** Interpreter **must stop without ‘crashing’ or ‘hanging’**. This is not to be confused with the interpreted program crashing – this means that **our Interpreter will not crash itself when finding errors in the interpreted program**.”*	**Design**: Error Output (p.23) **Technical Solution**: Error Handling (p.43-44) **Testing:** **Source Code Folders:** `Errors`	**Testing:** (Video: 23:12, though tested throughout)	✓

5. 5, 6 & 7. Programs & Syntax *These objects are combined as their testing represents full Interpreter functionality and facilities.*	**5.1**: Working output functionality **Design**: `FuncCall` Class (p.31) **Technical Solution**: ○ Parsing Function Calls (p.55 & 60) ○ Evaluating Function Calls (p.74) **5.2**: Working input functionality **Design**: `FuncCall` Class (p.31) **Technical Solution**: ○ Parsing Function Calls (p.55 & 60) ○ Evaluating Function Calls (p.74) **5.3**: Strongly typed variables & no reassignment **Design**: `VarDeclare` Class (p.31 & p.34) **Technical Solution**: ○ Parsing Declarations (p.52, 54, 59) ○ Evaluating Declarations Calls (p.73) **5.4**: Code blocks denoted by `{ }` **Design**: `Capturing IF code-blocks` (p.35) **Technical Solution**: ○ Capturing IF statements (p.58) **5.5**: Functional two-operand comparisons **Design**: `Capturing IF conditions` (p.35-36) **Technical Solution**: ○ Parsing Function Calls (p.55 & 60) ○ Evaluating Function Calls (p.58)	**Testing:** (Video: 23:25)	✓

Testing

I have decided to test each module individually, then all three modules chained together. The first three objectives are focused on the modules working individually, and the rest will be assessed when I test the overall program.

Testing video: https://www.youtube.com/watch?v=v69VTesDWaI

Timestamps can be found on table above.

Evaluation

Overview

I am pleased with the way my project has turned out – all objectives have been completed and I feel that I could make extensions quite easily due to the modularity and reusability of the codebase I have produced.

Objective Fulfilment

I consider all objects to be completed fully.

- **Objective 1 – Functioning Lexer Module**
  - **I have created a Lexer module that groups characters into tokens**
  - They are grouped with context (types) such as ‘string’, ‘identifier’, ‘grammar’, or ‘number’
  - Grouping works correctly and syntax errors are handled
- **Objective 2 – Functioning Parser Module**
  - **The Parser module is completely functional**
  - It recognises patterns in the tokens and creates a list of Step objects with useful information
  - Syntax errors that were not caught by the Lexer, such as invalid programming statements, are handled by the Parser
  - All syntax objectives are recognised (e.g function calls, if statements)
- **Objective 3 – Functioning Evaluator Module**
  - **The Evaluator module does exactly what is required**, though Boolean logical calculations could have been added instead of fixed 2-operand comparisons
  - It correctly interprets the evaluation step list given by the Parser and acts out the instructions
    - Outputs, inputs, variable changes, etc.
    - All syntax objectives are met here too
  - Non-syntax errors are handled correctly and output too
    - E.g invalid comparisons, variable reassignment
  - Mathematical calculations, including unary minuses, are carried out in the correct order
- **Objective 4 – Error Handling**
  - Throughout the Testing section, the interpreter program itself has not crashed at all
  - All errors in the interpreted program have been recognised and handled to prevent the interpreter crashing

    - All errors have descriptive output messages, though they could have been improved by having line / char number.
- **Objective 5 – Syntax**
    - As mentioned in Objectives 2 & 3, **all defined syntax in this objective is valid program code and can be run on the interpreter, completing this objective**
    - I could have added more syntax points that reused some code
        - E.g I could have added ‘else if’ statements which reused ‘if’ statement code but slightly modified
        - I could have also added function declarations, though this would have strayed further out of the required scope
        - I could have added more built-in functions, e.g data type conversions (like ‘int’ and ‘str’ in Python)
- **Objective 6 – Overall Facilities & Features**
    - As shown in the Testing video, **I was able to write programs with a good range of different features used**
    - For example, I was able to write the same program to output numbers from 1-99 incl. in three different unique ways
    - I was also able to create a complex guessing game with a limit on how many attempts could be made
    - There are enough facilities in the language to create these types of programs and potentially more, so I consider this objective complete
    - I could have added more built-in functions and perhaps more loops, though this would have made the project more complex
- **Objective 7 – Freedom of Solution**
    - As mentioned in Objective 6, **I have been able to write the same program listed in this objective and run it correctly all three times**
    - There is freedom to write a program in many different ways to solve the same problem

User Feedback

I showed Ed, someone who has programmed in a few languages already, to look at some example programs as if he was a beginner first and then as a programmer.

Ed looked at this program first:

```
int index = 0;

while (index <= 1000) {
    output("Count: ");
    outputln(index);
    index = index + 1;
}
```

Ed said he liked how the **syntax was clear** and **function names were self-explanatory to a beginner**. He also said it was **good that the language had while loops**, as they were one of the most important things to learn as a beginner for him. He also said it **would have been better to have conversion functions such as toString**, so that he could print a string and number in one line (‘`outputln(“Count: “ + toString(index));`’) – this would add more complexity to the project as toString would need to return a value for the expression to work, so **I do not regret leaving this out**.

After showing the number counting program, I then showed him a complex guessing game:

```
string name = "Torin";
string guess = "";

int attemptsAllowed = 5;
int guesses = 0;

while (guesses < attemptsAllowed) {
    output("Guess the name. Attempts left: ");
    outputln(attemptsAllowed-guesses);

    inputStr(guess);

    if (guess == name) {
        output("You guessed the name! Attempts needed before correct guess: ");
        outputln(guesses);
        guesses = attemptsAllowed;
    } else {
        guesses = guesses+1;
    }
}

if (guess != name) {
    outputln("You did not guess the name.");
}
```

Ed said that this was a complex program for beginners, and that it would probably not be the type of thing they would be required to write on their own. He also said that if the language had more facilities, this program could be made a lot shorter, but that these facilities would not be used by beginners. When running the program, he liked that the input statements automatically output a '>' character to indicate that an input is required.

```
Enter a valid file name to run: C:\Users\torin\Desktop\test\prog2.txt
-------------------- PROGRAM STARTED --------------------
Guess the name. Attempts left: 5
> ed
Guess the name. Attempts left: 4
> torin
Guess the name. Attempts left: 3
> Torin
You guessed the name! Attempts needed before correct guess: 2
-------------------- PROGRAM ENDED --------------------
```

Ed also commented that while the running of a text file is fine for now, it would be nicer to have an interactive shell implemented – this would allow the user to type in statements and instantly run them like commands, with expressions also being input and returning values.

Summary

- ☑ Ed liked how clear the syntax was, from both a beginner and experienced programmer's perspective
- ☑ Ed liked that the function names were descriptive, and the code blocks were clearly outlined
- ☑ He also liked that there was enough complexity to write a guessing game

- ☒ Ed wished there were pre-registered file extensions so that files could be automatically run when opened, instead of having to run the *NEA_ProgrammingLanguage.exe* file and type in the path to run
- ☒ Ed also wished that the interpreter had a little more complexity, such as better comparisons or type conversion functions, as he felt that he didn't want to limit the programmer too much and cause them to downgrade their program to adhere to these limits
- ☒ Ed wished there was an interactive shell implementation, similar to that of Python

What improvements & extensions would I add?

Reflecting on the user feedback I have been given, I would likely add more facilities to the language itself to create more efficient programs.

Potential Function Implementation

- I would have the ability to define functions, e.g:

```
func fib(int n) returns int {
    if (n <= 1) {
        return n;
    }
    return fib(n-1) + fib(n-2);
}
```

This is a recursive Fibonacci sequence function.

- This would include a new 'return' statement, with the option to have a subroutine that does not return a value
- As well as this, I'd include more built-in functions, such as *toString* and *toInteger* – these would return the value input converted to a string or integer, allowing better expressions to be made, e.g "10 + 1 is: " + toString(10+1).

Potential Interactive Shell Implementation

- I would add an interactive shell, allowing the user to type in statements or expressions and instantly run them line by line
- It would look something like this:

```
>> int x = 0;
>> x
0
>> x + 1
1
>> while (x < 10) {
2       x = x + 1;
3       outputln(x);
4   }
1
2
3
4
5
6
7
8
9
10
>> x
10
>>
```

- The user would be able to input a statement such as a variable declaration
- They would also be able to type in any expression and have the result returned. For example, 'x + 1' returns 1, though it does not actually change the value of X at all.
- Multi-line blocks will be possible as well, with line number displayed in green on the left

Potential Advanced Expression Analysis

- In the current project, if/while statements can only contain a two-operand condition such as the following:

```
if (x == 1)
```

- If I could implement it, I would add a modification to the Djikstra Shunting-yard Algorithm (used for maths only right now) for it to support Boolean expressions. This would allow an expression like this to work:

```
>> True && (True || False)
True
```

```
>> (1 == 1) && ((2 == 2) || (2 == 1))
True
```

Overall Project Reflection

I am satisfied with the result and feel that **this project could be used to solve the problem** of learning programming with a 'beginner language'. There is enough complexity to write some good programs to learn from, but also not too much that it results in confusion. However, I still think that there is more that can be added as mentioned above – **I am also not happy that I had to use limited two-operand comparisons** but building a more complex expression resolver would be out of scope.