OCR ERL Interpreter Project

Computer science NEA – PATRICK WILLIAMS (8123)  
Saint olave’s Grammar School

# Contents

## Analysis of the problem

* Problem identification. *(Establish Problem and Stakeholder)*
* Problem Research. *(Similar Problems and Analysing their Solutions)*
* Proposed Solution. *(Basics and Stakeholder Needs)*
* Solution Research. *(Technologies Used and Computational Methods)*
* Features of the Solution. *(Essential Features, Limitations, Requirements, Success Criteria)*

## Desgin of the solution

* Decompose the problem.
* Describe the solution.
* Describe the approach to testing.

## Developing the solution

* Iterative development process.
* Testing to inform development.

## Evaluation

* Testing to inform evaluation.
* Success of the solution.
* Describe the final product.
* Maintenance and development.

## Final codE

* Source code.
* Version History.

Analysis of the problem

# Problem Identification

## Establishing the Problem

All programming code given in the OCR Computer Science GCSE will be presented using the OCR Exam Reference Language (ERL), so it is important that the students have a solid understanding of the syntax of the language in order to be able to successfully answer many of the questions. Additionally, for questions where the student is required to write their own code, students are given the choice to respond with either the ERL or a high-level programming language.

Despite the code-based questions being written in ERL, a large proportion of GCSE students choose to answer writing questions with a high-level programming language instead of the ERL because of how unfamiliar they are with OCR’s Language. This brings up the question of whether students in school should be primarily taught with the ERL or a high-level programming language of the school’s choosing.

### **Which option is better?**

It is of course important that students are taught the basics of a high-level programming language in school, due to its real-life application in their future. However, in terms of the exam papers themselves, there are advantages in using the ERL instead of an alternate language.

#### syntax differences

The syntax of the ERL differs in some ways from high-level programming languages, which can create confusion during the exam. Although the exam board is lenient, allowing small changes from the formally defined ERL syntax and instead focusing on the logic behind the code, there are some large differences which can cause confusion and for marks to be lost in an exam.

🡺 INCLUSIVITY AND EXCLUSIVITY  
One difference between ERL and many high-level programming languages is very prominent with counter-controlled loops. Take these code snippets below:

|  |  |
| --- | --- |
| A black background with white text  Description automatically generated |  |

A screen shot of a computer code

Description automatically generatedDespite both using the same numbers, ERL will print “Hello World” 4 times, whereas Python will only print it 3 times. This is due to ERL being an inclusive language, which therefore can create confusion with students using Python, a very popular choice, and can cause errors throughout the paper.

This issue is not unique to Python, in languages with C-like style of creating for loops, including Java, PHP and JavaScript, the standard is to create for loops with a < operator, rather than <=, therefore also being naturally exclusive. With ERL being a rare case of inclusivity, this can easily cause issues, especially with “fill-in-the-gaps” style questions and for students who attempt to use ERL without a thorough understanding of it in writing questions, where this difference can easily lose a student marks.

🡺 SUBSTRINGS  
There is a drastic change to the formatting of producing substrings in ERL compared to the standard for many other programming languages. This can also be a source of confusion in the exam.

|  |  |
| --- | --- |
|  |  |

The OCR ERL has an unorthodox substring syntax, with the first argument remaining as the initial index, but the second argument being the length of the returned substring in characters, instead of the standard of the final character position +1. Again, an unfamiliar student may get caught out in an exam.

🡺 MISCELLANIOUS CHANGES  
There are also many small, but still potentially dangerous changes between languages.

* The use of “MOD” and “DIV” instead of symbols in typical languages
* The use of ^ as an exponent (\*\* is used in Python, a very commonly chosen language)
* .upper and .lower are properties rather than methods (have no brackets afterwards)
* The use of closing keywords such as endif and endswitch (uncommon in modern languages)

Although many of these minor changes will generally be overlooked by an examiner during the marking process, a student who is fluent in the syntax will feel much more comfortable in the exam when occurring the custom syntax which they might be unfamiliar with.

#### Learning Resources

A close-up of a book

Description automatically generatedA paper with text and a questionnaire

Description automatically generatedEven though it can be argued that a high-level programming language can be interchangeable with ERL in the exam itself, many of the textbooks which are given to students use ERL in their explanations and exercises for students to do. This is as there is no set high-level language that any given school teaches, and the studied language is chosen by the school itself. Therefore, ERL is used within resources as a universal language utilised by the majority of textbook. This means many programming concepts, such as sequence, selection and iteration, as well as algorithms, such as linear search and bubble sort, will have their code written in ERL and this is how students will encounter and learn these key concepts for the exams.

This therefore presents ERL as a superior language than any alternative for the OCR GCSE, due to its syntax aligning with the mark scheme, and its wide usage within resources which are taught to pupils.

### **WHy are alternative languages more commonly used?**

The main issue with the ERL is the lack of a way for students to use it. ERL is not a pseudocode, and is an equivalent to a high-level language, so has been written in a way where it is interchangeable with real, functional code. However, there is no translator available which allows students to write and consolidate their skills with the ERL. This alone makes using the ERL mainly redundant, as students can only become familiar and comfortable with a language after using it for a prolonged period, as it this process of trial and debugging that allows their code to be reliable and ensures that they can produce accurate and correct code during the exam.

**Therefore, our problem is that students have no way to practice ERL, so are unfamiliar with it by the time the exam occurs.**

## Stakeholders

The target audience for our solution would be, obviously, for students partaking in the OCR Computer Science GCSE, however more specifically, they would likely be:

* First time programmers, with little or no prior programming experience.
* Using restricted school devices or poor-performance home devices.
* Unable to install and setup any complex software.
* Require a very intuitive solution with lots of help at disposal.

In order to gain a greater insight into the experiences of potential stakeholders, I have interviewed 3 different individuals to understand their experience with programming. I will also return to these stakeholders later to ask for their needs towards a solution I decide on.

### **current gcse student**

My current GCSE Student is Abigail Williams: a year 10 student who has been learning Computing for 3 years and started the GCSE content a year ago.

#### What are you past experiences with programming?

“My previous experiences with programming have been rather bad because I struggle to understand the computing language format and how it all goes together. On top of this, it takes me a long time to type it out in the language, and I would prefer it was simpler and more like English.”

🡺 This instantly outlines to me what a new and different experience programming is to anything else taught in school. As somebody with 3 years of experience still struggles to formulate the logic behind a language, it shows how schools need to focus more on practicing programming so that all students get it.

#### What’s been your prior experience with the ERL?

“What’s the exam reference language? I have used python before in the past, but I have never heard of this one until like 5 seconds ago.”

🡺 Shows how schools generally ignore teaching ERL content, and how they focus on high-level languages, despite some clear advantages which would help the student. After showing and explaining what the ERL is to her, she said that.

“Okay, that seems important to know if it’s going to be used in the papers, so I should probably make sure I learn about it a little bit”.

🡺 Even if students choose to not write their answers in ERL, a complete lack of awareness is slightly concerning, so teaching about the ERL needs to become more widespread.

#### What are your plans to revise coding?

“I’m going to make a revision booklet and write notes in it about the rules of the language that I’m going to use so I can try and remember. I know practicing is meant to be good, but I don’t know how to, and it sounds very difficult so I’m not sure if I will.”

🡺 A lack of an easy and obvious way to practice coding is very clear, as these techniques are far from ideal in order to build familiarity: so, as well as being unaware of the language, general revision techniques for computing have not been taught, as they are not as standard as those for other subjects.

#### Conclusions

Programming seems very unfamiliar to students like Abigail: where the focus needs to be on increasing their familiarity with programming as a whole, as they won’t be aware of the different languages or practice techniques available to them. Therefore, a specific solution for GCSE students, which they know will be useful in their GCSE because it’s built specifically for students like them, would help to simplify the process and ensure that students at this stage continue to develop their programming skills.

### **past gcse student**

Tanish Arjaria is a year 12 student who did OCR GCSE Computer Science last year and achieved a grade 9, who is now continuing to take Computing for A-Level.

#### Did they use ERL or high-level in GCSE?

“I chose to write in python in the exam papers. Although I know a decent bit about ERL, I am not as confident in using it as I am with python, so I felt as if it was a less risky choice. However, for the questions where code had already been written in ERL – such as the fill in the gap ones – I did write in ERL.”

🡺 This high-achieving student chose to use a high-level language, despite being clearly aware of the ERL, showing how his lack of practice in it led him to make his decision.

#### How did they practice using ERL?

“I looked at online ERL cheat sheets before exams and saw questions where it was used in past papers. Apart from that, I have never practiced using ERL at home, because I didn’t know where to find questions like we did in class.”

🡺 Students are not aware of how they can familiarise themselves – even the best achieving students struggle to practice regularly – forcing them to select a language without providing the choice for them to choose which one works best for them.

#### Find the experience of revising erl easy or difficult?

“I didn’t spend much time deliberately practicing ERL, so it’s hard to describe it solely as easy or hard. It’s more of a thing where a random ERL question comes up whilst revising different topics, but I didn’t have a single set way to revise it – so I guess in that aspect it was kind of difficult as there’s not much option available.”

🡺 It doesn’t even occur to students that they could be revising programming too, despite it holding so much value in paper 2 of the GCSE. It has been normalised as something which can’t be revised which should be changed.

#### Potential solution would have aided revision?

“I be so for real; it would have genuinely helped as I didn’t spend any deliberate time on ERL at all. Even though I am confident in programming generally, so it wouldn’t have helped in terms of improving those skills but making me more familiar with the syntax and stuff would have just made the exam a bit easier”.

🡺 Even for confident coders, just a way to help memorise the syntax would be useful, and this extends even further to less confident ones: so, a tool to practice would be useful for all kinds of students.

#### Conclusions

Overall, Tanish has made it clear that revising ERL is just not something that occurs to students, and it is never a focus in their revision. This highlights the importance of a potential solution as it could lead to practicing becoming more normalised, which would be very beneficial to all kinds of students in the exam.

### **computer Science teacher**

* Programming focused on high-level or ERL?
* Limitations on using ERL?
* Potential solutions be useful towards teaching?

We will return to our stakeholders later to ask for their needs in a potential solution.

# Problem Research

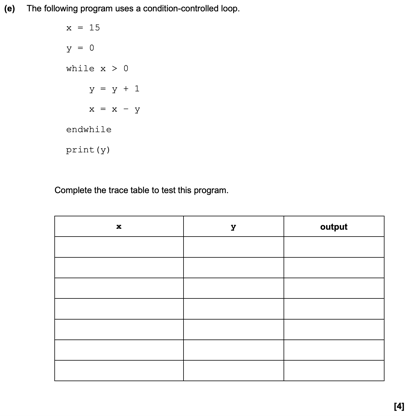
There are many other problems with pre-existing solutions, which can be analysed to provide useful insight into what my potential solution can be.

## Solutions FOR Erl

Initially, the focus will be on pre-existing ways that students can run ERL code.

### **trace tables**

A white sheet with black text

Description automatically generatedStudents should already be familiar with trace tables, as there are questions in the exam papers, for example this one from the sample paper, of students being required to use them to “run” ERL code. However, when used to run a student’s own code, it presents a slightly different task.

#### Advantages

🡺 Students are taught about trace tables, so it is an easy and familiar method for them to use on their own code, and it allows them to check if the logic behind their program is functional.

🡺 It serves as a form of debugging, as it allows students to analyse the logic behind the program that they have written and allows them to track the values of the variables throughout – much like a debugger – which can help produce more accurate code.

🡺 It does not require any software, just a pen and paper so it is a very feasible solution which can be done anywhere, making it very accessible.

#### disadvantages

🡺 There is no syntax checking at all, so whether the code is accurate cannot be determined as it is down to the students understanding of the language and may be incorrect in an exam.

🡺 This is also true of the logic, as if a student has a flawed understanding of how the logic of the language works, as they are the interpreter in this scenario it means that they may unintentionally gain an incorrect solution which they believe to be correct, as they have interpreted it based on their inaccurate knowledge of the language due to their unfamiliarity, and would lose them marks during an exam.

🡺 The task is very slow and tedious, especially when it comes to longer programs, so there becomes a point where it is intangible to use a trace table when the program reaches a certain size, because the number of variables becomes too tedious to track, and a much faster solution would be preferable.

#### What can i apply to my solution?

Although, overall, a trace table is a horribly unreliable and slow method of running a program, it does teach lessons about ease of use which are very important to my own program, in order to make it available to beginners. Also, the very accessible nature is of importance, as there is no use of coming up with a better solution if it is unavailable to a vast number of students.

### **translation to other languages**

This method involves a student writing the original code in ERL, then translating it to a high-level language of choice in order to run it and obtain a result.

#### Advantages

🡺 Does allow for the program to be ran, and the student can produce a result, which allows students to test programs and feel comfortable with their ability.

🡺 Allows simultaneous learning of ERL and a high-level programming language at the same time, widening the scope of the student and producing a more educated individual.

🡺 Having to re-read ERL to translate it allows the student to scan for errors more effectively, hence making them more accurate with the language,

#### Disadvantages

🡺 The ERL stage can become heavily redundant, with very minimal thought behind it, as the high-level programming language is what is ran, so the student would tend to become much more proficient in that, as the ERL was just an unnecessary stage which can become ignored and therefore not learnt by the student.

🡺 Very minimal focus is applied to how correct the ERL written is, as it is only the high-level language that is executed, so if a student makes lots of errors in the ERL, but produces a correctly functioning high-level code, it can result in them learning incorrect syntax for ERL in cases. This can make this method negative, as the false reinforcement could lead to very big flaws, as it is only their own interpretation of the syntax of the language which determines the result of the program.

🡺 Again, the process is very time consuming, and really discourages the student from attempting to write accurate ERL as it makes the programming process so much longer.

#### What can I apply to my solution?

The very negative evaluation of this method shows the importance of a fast solution that encourages the student to write in ERL, as otherwise using a different high-level language would be a much easier solution for them.

### **Conclusions**

The overall result of this analysis is that an interpreter for the ERL is required, as otherwise running ERL is such a time-consuming process for the student. This would require a program that would be able to be presented a program written in ERL, and directly produce a result from it, as otherwise students would be much better off with using a high-level language.

Therefore, researching how interpreters for other high-level languages function would be beneficial in identifying which key features are important for me to implement.

## Solutions for other languages

There is an abundance of interpreters available for other languages, and I will be analysing them based on their application towards our stakeholders as alternatives, not for their typical users.

### **NOde.JS in Terminal**

This is a command-line interpreter for JavaScript which can be run in a terminal on a device by calling it with “node” with no separate application.

#### Advantages

A computer screen shot of a code

Description automatically generated🡺 The obvious first advantage, is that it successfully runs JavaScript code when entered. This may sound irrelevant to mention as a solution, but as there is no option for ERL that does this, it already presents this interpreter as much superior to any ERL methods that could be chosen.

A screen shot of a computer

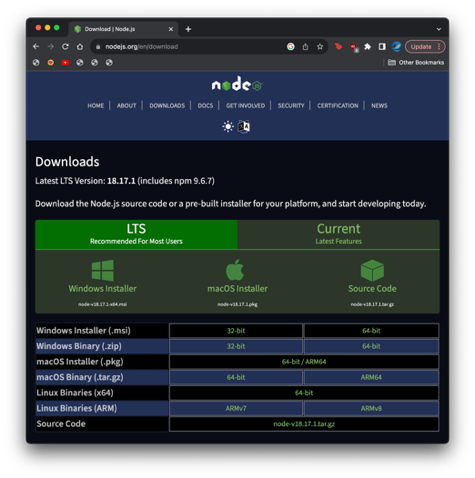
Description automatically generated🡺 It also contains error handling, referencing both the error type and where it occurred, which is very useful for anybody writing code, especially to students who are prone to mistakes, making learning the language much easier as well as aiding debugging.

A screenshot of a computer error

Description automatically generated🡺 Multiple use modes allow the students to use the program depending on their needs. The shell, for a few lines of code to be run individually; the editor for when a longer program is required, to all be interpreted as once; and the ability to select a file in local storage and run it. This makes the interpreter much more diverse and can adapt to the need of the user.

🡺 The basic help feature allows the user to be able to fluently use the compiler, as a reminder on how to operate it with the command-line interface.

#### Disadvantages

🡺 Lacking a graphical user interface of any type, this makes it very difficult to use, especially for the stakeholders who are first-time programmers. A command-line interface is very complex, and the terminal is often blocked on school devices, making it unapplicable for our student stakeholder who need to use it.

🡺 Installation is also difficult, requiring installation from the command line or downloads from the website, meaning a student requires an administrator to be able to access it, which reduces the scope of the interpreter.

🡺 The poor user experience stems from the command-line interface. Users must memorise many commands to access the different features of the interpreter. Although this makes it much faster to use for experienced programmers, for beginners it becomes much more tedious, and could do with a much more intuitive solution, as well as the shell being slightly clunky to use at points,

#### what can i apply to my solution?

🡺 Ensure that error handling is implemented as it is very important in an interpreter  
🡺 Provide an intuitive interface as the interpreter is intended for beginner students.  
🡺 Prioritise accessibility and that it is available to many.

### **Python Idle**

I will be building upon the previous analysis of node.js for this analysis, as many of the basic features, such as running and error handling are taken for granted, so will not be mentioned.

#### Advantages

A screen shot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a phone

Description automatically generated🡺 The user experience is vastly improved, with use of the ribbon to introduce a greater array of options that the programmer can use, allowing equal if not greater functionality than node without the need to memorise all the commands, making it vastly easier to use for beginners.

🡺 Includes a debugger, with persistent breakpoints, stepping, and viewing of global and local namespaces, which can aid the students understanding of the language as, after all, it is correcting mistakes that allows a student to develop their understanding of a language.

🡺 Overall a more easy-to-use solution. Some examples being the syntax highlighting making it easier to identify blocks of code, as well as the shell element of the program being very logical to use, which runs after the program is complete, even allowing the previous file that is run to be referenced from it, making testing very easy. Overall, the interpreter is very well made and intutive for beginners to use.

#### A screenshot of a computer Description automatically generatedDisavantages

🡺 There are still accessibility issues present. The program still needs to be downloaded from a website or using a package manager, and will still require administrative permissions to install, which can decrease the scope of the program in terms of who can use it.

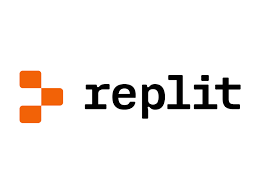
🡺 In terms of helping absolute beginners, the official documentation is still quite dense and hard for new programmers to undestand. Although there are a multitude of beginner guides avliable on the internet, when thinking soley about the official documentation, it would make the experience harder and is worth considering for an ERL interpreter which is aimed directly at new programmers.

🡺 Having to save files locally before being able to run them can be a nuisance a points: a very small critique but worth considering, as overall IDLE is a very strong interpreter.

#### WHAT can i apply to my solution?

Trying to replicate the visual, ease-of-use of IDLE is a strong target, as it makes focusing on the code itself very easy, as operating the interpreter itself is very straightforward. In terms of improvements, the focus is still towards making it available to as many people as possible, so considering low-performance devices and avoiding downloads would still be useful.

### **Replit**

Replit is an online IDE which allows people to write code on a website, where it is then run on their cloud servers and the results are returned to the client.

Again, the evaluation builds upon the points of the previous, as many of the advantages are shared with IDLE, including debugging features and syntax highlighting.

#### Avantages

🡺 The accessibility is the highest out of any solution. Being a website, it requires no installation, so can be run by any student without requiring assistance from an administrator. Additionally, the program itself is ran on cloud servers, so any low-performance devices are not affected due to the program being executed externally, resulting in the solution having a very large scope.

A screenshot of a computer

Description automatically generated🡺 The community aspect is extremely prevalent, with an extremely large number of users sharing their code, and aiding others by providing advice and feedback. This positive loop leads to a very good environment for new coders in order to learn programming.

#### Disadvantages

🡺 A potential negative is that the functionality of the solution has resulted in the interface becoming very convoluted, and potentially confusing to a new user, which could make programming become a much less approachable experience for them.

🡺 It doesn’t run ERL!

#### What can i apply to my solution?

A website solution seems extremely promising, as it allows it to be extremely accessible to lots of students. Due to the huge size behind Replit, some features such as the cloud servers and community feedback will be impossible to replicate, but the focus on creating an environment where anyone can run code, with a focus on learning is a strong takeaway.

# Proposed solution

Through research, I have decided the solution chosen will be to create an interpreter for the ERL, and I have decided upon a small list of high-level objectives which are my main priorities for the program.

🡺 The interpreter is available on a website so it is accessible to many students  
🡺 The interpreter can execute the majority of ERL code, including  
 🡺 Arithmetic features, with correct order of operations  
 🡺 Different data types: strings, integers, floats, bools  
 🡺 Basic data structures: variables, constants and arrays  
 🡺 Logical features, such as if statements and switch statements  
 🡺 Iterative features, with counter-controlled, while and do while loops  
 🡺 Function and procedure calls which can be user defined  
 🡺 Input and output features, taking in from users and printing out to a shell  
 🡺 Typecasting, commenting and a limited amount of smaller features (to be expanded on)  
🡺 Users are able to type code straight into the website, upload files and use the shell  
🡺 An easy-to-use and minimal user interface for beginner programmers  
🡺 Error handling, with reports with details on why a program cannot run  
🡺 A help system targeted towards inexperienced programmers

This is just an initial draft, of what I feel are the basics for what must be included at this stage. I will continue to research and check with my stakeholders in order to refine this list later in greater detail, along with how I can quantitively measure the success of each objective.

## Stakeholder Needs

After showing them my list of high-level objectives, I asked for their feedback.

### **Current gcse student**

“Even though I don’t understand a lot of the things, it has got the stuff I know like variables, if statements and the loops. For me, making it have a way to explain how the different bits work would be very important, as without instructions that I can see I would get very confused, so I’d find that useful.”

🡺 Even though coding a functioning interpreter is the priority, the help features are almost as important, as without them the beginners – who are the intended user of our interpreter – will still remain very confused and it won’t be any help for them.

### **past gcse student**

A person in a suit using a computer

Description automatically generated“I think the list of ideas involve the majority of things that are included in the ERL. I know there are some other smaller features of it, but I’m not sure if he’s including those, but they would be useful but not absolutely necessary. However, the main other thing I’d like is syntax highlighting, as I think it makes the code much more readable and would be a lot easier for beginner programmers, so if he could colour-code the keywords it would be a very good feature.”

🡺 Confirmation that all the key features have been included is helpful, and I will discuss the limitations later. The idea of syntax highlighting is also a good idea, as it would overall make the program easier to use, which is my priority, so I will try to include it as its useful for those of all skill levels.

### **computer science teacher**

# Solution research

Waffle.

## computational approach

Solving this problem with a computational approach is clearly a very logical solution. I have already researched other methods of trying to execute ERL code, the only method being a trace table, which I concluded was a very inefficient method to use. It is in its nature to use a computer to execute code, so therefore it is very clear that using a computer is the correct solution.

## Technologies used

It is important to choose the correct technologies from the start of the process, as it is difficult to change halfway through a program.

### **Programming language**

The programming language that is used is a very logical first decision before making a program. As one of our high-level objectives is that it is a website, we must ensure that it is possible to run on a website.

#### Javascript

The obvious initial choice is to use java script. The modern, most frequently used language which runs on websites.

## Computational methodS

# NOTES:

Problem Recognition:  
Problem Decomposition:  
Divide + Conquer:  
Abstraction:  
Backtracking:  
Data mining:  
Heuristics:  
Performance modelling:  
Pipelining:  
Visualisation to solve problems:

Design

# Explanation

This stage will be describing the structure and logic behind my coded solution. As an interpreter is a very computationally demanding project, this section will be very detailed on the overall structure of how the solution works, and will link very closely to the development section, where the code I write will be mainly explained here, so I will follow the same general layout in both.

### References

I have compiled my research from a list of different sources that I have listed below. Note that I have never used and will not use any actual code from any of my sources for my actual project, but the overall technique for a compiler is standard so I have used lots of examples in order to learn how I will structure my project.

🡺 [https://youtu.be/SToUyjAsaFk](https://youtu.be/SToUyjAsaFk?list=LL)  
🡺 [https://youtu.be/JO\_0e9mPofY](https://youtu.be/JO_0e9mPofY?list=LL)  
🡺 <https://craftinginterpreters.com/a-map-of-the-territory.html>

# An overview

This first section is going to be an overall demonstration of how my interpreter is going to function. We are going to take a relatively basic arithmetic expression, in plaintext, and go through the logical steps the interpreter will take to produce a result. This includes the three main stages: lexical analysis, parsing and then execution. I will later go into detail about the plans to code each stage, but this is a general abstracted explanation of the process which removes all the complicated details.

### Lexical Analysis

The interpreter will take a plaintext input, for example being the one below, which we will be using throughout this demonstration to eventually produce a result.Currently, to the computer, this string of characters holds absolutely no meaning or value, so in lexical analysis, the meaningless string of characters is converted into a list of tokens. This is done by a program called a lexer.

On a simplified level, the lexer takes a line of the input text, and goes character by character, looking at the type of character or arrangement of characters, and converts it into a list of objects, which are called tokens. Some characters are very easy to convert into tokens, such as parenthesis, however, when tokens which are longer than a character long become involves, it requires much more difficult code to distinguish multi-digit integers, strings, floats and variable names.

The lexer must also be trained on what to ignore, including white-space characters and comments, as what is produced at the end of the lexing process must be a list of meaningful tokens which can be given to the parser so the next stage can start.

A hexagon with black text

Description automatically generatedIn terms of the tokens that are produced themselves, they are all objects, as I am using an object-oriented approach to this project. Every single token has a token type, and every single token type has its own class, with different properties. Throughout the process, we will define and create different types of tokens, but for now we have four different groups of tokens.

🡺 Literal tokens hold values, with different types being integers, floats, Booleans etc. They will have a property called value which, obviously, holds their value.

🡺 Identifiers, at this stage, are variables, and will later include function and procedure names. For now, they would have a property which points to the location of their value in the program (to be expanded on later)

🡺 Binary operators are the arithmetic operations, including adding, subtraction, multiplication and so on.

🡺 Finally, there are other kinds of characters, such as parenthesis and dots, which may be included.

At this point, the objects still hold no logic or meaning behind them but is merely the plaintext described in a way that a computer could begin to understand it: the equivalent of us reading a sentence, but not yet understanding the grammar and meaning behind it.

A number and text on a black background

Description automatically generated

Using our example, at the end of the lexical analysis stage, we have been left with this array of tokens which is equivalent to the plaintext, ready to be passed onto the next stage of the process.

### A screenshot of a computer Description automatically generatedthe very basic concept of Parsing

Parsing is the most complex stage out of the three and is where the tokens from the lexer are given a meaning, a grammar, that means something to the computer. This is where our simple list of objects is built into an abstract syntax tree, which is a type of data structure. Where each of our objects is linked in some way to another object, as its children. The job of the parser is to correctly link all these objects together in the correct structure based off the rules that we give it.

These set of rules become very complex, but in our example, the computer needs to arrange the objects based on the mathematical principle, order of operations. This has two key parts:

🡺 Precedence: the order itself. This is how certain features have priority, with items in parenthesis being calculated first, then any exponents, then multiplication and division, then addition and subtraction, according to the BODMAS structure of mathematics that we use.

🡺 Associativity: this is the order in which operations with the same precedence are executed. For example, a chain of subtractions will result in a different value based off whether they are calculated from left to right or from right to left. In BODMAS, this is from left-to-right for arithmetic expressions.

Based on these rules, the tree is created, with the calculations that need to be performed first at the bottom of the tree, and this is the overall premise of parsing before the result is produced.

Although this may seem simple with our BODMAS example, the premises of precedence and associativity are still present for all other aspects of the language. When keywords and different data structures and functions and all the specific features of the language become involved, it makes this parsing step extremely complex to create.

### What are the set of rules?

In my brief description, I described how a set of rules determined how the abstract syntax tree would be created. This sounds extremely vague, but it needs to be extremely precise in the actual code. This is where a special kind of notation is used, called Backus-Naur form - or BNF for short - is a context-free grammar used to describe the how the logic behind a grammar is defined.

#### A basic example of bnf

For this we will be defining what an integer is using BNF. There are a few key aspects of BNF. The definitions by themselves may not make much sense but will be clear with context afterwards.

🡺 name = a terminal (aka lexemes), this means it is in its lowest state, and cannot be expanded upon, for example the digit 1 is a terminal as it is clearly in its defined form

🡺 <name> = a non-terminal, which can be expanded and defined by a series of terminals or non-terminals

🡺 | = or, used when defining a non-terminal which can have several different ways it can be defined

🡺 ::= = the equals of BNF, where the non-terminal on the left-hand side of the ::= is defined by what has been written on the right-hand side of it.

This may sound extremely confusing for now but let’s begin the process of defining an integer. We must first decompose the integer into the smallest item that it is comprised of: a digit, which we can easily define with the following statement:

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

This statement shows how a non-terminal called a digit, is defined as a 0 or a 1 or a 2 or a 3 and so on. This may seem redundant and obvious information to a human, but every little thing must be explained to a computer. This is why this example is a good introduction to BNF, as it shows how every little detail must be considered so the grammar is built from the ground up. Furthermore, we can then expand on this definition.

<integer> ::= <digit>|<digit><integer>

This statement says that the non-terminal integer is defined as a digit OR a digit combined with an integer. This recursive self-defining forms the basis of BNF, as therefore this means that an integer is defined as 1 or more digits, with no limits on the number of digits available, hence defining the integer as every single possible integer ever. It is the structure like this that allows BNF to represent the grammar of entire programming language by building up from the most basic steps.

However, our definition of an integer is still wrong, because an integer cannot begin with a 0 as the first integer, or two zeros as the first two, as these integers have already been defined. Therefore, the actual definition for an integer would be as follows:

<non-zero digit> ::= 1|2|3|4|5|6|7|8|9  
<digit> ::= 0|<non-zero digit>  
<digits> ::= <digit>|<digit><digits>  
<integer> ::= <digit>|<non-zero digit><digit>

By ensuring that it does not begin with a zero, this has already significantly increased the complexity of the BNF by at least double, but it does show how important being precise is, and the attention to detail that is required in order to produce an interpreter that does not contain lots of errors.

#### Extended backus-naur form

Shortened as EBNF, Extended Backus-Naur Form is a more concise way of writing in BNF. Although everything in EBNF can also be written in BNF, it just makes the process easier to read whilst avoiding lots of repetition and reduces the amount of self-recursion. Taking our previous example, the new definition of an integer would be.

<non-zero digit> ::= 1|2|3|4|5|6|7|8|9  
<digit> ::= 0|<non-zero digit>  
<integer> ::= <non-zero digit>{<digit>}

The introduction of the curly brackets { } mean that whatever is contained within them can be repeated zero or more times. This eliminates the need for something to be defined by itself, and makes it overall clearer to read. Other additions to EBNF include:

🡺 (a|b) = an option that needs to be made, for example  
<x> ::= (a|b)<y> 🡺 <x> ::= a<y>|b<y>

🡺 [a|b] = an option that doesn’t have to be made, for example  
<x> ::= [x|b]<y> 🡺 <x> ::= <y>| a<y>|b<y>

Again, these shortenings just reduce the amount of repetition that needs to be made whilst writing BNF. There are different variations of the syntax that use different variations, but the only one I will really be using are the curly brackets, which for clarification has the example:

<x> ::= a{<y>} 🡺 <z> ::= <y>|<y><z>  
 <x> ::= a|a<z>

As this is very useful for shortening and occurs very commonly, but the main point to take away is that any notation that I write in EBNF is always able to be written in the simplest form of BNF, showing that the entire grammar of the language stems from these small definitions.

#### Order of operations in bnf

Just as anything in a language can be expressed in BNF, our order of operations problem can also be written in BNF. Below is the basic version which we can use to solve our solution. We will expand upon this later to include exponents and prefix operators, but for now we will just include the necessary addition, subtraction, multiplication, division and parenthesis needed for our example. The required EBNF is:

1) <expr> ::= <term>{(+<term>|-<term>}  
2) <term> ::= <factor>{\*<factor>|/<factor>}  
3) <factor> ::= (<expr>)|identifier|integer

A number and symbols on a black background

Description automatically generatedThis simplified solution also defines identifiers and literals as terminals rather than non-terminals. It can be argued that they would be non-terminals, but for the sake of explanation I have defined them as terminals, with identifiers being variables and integers holding a value.

Take this small example for an explanation of the process, starting with this set of tokens:

A number and symbols on a black background

Description automatically generatedLooking at the initial tokens, we have 3 literals, which are integers, and 2 binary operators, a multiplication and an addition. Just for clarity, we can label our integers, to distinguish them from the binary operators, which hold no value and are only there for logic.

A screenshot of a computer

Description automatically generatedLooking at our integers, our third line of EBNF states that a <factor> can be defined as an integer, so therefore each of our terms can now be labelled as a <factor>.

A computer screen shot of a number

Description automatically generated with medium confidenceAccording to our second line, one of the definitions of a <term> is two <factors> multiplied by each other, so we can therefore group our <factor>\*<factor> and label it is a <term>. The remaining 4 is also labelled a <term>, as a <term> can still be defined by a <factor> by itself, so long as there are no \* or / binary operators next to it, which leaves us with two <terms> separated by a + binary operator.

Now, our first line of BNF defines an <expr> as a <term>, followed by a + or – binary operator, followed by a <term> (just once in this case but can be repeated). Therefore, we can now define our entire expression as a single <expr> non-terminal as shown below.

A diagram of numbers and symbols

Description automatically generatedA diagram of a mathematical function

Description automatically generatedThis expression is now completely defined, as an entire tree has been created, starting from the single <expr> non-terminal, which allows us to then progress to the next stage after parsing. This more detailed perspective of how the different steps of logic work lead to the following abstract syntax tree:

This holds the exact same meaning as the previous diagram, just presented in a different, less confusing way which is closer to how it will be treated for the final stage. In reality, each of the different boxes is an object, with each of the arrows being a property within the object that points to the other object as its children, so our diagram is just a visualisation. However, the key takeaway is that our parser turns our set of tokens from the lexer into this abstract syntax tree, stemming from a single initial object at the start, which descends into sets of different pairs until the whole expression is now defined aligning to the rules that we have chosen, so that our precedence (order of operations) will be correct.

A computer screen shot of a graph

Description automatically generatedTo go back to our example, our original string of tokens will produce a BNF tree like this:

This clearly shows how quickly the diagrams get messy using this approach, and how complex the process of parsing is, so it is best to use abstraction, and remove the details of how the parser works, leaving us with our final parsed tree which can be passed onto the next stage.

A screenshot of a computer

Description automatically generatedThis process of parsing is called recursive decent parsing, and the process itself is very detailed, and can cause lots of errors which I will not go into detail about now. In the later stages of design, I will explain the actual coded steps in order to perform this process, but for now it is just the logic behind it that is of importance. The key points from this are that:

🡺 The rules of the parses must be very carefully defined  
🡺 All possible inputs must be thought out  
🡺 The process itself is very long and complex  
🡺 The result that must be produced must start from a single object.  
🡺 The order in which the tree descends must match the precedence of the operations so that the execution stage is correct.

Our original other point of associativity has not been mentioned but it was defined in the DNF stage. If we were to reverse the way that we wrote our DNF rules, then the operations would have been performed right to left but ensuring that associativity is correct in the parsers process is something extremely important that I have not mentioned.

### Execution

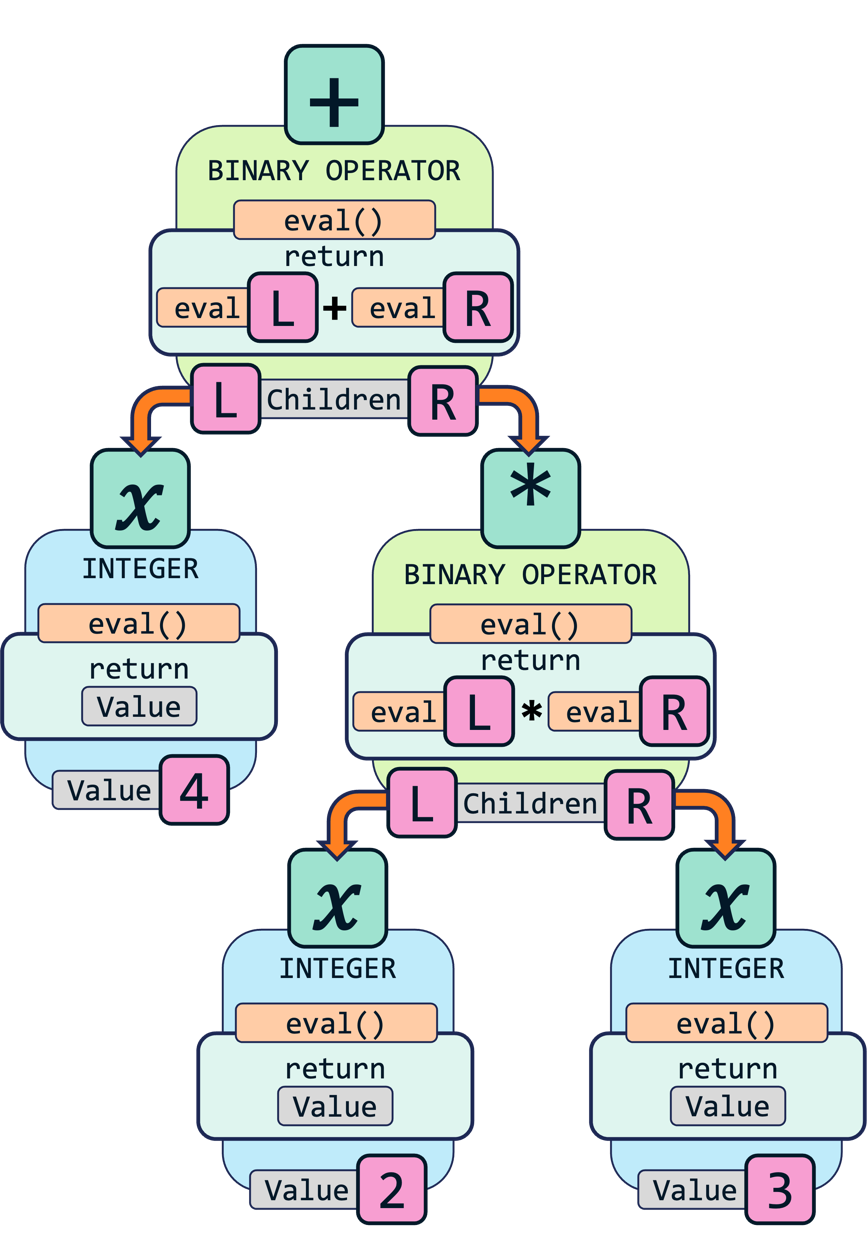
A diagram of a mathematical function

Description automatically generated with medium confidenceThe final stage is taking our abstract syntax tree and producing a single result. From now on, it is more useful to focus on the fact that the tree is a set of connected objects, rather than the very over-simplified diagram. Continuing from our small example earlier, we can represent the set of diagrams as shown:

This approach highlights a better way to visualise the tree, not just numbers connected by operations, but only integer objects, with their values being properties, rather than being the main part of the object itself/ Additionally, rather than them being magically “connected” to each other, the binary operator objects also have two properties: a left child property and a right child, which then each point to another different object. This focus on this object-oriented perspective will make the overall execution stage a lot easier.

There would be lots of different properties for the different types of objects that we could have. For example, any variable objects would have a property that points to the location where their value is stored. Or, the negate operator [for example –(3\*2)] would only have a single child, rather than a left and right, as it is not a binary operator.

Despite all of the objects having different properties, one of the things they would all share is an evaluation method, called eval(). This use of polymorphism – one of the pillarstones of object-oriented programming.

The evaluation function of each object returns it’s “value”. The most obvious instance of this is for an integer object, which returns the value of its value property as the evaluation method. Similarly, a variable would return the value stored in the memory location that its property points to. These are all very straightforward.

Where the complexity arises is with the binary operators. The result of the evaluation method of an addition object, is the result of the evaluation method of its left child, plus the result of the evaluation method of its right child.

This is relatively straightforward with just one binary operator, but when one of its children is another binary operation, its evaluation method has to call upon a further two evaluation methods in order to get it’s value, which it then returns to the previous.

A screenshot of a computer

Description automatically generatedThis means, that when you want to find the result of an expression, you call the evaluation method of the object at the top of the tree. In turn, this will then descend down the tree, calling the evaluation method of every “branch” object in the tree until the “leaf” objects are reached. The results are then returned back up the tree, so each method can obtain a value, back to the final starting object which returns the final answer

This is how the final evaluation stage of the interpreter. I will not fully explain our example, as it would merely be a lot of repetition, but the key focuses are the same. Descending down the tree, calling the evaluation method of each of its children, down to the bottom of the tree. Then, all the returns making their way back up to their parents, up to the top addition object, where the value of the equation has been determined.

Again, like lexing, this process is relatively straightforwards when you understand it, and it definitionly presents parsing as the most complicated step in the process, as it’s finding out the order of what to do that’s the most computationally complex.

Note that, in our simple example, the evaluation method only returns a value, but when we begin to impliment more complex features, such as seting variable values, iteration and function calls, this evaluation method will contain a lot more steps than merely returning a value, in order to actually change something in the code. But, for now, all we need to do is return values.

# Bodmas lexer

Now that the overview has been completed, I will go into more detail about how the lexer will function for our BODMAS example. Again, this is a foundation for the rest of the project, and I will therefore be designing it in a very open way which allows for expansion in the rest of the project.

Just to recap, the overall purpose of the lexer was to create a stream of tokens based on the plaintext input that it received, as we are using an object-oriented approach, that is how I will describe our code.

## Character cycling

Because a Lexer needs to be able to inspect a character at a time from the inputted text, the first stage is to create a way to go through each character on demand. However, the most important element to focus on at this stage, is to ensure that when the end of the input is reached, the program does not attempt to call the next character, as the index in the input it is looking for will not exist, so an error will be called.

I will approach this by creating a Lexer class, and in its constructor method – which is always called as a new instance of the class is created – I will take the text as an input – with the argument called input. The lexer will then assign different properties, creating an input property from the input argument, as well as creating a position property which is set to -1, and a character property which will be set as null for now. The constructor concludes by calling the continue() method.

The continue() method will ensure that the Lexer stays within the range of the input. It will begin by incrementing the position property by 1. It will then check if the position is equal to the length of the input: which would mean that the character at the position would not exist. If it is, it will set the character property to null. Otherwise, it will set the character to the character in the input at the position of the position property.

This allows the future part of the program to cycle through the characters until the character property is null, at which point it would know to stop. We could test this simply in the development phase in order to see if it would properly work by seeing if the string of characters is printed, and then stop printing with no error thrown when the end of the input is met.

## Token Definition

Before we can start creating tokens, we need to define al the tokens we need. At this point in the program, we will only have a small sample, but we will make sure we begin writing our code so we can expand upon it later in the process.

We will therefore create a new class called Token, with an empty constructor class. Again, this could be used for something later, so it is still worth implementing, but for now the Token class basically does nothing.

We will then create a new class for each type of Token that we will have at this stage. Each of these classes will inherit from the parent class of Token, so that later on, they all share the same properties that we will add to the Token class. So, we will create an Add, Minus, Multiply, Divide, LeftBracket, RightBracket, Integer and Float class, which are the basic tokens we will be working with for now. Each class will have a constructor method, which will call the super() method so that it will run the constructor method of the parent Token Class too, in order to make sure that we avoid code repetition in the future.

In addition to this, the Integer and Float classes require an extra property, called value, to be created when they are initiated. We will therefore add value as a parameter to the Integer and Float classes and set the value property to be the value parameter that they were created with.

At this point we can test if they work by manually creating a range of tokens.

## Basic token generation

The next step is creating a make\_tokens() method in the Lexer class. I will create an array called tokens, and then cycle through the characters until the character property is null. Then we need a way of checking which character the character property is. The cleanest way of doing this would be a switch statement, but when we need extra logic later down the line, as well as checking a large range of characters against the same token, then we are going to have to use a long chain of if statements.

This long chain, for now, will be checking if our character property against the corresponding character for each token. We will then add this token to the tokens array, and then call the continue() method before repeating the cycle.

Another important if statement is to consider what the interpreter must ignore. For now, the only thing we will be ignoring is empty space, so we will have an if statement which does nothing if a space is encountered, as it means nothing to the interpreter.

Finally, the default else clause will be executed if an unexpected character is encountered, which for now will just output an error message and then continue, before we implement proper error handling later on, as that as of itself is quite difficult and needs a lot of thought behind it to make it expandable.

Development

# Bodmas lexer

## Character cycling

A screen shot of a computer code

Description automatically generatedA screen shot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generatedA computer screen shot of a black screen

Description automatically generated

## Token Definition

A screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generated

A computer code on a black background

Description automatically generatedA screen shot of a computer code

Description automatically generated

A screen shot of a computer

Description automatically generated