Patrick Williams (0749)

Frobel learning (12144)  Private Candidate

ocr exam reference language interpreter

# contents

analysis

# project identification

In the OCR Computer Science GCSE, all programming code will be presented using the OCR Exam Reference Language (ERL). This is a language that does not actually exist, so cannot be used for actual programming tasks. Instead, this is their equivalent to a high-level programming language and is designed to have simple and pseudo-code like syntax to be widely understandable.

Especially in paper 2, it is vital that the students understand the language as they will need to be able to understand programs written in it, be able to fill-in-the-gaps using it in certain questions, as well as being given the option to write in it for the longer “Write” questions.

Unfortunately, students become familiar and comfortable with a language through usage, and the large majority of students lean towards using existing languages– primarily Python – throughout their paper due to a lack of usage prior to the exam. Therefore, to help benefit GCSE students, I intend to create some form of translator, which would allow students to write ERL code, and for the program to then translate and execute that code for them so that they can practice.

## issues with existing languages

A black background with colorful text

Description automatically generatedA black background with white text

Description automatically generatedOCR has attempted to design the papers in a way that ERL can be used interchangeably with other languages, however there are differences in the syntax which may lead to confusion and a potential loss of marks in the exam. I have listed out some examples below, outlining some of the common differences between ERL and primarily Python.

🡺 Inclusivity and Exclusivity

A screen shot of a computer code

Description automatically generatedWhen comparing the two code snippets on the right, they look very similar, but the ERL code will iterate once more than Python – the more familiar language – as its counter-controlled loops are inclusive rather than the typical exclusive.

This may seem unique to Pythons range() function, but in more standardly defined loops, such as in C, it is the equivalent of ERL using <= rather than <, which is expected, therefore potentially causing confusion during the exam .

🡺 Differing feature-sets

This, again, vastly changes depending on the language. For example, a Python user may be confused by the premise of a procedure compared to a function – because procedures do not exist in Python, as well as structures like switch statements not being frequently used (as match was only recently added to Python)

The worst instance of this, however, is with in-built methods. Only certain standard methods are allowed to be used in exams, primarily length and basic string manipulation methods.

However, if a student who is used to Python decides to use min(), max() or .sort() in their written example, then they will be awarded no marks for that section of the program. For example, if they are asked to write a program to return the largest item in an array, just using max() will not count as a valid solution, because they themselves have not written any code, and this attempted shortcut will not be accepted. I have seen this confusion before, as in my year 11 mocks multiple people did this, so it is important that students are educated on what is and isn’t allowed in an exam.

Being familiar with ERL, even if not using it for “Write” questions, will help educate students on which methods they can and cannot use within their exam to secure the best marks, because ERL represents the fundamental basics of what a student can use throughout the exam.

🡺 Small syntax changes

These changes may be unlikely lose the student marks in “Write” questions, but if the students are unfamiliar with them may cause confusion when reading ERL algorithms.

A black background with white text

Description automatically generatedA black background with green and orange text

Description automatically generatedFirstly, the typical parameters for the start and end position of a substring has been changed to the start and length in ERL, which is important to recognise.

Some other small syntax changes in ERL include:

* The use of closing keywords such as endif and endwhile (not frequently used in modern languages)
* The use of MOD and DIV rather than symbol equivalents which they may be used to
* The use of ^ as an exponent, compared to \*\* which is more frequently used.

🡺 Learning resources

Even though it could be argued that a high-level language could be used in the exam, the vast majority of textbooks use ERL in all their examples due to the fact that there are differences in all high-level languages, and it exists as a means to link together the GCSE students. Therefore, if a student is primarily learning how to code whilst using ERL, it makes sense to practice with it too.

## the solution

It is clear that the ERL is important to understand, so therefore a translator would help students complete practice questions written in ERL – and actually have a result to check and ensure that their solution works. Furthermore, it would help to transfer skills between different languages, as a semi-experienced programmer could test programs in ERL to see the transferability of their skillset into the exam. Finally, for complete beginners, it provides a way to run this very English-like language in order to introduce them properly to coding for the first time without the complexities of other languages.

It will also benefit teachers, as it will become much easier to explain the ERL to students and allow them to practice in lessons as it aligns with their teaching resources and is a much more logical approach to the exam. It would also be very useful for demonstration, as a key element of programming is the thought process behind it, so showing the class how to construct more complex algorithms – which at GCSE are generally sorting algorithms – alongside an actual result which also allows for easy debugging.

# stakeholders

There are two primary groups who would be utilising the solution: students and teachers, and there are generic characteristics for both groups.

|  |  |
| --- | --- |
| Students | Teachers |
| First time programmers with minimal experience | Knowledge of programming and ERL |
| Using restricted or low-performance devices | No restriction on software downloads |
| Difficulty installing complex software | No issues with installing software |
| Require an intuitive solution with help available | Still require basic error information |
| Would be helpful to include learning features | Solution must be presentable and teachable |

I will specify further with my stakeholders as to the exact details, but overall, a simpler solution seems to be ideal, as a translator which includes too many settings or features seems overly complex for new beginner programmers who are just trying to write very basic programs.

In terms of stakeholders, I have a current GCSE student who is the target user, a past GCSE student who has experience using ERL and revising with it, as well as a computer science teacher in order to see how a possible solution could be implemented into a classroom.

## beginner programmer

A person writing on paper

Description automatically generatedAbigail Williams is a student in year 10 who has been studying computer science in some form for 3 years but has only begun learning the OCR GCSE specific content this year. I asked her questions about her computing background.

🡺 What are your past experiences with programming?

A child typing on a computer

Description automatically generated“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.”

🡺 What’s your experience with the Exam Reference Language?

“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.”

**(After an explanation)** “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”.

🡺 What are your plans to revise coding?

A child pointing at a piece of paper

Description automatically generated“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.”

**🡺 Conclusions 🡸**

The key element of this is her very small amount of experience with programming. Although this may reduce how useful her feedback is during an iterative development process, it is important to consider that she is the main user of the program, so later on her ideas for usability should be very constructive.

## past gcse student

A child in a suit and tie sitting at a table with a computer

Description automatically generatedTanish 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. He is therefore a very high-achieving student, ideal for providing feedback about the process behind building the translator, however he is not a realistic end user as he is past the point of learning about ERL, and no longer has a need to use or practice it, but his experience will be useful.

🡺 Did you use ERL or a high-level language in the GCSE?

A person sitting at a table with his head on his face

Description automatically generated“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.”

🡺 How did you 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.”

A group of boys sitting at a table with a computer

Description automatically generated🡺 Did you 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.”

🡺 Would a translator for the ERL have aided revision?

A person in a suit using a computer

Description automatically generated“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”.

**🡺 Conclusions 🡸**

Tanish seems very comfortable with programming, so I feel like he will be very useful throughout the process, to iteratively provide feedback.

He does reinforce that this solution would be very useful for all students, even himself as a comfortable programmer, which helps to bring focus onto the translator. It is meant to be designed as an educational tool over a powerful machine, so making it friendly in design and clear to use will be very important.

## computer science teacher

D

# translator research

This section will comprise of three different sections: researching current solutions for running ERL code, other translator designs, and some initial research into translator design. Afterwards I will then go into more depth about features – what my stakeholders want included and researching what I feel is important.

## pre-existing solutions for erl

### A white sheet of paper with black text Description automatically generatedTrace Tables

A white sheet of paper 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 in this different context it operates slightly differently.

A white sheet with black text

Description automatically generated🡺 Advantages:  
+ Already understood by the student so is familiar  
+ Helps to check the program’s logic is correct  
+ Used as a debugging tool to track the values and easily identify errors  
+ No software required, very accessible to all students

🡺 Disadvantages:  
- No syntax checking, so invalid inputs are treated as correct  
- The end result is based upon the individuals understanding of ERL, so if they misuse a feature it will be executed based upon that flawed understanding, with no proper feedback on whether it is acceptable  
- Extremely slow progress to create, especially with larger programs and with nested loops, so is too inefficient to use regularly, especially when repeated run-throughs are required due to errors.

🡺 Conclusions:  
Trace tables are too unreliable and slow to ever be used as a translator equivalent, however the positive aspects of simplicity are important to consider. Furthermore, the accessibility is important, and this highlights that as we need a translator that will be available to a wide range of students.

Finally, this could be considered as a potential implementation to the translator, to provide some form of debugging tool which the students are used to, without some of the complexities of more capable debuggers. If it is within the overall scope of the program, this could therefore be considered.

### manual translation to other languages

This is not a very technically proper approach but was a method I used in class to avoid creating a time-consuming trace table to run through my program. As I was comfortable with Python, after creating my ERL code, I would go line by line and enter the Python equivalent into a file, then run it to create my result, which would then allow me to check if the logic behind my program was correct.

🡺 Advantages:  
+ The computer produces a result at the end, therefore checking the logic behind the program  
+ Develops skills in ERL and a high-level language, providing the student with some real life skills  
+ Re-reading their own ERL is a form of checking for errors whilst translating to the other language

🡺 Disadvantages:  
- The ERL stage becomes very redundant, with not benefit to even including it when a high-level language can be used in the exam anyway, with no short-term benefit to writing the program out twice  
- Similarly to the trace tables, the student is the translator, so if their knowledge of the syntax or logic behind ERL is incorrect, then their translated program will not be accurate either, which defeats the purpose  
- Too time consuming, essentially re-writing the program is painful

🡺 Conclusions:  
Another poor current solution to translating ERL and should never really be used in a classroom. This highlights that the solution must be quick to use, as otherwise a student would never choose to use it. The link to real high-level languages could be considered, with a potential feature translating the ERL code to Python to link the two, however that may be too far out of the scope as is not the core feature.

### in conclusion

Overall, a translator for ERL seems very useful as there are currently no quick and easy solutions to the problem. In terms of focusing the project, it seems better to lean towards an Interpreter: the program gets executed on a click of a button rather than creating separate executable files. Whether the program approaches this line by line or the entirely at once is unimportant currently, as it will be abstracted to the user, but ensuring the program is easy and logical to use is critical.

## translators

Interpreters are usually designed around experienced programmers, which is not ideal for our solution, so despite these interpreters being very well made, I will be analysing them based on their use for new programmers – such as GCSE students.

### A computer screen shot of a code Description automatically generatednode.js in the terminal

This is a command line interpreter for JavaScript which can be called by “node” to launch the program.

A screen shot of a computer

Description automatically generated🡺 Advantages:  
+ Good error handling, with detailed messages pointing to the position of the error in the line – useful for beginners  
+ Multiple use modes including the shell (useful for a few lines), an editor (useful for questions), and a feature to run local files (useful for any larger projects) – making it good in a classroom.  
+ Has a basic .help feature which it tells the user about, providing access to the useful features which is needed.

A screenshot of a computer error

Description automatically generatedA screenshot of a computer

Description automatically generated🡺 Disadvantages:  
- No graphical user interface, making it difficult to use for new programmers who need something intuitive.  
- The terminal is often blocked on school devices  
- Complicated installation may not be feasible for students at home, and would require administrator access at school  
- Must memorise commands for fluent usage, tedious for beginners.

🡺 Conclusions:

Overall, I feel there are 3 main takeaways from node.

* Ensure that proper error handling is implemented, with good detail and pointing to the position.
* Make sure the interface is intuitive and is targeted towards beginner students.
* Prioritise accessibility, avoiding complex installs in the final product.

### Python idle

A screenshot of a phone

Description automatically generatedMy points will build upon the previous, expanding upon the features it includes.

A screenshot of a computer program

Description automatically generatedA screen shot of a computer

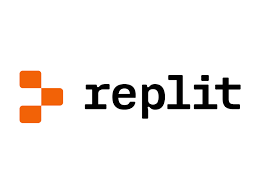
Description automatically generated🡺 Advantages:  
+ A vastly improved user experience, using a ribbon to implement the functionality, which is much more intuitive than commands for beginners.  
+ A much nice user interface, with syntax highlighting and proper indentation (which will also be important for ERL), which is much more pleasant than the command-line implementation.  
+ A greater array of features with “a debugger, with persistent breakpoints, stepping, and viewing of global and local namespaces” – which could be of use to a student

A screenshot of a computer

Description automatically generated🡺 Disadvantages:  
- As useful as some of these features are, they will never really be used by absolute beginners due to their complexity, so may be redundant.  
- An installation is still required, which may reduce accessibility  
- The official documentation is dense and hard to understand. There are still countless Python tutorials on the internet – but in terms of an official equivalent for ERL – an easier solution would be required.  
- Having to save files in the editor before running can be annoying and complicated.

🡺 Conclusions:  
Visually Python IDLE should be a strong influence, as its focus on the code is important. In terms of improvement, just further decreasing the complexity would be good, such as by removing installation and adding more buttons to the project to make it even simpler.

### Replit

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

🡺 Advantages:  
+ Very accessible solution, and a website seems like the best solution so that it is available to all students with an internet connection without any install.  
+ The cloud servers ensure that all low-performance devices can still run code, therefore making it available to all students and very suitable for school devices  
+ When talking to a student who has used it before, Dylan Matthews, he said the main positive was the community aspect, with lots of users sharing their code and problems creating a learning environment.

A screenshot of a computer

Description automatically generated

🡺 Disadvantages:  
- Although there are some good aspects of the interface, such as the run button and separate editor and shell windows, overall, it feels very cluttered and overwhelming for a beginner.  
- Having account systems and being required to sign in is also overcomplicated, as ERL questions would only require short algorithms to be written, not requiring saving, and creating an account just slows the whole process.

🡺 Conclusions:  
As great as running the code on cloud servers would be for my project, this is beyond the scope of the project, and would be largely redundant due to the simple ERL code. However, a website seems like a promising idea, and I would therefore like to take this forward as it seems perfect for students at school.

## Translator design

This is split into two sections: ERL specific and generally for translators.

### For Exam reference language

It is important to recognise the entire set of instructions that need to be implemented, which are available on the exam specification at the following link:   
<https://www.ocr.org.uk/Images/558027-specification-gcse-computer-science-j277.pdf#page=27>

It is going to be very important to perfectly follow what this outline, as it is the requirement for our program. It is not super detailed, and some judgement will have to be used throughout to make decisions on how certain aspects will operate. Overall, it is very Pythonic in nature, so I will most likely be taking inspiration from how Python handles certain inputs to make decisions for my Interpreter.

There are some important things to consider with the ERL syntax:

🡺 Uses closing keywords to bound structures, such as endwhile and endfunction  
- This is different to other languages that use curly brackets {}, or Python which uses indentation.  
- As indentation is not required, it should still be encouraged for ease of reading.

🡺 Very inconsistent syntax when defining structures  
- if and elseif require “then” after the conditions  
- else and while loops require nothing after their condition  
- switch statements need colons after each case.

Overall, ERL has some unexpected syntax in places, so it will be very important to constantly refer back to the document to ensure that it matches the definitions exactly.

### general translator design

When researching into translator design, the starting point for all designs was building a calculator, because it comprises of the basics for the whole system.

For this, I spent a long time using online resources to find how to construct this basic calculator framework, and I found that a few resources explained this the best.

The most useful one of these was at the following link:  
<https://www.youtube.com/watch?v=SToUyjAsaFk>  
This provided a lot of insight into how a potential interpreter would function, and introduced be to the basics of the design. Two other useful resources were as follows:  
<https://www.youtube.com/watch?v=JO_0e9mPofY>  
<https://craftinginterpreters.com/a-map-of-the-territory.html>

Apart from some small google searches, these were the 3 main resources I used to research into how an interpreter should be designed, which I am going to summarise to explain my research.

# an overview of interpreting

This will be a demonstration of how I understand interpreting to work. There are lots of different types of interpreters, however I found this one to be the most simple and straightforward for me to understand, which makes it best for this project. This section, as well as being an explanation of my research, is also somewhat a basic plan for when I will implement arithmetic into my interpreter.

Overall, there are 3 main stages in interpreting: lexical analysis, parsing and execution. Here, I will provide an overview of the system, based off my research, in order to lay out how my interpreter will function.

## 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.

## The basic concept of parsing

A screenshot of a computer

Description automatically generatedParsing 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.

A diagram of a mathematical system

Description automatically generated with medium confidenceThe 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.

# stakeholder needs

Before confirming the essential features of the program, I will first talk to my different stakeholders to figure out which aspects of the program they feel are most critical to include.

## current gcse student

I started off by asking Abigail about her needs and wants.

🡺 What do you are the most important features that must be implemented in the solution?

*“For it to work, but that’s self-explanatory. Two, I would appreciate a feature that could correct your wrong answer and help you to fix it if you are stuck. It also needs to contain all the different bits of pseudocode that I need to know.”*

🡺 Are there any non-necessary feature you feel could be added?

*“If there was a way where it could give you different questions that you needed to solve, with different difficulty options. Some kind of streak feature, like Duolingo, might make me want to practice a bit more, maybe with a leader board. A help page would be useful, with links to external videos to help when you get stuck.”*

🡺 What do you think is important to make the program easy to use?

*“Not an overwhelming amount of information. Some kind of introduction when first visiting the website would be good, maybe a video or tutorial. For loops, if there was an option for coloured brackets, to help show where a loop begins and ends. Also, making it touchscreen and work on phones and stuff is good.”*

🡺 What are the two most useful help features that could be added?

*“An option that shows you all the different pseudocode bits, because now I don’t know them all, and instructions on how to use them would be great. Also, when you make a mistake in the code, it telling you exactly what went wrong and how to fix it is also very important”.*

### Conclusions

Overall, the priority is still having all the features. As good as some of her possible suggestions might be, such as a streak or leaderboard system, I think they are outside of the scope of the program because it very far away from the key concepts in the rest of the program.

In her most important features, she did say that it working and all the pseudocode instructions being were critical to the program, so I will mainly be focusing on those, as implementing a working Interpreter will take up the majority of development time. Her other point was to do with help features which she mentioned a lot. Even though she mentioned linking to other websites, I feel like I will primarily focus on having very clear and useful error messages, and some kind of final documentation to explain the ERL features, however including these was already kind of on my mind.

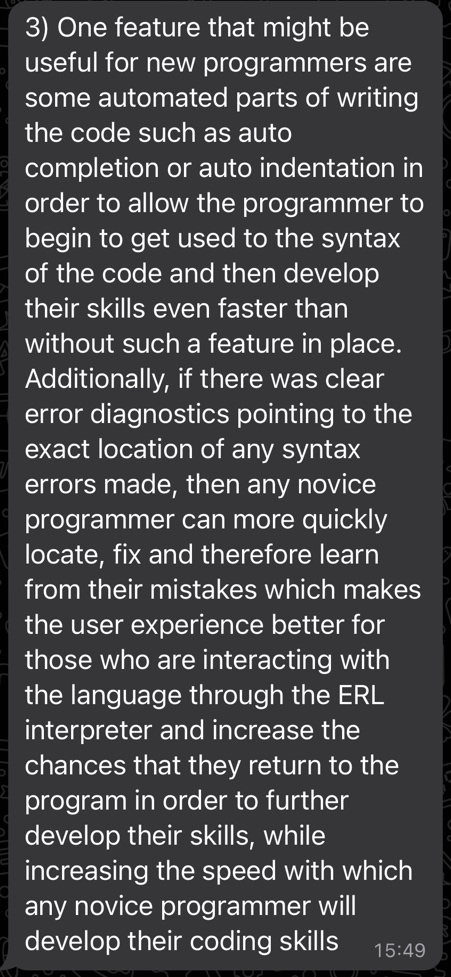
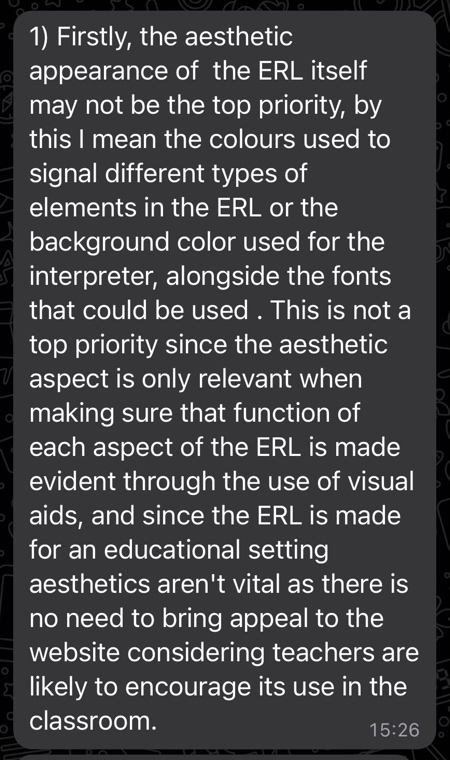
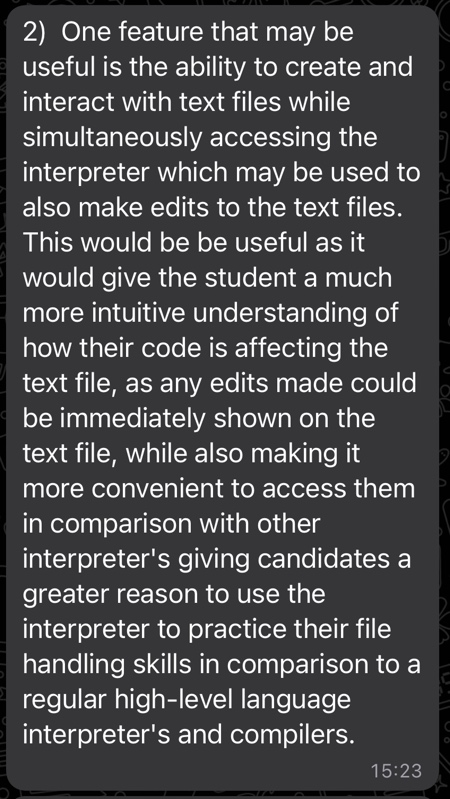
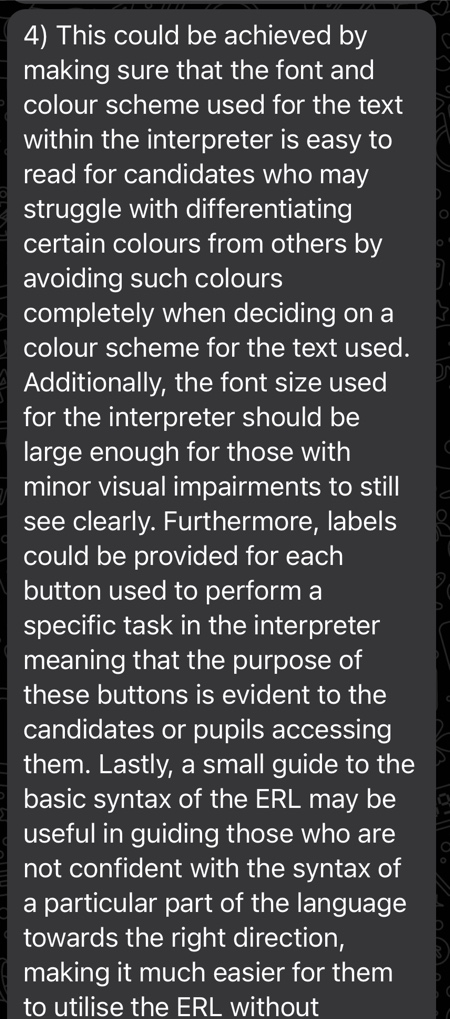
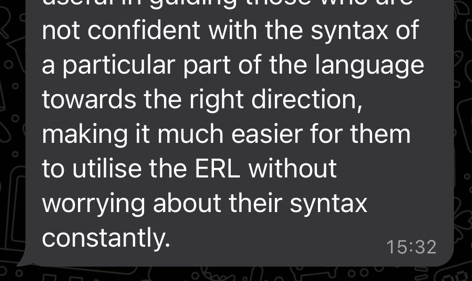
In terms of new features, her three main suggestions were syntax highlighting, it giving you questions to practice and a tutorial feature to know how the interface works. I’m hoping to create such an intuitive and basic interface that it won’t require a tutorial, so I will not carry that idea forward. However, the other two sound like very promising ideas. They won’t be at the forefront of development but could be additional features that are added at the end which sound very positive.

They still may be outside of the scope however, so overall the focus is still on implementing the Interpreter properly, and adding other bonus features last, as she agreed that they were the most important and required elements.

## past gcse student

Tanish has a more in depth understanding of the ERL and computing I can ask him some more specific questions about the ERL features.

🡺 Out of all the ERL features, are there any you think are a lower priority to add than the rest?  
🡺 Excluding the Interpreter functioning, are there any features you think may be useful?  
🡺 What help features do you think could be useful for new programmers?  
🡺 How do you think the program could be made as intuitive and easy to use as possible?



Overall, a lot of Tanish’s points were about the user interface, instead of the actual interpreter features itself and what it should include.

This is because overall, the main criteria for this project are very concrete: it is implied that an ERL interpreter must contain all the features.

My first question was asking Tanish if there were any of the ERL aspects I could potentially not implement, in order to make development easier, however the fact that he didn’t consider this at all shows that they are all very neccecary, and he does state in the first answer that the appearance does not matter, but the functionality should take priority over any additional features.

However, the other suggestions Tanish made for the interface were as follows:

* Built in file handling in the interface, to make changes to files more tangible.
* Auto-completion and indentation to help get used to writing ERL.
* Clear error diagnostics which help students to fix issues.
* Font size, clear colours, and labels for accessibility reasons
* A syntax guide for the ERL (like the specification)

Overall, I feel that these are all valuable points, and a lot of them align with my own additional intentions for the program. I will consolidate which of these I bring forwards

## my potential ideas

The number of suggestions so far by the stakeholders has been a lot, so with my ideas I’d like to mainly focus on the core aspect of the project: ensuring that ERL is correctly ran, and diving into some of the details behind that rather than additional features.

Of course, the majority of the project’s work will consist of implementing all of the features in the interpreter, but ensuring they integrate seamlessly with how the user experiences them is important. For example, for the file handling to correctly function, the user will need to be able to upload files and be able to visualise the changes to the file after the program runs, as otherwise adding file handling features is useless if they cannot actually be used.

Therefore, I feel the final product should have a file management interface as it should make it a lot easier for students to see the impact of their file handling.

### my extra features

The only other feature I could think of being feasible is a potential trace-table debug feature, as when researching into ways of running ERL I feel like it could be a good tool to build into the program.

Not only would it allow students to check the values of their values to help with debugging any logical errors in their program, it also seamlessly integrates with different types of questions in the GCSE, as students can be asked to make trace tables of an algorithm, which would be written in ERL, so therefore familiarising them with how this process works would be very useful.

Apart from this, I largely agree with my stakeholder suggestions, and incorporating syntax highlighting and other quality-of-life features is something I would aim to include regardless, just because it should make the experience of writing code a lot better, even if it is just with the interface.

# the computational side

Now I will be discussing the computational side of the analysis section, including the computational methods to use, the hardware and software requirements and the plans for the programming languages and any libraries I may use.

## Computational approach

My task is a perfect example of a task that is suited for computers, because executing a programming language, and code in some format, is the entire purpose of a computer.

As discussed, when researching, this can be done by hand, which is a very slow and error-prone process, which can easily be fixed by a computer: providing almost instant feedback with correct results every time, being able to be used in a variety of different cases.

Alongside this, there is no way to manage inputs when attempting to dry-run ERL by hand, so this system will also allow for different inputs to be quickly tested on the system which is again a lot easier than having to re-run the system for another time. All of these negatives have been discussed when researching other solutions to the ERL, but a computer is definitely needed to make the process of executing ERL more efficient.

More specifically, the nature of the code following sequential, logical steps, with quick but pre-defined operations needing to be followed, this approach is perfect for this issue.

### computational methods

Overall, there are a number of computational methods that I will use in this project.

**🡺 Abstraction**

The entire program is one, large abstraction for the user. They are merely presented with an input window and an output window, and they do not have to consider at all how the ERL is executed in the background. This allows the user to focus all of their time into writing valid ERL, and practicing their coding abilities, and providing this huge layer of abstraction between visualising their results and the process to make them is central to the entire program and making it as simple as possible for them to learn.

In terms of abstraction whilst used in the interface, there will be a large amount of this used. As described in my brief overview of how an interpreter functions, each operator and branch in the abstract syntax tree will call eval() on both of its children, not caring about what they actually do when executing, but ensuring that the entire program will be evaluated.

Although the program will become a lot more complex than the arithmetic example, it essentially is still forming lots of these big abstract syntax trees, and calling eval() on the top, which will then call eval() on every single other item in the tree, executing the program. There will be more examples of this when I begin the coding, but lots of abstracting will be used whilst coding the project as it is in its nature.

**🡺 Divide and Conquer**

Likewise, as also discussed in my overview of interpreters, the code will break down any plaintext input into tokens, and then will repeatedly arrange them into <expr>s, <term>s and <factor>s, and this process is decomposing the input into their smaller components: starting by grouping each number in the arithmetic, then grouping each multiplication or division, and then grouping the addition and subtraction to formulate the entire expression in the tree.

When increasing in complexity from here, parsing will still be this whole process of decomposing the problem. When originally being presented with the entire file, it will then have to decompose it into separate structures, such as if statements and while loops, and will then decompose these components, such as the condition of a while loop, which it will further decompose, being a comparison between two expressions, which it will decompose down into the arithmetic until it reaches the fundamental building block: being a number or variable or any other literal.

This shows how divide and conquer is intertwined with recursive descent parsing, so therefore this computational method will be used significantly in the process.

**🡺 Pipelining and Concurrency**

There is potential for these methods to be used in the interpreter: for example, one line of instructions could be undergoing lexical analysis whilst the previous is being parsed, whilst the one before is being evaluated. The likely issue is that the programs will be so simple that these performance increases would be too marginal to notice, because ERL is intended for basic, school algorithms instead of large-scale industrial coding, but I may consider it when programming the project, but it is not a priority.

However, issues may arise when user inputs are required, as concurrency breaks down when the program needs to await for inputs from a user, as pipelining is best used for a constant stream of inputs, however, again, I will consider this later in development, keeping the option of pipelining open if possible.

**🡺 Other methods**

Apart from these, I cannot think of how any of the other methods I have studied could be used with the project, but abstraction and divide and conquer will be used in such significant amounts throughout, because it is core to the entire premise of the project, so the program will still use the methods massively.

## the software side

When researching, the concept of having the final product be available on a website was very promising, as it allows it to be easily accessible to students from an array of different devices, without having to be concerned about installing different programs without permissions or experience.

Immediately, this points to me having to use HTML and CSS for creating the interface, and the language that interacts with this best is JavaScript. This is the language of the internet, and is run by all web browsers, so there will be no issues with compatibility. The language is also optimised to run on many devices, which should make it much easier for all students to access.

Therefore, I will be coding this language in JavaScript. Because the interface will be added last, I will be using Node JS to run JavaScript on my computer, in order to use the prototype versions of the interpreter without needing a full interface to be implemented.

I will be using completely pure JavaScript for this project and have decided to avoid using TypeScript or any frameworks for the project – mainly so that it counts as a valid language – but also to avoid a very steep learning curve with lots of additional features that I will n.ot need, as overall the project should be possible to complete with Object Oriented Programming.

Speaking of OOP, JavaScript is an OOP language, which is very important for my interpreter as I will be using lots of classes for all of the different elements, and polymorphism is very important to the interpreter functioning, so it is suitable for this approach.

### additional development requirements

Throughout, I will be using VSCode as my IDE for the project, because I am very familiar with it, and will be switching between macOS and Windows operating systems throughout. Overall, though, development will mainly occur inside a single JS file, so it is not very demanding to develop this project, and so long as NodeJS is installed, it is a complex project rather than a computationally demanding one so I am not concerned about the computational power of my machine.

## system requirements for the end user

My website is going to be designed for low-end computers, such as the ones found in schools. This will require a decent web-browser, so I will end up not supporting early versions of Internet Explorer due to the older versions of JavaScript which I do not want to account for, however, any device running Google Chrome, Firefox, Safari or any other up-to-date browser should have no issues.

The benefits of having a website-based program is that it will be accessible to almost everybody, because everyone will have a web browser installed, and not requiring to need any additional software is perfect for any device with restrictions or low secondary storage.

Additionally, an Interpreter should not be a very computationally demanding program, as in the worst-case scenario, any extremely low-end devices will just experience slower run-times. However, because ERL programs are always extremely simple in nature, compared to the complexities of loading a web page and running an OS on a computer, I am not concerned with the small algorithms that need to be run.

If larger algorithms are ran, then there may be some performance issues, because the Interpreter will not be extremely optimised for super-fast processing, but mainly ensuring it provides positive error feedback throughout, and for a web-based interpreter this is not expected. However, large programs are not the primary intention of the program, so this issue is unimportant.

Overall, the program is intended for these low-end, school and restricted devices, so during development I will ensure that it is target towards this success

criteria

# essential features

Overall, the essential features are very set in stone for this project, because OCR has documentation for the syntax of the language in their specification, at the following link:  
<https://www.ocr.org.uk/Images/558027-specification-gcse-computer-science-j277.pdf#page=27>

To decompose the project slightly, I have split the interpreter side into 11 different modules, each containing the different elements of the ERL which can be included. This is ordered based on their importance within the program, as well as the later modules being based off the framework of the previous, so I will implement them in this order during development.

1. Arithmetic: allowing for mathematical expressions to be evaluated.
2. Variables: storing values into variables to be read and rewritten in the code.
3. Logic: adding Booleans, comparisons, and logical operators to evaluate conditions.
4. If statements: including if, elseif and else cases.
5. Loops: adding while, do until and for loops.
6. Strings: adding the string data type, with concatenation and basic type casting.
7. Subroutines: allowing functions and procedures to be created and run.
8. Native functions: adding print(), input() alongside the other default functions.
9. Switch and Arrays: adding the switch structure and mutable arrays.
10. String methods: adding the different properties to allow strings to be manipulated.
11. File handling: allowing files to be created, opened, read, and written to.

Overall, it is vital that these are all implemented exactly how the specification states, in order for it to be a useful tool for students so that it perfectly aligns with the exam questions. I will further break down what must be included when I get to the success criteria, however now I am taking a more general view of the features, but ensuring I follow the syntax perfectly is extremely important.

### justifying what is essential

I feel that modules 1 to 9 are absolutely essential to this project, and without them the project would be a failure, because they are the core features that make up the language. My stakeholders agreed that the entire purpose of the project is to have a complete implementation of the language, and none of the additional features or options can be added if the base interpreter is not included.

Therefore, having the core features of the interpreter implemented is the absolute priority, because otherwise the program would be useless to GCSE students as it would not fulfil its purpose and they would not be able to practice with such a narrow set of instructions to use.

Ideally, I would like to also include 10 and 11, however I accept that these features are less commonly used by students in the exam, so they can be discarded if necessary due to time restraints later on in the program. However, they should still be included if possible, and definitely added before any other quality of life features, as they would complete all of the ERL features, however the program would still be useable without them because the majority of use-cases are accounted for with the first 9 modules.

## non-interpreter features

The Interpreter side of the project will be the majority of the coding, because it will be very computationally demanding, however there are additional features that need to be included.

My stakeholders largely mentioned these additional features when prompting them for potential features, so it is clear that implementing as many additional features will be very useful.

Overall, though, the Interpreter will still be the most important aspect, so incorporating all of the elements is the most essential, however I have broken down the list of suggestions from my stakeholders into the features that I wish to include in my final project.

**🡺 User Interface**

This is the most important feature, potentially of the whole project, as even if I have a working Interpreter, not having an accessible interface will mean that it cannot be used.

This was heavily researched in the interpreter design stage, but having an interface which students can easily use, with clear input and output fields is absolutely essential. As decided previously, the interpreter will be implemented into a website for the best accessibility, so this will need to be formatted so that code can easily be ran on school devices.

**🡺 Error Handling**

An aspect that was frequently mentioned by the users was debugging features. The most basic, and most important version of this is having clear error messages. Because students will not be experienced with the ERL’s syntax, and will be used to other languages, they will write code incorrectly a lot of the time, largely the syntax required for the different structures in ERL.

Therefore, having very descriptive error messages is extremely important, so that when they run an invalid program, they can easily fix it and test their program. The overall purpose of this interpreter is for students to learn how to write ERL, and if they do not get good feedback which can help sort out the issues, then the program will be very annoying to use.

This will be implemented throughout the whole development process, but mainly will incorporate adding lots of different cases for each invalid input and returning a different message for each of these cases. I am not yet sure how I will implement this, but a focus on errors is important.

**🡺 Clear Documentation**

This is not a computational aspect, but having clear instructions on how to write ERL, alongside the specification will be very important, as students will not be exactly sure how to write it. This will just require a document to be made that outlines the instructions, but will be very useful to students.

**🡺 Syntax Highlighting**

Both stakeholders have specifically mentioned this feature, as he said it would be a lot easier to write and read different sections of code. Although advanced highlighting does exist in some IDEs, such as Visual Studio Code where errors will be highlighted in advance and known variables will be coloured differently, this will largely be outside of the scope of the program.

Therefore, I aim to colour the different keywords differently, to increase how digestible it is to write the code, as otherwise staring at a block of text will not be very easy for a user.

**🡺 Exam Questions**

Finally, the last feature I would like to take forwards is Exam Questions. The basics of this would be a list of questions, with answers that could be viewed afterwards. These would be categorised into different difficulties and would largely be taken from past exam papers. The purpose of this is to provide some motivation for students to actually practice ERL by making it a lot easier.

This could be further expanded to automatically test the user’s input with a range of test data to provide live feedback, however I will most likely decide this later as it will be the last feature to be added.

# limitations

If possible, I would like to completely avoid any limitations in the actual Interpreter design itself. This is because having the complete set of instructions is critical to the program being usable.

The only possible limitations may occur during development if I cannot implement a certain feature. For example, if I find recursion too challenging to add to the program, I could remove the ability to use it, because GCSE students will not be required to use recursion in any questions because it is too complex.

Ideally, I would still like it to be incorporated, so I am not planning any limitations on the design of the core interpreter, but they may arise, so long as they do not fundamentally ruin the program.

### quality of life features

Mainly I will be including limitations in my additional features. I have already discussed Syntax Highlighting, but to recap, adding additional intelligent checks before runtime, such as scanning for errors, will be outside the scope of the program, so I will not be incorporating them. Even though it would make it easier for a student to program, the already included error messages should allow students to fix any errors, so it is not critical to be added.

Likewise, Abigail’s suggestions to have the Interpreter make suggestions on how to fix any issues with the code is massively outside the scope of the program. Although some basic versions could be added, such as custom error message after if statements when a colon is used instead of then, having the interpreter be able to fix these issues by itself is too complex for this project, so will not be included.

I love the idea of a trace-table debugging feature, as it seems very easy to use and is very intertwined with the GCSE itself, however it would also be outside of the program scope. However, it could possibly be a potential option for any future upgrades to the system because its functionality would be very useful. Currently though, it is not important to the core purpose of the project, so will not be included.

Finally, a lot of the additional suggestions are far too outside of the scope to be considered in my opinion. Largely, adding a streak feature with accounts and leaderboards detracts from the main purpose: having a way to practice writing ERL. Although they may be motivational, I will not be including them in this iteration of the project as they would require a lot of additional coding for not much benefit to the core need.

# success criteria

Now for a more formal breakdown of what must be included. This will largely be a detail of the Exam Reference Language code that must be added, which will be based off of the specification link. I will not individually justify every single aspect, because they all share the same general purpose: all the features are needed to produce a program that can consistently, correctly run any ERL script that a student makes, in order for them to properly practice and test their skills.

### the format

Each section of the success criteria will contain a table, laying out the features that need to be included, as well as some examples of how to test these. These examples will be very basic, trying to only include the features that the new module includes, and more integrated tests will be added at the end.

However, the tests are designed for the end of development. This means that, when trying to run them at the end of each module, they may contain features that do not exist yet: mainly print(), which will be incorporate in the later stages of development.

Therefore, some modification will be required in order to correctly attempt each of these test cases during development, but the overall idea behind them should be the same.

## 1: arithmetic

A white rectangular box with black text

Description automatically generatedThe first module will be allowing mathematical expressions to be evaluated, following the BODMAS order of operations. Although sounding simple, this will build the framework for the whole program so is very important to include.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case |
| 1.1 | Allow for Addition, Subtraction, Multiplication, Division, Exponentiation, Modulus and Quotient division to be taken of two Integers | print( 2 + 3 ) 🡺 5 print( 3 \* 7 ) 🡺 21 |
| 1.2 | Allow multiple operations to be chained together, following the BODMAS order of operations. | print( 2 + 3 \* 4 ) 🡺 14 print(10 – 2 ^ 3 ) 🡺 8 |
| 1.3 | Allow for Floats to be used and created, and for any calculation involving Floats to return another Float | print( 9 / 2 ) 🡺 4.5 print( 2.0 + 3.0 ) 🡺 5.0 |
| 1.4 | Allow brackets and infix operators + and – to be used in expressions | print ( (-3 \* 4) + 2 ) 🡺 -10 |
| 1.5 | Proper error handling is implemented, ensuring division by zero, unexpected characters and invalid syntax errors are managed | 5 / 0 🡺 Error 3 \* / $ 🡺 Error |

## 2: variables

This module will allow values to be stored in variables using identifiers, to later be used inside expressions.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case |
| 2.1 | Allow for identifiers to be used to store values, defined with the ‘=’ syntax, which can later be read. | variable = 7 print(variable) 🡺 7 |
| 2.2 | Ensure the new variable system integrates with the existing arithmetic, for both setting values and inside expressions. | variable = 2 \* 8 print( variable – 5 ) 🡺 11 |
| 2.3 | Let constants be created with the “const” keyword, which are variables that will return errors if the user attempts to redefine them | const pi = 3.14 pi = 7 🡺 Error |
| 2.4 | Ensure full error handling, so that variables cannot be accessed before assignment, and that all invalid cases are accounted for. | print( var ) 🡺 Error var = 🡺 Error |

## 3: logic

A screenshot of a comparison operator

Description automatically generatedThis section adds the Boolean data type, and allows for comparisons of different expressions using comparisons, as well Boolean operators to form logical expressions.

Overall, this is largely a framework stage for the rest of the program, as it will be needed for the conditions in loops and if statements in the future.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case |
| 3.1 | Allow for Booleans to be created with “True” or “False” | n/a |
| 3.2 | Let the comparisons ==, !=, >, >=, <, and <= be used between two expressions, Booleans or values and return a Boolean | print( 3 + 5 > 2 ) 🡺 True print(True != False) 🡺 True |
| 3.3 | Let NOT be used to negate comparisons and Booleans, with a lower precedence than comparison expressions. | print( NOT 3 > 5 ) 🡺 True print( NOT False) 🡺 True |
| 3.4 | Let AND and OR chain multiple different expressions together, with a lower precedence than NOT. | print(True AND False) 🡺 False print(NOT False AND True) 🡺 True |

## 4: if statements

A screen shot of a computer

Description automatically generatedThen, if statement structures will be added, with a focus on ensuring that the syntax laid out in the specification is perfectly matched due to the inconsistencies in it.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case |
| 4.1 | Allow if statements to be created by the “if” keyword, followed by a condition and “then”. Then, on a newline, all of the contents, followed by a final, closing “endif” keyword. | if 2 + 2 == 4 then  print(“Passed”) endif 🡺 Passed |
| 4.2 | Allow additional cases to be added with the “elseif” keyword, followed by a condition and “then”, with the corresponding code, followed by more branches and finally “endif” | if False then  print(“Fail”) elseif 5 > 3 then  print(“Passed”) endif 🡺 Passed |
| 4.3 | Allow a default case to be made by the “else” keyword alone on a line, then followed by its contents and then “endif” with no other cases, which will be run if no other conditions are. | if True == False then  print(“Fail”) else  print(“Passed”) endif 🡺 Passed |
| 4.4 | Ensure that the cases are always laid out in the correct order, of “if”, then optional “elseif”, then optional “else” then “endif”, to ensure the syntax is maintained throughout | elseif 4 > 5 then 🡺 Error endif 🡺 Error |
| 4.5 | Ensure that descriptive error messages are returned when the syntax is not followed, with details about the valid conditions, and ensuring that conditions are always valid | if True: 🡺 Error if 2 then 🡺 Error else then 🡺 Error |

## 5: loops

This involves adding the 3 different types of loops, as outlined by the specification. These all have very specific outlines of how they function, so following the specification which is outlined on the next page is very important to follow, especially for “for” loops because they are very specific.

A screenshot of a computer program

Description automatically generated

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case |
| 5.1 | Allow while loops to be created by “while”, then a condition, then the contents of the loop on the next line, and closed by the “endwhile” keyword | counter = 1 while counter < 3  print(“A”)  counter = counter + 1 endwhile print(counter) 🡺 A, A, 3 |
| 5.2 | Ensure that the contents will be evaluated before the loop is run, and it will iterate while the condition is True | while False  print(“Fail”) endwhile 🡺 No output |
| 5.3 | Allow a do until loop to be created by “do”, followed by the contents, and being closed by “until” then the condition of the loop | var = 1 do  var = var – 1  print(“B”) until var == 0 print(var) 🡺 B, 0 |
| 5.4 | Ensure that the loop is run once before the condition is checked, and from there it iterates while the loop is False | do  print(“C”) until True 🡺 C |
| 5.5 | Allow for loops to be created by “for”, then an identifier, then “=”, then a starting value or expression, then “to”, followed by a final value or expression. Then, the contents, closed by “next”, followed by the same identifier that the for loop was defined with | for i = 1 to 3  print(i) next i 🡺 1, 2, 3 |
| 5.6 | Ensure that, on each iteration, the variable with the identifier name is set to the new value, and that the loop will iterate until the value is greater than the final value, so it is inclusive | n/a (see above) |
| 5.7 | Allow an optional “step” keyword to follow the final value, which must be followed by a value or expression to represent how much the value will increment on each iteration. This can also be a negative value; in which case the loop iterates until the value is lower than the final value. | for i = 5 to 1 step -2  print(i) next i 🡺 5, 3, 1 |
| 5.8 | Ensure that all values in the for loop are integers, as well as full error handling for all invalid cases for conditions for the other loops, such as non-Booleans in conditions, with descriptive error messages. | for var= 0.3 to 2.3 🡺 Error while 3 🡺 Error do True 🡺 Error |

## 6: strings and casting

This smaller module adds a string data types, and general upgrades to types.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case |
| 6.1 | Add a string data type, which can be defined with single or double quotations, and concatenated with + | print(“test”) 🡺 test print(“a” + “b”) 🡺 ab |
| 6.2 | Ensure that the whole system has tighter type handling, as now that operators have different uses across different types, different cases will be required so that invalid cases are rejected | 2 + “test” 🡺 Error “hello” \* 3 🡺 Error |
| 6.3 | Allow for type casting between the 4 primitive datatypes, with errors in some illegal cases | print( int(“7”) + 1 ) 🡺 8 float(“test”) 🡺 Error |

I will decide when planning this section exactly how I want the type casting to occur, as the specification is not very specific, and moreso just outlines the names of the functions to be used.

## 7: functions and procedures

A screenshot of a computer program

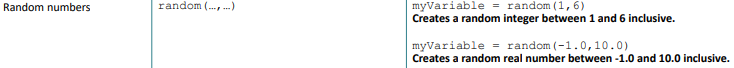
Description automatically generatedI am expecting this module to be the most complex, as it requires lots of different features to be added, alongside handling different scopes which will likely not be very easy.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case |
| 7.1 | Allow a procedure to be defined by “procedure”, followed by the identifier to represent it, followed by brackets () containing an optional list of parameter names. Then, the contents of the procedure, closed by the “endprocedure” keyword | procedure square(x)  print( x^2 ) endprocedure |
| 7.2 | Allow subroutines to be called by their identifier after definition, followed by brackets containing the arguments | square(7) 🡺 49 |
| 7.3 | Ensure that a new variable scope is created whenever a subroutine is run, with the parameters being assigned as variables in this scope. Ensure that this scope is removed after the subroutine is complete, and global variables being able to be read, but not written to. | a = 7 procedure test()  a = 5 endprocedure test() print(a) 🡺 7 |
| 7.4 | Allow the “global” keyword to be used before an assignment inside a subroutine definition, so that global variables can be re-written inside of the local scope. | a = 7 procedure test()  global a = 5 endprocedure test() print(a) 🡺 5 |
| 7.5 | Allow functions to be created in the same format as procedures, but with the “function” and “endfunction” keyword instead. Allow the return keyword to be used within functions, at which point the subroutine will immediately stop processing | function double(x)  return 2 \* x  print(“Ignored!”) endfunction double(3) 🡺 No output |
| 7.6 | Allow the returned value of a function to be stored in values and integrated with other expressions as if a normal value. | print( double(3) + 1 ) 🡺 7 |
| 7.7 | Ensure recursion can occur, with multiple instances of the same subroutine existing at once, each with separate scopes and parameters which cannot interact. | function factorial(n)  if n <= 1 then  return 1  endif  return factorial(n – 1) \* n endfunction print(factorial(5)) 🡺 120 |
| 7.8 | Ensure complete error handling, with the number of arguments always matching the parameters, the syntax being perfectly followed and so that return statements cannot be used in procedures, all with descriptive error messages. | factorial(7, 9) 🡺 Error procedure error()  return “Test” endprocedure 🡺 Error |

## 8: native functions

A screenshot of a computer

Description automatically generatedThis section is very important, mainly due to input() and print() being incorporated, which will allow the program to finally be used like a normal language.

This involves adding all of the default functions in the languages and giving them proper functionality.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case |
| 8.1 | Allow print() to be used, accepting 1 or more arguments, and outputting each argument separate by a space. | print(“a”, “b”) 🡺 a b |
| 8.2 | Allow input() to be used, accepting an optional parameter to be outputted as an argument. This will prompt for a user input, which will be returned by the function as a string containing the text that the user gave to the program | x = input(“Enter name: ”) print(x) 🡺 Enter name: Joe 🡺 Joe |
| 8.3 | Allow str(), int(), float(), real() and bool() to return the corresponding data type that was defined in module 6. | n/a (see module 6) |
| 8.4 | Allow ASC() to be used, accepting a string with a length of 1, and returning the character’s corresponding ascii value as an integer | print( ASC(“A”) ) 🡺 65 print( ASC(“a”) – 7) 🡺 90 |
| 8.5 | Allow CHR() to be used, accepting an integer as an argument, and returning the corresponding ascii value as a single character string | print( CHR(65) ) 🡺 A |
| 8.6 | Allow random() to accept two Integers as arguments, where it will return a random integer inclusively, or two floats, where it will return a random float inclusively. | roll = random(1, 6) temp = random(10, 35) |
| 8.7 | Ensure all of these functions cannot be redefined, receive the correct number of arguments in the correct types, and are accessible globally throughout the entire program | input(5) 🡺 Error ASC(“test”) 🡺 Error CHR(2.7) 🡺 Error |

## 9: switch and arrays

A white background with black text

Description automatically generatedA screenshot of a computer

Description automatically generatedEven though very different, I have merged these two features into the same module as I am hoping they should both be reasonably simple to implement, and each module alone would be too small.

Hopefully switches can build off of the pre-existing if statement code, and arrays should hopefully not need masses of implementation.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case |
| 9.1 | Allow a switch statement to be created by “switch”, followed by an expression, followed by a colon, then containing a list of 1 or more cases, closed by “endswitch”. | switch 2+2:  // contents endswitch |
| 9.2 | Allow cases to be created by “case”, followed by an expression and colon, followed by the contents of that case, until either “endswitch” or another case is encountered. When running, a case’s contents will be executed if the expression matches the expression in the main switch statement. | switch “test”:  case “te” + “st”:  print(“Passed”) endswitch 🡺 Passed |
| 9.3 | Allow a final default case to be added, by the word “default” and a colon, followed by any contents. It must be the final case and should be followed by an “endswitch” after its contents. | switch 3:  case 4:  print(“Fail”)  default:  print(“Pass”) endswitch 🡺 Pass |
| 9.4 | Ensure that the order of the cases in the switch statement is always maintained, with descriptive error messages with any issues in the syntax to help students fix issues | switch “no colon” 🡺 Error case 3 then 🡺 Error default 🡺 Error |
| 9.5 | Allow empty arrays to be defined by “array”, followed by the name, followed by square brackets, containing an expression for the length of the array, with optional additional expressions separated by commas producing multiple dimensions with the respective lengths. | array countries[4] array chessboard[8, 8] |
| 9.6 | Allow a pre-defined array to be defined with “array”, followed by the name, followed by an equals, then with square brackets containing all of the elements in an array. | array cols = [“Red”, “Blue”] array nums = [1, 2, 3, 4, 5] |
| 9.7 | Allow a value in array to be accessed by its identifier, followed by square brackets with the position to be accessed, with multiple positions separated by commas for many dimensions. | print( cols[1] ) 🡺 “Blue” print( nums[3]^2 ) 🡺 16 |
| 9.8 | Allow a value in an array to be changed with the same syntax, but followed by an equals sign with the new value of that position | cols[0] = “Green” |
| 9.9 | Ensure that arrays are 0-indexed, can only ever store a single data type, if they are multiple dimensions, they must be rectangular, and their lengths are constant so they cannot be popped or pushed to. | cols[0] = 3 🡺 Error array test = [[1, 2, 3], [4, 5]]  🡺 Error |

## 10: string methods

A white background with black text

Description automatically generatedThis module involves adding all of the properties and methods for the string data type, and a lot of the syntaxes are very different from other languages.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case |
| 10.1 | Allow .length to be used on a string, which will return the length of that string | print(“hello”.length) 🡺 5 |
| 10.2 | Allow .substring() to be used on a string, accepting two integer arguments, returning the substring at the position of the first, with the length of the second | print( “hello”.substring(1, 2) ) 🡺 el print( “test”.substring(3, 1) ) 🡺 t |
| 10.3 | Allow left() and right() to be used on a string, returning the corresponding substring from either side of the string. | print( “hello world”.left(4) ) 🡺 hell print( “hello world”.right(3) ) 🡺 rld |
| 10.4 | Let .upper and .lower to be used on strings, returning the string in full capitals or full lower case respectively | print( “hello”.upper ) 🡺 HELLO print( “WORLD”.lower ) 🡺 world |
| 10.5 | Ensure all of the methods have proper error handling for invalid cases, such as impossible parameters for the substring functions, and being used on the wrong types | 3.substring(4, 5) 🡺 Error “test”.substring(2, 7) 🡺 Error “”.right(3) 🡺 Error |

## 11: file handling

A screenshot of a computer

Description automatically generatedThis will be the final module to be implemented, as it is the least important, but upon completion then every single part of the specification will be incorporated, as I will have implemented the entire list of features.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case |
| 11.1 | Allow a file to be opened with open(), which should be stored in a variable so it can be read of written to afterwards. | file = open(“test.txt”) |
| 11.2 | Allow a file to be closed with .close(), at which point it can no longer be read or written to | file.close() file.readLine() 🡺 Error |
| 11.3 | Allow a file to be read with .readLine(), which will return the next line in the string based on the usage of \n. This will be kept track off and will change upon each usage, but an error may occur if there are no more lines to read | print( file.readLine() ) 🡺 prints the line |
| 11.4 | Allow .endOfFile() to be used on the file, which will return True or False depending on if the end of the file has been reached by using .readLine(), allowing for the user to access the entire contents of a file without causing any issues | while NOT file.endOfFile()  print( file.readLine() ) endwhile 🡺 will output entire contents |
| 11.5 | Allow .writeLine() to be used on a file, which will accept a string as an argument, and will write this at the end of the file. Allow \ns to be used in order to write new lines. | file.writeLine(“testing”) file.writeLine(“A\nB\nC”) |
| 11.6 | Create a new, empty file with newFIle(), which will accept the filename as an argument, and create but not open it | newFIle( “names.txt” ) f = open( “names.txt” ) |
| 11.7 | Ensure proper error handling throughout, so that all arguments are correctly checked, and so that a file can be opened multiple times without any issues. | file1 = open( “test.csv” ) file2 = open( “test.csv” ) print(file1.readLine()) print(file2.readLine()) 🡺 Outputs first line twice |

## 12: the user interface

Now that the whole specification has been covered, I will be outlining the details I feel are needed for the interface, so that it is easy for a user to interact, however I will plan this more precisely in the design stage, but this is the overall key elements. For this, I will have no test cases, but instead a justification column, because these are not as clearly needed as the ERL features.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Justification |
| 12.1 | Have an input and an output window, for which ERL can be entered, and then the result is displayed in the output window whenever a run button is pressed. | This will emulate many IDEs the student would have used before, providing a comfortable and familiar layout |
| 12.2 | Allow for user inputs to be taken directly from the output window, like a console interface, and allowing the user to properly interact with the program | Again, ensuring ease of use is important, and avoiding any pop-up windows is best for familiarity. |
| 12.3 | Allowing for ERL files to be uploaded from the local computer, and put into the input window, and also downloaded, where the current contents of the input window will be saved into a file | Allows students to save their work overtime, as well as files to be given out by teachers for practice: overall very good for sharing of work and progress |
| 12.4 | Allow files to be uploaded from the local computer, as well as modified in the interface, so that the user can use the file handling in the ERL and see the results. | This is important as otherwise file handling is useless without being able to view any actual changes to files |
| 12.5 | Allow for basic highlighting of different keywords in the input window, as well as different colours for strings, numbers, Booleans, methods and each type of token that the language contains. | As pointed out by the stakeholders, this will make the program a lot more usable, which is important for encouraging good coding practice. |
| 12.6 | A resource containing a clear description of the ERL is available, with details on the syntax to be used by students, as a more friendly alternative to the badly organised and confusing specification. | Having this built-in to the program should make it easier for students to learn how to code in the ERL by having the details at hand at any point. |
| 12.7 | A resource containing exam questions and solutions is available, so that a student can use it to help practice questions for an exam if they are using ERL in a revision setting rather than inside of lessons | Brings some purpose to the program, so that students do not need to come up with their own concepts or search past papers by themselves. |

Overall, the focus on the interface is making it as easy to use as possible. This is not a complete list, and during the development of the interface I would like to add as many smaller features as possible to make the overall experience a lot easier and more enjoyable for any students, in order to encourage them to continue practicing, as the interface being frustrating and difficult to use will detract from the main purpose of the program and will stop students from coding.

Overall, this is the outlines of the project complete!