success criteria

This section will now be outlining what features are required: not just the success criteria but the process to decide them as well.

# essential features

## for the erl translation alone

Overall, the essential features for this aspect are very set in stone, because the OCR GCSE specification clearly lays out the different features of the ERL at the following link:  
<https://www.ocr.org.uk/Images/558027-specification-gcse-computer-science-j277.pdf#page=27>

I have decided to split the translator side of this project into 11 different modules, each of which will align with one of the high-level objectives. These will all follow the syntax that has been laid out by OCR to ensure that they align exactly with what students will be used to. These will be as follows:

🡺 Arithmetic calculations: using order of operations to yield answers to mathematical expressions  
🡺 Variables: being able to store and get values from variables, including setting constants  
🡺 Logic: introducing booleans, comparators and logical operators to evaluate logical expressions  
🡺 If statements: evaluate the corresponding section of code based on their conditions  
🡺 Loops: allow While, Do Until and For loops to iterate over their code correctly  
🡺 Strings and Casting: Introduce a string data type and add casting between different types  
🡺 Functions and Procedures: Allow subroutines to be defined and called in the program  
🡺 Input and Output: Define custom functions which allow users to interact properly with the program  
🡺 Switch and Arrays: Implement a switch structure and the array data structure

These last two are not as important but should still be included if possible:  
🡺 String methods: Add the ability to manipulate strings with all the different methods and properties  
🡺 File handling: Allow files to be opened, read from, written to, and closed.

These will be implemented in the order they are laid out, because certain features need to exist before others because their features build off the previous foundations. Additionally, some features are a lot more important than others: for example, having loops and functions is a lot more important than having file handling or string methods because they are used more frequently by GCSE students, therefore these are more of a priority to implement if development is slowed.

Ideally, I would like to be able to implement all these different objectives, however I feel like string method and file handling could be potentially left out, because they are less frequently used and are not as core features. However, I feel that everything else up to that is critical to the project being a success, so must be included in the final program otherwise it will be a failure.

### justification

It is very clear why these need to be added: the entire purpose of a translator for ERL is that it can handle all the features that ERL includes, so therefore these are all necessary to be in the final product. When interviewing them, all my stakeholders agreed that this was the number one priority, so making sure that the program will run with every single feature is very important, as even if there were then no other features, they would still be able to practice exam questions using the program, and that is the entire purpose.

## excluding translation

There are still essential features excluding the translator itself. I would say that, expect the user interface, these features are not as important as all the translator features, so should not be as much as a priority but could potentially be included.

🡺 User Interface

This is the most important out of the other features. Because the project is aimed towards new programmers, this will have to be very simple and intuitive to use, with clear places to input the ERL code, a section where outputs are displayed and clear run and help buttons.

🡺 Very clear error handling

Although this will be a part of the entire development process for the translator, it also must be focused on separately. Ensuring that, when it is not possible to translate due to a syntax error or any other type of error, this is clearly outlined to the user so that it is easy to fix.

🡺 Clear documentation

This will be one of the very last stages of development but involves writing a help page to explain the features of ERL to any unfamiliar users. This is not computationally complex, but will be easy to do, and should be accessible via the user interface by some form of help button.

These last two are not essential, but should try and be included:

🡺 Syntax highlighting

Both of our GCSE students have mentioned this as being a good feature to try and include, and anything that improves the user experience to improve clarity should be attempted. This will involve highlighting different keywords or blocks of code differently to make them easier to distinguish.

🡺 Question suggestions

This could take questions from past papers or online resources and present them to the user, with an option to see the intended solution. This could be expanded further however will likely be limited to the basics.

## limitations

If possible, I would like to take no limitations within the ERL features itself. The only aspects that could not be included are aspects which would never be used within a GCSE question, such as recursion – which although could technically be possible within the defined ERL guidelines – would never be used in a paper so is not required for the student. However, any potential limitations for the translator will have to be evaluated in the Design stage as I currently cannot foresee how development will work for said features.

There will have to be limitation on the other features. Our current GCSE stakeholder suggested it aiding you to fix errors, and although good error messages should hopefully make this a lot easier, implementing a solution that a lot of IDEs have which would point out and suggest any problems is outside of the scope of this project, so therefore won’t be concluded.

Along the same lines, any syntax highlighting will be very basic, and not as in depth as some IDE’s, which use different colours for different levels of nesting; if a variable is being referenced but not defined; and more niche features. My syntax highlighting will most likely consist of only highlighting keywords and different data types differently. Likewise, a question suggestion feature could be significantly expanded, to include checking the proposed answer by the user and running test cases, however this is too far beyond the scope of the project so will have to be excluded.

Finally, a lot of features will have to be completely excluded, such as adding a trace table feature to aid debugging, or pointing to external videos which can help with specific errors, or a streak and leaderboard system. This is because they detract from the main purpose of the program slightly: which is to allow ERL to be executed and practiced and would add a lot of extra work to an already complex problem.

# success criteria

This will break down each of my modules and then additional features, laying out exactly what needs to be included in each section. I will not provide a justification for every single criteria, however they are all laid out in the specification, so therefore are all required.

A lot of the test cases will use print(), but will need to be tested differently in development as half of the modules will be implemented after subroutines are introduced.

### A white rectangular box with black text Description automatically generated1: arithmetic

This is a relatively obvious aspect of the program, with a focus on BODMAS being properly followed, and proper error handling for the different invalid cases.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case |
| 1.1 | Allow addition, subtraction, multiplication, division, exponentiation, modulus, and quotient division to be taken of two values | print(2+2) 🡺 4 print(3\*7) 🡺 21 |
| 1.2 | Allow multiple calculations to be chained together, and to be evaluated based on the BODMAS order of operations | print(2 + 3 \* 7) 🡺 23 print(3 – 7 ^ 2) 🡺 -46 |
| 1.3 | Allow integer and float values to be used, with floats being returned if a float was used in the calculation | print(2.0 + 4.0) 🡺 6.0 print(5 / 2) 🡺 2.5 |
| 1.4 | Allow brackets to be used to priorities certain calculations, as well as infix operators + and – being allowed, mainly for negatives. | print( (2 + 3) \* 7 ) 🡺 35 print( -7 \* -2 ) 🡺 14 |
| 1.5 | Return errors when invalid inputs are entered, such as unexpected characters, incorrect syntax or dividing by zero. | 2 + $ 🡺 Error 3 / \* 7 🡺 Error |

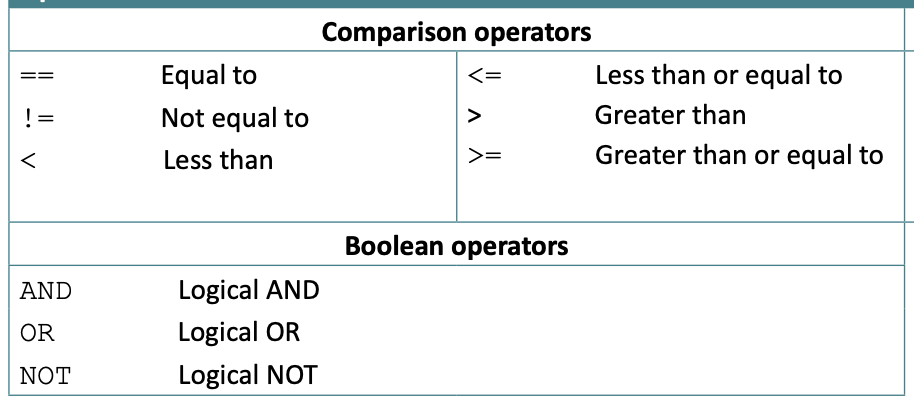
### 2: Variables

A blue and white rectangular object

Description automatically generatedThis section focuses on adding variables and ensuring they interact with the arithmetic evaluation.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case |
| 2.1 | Allow identifiers to be used to set and read the value of a variable | num = 5 print(num) 🡺 5 |
| 2.2 | Allow identifiers to be used within arithmetic expressions and for the correct result to be evaluated | a = 7 print(2 \* a – 3) 🡺 11 |
| 2.3 | Allow the const keyword to create constants, which cannot be redefined after assignment | const var = 3 var = 7 🡺 Error |
| 2.4 | Ensure relevant error handling: accessing variables before assignment and not allowing invalid values to be set to variables | b 🡺 Error c = 2 \* @ 🡺 Error |

### 3: logic

This module is building up how the program treats statements, introducing True and False and different comparison and boolean operators that can be used to compare different expressions.

This is building up the foundations for Loops and If Statements, which require this logic evaluation for the conditions that they use.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case |
| 3.1 | Add a Boolean data type represented by “True” and “False” | print(True) 🡺 True |
| 3.2 | Allow comparison operators ==, !=, <, <=, > and >= to be used to compare two arithmetic expressions, values, or Booleans | print(3 + 5 > 2) 🡺 True print(True == False) 🡺 False |
| 3.3 | Allow NOT to be used in front of statements or single booleans to alter their value, with a lower precedence than comparators | print(NOT 3 > 2) 🡺 False print(NOT True) 🡺 False |
| 3.4 | Allow AND and OR to be used with a lower precedence than NOT to chain these statements together | print(True AND True) 🡺 True print(False OR True) 🡺 True |
| 3.5 | Ensure complete error handling, with incomplete statements and invalid use of operators resulting in errors | 4 == 🡺 Error True == OR False 🡺 Error |

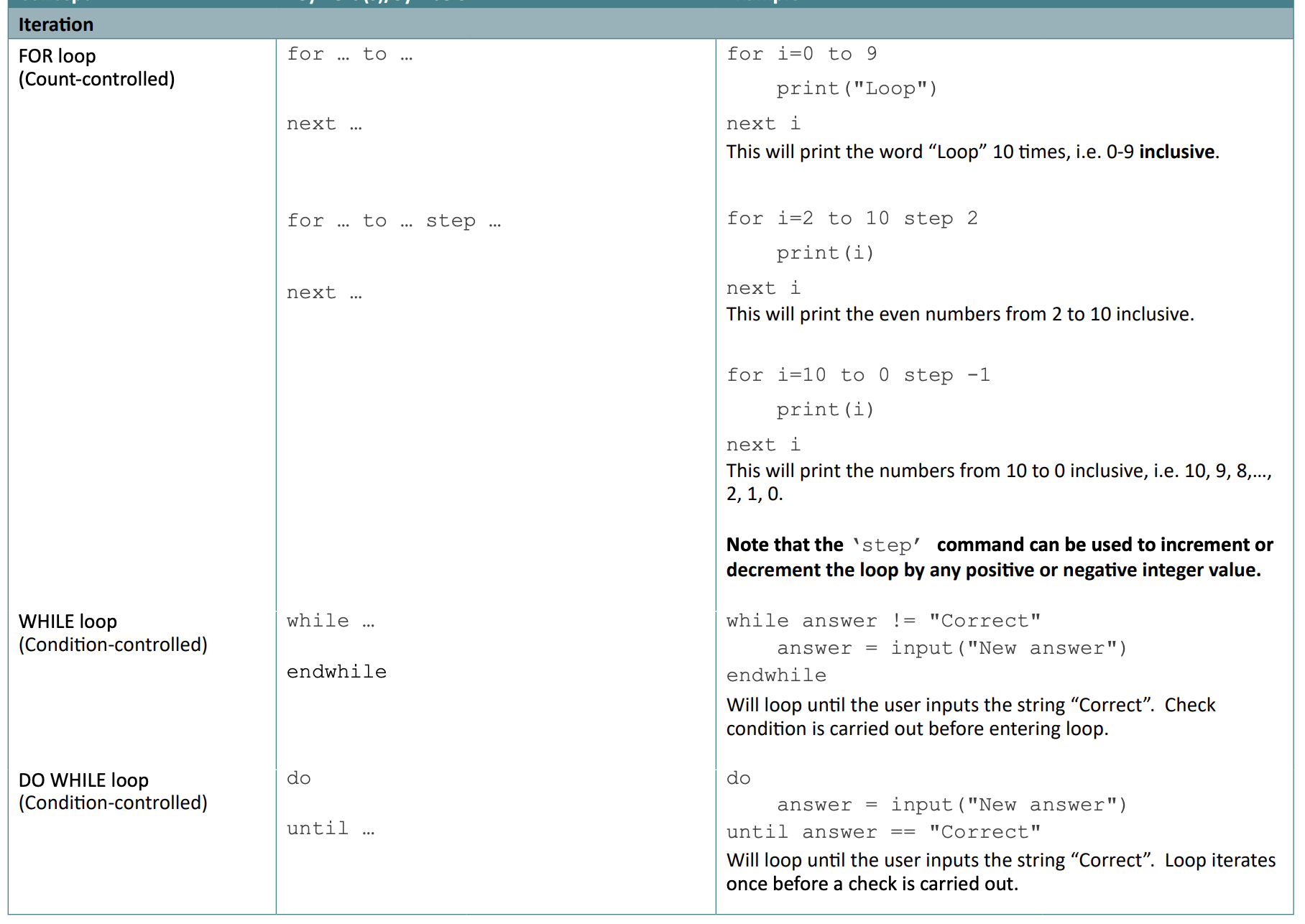
### 4: if statements

A screenshot of a computer

Description automatically generatedThis section adds if statements, focusing on following the exact syntax that the ERL requires, with lots of cases ensuring that it is perfectly followed. The correct case must also be evaluated based off of their conditions.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case / Example |
| 4.1 | Allow if statements to be created by the keyword “if”, followed by a condition and “then”, then containing the code for that case, and closed by the closing keyword endif | if 2 + 2 == 4 then  print(“Pass”) endif 🡺 Pass |
| 4.2 | Allow additional cases to be added using “elseif”, followed by a condition and “then”, followed by the corresponding code for that case before another “elseif” or “endif”. | if False then  print(“Fail”) elseif 9 > 7 then  print(“Pass”) endif 🡺 Pass |
| 4.3 | Allow a final default case to be added, by just the keyword “else” on a line by itself, followed by the corresponding code for that case, evaluated if no other conditions are True | if False then  print(“Fail” else  print(“Pass”) endif 🡺 Pass |
| 4.4 | Ensure that the order of the cases are regulated: always started by if, followed by elseif, then an optional else, then endif | elseif 5 > 3 🡺 Error endif 🡺 Error |
| 4.5 | Ensure that the correct syntax is perfectly followed, giving detailed feedback about missing keywords so that it matches the specification exactly, as well as only valid conditions. | If True 🡺 Error else then 🡺 Error elseif 2 then 🡺 Error |

### 5: loops

This module implements the 3 different types of loops, with focus on matching the syntax that has been shown on the right which the specification requires.

All of the loops function in slightly different ways, and it is important to include them all in the program because they are a very core feature that the student will definitely need.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case / Example |
| 5.1 | Allow while loops to be created by the keyword “while”, followed by a condition, then the contents of the loop, and then closed by the keyword “endwhile” | var = 7 while var < 10  var = var + 1 endwhile print(var) |
| 5.2 | Ensure that the condition of the while loop is evaluated before the loop is ever run, and the contents are continually repeated whilst the condition of the loop is True | while False  2 + 2 endwhile 🡺 No output |
| 5.3 | Allow do until loops to be created by the keyword “do”, followed by the contents, and closed by “until” which is followed by the condition of the loop | do  var = var / 2 until var <= 4 |
| 5.4 | Ensure that the contents of the do loop is run once before the condition is evaluated, and then the contents is continually looped while the condition is False | do  4 + 4 until False 🡺 8 outputs |
| 5.5 | Allow for loops to be created by the keyword “for”, followed by an Identifier, then ‘=’, then an expression for the starting value, followed by “to”, followed by an expression for the final value. Then followed by the contents and closed by “next” followed by the same identifier that defined the for loop. | for i = var to max – 1  age = age – 1  var = var + i next i |
| 5.6 | Ensure that on each iteration, the value of the variable is incremented and the next iteration occurs if the value of the variable is less than or equal to the final value | for i = 0 to 3  i next i 🡺 outputs 0, 1, 2, 3 |
| 5.7 | Allow an optional “step” keyword to follow the finishing value, followed by an expression which determines how much the variable changes by, also allowing negative values | for I = 3 to 1 step -1  i next i 🡺 outputs 3, 2, 1 |
| 5.8 | Ensure full error handling so that the syntax is perfect, and conditions are valid and For loop values are Integers | while 3 then 🡺 Error for i = 0.3 to 2.7 🡺 Error |

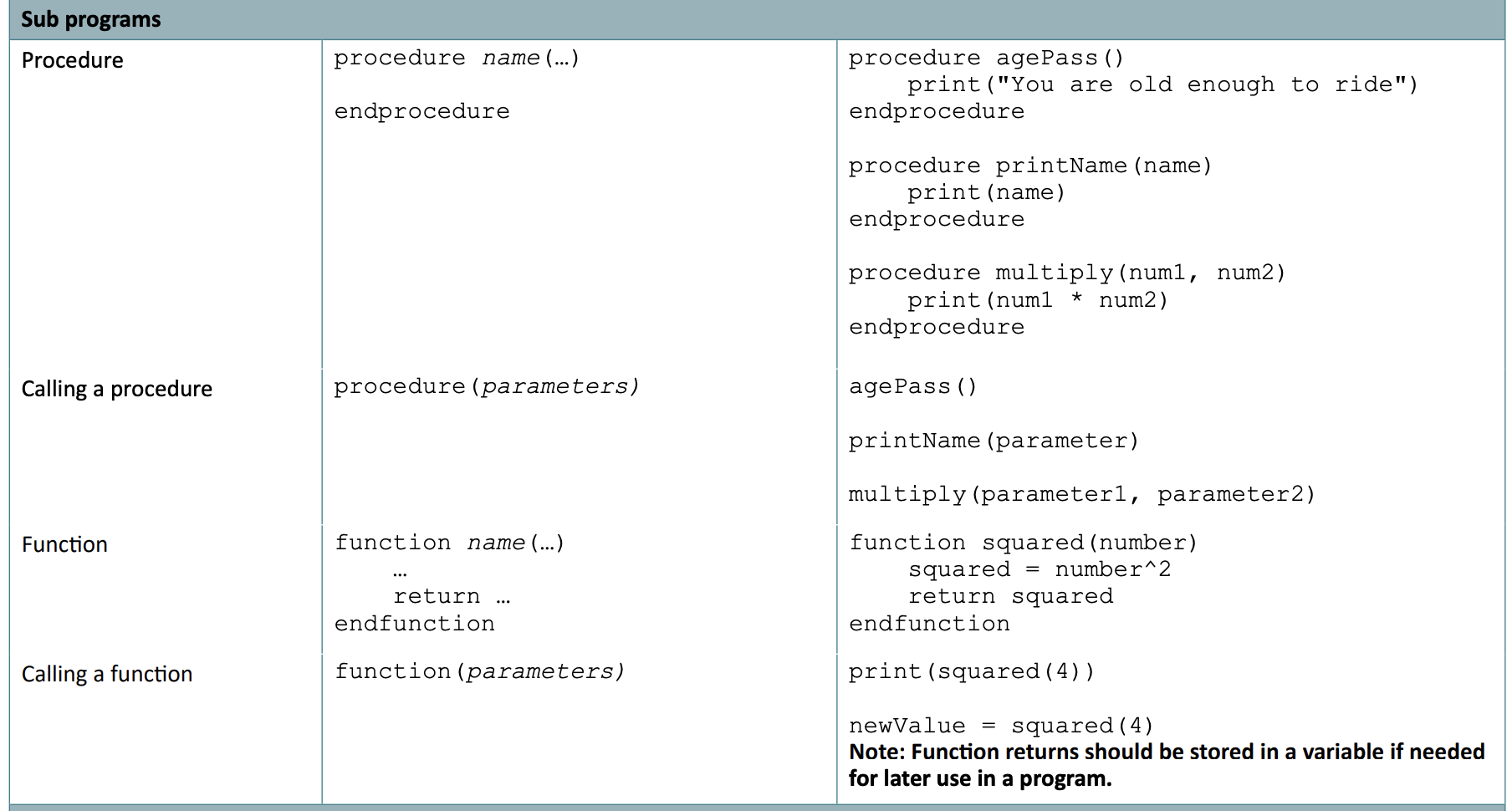
### 6: strings and casting

This is a smaller module, introducing strings and concatenation, but overall stricter type handling for the entire system, as well as some casting between the different data types.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case / Example |
| 6.1 | Allow a string data type to be created with either “ or ‘, which can be concatenated using the + operator | name = “Patrick” “a” + “b” 🡺 “ab” |
| 6.2 | Ensure stricter type handling in the whole system. Returning detailed error messages if invalid types are combined | 2 + True 🡺 Error “test” \* 3 🡺 Error |
| 6.3 | Allow type casting between the four different data types, allowing for valid and invalid cases returning errors if needed. | int(“7") 🡺 7 float(“test”) 🡺 Error |

Note at this stage, functions will not have been implemented so type casting will not be accessible directly through plaintext and will only be possible by direct instruction to the program.

### 7; functions and procedures

This module will allow the user to create custom subroutines which they can call. The focus here will be ensuring that variable scope is handled correctly, so local variables in functions are stored separately. Overall, I predict this to be the most complex module in terms of the code complexity.

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Test Case / Example |
| 7.1 | Allow |  |