**Language Design and Implementation**

**6CC509**

[Diagram

Description automatically generated](https://xkcd.com/2309/)

Assessment 2

Rich Conniss

Language Design and Implementation

6CC509

Contents

[Module Leader 3](#_Toc93937093)

[Key dates and details 3](#_Toc93937094)

[Description of the assessment 3](#_Toc93937095)

[Reading 3](#_Toc93937096)

[Crafting Interpreters by Robert Nystrom 3](#_Toc93937097)

[F# for Fun and Profit by Scott Wlaschin 3](#_Toc93937098)

[Build Your Own: Programming Language by Daneil Stefanovic 4](#_Toc93937099)

[Assessment Content 4](#_Toc93937100)

[Language Choice 4](#_Toc93937101)

[Assessment Rubric 5](#_Toc93937102)

[Stages 5](#_Toc93937103)

[Anonymous Marking 10](#_Toc93937104)

[Assessment Regulations 11](#_Toc93937105)

# Module Leader

Rich Conniss

Email: [r.conniss@derby.ac.uk](mailto:r.conniss@derby.ac.uk)

Phone: Teams call

Office Hours: <https://calendly.com/rconniss/office-hours>

# Key dates and details

|  |  |
| --- | --- |
| **Assessment Type:** | **Language Design, Viva** |
| **Assessment weighting:** | 60% |
| **Word count/Length** | NA – 15 min Viva |
| **Learning Outcomes:** | 2 |
| **Submission Method:** | BlackBoard |
| **Submission Date:** | 23:59 UK time, 20/05/2022 |
| **Provisional Feedback Release Date:** | 10/06/2022 |

# Description of the assessment

Throughout your studies, you have been the benefactor of programming languages developed by other people. In this assignment, you are expected to use one of these languages to implement your own Turing-complete language parser and interpreter.

# Reading

It is **highly recommended** that you engage with the provided reading material as part of this module throughout development of your language. These materials provided do an exceptional job at demystifying jargon that many of you will initially find overwhelming.

## Crafting Interpreters by Robert Nystrom

A book focused on the practical challenges of language design. The materials will walk you through two distinct approaches to writing language interpreters in Java and C. An online version of the book is available at <https://craftinginterpreters.com/contents.html>.

## F# for Fun and Profit by Scott Wlaschin

The materials on this website will expose you to a separate way of problem solving than you will have been taught in your studies, demonstrating a more human-friendly model for writing things like parser logic. The website can be found at <https://fsharpforfunandprofit.com>.

## Build Your Own: Programming Language by Daneil Stefanovic

A Git repository that lists various online tutorials for designing and implementing programming language features. The repository can be found at <https://github.com/danistefanovic/build-your-own-x#build-your-own-programming-language>.

# Assessment Content

The assessment is separated into distinct stages of necessary functionality. As such, you will need to implement progressively more advanced features for higher marks.

Lab sessions are focused on building the foundations necessary to bootstrap your programming language up to a simple postfix-to-infix expression parser. From there, you will be free to modify functionality within the requirements of each stage.

## Language Choice

This assignment is open for you to choose whatever tooling you prefer, with the caveat that all dependencies be installed on your Azure Labs virtual machine instance for the final submission. This is necessary for your assignment to be markable, and failure to do so will result in 0 marks being awarded.

When choosing your implementation language, you will want to consider the following things:

* What data structures does the language provide? The assignment will require you to use stacks and associative arrays throughout it, so the language will either need to provide these features or you will need to be able to write them.
* What support does the language have for discriminated values? Java has strongly typed Enums that are distinct from numeric types, while C Enums are simply aliases to numbers.
* Does the language facilitate any sort of polymorphism? For example, C# has abstract classes, while F# has discriminated unions. Being able to have a type that represents many other sub-types will be a necessary for implementing the parser and interpreter.
* How easy is it to represent contiguous data in the language? Java classes are all allocated on the heap, wherever it can find space. Meanwhile, C# supports stack-allocated data in the form of structs. This will be an important consideration if you are trying to do something performance critical.

Finally, to save implementing your own garbage collector, it is recommended that you use a language which already has one built in, such as Go, Java, or any of the DotNet languages. Hand-rolled garbage collector implementations are not rewarded until the later grades.

# Assessment Rubric

Your work will be marked by Viva. You will have 15 minutes to present your language and show off its capabilities. Vivas will take place during the spring assessment period. Your code should be submitted to Course Resources prior to the Viva sessions taking place. Viva dates will be organized after week 12 of teaching.

## Stages

Full completion of previous stages is necessary to receive any marks for sequential stages. For example, failure to fully complete stage 2 will mean no marks are awarded for completion of stage 3 and onwards.

#### Stage 1: Basic Calculator (0 – 20%)

As computers are nothing more than complicated calculators, it should be no surprise that the first step in language development is the creation of a simple arithmetic expression parser.

Start by implementing a program that accepts the path to a source file containing an arithmetic expression on any valid real numbers in infix notation. The program should be capable of parsing the contents of the input file, executing its expressions, and printing the product.

Your parser should – at a minimum – handle unary negation and binary addition, subtraction, multiplication, and division. With any combination of these arithmetic operators, your parser should be capable of producing correct outputs. Example inputs that you will want to test against include but are not limited to:

**1 – 2**

**2.5 + 2.5 - 1.25**

**(10 \* 2) / 6**

**8.5 / (2 \* 9) - -3**

How you handle the storing of data required for the computation is entirely up to you. A popular approach is to parse the symbols to an imaginary instruction set and then execute them one-by-one (a virtual machine). Another approach is to parse the symbols into a tree of nodes and traverse it, executing the logic defined in each node (an abstract syntax tree).

#### On Regex

While implementing the parsing logic, you may be tempted to reach for regular expressions (regex). If all you were going to parse is numbers and arithmetic operators, this would be a reasonable approach, however it is not a scalable solution for programming language parsers

#### Stage 2: Boolean Logic (20 - 40%)

Booleans are foundational to all conditional logic in programming languages and will be necessary for control flow functionality introduced in later stages. After implementing support for numbers, further extend it to support Booleans and Boolean-producing expressions.

Your parser should – at a minimum – handle:

* Binary comparison, equality, and inequality between numbers.
* Binary equality and inequality between Booleans.
* Logical “AND” and “OR” between Booleans.
* Unary negation of Booleans.

With any combination of the above arithmetic expressions, your parser should be capable of producing correct outputs. Example inputs that you will want to test against include but are not limited to:

**true == false**

**true != false**

**(5 < 10)**

**!(5 – 4 > 3 \* 2 == !false)**

**true and true**

**false and true**

**(0 < 1) or false**

**false or false**

How you want to handle comparison, equality, and inequality across types is left to you. For example, JavaScript considers **1 == true**. In stricter type systems, this would be considered an illegal comparison. Whichever approach you choose, will be expected to defend your decision in your Viva.

#### On Value Storage

As your language is dynamically typed for the time being, you will want to consider how adding a new data type to it will alter its in-memory representation of values. It is likely you have been storing numbers directly, but now a value may either be a number or a Boolean.

#### On Floating Point Numbers

If you are using floating point values to represent numbers, note that there are some non-obvious traps surrounding their underlying implementation that you will want to consider when implementing your equality and comparison operations.

#### Stage 3: Text Values (40 – 50%)

Text is a necessary component of any programming language that wishes to communicate with the end-user. In C, we have fixed-size character buffers, whereas in JavaScript we have dynamic string types.

Once you have real number and Boolean expressions working, further expand your language to support text values. Text values should be declarable by writing a string literal, such as **"hello world"**.

Your parser should, at a minimum, handle binary concatenation, binary equality, and binary inequality between text values. With any combination of these operations, your parser should be capable of producing correct outputs.

**"hello" + " " + "world"**

**"foo" + "bar" == "foobar"**

**"10 corgis" != "10" + "corgis"**

Above is an example of how text values may work in your language. Use these examples – adjusted for your own language syntax – to test your implementation works as expected.

The introduction of a new type in your language means new considerations surrounding how they should all interact with each other. For example, what is the behaviour of **1 + "0"**, and should **false == "false"**? You are expected to decide how to handle this and defend your decision during your viva.

For top marks in this stage, you will want to consider how other languages handle embedding non-printable character codes into text and handle them accordingly in your parser.

#### On Text Storage

Unlike the number and Boolean types that your parser currently supports, sequences of text are not a fixed size in memory. It is therefore likely that you will need to make special considerations in to store and handle them.

#### Stage 4: Global Data (50 – 60%)

Currently, only one expression may be computed per execution. This is less than adequate for more complex computer programs, which will want to compute many different values over their lifetime. For completion of stage 4, you are expected to implement support for global variables that may be assigned to, read from, and printed.

**quickMaths = 9 + 10**

Above is a simple example of how assigning to global variables might work in your language. When a global get or set operation is performed, your interpreter should look up a global variable matching that name and perform the appropriate get or set operation.

**quickMaths = quickMaths + 2**

Accumulation of operations on existing values should be supported by using the global variable recursively in its own expression, as shown above.

**print quickMaths**

The printing operation may work however you wish. The above example takes inspiration from the built-in print operation in Python 2. In Python 3, this was changed to use a regular function call syntax.

You will want to consider how you denote the end of expressions. Many C-like languages use semi-colons to mark the end of an expression or statement, while newlines are the preferred approach in other languages. This is a design decision left up to you, which you will be expected to defend in your viva.

For top marks in this stage, you will want to minimise the amount of work performed each time a variable is read from or written to. To start, you will want to make sure the data structure you are storing global variables in has a reasonable time complexity for the random access of string index values.

#### On Data Persistence

In most scripting languages, global variables exist in memory from the moment they are created to the moment they are unassigned. This will require you to consider how your interpreter will find this data – as well as how it will know if a value exists. Python has a “None” value to indicate anything, but deleting global variables requires the use of the “del” keyword. The developers of Lua decided to take a different approach and make its “nil” value delete any global it is assigned to.

#### Stage 5: Control Flow (60 – 70%)

By this stage, your language should be close to Turing complete. Correct completion of stage 5 requires the implementation of if statements, condition-based repetitive logic, and some way to read input from the end-user of the program.

**is\_running = true**

**shopping\_list = ""**

**while (is\_running == true) {**

**item = input("add an item to the shopping list: ")**

**if (item == "") {**

**is\_running = false**

**}**

**shopping\_list = shopping\_list + ", " + item**

**}**

**print shopping\_list**

Above is an example of some pseudo-code demonstrating how the requirements might be implemented alongside composing all previous stages of functionality.

How you decide to implement the requirements is entirely up to you. It is not expected that your language look exactly like this, but it should be able to write the above program in one way or another. Whichever way you approach implementing these features, you will be expected to defend them in your viva.

For top marks in this stage, you will want to implement support for “else” and “else if” chains on regular if statements.

#### On Parsing Branches

For the most-part, all logic required for parsing branching logic should already exist in your parser. The difficulty will be in figuring out where in the bytecode or abstract syntax tree that you need to jump to.

#### Stage 6: Additional Features (80%+)

Now that you have a minimum implementation of a programming language, you can begin to extend it with convenient functionality and syntactic sugar to make it nicer to use.

Each additional feature listed has a grade mark that is relative to its implementation difficulty. It is recommended that you implement easy features first, rather than jump immediately into the top-mark tasks.

#### List Data Structure (10%)

There is currently no way to associate many individual values as a list in your language. Implement some form of list-like data structure type that supports back-insertion, random removal, and random access with number-based indices.

#### Dictionary Data Structure (10%)

Dictionaries, sometimes referred to as associative arrays, are a very versatile data structure in that they can mimic any other data structure with varying efficiency. Implement some form of dictionary-like data structure type that supports assigning, removing, and querying for any type of value with any type of key.

#### Function-Based Code Reusability (10%)

Functions are a popular approach to encapsulating a simple unit of execution into reusable logic that may be called from elsewhere. Implement support for declaring and calling functions in your language.

#### Local Variables (15%)

Global variables are powerful; however, they are often considered an anti-pattern. Within the domain of dynamically typed languages specifically, global variables are also far-more difficult to optimise. To resolve this, implement local variables into the parser and interpreter in an efficient manner.

#### Further Features

Should you have other ideas for things that you want to implement as part of your language, speak to your assignment tutors for guidance.

# Anonymous Marking

Your assignment **MUST** be submitted electronically via Course Resources by the due date and time. The submission must consist of **ONE** file, named the same as your *student ID*, that is in the **ZIP** archive format.

Your zip archive must contain:

* A BUILD.txt file explaining clearly how to build the project from source using software you have made available on your virtual machine instance.
* A README.txt file explaining clearly how to use the program.
* Five example source files containing valid syntax for your language parser and interpreter.
* The complete source and project files necessary to build your language from source.

Failure for the project to build will result in a grade of **0%** for the deliverable. Further, if after building, your example syntax files do not work with your language you will not be able to achieve a grade greater than **40%** for the deliverable.

No personally identifiable information may be included in the file or filename beyond your student ID or student ID email address, to conform with the University anonymous marking policy.

# Assessment Regulations

The [University’s regulations, policies and procedures](https://www.derby.ac.uk/about/academic-regulations/) for students define the framework within which teaching and assessment are conducted. Please make sure you are familiar with these regulations, policies and procedures