Solent University

Faculty of Business, Law and Digital Technologies

Software Engineering
2022

Bradley Marshall

“How can logical understandings of programming concepts and semantics be used to construct a programming language?”

https://github.com/bradley499/laturon

Supervisor	:	Joe Appleton
Date of submission	:	May 2022

Acknowledgements

The completion of this study could not have been done without the guidance and support of my dissertation supervisor Joe Appleton. As well as the friends and family who gave me continuous support throughout this journey.

Abstract

Many components of commonly used programming languages are typically too complex for many people to understand, as the level and scope of knowledge required to understand all the different types of syntax and conventions can be inconsistent, and daunting — especially to novices. It was for this reason that the Laturon programming language was created. It was designed and developed around user-based feedback, as an amalgamation of people's understanding of programmatic terminology. With an additional focus on simplicity, the logic present within the syntax of the Laturon programming language, is not only simple, but is functionally interchangeable with logic from other programming languages. The Laturon programming language was revered as having a simplistic syntax that could be easily understood with little to no experience.

Table of contents

Figures

Equations

Tables

Acronyms

Abbreviation	Meaning
WASM	WebAssembly
IDE	Integrated Development Environment
API	Application Programming Interface
emcc	Emscripten Compiler Frontend
LLVM	Low-Level Virtual Machine
GCC	GNU Compiler Collection
GDPR	General Data Protection Regulation
EOF	End Of File
GDB	GNU Debugger
LIFO	Last-In-First-Out
UI	User Interface
OOP	Object-Oriented Programming
IO	Input/Output
P1	Person 1
P2	Person 2
P3	Person 3
P4	Person 4
P5	Person 5

1. Introduction

1.1 Project

This project attempted to design a programming language with a formative structure and syntax; generated from interviewing and questioning 17 different people's own understanding of logical terminology and design of logical steps. The programming language designed from the understanding of this project is named: Laturon. The project aimed to generate a WebAssembly (WASM) compilation target, to allow for people to program in the Laturon programming language from any device from within any major web browser.

1.2 Problem

The scope of knowledge required to understand a predefined programming language can be intimidating for a novice. As "programming languages are hard to learn" (Engebretson and Wiedenbeck 2002, pp.11-18) even for an experienced professional, partly due to the differing layers of abstraction, and syntactic stylings. I had noted that there is a growing sentiment of disgrace towards the idea of programming, or towards certain programming languages, as many people regard the idea of it being simply too complex to begin with which reflects upon the usability of other existing programming languages. However, this can in fact be remedied to account for these hardships, by instead having a programming language designed to be simpler to understand and based around terminology that people can understand.

Additional biases of a certain technology can also hinder the desire to learn a language. As, the "conceptual pragmatism" (Lewis 1956) of knowledge relating to the field of computing can generate bias, resulting in the dismay of a formation, or structure of a language, as typically programming languages "[appeal] to the ego, not the intellect" (Joyner 2022). With this overarching aspect, cognitive attribution towards linguistic choices could result in a change of a person's viewpoint as "cognitive data may attach to the data of sense but cannot simply coincide with such given data" (Lewis 1956). Albeit the scale of how a language can interlace the internal operations, can be learnt from semantics that closely follow the operative structure.

The intuitiveness of a semantic lexicon should prescribe a cognitive understanding of syntax. This is important, as "semantics [...] can be turned into a formal denotational [semantic] that globally defines the output sequence" (Berry and Gonthier 1992), which would further allow for a simplistic understanding of conceptual logic. In summary, the problem with the design of programming languages is that they are typically designed with the constructs of what the developer of the programming language wants. Which is why the project was to create the Laturon programming language, that has a syntax based on peoples' conceptual understanding of formative logic.

1.3 Research Question

The research question for this study is: "*How can logical understandings of programming concepts and semantics be used to construct a programming language?*"

2. Literature Review

The concept of programming, and programming language design, are subjectively divisive fields to observe. It is for this reason that a study into pre-existing publications occurred. Multiple authors had noted observations regarding programming language interoperability and syntactic style choices, which are reflected within this review. The addressed points all contribute towards the design choices of programming language design.

Fedorenko *et al* in 2019 had noted that programming bears parallels to natural languages, which is evident in the rise of programming languages that use functional operations within the terminology of spoken languages. However, the ability for one to comprehend the scale of different terminologies that are present within all programming languages is often too complex for many. Within Fedorenko *et al*'s findings, they had cited work from Fakhoury *et al*'s 2018 publication, stating that "keywords, variable names, function names, and application programming interfaces follow naming conventions that indicate their function; it has been shown that 'unintuitive' naming increases cognitive load" which in terms of a programming language's ability to be widely understood, can hinder a novice's' ability to fully understand, or interpret, what is meant by a functional approach when naming conventions do not fully relate to what is meant by a function.

Alternative thoughts constructed by Hughes in 1985, were that "programming language influences its uses at least with respect to the style of programming, the conceptual understanding of how a problem can be solved by a computer and the range of problems which can be attacked by programming", which contradicts what Fakhoury *et al* had discussed, by the means of how the natural flow of the unintuitive naming conventions. However, Hughes echoes a divisive viewpoint, that "programming languages should be viewed only as tools for users to describe algorithms to computers", negating the necessary worth of readability for a programmer, inciting that the source code itself should be designed with the computing/execution environment mind, rather than the developer, which is widely regarded — within the industry — as an important thing. Source code legibility is important, as it allows for multiple developers to engage with the source code, allowing for a faster development with more legible source code lexicalization.

In 1983, Bonar and Soloway debated how "it is widely known that programming, even at a simple level, is a difficult activity to learn", with this in mind they conducted a series of questions based upon the participants ability to construct a logical answer based upon the programmatic nature of their questions. Within these questions they contrasted parallels between different conventions of programming, where novices in the field could understand "role or strategy of statements more clearly than standard semantics" which would evaluate towards conclusive understanding that more natural language could be a syntax in itself. Later programming languages developed this view, by designing their syntax around an abstraction of natural language, such as within the Python programming language. The use of natural language when applied to programmatic problems, has yielded a vast adoption within programming languages — such as Python — with syntax closely mimicking spoken language that people can understand. A greater scope of development can be ascertained by people with a lesser understanding of knowledge on the subject. This is what Bonar and Soloway had concluded when conducting their interviews. Participants who were interviewed prefer the use of the English language's terminology, when reflecting upon cognitive programmatic questions.

In 1978 Kurtz evaluated that "if ordinary persons are to use a computer, there must be simple computer languages for them". Relating to the findings of Bonar and Soloway, in which under a cognitive approach their study concluded that with the semantics that are closely related towards a natural spoken language, reflected a greater understanding of programmatic concepts, as typical "programming languages do not accurately reflect the cognitive strategies used by novice programmers". Furthermore, Kurtz also concluded that the use of the programming language BASIC would be a suitable option for novice programmers (accounting for the time period of publication), due to an abstraction of removing unnecessary technical terminology should be taken into consideration when designing a programming language for the masses. Kurtz also noted that over the major different version of BASIC, there had been a constant differing of syntax and terminology, which resulted in the non-standardised structure to adhere to when designing the programming language, which when designing a simplified programming language can lead towards a collection of indecipherable syntax changes that no novice would be able to comprehend. An example of a more recent change to a programming language's syntax, is the change from Python 2 to Python 3, which reflected a major change in calling conventions within the language, which initially proved tedious for developers to adapt existing codebases for.

Later noted in 1999, Bullinger and Ziegler argued that there is not a programming language where the syntax is named without any form of cohesive dictionary. Typically, the lexical rules of a language are part of the syntactic layout of how functional operations interlink with one another, this links back to the arguments of Fedorenko *et al*, where an intuitive naming convention and consistency between functional operations would merit a programming language's ability to be simply understood. Further backing up the claims made by Kurtz, that there needs to be a "simple computer language" as with a structured syntax, a programming language would be easier to understand, compared to a complex programming language; such as esoteric programming language implementations, which are often far too complex for someone without a broad understanding of the programming languages to comprehend.

To conclude, the sources mentioned have equally noted that a programming languages' syntax needs to be easy to comprehend for a novice, yet still having a well-versed structure. To evaluate syntax as being simple, yet functional enough to develop a programming language, which would have wide appeal is a challenging endeavour, as a harmony between functionality and simplicity needs to be modulated. Yet if a cohesive syntax is designed, a language that it simple enough to understand — which Kurtz had argued — would admit usage from developers alike.

3. Project Specification

3.1 What is The Project Artefact

The software artefact of this project is a programming language interpreter, where the structure of the syntax is designed by the responses gathered from the user questionnaire. The interpreter is written in the C programming language, for a fast and resource efficient executable binary; with the aim of having a small memory footprint likened to the size of the executable.

3.2 What was Needed for The Project

The project was developed in the C programming language, which is a general-purpose programming language, that has wide industry appeal, and is used to create performant programs and systems. The C programming language was chosen as "C is a general-purpose programming language featuring economy of expression, modern control flow and data structure capabilities, and a rich set of operators and data types" (D. M. Ritchie *et al.* 1978). Without many high-level conventions, applications and systems are developed for both specialised and non-specialised environments, as its "generality and an absence of restrictions make it more convenient and effective for many tasks than supposedly more powerful languages" (D. M. Ritchie *et al.* 1978). Although the C programming language does have some caveats compared to modern languages, such as a lack of automated memory management; many benefits can also be deduced from this, as: pointers, and references; can be passed around and referenced by other parts of the system, whilst retaining memory associated with a reference.

Performance gains are also offered over other (interpreted) languages, as C is a compiled language, meaning that the source code is converted into native machine code, which is a much faster base of execution. Additionally, in the case of this project the compilation target is WASM which is low-level bytecode - which further proves the versatility of the C programming language as a means for development of the project. WASM compilation is achieved by using the Emscripten toolchain, as it is a near drop-in replacement to other cross compilers; but with using some additional compilation parameters a minified and interactive JavaScript file can also be generated, alongside a WASM binary, to allow for easy cross-communication between each format. The Emscripten toolchain can produce a WASM binary through as Emscripten is a "complete compiler toolchain to WebAssembly, using LLVM, with a special focus on speed, size, and the Web platform" (Emscripten 2022),

As a compilation toolchain, Emscripten, has extensive support and is one of the most widely used toolchains for WASM compilation. Additionally, there are minor steps made to make an existing C code base WASM compatible, via Emscripten compilation. Meaning that Emscripten can be used as a near drop-in replacement to other C cross compilers. As a complete compiler toolchain, the compilation target of WebAssembly, can be used on all major browsers, as WebAssembly can be easily integrated into any web-based application, as the:

> "[The WebAssembly] computational model is based on a stack machine in that instructions manipulate values on an implicit operand stack, consuming (popping) argument values and producing or returning (pushing) result values" (Rossberg 2019).

An additional front-end application had to be developed, to serve as an Integrated Development Environment (IDE) for an end-user. This system, however, was written in JavaScript, which is a programming language supported by all major browsers, that has further integration with WASM. JavaScript is a common programming language within the domain of website development. As an interpreted language, it can be handled directly by the browser, and integrated with a browser level Application Programming Interface (API).

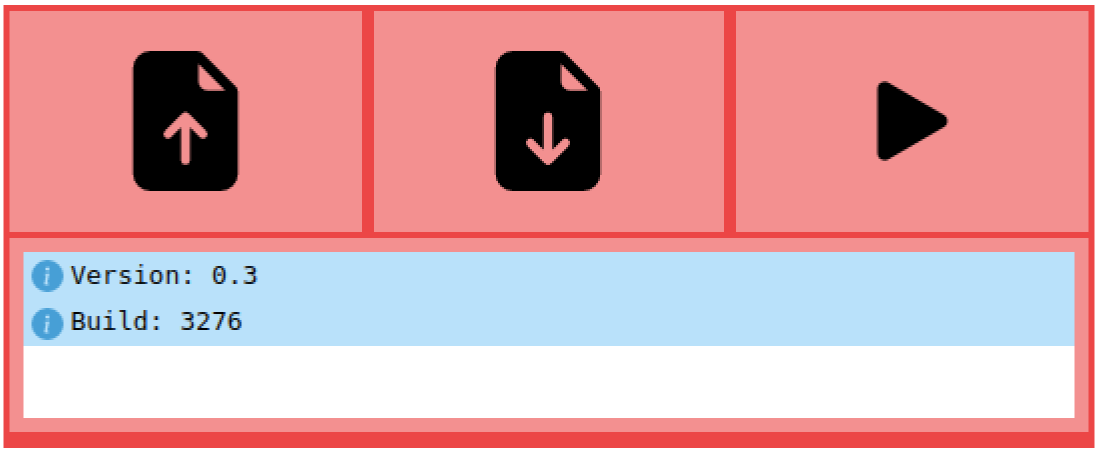
3.3 List of Requirements

Emscripten Compiler Frontend (emcc), which is a toolchain that uses Clang and Low-Level Virtual Machine (LLVM) to compile to WebAssembly. Also generated by emcc is a JavaScript file that would provide itself as a bridge between code written in C and

JavaScript that provides API support to the compiled code. A benefit of emcc using Clang is that it is often more performant than other cross compilers, such as GCC (GNU Cross Compiler), and would typically yield a smaller compilation size; by using an LLVM - which is a complete series of modularized compiler components and toolchains which can optimise multiple "programming languages and links during compilation, runtime, and idle time and generate code" (Alibaba Tech 2019).

3.4 Outcome

The desired outcome of this project was the definition and design of a new programming language; where an overall comparison of simplicity, compared to other languages, was to be the defining metric, whilst retaining the structure of what user responses would generate as the programming languages' syntax. This is evident in the generation of the Laturon programming language, as the overbearing feature-set consisted of terminology and structures based upon what the collected user data resulted as. Although this study aimed to design a programming language, the design and implementation of a WASM binary was also used as a targeted criterion, as a platform agnostic compilation target yields a wider user base. The WASM binary is executable from within the Laturon IDE, as seen in Figure 1.



```
Version: 0.3
Build: 3276
```

Figure 1: Image of IDE running Laturon from WASM binary

The output shown in Figure 1, is what is run on the initial load of the interpreter. The information that is outputted is versioning information, which is always shown to the user when opening the IDE. The versioning information are values which are populated during compilation. The build number automatically increments each time the interpreter is compiled, whilst the version number is incremented upon each release.

3.5 What Could be Done if More Time was Granted

If more time was granted, an implementation of a functional list system would have been feasible. Although initially designed and developed, as a system, the integration had proved to be a complex endeavour which would have delayed the base system of logical stack-based operations. Additional tests and runtime checks would have had to have been implemented, which would have further added to the development schedule. With a system as complex as lists (with nested references), there is an added level of memory operations which are also taxing to a system, an issue that would require additional memory checks to be complacent used within the system.

4. Methodology

4.1 Ethical Collection of Data

The ethical collection of data was attributed towards this project, as in order to gain an insight into peoples' own programming techniques, qualitative data had to be ethically collected. An ethical release form was submitted and approved by the institution (see Figure 13). This approval allowed the qualitative data to be collected.

4.1.1 Data Gathering

A digital user survey — in the form of a questionnaire — was accessible for participants. The collection of users submitted data was completed via a digital form, where people would submit their responses, and are then stored. In relation to the Research Question, the characteristics of a person's programming styles were identifiable, via the collection of responses to the questionnaire, resulting in gaining an insight, which was used within the implementation of the project. The questionnaire hosted a vast array of questions to gather insights into the person's depth of knowledge of computing terminology, as well as their ability to complete a formative task with logical constructs (see: Table 3, Table 15, and Table 16; for summary of user responses). With the collected data, it is then fed live into a spreadsheet, where automated calculations are used to denote the worth of the person's knowledge of computing skills, then mathematically normalising it against the other people's responses. However manual interjections had to be done to filter and remove unsolicited entries. Based upon the calculated worth of each user, their categorical selections — from their responses — are accounted for, and the sum of each person's worth totalled in the score for each category. The highest scoring category is then selected as the defined syntax structure from each response.

Initial surveys have been sent out, in the form of a digital questionnaire to individuals, technology agencies, as well as having the link for the questionnaire publicly accessible via online publications. The digital questionnaires allow for the moderation of data to be collected in a cohesive manner — where cohorts can be distinguished.

4.1.2 Legal and Ethical Issues

The data was provided by the user surveys, where all user submitted responses were stored in accordance with the 2018 General Data Protection Regulation (GDPR). The participants were made aware of the personally identifiable data that was to be collected, and how it would be used. In order to ethically collect the user data, a minimum age of 16 was imposed, and did not allow underaged participants to enter any additional information (or submit responses). Participants in the user surveys acknowledged the expressed agreement/permission of informed consent, which was used in order to regulate the design and attribution of this project with the results of the data that they provided. Additionally, participants gave consent to have their data sorted, and operated upon (and resulting in) the creation of a programming language via the shared syntax traits and logical understandings, which was designed by the amalgamation of all participants' responses.

Participants had to be accepting of the fact that by proceeding with the questionnaire, the data that they provided was shown/abbreviated within - this - published material; however, it did not show, or correlate any specific person in any way to the data within the published material. The mutual respect of confidentiality and anonymity was assured and agreed by the participant prior to the submission of their responses.

Additionally, two radio buttons: "Yes", and "No"; were given with a question of if the user "Would [be accepting] to be potentially contacted in the future regarding updates [...] of this study?", and for all participants who responded with "Yes" are to be contacted regarding the completion of the software artefact. Upon completion of the user survey, participants were also made aware of contact details, to contact regarding deletion of their data, or to enquire about any other questions regarding the questionnaire.

4.2 Methods

Qualitative data was recorded from users, however quantitative data was generated from the manual parsing of primary responses, to allow for numeric categorisation for certain responses which were interpreted to be of a similar styling. The interpreter was developed, alongside an experimental design methodology. Due to the project being of an experimental nature, as the complexity and anarchy of implementing different subsystems, to correlate with the user responses, proved to not be suitable for an otherwise Agile approach, as the interval between tasks could differ depending on the apparent complexity of the subsystem. So instead of operating with the confounds of Agile, the experimental methodology was employed instead, as "speed is an important factor during the exploratory phase of an experimental work" (Elio *et al.* 2011).

4.3 Project Management

The overall scope of the project was managed within routine intervals, which had a quick turnaround time (reflected below in Table 1). Each task was implemented at different stages, with multiple occurrences of overlapping development sprints of different tasks. However, a development turnaround of 10 weeks was achieved. Initial tasks were set out on a Kanban board, which was associated with the project, and each task was completed within succession of each other. The number of tasks to complete differed throughout development as the priority of feature implementation, and fixes rose above other tasks. The relative priority was a key focus in the management of development of the project, as the tasks yielded relative priority over other tasks, further allowing for development to be cohesively structured.

Table 1: Gantt chart of development

Task name	Week									
	1	2	3	4	5	6	7	8	9	10
Dynamic variable manager	■	■								
Array structure		■	■							
Nested array			■							
IDE			■	■	■	■				
File IO				■	■					
Tokenizer				■	■	■	■			
Parser					■	■	■	■		
Event execution loop					■	■	■	■	■	■
Testing						■	■	■	■	■

With optimum workloads, a timescale of an average of 3 weeks per feature proved to be obtainable, with an additional week later in development to link the feature towards other subsystems, as well as to perform changes to further cater towards the recently implemented subsystems. The benefit of later referring towards implemented features, yielded a cohesive bounty for testing — as appropriate — which had led to large scale fixes upon failed tests.

A set of milestones were expected to be completed within succession of one another. With having a set of milestones defined, the overall development could be measured upon the reaching of milestones, which within a short development time, the scope of milestone completion was an important endeavour.

4.3.1 First Milestone

Initially the first milestone was the gathering of user surveys, which took longer than expected to get all 17 responses. However, once enough responses were accumulated from the user surveys, the data was used to perceive an insight into how well versed a person is with programming knowledge. From the data collected, it was apparent — those who knew how to program — had certain biases towards one way of programming which they stuck to, compared to others who were not skilled in this area that were having a collective struggle to keep a consistent styling towards their responses. With the data — from the questionnaire — having been fed live, into a spreadsheet (as seen below in Table 2), the use of automated calculations also presented itself to have an ever-changing effect on what the selected outcomes were.

Table 2: Values to calculate, and calculated user worth

User submitted data					Automated calculations		
Experience (years)	Age (years)	Exposure	Outside exposure	Terminal usage	Worth	Worth expressed	Worth normalised
2	32	33	1	1	-15.07209263	5.92790737	65.42%
1	20	3	0	0	-19.43703516	1.562964841	17.25%
0	52	2	1	0	-20.34364435	0.1969066937	2.17%
1	47	7	0	0	-19.14393171	0.5568204868	6.14%
2	27	12	0	1	-16.7775524	4.2224476	46.60%
5	35	21	5	1	-12.28473631	8.715263692	96.18%
5	38	28	2	1	-12.27180527	8.728194726	96.32%
5	39	28	4	1	-11.93847194	9.061528059	100.00%
5	18	7	0	1	-14.14393171	6.856068289	75.66%
2	26	31	2	1	-15.05197769	5.948022312	65.64%
0	18	3	2	0	-20.10370183	0.2688894523	2.97%
1	18	1	1	0	-19.41692022	1.583079784	17.47%
5	22	24	1	1	-12.73157539	8.268424611	91.25%
5	22	16	3	1	-12.98444895	8.015551048	88.46%
0	43	0	1	0	-20.49019608	0.19	2.10%
5	51	14	0	1	-13.63100068	7.368999324	81.32%
1	22	2	1	0	-19.34364435	1.656355646	18.28%

$$f = ((\frac{x \in c}{\underline{c}} + \frac{x \in d + 1}{6} + x \in e) - (max(b) - \underline{b} - x \in a))$$

$$g = (\lceil |min(f)| \rceil - |x \in f|)$$

$$w = \frac{x \in g}{max(g)}$$

Equation 1: Calculation used to denote a user's worth

The "Worth" of a participant's responses, was calculated using the algorithm (see Equation 1), where the value for each user is individually calculated to be the value of w, with the values of: a, b, c, d, e, f, and g; all representing their associative alphabetical stance in relation to their column position, as with each growing letter being an incrementation across the columns. The basis for the calculation consists of an assortment of operations that stem from the user submitted values in the columns: "Experience", "Age", "Exposure", "Outside exposure", and "Terminal usage". Where the value that is further utilised, is the "Worth normalised", as with normalisation calculations, the fluctuations of positional changes are relational, which showed changes each time another person submitted their responses to the questionnaire, all based upon the user data. The calculation of Equation 1, is performing calculations on individual user responses, in correlation with all other users' responses, to generate a relative understanding of their worth, which is the "Worth" of each user, which is then expressed to a positive coherent value ("Worth expressed") prior to being represented as a normalised value. The normalised value — itself — is used within the valuation of each user's responses for later questions within the questionnaire.

The collection of user data — provided by the questionnaire — is automatically stored within a spreadsheet, which allows for the data to be correlated, and operated on, independently of any additional interactions. The calculated data which accompanies the user provided data, is also organised alongside, to allow for identification of trends, and commonalities between entries. This allows for a simple view of submitted data, and calculated results, within an organised structure, further allowing for the data to be easily transferable, and conspicuous regarding the depiction of calculated components from the user data.

For the digital collection of data performed by the questionnaire, the service Google Forms was utilised. As compared to other digital questionnaire services, it allowed for raw text fields to be entered, which allowed the user to put indentation within their responses — without it being removed by the provider — like other competing services. Also another additional benefit from using Google Forms, was the integration with Google Sheets, which allowed for automated data calculations to be performed; such as the calculation for the users' responses' worth (see Equation 1), which was written as a formula that Google Sheets could interpret and operate on, even when the user responses were coming in live.

4.3.2 Second Milestone

Once the 17 respondents had submitted, the valuation of user responses, could be utilised from using the scoring of users' worth, and their responses were accounted for, with the "Worth normalised" providing merit to their responses, which led to the categories selected by each user, being calculated not by occurrence, but by the participants' individual worth ("Worth normalised"). This resulted in the structured language syntax being formatted alongside the calculated responses, and later the differences between responses were accounted for, and a complacent syntax was deduced.

Table 3: User responses for programmatic questions

Worth normalised	Variable declaration	Addition function	Zero validation	Greater than validation	Output
65.42%	1	1	1	1	1
17.25%	1	7	2	7	3
2.17%	2	6	2	2	4
6.14%	2	2	Did not know	4	1
46.60%	2	3	3	3	1
96.18%	3	5	5	4	3
96.32%	4	4	4	5	3
100.00%	4	5	7	8	3
75.66%	2	7	6	6	3
65.64%	4	5	7	6	4
2.97%	Did not know	Did not know	Did not know	2	2
17.47%	Did not know	2	2	Did not know	3
91.25%	2	8	7	9	3
88.46%	2	9	8	8	3
2.10%	2	2	Did not know	2	1
81.32%	2	Did not know	Did not know	Did not know	3
18.28%	2	Did not know	9	Did not know	5
Selected	2	5	7	8	3

From the user survey, participants were asked to write a rudimentary program, where they were tasked to write any logical construct that met the target criteria of the questions. Based upon user responses the most common format for each question was generalised and categorised. For each question, the resulting "selected" category is selected based upon the total occurrence of the category with the valuation of each being attributed to each user's worth ("worth normalised"); which are shown above in Table 3. The first question asked was used to collate a way of a general "variable declaration", which was questioned as: "Declare a variable called `test`, and set the value to the floating-point number 1234.56". Based upon user responses category 2 was selected (see Table 10). However, there were two participants who did not know how to answer — in this case — so their responses were not accounted for in the selection calculation.

The second question that the participants were asked was to write syntax for an "addition function" which was phrased as "Write a function (called `add`) to add 2 different parameter values together and return the result". Based on the user responses category 5 was selected (see Table 11). Yet in this case three different participants did not know how to answer. Third, another function was requested from the participants. This time a "zero validation" which is to check if a value is equal to 0, which was phrased as "Write [a] function (called `isZero`) to validate if a parameter is equal to 0; and should return: True, or False; based upon the logical operation you'll write". The results for this question resulted in category 7 being selected (see Table 12). Increasing from the previous question, four different participants did not know how to answer this question.

The fourth question required participants to write a "greater than validation" where responses could be tailored by slightly changing the operator, without having to ask a broad range of questions relating to the different types of relational logical operations (see Table 5). The question asked was "Write [a] function (called `isMoreThanFive`) to validate if a parameter is greater than 5; and should return: True, or False; based upon the logical operation you'll write". Compared to the previous question asked, only three participants did not know how to answer. The resulting category for this question was category 8 being selected (see Table 13). Finally, the participants had one final question where they were required to write source code. A simple program to output text was asked as "Write a piece of code that'll output the text: `Hello world!`". Which compared to all prior questions all users were able to answer, with no participant not knowing an answer. The selected category for "output" was category 3 (see Table 14).

4.3.3 Third Milestone

Within the second milestone, a set of objectives were set out, each being: a front-end IDE, tokenizer, as well as a parser, which all had to be completed one after the other. These objectives consisted of, each a:

1. Graph data structure,
2. Front-end IDE,
3. Tokenizer,
4. Parser,
5. Variable management system,
6. Executor.

However, a change from a graph data structure did occur between the tokenizer and parser, to be an execution stack instead. An in-depth development of these objectives is explained in the section Design and Implementation.

4.3.4 Fourth Milestone

Concluding the milestones of developments, testing occurred. The tests spanned from internal (self), to external (participants); test, and allowed for the debugging and implementation checks of the objectives declared in the Third Milestone. The inclusion of external tests allowed for an unbiased way of programming to be observed and monitored for any issues with the interpreter. A further in-depth explanation of internal, and external tests, occur in the sections: Design and Implementation, and Results.

5. Design and Implementation

The language's syntax was defined from the user responses. Which resulted in standardised conversion being established for the Laturon syntax.

The standardised use of mathematical notations (chosen from user responses — shown in Table 16), allowed for a simple construct of arithmetic and logical operations, which are defined in Table 4. As well as the definition of using braces, and a function definition of "function" (results calculated in Table 15) proved for a well-defined construct that loosely mimicked other programming languages.

Table 4: Arithmetic operators

Operator	Action
+	Addition
-	Subtraction, unary minus
*	Multiplication
/	Division
%	Modulos

The: +, -, *, and /; operators (as shown in Table 4) work in the expected fashion. The % operator returns the remainder of an integer, or floating-point division.

Table 5: Relational and logical operators

Operator	Action
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
&&	And
||	Or
!	Not

The relational and logical operators (as shown in Table 5) are used to produce Boolean (true or false) results. The outcome of any of the relational or logical operators will always evaluate to a Boolean value or would throw an error to the user that the datatype of the input is not of an operable type. Additionally, numbers that are nonzero will equate to true, otherwise false. The logical operators: &&, || and *!*; are used to compare two Boolean values, or in the case of *!*, which is used to reverse a value.

Initially a graph data structure, which operated purely on variable references, was constructed, consisting of nodes holding up to two different C pointers and references towards different structures.

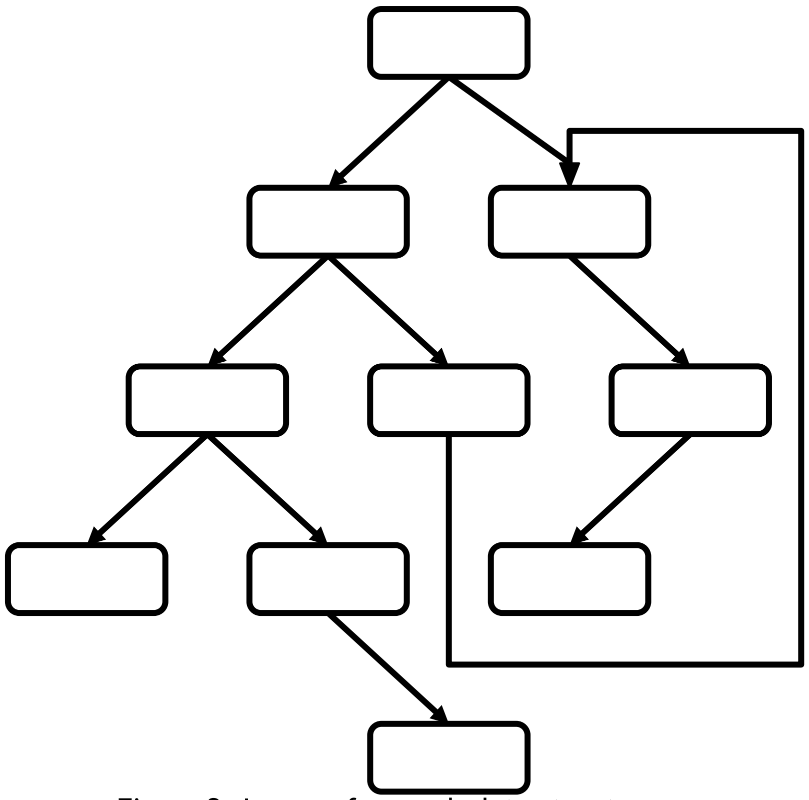


Figure 2: Image of a graph data structure

This initially proved to be a tedious endeavour as by having each operation and value referring to up to two different nodes, added further complexity. However, once a defined structure between nodes was designed, logical and mathematical operations between different nodes were implemented. A rudimentary render of the designed graph data structure (with a small number of nodes), including a functional reference, can be seen in Figure 2. An important design decision when implementing mathematical operations was the predetermining of numeric underflows and overflows, on integer and floating-point values; as although a wise programmer would avoid this sort of state, a novice programmer could not be aware of such things, so if one was detected, then a warning could be given to the user to notify them of this.

Once the initial graph structure was developed, the development of a front-end web-based IDE was developed. Which is completely generated by JavaScript on a client's machine. This choice was done to provide system specific changes to the editor where certain browser APIs differ depending on the browser vendor. With the IDE being locally generated via JavaScript, state changes of elements could be encapsulated, within a host scope, which avoids exposing internal function calls as a potential attack vector for arbitrary execution of source code. The IDE was designed to be usable on both desktop and mobile devices. Both types of devices are targeted as the portability of the programming language is an important design goal of the project; and with the support of WASM within all major browsers, further emphasis on a device agnostic WASM execution influenced the design on mobile devices.

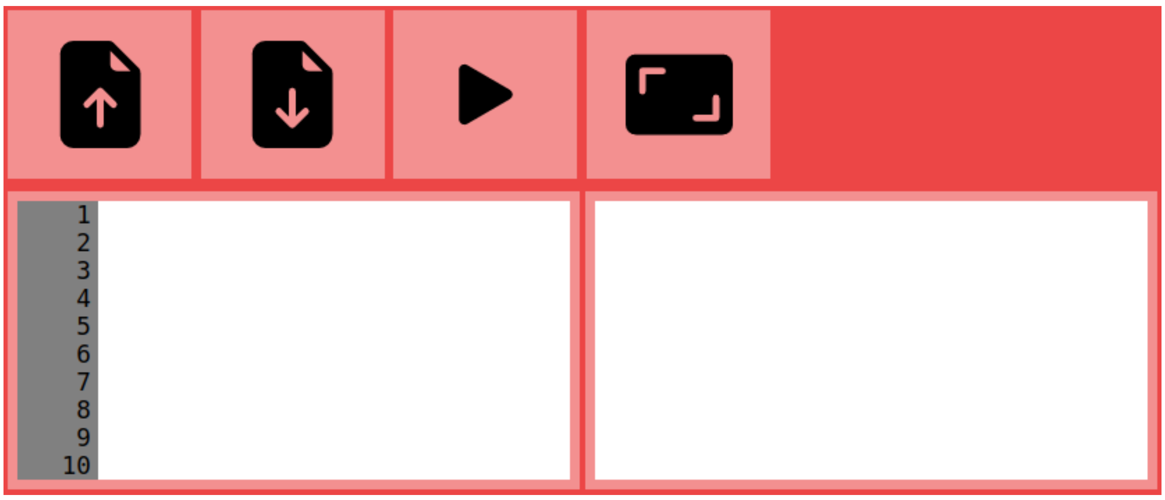


Figure 3: Image of IDE (rendered by the Blink Chromium rendering engine)

With the IDE being usable on both types of devices, the input and output had to be interchanged on mobile devices (or any window with a viewport width less than 600 pixels), the split state of a user editable area, alongside the programs output, had to be changed to account for, and instead the entire width of the window would be filled up with the user editable area, and would be replaced with the programs output during execution. However, on a device with a large enough viewport, the split state of both the user editable area, and the execution output would be kept, which can be seen in Figure 3.

Once the IDE was developed, interactions both ways (inputs and outputs) could be accounted for, which led towards the development of a tokenizer, which would read user source from a file that was sent across from the IDE. Although within a web-based environment, no filesystem exists, so a rudimentary filesystem is implemented via Emscripten, which emulates all filesystem operations that any C program could require. The tokenizer reads data from the user source file. The design of the tokenizer was based upon the syntax designed by user responses. When executed, the tokenizer would read each character of the user source file, one by one, and would construct an identifier string consisting of each character, until the string met certain requirements, such as the string being the text "function" (a function definition), or if the actively read character was of a specific type, such as "+" being an operator. With the characters and generated strings being verified upon each step, a token for each type of operation would be generated, which held accordant structures of numeric, floating-point, and string literal inputs; generated by the identifier string and would be linked to the previous within a linked list.

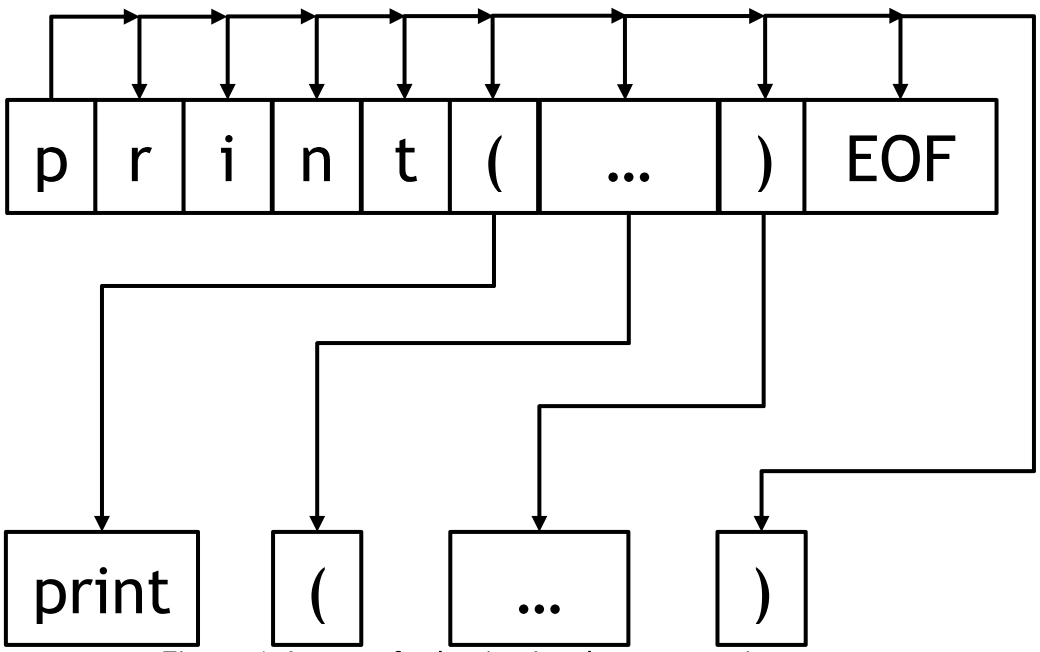


Figure 4: Image of tokenizer's token generation process

Within the tokenizer if certain criteria were met, then previous or active tokens would be modified to account for this. A case of this would be if the previous token type was declared as a variable, but the next character was a "(" (open parentheses), then the previous token would be converted from a variable, to be a function call. As this is the designed function calling convention of the program, any additional function call parameters would be accounted within the scope of the function call's parameters, until a matching scoped character of ")" (close parentheses) was utilised - meaning the function call had ended. This can be seen in Figure 4, where the program consists of a print function, and the contents of the function's name are declared once the character of "(" was detected. This then created a variable token with the contents of "print", and the current being an open parenthesis would then switch the value type to be a function, similarly all other contents that are read that met similar conditions, would subsequently be generated into the appropriate tokens (or later converted), until the current character is the end of file (EOF). Once the EOF has been detected, whatever is currently held within the current string, a generic token can then be generated.

With a tokenizer complete, parsing of each token could be obtained. Although the tokenizer made logical changes towards tokens, a parser was required to organise and validate each token. Initially this was done to match the graph data structure, however due to the complexity of the graph data structure, with a limited number of node connections, a purge of that code was required. This was done to allow for faster operations, whilst allowing for the same syntax to be usable by the user. So, a stack-based data structure was designed, a Last-In-First-Out (LIFO), which would preserve the positions of tokens (when executing) but would still allow for performant operations on values to occur. With this new stack-based operation defined, organisation of user submitted code could be done; for example, if a user attempted to execute the following code:

```
call(a,b,c)+(d*(e/f))
```

Internally tokens generated by the tokenizer, would still be represented as that structure, however the parser would need to convert it towards stack base operations. Which would have the contents internally represented as:

```
(a,b,c)call(d(ef/)*)+
```

The reason for this is that within this designed execution stack, the values are initially pushed onto the stack, and are only popped out of the stack once the appropriate operation has concluded. This is one of the most computationally intensive parts of pre-execution as the dynamic reorganising of tokens towards different memory addresses needs to be done, in order to have this internal representation complete.

Once all tokens have been reorganised by the parser, numeric attribution is then done towards: function definitions, function calls, and variable references; this is done before execution as validating matching identifier strings, for each token during execution (where certain tokens could be operated on multiple times), would prove to slow down the execution of the program. So numeric attribution is done prior to execution. This simply replaces each function reference (definition or call), with a numeric function identifier, and variables are also treated similarly with numeric variable identifiers. However, for variables their starting value is 3, this is done as the first three values are reserved for: false, true, and null; values, but for any variables that are defined within a function (that are not in the global scope), their numeric values are reset to count upwards of a starting value. This is done to allow for a larger number of variable references to exist other than those explicitly defined within the global scope. All variables that are declared on the global scope are automatically created before execution, whereas all variables that are declared within functions are created when declared within their associated function's calling. All variables are managed within heap memory and are then later inserted into the execution stack when a variable is referenced. A variable management subsystem was designed and constructed, which would store all variables within a continuous array which held all data associated with each variable, such as the state of value assignment. This subsystem would be utilised for the generation of variables as well as the assigning of values to each variable. Another part of the variable subsystem is the automated variable garbage collection, as for when a variables' declaration is no longer in the active scope of execution, the garbage collector will free the memory used by those variables.

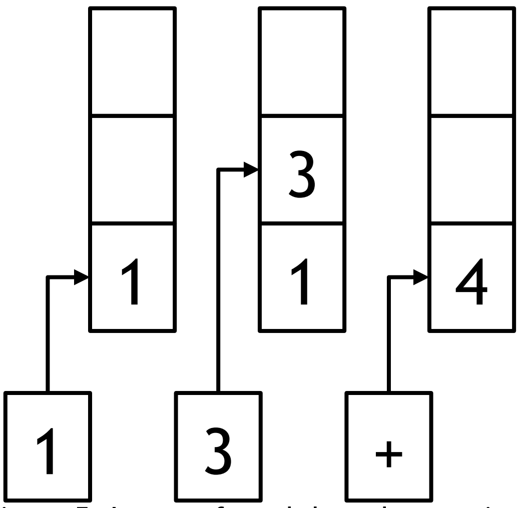
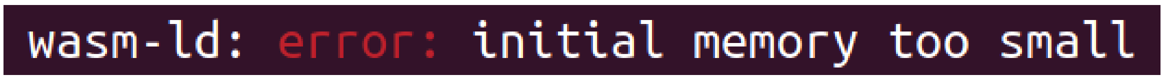


Figure 5: Image of stack-based operations

Once the parser was constructed, and was correctly organising tokens, the execution stack could be developed. This loosely followed the operations that were originally purged from the project, where different values were operated on, but in this case were pushed onto the stack with succession. On the occurrence of an operation that would use the values on the stack, the values were either updated, or popped off of the execution stack as seen in Figure 5 which pushes two different values onto the stack, then with the addition operation token, pops them both off to push on the sum of the two values. The execution stack had to step over each token, and perform operations on the stack, but prior to each operation, a set of checks were done to validate if the state of the stack met the requirements of what the operation required. Such as if two variables were being added together, but had no value assigned to them prior, the execution stack would throw a predetermined runtime error which would reference the issue to the user, and then terminate the program. Some of the checks that were implemented within the original graph data structure operations were once again, implemented within the execution stack, such as the predetermining of numeric underflows and overflows, on integer and floating-point values.

Throughout the development of the execution stack operations, multiple issues arose which were not originally determined to be expected, one of which was the stack size not being large enough to perform any form of recursive operation. Also, upon resizing the execution stack to a much larger value yielded errors, which occurred with the linkage of objects when compiling. This was due to the execution stack being too large for the memory defined within the compilation arguments, which resulted in a linkage error — see Figure 6 (below).



```
wasm-ld: error: initial memory too small
```

Figure 6: Error message for initial memory too small

So, a length of the execution stack had to be negotiated with the Emscripten toolchain, to allow for a large number of nested operations, yet also fitting within the confounds of the memory constraints. It was later noted that referencing to heap values avoided errors, due to the size of the execution stack being too large to hold sequentially within memory, so by having the stack retaining the size, but only referencing pointers prevented this issue.

An additional issue that should have been expected in hindsight was variable references not being accessible within the current function scope; this led to an exception being thrown (see Table 8 — error code: 7) to the user before termination, but it was later noted that in an effort to save memory a function scope negation counter was being omitted, due to the token being removed, which did not negate the count of function stack calls. However, to fix this whenever a function had reached the end of its tokens, it would automatically negate this value, which fixed the variable access error.

An important feature-set of any programming language is to handle inputs and outputs, which was one of the last things to implement. Depending on the compilation target inputs and outputs are handled differently. With a WASM target, inputs and outputs required translations between different data constructs of the WASM which internally represented characters within a JavaScript array buffer, so a conversion had to occur, before being posted as a message to the IDE via a web worker message; whereas if it was a native executable the values would just be written stdout and read from stdin accordingly (on UNIX based systems).

A web worker was selected as it allowed for parallel execution, next to the main event loop of JavaScript, so the main user interface (UI) would not hang until the execution of the user source code would terminate (if not within a user created infinite loop). A web worker is a browser technology that, like WASM, is too supported by all major browsers, and operates through a simple message interface, which is more than what is required for Laturon to operate, as only textual inputs and outputs are currently supported. With parallel execution a user is also able to interact with the IDE, and can edit source code, or interact with the execution — such as stopping their running program — whilst still having an operational UI, which proves to be a much better user experience.

Testing of the execution of the interpreter could be done simply by writing valid source code that met the designed syntax. This was a fitting way to test the program, as the development of tests within its own syntax was a proof of the viability of the structured syntax.

```
print("Hello world")
```

Figure 7: Hello world in Laturon programming language

```
name = ""
while (len(name) == 0) {
    print("What is your name?")
    name = input()
}
print("Hello " + name)
```

Figure 8: Hello name in Laturon programming language

The tests outputted the expected outputs for what they programmed to output. Figure 7, was to output the text "Hello world", of which it accurately did. This was a simple test to deduce if a string could be outputted from a stack reference. Whereas in Figure 8, the test also introduced the ability to take in an input from the user. This input would too be referenced from the stack, and the length of it would be validated to be equal to 0, if this was proved to be true, then it would be repetitively asking for the users' input while this condition was met. Then once the length of the input exceeded 0, the program would concatenate the input to another string, and output the new string to the user.

```
function fizzbuzz(max) {
    num = 1
    while (num <= max) {
        output = ""
        if (num % 3 == 0) {
            output = "Fizz"
        }
        if (num % 5 == 0) {
            output = output + "Buzz"
        }
        if (output == "") {
            print(num)
        } else {
            print(output)
        }
        num = num + 1
    }
}
fizzbuzz(100)
```

Figure 9: Fizz buzz function in Laturon programming language

Further tests were conducted, such as Figure 9, which was an implementation of FizzBuzz, which is a logical programming task used to identify divisible operations for two certain values, where it is "a program that prints the numbers from 1 to 100. But for multiples of three print 'Fizz' instead of the number and for the multiples of five print 'Buzz'. For numbers which are multiples of both three and five print 'FizzBuzz'" (Ghory 2007). This outcome was achieved through the test source code, and outputted the correct sequence, see Figure 10 (below) for a partial sequence of the output.

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
Fizz
22
```

Figure 10: Partial output of Fizz buzz function sequence (from 1 to 22)

```
function fibonacci(max, output) {
    n1 = 0
    n2 = 1
    count = 0
    while (count < max) {
        if (output) {
            print(n1)
        }
        nth = n1 + n2
        n1 = n2
        n2 = nth
        count = count + 1
    }
    return n1
}
total = 50
number = fibonacci(total, false)
print("The " + str(total) + "th Fibonacci number is: " + str(number))
```

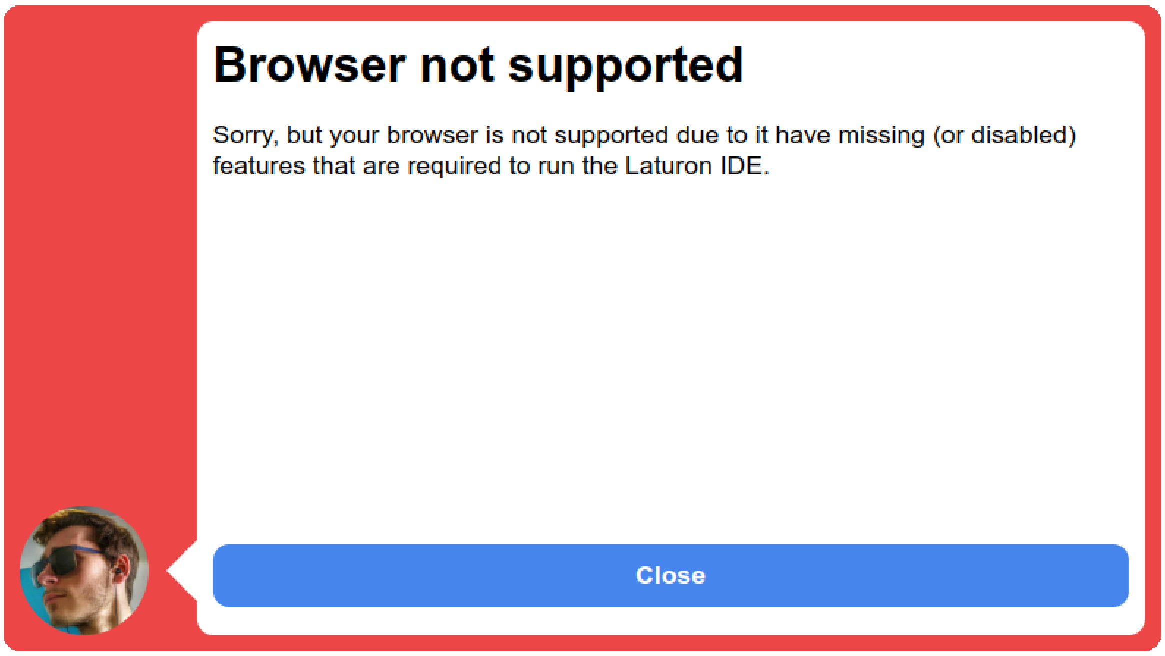
Figure 11: Fibonacci sequence in Laturon programming language

A final test of the programming language was to calculate a Fibonacci number at a certain position, which was conducted with the source code within Figure 11. This test employed looping, as well as an assortment of variable addition and logical valuations, to calculate the Fibonacci number at the position of 50, which was given as a parameter (via a variable value) to the function, which then calculated the correct value and return, where it would be outputted to the user. Supplemental tests occurred with debug flags set on the compilation, and the compiled binary with debugging symbols, would then be executed by the GNU Debugger (GDB). GDB monitored memory: allocations, deallocation, and references; and had detected mishandled memory where a few segmentation faults occurred under certain conditions due to attempting to read from a portion of memory which was set to NULL. These were later remedied by adding checks for NULL values, and then a runtime error would be thrown letting the user know of an error with memory allocation which would output the text "Failed to allocate memory" (see Table 8 — error code: 2).

When implementing some changes, subsequent systems had broken which were not evident until push and reviewed. This resulted in a change to the stack operation system, where logical operations are reorganised. Multiple iterations of subsequent changes were put in place and were tested upon until the right combination of variables were met.

The change involved reorganising the logical and arithmetic operations from their previous position. Throughout the many iterations of development, misalignment of these tokens occurred, and resulted in the execution stack not having the correct values pushed or popped off of it. This broke the LIFO structure of the execution stack. However, this was fixed by removing a subsystem used to move operations, and instead adding functionality to the functional alignment subsystem, which prior to this change moved the function call to after the parameter values were pushed onto the stack. A slight rewrite of the subsystem, to add the checks to allow for logical and arithmetical operations, to be reorganised with the relative proceeding tokens type being accounted for, to allow for mathematical statements holding the same operations, yet being correctly reorganised for accurate execution stack stepping.

However, people who had conducted the user survey — who had consented to being contacted regarding updates to the project (see Legal and Ethical Issues) — had been given access to the IDE and interpreter were able to test the programming language for their own uses. This was done through their own observations of the language as well as looking at some example programs for inspiration. The examples in: Figure 7, Figure 8, Figure 9, and Figure 11; were provided towards the people who were given access to test the programming language. Upon their initial observations of the Laturon programming language and IDE, a mutual consensus of interest and appreciation of the syntax was immediately apparent (see Results for responses). However, some initial errors with the IDE were discovered, such as the failure to initiate the Laturon interpreter on certain releases of the Safari browser, on devices that did not have 64-bit processors. However, the same issue could be expected to occur on devices that have lower than 64-bit processing. However, due to the interpreter requiring a 64-bit processor to perform operations on values that are larger than 32 bits; a check was implemented to alert the user of their browser being unsupported as seen in Figure 12 (below).



Browser not supported

Sorry, but your browser is not supported due to it have missing (or disabled) features that are required to run the Laturon IDE.

Figure 12: Browser not supported error message

Another issue that was detected by a person who was testing their own code was that variables were being deleted once a scope was closed that was resting on the global scope, and not within a function. This was due to the negation of an internal scope reference variable which was being negated before the cleanup of variables within the scope, where instead it should have been negated after the cleanup of variables within a scope. However, once this fix was implemented the correct garbage collection routine was operating correctly, which freed memory used for variables that were declared within that execution scope, and not the global scope.

Table 6: Built-in datatypes

Numeric reference	Datatype
1	integer
2	float
3	string
4	boolean
5	list
6	null

The Laturon programming language has 6 built-in datatypes, which as listed within Table 6, each of the datatypes are associated to variables, and tokens. The contents within variables are dynamically set upon each assignment to the respective variables within their unionised construct, but the datatype declaration (of the variable) is set to the corresponding datatype of the variable's contents. Numerical values are set to either an: "integer" (see Table 6 — numeric reference: 1), or a "float" (see Table 6 — numeric reference: 2); where an "integer" is a whole number, whereas a "float" is a floating-point number, both of which are signed, to allow for negative and positive deviations of the number. An "integer" can hold any value between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807, which is the confounds of a signed 64-bit integer (signed long long). Like an "integer", a "float" is a signed 64-bit (long double) floating-point datatype, which can hold values ranging between: 2.22507385850720138e-308, and 1.79769313486231571e+308; with a negative or positive signage both at the extent of and internally can hold up to 15 digits of precision. This is due to an internal limit of 53 reserved bits for precision of the mantissa (IEEE 2008, pp.1-70). Also, a numeric value is a "boolean" (see Table 6 — numeric reference: 4), which internally builds upon the same datatype of an "integer", however it limits the value range to: 0, and 1; equating to: false, and true; respectively.

Unlike the previous two datatypes, a "string" (see Table 6 — numeric reference: 3) is a C pointer to a memory location within the heap. The location within the heap holds the contents of the string. The length of a string can be as large as possible — with fitting within the heap/dynamic memory of the interpreter — which is referenced by the C pointer, so a string can be stored anywhere in memory, not necessarily in succession of a variable or token. Similarly, to a "string", a "list" (see Table 6 — numeric reference: 5) datatype is a C pointer towards a memory location which holds a unionised value (consisting of all datatypes), with other list items referenced as a linked list. Although internally within the source code of the Laturon programming language, a "list" is referred to as an "array", it functions as a linked list, as it does not require the contents of the "array" to be of an explicit type and can have nested lists within itself.

Unlike all other datatypes, a "null" value (see Table 6 — numeric reference: 6), is used to identify the absence of a value; this datatype can be declared as the value of a variable. Additionally, if a function is called, in place of where a value would be assigned, and no returned value is declared, a null value will automatically be constructed.

Table 7: Built-in functions

Function	Total expected parameters
print	1
input	0
str	1
int	1
float	1
bool	1
len	1
type	1

Within the contents of Table 7, a collection of built-in functions is listed. These functions are all usable from within the Laturon programming language. The first two functions are primary functions, where input/output (IO) is defined, as the only two points where Laturon can: output ("print"), and input ("input"); text to the user. The first function, "print", outputs text to the user, can accept any datatype (see Table 6), and will format the respected datatypes as a string to be outputted, and returns null. However, unlike all other built-in functions, an "input" accepts no parameters, as this function is used to get a user's input, which will be returned from this function as a "string" datatype (see Table 6).

Each: "str", "int", "float", and "bool"; are used to convert a value to a different data type. The function "str" converts any datatype to a "string", whereas "int" converts a value to an "integer" but will only convert the following datatypes: "integer", "float", "string", "boolean"; any other datatype, or an invalid "string" equivalent conversion, will result in the fatal error of "Failed to convert to another type" (see Table 8 — error code: 4). The "float" function will convert any: "integer", "float", "string", or "boolean"; to a "float" datatype, any other datatype, or in the case of an invalid "string" equivalent being used as a parameter, a fatal error of "Failed to convert to another type" would be thrown to the user. Similar to the "int" function, the "bool" function converts the value to a "boolean" datatype (see Table 6), but for any numeric datatype ("integer" or "float") the conversion would result in a Boolean true if the value is equal to 1, otherwise it would result in the value of Boolean false being returned; however, for a "string" or an "list" Boolean true will be returned if the length of either if more than 0, otherwise Boolean false would be return, any if a "null" value is given, a Boolean false will be returned.

The function "len" returns the length of the value as an "integer", this function accepts only a "string" or a "list" datatype, otherwise the fatal error of "Failed to execute operation" (see Table 8 — error code: 7) is thrown to the user. In the case of a "list" datatype being used as the function's parameter, then only the first level listings will be accounted for in the length, and the length of all nested lists will not be totalled in the returned value. The final built-in function is "type" which accepts any datatype as its parameter, yet it will always return a "string", as this function returns a literal datatype definition to the user, the respective datatype will be the literal contents of Table 6.

Table 8: Fatal error messages

Error code	Error message
1	An unknown error occurred
2	Failed to allocate memory
3	Logical operation failed
4	Failed to convert to another type
5	Attempted to use a value from a variable has not been assigned
6	Failed to cleanup variables outside of current scope
7	Failed to execute operation
8	An invalid reference to a call stack scope occurred
9	Failed to correctly read in user input string
10	A reference to a compound literal does not exist
11	An array routine was not given an array to operate on
12	A reference to an item within an array that is out of range
13	Failed to perform an operation on source file
14	The source provided has invalid syntax
15	The total amount of stack memory available to execute your program has been reached
16	Too many functions or variables are declared within your program to be handled within memory
17	Zero division error
18	You are attempting to use syntax that has not been finalised

The error messages in Table 8 are outputted to the user when a fatal error occurs with the interpreter, these errors can occur at any of the three stages that the interpreter operates in: tokenization, parsing, or execution; these types of errors are general errors that can stated at any time for any relative reason, such as "Failed to allocate memory" (see Table 8 — error code: 2), which would be outputted when memory allocation false, which typically occurs if not enough memory is remaining to meet the allocation requirements. Once a fatal error has been declared, it is outputted, and the interpreter — at any state — is terminated. Certain errors are only outputted when in the execution state of the interpreter, such as "Attempted to use a value from a variable has not been assigned" (see Table 8 — error code: 5) or "Zero division error" (see Table 8 — error code 17), as these types of errors can only occur if a users' programs' source code does have a proper control flow structure.

If a fatal error is declared, and is not listed within the error messages of Table 8, by default it would default to "An unknown error occurred" (see Table 8 — error code: 1), additionally this error message is intentionally given to the user when certain unexpected values are present. Fatal error messages are also appended with text identifying the line number that the error originated from, if the error originates from a token generated from the users' source code. Although an error message of "The source provided has invalid syntax" (see Table 8 — error code: 14), there are also more explicit syntax error messages as defined in Table 9.

Table 9: Syntax error messages

Syntax code	Error message
1	Invalid syntax
2	No function definition or reference was given
3	Unable to call a function that has no reference
4	No variable definition or reference was given
5	Attempting to open a scope within an expression
6	Closing a scope that has not been opened
7	Closing a parenthesis that has not been opened
8	Closing a bracket that has not been opened
9	Invalid function definition
10	Too many parameters were declared to be handled within a function
11	Invalid variable definition
12	Unable to define a function with a recursive function definition
13	Unable to define a variable with a recursive variable definition
14	The line ended abruptly
15	A variable is referenced but not utilised
16	A variable has a name that is not supported
17	A function has a name that is not supported
18	The total amount of parameters used in a function call does not match the function definition
19	A parameter has the same name as a variable that is declared within the global scope
20	An invalid operation was defined within the syntax
21	An invalid numerical value was given
22	Unable to return when not use within a function
23	Attempting to return where there is nothing given
24	A statement was declared after a return without closing a function scope
25	Attempting to remove list element where list is not present
26	A break statement has been invalidly given additional parameters
27	A while loop was declared incorrectly
28	Unable to convert numeric value to a numeric type

29	A function was defined that already exists
30	A reference to a variable could not be established due to an invalid format
31	A functions parameters were incorrectly defined
32	A string was incorrectly terminated
33	A list was not closed off
34	An open scope was not closed
35	A dangling variable reference occurred prior to an operation

Unlike the fatal error messages in Table 8, syntax error messages (Table 9) differ in the sense that they give an explicit explanation of the programmatic error, which relates to the users' source code. With all the different types of syntax, and how syntactic operations can interact, a wide assortment of possible error messages are required to give the user the appropriate error message to help them fix their programmatic issue. With multiple variants of syntax having the possibility of being wrongfully organised in different ways than expected, it is the job of the tokenizer and parser, to understand the syntax that the user provides, these subsystems are the ones that would output a syntax error if the users' source code provided is not in a structure that it can understand. All syntax errors are also appended with text identifying the line where the syntax error originated from.

If a syntax error is declared, and is too not listed within the syntax error messages of Table 9, by default it would default to "Invalid syntax" (see Table 9 — syntax code: 1), additionally this syntax error message is intentionally given to the user when broadness of the error is too large to be generalised into the defined syntax errors.

6. Results

From initial use of the Laturon programming language, the devised syntax was simple enough to develop applications in, and was also efficiently operable within a browser environment, due to the WASM compilation target. Although secondary tests were utilised, from participants; those of which that had been given access to the Laturon web-based IDE, had given initial feedback, of which is as follows.

Most of the user feedback reflected a similarity to other programming languages, as aesthetically it has commonalities to Python and JavaScript. However, an influence of Python was declared by multiple people, for example P1 had stated:

> "The decision to remain as close to Python's syntax makes the code easily readable. Utilising curly brackets to encompass snippets, and likewise parentheses to note conditions adds clarity to their purpose. As parentheses are used to pass arguments to functions, it makes sense that the code within the parentheses are 'arguments' to the logic condition."

Whilst P2, had similarly stated that it too has had a Python-like structure, which was declared in their statement of:

> "Structure-wise it is almost pythonic which allows it to be used for throwing together quick scripts and I can see myself using it easily with little experience. This also makes it near human-readable and allows for only a quick skim to gauge what a programme does, increasing the chance I'd actually use it. Overall, a simple but effective language with good reading comprehension and due to its near-pythonic nature makes it simple for someone versed in other languages to switch to."

Subsequently P3 had noted that Laturon could be a "great starter language" due to the inheritance of other programming languages, as stated in:

> "I'd considered a great starter language which inherits from other languages like Python and PHP."

Although: P1, P2, and P3; each had different responses, an appreciation of the syntactic structure was also noted, as the syntax had been based around user responses, which when a user was provided with a language supplementing their amalgamated responses (Laturon), comparisons based upon their preferred languages were expected to be declared by the participants. The Laturon syntax was also distinguished as having "clarity" (as stated by P1), as it is "near human-readable" (as stated by P2) and "intuitive" which was noted by P4's response of:

> "I like it. Its similarity to, and influence from other programming languages made it feel intuitive to develop in. I also like the lack of requirement for indentation and whitespace, as that can be frustrating in Python."

Likewise, compared to P4's response, P5 had also noted an "intuitive" learning curve. The response from P5 is as follows:

> "I appreciate the minimalist design, whilst it still provides a sufficient number of built-in functions; it is fairly intuitive for someone with programming experience. Although more data types could be bundled into it."

Additionally P5, noted that the level of built-in functions was sufficient enough for a person with experience in programming, however had noted that "more data types could be bundles into it", which could be due to an expectation of an object-oriented programming (OOP) paradigm compared to the functional approach of the Laturon language, however with the dynamic variable type of the Laturon language, a potential addition to the Language syntax could be object creation, however that would result in a notable development of a new subsystem to account for that.

From previous studies it can be reasoned that developers prefer to use source code as a form of documentation (de Souza, Anquetil and de Oliveira Sep 21, 2005, pp.68-75), which supports the locality of the Laturon programming language, as the simple syntax can easily be understood by developers, as it loosely mimics other programming languages, to an extent that programming paradigms can be transferred from other learnt programming languages.

Based on the results from the Design and Implementation section of this report, the Laturon programming language interpreter could operate with the syntax correctly and would output the expected outputs. Through conducting tests that utilise iteration or nested function calls, an in-depth analysis of the execution stacks' operations could be monitored, and with the use of GDB (on native builds), memory could be viewed, and the flow of execution could be viewed to verify the expected scope of logical operations were being operated on, which after the generations of different builds, could be verified as successfully operating towards the expected output.

7. Conclusion

The design and development of a programming language, based upon people's own understanding of cognitive understanding of programming semantics and paradigms was successfully achieved. The syntactic structure mimics other programming languages conventions yet amalgamates the overall design choices made via the user survey responses. This proved to be a usable language due to it being a collection of subsets from other programming languages, resulting in a simple to understand syntax.

The Laturon programming language being based upon a user designed approach resulted in a high-level interpreted, garbage collected, dynamic programming language, which was easy to understand for users that had development experience. Having indirectly drawn inspiration from other languages (from the results of the user surveys), the language follows a simple development flow, allowing users to define functional operations, within this paradigm, a broad logical and scoped set of complex programs can be developed. The Laturon programming language successfully allowed for a WASM compilation target, which further allowed for the portability of the interpreter, so it could be used from any major web browser; as a compilation target WASM proves itself to be an extremely efficient bytecode format that is extremely portable, and with the Emscripten toolchain, compilation was easily achievable on an existing C codebase. Today, computers are smaller, cheaper, faster and better than ever before; so, capitalising on the portable aspect of WASM allows for development within the Laturon programming language from almost all device paradigms.

8. Recommendations

Although the Laturon interpreter is working, there can be further developments to it…

The variable manager's array could be implemented to be stored within a heap memory region, instead of stack memory. This could be done to allow for a larger number of variables, as well as allowing for dynamic reallocation of the relative array, which would cater for an increase or decrease in the total amount of active variables and would modulate the total amount of memory used by the interpreter accordingly.

Additional operators could also be implemented, such as an operator for: incrementing, or decrementing; a value rather than assigning the value of the variable to itself with the addition or subtraction of a value. In relation to operators, a working list implementation could be integrated into the interpreter to allow support list-based logic and operations to occur.

A more complex endeavour could be the regeneration of interpreter to be a compiler, as due to the structure of the parsed operations being constructed to be operable on the execution stack, the tokens could also be used within a similar fashion on natively compiled binaries to run on a native call stack — for a much faster execution. A linked library to handle the dynamic type of variables could be compiled and dynamically linked, to allow for native execution.

9. References

ALIBABA TECH, 2019. GCC vs. Clang/LLVM: An In-Depth Comparison of C/C++ Compilers. In: Medium. Aug 29, Available from: https://alibabatech.medium.com/gcc-vs-clang-llvm-an-in-depth-comparison-of-c-c-compilers-899ede2be378

BERRY, G. and G. GONTHIER, 1992. The Esterel synchronous programming language: design, semantics, implementation. Science of Computer Programming, 19(2), pp.87-152, https://doi.org/10.1016/0167-6423(92)90005-V 10.1016/0167-6423(92)90005-V

BONAR, J. and E. SOLOWAY, 1983. Uncovering principles of novice programming. ACM; pp.10-13, https://doi.org/10.1145/567067.567069 10.1145/567067.567069

BULLINGER, H. and J. ZIEGLER, 1999. Ergonomics and User Interfaces. Mahwah, NJ: Lawrence Erlbaum; pp.609

D. M. RITCHIE et al., 1978. UNIX time-sharing system: The C programming language. The Bell System Technical Journal, 57(6), pp.1991-2019, https://doi.org/10.1002/j.1538-7305.1978.tb02140.x 10.1002/j.1538-7305.1978.tb02140.x

DE SOUZA, S.C., N. ANQUETIL and K. DE OLIVEIRA, Sep 21, 2005. A study of the documentation essential to software maintenance. ACM; pp.68-75, https://doi.org/10.1145/1085313.1085331 10.1145/1085313.1085331

ELIO, R. et al., 2011. About Computing Science Research Methodology. About computing science research methodology

ENGEBRETSON, A. and S. WIEDENBECK, 2002. Novice comprehension of programs using task-specific and non-task-specific constructs. Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments. IEEE; pp.11-18

FAKHOURY, S., et al., 2018. The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load. Association for Computing Machinery, pp.286–296, Available from: https://doi.org/10.1145/3196321.3196347 10.1145/3196321.3196347

FEDORENKO, E. et al., 2019. The Language of Programming: A Cognitive Perspective. Trends in cognitive sciences, 23(7), pp.525-528, https://doi.org/10.1016/j.tics.2019.04.010 10.1016/j.tics.2019.04.010

GHORY, I., 2007. Fizz Buzz Test. In: Imran On Tech. Jan 24 [viewed Apr 28, 2022]. Available from: https://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/

HUGHES, H.D., 1985. A programming language engineered for beginners. Computer Languages, 10(1), pp.23-36 https://doi.org/10.1016/0096-0551(85)90008-6 10.1016/0096-0551(85)90008-6

IEEE, 2008. IEEE Standard for Floating-Point Arithmetic. IEEE, pp.1-70, https://doi.org/10.1109/IEEESTD.2008.4610935 10.1109/IEEESTD.2008.4610935

JOYNER, I., 2022. As a coding language, C++ appeals to the ego, not the intellect [viewed Feb 02, 2022]. Available from: https://www.efinancialcareers.co.uk/news/2022/02/c-coding-language-problems

KURTZ, T.E., 1978. BASIC. History of Programming Languages. New York, NY, USA: Association for Computing Machinery; pp.515–537

LEWIS, C.I., 1956. Mind and the world-order: Outline of a theory of knowledge. Courier Corporation

ROSSBERG, A., 2019. WebAssembly Core Specification. Available from: https://www.w3.org/TR/wasm-core-1/

10. Bibliography

AHO, A.V., 2012. Computation and Computational Thinking. The Computer Journal, 55(7), 832-835, https://doi.org/10.1093/comjnl/bxs074 10.1093/comjnl/bxs074

BLACKWELL, A.F., 2017. 6,000 Years of Programming Language Design: A Meditation on Eco's Perfect Language. In: S. DINIZ JUNQUEIRA BARBOSA and K. BREITMAN, eds. Conversations Around Semiotic Engineering. Cham: Springer International Publishing, pp.31-39, https://doi.org/10.1007/978-3-319-56291-9_5 10.1007/978-3-319-56291-9_5

BRADY, E., 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming, 23(5), pp.552-593, https://doi.org/10.1017/S095679681300018X 10.1017/S095679681300018X

BROWN, C.W., 2009. Section 2.1 of Programming Language Pragmatics. [viewed Feb 02, 2022]. Available from: https://www.usna.edu/Users/cs/wcbrown/courses/F19SI413/lec/l07/lec.html

EMSCRIPTEN, 2022. Emscripten [viewed Feb 02, 2022]. Available from: https://emscripten.org/

GOOD, J. and K. HOWLAND, 2017. Programming language, natural language? Supporting the diverse computational activities of novice programmers. Journal of Visual Languages & Computing, 39, pp.78-92, https://doi.org/10.1016/j.jvlc.2016.10.008 10.1016/j.jvlc.2016.10.008

HOARE, C.A., 1973. Hints on Programming Language Design. Available from: https://apps.dtic.mil/sti/citations/AD0773391

KWON, D., I. YOON and W. LEE, 2011. Design of Programming Learning Process using Hybrid Programming Environment for Computing Education. KSII Transactions on Internet and Information Systems, 5(10), pp.1799-1813, https://doi.org/10.3837/tiis.2011.10.007 10.3837/tiis.2011.10.007

L. MCIVER and D. CONWAY, 1996. Seven deadly sins of introductory programming language design. Proceedings 1996 International Conference Software Engineering: Education and Practice; pp.309-316, https://doi.org/10.1109/SEEP.1996.534015 10.1109/SEEP.1996.534015

MACLENNAN, B.J., 1997. “Who Cares about Elegance?” The Role of Aesthetics in Programming Language Design. SIGPLAN Not., 32(3), pp.33–37, https://doi.org/10.1145/251634.251637 10.1145/251634.251637

MAYER, R.E., 1997. Chapter 33 - From Novice to expert. In: M. G. HELANDER, T. K. LANDAUER and P. V. PRABHU, eds. Handbook of Human-Computer Interaction (Second Edition). Amsterdam: North-Holland, pp.781-795, https://doi.org/10.1016/B978-044481862-1.50099-6 10.1016/B978-044481862-1.50099-6

MCKEEMAN, W.M., 1966. AN APPROACH TO COMPUTER LANGUAGE DESIGN. Available from: https://apps.dtic.mil/sti/citations/AD0639166

MCKEEMAN, W.M., 1974. Programming Language Design. In: F. L. BAUER, et al., ed. Compiler Construction: An Advanced Course. Berlin, Heidelberg: Springer Berlin Heidelberg, pp.514-524, https://doi.org/10.1007/978-3-662-21549-4_19 10.1007/978-3-662-21549-4_19

METCALF, M., J.K. REID and M. COHEN, 2004. Fortran 95/2003 explained. Repr. (with corr.) ed. Oxford [u.a.]: Oxford Univ. Press

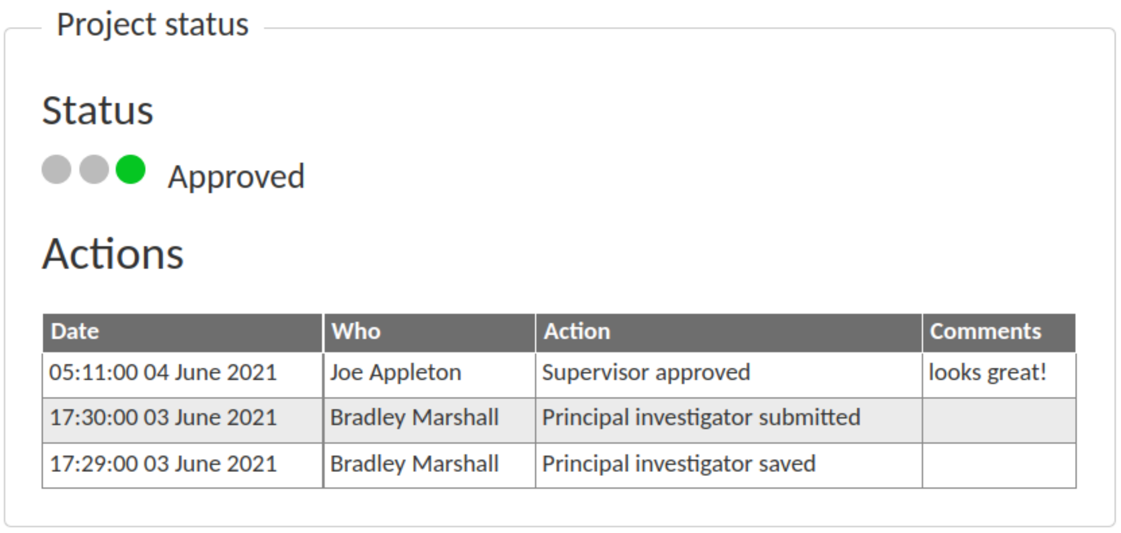
MOSSES, P.D., 2001. The Varieties of Programming Language Semantics And Their Uses. In: D. BJØRNER, M. BROY and A. V. ZAMULIN, eds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp.165-190, https://doi.org/10.1007/3-540-45575-2_18 10.1007/3-540-45575-2_18

PANE, J.F. and B.A. MYERS, 2004. More Natural Programming Languages and Environments. End User Development. Dordrecht: Springer Netherlands, pp.31-50, https://doi.org/10.1007/1-4020-5386-X_3 10.1007/1-4020-5386-X_3

WHITE, G.L. and M.P. SIVITANIDES, 2002. A theory of the relationship between cognitive requirements of computer programming languages and programmers' cognitive characteristics. Journal of Information Systems Education, 13(1), pp.59-66

11. Appendix

Appendix A: Ethical clearance for research projects



Project status

Status

Approved

Actions

Date	Who	Action	Comments
05:11:00 04 June 2021	Joe Appleton	Supervisor approved	looks great!
17:30:00 03 June 2021	Bradley Marshall	Principal investigator submitted	
17:29:00 03 June 2021	Bradley Marshall	Principal investigator saved	

Figure 13: Ethical clearance for research projects

Appendix B: Variable declaration numeric references

Table 10: Variable declaration numeric references

Numeric reference	Syntax
1	type identifier (value);
2	identifier = value
3	type identifier = value;
4	variable identifier = value;

Appendix C: Addition function numeric references

Table 11: Addition function numeric references

Numeric reference	Syntax
1	type function_identifier <type parameter_identifier, type parameter_identifier> variable_identifier = parameter_identifier + parameter_identifier; return variable_identifier;
2	function_identifier parameter_identifier + parameter_identifier = variable_identifier
3	type function_identifier(parameter_identifier: type, parameter_identifier: type): return_type { return parameter_identifier + parameter_identifier; }
4	type function_identifier(type parameter_identifier, type parameter_identifier) { return parameter_identifier + parameter_identifier; }
5	type return_type function_identifier(type parameter_identifier, type parameter_identifier) { return parameter_identifier + parameter_identifier; }
6	+
7	parameter_identifier + parameter_identifier
8	type function_identifier(parameter_identifier, parameter_identifier) { return parameter_identifier + parameter_identifier; }
9	return_type function_identifier(parameter_identifier, parameter_identifier) { return parameter_identifier + parameter_identifier; }

Appendix D: Zero function numeric references

Table 12: Zero function numeric references

Numeric reference	Syntax
1	type function_identifier <type parameter_identifier> return parameter_identifier == truthy
2	=truthy
3	function_identifier if parameter_identifier == truthy, truthy_true, truthy_false
4	type function_identifier(parameter_identifier: type): return_type { return parameter_identifier == truthy; }
5	type return_type function_identifier(type parameter_identifier) { return parameter_identifier == truthy; }
6	type function_identifier(parameter_identifier): return parameter_identifier == truthy
7	type function_identifier(parameter_identifier) { return parameter_identifier == truthy }
8	return_type function_identifier(type parameter_identifier) { return parameter_identifier == truthy; }
9	(function_identifier=truthy)

Appendix E: Greater than function numeric references

Table 13: Greater than function numeric references

Numeric reference	Syntax
1	type function_identifier <type parameter_identifier> return parameter_identifier > value
2	>value
3	function_identifier if parameter_identifier > value, truthy_true, truthy_false

4	type return_type function_identifier(type parameter_identifier) {
	    return parameter_identifier > value;
	}

5	type function_identifier(parameter_identifier: type): return_type {
	    return parameter_identifier == value;
	}

6	type function_identifier(parameter_identifier):
	    return parameter_identifier > truthy

7	parameter_identifier>value

8	return_type function_identifier(type parameter_identifier) {
	    return parameter_identifier > value;
	}

9	type function_identifier(parameter_identifier: type) {
	    return parameter_identifier == value;
	}

Appendix F: Output numeric references

Table 14: Output numeric references

Numeric reference	Syntax
1	function_call_identifier value;
2	function_call_identifier: value
3	function_call_identifier(value)
4	function_call_identifier = value
5	(function_call_identifier value)

Appendix G: Language style preferences

Table 15: Language style preferences

Worth normalised	Preferred selected language	Preferred selected scope style	Preferred function identifier
65.42%	Python	Braces	func
17.25%	Python	Tabs	Did not know
2.17%	Python	Tabs	Did not know
6.14%	Python	Tabs	Did not know
46.60%	Python	Tabs	Did not know
96.18%	C	Braces	function return_type
96.32%	JavaScript	Braces	function
100.00%	JavaScript	Braces	function return_type
75.66%	Python	Tabs	def
65.64%	C	Braces	return_type
2.97%	Python	Tabs	Did not know
17.47%	Python	Tabs	function
91.25%	Python	Braces	function
88.46%	C	Braces	return_type
2.10%	Python	Tabs	function
81.32%	JavaScript	Braces	Did not know
18.28%	Python	Braces	Did not know
Selected	Python	Braces	function

Appendix H: Mathematical operations symbolic reference

Table 16: Mathematical operations symbolic reference

Worth normalised	Addition/ subtraction/ multiplication/ division	Equals	Not	And	Or
65.42%	Notations	Functional	Functional	Functional	Functional
17.25%	Notations	Notations	"not"	"and"	"or"
2.17%	Functional	Functional	Functional	Functional	Functional
6.14%	Notations	Notations	Did not know	Did not know	Did not know
46.60%	Notations	Notations	Notations	"and"	"or"
96.18%	Notations	Notations	Notations	Notations	Notations
96.32%	Notations	Notations	Notations	Notations	Notations
100.00%	Notations	Notations	Notations	Notations	Notations
75.66%	Notations	Notations	Notations	"and"	"or"
65.64%	Functional	Functional	Functional	Functional	Functional
2.97%	Notations	Notations	"not"	Notations	Did not know
17.47%	Notations	Notations	Notations	Notations	Notations
91.25%	Notations	Notations	Notations	Notations	Notations
88.46%	Notations	Notations	Notations	Notations	Notations
2.10%	Notations	Notations	Notations	Notations	Notations
81.32%	Functional	Notations	Notations	Notations	Functional
18.28%	Notations	Notations	Notations	Notations	Notations
Selected	Notations	Notations	Notations	Notations	Notations

Appendix I: GitHub repository

GitHub repository: https://github.com/bradley499/laturon

Hosted IDE: https://bradley499.github.io/laturon