Design and Implementation of Wyltl: An Imperative, Embeddable and Portable Programming Language

Dineth Mallawarachchi
Dept. of Computer Science University of Bedfordshire,
Luton, LU1 3JU, UK

Yasas Jayaweera
Dean
SLIIT City University
Colombo 3, Colombo, SL

ABSTRACT
Wyltl is a programming language with a strong focus on simplicity, portability and functionality. It supports core imperative programming features such as variables, conditionals, loops, functions, and closures. Additionally, Wyltl provides a rich standard library with support for mathematical, date / time, JavaScript Interoperability, i /o, type conversion and more! Its implementation allows Wyltl code to be executed on a variety of platforms such as Windows, Linux, Web (via Web Assembly), and allows Wyltl to be used as an embedded programming language for applications written using the Go programming language. Wyltl offers two reference implementations, a tree walking interpreter [3], and a stack based virtual machine. The primary distinction between the two is execution speed. Executing compiled Wyltl code through the virtual machine is 1.5x - 4x faster depending on the Wyltl code that is executed. However, developers who wish to extend Wyltl with new language features will find the Interpreter easier to modify. Wyltl includes a compilation format named 'wyltlc' which allows developers to compile their existing code to an intermediate format which can be executed with the Wyltl stack based virtual machine. This paper provides a breakdown of the design of the Wyltl language and the implementation of its interpreter, compiler and virtual machine.

General Terms
Programming Languages, Interpreter, Virtual Machines, Compilers, Imperative Programming

Keywords
Embeddable Programming Language, Portable Programming Language, Stack based Virtual Machine, Tree Walking Interpreter, Web Assembly, Go.

1. INTRODUCTION
Simplicity and portability in programming languages should not be ensured at the expense of functionality. Striking a balance between the three is a classical problem in programming language design. While there are several programming languages that claim to maintain this balance, none has managed to properly stay within the sub consciousness of the developer community. Taking inspiration from venerable languages such as Lua [4] Wyltl was created with the belief that by ensuring that the three characteristics of programming design are given an equal priority in the design and implementation of the language, it will in turn ensure that its usefulness to developers. That is, the goal of Wyltl is to provide all developers the ability to create applications and programs with ease, as a standalone language or embedded language in complex Go projects.

1.1. Motivation
In implementing a project in Go, a mandate was given that the end user be given the ability to program using a simple yet fully functional programming language. Additionally, it must not harm the project's deployment. Within the Go ecosystem and beyond, no language implementation could meet all the requirements. It was this that led to the conceptual idea of Wyltl as a programming language.

1.2. Key Features of Wyltl Implementations
The key features of the Wyltl Language and its implementations are based on ensuring that simplicity, portability, functionality and embed-ability are all considered in the design and implementation of the language

The Wyltl language is simple and familiar. It is easy for a new programmer to learn, and experienced programmers can start using the language within the timespan of an afternoon. Its simple syntax was also designed with the goal of ensuring that the language is readable, akin to traditional pseudocode. Wyltl's functionality is ensured with the implementation of all common imperative programming concepts. Pseudocode or code written in another language can easily be re-written in Wyltl. As such, Wyltl can also be used as an introductory programming language.

Wyltl offers an interpreter and compiler. The implementation of the Wyltl compiler is unique as it has been implemented with easy modification in mind. These implementations are portable and can be run on all major desktop operating systems and Web Browsers. This ensures that users can execute their Wyltl applications on any modern platform. For experienced developers, the Wyltl implementations are easily embeddable within a Go application. Additionally, it is also possible to execute JavaScript code through Wyltl in the web version of the Wyltl implementations.

1.3. Development Approach
Some overlap was present in the development of the Wyltl Interpreter and Compiler, as two core components required for each (the parser and the tokenizer) were shared. This significantly assisted in maintaining functional parity between the two implementations, as using the same parser ensures that both implementations handle Wyltl parsing properly, proper logic is ensured by using unit tests and running complex Wyltl code in both implementations and comparing the results. When combined with user acceptance tests, it was possible to minimize undefined behavior to a minimum. No external dependencies were used in implementing the project, this greatly assisted in ensuring portability, and ensuring Wyltl could be used as a scripting language [6].

2. DESIGN OF THE WYLTL LANGUAGE

Out of simplicity, portability, embed-ability and functionality. The core language design of the Wyltl language is responsible for ensuring the characteristics of simplicity and portability. A literature review was performed to ascertain what could be done to ensure how could be designed. It was also important to consider practicality of implementing language features as well After investigation, a bottom – up approach was followed, in which the fundamentals would be designed first.

2.1. Simmilar Programming Languages

It is generally accepted that programming languages inherit features and characteristics from 'parent' programming languages. It is not excessive to say that the lineages of popular programming languages of today can be traced back to different programming languages from decades ago – such as Java and its grand-parent-language simula-67. In the case of Wyltl there are several programming languages which hold characteristics that Wyltl draws inspiration from.

2.1.1 Lua – The programming language for application extension

Lua [5] is a programming language with a rich history and a passionate user-base. It has enjoyed success in many endeavors, but it's main use has come as an 'embeddable language', meaning that Lua is often embedded into applications and systems, and used to implement features and functions. Two popular examples are Roblox (the popular children's video game platform) and MPV (a media player). Though used in wildly differing applications the use of Lua is always done in a similar manner – with it being used to implement additional features. The pain points many users express with Lua are its unique syntax. However, it is undeniable that Lua's syntax is simple and effective. Wyltl draws in heavy inspiration from Lua's overall philosophy. Lua's main limitation comes from its most important strength, as being written in C guarantees high portability and room for optimization, attempting to interop with languages such as Rust and Go severely affect portability. However, it should be noted that this not a problem with Lua, Rust or Go, but with the fragmentation of C compilers.

2.1.2 Wren – A small concurrent scripting language

Wren [4] is a language created by Bob Nystrom, the author of 'Crafting Interpreters'. It is similar to Lua in many ways, they are both written in C, are primarily created as scripting languages which can be embedded within existing projects and implementations. Additionally, Wren's syntax is generic, but this plays into its favor as it makes the language understandable and easy to learn. Wren itself has no qualms about stepping into complexity with support for concurrency and objected oriented programming paradigm support. Its weaknesses are very similar to those used of Lua. 'C' as a programming language is only used in certain sectors and areas of development, with most casual users preferring to use languages such as JavaScript, Python, Go, Rust, etc. It is apparent that Lua has already taken the fundamental role of the 'embeddable language for C' niche. That combined with the portability issues that arise when trying to interop with other language harm the possible of Wren's adoption. Regardless, Wren's philosophy of clear and simple language design, is worthy of admiration.

2.2. What makes the Wyltl Programming Language Unique?

Wyltl has three main focuses – that is simplicity, portability and functionality. The main research problem that Wyltl attempts to tackle is the fine balance between the three aspects. This section is an overview of how Wyltl uses these aspects to improve the experience of its users, and how it attempts to evolve the design and implementation decisions of similar languages.

Regarding simplicity, Wyltl draws in inspiration from Lua, Wren, Monkey and Lox. The fundamental goal of Wyltl is to provide a simple syntax – that is easily readable and writeable. While some liberties have been taken in regards to the design of the language, it is still similar in structure to the Programming Languages experienced developers will likely be familiar with. Keywords are used instead of symbols to increase the readability of the language. This design choice was received positively and is a part of why Wyltl stands out.

In terms of Portability - a key design decision was to base the language on the Go programming language instead of the C programming language. This allowed for increased portability and support for a wide range of platforms. It is important to note that Go by itself does not guarantee portability. The dependency free design used in Wyltl ensured that the full cross compiling functions of the Go language could be used. Neither Lua or Wren feature first class support for Web execution with Web Assembly, but it is well supported in Wyltl. This mature support for multiple platforms, despite being a young language, makes Wyltl unique among programming languages. Especially among the new generation of programming languages (languages that saw their first release in the last decade).

In terms of functionality Wyltl aims to be a language that can meet the needs of many users. While Wyltl is capable of being used as a language for extending applications, as a scripting language, or even a language for developing web applications, it is defined as a general purpose programming language. The difference in the Wyltl implementations is how its functionality is implemented. Wyltl strictly implements most of its features as parts of the standard library. This standard library itself is implemented in a similar manner to a plugin system, where users can add and remove functions as required.

A key feature of Wyltl is its interoperability. Wyltl can interop with JavaScript quite easily and effectively using its Web Assembly release. Additionally, Wyltl is not isolated from its parent language as Wyltl – Go interoperability can be achieved with little effort. While this feature is not unique to Wyltl (in fact Lua is capable of interoperating with C as required), this kind of robust first party support for interoperability is rare and makes Wyltl unique. It also heavily increases the value proposition of Wyltl for experienced developers.

Wyltl's approach with a compilation format (.wyltlc) is not common, but is useful to users who might desire performance or obfuscation – as it offers an easy way to share their code while stopping other from easily viewing its inner workings. It should also be mentioned that dual implementation of Wyltl which allows users to have the choice between the Wyltl interpreter and compiler depending on their requirements or interest in modifying the language is an uncommon feature.

2.3. Programming Fundamentals in Wyltl

The choice to design Wyltl as an imperative programming language was influenced by its popularity among new and

experienced developers. In practical use it was expected that developers would use Wyltl to write straight forward logic focusing on application state, as such an objected oriented approach was determined to be illogical. It is possible that Go's imperative nature played a role in this decision as well.

The first of the three basic constructs is variable creation and variable assignment.

```
suppose x is 1.
suppose y is 2.5.
suppose z is "Hello World".
```

Fig 1: Variable Definition and Assignment in Wyltl

As shown by above figure, Wyltl implements variable definition using the 'suppose' keyword. It is also used to implement variable re-assignment as well. Additionally, Wyltl uses dynamic typing, as such defining data types for variables is not required.

```
suppose x is 3 plus 1.
suppose y is 2 minus 1.
suppose z is 3 times 2.
```

Fig 2: Mathematical Operators in Wyltl

As shown by the above figure, Wyltl implements all standard mathematical operators in addition to what is shown above (over for division and modulo for modulus). Word based keywords are used, however the user has the freedom to use the symbol notation if required. This default notation was chosen to improve the readability and simplicity of the language

```
suppose x is 1 below 2.
suppose x is 3 above 2.
suppose x is 3 equals 2 or 2 above 0.
```

Fig 3: Logical Operators in Wyltl

Wyltl implements all common logical operators (and, nequals and not are implemented in addition to the above). In Wyltl, the use of a logical operator will always result in a Boolean value.

```
if (3 above 2) {
  return 1.
} else {
  return 2.
}
```

Fig 4: Conditionals in Wyltl

The second basic construct – conditionals are implemented by Wyltl as well. Wyltl, also offers a standalone if statement as well. When designing the conditions in Wyltl, it was decided to not force indentation rules regarding formatting code, which was strongly requested by potential users.

```
for(suppose y is 3 : y below 12 : suppose y is y plus 1) {
        print("Value of y is " plus y).
}
```

Fig 5: For loop in Wyltl

As shown by the above figure Wyltl's implementation of the for loop is slightly different, with the ':' symbol used to denote separation. Standard while loops were designed to for Wyltl as well. These constitute Wyltl's implementation of the third basic programming language construct.

2.4. Advanced Programming in Wyltl

While not considered as fundamentals there are several features that are considered important in programming languages. These can be loosely grouped together as 'advanced programming constructs'

```
suppose myArray is [1, 2, 4, minus 3, 0].
suppose sum is myArray[2] plus myArray[3].
```

Fig 6: Array and Index Operator Implementation in Wyltl

The basic data structure employed by Wyltl is arrays, which in turn can be used within the language to create virtual data structures such as graphs, stacks and queues. Arrays in Wyltl are dynamically typed themselves thus an array in Wyltl can hold different types of data. As shown in the above figure, the standard index operator can be used to retrieve an array element.

```
suppose concatWithSpace is compose(x, y) {
  return x plus " " plus y.
}.
concatWithSpace("John","Doe").
```

Fig 7: Function Creation and Execution in Wyltl

Functions can be defined with the use of the 'suppose' and 'compose' keywords within Wyltl. A function definition is relatively flexible in that there are no defined limits regarding what is possible within them. There is also no requirement to return values, a function within Wyltl can execute without returning anything.

```
suppose x is 3 minus 2.
switch(x) {
   case(2) {
      return "The wrong answer".
   }
   case(1) {
      return "The correct answer!"
   }
   default {
      return "The wrong answer".
   }
}
```

Fig 8: Switch Case Conditional in Wyltl

Switch – Case conditionals are provided to use for Wyltl programmers. Within Wyltl, switch case statements are flexible in that they are not limited to evaluating the value of the given reference. Instead, they can be used to evaluate expressions of even perform type checking as required by the developer.

A common concern in dynamically typed programming languages is automatic type conversion. A conscious decision was taken to limit Wyltl's automatic type conversions to converting integers, floats and bool values to string in the case of string concatenation. An additional fact to note is that the 'null' data type was removed upon user request during acceptance as a measure to ensure null safety within programs written in Wyltl.

3. IMPLEMENTATION OF THE WYLTL INTERPRETER

The reference Wyltl implementations are responsible for implementing the portability and functionality of the Wyltl language. These implementations are composed of several packages.

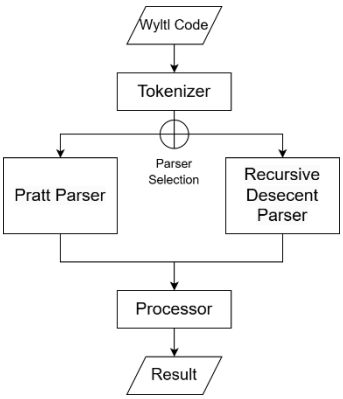

Fig 9: Structure of the Wylt Interpreter

In the case of the Wyltl Interpreter these are the tokenizer, Pratt parser [1] (or recursive descent parser), and the Processor. These packages must coordinate together to prepare and execute any given Wyltl code. In the distributed releases of the Wyltl Interpreter and Compiler this coordination is performed by the REPL package, or the Main package (in the web releases of Wyltl). Care was taken to ensure simplicity and ease of modification, and Wyltl users are encouraged to add new features to the implementations if they desire to do so.

3.1 Tokenization of Wyltl Code

Tokenization in Wyltl is straightforward. The tokenizer parses the given Wyltl code character by character. A point of interest is that the Tokenizer supports Unicode characters, so there should be little friction in using any language with Wyltl. Additionally, comments are handled directly in the Tokenizer, instead of being sent to the parser, which is a minor optimization to increase processing speed.

```
SUPPOSE : suppose
IDENTIFIER : x
ASSIGN : is
INTEGER : 1
DOT : .
```

Fig 10: Tokenized Output of "suppose x is 1."

The above figure represents the output of the Tokenizer given a simple suppose statement. It is important to note that it is possible to define multiple definitions for the same token. For example, it is possible for a hypothetical user to easily modify the token definitions to consider ‘let’ as the keyword for the ‘suppose’ statement if they so desire.

3.2 Parsing of Wyltl Code

After a given Wyltl code is tokenized, the resulting list of Wyltl tokens is passed to one Wyltl parser. The parsing process results in two important results. Firstly, the given Wyltl code is checked for any syntax errors or code structure violations within the parser, as the Wyltl tokenizer forgoes any form of error checking or validation to focus on execution speed. Secondly, the parser is responsible for the creation of the abstract syntax tree which are very important for both reference implementations of the Wyltl language.

The reference implementations of Wyltl are unique in that they support three different types of Parsers. Only one Parser can be used at a time, and in fact Wyltl’s implementations by default use the Wyltl Pratt parser as the default token parser. However, a Packrat parser [2] and a recursive descent Parser are available to be used if the user desires.

Within the context of the Wyltl implementations they return the same output, however the way they function is different. A Pratt Parser is built on the work described Vaughan Pratt’s paper on parsing, it is both token centric and precedence base, where each token has its own parse rules. It is easy to modify. On the other hand, the Packrat Parser implemented in Wyltl is grammar based and uses Strict grammar rules. It also uses elements of dynamic programming as it caches partial results. In comparison to them, the recursive descent parser is quite simplistic as it uses top-down parsing in combination with simple grammar rules.

```
SUPPOSE : suppose
IDENTIFIER : x
ASSIGN : is
INTEGER : 1
PLUS : plus
INTEGER : 3
TIMES : times
INTEGER : 6
OVER : over
INTEGER : 2
```

Fig 11: Parsed Output of 'suppose x is 1 plus 3 times 6 over 2.'

As shown by the above figure, the Parsed output from the Wyltl Parser will be properly structured. Any syntax errors within the given Wyltl code will be caught during this process. Wyltl does not continue parsing the code if any error is found, as such the user will be asked to fix any errors before the processing can continue.

3.3 Processing of Wyltl Code

The final step in the Wyltl Interpreter is the processor. The processor is ‘brain’ of the Interpreter so to speak. The parsed abstract syntax tree from a Wyltl parser is the input taken by the Processor. The structure of the Processor itself is closely modeled after a standard tree walking interpreter and implemented using a visitor pattern, this approach was chosen due to its ease of implementation, while allowing users to easily modify and add new features to the language. However, the heavy dependence on traversing the abstract syntax tree with recursion results in relatively slow Wyltl code execution. Variables and Wyltl runtime data is stored within ‘environments’ which are closed off tables. A global table is always maintained, and for block statements closed off tables are created to ensure that variable scopes are ensured. Variable are passed by reference to ensure memory usage and increase the execution speed of Wyltl code.

A potential weakness of the Processor is its heavy dependence on runtime type checking. To counteract this weakness effort has been taken to ensure that the Processor can infer basic Wyltl data types such as Integer, Float and Bool as required, however reflection is used as a fallback if automatic type inference results in failure. Additionally, measures have been taken to ensure type safety in the processing of all Wyltl code.

As additional measures for safety bounds checking, null safety, scope protection, type safe error propagation and graceful error handling have been implemented to ensure that users can use the Wyltl Interpreter with confidence and trust.

4. IMPLEMENTATION OF THE WYLTL COMPILER

The second reference Wyltl implementation is the Wyltl Compiler. In reference to the Wyltl Interpreter, there are two main differences which a regular user would be able to notice when comparing the two reference Wyltl Implementations.

Firstly, the Wyltl Compiler is significantly faster when compared with the Wyltl Interpreter. This is not true for all tasks, but the in most operations, the difference is very much noticeable. Secondly, the Wyltl compiler can compile any Wyltl files (.wyltl) to the Wyltl compiled format (.wyltlc). In terms of the implementation, the packages which comprise the Compiler are unique to itself.

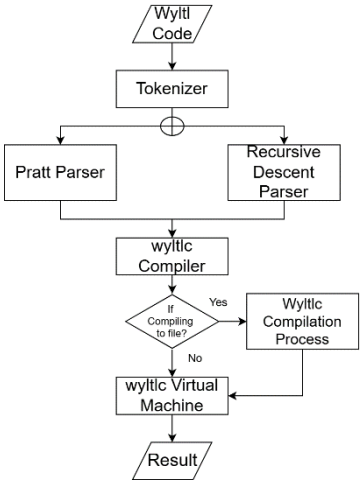

Fig 12: Structure of the Wylt Compiler

As shown by the above figure, the Wyltl compiler uses the same tokenization and parsing process used by the Wyltl Interpreter. However, after the chosen parser creates the relevant Wyltl abstract syntax tree, instead of passing to the processor package to be directly executed, instead, the tree is passed to the Wyltlc Compiler, which compiles the code and executes it within the Wyltlc Virtual Machine.

4.1 Compilation of Wyltl Syntax Trees

The primary task of the Wyltlc compiler is to compile or convert the given Wyltl code into the Wyltlc format.

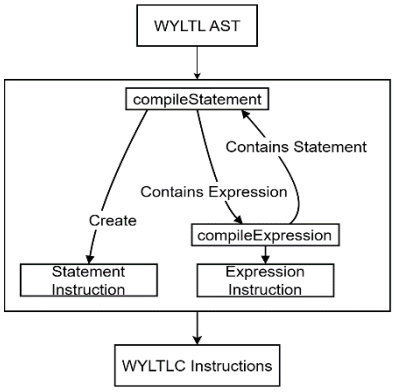

Fig 13: Compilation of Wyltl Code to Wyltlc

As shown by the above diagram, it is more accurate to call the compiler a 'tree walking compiler', as the compiler recursively travels the given abstract syntax tree to generate Wyltlc instructions. An instruction is composed of a node which is extracted from the input abstract syntax tree. This effectively simplifies each operation to a single instruction.

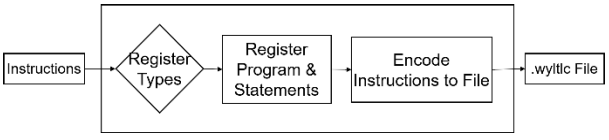

Fig 14: Creation of standalone Wyltlc file

4.2 Implementation of the Wyltlc Format

The desktop versions of Wyltl have the ability to compile a given Wyltl code file and save it as a Wyltlc file. This file contains binary data which can be executed using the Wyltlc virtual machine. The process of compiling Wyltl code to an output file is done as a possible intermediate in execution if enabled by the user.

The above figure showcases the creation of an output Wyltlc file from the Wyltlc code created during the compilation process. This process makes heavy of the 'Gob' packaged within Go to serialize the compiled instructions to a storage format. While Gob is slower when compared with implementations such as protocol buffers, it offers a dependency free and cross platform implementation, which is very important in ensuring simplicity and portability. The process is reversed when reading the output wyltlc using the execute function in the Wyltl reference implementations. With Gob being used to de-serialize the file and the encoded statements and expressions being un-marshalled and sent to the next stage of the execution process.

4.3 Implementation of the Wyltlc Virtual Machine

The compiled Wyltlc code which is comprised out of instructions is passed to the Wyltlc virtual machine for execution. In essence it performs the same function as the processor used within the Wyltl Interpreter, however the main difference is that the Wyltlc virtual machine processes the compiled Wyltlc instructions instead of the Wyltl abstract syntax tree. This a middle point between a traditional interpreter and a bytecode compiled that allows Wyltl to have the advantages of both implementations, while reducing the overall complexity significantly.

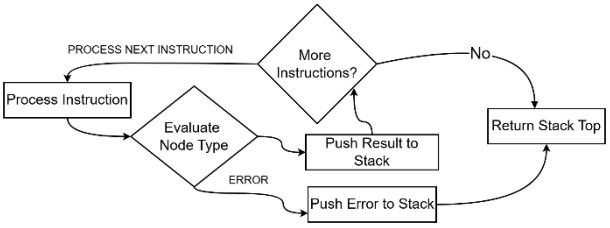

Fig 15: Instruction Processing Process within the Wyltl Virtual Machine

The above figure elaborates on the fetch – decode – execute cycle run by the virtual machine as it loops through the given Wyltlc instructions. In this instance the instructions act as a virtual 'instruction set'. The virtual machine itself makes heavy use of stacks in processing data, this allows it to avoid a significant amount of recursion and achieve a baseline speed that is 3 – 4 times faster than the interpreter when executing its compiled Wyltlc code. Unlike traditional virtual machines, several stacks will likely exist at time, since different scopes maintain their own stacks. However, a central global stack is used for managing global operations. To manipulate and control the stack several helper methods such as push, pop and peek are used. However, to manage the overall resource usage of the stack itself Wyltl depends on Go's garbage collection mechanisms.

5. IMPLEMENTATION OF THE WYLTL STANDARD LIBRARY

The Wyltl reference implementations, that is the Wyltl tree walking interpreter, and the instruction-based compiler are implementations of the core Wyltl Language. The 'core' Wyltl

language implements the basic and advanced programming constructs touched upon in the 'design of the Wyltl language' topic. 'Functionality' is a core design goal of Wyltl, and it is not implemented in the core design of the language, but instead using a sibling package known as 'standardLibraryFunctions'. This results in a significant boost to the maintainability and development velocity of the language. It should be noted that many of Wyltl's standard library functions are 'return' functions which take a given value and return another value. Therefore, many standard library functions are used with the 'suppose' statement.

5.1 Implementation of I/O Functionality within the Wyltl Standard Library

A basic feature of any programming language is the ability to accept inputs and display outputs to the user. Wyltl implements these using its standard library as follows.

Table 1. I/O functions in Wyltl

Function	Purpose
print()	Print an item using STDOUT
inputText()	Take string input
inputNumber()	Take integer input
inputFloat()	Take floating point input

5.2 Implementation of String Functions within the Wyltl Standard Library

While string manipulation is admittedly not a strong focus of the Wyltl language, the standard library does offer some convenience functions for dealing with strings.

Table 2. String focused functions in Wyltl

Function	Purpose
length()	Returns the length of an item*
upperCaseString()	Returns an uppercase string
lowerCaseString()	Returns a lowercase string
reverseString()	Returns of reversed string

*This function can be used on both Arrays and Strings

5.3 Implementation of Array Functions within the Wyltl Standard Library

Arrays are the principal data structure in the Wyltl language, and they are core of many a complex program written in Wyltl. As such Wyltl provides a complete set of functions focused on demystifying usage and manipulation of arrays.

Table 3. Array functions in Wyltl

Function	Purpose
firstArrayElement()	Returns the first array element
lastArrayElement()	Returns the last array element
pushArrayElement()	Returns a copy of the array with the given element added to the end
popArrayElement()	Returns a copy of the array with the first element removed
stringToArray()	Converts the given string to an array and returns it
arrayToString()	Converts the given array to a string and returns it
changeArrayElement()	Takes an array, a index, and an element as a parameters then returns an array with the element at the given index replaced with the given element
enqueueArrayElement()	Returns an array with the given element added to its end
dequeueArrayElement()	Returns an array with its first element removed
cutLineArrayElement()	Adds the given element to the given index of the given array, and returns the new Array

5.4 Implementation of Math Functions within the Wyltl Standard Library

Alongside arrays, Wyltl places a strong emphasis on mathematical calculations. As such, the Wyltl standard library implements many common mathematical functions.

Table 4. Mathematical functions in Wyltl

Function	Purpose
exponent()	Return the exponential value of the given number
logarithm()	Return the base 10 logarithmic value of the given number
minimum()	Returns the smaller of the given two numbers
maximum()	Returns the larger of the given two numbers
sine()	Returns the sine equivalent of the given value
cosine()	Returns the cosine equivalent of the given value
tangent()	Returns the tangent equivalent of the given value
arcSine()	Returns the arc sine equivalent of the given value
arcCosine()	Returns the arc cosine equivalent of the given value
arcTangent()	Returns the arc tangent equivalent of the given value
absoluteValue()	Returns the absolute value of the given number
toPowerOf()	Returns the value of the given base to the given power i.e. (2,2) is 4
squareRoot()	Returns the square root of the given integer
floatToInteger()	Returns the integer conversion of the given float
integerToFloat()	Returns the float conversion of the given integer
random()	Returns a random float between 0 and 10
round()	Rounds the given number up or down. The positive or negative position can be given as an argument (this is the only Wyltl function with a variable number of input parameters)

5.5 Implementation of Date &Time Functions within the Wyltl Standard Library

Wyltl implements several Standard Library functions focused on allowing users to easily capture and measure a particular

date and time. The way Wyltl implements these functions are somewhat unique, with a focus on simplicity and functionality.

Table 5. Date & Time functions in Wyltl

Function	Purpose
presentYear()	Returns the current year as an integer
presentMonth()	Returns the current month as an integer between one and twelve
presentDay()	Returns the current day of the of the month as an integer between one and thirty-one
presentHour()	Returns the current hour of the day as an integer between zero and twenty-four
presentMinute()	Returns the current minute of the hour as an integer between zero and sixty
presentSecond()	Returns the current second of the minute as an integer between zero and sixty
presentMilliSecond()	Returns the current milli second of the second as an integer between zero and nine hundred and ninety nine

5.6 Implementation of Type Checking Functions within the Wyltl Standard Library

Wyltl provides a standard type checking function to ensure that type safety can be ensured within programs. Generally, type checking is intended to be verbose within Wyltl, as such users are expected to use the type checking function declaratively as required. The data types implemented in Wyltl are INTEGER, FLOAT, STRING, ARRAY, and FUNCTION.

```
typeOf(Data)
```

Fig 16: Standard Type Checking Function in Wyltl

5.7 Implementation of JavaScript Interoperability within the Wyltl Standard Library

While Wyltl attempts to maintain parity among the different releases of Wyltl there is a feature supported in the Web or Web Assembly release of Wyltl that is not supported in the standalone desktop or embedded releases of Wyltl – that is the ability to execute JavaScript code. This interoperability was started with the request of allowing HTML document object model manipulation in the Wyltl web release. However, it gradually grew in scop until full JavaScript Interoperability was achieved. This interoperability is made more robust with Wyltl' ability to share data and variables with the executed JavaScript code, and the ability of the JavaScript code to return values that map to Wyltl's native data types. It is important to note that errors and undefined data will automatically be mapped to strings. However, integers, floats, strings and arrays will map to Wyltl as expected.

In terms of limitations, it must be noted that due to how the Wyltl and JavaScript processes communicate with each other, all objects and variables within JavaScript must be created as objects of 'window' lest they become undefined.

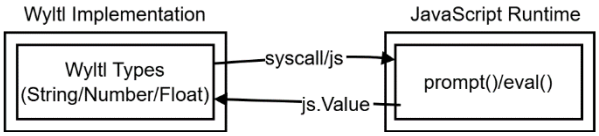

Fig 17: Transfer of Data between Wyltl and JavaScript

The above diagram provides a high-level overview of the transfer of data between the two language runtimes.

5.8 Porting Wyltl to Web Assembly

Go's support for Web Assembly made the initial exporting process easy, only becoming easier due to Wyltl's dependency free implementation philosophy. However, the initial port suffered from several limitations, it only ran in the Web Browser's developer console, and had no support for standard input and output. Thus, some glue code had to be written to connect them, which replaced the traditional console application used on desktop, with a standard input and output GUI made in JavaScript, HTML and CSS.

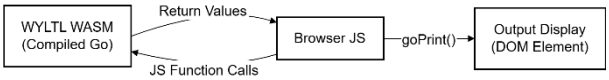

Fig 18: Glue connecting Wyltl and the Web Browser

This functionality was used to introduce interoperability between Wyltl and JavaScript.

6. EVALUATING THE WYLTL IMPLEMENTATIONS

The native benchmarking suite available in Go was used to test the efficacy of Wyltl's performance.

Table 6. Wyltl Algorithm Benchmarks

Name of Test	Compiler Speed (µs)	Interpreter Speed (µs)	Interpreter Memory Usage (KB)	Compiler Memory Usage (KB)
Iterative-Fibonacci	555.33	952.26	91.57	98.75
Bubble-Sort	97.60	374.46	3.00	9.85
Selection-Sort	121.48	370.69	3.09	10.17
Dijikstra-Search	531.23	939.29	3.09	10.17
Prims-Search	327.99	794.49	10.09	29.22
Breadth-First-Search	230.63	566.80	8.48	22.82
Depth-First-Search	184.18	501.90	7.76	20.36
Quick-Sort	332.93	648.64	14.08	26.07
Merge-Sort	629.57	963.25	29.13	41.23

The above table showcases the difference in execution speed and efficiency between the Wyltl Compiler and Interpreter when tested using Go's native benchmarking tools.

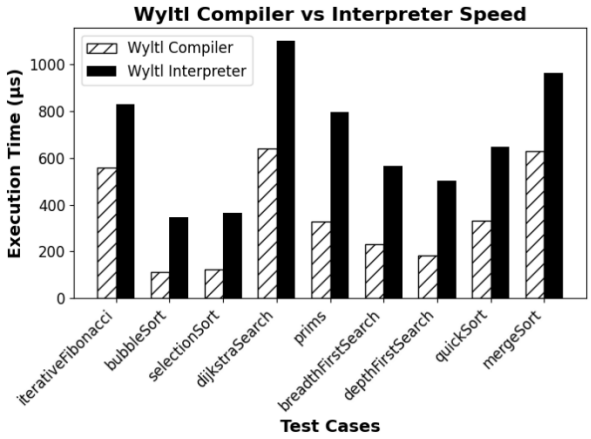

Fig 19: Differences in Wyltl Implementation Speed

The above graph showcases the difference in execution speed between the Wyltl Compiler and Interpreter. For a given period of, the compiler was able to consistently beat the interpreter in speed ranging from a rate of 1.5x to 4x. This is a showcase of different scenarios which the stack based virtual machine can showcase its improved efficiency and direct execution speed over the direct implementation of array based environments as used in the Wyltl Interpreter.

A Comparison of memory usage between the Wyltl interpreter and compiler when running the same bubble-sort operation is provided below.

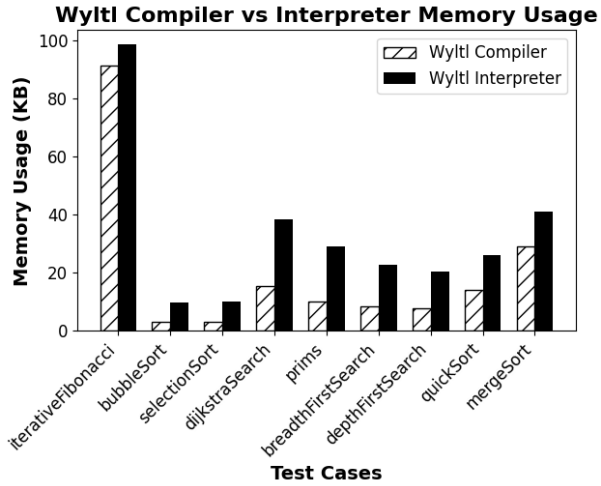

Fig 21: Differences in Wyltl Memory Usage

As shown by the below two diagrams, the Wyltl Compiler is far more efficient in the amount of memory used. This is of considerable importance, as lesser overall memory usage translates to lesser burden on the Go garbage Collector, which in turn ensures more consistent performance.

7. CONCLUSIONS AND FUTURE WORK

The central conclusion of the project is that the creation of a programming language with a potential balance of simplicity, embeddability, portability and functionality using Go is not only valid, but holds great promise and potential. The mix of low and high level functionality provided by go – along with its excellent cross compilation provides an excellent base for languages to build upon.

While we were able to achieve a reasonably high level of performance and optimization, there is some further room for improvements in regard to Wyltl. The implementation of detailed networking libraries, and the implementation of binary compilation using flat assembler or net assembler are valid but beyond the initial scope of the project. While there is always room for further optimization, and such efforts were attempted, it came at the cost of overall readability and extendibility. As optimizations resulted in Wyltl code and definitions that were difficult to modify. A particular aim to strive towards in the future is dogfooding. The process in which parts of the language are written using itself. This is implemented to an extent in Wyltl with array standard library functions. However, further work can be performed to implement many other core functions using Wyltl itself.

8. ACKNOWLEDGMENTS

The author would like to express his gratitude towards University of Bedfordshire and SLIIT City University for their support during this research. Special thanks to Dr. Yasas Jayaweera for his invaluable guidance and supervision throughout the research.

9. REFERENCES

[1] Pratt, V.R. (1973) 'Top-down operator precedence', Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '73, pp. 41–51. doi:10.1145/512927. 512931.

[2] Ford, B. (2002) 'Packrat parsing: Simple, Powerful, Lazy, Linear Time', Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, pp. 36–47. doi:10.1145/581478.581483.

[3] Nystrom, R. (2021) *Crafting interpreters*. United States Genever Benning.

[4] Nystrom, B. (no date) *Wren*. Available at: https://wren.io/ (Accessed: 22 February 2025).

[5] Ierusalimschy, R., de Figueiredo, L.H. and Filho, W.C. (1996) 'Lua—an extensible extension language', *Software: Practice and Experience*, 26(6), pp. 635–652. doi:10.1002/(sici)1097-024x(199606)26:6<635::aid-spe26>3.0.co;2-p.

[6] Ousterhout, J.K., 1998. Scripting: Higher level programming for the 21st century. Computer, 31(3).