Implementation of Wyltl: An Imperative Language with a Dual Interpreter – Compiler Architecture

Dineth Mallawarachchi
School of Computing
SLIIT City Uni
lochandineth@gmail.com

Yasas Jayaweera
School of Computing
SLIIT City Uni
yasas.j@sliit.lk

***Abstract*— When using a programming language, a common drawback is the prevalence of resource constraints. A lack of resources often results in programs executing faster on high end hardware in comparison to middling or low-end hardware. While a core tenant of programming is optimization, with which entire industries have been built upon, when implementing a programming language, the process becomes significantly more complex. Minute slowdowns in a programming language implementation could very quickly result in major slowdown when executing some code. This paper examines the process of implementing the Wyltl language while balancing the need for performance and resource efficiency provided precious insight and hints towards future optimization, and the unique challenges and opportunities for evolution. Including evaluation of varying parsers, processors and technologies. The evaluation of different parsers and processors was done throughout development with a varied range of programming language optimization techniques being followed as required.**

***Keywords*— Parser Evaluation, Optimization, Programming Language Implementation, Wyltl, Interpreter, Compiler, Web Assembly, JavaScript Interoperability, Go.**

I. Introduction

Wyltl [1] is a programming language with a focus on maintaining a balance of simplicity and functionality (programming language design techniques), and portability and embeddability (programming language implementation techniques). The central design concept of Wyltl focused on the idea that balancing all traits would provide a complete and capable implementation for experienced and new developers. An emphasis on performance was considered secondary during the latter stages of the project. Looking at Wyltl with a focus on performance provides a unique perspective not commonly found in many software development situations.

II. Literature Review

When developing Wyltl a wide range of alternative systems and approaches were considered with their respective strengths and weaknesses.

A. Pratt, Packrat and Recursive Descent Parsing

Parsing is the second stage of execution within a programming language – responsible for taking a stream of tokens given by the tokenizer and combining them into a specific structure which is later processed or compiled. Parsing in modern programming languages makes use of either Pratt Parsing [2] or Packrat Parsing [3]. Out of these approaches Packrat parsing introduced by Ford is often avoided due to its relatively high memory usage. This is a waste of resources and is not efficient when handling complex expressions as what would be found when parsing programs. Pratt's parsing techniques in comparison are quite efficient in parsing complex mathematical expressions but would themselves struggle with efficiently handling complex programs – and more importantly, error handling and recovery. The solution was to investigate classical LR (left-right) recursive descent parsing more closely. Which provided fast execution speeds and low memory usage with increased flexibility, while keeping many of the improvements brought forward by a custom parsing implementation as used in common Pratt and Packrat parsing implementations.

B. Efficiency of Compiled and Interpreted Scripting Programming Language Implementations.

The difference between Compiled and Interpreted languages is a common point of concern for many users. It becomes a key decision in implementing Wyltl as compiler and interpreters are fundamentally different systems with their own advantages and drawbacks. However, the role of a scripting programming language is flexible as described by Ousterhout in his historical analysis of programming languages [4]. He correctly predicts the importance of focusing on simplicity in a manner that does not compromise on functionality. This applies to Wyltl in the way both a compiler and interpreter are present and supported. For absolute efficiency, a compiler is

https://doi.org/10.54389/CWBL6597

undoubtedly better as programs execute faster. However, when it comes to embeddability an interpreter can offer more flexibility. The key decision was to offer a dual implementation that provided efficiency regardless of the implementation type used, while keeping the inherent performance differences between a compiler and interpreter implementation.

C. Inline Substitution

One of the most popular optimizations done by programming language implementations is inline substitution. Which substitutes expressions with simplified forms of themselves, this can take the form of replacing expressions such as '2 + 1' with '3'. The approach is outlined by Scheifler [5]. In practice, using inline substitution on complex statements can be difficult and potentially risky. Wyltl makes use of inline substitution when compiling mathematical expressions. Scheifler examines the advantages brought forward by implementing such a compilation structure – which primarily consists of improving execution time by reducing recursive function call. An approach which proves to be very useful as recursion remains a complex and heavy process which frequently bottlenecks and slows – down the execution of many programming languages, including Go and by extension Wyltl.

D. Slim Compilation

In compiling a program to a binary format, the process in which the compilation process occurs can also be considered for optimization as described by Kistler and Thomas [6]. 'Slim Compilation' in this context can contest resulted in multiple approaches – starting from removing unwanted debug data from executables to implementing compression algorithms with binary data when required. While reducing the size of compiled executables with a high improvement ratio, such compression in turn results in a need for increased processing power for decoding the compressed data. Unlike other factors, slim compilation presents a hard boundary of optimization, as it becomes impractical beyond slight space savings.

III. Uniqueness of Wyltl Implementations

Striking a balance between simplicity, portability, embeddability and functionality is considered one of the key challenges in programming language design and optimization. Opinions regarding which of the trio should be prioritized is what differentiates the syntax and semantics of many programming languages. Wyltl forgoes that by placing equal focus on all four factors. Simplicity is ensured with Wyltl's language design as shown by listing one below. All fundamental programming components are showcased with simple syntactical structures

Portability is ensured using the Go toolchain. Which by itself ensures portability as Go does not depend on any external dependencies for its cross-platform support. Although it's important to note that using Go by itself is not indicative of cross platform support. Wyltl is specifically written in a manner that it is completely platform agnostic.

Embeddability is ensured in with Wyltl's ability to be embedded within existing Go applications. This is implicit with how Wyltl's components are designed and partitioned

IV. Shared Architecture of The Wyltl Implementations

Wyltl provides a dual interpreter and compiler implementation of which both are combined to form the reference Wyltl executable. This form of distribution is itself an optimization geared towards reducing the amount of re-used code and duplicated downloads. This is apparent with the distributed file size for the desktop and web Wyltl versions, which fall around 3.00 Megabytes. Additionally, the same approach is followed when distributing Wyltl for Web Platforms, with a 3.10 Megabyte Web Assembly binary being primarily embedded within HTML pages as required. The internal structure followed by Wyltl is shown by Figure One below.

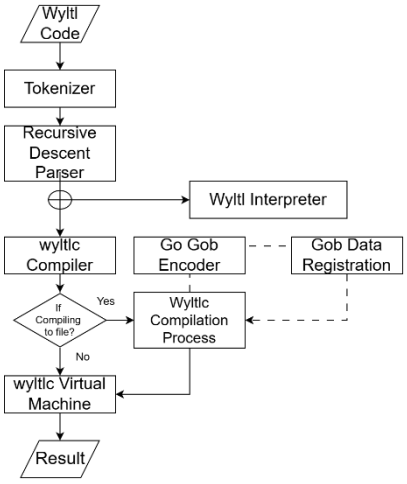

Fig 1 Shared Architecture of Wyltl Implementations

The Wyltl implementations achieve operational efficiency by using this shared architecture to create a 'base' on which the Wyltl Interpreter, and Compiler build upon. The Wyltl tokenizer and recursive descent parser are common among both implementations. This is of importance to the language as it ensures that benchmarking is accurate and is not affected by changes in different components.

The main divergence between the Wyltl Compiler and Interpreter occurs when the input Wyltl code is broken into

tokens by the tokenizer and then arranged into an abstract syntax tree. The Wyltl Interpreter can process this data directly, while the Wyltl Compiler requires compilation to Wyltlc as an intermediate step, then the generated Wyltlc binary would require to be executed by the Wyltlc virtual machine, which would effectively decompile the Wyltlc file back to an abstract syntax tree. With that perspective the Wyltl interpreter can be more effective in execution, as no binaries or intermediaries are created.

The key player is the 'gob' standard library provided by Go, which is responsible for the 'compilation', 'decomplication' and binary creation performed by the Wyltl compiler and Wyltl virtual machine respectively.

V. Evaluating the Wyltl Compiler and Interpreter using Common Algorithms

Wyltl offers the unique research opportunity to evaluate a compiler and interpreter with a shared architecture against each other. De-mystifying the implementation and performance of programming languages with the understanding of them being programs themselves. The key work in evaluating Wyltl is starts with the port of several popular algorithms.

Fig 2 showcases the difference in execution speed between the Wyltl Interpreter and Compiler. It is apparent that the primary speed difference between the compiler and interpreter is primarily visible when executing Wyltl in moderate and low-end devices. To determine the state of optimization in Wyltl, the execution speed in high-end devices must be examined. In this scenario it is visible that the difference between the Wyltl compiler and interpreter becomes narrow, with the interpreter being 8.95% slower than the compiler on average.

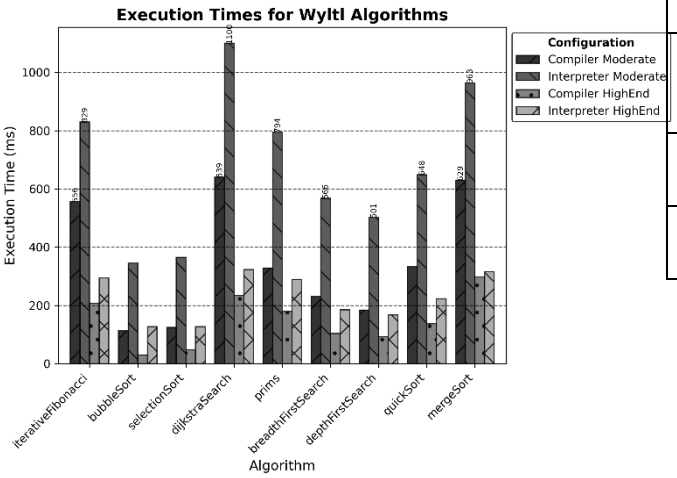

Fig 2 Algorithm Execution Speed of Wyltl Implementations

These results also showcase the dependency Wyltl has on the speed of the CPU (central processing unit) used by the device it is run on.

A somewhat common myth is that a program would use less memory in a faster device due to application executing faster. However, the test-data collected by Wyltl disproves it, by showing that the memory usage in moderate and high-end devices is the same. This consistency in memory usage is caused by the explicit nature of how Wyltl code is handled. Of importance is how the data goes against the concept of memory usage and execution speed being strictly tied together in programming languages.

To accurately gauge the nature of the connection between execution speed and memory usage in Wyltl, the data must be examined with nuance using Go's benchmarking capabilities.

Table XII Performance Metrics of Wyltl Implementations

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 data shown in Table 1 was primarily evaluated after visualization as shown in Figure 4 below.

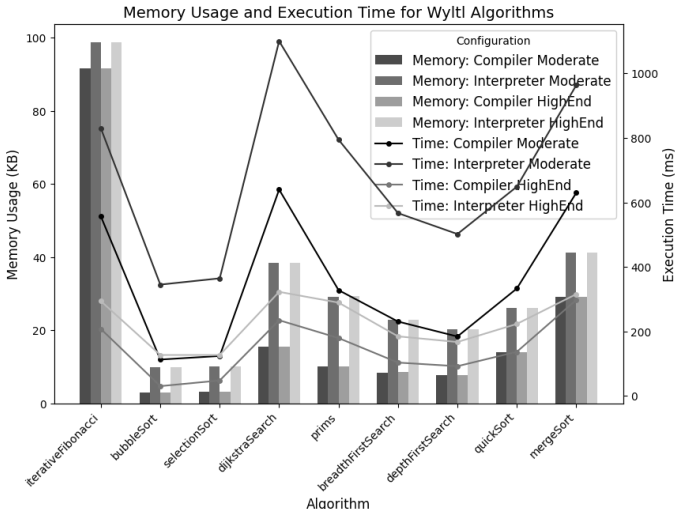

Fig 3 Algorithm Memory Usage of Wyltl Implementations

The primary findings from the visualization of Wyltl performance data is concerned with showcasing the link between the memory allocation count and the execution speed. In a nutshell, the speed of the Wyltl Implementations is loosely tied to the memory usage. However, it is important to note that there is no connection between the memory usage and execution speed across the implementations of different algorithms in Wyltl. This can be further seen when comparing the following calculations of Pearson's coefficient.

Fig 4 Pearson's Coefficient of Wyltl Implementation Memory and Execution Speed

Compiler Memory Allocation Count Vs Compiler Speed (High-End) Slope = 0.1441, Intercept = 37.8236 Pearson r = 0.8957, p-value = 0.0011

Interpreter Memory Allocation Count Vs Interpreter Speed (High-End) Slope = 0.1231, Intercept = 75.6166 Pearson r = 0.9594, p-value = 0.0000

Compiler Memory Allocation Count Vs Compiler Speed (Moderate) Slope = 0.0404, Intercept = -10.4956 Pearson r = 0.8024, p-value = 0.00

```
printFunc := js.FuncOf(func(this js.Value, args []js.Value) interface{} {
        output := args[0].String()
        js.Global().Get("document").Call("getElementById",
        "output").Set("value",
        js.Global().Get("document").Call("getElementById",
        "output").Get("value").String()+output+"\n")
        return nil
})
js.Global().Set("goPrint", printFunc)
```

Fig 5 Overwriting Pre-Defined Web Assembly System Calls with Custom Functions.

Interpreter Memory Allocation Count Vs Interpreter Speed (Moderate) Slope = 0.0325, Intercept = -6.8547 Pearson r = 0.7658, p-value = 0.0161

An important observation to be noted in relation to the Wyltl is its effectiveness in handling processor intensive programs such as Fibonacci and Dijkstra Search. Additionally, while Wyltl memory usage is not affected by the efficacy of the hardware it is run on – the execution speed of the implementations is linked together with hardware. Additionally, the execution of simple programs such as the fundamental sorts are more effective on all hardware. There are algorithms such as iterative Fibonacci and the graphing algorithms– which tend towards comparatively high memory usage as they focus quite heavily on recursion. Recursion is very CPU heavy on Go and Wyltl by extension.

```
wrappedCode := fmt.Sprintf(`
  (function() {
    try {
            return eval(%q);
    } catch (e) {
            throw new Error(e.message);
    }
  })()`, code.Value)
jsValue := js.Global().Get("eval").Call("call", js.Global(), wrappedCode)
```

VI. Role of web-assembly within Wyltl implementations

While programming languages usually support a variety of platforms and systems without issues, the web is usually ignored or left behind. This is a major loss considering the prevalence of web sites and web development in general. Wyltl supports the Web as a first-class platform through the compilation target known as Web Assembly [7]. The central challenge in this approach is that Wyltl is first and foremost a text user interface application, which running on the windows command line or Unix terminal.

The method in this challenge was overcome is by overwriting the default Web Assembly functionality defined by the Go Compiler. This functionality allows Wyltl to print to a defined area of a web page without having to depend on extensive development debts or functions.

VII. JavaScript interoperability in wyltl

Out of the few languages that support seamless execution on Web and Desktop platforms, through Web Assembly or other platforms, none supports interoperability with the JavaScript runtimes present in Web Browsers. This is often a major detriment considering that JavaScript reigns as the de-facto language of the web besides HTML. In the case of Wyltl, JavaScript interoperability is achieved by using web assembly as a 'glue' in the sense of using it as a middleman between the Go-Runtime and the JavaScript engine used by the browser. This allows the Web version of Wyltl to execute any JavaScript code without problems, as compatibility is guaranteed by the Web Browser's own JavaScript runtime. The disadvantage of this process here is the dependency on the JavaScript runtime. This is the only feature that differs among the different versions of Wyltl as the desktop version will not execute such code.

The process of calling the Web Browser's JavaScript runtime to execute code is shown by the code snippet in Fig 6 This code is selectively compiled exclusively for the Web releases of Wyltl. A secondary feature of the JavaScript interoperability present within Wyltl is the conversion of datatypes between Wyltl and JavaScript. This is necessary as even though Wyltl and JavaScript are compiled languages, the way they map and deal with datatypes are somewhat different. However, since the fundamental data types remain the same regardless of the programming language, conversion is mostly a straightforward process.

```
switch jsValue.Type() {
  case js.TypeString:
     return &String{Value: jsValue.String()}
  case js.TypeNumber:
     floatVal := jsValue.Float()
     ,
     return &Float{Value: floatVal}
  case js.TypeBoolean:
     return &Boolean{Value: jsValue.Bool()}
   case js.TypeNull, js.TypeUndefined:
     return nil
  default:
     return &String{Value: jsValue.String()}
}
```

Fig 6 Passing JavaScript code to the Web Browser's JavaScript *runtime*

The simplicity of this implementation is evident by the code snippet presented in figure 7. Wyltl's type system through simplified is feature rich and capable of supporting all the basic types used by JavaScript. The impact of this is twofold. Primarily, data can be transferred without having to depend on 'string' as the primary communication type. Secondly, Wyltl's null safety continues to be insured through this process, which discards any null data returned by JavaScript. However, there is still an expectation that all JavaScript code written by users will contain the necessary safety checks and balances.

VIII. Wyltl and Wyltlc

While the Wyltl Interpreter and Compiler accepts ". Wyltl" files as inputs. The way they are used is fundamentally different. The Wyltl Interpreter works by directly executing the file. This is often slow as the tokenizer and parser would have to execute before the actual interpretation begins. The Wyltl Compiler takes the input Wyltl file and splits it into instructions using the Wyltl tokenizer and parser. Then the compiler registers the types, and statements used within the program using the 'GOB' Go encoder. Then these data is saved to a file with the label '. Wyltlc'

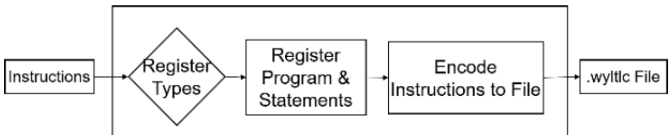

Fig 8 Compilation of Wyltl to Wyltlc

The actual execution of Wyltlc cannot occur directly as the Wyltl virtual machine needs to de-serialize the file and process it instruction & instruction. Whenever an instruction regarding data modification is found, then it as added / popped from the Wyltl Virtual Machine Stack as shown by figure 9 below.

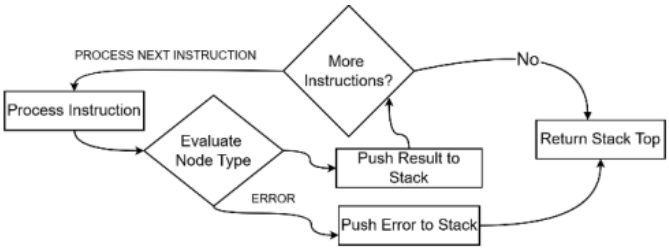

Fig 9 Execution of Wyltlc within the Wyltl Virtual Machine

Processing data through a stack is significantly faster and this is a major contribution towards Wyltl compiler's increased execution speed.

IX. Long Term Maintainability of Wyltl

Many see programming languages as a form of long-term skill investment. Developers expect a language to serve them well for many years or decades to come. One of the main ways that Wyltl ensures this is platform support. Wyltl currently supports Window, Macintosh, Linux and Web as primary platforms. With BSD and bare-metal programming being possible, yet untested platforms. Even in these platforms Wyltl strives to support a range of diverse systems. For instance, the windows release of Wyltl is backwards compatible with Windows 7, in a time where even the Go compiler has dropped support for it. The promise of the Wyltl language is clear and consistent support for all mainstream platforms (and some niche platforms such as BSD).

Wyltl's license has also been the subject of some discussion. A closed source / proprietary approach might bring some initial advantages, but they will all be overshadowed by long term problems – such as contribution. As a programming language grows a collaborative development approach becomes necessary. To ensure that Wyltl is not ill-prepared for such a future – from the main release of Wyltl, it is first and foremost an open-source project licensed under the GNU public license which ensures that all forks and derivatives of Wyltl also remain free and open source so that they can trickle back upstream. As such users can be assured of Wyltl's future.

X. Turing Completeness of Wyltl

A fundamental question asked about any programming language is its completeness. The main measure of a programming language is what applications can be written using it. The main measure of which is algorithm implementation. To this end, many classical algorithms have been implemented in Wyltl. A non-exhaustive list of which is as follows.

1. Dijikstra Search Algorithm
2. Depth First Search Algorithm
3. Breadth First Search Algorithm
4. Knuth Morris Pratt Algorithm
5. Bellman Ford Algorithm
6. Iterative Fibonacci Algorithm
7. Quick Sort Algorithm
8. Merge Sort Algorithm
9. Prims Algorithm
10. Bubble Sort Algorithm
11. Selection Sort Algorithm
12. Insertion Sort Algorithm

The Wyltl implementation of these fundamental algorithms can be interacted with in the Wyltl Playground (https://dineth-lochana.github.io/Wyltl/practice.html). These are 'pure Wyltl' applications in the sense that they do not make use of any external functions brought over from Wyltl's JS language interoperability. The ability of Wyltl to implement these 'Turing complete' functions effectively showcase that Wyltl is a Turing Complete language itself. Wyltl's syntax is flexible enough to allow for reasonably complex constructs such as 2D or 3D arrays along with imperative & functional programming flows. This is shown by the most complex algorithm implemented in Wyltl which is Dijikstra's path finding algorithm. Focusing on the opposite, that is simplicity. An example of a Turing Complete Wyltl Program that also showcases all the fundamental programming traits of – sequence, selection (execution of programming), selection (conditional execution of programs), repetition (re-execution of code) is the 'greeter' program that is showcased by listing 2.

```
suppose x is 1.
suppose y is inputNumber("How many Loops?").
suppose name is inputText("Enter name").

if (name equals "Dineth") {
  print("Hey I'm Dineth too!")
} else {
  while(x below y) {
    print("Hello " plus name)
    suppose x is x plus 1.
  }
}

suppose wellWisher is compose(input){
  print("Have a nice day " plus input)
}
wellWisher(name).

suppose nameArray is stringToArray(name).
for(suppose count is 0 : count below length(name) : suppose count is
count plus 1) {
   print(nameArray[count])
}
```

The greeter program can be executed at the Wyltl Web Playground of which the URL is as follows (https://dineth-lochana.github.io/Wyltl/practice.html?file=greeter.wyltl).

XI. Conclusions

The research insight provided by Wyltl helps to determine the difference between a compiler and interpreter built on a stable and shared base. The performance and memory usage comparisons provided a new perspective on how memory usage cannot be used to accurately predict execution speed. This is a novel finding specific to Wyltl and is of interest to both users of Wyltl and developers using the Go programming language to implement applications or programs. A wider range of testing across the 'unique' platforms supported by Wyltl such as WASM (Web Assembly), would be more useful, however these are not possible currently due to the inaccurate nature of the JavaScript Runtimes of Web Assembly present in popular web browsers.

ACKNOWLEDGMENT

The author would like to thank Dr. Yasas Jayaweera for his invaluable guidance and advice throughout the duration of the project, along with the University of Bedfordshire and SLIIT City Uni for providing the research opportunity for Wyltl.

REFERENCES

[1] Mallawarachchi, D. and Jayaweera, Y. (2025) 'Design and implementation of WYLTL: An imperative, embeddable and portable programming language', International Journal of Computer Applications, 186(76), pp. 22–29. doi:10.5120/ijca2025924669.

[2] 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.

[3] 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

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

[5] Scheifler, R.W. (1977) 'An analysis of inline substitution for a structured programming language', Communications of the ACM, 20(9), pp. 647–654. doi:10.1145/359810.359830.

[6] Franz, M. and Kistler, T., 1997. Slim binaries. Communications of the ACM, 40(12), pp.87-94.

[7] MozDevNet (no date) WebAssembly concepts , MDN. Available at: https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Concepts (Accessed: 20 May 2025).