May 8, 2022

Office: J 331

Dean Sumanth Yenduri,

We, Ethan Hunt, Daniel Tebor, Mohammed Rehaan, and Justin Bull, are a group of CS students looking into how the course content in CS 4308, Concepts of Programming Languages, is received by the students learning the material.

We believe that there is something lacking regarding the delivered concepts surrounding interpreters and believe there should be a language that is easy to interpret to facilitate a bridge between the conceptual knowledge of how interpreters work and the actual implementation that the students are required to do. With how complex languages have become in the modern era, we believe that there should be a very simple and easy to interpret language for educational purposes.

Dr. Sherry Parron, a professor teaching Concepts of Programming Languages, has spoken to only having a 50% success rate for some assignments regarding interpreters. On the surface there seems to be a lack of either understanding of the material, or a lack of appropriate materials for students to interact with. Also, more than half of the respondents to our survey desire a language which would be helpful in building interpreters.

Our dive into the topic is meant to see if creating an easy to interpret language is feasible and would be wanted by the professors and students that are involved in the course.

Regards,

Ethan Hunt, Daniel Tebor, Mohammed Rehaan, Justin Bull

Kennesaw State University

Simple Interpreted Programing Languages

and Corresponding Algorithms (SIPL)

TCOM 2010 Technical Writing

DANIEL TEBOR, ETHAN HUNT, JUSTIN BULL, MOHAMMED REHAAN

Section 07 Spring Semester

5/8/2022

# Executive Summary

This report delves into the research and development of a solution found in the Concepts of Programming Languages class. Part of the course requirement is for students to create a programming language interpreter for simple form of Julia, but due to the complexities of modern-day languages, a large portion of students never even turn in a functional interpreter.

This report includes several highlights. First, we showcase the results of our survey, whose respondents are the students who are taking the Concepts of Programming Languages course. Next is an exploration of the grammar of Julia as well as the *scanner*, *parser* and *executor* algorithms used to interpret it. This includes the potential problems when writing a Julia interpreter due to the language’s complexity. Finally, is a dive into the process of determining “What makes an *easily interpreted language*?”, creating one, and finally creating an interpreter for it.

The schedule for the research spanned from April 8th, 2022, until May 5th, 2022, meaning that the research stayed within the time frame. Also, as planned we were able to complete the research without spending any money.

We appreciate you taking the time to assess our report. Our two final recommendations are to either implement an *easily interpreted language* such as “Esoteric” (the details of which can be seen within this document), or to allow students to create their own *easily interpreted language* and then create an interpreter for said language. Again, we appreciate you for taking the time to consider our recommendations.

# Contents

[Executive Summary 3](#_Toc102942108)

[Contents 3](#_Toc102942109)

[Introduction 5](#_Toc102942110)

[Survey Data 5](#_Toc102942111)

[Simple Form of Julia BNF 7](#_Toc102942112)

[Julia Interpreter 7](#_Toc102942113)

[Scanner 7](#_Toc102942114)

[Parser 8](#_Toc102942115)

[Executor 8](#_Toc102942116)

[Julia Takeaways 8](#_Toc102942117)

[Criteria for an Easily Interpreted Language 8](#_Toc102942118)

[Experimental Language Designs 9](#_Toc102942119)

[Level Based Grammar (LBG) 9](#_Toc102942120)

[Header Based Grammar (HBG) 10](#_Toc102942121)

[The “Esoteric” Language 11](#_Toc102942122)

[The “Esoteric” Interpreter 12](#_Toc102942123)

[The Scanner 12](#_Toc102942124)

[The Parser 12](#_Toc102942125)

[The Executor 14](#_Toc102942126)

[Findings 14](#_Toc102942127)

[Recommendations 14](#_Toc102942128)

[Conclusions 15](#_Toc102942129)

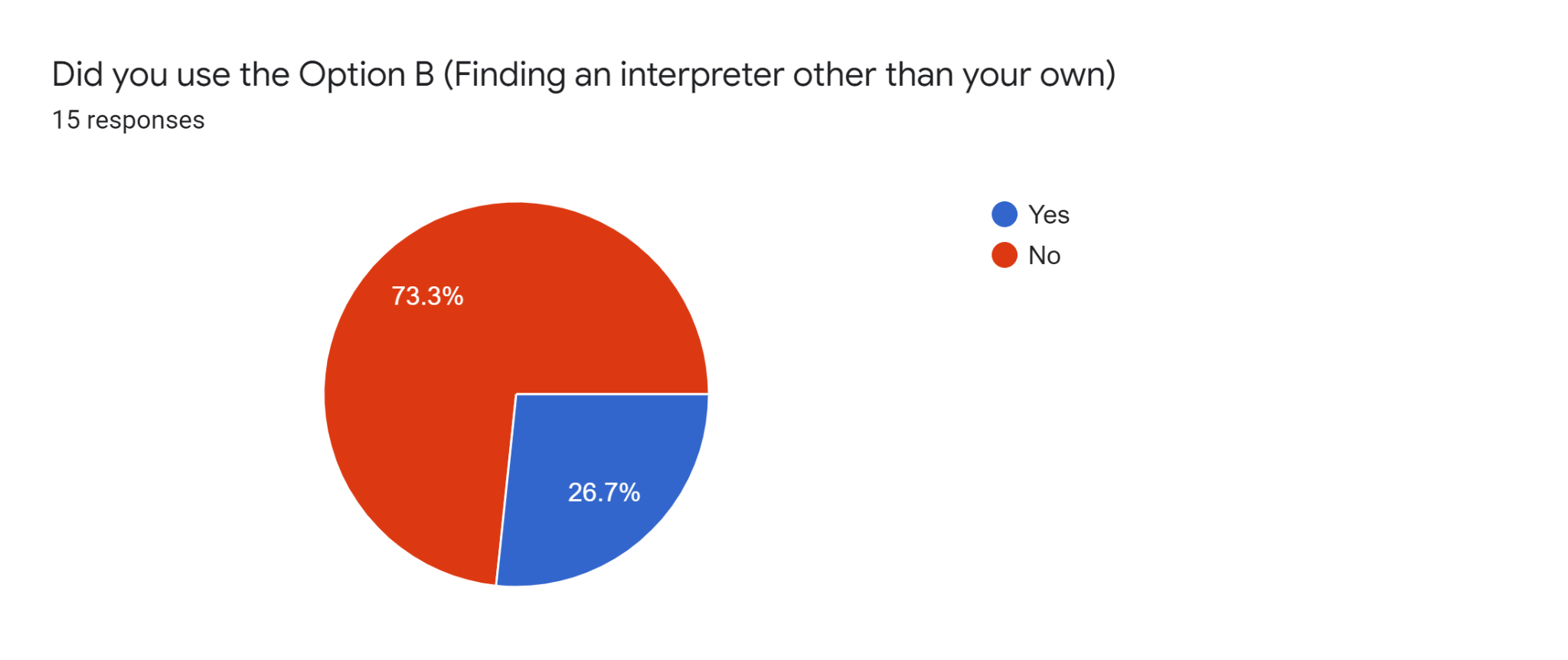
[Sources 16](#_Toc102942130)

# Introduction

Due to the many features and additions that modern programming languages have added to improve development processes, the complexity of these languages has increased dramatically as a result. Early programming languages required you to follow strict spacing conventions making writing a program much more tedious and difficult. Naturally, those languages have been left behind in favor of languages that have more flexible syntax and many more features. Obviously, those features allow for innovative, powerful software. However, for students in the CS Department tasked with creating and demonstrating their knowledge of interpreters, all the possible permutations of code that must be interpreted in a modern language can prove to be an immense challenge, especially when they must create their interpreter in a relatively brief period. Thus, our research team set out to determine whether an *Easily Interpreted Language* (EIL) would be wanted, to determine what makes a language easily interpretable, to create one or more EILs (Easily Interpreted Language) and interpreters for them, and finally to compare the created interpreter one an interpreter made in the Concepts of Programming Languages class for the Julia programming language. The content that follows shows our findings and some recommendations that we have thusly ascertained.

# Survey Data

Our team conducted a simple survey to see whether the language interpretation problem had any merit in the first place. We created a Google Forms survey with 8 questions ranging from measuring people's experiences with the course, the course content, specific choices they made for assignments, etc. Our expectations were that the opinions on the course would be split about 50 / 50, while the open-ended questions would generate some better responses on what specifically the students felt was challenging.

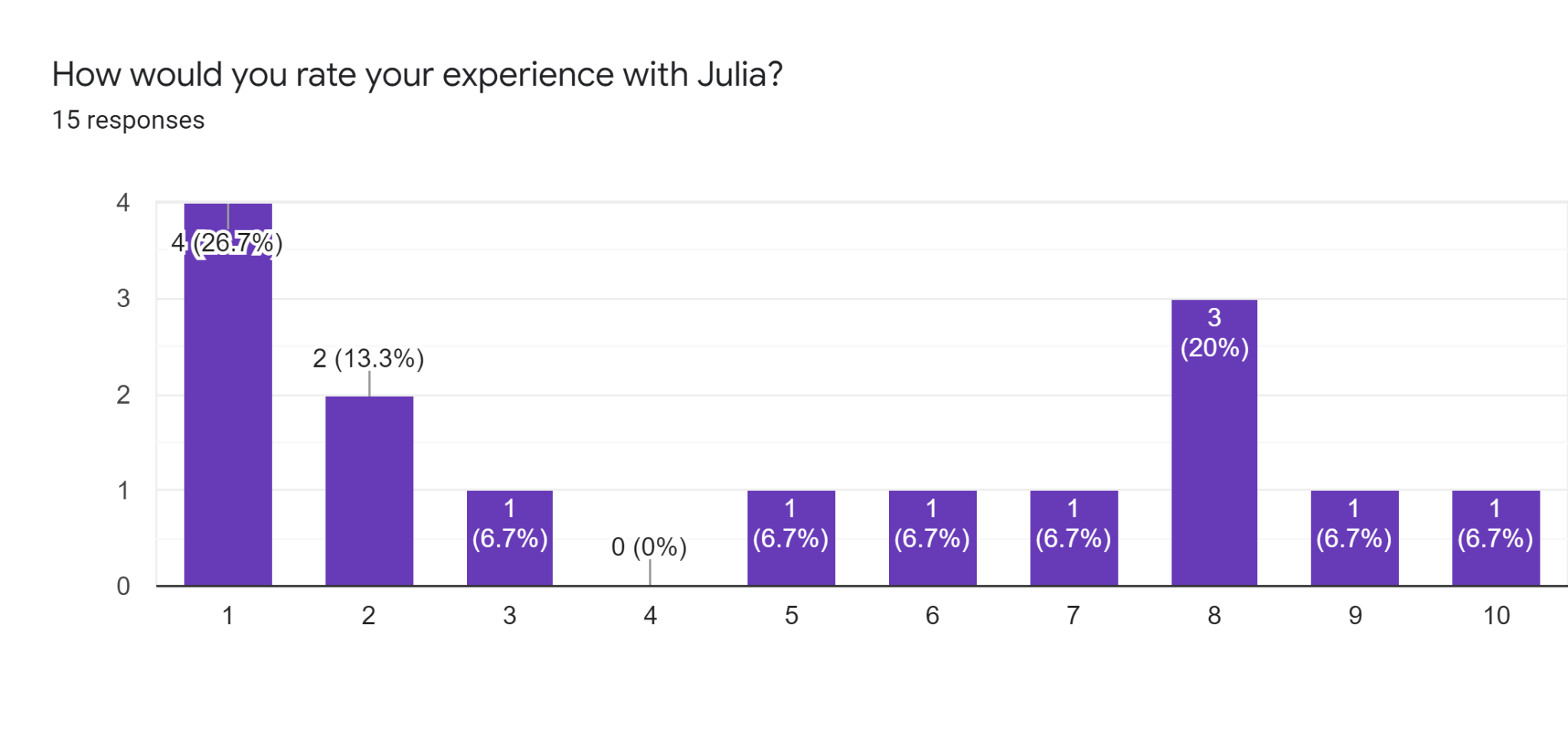
Our first question was an inquiry on if any of the students took an easier assignment with a lower max grade on the Interpreter assignment. We saw that most of the students attempted to create their own Interpreter for the assignment, even if it was the more challenging task.

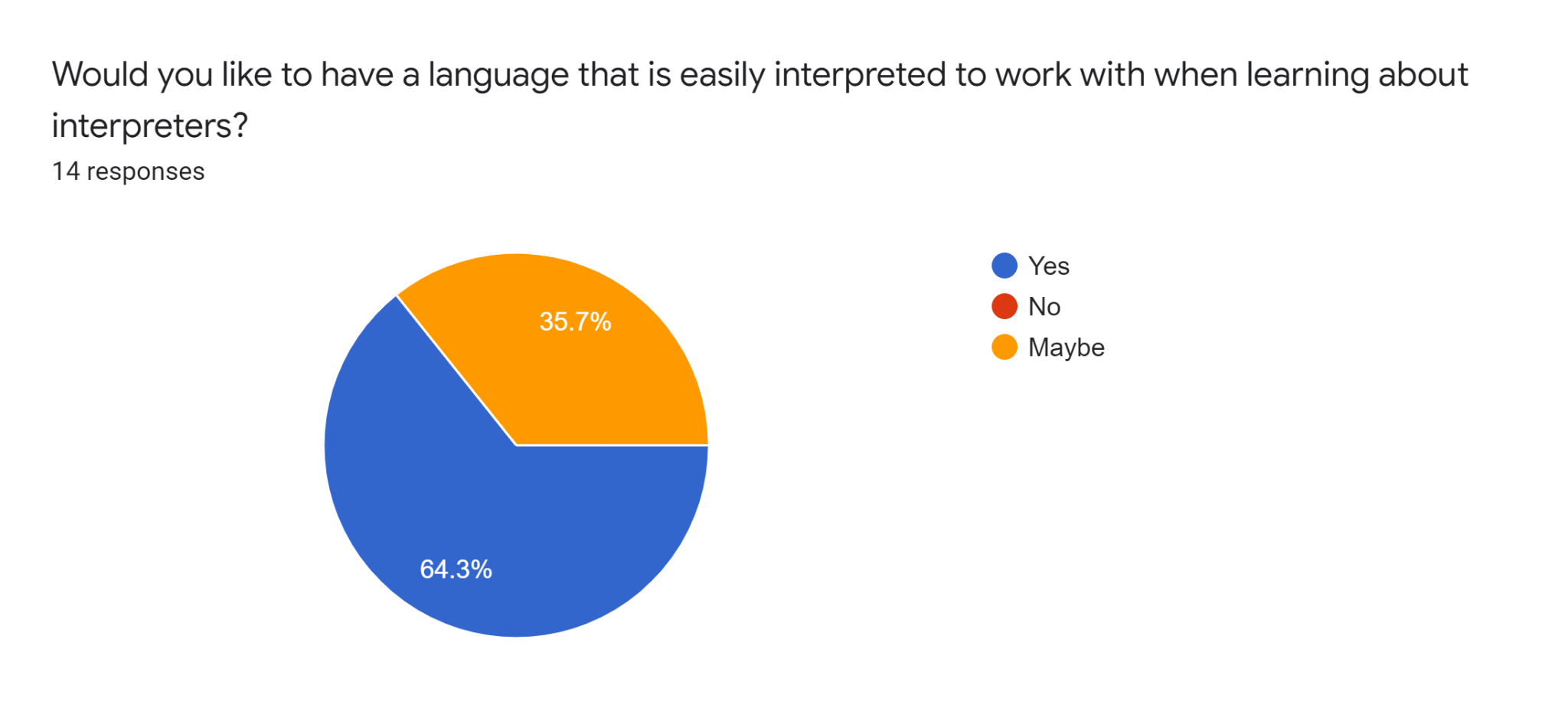
Our second question was rating the students' knowledge on interpreters following the assignment and course material. We found that the students had an average understanding of 6 / 10, which was higher than we believed but was still understandable. However, we still believe that it should be higher.

Third, we wanted to see how the students felt about the instruction on interpreters. The average was 5.9 / 10 for all responses, which we did not find particularly surprising.

Fourth and fifth, we wanted to see how students performed on the interpreter assignment and how they completed the assignment, and surprisingly most of the class had above a 75 on the assignment, with a few lower scores scattered. Along with that, most students used Java as the language of choice when creating the interpreter. This was surprising since this is not the primary language that the students were interacting with in the course.

Sixth was another inquiry on the assignment. We wanted to see what the biggest challenge was from the assignment, so we left an open-ended question for the students to elaborate. What we found from this is that most of the issues with the assignment came from general understanding of the assignment. This came from the instructions on how to interact with the materials, then there were also issues on procrastination, which is something we were expecting to see.

We also asked about students' experience with Julia, as that is the language students were interacting with throughout the course. These results were a little surprising as the students had plenty of time throughout the course to become familiar with Julia, and yet their experiences were negative. The experience on average was a staggering 5 / 10, which is something that we hope to change with an EIL that could be designed for this course in mind.

Finally, we asked if the students would like to have a language that is easier to interpret to interact with and apply the knowledge gained in the course. We were ecstatic to see that most of the students would like to have something like this, with 64 % in favor with the rest opting for a “maybe”, and nobody replying with a “no”.

# Simple Form of Julia BNF

Before looking into new, easily-interpreted, languages, Julia must first be analyzed as it is what students must interpret. The following is a simple form of Julia BNF:

print ()

This grammar can potentially cause problems to people who have entry level knowledge of interpreters because current programming languages do not provide sufficient functionality to compute key functionality of the interpreter. For example, this grammar allows the user to introduce unnecessary white spaces within the statements which can cause problems with generating tokens, and it does not clear delimiters between the statements causing problems to recognize different statements and it also does not provide a mechanism to differentiate between if blocks and the while blocks and its embedded statements.

# Julia Interpreter

The following is the description of the *scanner*, *parser*, and *executor* for an interpreter created for Julia in the Concepts of Programming Languages class:

## Scanner

Reads in a file written in Julia and breaks the lines of the file into separate tokens by checking whether the token matches the content in a hash table which contains the keywords of the language and its associated numeric code. The time complexity of this algorithm is O(n) because we will only have single loops and no embedded for loops and the space complexity is O(n) because of the hash table and the list storing the tokens.

## Parser

The Parser takes in the tokens created by the parser and tries to create a top-down parse tree using recursive parse functions. The parser contains a node element for every nonterminal element of the Julia grammar and has an associated parse method for that node. If the parser is unable to create a parse tree, then there is a high probability that we have a syntactic error in the source file. The time complexity and the space complexity of the parser is O(n).

## Executor

Interpreter takes in the parse tree from the Parser and walks through each node of the parse tree using a loop and it has an interpret method for every nonterminal element of the grammar when it finds a particular node it will call it's associated interpret method. The interpreter has a symbol table which is a hash table holding the variables and their associated values. The time complexity and space complexity of the interpreter is O(n).

# Julia Takeaways

As seen in the Julia BNF and the interpreter made for it, there are complexities in Julia, that make interpreting it somewhat complicated, especially when it comes to differentiating statements. This makes the process of creating an interpreter much more complex than it needs to be for a student just trying to demonstrate their knowledge of interpreters. This is the reason for delving into “What makes an *easily interpreted language*?”.

# Criteria for an Easily Interpreted Language

To determine what is and what is not an easily interpreted language, a set of rules that all EILs must adhere to had to be created. To make concrete definitions, various programming languages were analyzed and compared in terms of how language syntax and semantics are evaluated. The number one thing that was compared in different languages was how each language handled punctuation and parsing. Most programming languages fall under one of two categories when it comes to punctuation.

The first includes languages like Python which do not require the use of strict functions but instead use new lines to tell statements from one another, The concept of one line of code per statement allows for an easier time initially parsing the statements from one another but comes with a major drawback. To preserve the integrity of the instruction set, the statements must be stored in separate sections from one another. This is because if all the lexemes of the input are stored in a simple queue or array there would be no way to tell the difference between statements as they exist with no punctuation. While it seems a trivial issue, it requires that many added steps be taken to make sure that compound statements can be constructed.

The second category consists of languages such as Java, C++, and C that use semicolons to differentiate their end of statements and use various functions to exculpate groups of statements. Unlike the previous category, all instructions end with punctuation, which makes differentiating them from one-another significantly simpler, significantly cutting down on potential development time.

In addition to looking at the punctuation and parsing of languages, other aspects such as naming conventions, embedded statements, and general semantics were compared between languages. Finally, after comparing the pros and cons of various languages the following set of rules were created.

A language *L* is an EIL if it meets 6 out of the following 13 rules

* All Statements have punctuation lexemes
* All variable names are single lower-case letters
* For all *keywords*, the first letter is capitalized, and the *keyword* is no more than 8 letters long
* Must have only one way to write a given statement
* Must have an IF/CONDITIONAL construct
* Must have a LOOP Construct
* Must have the ability to perform abstract substitution
* Must have at least one data type
* Must have I/O system i.e., Print and Input statements
* All Implied Semantics through abstraction of grammar must be interpretable
* Must have Context-free grammar
* All lexemes are separated by white space

# Experimental Language Designs

Many EILs can be created by taking any existing language which has a subset that meets the requirements of an EIL and using that sublanguage to implement the ideas of EILs. This, however, does not allow us to push and expand on the concept of what makes a language easier to implement vs. what does not, as the design of the parent language may or not meet the requirements of an EIL. To answer this question multiple languages were designed with the sole purpose of this in mind to test what works and what does not. Each experimental language had its own grammar and algorithms designed to make interpreting the instructions written in the language as easy as possible. There were many sublanguages and grammatical categories that were created, the most important ones are as follows.

## Level Based Grammar (LBG)

For LBG grammar, all statements that are a child of another statement can be parsed by a single token. This allows for the use of *level parsing algorithms* as that token can be used to denote when to increase the level. The following BNF shows the syntax of a language that follows LBG:

## Header Based Grammar (HBG)

For this type of grammar, each statement has a header that denotes what type of instruction the statement will do. For example, an *assignment* statement would be written as “ASGN a = 10;”. This theoretically allows for statements to easily be parsed based on their header without much fuss and allows for them to easily be sorted into a syntax tree. The following BNF shows the syntax of a language that follows HBG:

## The “Esoteric” Language

“Esoteric” is the language that was created using combinations of LGB and HBG. This language had an interpreter created for it, and the LGB and HBG grammatical rules made creating the interpreter, particularly the *parser* much simpler. The following BNF shows the “Esoteric” language syntax.

# The “Esoteric” Interpreter

The “Esoteric” Interpreter is an interpreter written to interpret “Esoteric”. The purpose of writing it was to prove if the grammatical rules previously established were effective in making “Esoteric” easy to interpret. “Esoteric Interpreter” has three parts: a *scanner*, a *parser,* and an *executor*. To view the actual code for the interpreter, go to <https://cloud.danieltebor.com/index.php/s/38xaAdGMpmYaEfA>.

## The Scanner

The *scanner* takes in an input file that contains code written in “Esoteric” and then processes that code into a readable format that can then be used by the parser. First, all text at and after a “#” is removed. Secondly, all white space and empty lines are removed. As an example:

# The Following is some “Esoteric Code”

a = 30; # Variable Assignment

b = a – 10;

print b;

if a == 30:

b = 0;

$

print b;

a=30;

b=a-10;

printb;

ifa==30:

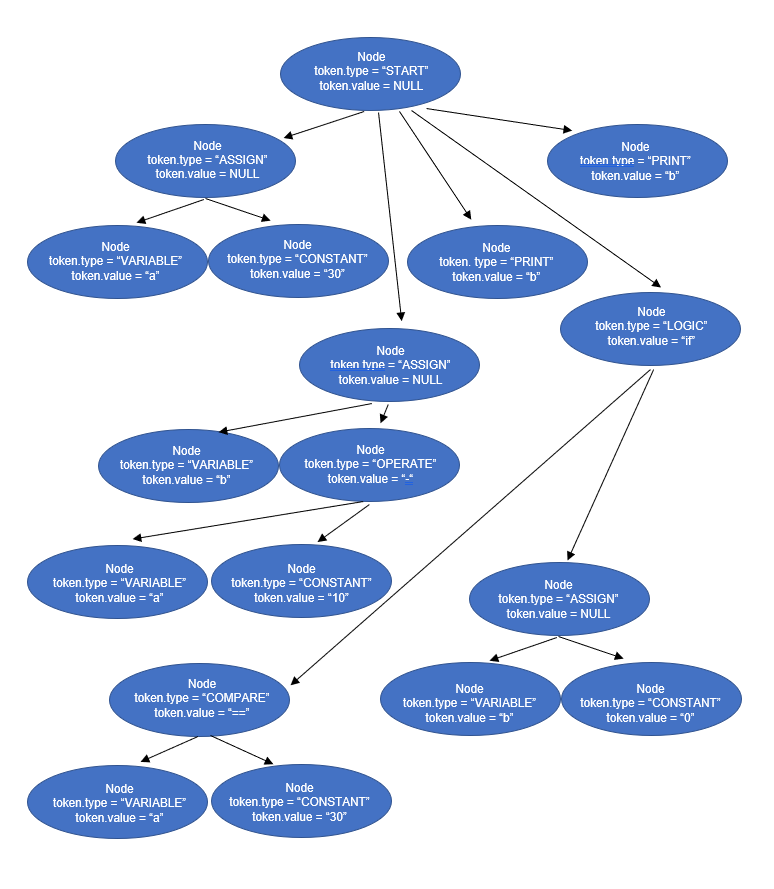
b=0;

$

printb;

## The Parser

The *parser* takes the raw code and then constructs a syntax tree that can then be executed. Because of the simplicity of the code, the *parser* didn’t need to be particularly complex, which significantly cut down on development time during development. Using the code above, the following is the tree is the output of the *parser*:



## The Executor

Finally, the *executor* takes in the *syntax tree* and then executes each node from left to right. The executor determines what to do based on the *token* of each *node*. The following is the output for the above *syntax tree*.

Output:

<< 20

<< 0

# Findings

Through the process of determining “What makes an *easily interpreted language*?” and then creating an interpreter for one, we found that having an EIL makes creating an interpreter substantially simpler, particularly for the *parser*. The lack of necessity to differentiate between many different permutations of statements, makes the parsing algorithm much less complex. This is especially apparent when comparing the interpreter for “Esoteric” to the interpreter for Julia. The interpreter for Julia significantly more complex, as it should be. “Esoteric” doesn’t have many of the features that other modern programming languages have. This is the tradeoff for simplicity, and for a class such as Concepts of Programming Languages in which students must demonstrate their knowledge on interpreters, this is perfect. In fact, our survey findings corroborate this, as a substantiate population of the students who took the survey said that they didn’t have a great understanding of Julia, and an overwhelming portion of the students said that they would like to have an EIL as an option.

# Recommendations

We believe that implementing an *easily interpretable language* could be a workable approach for creating better course materials when teaching the topic of interpreters. While there are many steps, we believe that there wouldn’t be much of an overhead investment in implementing something of this nature, and we know that having an EIL is something that students would appreciate. Thus, we have two recommendations:

One is to implement an EIL such as “Esoteric” into the course material instead of Julia, or perhaps have said EIL as a possible alternative option.

Two is to allow students or groups of students to create their own EIL, including a BNF for it as well as their rationale for making it the way that they did, and then creating an interpreter for their EIL. This would allow students to gain a deeper understanding of the course material by creating their own language, exploring what about a language makes it easy to interpret, and finally creating an interpreter for their language.

Obviously, these two choices have their ups and downs. The first option is much easier to grade, since the professor would only be dealing with one (or maybe two) programming languages, but the downside is that students may not gain as deep an understanding of the material as they would if option two were to be implemented.

# Conclusions

Having created a mock-up of what an EIL could look like, we have found that implementing something like an EIL into the course material could be greatly beneficial to the Concepts of Programming Languages course. Exploring EILs is something that could improve students' outlook on a course that covers topics like interpreters, as having something that is easy to interact with may lead to the students interacting with the course materials more. With the average experience being shown through survey results to be around a 5-6 / 10, it would be beneficial to push that number as high as possible, while not sacrificing anything in the process. Finally, we hope that our findings are insightful and make their way into the Concepts of Programming Language course.

# Sources

Corob-Msft. “C++ Docs - Get Started, Tutorials, Reference.” *C++ Docs - Get Started, Tutorials, Reference. | Microsoft Docs*, https://docs.microsoft.com/en-us/cpp/cpp/?view=msvc-170.

“Java™ Programming Language.” *Java Programming Language*, https://docs.oracle.com/javase/7/docs/technotes/guides/language/.

“Julia 1.7 Documentation.” *Julia Documentation · The Julia Language*, https://docs.julialang.org/en/v1/.

“Python 3.10.4 Documentation.” *3.10.4 Documentation*, https://docs.python.org/3/.