April 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.

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

Comp Sci Dev Research Proposal

Ethan Hunt, Daniel Tebor, Mohammed Rehaan , Justin Bull

Kennesaw State University

Technical Writing 2010 Section 07 Spring Semester 2022 CO

Jack Labriola

8 April 2022

# Executive Summary

The purpose of this proposal is to bring to light and provide a solution to an issue found in the Concepts of Programming Languages class. Part of the course requirement is for students to create a programming language interpreter, but due to the complexities of modern-day languages, a substantial portion of students never even turn in a functional interpreter.

This proposal includes several highlights. The first is detailed background information on the reasons why complexities in programming languages can create complications for students trying to demonstrate their knowledge on building interpreters. The proposed solution to the issue at hand is, briefly put, the development and implementation of a programming language designed around standards that make it affable for the implementation of an interpreter. In addition, the planned phases for research and development are also detailed in this proposal.

Finally, the schedule for the proposed research will span from now until May 2nd, and the final report will be completed by May 5th. As one final note, there is no proposed budget for this research.

# Table of Contents

Contents

[Executive Summary 2](#_Toc100260747)

[Table of Contents 2](#_Toc100260748)

[Introduction 3](#_Toc100260749)

[Background Information 4](#_Toc100260750)

[Examples of Common Issues 4](#_Toc100260751)

[Research Methods 6](#_Toc100260752)

[Phase 1 6](#_Toc100260753)

[Phase 2 6](#_Toc100260754)

[Phase 3 7](#_Toc100260755)

[Phase 4 7](#_Toc100260756)

[Phase 5 7](#_Toc100260757)

[Project Schedule 7](#_Toc100260758)

[Conclusion 8](#_Toc100260759)

# Introduction

Due to the complexities that make up most modern programming languages, interpreting these languages has become increasingly more complex. With many added features and complex syntax, those tasked with creating either an interpreter or compiler for the language will have a more difficult time. Such is the plight of students in the CS Department, who are tasked to create an interpreter to demonstrate their understanding of the core concepts of implementing programming languages. Unfortunately, many students struggle to get past the more complex functionality of the languages and fail to express their true understanding of the material due to the complex languages they must work with. Based on the statements of Dr. Sherry Parron given in her *Concepts of Programing Languages* lectures, she expressed that a little under 50% of the students were able to complete the task of creating a simple interpreter. To help solve this issue we propose the research and development of an esoteric programing language designed for the sole use of allowing students to more easily demonstrate their understanding the concepts surrounding a programming language’s implementation. The foundation for this research will be on existing programming language theories and the esoteric programming *EIL*, which stands for the *Easily Interpreted Language*.

# Background Information

The process of implementing a language with an *interpreter* can be summed up to three main components. The first is the *scanner* does the job of taking in the input code written in a specific language and then identifying all the different “atomic” components of the programs known as lexemes. Each lexeme is assigned a token which is a unique identifier that tells the lexemes type. The second part is the *parser* which takes the information gathered from the *scanner* and organizes it into meaningful statements that follow language syntax and semantics. The third and final part of an *interpreter* is the *executor* who takes each of the statements created by the *parser* and executes the task specified in each statement. Now with a brief background of the general process examples of the unnecessary complexity some languages have that would cause trouble to students can be explored.

## Examples of Common Issues

The first big issue that needs to be addressed is the lexeme format, which involves how different lexemes are differentiated from each other and how to assess valid vs invalid lexemes. This issue is found in the first phase of implementing an interpreter, the *scanner*. An intuitive approach to solving these problems uses a similar syntax approach to the English language. This would have each lexeme like a word in English be separated by white space. This does not however account for mathematical expressions. Take for example the expression “x = 2 + 3*”* written in no particular style. The following are all valid ways of witting the expression in C++:

* x=2+3;
* x= 2+ 3 ;
* x=2+ 3;
* x= (2 + 3) ;

Like all the expressions listed as well as many more, it is easy in isolation to distinguish that they all mean the same thing. As white space is something that is heavily stylistic, it would be advantageous to eliminate unnecessary ambiguity in programming languages. However, many programming languages allow similar things to be allowed as far as spacing is concerned. The *scanner* must decide if only one or all those ways of writing the expression are valid. It does this in part by checking the input for valid lexemes. Which leads to the question of whether lexemes must be separated by white space and if not, how do you split them apart?

Another issue is the limitations on possible variable and function naming conventions. What constitutes a valid name from an invalid one? What if a variable is named the same as an already defined keyword? These cases must all have a way to be handled which requires additional code on top of the code used to evaluate simple cases. This greatly increases complexity and development time. Students who have never done anything on such a scale before might be at a loss with how to even start and thus might have tremendous difficulty even getting the simple parts of their *scanner* completed.

The second phase of implementation, the *parser*, deals with how we differentiate between different statements. This phase also has its own share of unique problems when it comes to language complexity. One of the most important examples is how to tell when a statement ends and a new one begins. Should there be punctuation like a period like used in English or should there only be only one statement per line? If punctuation characters are used, how are embedded statements differentiated? Take for example the English statement “If x equals y, then z = 2x. Else, z = x + y.”

**The following code is the above statement written in C++**

if (x == y) {

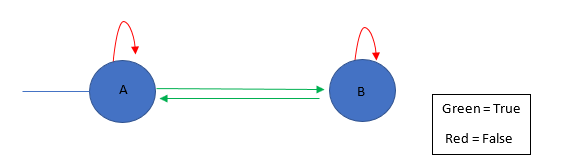
z = 2 \* x;

} else {

z = x + y;

}

As seen in the example above the characters “{“ and “}” act as an encloser for the “if“ and “else“ statements where the simple assignment statements use the character “;” to punctuate themselves. It seems easy enough to implement a simple check for a “{“ character and then find its counterpart. However, issues lie on two different fronts. The first is how do you change from separating based on a semicolon to separating based on a curly bracket and vice versa. A change in the state must occur which lends itself to the creation of a finite state machine to handle this. A finite state machine is a logical circuit that keeps track of the state of a series of inputs. These can be amazingly simple to extremely complex and sometimes are often challenging to implement.

**The following is an encloser finite state machine**

In part three, the *execution*, a common issue is the order in which we execute and perform the statements. For example, the order of operations in an expression is another common design issue. Should a prefix, a postfix, and/or a infix notation be used for writing out the expressions? The n-fix notation is used to describe the order in which the value and operator are placed in an expression. The expression “A+B” written in infix can be expressed in prefix as “+AB” or in postfix as “AB+.” Each has its own benefits to using but most human language has such expressions as infix, so programming languages tend to go with infix as well.

Another question that arises is whether expressions should follow standard orders of operation or a separate set of priorities. For example, the statement “x = 2 + 3 \* 1 – 0 / 4” can be interpreted in a few ways. One way is to read the statement recursively from left to right, doing the operations in the order in which they appear. Another way is to assign priorities to each of the operators is similar to traditional “order of operations” used for mathematics. It might be easy to say the first option is wrong because it violates the traditional mathematical order of operations, however this can be circumnavigated using abstraction. If you wanted to write the following expression of “x = 2 + 3 \* 1 – 0 / 4” in a “first-come, first-serve” implementation you would write the following statements. Let a = 3 \* 1 and b = 0 / 4 such that x = 2 + a - b. By using additional variables, we can achieve the mathematically correct answer with a “first-come, first-serve approach.” While the issue seems trivial at first glance, implementing a “first-come, first-serve” approach is far simpler than an approach founded in “order of operations.” By using an easier method of interpreting an expression, the time spent developing the code to write a more complex version could be spent on more key features.

# Research Methods

The research process for the development and implementation of an easy to interpret language will have 4 distinct phases:

## Phase 1

In phase 1, metrics and definitions on what we will use to base our analysis will be determined. This will consist of first defining what constitutes an easily interpreted language vs one that is not. After a preliminary round of definitions has been created data would be collected from preexisting languages to help validate and improve the definitions. Once complete, a bedrock of standards will be established.

## Phase 2

Phase 2 will consist of developing a high-level understanding of a language using the standards established in phase one as its foundation. Multiple languages will be created using the established standards and each will be subjected to analysis based on the ease of implementing an interpreter. After defining our languages and comparing them to one another, algorithms used to implement the languages will be developed. Multiple algorithms used for the task will be created using a variety of methodologies. These methodologies will range from greedy to brute-force techniques among many more. The algorithms will be split up into the categories of Lexeme Collecting, Parsing, Memory/Maintenance, and Execution algorithms. The performance of the algorithm will be assessed with both time and space requirements. In addition, diagrams of the algorithms and their implementation will be presented to help further the understanding of the methodologies presented. Comparisons with other existing languages will be conducted to provide additional context and help verify that the standards that were set in phase one were met.

## Phase 3

During phase 3, prototypes of the top-performing algorithms and interpreters of top-performing languages will be developed. The development process of making the prototypes will be tracked using time spent on development, number of large revisions, and percent complete/accuracy of created algorithms. The accuracy of the algorithms will be assessed by comparing hand complied outputs vs machine complied outputs. Test files and the output created from those test files will be used to track how the algorithms were tested.

## Phase 4

For phase 4, we will then use the data collected from phase 3 to assess the viability of implementing the languages and algorithms. In addition to using similar standards to what was set forth in phase one, some additional standards will be added based on the results of phase 3. Some possible standards may involve looking at the estimated time required to complete the interpreter in the language. Another possible standard would be assessing the number of possible issues that may lead to pitfalls in development. These factors will allow a better understanding of the feasibility of implementing the developed language in given scenarios. In addition, this phase will also include comparisons of the development times of other languages and comparing them against our language.

## Phase 5

Based on the results of the previous four phases of research Phase will be the compiling of all the information and data gathered. This phase will also include our findings and results as well as any additional information pertaining to the subject. This may future research and applications.

# Project Schedule

\*schedule is subject to change but the project will be completed on May 5

* Tuesday, April 12: Phase One complete
* Tuesday, April 19: Phase Two complete
* Tuesday, April 26: Phase Three Complete
* Monday, May 2: Phase Four and Presentation Complete
* Thursday, May 5: Final Report Due

# Conclusion

To reiterate the topic of the proposed research, the information gained from understanding the process and requirements of implementing a programming language designed specifically for simplifying interpretation would allow students to potentially have a better opportunity to showcase their knowledge on the subject matter. This would allow for a better understanding of what parts of language design would hinder students and what parts would enable students. If complex portions of a language cannot be simplified, at least simplifying trivial parts would allow for focusing on more important skills rather than extremely trivial details that are essential for the completion of an interpreter in certain languages. This research may bear unexpected results as some design techniques could be implemented by other languages in the future to help the overall field of computer science.