

The Inoxis Language and Interpreter

Project Proposal by

Michael Clausen

CPSC 589

Fall 2024

Professor Dr. Kenneth Kung

Cal State University Fullerton



CALIFORNIA STATE UNIVERSITY
FULLERTONTM

Department of Computer Science

CPSC 597 / 598 PROJECT / THESIS DEFINITION

To the graduate student:

1. Complete a project proposal, following the department guidelines.
2. Have this form signed by your advisor and reviewer / committee.
3. Submit it with the proposal attached, to the Department of Computer Science.

☒ Project

☐ Thesis

Please print or type.

Student Name: Michael Clausen Student ID: 886601202
Address: 801 N Loara St # 66, Anaheim, CA 92801
Street City Zip Code
Home Phone: (661) 706-8865 Work Phone: ----
E-Mail: mbclausen@csu.fullerton.edu Units: 3 Semester: Spring

Are you a Classified graduate student?

☒ Yes

☐ No

Is this a group project?

☐ Yes

☒ No

Proposal Date: 12/18/24

Tentative Date for Demonstration

/Presentation/Oral Defense: 5/8/25

Completion Deadline: 5/16/25

Tentative Title: The Inoxis Language and Interpreter

We recommend that this proposal be approved:



Faculty Advisor	<u>Kevin Wortman</u>	<u></u>	<u>12/17/24</u>
	Printed name	Signature	Date
Faculty Reviewer	<u>Kenytt Avery</u>	<u></u>	<u>12/17/24</u>
	Printed name	Signature	Date
Faculty Reviewer	_____	_____	_____
	Printed name	Signature	Date

Table of Contents

1. Introduction
2. Objectives
3. Project Activities - Software Requirements Specifications
4. Project Environment
5. Project Results
6. Project Schedule
7. Appendix - Inaxis Example Program

1. Introduction

In this introduction, I will first survey the field of compilers and interpreters. Then, I will overview the current problems in the field and discuss the strengths and weaknesses of proposed solutions. Lastly, I will outline my project, which will provide a solution to the problems mentioned.

Fundamentally, a compiler is an implementation technique that translates a program in one language into a different language. These are known as the source and target languages, respectively. Usually, the source language is a higher-level language that is easier to program in, such as C++, and the target language is one that a computer can easily understand, such as assembly language (Aho et al., 2023). A stereotypical compiler like GCC generates an executable at the end of this process, which the user runs (Nystrom, 2021).

An alternative to generating such an executable is directly running the code from source on user inputs. This is what an interpreter does. Compilation and interpretation are not mutually exclusive, as interpreters can contain a compiler that works behind the scenes. For example, CPython executes programs from source but internally compiles the source code into bytecode. There are few “pure” interpreters that do not contain such compilers internally, one counterexample being the original implementation of the Ruby language (Nystrom, 2021).

A compiler translates the source code in eight phases. The first four, lexical analysis, syntax analysis, semantic analysis, and intermediate code generation, constitute the analysis phase, commonly known as the compiler's front end. This is followed by the synthesis phase, which generates the code for the target language, known as the back end of the compiler. Between these two phases, code optimization is typically performed (Aho et al., 2023).

The first step is lexical analysis, also known as scanning. This involves taking the source code character stream and breaking it into lexemes, usually known as tokens. Irrelevant characters, such as white space or comments, are also removed. These tokens are words in the source programming language, comparable to words and punctuation in an ordinary language such as English. For example, the C++ statement “int num = 81;” can be broken into five tokens: int, num, 81, ‘;’, and “=” (Aho et al., 2021).

Next, syntax analysis, also known as parsing, is performed. The parser takes all of the tokens and stores them in an intermediate data structure such as a syntax tree. The syntax tree contains the program's grammatical structure. For example, an operator such as ‘+’ would be a node in the tree, while operands such as ‘12’ and ‘7’ would be its children (Aho et al., 2023).

This step is followed by the semantic analyzer checking the intermediate representation created by the parser for correctness. In other words, it ensures that the source program conforms to the rules of the language in which it was written. Information on the data types of the variables can also be stored at this phase (Aho et al., 2023).

Next, many compilers use the data from the analysis phase to generate code in an intermediate representation that runs on a hypothetical machine. The representation will ideally be straightforward to translate into the instruction set architecture of the various physical chips that the source language will be run on, such as ARM or x86. This way, only a code generator translating the intermediate representation into each ISA needs to be created instead of a whole other compiler for each ISA (Aho et al., 2023; Nystrom, 2021).

After the analysis phase, most compilers do a code optimization pass. The source code, stored in an intermediate representation, can be optimized in various ways, usually by decreasing the run time. This is significantly easier, given the intermediate representation. Optimization of

the run time is usually accomplished by eliminating “unnecessary instructions in the code” and the “replacement of one sequence of instructions by a faster sequence of instructions that does the same thing” (Aho et al., 2023, chapter 9 para. 1).

Finally, the back end has been reached, and the code for the target language is generated. There are two options at this stage: generating actual code for a target CPU or generating code for one that does not exist, known as a virtual machine. Generating virtual machine code allows the compiler to be used with any architecture. However, the more difficult task of writing a back end for each ISA that generates actual machine code will significantly increase the program’s performance. All compiler designers must weigh these tradeoffs when designing this final stage (Nystrom, 2021).

With the overview completed, current issues in the field of compilers and languages can be discussed. The main issue that will be the focus of my project is the need for security-conscious languages and compilers for said languages. Software security has become a touchstone issue in recent years, with security failures becoming global news stories, such as the CrowdStrike fiasco (Banfield-Nwachi, 2024). The language that an application is written in has a significant impact on how secure it is. A prime example of commonly used insecure languages is C and C++. The security issues with C and C++ are so pressing that the Biden administration released an official memo, “Back to the Building Blocks: A Path Towards Secure and Measurable Software,” which urges developers to avoid using these languages in the future (The White House, 2024).

Alternatively, Rust has increasingly become the exemplar of security-conscious languages. It provides the speed of C and C++ while safely managing memory. This is accomplished by the Rust compiler enforcing strict rules concerning memory

management. For instance, all variables are “immutable” by default, meaning their values cannot be subsequently changed. Further, every value has an “owner” in Rust, which becomes especially important when memory is allocated from the heap. Only the owner can change the data that it points to. When the owner variable goes out of scope, the heap memory it points to is automatically freed. This way Rust manages memory safely without utilizing a garbage collector, keeping code written in Rust fast. Rust does allow references to heap data, which can be accessed by non-owning variables, which is called a “borrow” (The Rust Project Developers, n.d.).

Though Rust is impressive in that it maintains memory safety while having the speed of C and C++, it is a difficult language to learn, especially for C/C++ programmers who want to make the transition. The syntax is significantly different, and many features that the C++ standard library supports are instead implemented using macros in Rust. I propose to create a new language called Inoxis, which will have Rust's memory safety features while using syntax much closer to C++.

In order to implement the language, I will need to build a compiler for it. Of the multiple different ways to build said compiler, I am going to write an interpreter in C and C++ which will utilize a bytecode virtual machine for the back end. In other words, the interpreter will not target any real CPU, but a virtual one whose instruction set I will develop myself. This will make sure that my project is unique within the field.

2. Objectives

The primary objective of this project is to develop a new programming language called Inoxis and to write an interpreter for the language. Inoxis will be Rust-like with C++-style syntax. The interpreter will utilize a bytecode virtual machine with an instruction set of my own design. I will also write a test suite of Inoxis programs to test the interpreter.

3. Project Activities - Software Requirements Specifications

The Inoxis Language

- Inoxis will be statically typed.
- It will have one data type, the integer type.
- It will have arrays which can be allocated from the heap.
- It will have mutable and immutable data types.
- Variables will be immutable by default.
- Each value will have an owner.
- Once the owner goes out of scope, its memory is freed automatically.
- Inoxis will have pointers and references.
- References allow values to be borrowed.
- Borrowers cannot change the values they borrow.
- Each function has one parameter.
- Each function must return a value.
- Pointers and references can be passed as function arguments.
- Pointers and references can be returned by functions.
- Inoxis will have a built-in “print” function.
- It will have “while” loops.

The Inoxis Interpreter

- The interpreter will read an Inoxis source file from the command line.
- It will run on Windows 11.
- It will be written in C and C++.
- It will interpret all Inoxis programs correctly.
- It will use ANTLR to generate the parser.
- It will run in a reasonable amount of time.
- It will have a bytecode virtual machine for the backend.
- The virtual machine will use an instruction set of my own design.
- The virtual machine will be register-based.

4. Project Environment

Hardware

I will be using my Gigabyte G5 KC custom laptop PC using Windows 11 Home. It has an x64-based Intel Core i5 2.5 GHz 6-core processor and 16 gigabytes of RAM.

Software

I will be using Microsoft Visual Studio as my IDE, the C language and standard libraries, the Microsoft C Compiler, and ANTLR as the parser generator.

5. Project Results

Project Deliverables

- A fully defined grammar for the Inoxis language.
- A test suite of example Inoxis programs.
- A full interpreter with a bytecode virtual machine that can interpret and execute any Inoxis program correctly.

Quality Measures

- The interpreter will correctly interpret all of the test suite programs. If the program is correct, the interpreter will execute the program, generating the correct output. If the program is incorrect, the interpreter will generate an error that will correctly identify the problem by line number and error type.
- The interpreter will accomplish the above in under 45 seconds.

6. Project Schedule

Each Component (eg. Weeks 2-3 Write Language Grammar) will take approximately 30 hours.

Week 1: Project Initialization and Planning (this should be done this semester)

1. Document high-level specifications for the toy language
2. Create small example programs written in the toy language
3. Find and Compile Resources

Weeks 2-3: Write Lexer

1. Set up code for error handling.
2. Write code to eliminate white space.
3. Write tokenizer.

Weeks 4-5: Write Language Grammar

1. Write the list of all syntax production rules for Inoxis in Extended Backus-Naur Form (EBNF) for simplicity.
2. Create the list of reserved words for Inoxis.
3. Convert the grammar into the input format for ANTLR.

Weeks 6-7: Create Parser using ANTLR

1. Grammar should already be in the input format for ANTLR.
2. Integrate ANTLR into the project. Make sure the scanner works with it.

Weeks 8-9: Create Abstract Syntax Tree

1. ANTLR creates the AST as output. Integrate it so that the AST can be used as input for the VM.

Weeks 10-11: Design VM Instruction Set

1. Design VM architecture, including number of registers, the program counter, and the addressing modes.

2. Decide on which operations the VM supports (arithmetic, jump, etc).

Weeks 12-13: Write VM

1. Implement VM instruction set
2. Implement function declarations.
3. Implement Local Variables
4. Implement Expressions
5. Implement return
6. Write Control Flow

** Time Permitting **: Write Code Optimizer

1. Do a peephole pass
2. Inline functions

Weeks 14-15: Testing, Integration, and Documentation

1. Create test suite of example Inaxis programs. These will include correct and incorrect programs.
2. Integrate all the parts of the project.
3. Add comments and headers.
4. Write final report.
5. Make presentation.

7. References

- Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D., & Bansal, S. (2023). *Compilers, principles, techniques, and tools* (Updated second edition.). Pearson India Education Services Pvt.
- Banfield-Nwachi, M. (2024, July 19). Windows global it outage: What we know so far. The Guardian. <https://www.theguardian.com/technology/article/2024/jul/19/windows-global-it-outage-what-we-know-so-far> Ltd.
- Cooper, K., & Torczon, L. (2012). *Engineering a compiler* (2nd ed.). Elsevier.
- Introduction to Rust Programming Language. (2021, March 18). GeeksforGeeks. <https://www.geeksforgeeks.org/introduction-to-rust-programming-language/>
- Nystrom, R. (2021). *Crafting interpreters*. Genever Benning.
- The Rust Project Developers. (n.d.). *The Rust programming language*. The Rust Project. Retrieved December 7, 2024, from <https://doc.rust-lang.org/book/title-page.html>
- Scott, E., Johnstone, A., & Economopoulos, R. (2007). BRNGLR: A cubic tomita-style GLR parsing algorithm. *Acta Informatica*, 44(6), 427–461. <https://doi.org/10.1007/s00236-007-0054-z>
- The White House. (2024). *Back to the Building Blocks: A Path Toward Secure and Measurable Software*. <https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf>

8. Appendix – Example Inoxis Program

```
// function declarations

int* gives_ownership(param);

int* takes_and_gives_back(int* arrayParam);

int calculate_total(int& arrayParam);


// main

int main(param)
{
    // variable is read-only (immutable by default)

    int var1 = 10;

    print("The value of var1 is {var1}");

    // variable can be changed

    int mut i = 0;

    // gives_ownership moves its return value into array1

    int* array1 = gives_ownership(1);

    // array object allocated from the heap, array2 comes into scope

    int* mut array2 = new array[10];

    while(i < 10)
    {
        myArray[i] = i;

        i = i + 1;
    }
}
```

```

// array2 is moved into takes_and_gives_back, which also moves its return
// value into array3
int* array3 = takes_and_gives_back(array2);
// pass array2 by reference to calculate its sum
int total = calculate_total(&array2);
print("Total = {total}");
return 0;

// here, array3 goes out of scope and is dropped. array2 was moved,
// so nothing happens. array1 goes out of scope and is dropped.
// (dropped meaning its memory is automatically freed).
}

```

```

// function definitions

// gives ownership will move its return value into the function that calls it
int* gives_ownership(param)
{
    // A comes into scope
    int* mut A = new array[10];
    int mut i = 0;
    while(i < 10)
    {
        A[i] = i;
        i = i + 1;
    }

    // A is returned and moves out to the calling function
}

```



```

        return A;
    }

// This function takes a pointer to an integer array and returns one
int* takes_and_gives_back(int* arrayParam) // arrayParam comes into scope
{
    // arrayParam is returned and moves out to the calling function
    return arrayParam;
}

// this function takes a reference to an integer array
// and calculates the sum of its contents
int calculate_total(int& arrayParam)
{
    int mut i = 0;

    int mut total = 0;

    while(i < 10)
    {
        total = total + arrayParam[i]
    }

    return total;
}

```