CALIFORNIA STATE UNIVERSITY
FULLERTON™

Department of Computer Science

This project has been satisfactorily demonstrated and is of suitable form.

This project report is acceptable in partial completion of the requirements for the Master of Science degree in Computer Science.

The Inoxis Language and Interpreter
Project Title (type)

Michael Clausen
Student Name (type)

Kevin Wortman
Advisor's Name (type)

Advisor's signature Date

Kenytt Avery
Reviewer's name

Reviewer's signature Date

The Inoxis Language and Interpreter

Project by

Michael Clausen

Abstract

This report presents an overview of compilation and then describes the Inoxis language and interpreter project. The Inoxis language implements the memory safety features of Rust while maintaining C++ syntax. The interpreter utilizes ANTLR to generate the parser from the Inoxis grammar with a lexer of my own making. The parse tree is walked three times, creating a symbol table, checking for memory safety violations, and creating the intermediate representation that is the input to the back end. The back end is implemented using a stack-based virtual machine.

Key Word List

Compiler, interpreter, virtual machine, Rust, C, C++, memory safety, ANTLR, lexing, parsing, Inoxis, programming languages, intermediate representation

Table of Contents

1. Introduction

In this introduction, I will first survey the field of compilers and interpreters. Then, I will provide an overview of the current problems in the field and discuss the strengths and weaknesses of proposed solutions. Lastly, I will outline my project, which provides 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 the 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 implemented using macros in Rust instead. I have created a new language called Inoxis, which has Rust's memory safety features while using syntax much closer to C++.

In order to implement the language, I have built a compiler for it. Of the diverse ways to build a compiler, I chose to write an interpreter in C and C++, which utilizes a virtual machine for the back end. In other words, the interpreter does not target any real CPU, but a virtual one whose instruction set I developed myself. This ensured that my project is unique within the field.

2. Objectives

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

3. Project Environment

Development Environment

Hardware

I used 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 used Microsoft Visual Studio as my IDE, the C and C++ language and standard libraries, GLib for extended C data structures, the Microsoft C/C++ Compiler, and ANTLR as the parser generator.

Operational Environment

The program runs in Windows 11.

4. Project Results

Project Deliverables

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

Quality Measures

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

5. Requirements Description

The program's external dependencies are:

- ANTLR4 – which requires:
    - Java 8
    - JDK 23
    - Python 3
- GLib (I used vcpkg to install it)
- Visual Studio 2022
- Microsoft Visual C/C++ compiler and runtime libraries

6. Design Description

The Inoxis Language

- Inoxis is statically typed.
- It has one data type, the integer type.
- Data can be allocated from the heap using the C++ style “new” keyword.
- Once the owner of a heap allocation goes out of scope, its memory is freed automatically.
- It has arrays that can be allocated from the stack or the heap.
- Arrays cannot use variable sizes when declared.
- It has mutable and immutable data types.
- Immutable variables cannot change their data.
- Variables are immutable by default.
- Each value has one owner.
- Inoxis has pointers and references.
- References have their own space in memory and function as pointers.
- References allow values to be borrowed.
- References can be either mutable or immutable.
- Mutable References can change the data they borrow.
- Immutable references can only read and own the data they borrow.
- Each function has one integer parameter.
- Each function must return an integer.
- It has If/elif/else block and while loops.
- Variables cannot be declared inside control flow blocks.
- There can be no return statements inside control flow blocks.
- It can print string literals and variables using C++ style cout <<.

The Inoxis Interpreter

- The interpreter runs on Windows 11.
- The front end is written in C++, and the virtual machine backend is in C.
- It interprets all Inoxis programs correctly and executes them, either generating one or more errors or the correct outputs.

- It interprets all Inoxis programs in a reasonable amount of time.
- The parser is generated by ANTLR from the Inoxis grammar file I designed.
- The interpreter first reads an Inoxis source file from the command line.
- The input is then passed to the lexer, which tokenizes the input and passes the tokens to the generated ANTLR parser.
- The parser generates an abstract syntax tree (AST).
- The interpreter then walks the AST three times:
    - The first pass creates a symbol table and then validates the symbols, checking for errors
    - The second pass checks for memory safety violations and outputs any errors.
    - The third pass creates the input for the virtual machine, stored as a GArray of function structs. These structs contain arrays of statement structs as well as a memory array allocated for the local variables.

The Virtual Machine

The virtual machine then compiles all of the statements for each function into a list of instructions. It then executes the program, looping through and executing instructions until the end of the program is reached. The VM utilizes a data stack, a local memory array for each function, and a hash table of jump labels.

The instruction set is:

Instruction Set

- MOVE_I(index) – move a value from one location to another. It has seven types:
    - MOV_TO_STACK - gets the value at memory INDEX and pushes it onto the data stack.
    - MOV_TO_MEM - pops the value off the data stack and moves it to the memory INDEX.
    - MOV_FROM_STACK_INDEX - pops the memory index off the data stack, gets the value there, and pushes it onto the data stack.
    - MOV_TO_STACK_INDEX - pops the memory index off the data stack, pops the data stack again, and moves that value to the memory index.
    - MOV_TO_HEAP - pops the value off the data stack and moves it to the address stored in memory INDEX.
    - MOV_FROM_HEAP - gets the value stored at the address in memory INDEX and pushes it onto the stack.
    - MOV_STACK_TO_HEAP - pops the heap address off the data stack, pops the data stack again, and moves that value to that address.

- STORE_I(val) – pushes a value onto the data stack.
- ADD_I – add two values and push the result onto the data stack.
- SUBTRACT_I - subtract two values and push the result onto the data stack.
- Conditional operator instructions that compare the top two values on the stack and push a zero for true and a one for false onto the stack:
    - LESS_I
    - LESS_EQUAL_I
    - GREATER_I
    - GREATER_EQUAL_I
    - DOUBLE_EQUALS_I
    - NOT_EQUAL_I
- NOT_I – pops the Boolean value off the stack, reverses it, and pushes it back onto the stack.
- CALL_I(label) – pushes the argument onto the stack, finds the jump label for the function call, and sets the program counter to that location.
- RETURN_I – pops the program counter off the stack, sets it, and pushes the return value onto the stack.
- PRINT_I(num) – pops num values off the stack and prints them.
- JUMP_I (label) – looks up the label and sets the program counter to that location.
- JUMP_NOT_ZERO_I(label) – lookups the label, pops the stack, if top = 0, do nothing, if it's 1, set the program counter to the label location.
- ALLOCATE_I(index, size) – allocates SIZE number of integers using malloc, then sets memory INDEX to the returned pointer.
- FREE_I(index) – calls free with the pointer stored in memory INDEX as the argument.
- REF_I(index) – If the memory value at INDEX is a pointer, this instruction creates a new pointer and sets it to point to the same address. Otherwise, it pushes INDEX onto the stack.
- SUBSCRIPT_I(index) - given a memory index for the first element of the array, pop the stack to get the array index, then, if the memory value holds an int, add INDEX to the popped index. If it's an unsigned or pointer value, add the popped value to that value. Then push the resulting value onto the stack.

7. Implementation

ANTLR uses the grammar in Inoxis.g4 and lexRules.g4 to generate the parser and the files stored in the .antlr folder. The interpreter takes the name of an Inoxis program file as a command-line argument. Main.cpp then opens the file and creates an ANTLRInputStream object with it. This object is then used to initialize an Interpreter object. That code is contained in Interpreter.h/cpp. Interpreter.cpp calls the run() method, which first passes the input to myLexer.h/cpp, which tokenizes the input. Back in run(), the tokens are used to create an ANTLR common token stream object, which is then passed to the generated ANTLR parser. The parser is then used to create an ANTLR ParseTree object, an abstract syntax tree, which is then walked three times if no errors are encountered.

The first walk creates a symbol table, made up of funcSymbol and varSymbol objects, and validates the symbols. That code is contained in symbolTable.h/cpp, funcSymbol.h, and varSymbol.h/cpp. The second walk checks for memory safety violations in the program. The code is contained in MemSafetyPass.h/cpp. Finally, the last walk, contained in VMInputPass.h/cpp, generates the input that is passed to the virtual machine. The input is a GArray of function structs. Function structs contain a GArray of statement structs as well as a GArray of memVal structs, which serve as local memory for the function. These data structures and helper methods, written in C, are stored in VMInput.h/c.

Back in Interpreter.run(), the VMInput is then passed to the VMMain function, the main diver function for the virtual machine that's written in C. That code is contained in VirtualMachine.h/c. Then, the input is compiled into a GArray of instructions. A hash table of jump labels is also filled during this phase. Then, the instructions are executed until the program terminates. The instruction execution code as well as the data structures are contained in VMInstruciton.h/c.

File Structure

The Visual Studio Solution is split between two Projects, the antlr4cpp-vs22 source files from ANTLR and Interpreter, which contains the actual interpreter program. Interpreter contains a folder ".antlr" which contains the files generated by the ANTLR parser. Interpreter contains the following files:

- Inoxis.g4 – Contains the ANTLR grammar parser rules for the Inoxis language.
- lexRules.g4 – The ANTLR grammar lexer rules for Inoxis. Since I wrote my own lexer, these rules were never invoked in the rest of my program.
- main.cpp – Gets the Inoxis input file from the command line, creates an ANTLR input stream object with it, and then passes this to Interpreter().

- Interpreter.h – The class definition for the Intepreter class.
- Interpreter.cpp – The member function definitions for the Interpreter class. Interpreter.run() is the main driver for the whole program.
- myLexer.h – The class definition for the myLexer class. It inherits from the antlr4 TokenSource class.
- myLexer.cpp – The member function definitions for the myLexer class. The main function here is the overridden nextToken() function. This lexes the next token from the input stream and returns it to the parser.
- symbolTable.h – The class definition for the symbol table class, which inherits from the InoxisBaseListener class. This is one of the parse tree walker classes. It creates a symbol table of funcSymbol objects and validates the symbols.
- symbolTable.cpp – The member function definitions for the symbolTable class. Most of these are overridden functions from InoxisBaseListener. It includes an enterRule and exitRule function for each parser rule in the Inoxis grammar. They are called when rules are entered and exited while walking the AST.
- varSymbol.h – The class definition for varSymbol. It represents a local variable in a function, containing all of its information.
- varSymbol.cpp – The member function definitions for varSymbol. These methods mainly change memory permissions for the variable.
- funcSymbol.h – The class definition for the funcSymbol class. It represents a function definition and includes a hash table of local variables as varSymbol objects.
- dataType.h – An enum representing the three different data types: pointer, reference, and regular integer.
- MemSafetyPass.h – The class definition for the MemSafetyPass class. Since it also walks the AST, it inherits from the InoxisBaseListener class. As it walks the AST, it sets memory permissions for varSymbol objects and also checks that they have the required permissions for the operation.
- MemSafetyPass.cpp – The member function definitions for MemSafetyPass. Like the other AST walkers, these are mostly overridden enter and exit rule functions.
- VMInput.h – Contains the C language data structure definitions that will be passed as input to the virtual machine. It also contains helper function declarations for manipulating the data structures.
- VMInput.c – Contains the function definitions for the helper functions in VMInput.h. These initialize, free, and print the struct objects.

- VMInputPass.h – The class definition for the VMInputPass class, another AST walker that inherits from InoxisBaseListener. Walks the AST and creates the virtual machine input defined in VMInput.h.
- VMInputPass.cpp – Member function definitions for VMInputPass.
- VirtualMachine.h – The function declarations and data structures that run the virtual machine backend. The main functions are VMMain, which is the main driver, compile, and execute.
- VirtualMachine.c – The function definitions for VirtualMachine.h.
- VMInstruction.h – The data structures for the VM instruction set, as well as function declarations.
- VMInstruction.c – The function definitions for the VM instructions. Each instruction has an execute_() function as well as initialization and print methods.

The .antlr folder contains the following files that were utilized:

- Inoxis.tokens – Contains a list of all tokens for Inoxis.
- InoxisParser.h – Contains the parser class definitions. Each grammar rule has its own class.
- InoxisParser.cpp – The generated class method code for the parser.
- InoxisBaseListener.h – The InoxisBaseListener class. It has an enter and exit method for each grammar rule that can be overloaded by the user.
- InoxisBaseListener.cpp – Empty file.

It also contains the following files, which were generated but never utilized or referenced:

- InoxisBaseVisitor.h – This class is similar to InoxisBaseListener; visitors don't require a parse tree walker, however.
- InoxisBaseVisitor.cpp – Empty file.
- InoxisLexer.h – The generated ANTLR lexer.
- InoxisLexer.cpp – Function definitions for the lexer.
- InoxisLexer.interp – Same as Inoxis.interp but for the lexer grammars.
- InoxisLexer.tokens - Same as Inoxis.tokens, but for the lexer tokens.
- InoxisListener.h – Similar to InoxisBaseListener. However, classes that inherit from this must overload every function, which can be impractical.
- InoxisListener.cpp – Empty file.
- InoxisVisitor.cpp – Empty file.
- InoxisVisitor.h – Same as InoxisListener.h but for visitor methods.
- Inoxis.interp – This file contains information that helps debug ANTLR parser grammar files.

The interpreter also utilizes and links to the GLib library. Those files are not included in the Visual Studio solution.

8. Testing and Integration

Compiler Testing History

Testing a compiler is an imposing task, since the number of possible inputs is immense. The main options for making compiler tests are manually writing a test suite of input programs or using the language grammar to generate test programs (Chen et al., 2021). Some older compilers, like GCC, ship with large, manually written test suites that have been developed over the course of decades by a whole community of users. However, the manual approach has fallen out of favor in the research community in the past decade (Chen et al., 2021).

Further, understanding when test cases generate bugs, also known as a test oracle, is difficult when dealing with compilers since the toolchain has many phases. Bad input will be detected early in the program, and the compiler will not advance to the next phase. A common way to deal with this is to test the inputs with another compiler built with similar specifications. This method is referred to as cross compilation (Chen et al., 2021).

My testing methodology

My testing approach has been to manually create a suite of Inoxis test programs. I was able to write over fifty such programs, along with variations of them.

I have chosen twelve of the most paradigmatic test programs to include in the report, showcasing the main features of the language and the interpreter. They are displayed below, side by side with the output that they generate.

Testing Results

Inoxis Program	Output

1. MutableReference-Rejected.txt
 	
int main()
{
// create a new mutable int pointer
int mut *var1 = new int;
 	
// set its data
*var1 = 10;
 	
// var2 make a mutable reference of var1, meaning it can change var1's data
// var1 loses all permissions, var2 gains read/write/ownership permissions
int &var2 = & mut var1;
 	
// LINE 16: attempt to read var1 before var2 goes out of scope, unsafe, var1 might not have the data expected at runtime.
cout << "attempt to read var1" << *var1 << endl;
 	
// var1 now goes out of scope and is dropped
 	
// print var2's data
cout << "*var2 = " << *var2 << endl;
 	
// var2 goes out of scope and is dropped, var1's permissions are restored
// the heap data var1 and var2 were pointing to is freed automatically after its owner (var2) is dropped
 	
return 0;
}
	Line 16: var1 does not have required Read permissions

2. Mutable reference-accepted.txt

```
int main()
{
	// create a new mutable int pointer
	int mut *var1 = new int;

	// set its data
	*var1 = 10;

	// var2 make a mutable reference of var1, meaning it can change var1's data
	// var1 loses all permissions, var2 gains read/write/ownership permissions
	int &var2 = & mut var1;

	// var1 now goes out of scope and is dropped

	// print var2's data
	cout << "*var2 = " << *var2 << endl;

	// var2 goes out of scope and is dropped, var1's permissions are restored
	// the heap data var1 and var2 were pointing to is freed automatically after its owner (var2) is dropped

	return 0;
}
```

```
*var2 = 10
```

3. while loops and arrays.txt

```
int main()
{
	int mut* var1 = new int[3];

	int mut index = 0;

	while(index < 3)
	{
		var1[index] = index;

		index = index + 1;
	}

	index = 0;
```

```
var[0] = 0
var[1] = 1
var[2] = 2
```

`while(index < 3)`
`{`
`cout << "var[" << index << "] = "`
`<< var1[index] << endl;`
 	
`index = index + 1;`
`}`
 	
`return 0;`
`}`	

4. if-else.txt
 	
`int main()`
`{`
`int mut var1 = 10;`
 	
`if(var1 == 25)`
`{`
`var1 = 1;`
`}`
 	
`elif(var1 == 11)`
`{`
`var1 = 2;`
`}`
 	
`else`
`{`
`var1 = 3;`
`}`
 	
`cout << "var1 = " << var1 << endl;`
 	
`if(!var1 > 2)`
`{`
`cout << "in second if block" <<`
`endl;`
`}`
 	
`else`
`{`
`cout << "in second else block"`
`<< endl;`
`}`
 	
`return 0;`
`}`
	var1 = 3
	in second else block

5. function calls.txt
 	
int func1(int param);
 	
int func2(int param);
 	
int main()
{
    int mut arg = 25;
 	
    int mut retVal = func1(arg);
 	
    cout << "first return value is " << retVal << endl;
 	
    retVal = 10 - func2(5);
 	
    cout << "second return value is " << retVal << endl;
 	
    return 0;
}
 	
int func1(int param)
{
    int var2 = param + 25;
 	
    return var2;
}
 	
int func2(int param)
{
    int mut *retVal = new int;
 	
    *retVal = param + 10;
 	
    return *retVal;
}
	first return value is 50
	second return value is -5

6. using dropped pointer-rejected.txt
 	
int main()
{
    int mut* var1 = new int;
 	
    *var1 = 8;
	Line 12: var1 does not have required Read permissions

int mut *var2 = var1;
 	
// line 12
cout << "try to print var1 " << *var1 << endl;
 	
return 0;
}	

<u>7. pointer assignment.txt</u>
 	
int main()
{
int mut* var1 = new int;
 	
*var1 = 8;
 	
int mut *var2 = var1;
 	
cout << "var2 = " << *var2 << endl;
 	
return 0;
}
	var2 = 8

<u>8. regain permissions after borrow.txt</u>
 	
int main()
{
// create a new mutable int pointer
int mut *var1 = new int;
 	
// set its data
*var1 = 10;
 	
// var2 make a mutable reference of var1, meaning it can change var1's data
// var1 loses all permissions, var2 gains read/write/ownership permissions
int &var2 = & mut var1;
 	
cout << "*var2 = " << *var2 << endl;
 	
// var2 goes out of scope, var1 regains permissions
cout << "var1 has regained permissions and is now: " << *var1 << endl;
 	
return 0;
}
	*var2 = 10
	var1 has regained permissions and is now: 10

9. immutable write – rejected.txt

```
int main()
{
        int var1;

        var1 = 5;

        return 0;
}
```

Line 6: var1 does not have write permissions

10. borrowee attempts to write – rejected.txt

```
int main()
{
        // create a new mutable int pointer
        int mut *var1 = new int;

        // set its data
        *var1 = 10;

        // immutable reference, var2 can't
change data
        int &var2 = & var1;

        // LINE 14: var1 attempts a write
        *var1 = 5;

        // print var2's data
        cout << "*var2 = " << *var2 << endl;

        // var2 goes out of scope and is
dropped, var1's permissions are restored
        // the heap data var1 and var2 were
pointing to is freed automatically after its
owner (var2) is dropped

        return 0;
}
```

Line 14: var1 does not have write permissions

11. symbol table errors – rejected.txt

```
int  func1(int stuff);

int main()
{
```

Line 9: variable name conflict with: var1
Line 11: Variable: var2 has not been declared
Line 16: func1 has NOT been declared

int var1;
 	
    // line 9
    int var1 = 10;
 	
    // line 11
    var2 = 10;
 	
    return 0;
}
 	
// line 16
int func1(int param)
{
    return 0;
}	

12. grammar errors – rejected.txt
 	
int main()
{
    int var1
 	
    // line 6
    bool var2;
 	
    else
    {
        cout << endl;
    } // line 11
 	
    if var1 < 10
    {
        cout << endl;
    }
 	
    // line 13
    if var1 < 20
        cout << endl;
 	
    alsdkjfl kadjf
 	
}
 	
int func1(param);
	line 6:1 missing ';' at 'bool'
	line 11:1 mismatched input '}' expecting 'return'
	line 13:1 mismatched input 'if' expecting <EOF>

9. Installation Instructions

The source code, along with the Visual Studio solution and project files, is stored on my GitHub at https://github.com/mbclause/Inoxis-Interpreter. Git can be used to clone the repository. This also includes the ANTLR C++ runtime library and the generated Inoxis files.

What is not included is GLib, which needs to be cloned from https://gitlab.gnome.org/GNOME/glib. I used vcpkg to install it to allow better integration with Visual Studio. Since I use a static build for my Visual Studio solution, the static version of GLib needs to be installed.

10. Operating Instructions

The exe file, Inoxis-Interpreter.exe, is contained in the exe-files folder in the repository. To run it, open a command prompt and navigate to the directory where the program is located. Then, type: Inoxis-Interpreter.exe <input-file-name.txt>. Make sure the input file is in the same directory as the exe. The other file in the folder, “iconv-2.dll,” must be in the same directory as well.

11. Recommendations for Enhancement

Since Inoxis is a fairly limited programming language, there are plenty of avenues for improvement. There are also numerous ways of enhancing the interpreter as well.

Language Enhancements

Limitations

Here are the most glaring limitations that should be resolved:

- Inside control blocks (if/else/while, etc.), variables cannot be declared, and functions cannot return.
- Variable names cannot include the names of other variables. For example, you cannot have a variable named “var” as well as a variable named “var1”. This is due to the algorithms for determining when a variable goes out of scope, which utilize string searches for variable names in statements.
- Pointers, references, and arrays cannot be passed as function arguments or returned by functions.
- When an array is declared, a variable cannot be used for the array size.

Recommended Improvements

These are the next improvements that should be made:

- All of the standard data types should be added, such as float, double, string, char, bool, etc.
- Allow functions to have a void return value.
- Allow functions to have more than one parameter or zero.
- Add the other arithmetic operators, multiplication, division, and modulus. Also, add parentheses for arithmetic expressions.

Interpreter Enhancements

Similarly, these are the first improvements that should be added to the interpreter:

- The most important would be to add at least one optimization pass. A peephole pass and loop optimization would be the most obvious.
- Adding a cleanup phase to the virtual machine that frees all allocated memory, instead of inserting calls to free() in the instruction list. Currently, there are issues

with this implementation. While loops can cause the free instruction to be inserted after the return instruction, causing it not to be executed, and if a heap-owning variable is part of a return value, free will not be called in time either.

- As stated above, using another method besides string search for determining when variables go out of scope would get rid of the variable name restrictions. Utilizing the ANTLR parse tree properties to pass variable names up the tree would work.
- The speed of interpretation would be greatly improved by using faster algorithms for determining when variables go out of scope. Currently, a brute force method is used. Adding another parse tree walk before the memory safety pass that creates a hash table for variable names and the line numbers they are used in could work. This would definitely have better time complexity than the current approach.
- In arithmetic expressions where an addition operation follows subtraction, the sub-expressions are evaluated in the wrong order (e.g., in the expression 55 – 2 + 1, 2+1 is evaluated first, then 55 – that result, for the incorrect answer of 52). This is because the input to the VM stores the expression using an in-order binary tree traversal, while the VM evaluates the expressions using a post-order traversal, since it's stack-based. Some sort of binary tree reconfiguration is required to solve this problem. Unfortunately, I caught this problem too late to attempt such a large change in the code.

I have also considered building another version of the interpreter using LLVM for the back end, which would probably be faster but would sacrifice portability.

12. Bibliography

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.

C and C++ reference. cppreference.com. (n.d.). https://en.cppreference.com/w/

Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., & Zhang, L. (2021). A Survey of Compiler Testing. ACM Computing Surveys, 53(1), 1–36. https://doi.org/10.1145/3363562

Cooper, K., & Torczon, L. (2012). Engineering a compiler (2nd ed.). Elsevier.

Crichton, W., Gray, G., & Krishnamurthi, S. (2023). A Grounded Conceptual Model for Ownership Types in Rust. *Proceedings of ACM on Programming Languages*, *7*(OOPSLA2), 1224–1252. https://doi.org/10.1145/3622841

Crichton, W., Gray, G., & Krishnamurthi, S. (n.d.). *The rust programming language*. Understanding Ownership - The Rust Programming Language. https://rust-book.cs.brown.edu/ch04-00-understanding-ownership.html

Eaton, P. (2021, December 28). Writing a minimal Lua implementation with a virtual machine from scratch in Rust. Writing a minimal Lua implementation with a virtual machine from scratch in rust. https://notes.eatonphil.com/lua-in-rust.html

GeeksforGeeks. (n.d.). https://www.geeksforgeeks.org/

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.

Parr, T. (2013). *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Programmers, LLC, The.

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

Sheridan, F. (2007). Practical testing of a C99 compiler using output comparison. *Software, Practice & Experience*, *37*(14), 1475–1488. https://doi.org/10.1002/spe.812

Stack Overflow. (n.d.). https://stackoverflow.com/questions

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