

Rogue Programming Language

Mujeeb Asabi Adelekan

madelekan@mail.stmarytx.edu

CS6395 Comprehensive Project

Summer 2023

Contents

Overview	3
Problem/Issue	3
Background	3
Existing Products	5
The Proposed Product	7
Requirements.....	9
Functional Requirements.....	9
User Hardware and Software.....	10
Developer Hardware and Software	10
Design.....	12
Data Structures	12
Software Modules.....	18
Interface Design	18
BNF Grammar for Rogue.....	22
Pointers	27
Enumeration Types	28
Associative Arrays	29
File Handling.....	31
Plan	34
Tasks.....	34
Schedule.....	35
Cost	36
Marketing.....	37
References	38

Overview

This document is a proposal for the Rogue Programming Language. It will start with an overview of the Rogue product, and will outline the requirements, design, plan, and marketing of Rogue.

This section will start with a discussion on how programming languages greatly aid software development and how a common hurdle of these languages often slows development. This is followed by a brief history of high-level programming languages and their development. Existing solutions to the problem statement are discussed, followed by a detailed description of the Rogue language and its intended users.

After the overview of the product, this section will end with a preview of the remaining content of this proposal.

Problem/Issue

The problem that prompted the creation of Rogue is discussed in this subsection.

High-level programming languages enable us to speak to computers by allowing programmers to write instructions using a higher level of abstraction. An important concept that permits abstraction is the ability to break programs into smaller, comprehensive modules (i.e., loops, subprograms, and class declarations). However, most languages use syntax for group modules that may prove confusing for larger programs. For example, it becomes difficult to know which closing curly brace is tied to which opening curly brace in C++ and C#, and it becomes even harder to delimit modules in indentation-based languages like Python. Although most IDEs work to alleviate this issue, it would be helpful if the syntax of a language itself aided the programmer, thus improving the writability of the high-level language. This is the mission of the Rogue programming language.

Background

Here is an in-depth overview on the history of programming languages.

In 1945, German scientist Konrad Zuse conceptualized a way to express computations for his Z4 computer. Named *Plankakul*, Zuse's language may be the first description of a high-level programming language. Features of the language included single-bit, integer, and floating-point datatypes, the ability to declare arrays and define records (like C structs), an iterative "for" statement, a selection statement, and assertions. Zuse used his language to write algorithms for sorting, graph connectivity testing, numerical operations like the square root, syntax analysis, and even chess playing. However, despite the depth of Zuse's language project, *Plankakul* was developed in isolation, was never implemented, and its description was not published until 1972, nearly three decades after its conception.

Although *Plankakul* could not receive recognition in the computer science world at the time, there were many attempts to implement a way to write programs at a higher-level than machine code. These "pseudocodes", a term which in this era refers to code that is marginally more abstract than machine code, are a stepping stone in the creation of the high-level

languages we are familiar with. Examples include Short Code, a pseudocode developed by John Mauchly for the BINAC computer, which represented math expressions as a pair of 6-bit bytes, and the UNIVAC compiling system, a series of compilers spearheaded by Grace Hopper, which allowed for higher-level code to be expanded into machine code, much like macro expansion in assembly.

The introduction of the IBM 704 in the mid-1950s, which included floating-point and index instructions in hardware, prompted the development of the Fortran programming language. The developers of Fortran claimed that the language will combine the efficiency of hard-coding with the ease of interpretive pseudocode. The original 1957 release of Fortran allowed input and output formatting as well as subroutines. The language also included an if-statement and a do-statement to modify the flow of control. Unlike most modern languages, data types in Fortran were implicitly determined by the variable name. Fortran proved popular among IBM 704 programmers, with the language having an adoption rate of around 50% a year after release. Fortran would see many updates, with its second release introducing independent subroutine compilation, removing the need to recompile an entire program, and thus making larger programs more practical.

Fortran's relationship with the IBM 704 was typical of the era. Most programming languages in the late 1950's were machine-dependent, which complicated program sharing. In 1958, American and European developers worked on a joint project to create a machine-independent programming language. This language, ALGOL, generalized Fortran's features to make a more flexible language that the developers hoped would be used to describe algorithms. ALGOL formalized the concept of explicit data typing, introduced block structure, or scoping, allowed subprogram parameters to be passed-by-name or value, allowed recursion, and had stack-dynamic arrays. ALGOL was also the first programming language with formally described syntax, as Backus-Naur form, or BNF, was created to describe the language during its design process.

Up until the late 1960s, the most popular programming languages were purely procedural. That is, the main building blocks of these languages were the statements, and eventually the subroutines. At the turn of the decade, the concept of object-oriented programming began to coalesce. In 1967, the SIMULA 67 language introduced the class construct, and thus the concept of data abstraction, to support simulation applications. Two years later, Alan Kay, whose work was inspired by SIMULA 67, believed that desktop computers will primarily be used by nonprogrammers in the near future. As such, Kay argued, those computers will need to offer an interactive, intuitive user interface.

Kay created Dynabook as his idea for a graphical user interface. Dynabook simulates a desk that has sheets of paper, represented by windows, with the top sheet being prioritized, and the user interacts through keystrokes and the touchscreen. To support and implement Dynabook, Kay developed the SmallTalk programming language. SmallTalk consists entirely of objects, and the language uses an unconventional, message-based system in which messages are sent to an object so that the object's methods are executed. SmallTalk's support of the object-oriented

paradigm and its promotion of graphical user interface design established a link between the former and the latter that persists to this day.

These are only a few examples of how programming languages evolved and the motivations behind their development. Many more languages have been developed for use in artificial intelligence (Lisp), business applications (COBOL), education (Pascal), national defense (Ada), and even music composition!

Existing Products

Here is a discussion of three languages developed to address issues that their designers felt were not being addressed by popular languages of the time.

Lisp

Lisp is the first functional programming language. Designed by John McCarthy and Marvin Minsky, Lisp supports linked list processing, which was a feature needed for early AI applications. In particular, Lisp introduced recursion, conditional expressions, dynamic storage allocation, and implicit deallocation to list processing.

As a functional language, all computations in Lisp are done by applying functions to arguments, removing the need for variables and assignment statements. Loops are unnecessary as well since repetition can be defined with recursion.

Lisp remains a popular language in AI applications. Two popular dialects of Lisp exist today: Scheme, a smaller version of Lisp designed by MIT that is tailored for teaching functional programming, and Common Lisp, a complex amalgamation of various Lisp dialects.

Below is an example Lisp Program in Robert Sebesta's *Concepts of Programming Languages* (11th edition).

```
; Lisp Example function
; The following code defines a Lisp predicate function
; that takes two lists as arguments and returns True
; if the two lists are equal, and NIL (false) otherwise
(DEFUN equal_lists (lis1 lis2)
  (COND
    ((ATOM lis1) (EQ lis1 lis2))
    ((ATOM lis2) NIL)
    ((equal_lists (CAR lis1) (CAR lis2))
     (equal_lists (CDR lis1) (CDR lis2)))
    (T NIL)
  )
)
```

Pascal

Pascal is a programming language designed to teach programming. Designed by Niklaus Wirth, it is based on Wirth's efforts on ALGOL's development team to create an update to ALGOL 60. The update Wirth helped create, later named ALGOL-W, was rejected for being too modest of an improvement. After Wirth left the ALGOL project, he created Pascal as a spiritual successor to ALGOL-W.

Pascal included many features of the nascent programming languages of its time, like user-defined data types, records, and the case selection statement. However, Pascal was simple by design to meet its educational purposes. This simplicity prompted the creation of various dialects, like Turbo Pascal and Object Pascal.

Pascal proved successful in its goal, as it was the most popular language used to teach programming from the mid-1970s to the late 1990s.

Below is an example Pascal program from Robert Sebesta's *Concepts of Programming Languages (11th edition)*, slightly altered and written in an online compiler.

```

1 { Pascal Example Program
2 Input:  An integer, listlen, where listlen is less than
3        100, followed by listlen-integer values
4 Output: The number of input values that are greater than
5        the average of all input values }
6
7 program pasesx (input, output);
8   type intlisttype = array [1..99] of integer;
9   var
10      intlist : intlisttype;
11      listlen, counter, sum, average, result : integer;
12 begin
13   result := 0;
14   sum := 0;
15   readln (listlen);
16   if ( (listlen > 0) and (listlen < 100)) then
17   begin
18 { Read input into an array and compute the sum }
19   for counter := 1 to listlen do
20   begin
21     readln (intlist[counter]);
22     sum := sum + intlist[counter]
23   end;
24 { Compute the average using integer division }
25   average := sum div listlen;
26 { Count the number of input values that are > average }
27   for counter := 1 to listlen do
28   if (intlist[counter] > average) then
29     result := result + 1;
30 { Print the result }
31   writeln ('The number of values > average is: ', result)
32   end { of the then clause of if ( (listlen > 0 ... }
33   else
34     writeln ('Error-input list length is not legal')
35 end.

```

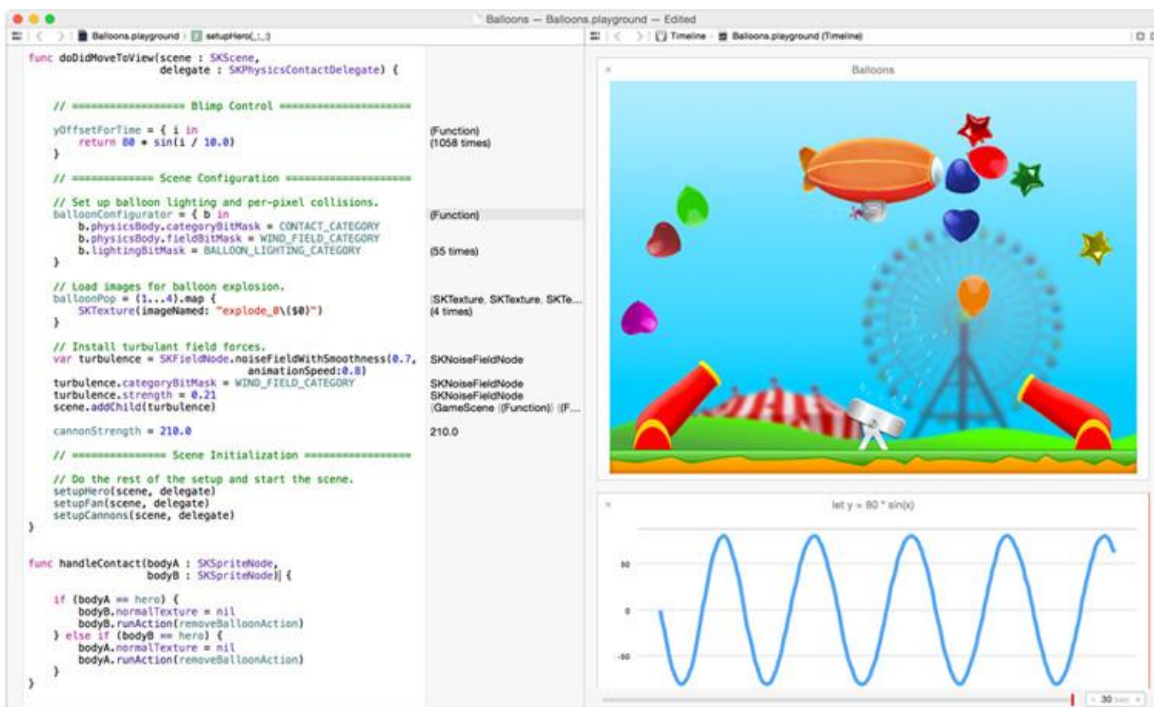
Swift

Swift is a programming language developed by Apple as a replacement for Objective-C, Apple's previous programming language. Swift, like Pascal, attempts to combine various elements from other programming languages into an easy-to-learn language.

Apple intends for Swift to have a lightweight syntax tailored for modern applications. One instance of this modernity is Swift's ability to support multiple languages and emoji by having its strings use UTF-8 encoding. In addition, Swift includes many safeguards against unsafe code. In Swift, memory is automatically managed, and Swift objects can never be null (`nil` in Swift) unless the object is explicitly declared as an "optional".

Swift is interoperable with C++ and Objective-C, and the language is open-source, with a development community at <https://developer.apple.com/swift/>.

Below is example Swift code written in an Apple IDE, alongside "playgrounds" that show results as code is written (Image source: <https://www.pcmag.com/news/apples-swift-language-a-really-big-deal.>)



The Proposed Product

To address the problems of popular programming languages, the author has developed Rogue.

Rogue is a high-level, free-format programming language that prioritizes readability, writability, and reliability. Rogue enables the programmer to see exactly what kind of module is ending. For example, Rogue's main function ends with the keywords "end main", for-loops end with the keywords "end for", if-else branches end with the keywords "end if", and so on. Rogue supports several structural flow-of-control statements, integer and Boolean datatypes, assertions, input statements, arithmetic and logical computations, and the ability to write subprogram modules such as procedures and functions.

The heartbeat of Rogue is the Rogue Compiler, which translates the code into a proprietary assembly language designed to run on STM, a virtual machine (VM) developed by Dr. Hanna

that simulates the von Neumann architecture. Using the Rogue Compiler, programmers can write the logic of an application in the Rogue language, run the Rogue source file through the compiler, see logged information about the compilation, including any syntax errors, and then execute the compiled source file as an application process.

Rogue is designed to be used by application programmers, regardless of their level of experience. Rogue will also support associative arrays, pointer variables, file input and output features, enumerations, and four main data types (int, bool, char, and float). These functionalities will make Rogue suitable for traditional software development and will allow the programmer to construct commonly used data structures like stacks, queues, linked lists, trees, and networks.

The remainder of this document discusses Rogue's functional and non-functional requirements, the design of Rogue's data structures, software modules, and interfaces, and Rogue's development and marketing plans.

Requirements

This section lists the functional and non-functional requirements of the Rogue Compiler and STM.

Functional Requirements

Here is a list of the functional requirements that must be met in this project.

This project is a continuation of my CS6375 Compilers project, which was extended on for CS6340 Advanced Software Engineering. Rogue in its current state has the following features, as supported by the Rogue Compiler:

- Flow-of-control statements
 - Loops
 - While/Do-while loops
 - Mid-test do-while loops
 - For loops
 - Selection statements (If—else-if—else)
- Single-line and multi-line comments
- Print and input statements
- Variables and Constants
 - Integer, Boolean, Character, and Floating-point datatypes
- Assignment statements
- Arithmetic and Logical Expressions
 - Arithmetic Operators:
 - + (add), - (subtract), * (multiply), / (divide), % (modulo)
 - ^ ** (exponentiation)
 - abs (absolute value)
 - + - (positive and negative symbols)
 - ++ -- (preincrement and predecrement)
 - Logical Operations
 - or, nor, xor, and, nand, not
 - Comparison Operators
 - <, <=, ==, >, >=, !=
 - <> (greater or less than, i.e., not equal alternative)
- Subprogram Modules
 - Procedures and Functions
- Assertions
- Array declarations
- Pointer variables

For my comprehensive project, the Rogue programmer will be able to write the following **new** Rogue language features in a code editor:

- File creation, input, output, and deletion
- Pointer **dereferences**, enabling the creation of the following data structures:
 - Stacks
 - Queues
 - Linked Lists
 - Trees
 - Networks
- Enumerations
- Associative Arrays

The Rogue Compiler will:

- Identify tokens and parse elements (i.e., analyze the syntax) of the Rogue programming language.
- Perform the compilation of Rogue source code into STM assembly code.
- Raise error messages and quit compilation if a syntax error is encountered.
- Create a listing file annotated with compilation information, including errors and their source line.

User Hardware and Software

Here is the hardware and software that is required to run this project.

Since the Rogue Compiler and STM are written in C++, the user must have a machine that can run a C++ console application and/or compile a C++ program. The vast majority of modern-day desktops and laptops running the most popular operating systems (i.e., Windows, MacOS, and Linux) are more than capable of running these applications.

To operate the Rogue Compiler and STM, the user must have a basic understanding of the English language and should be familiar with computers to the point where they can use a keyboard (or an equivalent means of input), and can launch, operate, and close applications with ease.

Developer Hardware and Software

Here is the hardware and software that will be used to develop the project.

The Rogue Compiler will be developed in C++ in Visual Studio 2019 on a Windows 10 machine with the following hardware specs:

- Processor: AMD Ryzen 3700U with Radeon Vega Mobile GFX 2.30 GHZ

- Memory (RAM): 16 GB

Version control will be maintained using GitHub.

Rogue source programs, used to test and debug the compiler, will be mostly written using Visual Studio Code.

Why the developer chose the hardware and software:

- Windows 10 and Visual Studio 2019 are the operating system and IDE the developer is most familiar with, respectively.
- Preliminary development for the Rogue Compiler was done in C++ using Visual Studio 2019.
- The developer already has a GitHub account and Visual Studio has git implementation.
- Visual Studio Code allows the developer to edit Rogue source programs and view listing files and assembly code in the same editing environment.

Design

This section will discuss the design of the Rogue language. First, the data structures that will be used in the Rogue Compiler and STM Machine are discussed. Next, the compilation process of the Rogue Compiler will be discussed, with a flowchart showing the compilation process and the major software modules of that process. Afterwards, the interfaces of the major modules will be shown with a simple “Hello World”-type program being used for demonstration. Finally, the context-free syntax of the language will be described using a modified version of Backus-Naur Form, and this grammar will display how the functional requirements of Rogue are met. In particular, the new features that are being added to Rogue will be highlighted and explained in detail.

Data Structures

This subsection will discuss the data structures in Rogue Compiler and the STM main memory.

Identifier Table

The Rogue Identifier Table is an array of structs (called IDENTIFIERRECORDs) which contain each identifier’s scope, data type, number of dimensions, identifier type, reference, and lexeme. The identifier table is updated during compilation to keep track of which identifiers are already defined in the current scope and how each identifier is referenced in STM assembly language.

Consider the following Rogue source code:

```

$$-----

$$ Mujeeb Adelekan

$$ Demo #3

$$ Demo3.rog

$$-----

$**** Global variables ****$

int x, y;

$*****$

$$ P(in1, out1, io1, ref1):
$$   in1: input-only parameter (passed-by-value)
$$   out1: output-only parameter (passed-by-result)
$$   io1: input and output parameter (passed-by-result/value)
$$   ref1: passed-by-reference
$*****$

proc P(IN in1 : int, OUT out1 : int, IO io1 : int, REF ref1 : bool)

```

```

$**** Local variable C1 ****$
perm int C1 = 101;

$**** out1 = 1 + 101 ****$
out1 = in1+C1;

$**** io1 = 3 + 1 ****$
io1 = io1+1;

$**** ref1 = NOT false ****$
ref1 = NOT ref1;
end proc

$*****$
$$ Main function
$*****$
main
    $$ declare the constant p1
    perm int p1 = 1;

    $$ declare values p2, p3, and p4
    int p2, p3;
    bool p4;

    $$ assign values to p3 and p4
    p3 = 3;
    p4 = true;

    $$ call the procedure
    call P(p1, p2, p3, p4);

    $$ print p1, p2, p3, p4
    print("p1 = ", p1, "\n");

```

```

    print("p2 = ", p2, "\n");
    print("p3 = ", p3, "\n");
    print("p4 = ", p4, "\n");
end main

```

Upon compilation of the above Rogue program, the identifier table will contain the following information at different points of compilation:

=====

Contents of identifier table after compilation of global data definitions

#	Scope	Data type	Dimensions	Type	Reference	Lexeme
1	0	INTEGER		GLOBAL_VARIABLE	SB:0D0	x
2	0	INTEGER		GLOBAL_VARIABLE	SB:0D1	y

=====

=====

Contents of identifier table after compilation of PROCEDURE module header

#	Scope	Data type	Dimensions	Type	Reference	Lexeme
1	0	INTEGER		GLOBAL_VARIABLE	SB:0D0	x
2	0	INTEGER		GLOBAL_VARIABLE	SB:0D1	y
3	0			PROCEDURE_SUBPROGRAMMODULE	P	P
4	1	INTEGER		IN_PARAMETER	FB:0D0	in1
5	1	INTEGER		OUT_PARAMETER	FB:0D2	out1
6	1	INTEGER		IO_PARAMETER	FB:0D4	io1
7	1	BOOLEAN		REF_PARAMETER	@FB:0D5	ref1

=====

=====

Contents of identifier table after compilation of PROCEDURE local data definitions

#	Scope	Data type	Dimensions	Type	Reference	Lexeme
1	0	INTEGER		GLOBAL_VARIABLE	SB:0D0	x
2	0	INTEGER		GLOBAL_VARIABLE	SB:0D1	y
3	0			PROCEDURE_SUBPROGRAMMODULE	P	P
4	1	INTEGER		IN_PARAMETER	FB:0D0	in1
5	1	INTEGER		OUT_PARAMETER	FB:0D2	out1
6	1	INTEGER		IO_PARAMETER	FB:0D4	io1
7	1	BOOLEAN		REF_PARAMETER	@FB:0D5	ref1
8	1	INTEGER		SUBPROGRAMMODULE_CONSTANT	FB:0D8	C1

Contents of identifier table at end of compilation of PROCEDURE module definition

#	Scope	Data type	Dimensions	Type	Reference	Lexeme
1	0	INTEGER		GLOBAL_VARIABLE	SB:0D0	x
2	0	INTEGER		GLOBAL_VARIABLE	SB:0D1	y
3	0			PROCEDURE_SUBPROGRAMMODULE	P	P
4	1	INTEGER		IN_PARAMETER	FB:0D0	
5	1	INTEGER		OUT_PARAMETER	FB:0D2	
6	1	INTEGER		IO_PARAMETER	FB:0D4	
7	1	BOOLEAN		REF_PARAMETER	@FB:0D5	

Note that when a subprogram scope has been exited by the compiler, all the local variables and constants in that scope are removed from the table. However, the subprogram's formal parameters remain. This is so references to the subprogram can be compiled. **The identifiers are removed from the table to allow them to be used outside of the subprogram's scope.**

Contents of identifier table at end of compilation of PROGRAM module definition

#	Scope	Data type	Dimensions	Type	Reference	Lexeme

1	0	INTEGER		0 GLOBAL_VARIABLE	SB:0D0	x
2	0	INTEGER		0 GLOBAL_VARIABLE	SB:0D1	y
3	0			0 PROCEDURE_SUBPROGRAMMODULE	P	P
4	1	INTEGER		0 IN_PARAMETER	FB:0D0	
5	1	INTEGER		0 OUT_PARAMETER	FB:0D2	
6	1	INTEGER		0 IO_PARAMETER	FB:0D4	
7	1	BOOLEAN		0 REF_PARAMETER	@FB:0D5	
8	1	INTEGER		0 PROGRAMMODULE_CONSTANT	SB:0D30	p1
9	1	INTEGER		0 PROGRAMMODULE_VARIABLE	SB:0D31	p2
10	1	INTEGER		0 PROGRAMMODULE_VARIABLE	SB:0D32	p3
11	1	BOOLEAN		0 PROGRAMMODULE_VARIABLE	SB:0D33	p4
=====						

STM Main Memory

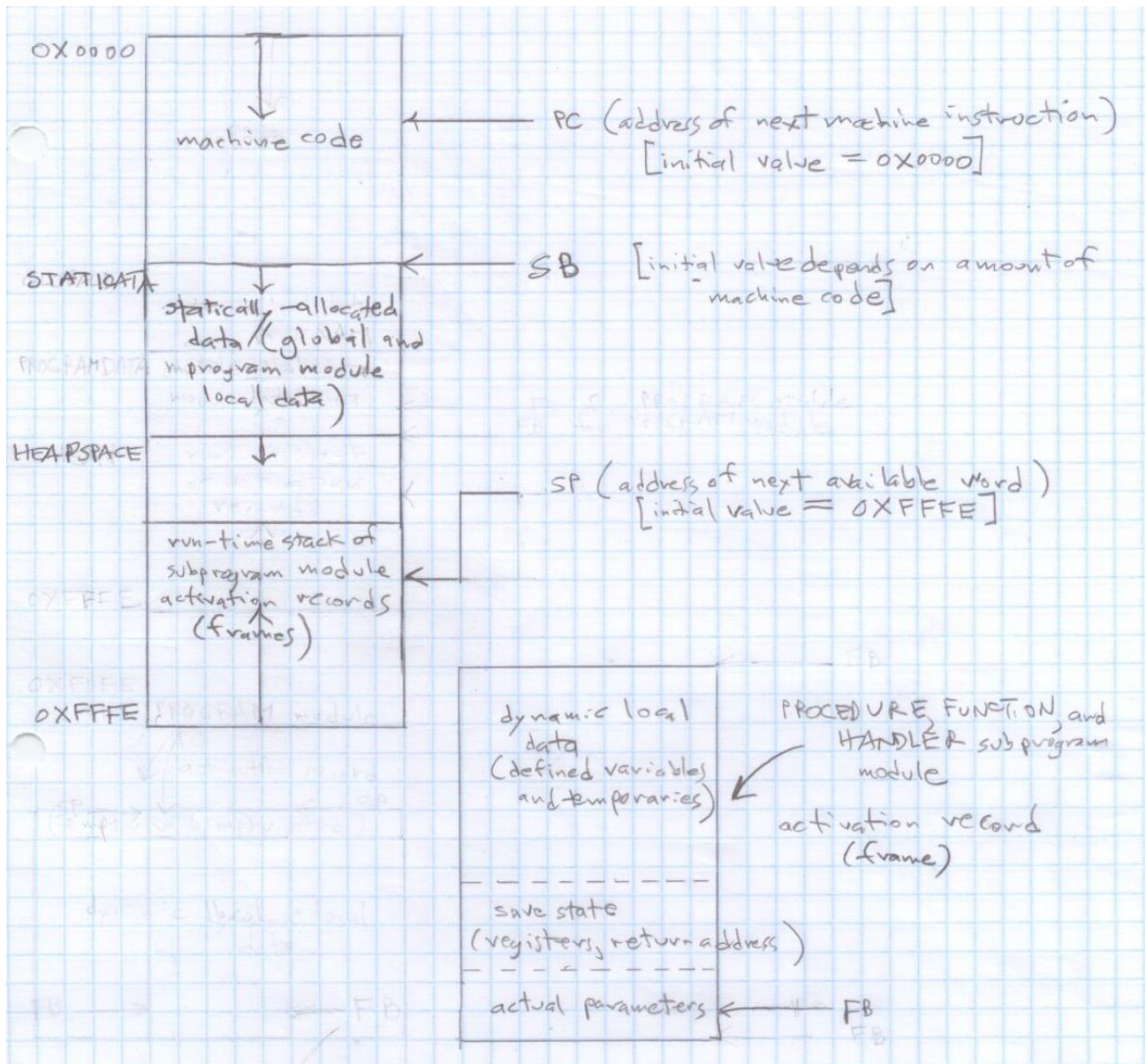
STM uses 16-bit addresses for its main memory (i.e., its address range is [0x0000, 0xFFFF]). Each address stores one byte, and each STM **word** is 2 bytes. STM words are stored in a big-endian manner, that is, the most significant byte in a word is stored first. For example, the word 0x12FE would be stored as:

Address 0x0000	Address 0x0001
12	FE

The runtime stack starts at address 0xFFFFE and grows downward (the addresses get *smaller*).

The static data record is stored after the machine code (which starts at address 0x0000) and grows upward (the addresses get *larger*).

The following page contains Dr. Hanna's map of STM's main memory.



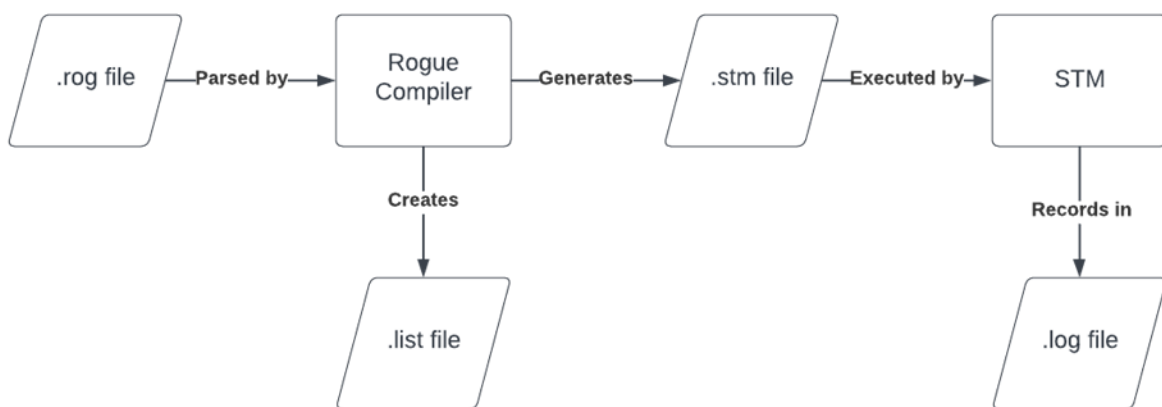
Software Modules

Here is a discussion of the major software modules that make up the Rogue project.

The Rogue compilation process consists of two major modules, both of which are console applications:

- (1) The **Rogue Compiler**, which compiles Rogue source programs and returns compilation information and errors to the programmer. Rogue Compiler is a **single-pass** compiler.
- (2) The **STM**, the target virtual machine developed by Dr. Hanna, simulates a von Neumann machine and can execute files written in its proprietary assembly code.

The following is a flowchart of how the Rogue compilation-to-execution process works:



The Rogue Compiler application asks for a Rogue source file, which must have the extension **.rog**. The compiler will read that source file and, given that there are no syntax errors, will translate it into an STM source file, which has the extension **.stm**. The compilation process will be recorded in a listing file with extension **.list**.

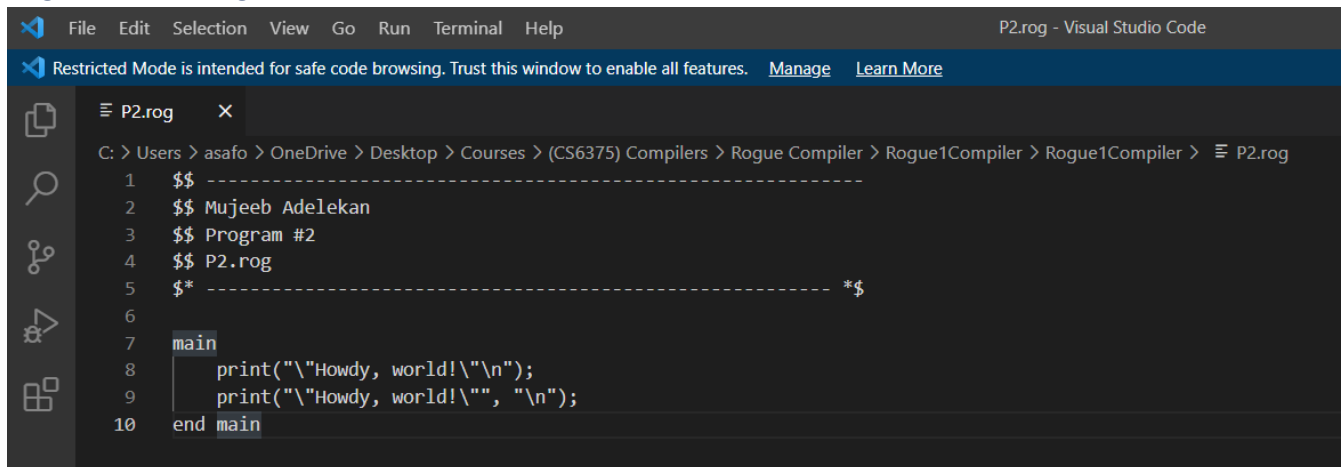
The user must then launch the STM application, which will ask for the **.stm** source file. STM will then read that source file and execute the process until its termination. Information about the machine during execution will be recorded in a **.log** file.

This will fulfill all the Rogue Compiler requirements as well as the requirement of allowing the Rogue programmer to compile Rogue source programs into STM assembly code.

Interface Design

Here is an overview of the four major interface screens in the Rogue compilation process: the Rogue source program, the Rogue Compiler application interface, the STM assembly code, and the STM application interface.

Rogue Source Program



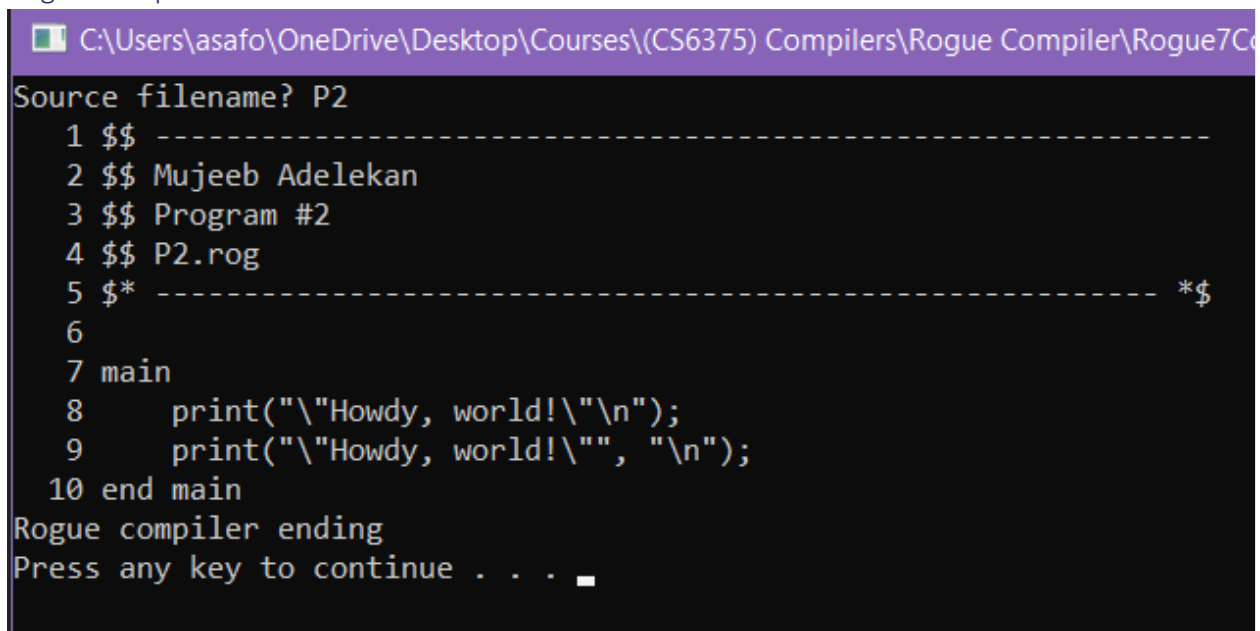
```

File Edit Selection View Go Run Terminal Help
P2.rog - Visual Studio Code
Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More
C: > Users > asafo > OneDrive > Desktop > Courses > (CS6375) Compilers > Rogue Compiler > Rogue1Compiler > Rogue1Compiler > P2.rog
1  $$ -----
2  $$ Mujeeb Adelekan
3  $$ Program #2
4  $$ P2.rog
5  $$* ----- *$
6
7  main
8      print("\Howdy, world!\n");
9      print("\Howdy, world!\", "\n");
10 end main

```

A Rogue source program is any text file with the extension **.rog**. The programmer can write a Rogue source program in any text or code editor, such as Visual Studio Code. Above is an example Rogue program that demonstrates the main function, print statement, and comment system of Rogue.

Rogue Compiler



```

C:\Users\asafo\OneDrive\Desktop\Courses\CS6375 Compilers\Rogue Compiler\Rogue7C
Source filename? P2
1  $$ -----
2  $$ Mujeeb Adelekan
3  $$ Program #2
4  $$ P2.rog
5  $$* ----- *$
6
7  main
8      print("\Howdy, world!\n");
9      print("\Howdy, world!\", "\n");
10 end main
Rogue compiler ending
Press any key to continue . . .

```

The compiler will ask for the name of the .rog source file. When the user hits Enter (or Return on Mac), if the source file is found, the compiler will print each line of that file. If the source file contains a syntax error, the compiler will stop printing each line of the source file and will raise a compiler error. The compilation information along with any errors will be printed in the listing file created in the same directory of the compiler. If the file compiles correctly, the programmer can see the translated STM assembly code in the same directory as well.

STM assembly code

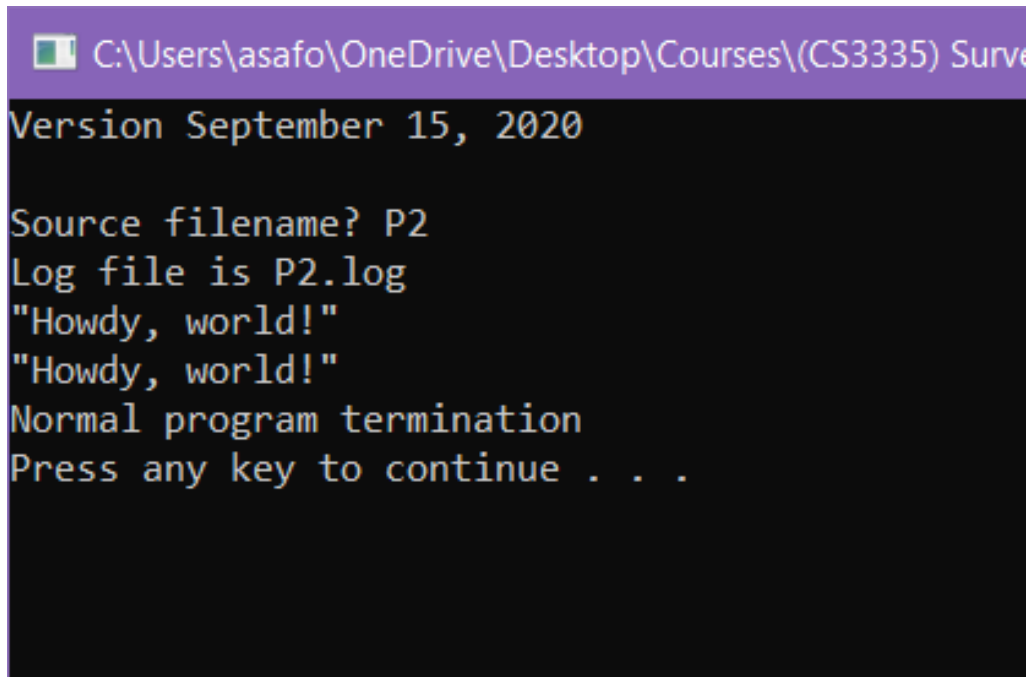
```

31 ; 7 main
32 ; 8   print("\Howdy, world!\n");
33 ; **** =====
34 ; **** 'main' program ( 7)
35 ; **** =====
36 MAINPROGRAM      EQU      *
37                 PUSH      #RUNTIMESTACK      ; set SP
38                 POPSP
39                 PUSHA      STATICDATA        ; set SB
40                 POPSB
41                 PUSH      #HEAPBASE          ; initialize heap
42                 PUSH      #HEAPSIZE
43                 SVC        #SVC_INITIALIZE_HEAP
44                 CALL      MAINBODY0010
45                 PUSHA      SB:0D0
46                 SVC        #SVC_WRITE_STRING
47                 SVC        #SVC_WRITE_ENDL
48                 PUSH      #0D0              ; terminate with status = 0
49                 SVC        #SVC_TERMINATE
50
51 MAINBODY0010     EQU      *
52 ; 9   print("\Howdy, world!\"", "\n");
53 ; **** 'print' statement ( 8)
54                 PUSHA      SB:0D28
55                 SVC        #SVC_WRITE_STRING
56 ; **** 'print' statement ( 9)
57 ; 10 end main
58                 PUSHA      SB:0D49
59                 SVC        #SVC_WRITE_STRING
60                 PUSHA      SB:0D68
61                 SVC        #SVC_WRITE_STRING
62                 RETURN
63 ; **** =====
64 ; **** 'end main' ( 10)
65 ; **** =====
66

```

The STM assembly code is generated by the Rogue Compiler and can be viewed in any text or code editor. This assembly code is designed to run on the STM target machine.

STM (running the code)

A screenshot of a terminal window with a purple title bar. The title bar text is "C:\Users\asafo\OneDrive\Desktop\Courses\CS3335) Surve". The terminal content is as follows:

```
Version September 15, 2020  
  
Source filename? P2  
Log file is P2.log  
"Howdy, world!"  
"Howdy, world!"  
Normal program termination  
Press any key to continue . . .
```

The user will be prompted to enter the name of the STM source file to be run. Upon hitting Enter/Return, STM will run the code then terminate. After termination, STM will create a log file that contains information about the machine during execution.

BNF Grammar for Rogue

Here is a listing of the context-free syntax of Rogue, described using Backus-Naur form, or BNF.

Note the BNF conventions used is a modified version developed by Dr. Hanna, which follows the following rules (as written by Dr. Hanna):

- (1) $\langle x \rangle$ means “ x is a non-terminal symbol”
- (2) $::=$ means “is defined as”
- (3) $|$ means “or”
- (4) $\{ x \}^*$ means “ x repeated zero or more times”
- (5) $\{ x \}_n$ means “ x repeated exactly n times”
- (6) $[x]$ means “ x is optional”
- (7) an underlined metasyMBOL like $\underline{[}$ and $\underline{] }$ means “ $[$ and $]$ should be treated as terminal symbols”
- (8) $((x_1 \mid x_2 \mid \dots \mid x_n))$ means “chose exactly 1 from the list of alternates x_1, x_2, \dots, x_n ”
- (9) $||$ introduces a BNF comment that extends to end-of-line.

The full BNF grammar of Rogue is shown on the following four pages. The Rogue compiler will enforce this grammar using a recursive-descent parser, where each nonterminal corresponds to a function in which the compiler expects to read the terminal symbols in the order listed in the grammar. If the compiler determines that the syntax is not being followed, a descriptive error will be returned to the programmer and the compilation process will be terminated.

This will meet the requirement of allowing the programmer to write the language features of Rogue in a code editor to be read by the Rogue compiler. Note that due to the nature of BNF, only the context-free syntax is described. Context-dependent syntax of the new language features being added will be discussed following the BNF.

The non-terminals highlighted in **yellow** represent **updated** Rogue grammar, with the specific updates to the definitions highlighted as well. The non-terminals highlighted in **cyan** represent **new** Rogue grammar.

```

<RogueProgram> ::= { <enumDeclaration> }*
                  { <dataDefintions> }*
                  { (( <ProcedureDefinition> | <FunctionDefinition> )) }*
                  <MainProgram> EOPC

<enumDeclaration> ::= enum : <enumType> { <enumConstant> {, <enumConstant> }* };

<enumType> ::= <identifier>

<enumConstant> ::= <identifier>

<dataDefinitions> ::= <variableDefinitions> | <constantDefinitions> | <AssociativeArrayDefinition>

<variableDefintions> ::= <datatype> <identifier> [ [ <integer> {, <integer> }* ] ]
                      {, [ <datatype> ] <identifier> [ [ <integer> {, <integer> }* ] ] };

<constantDefinitions> ::= perm <datatype> <identifier> = <literal>
                      {, [ <datatype> ] <identifier> = <literal>}* ;

<AssociativeArrayDefinition> ::= assoc <identifier> { <integer> };

<datatype> ::= (( int | bool | char | float | ptr | file | <enumType> ))

<MainProgram> ::= main
                  { <enumDeclaration> }*
                  { <dataDefintions> }*
                  { <statement> }*
                  end main

<ProcedureDefinition> ::= proc <identifier> [ ( <formalParameter> { , <formalParameter> }* ) ]
                      { <dataDefinitions> }*
                      { <statement> }*
                      end proc

<FunctionDefinition> ::= func <identifier> : <datatype> ( [ <formalParameter> {, <formalParameter> }* ] )
                      { <dataDefinitions> }*
                      { <statement> }*
                      end func

```

```

<formalParameter>      ::= [ (( in | out | io | ref )) ] <identifier> : <datatype>

<statement>            ::= { <assertion> }*
                        ((
                          <PrintStatement>
                          | <InputStatement>
                          | <AssignmentStatement>
                          | <IfStatement>
                          | <DoWhileStatement>
                          | <ForStatement>
                          | <CallStatement>
                          | <ReturnStatement>
                        ))
                        { <assertion> }*

<assertion>            ::= assert( <expression> );

<PrintStatement>       ::= print( (( <string> | <expression> )) {, (( <string> | <expression> )) }* ) ;

<InputStatement>       ::= input( [ <string> ,] <variable> );

<AssignmentStatement> ::= (( <variableList> | <AssociativeArrayReference> )) = <expression>;

<variableList>         ::= <variable> {, <variable> }*

<IfStatement>          ::= if ( <expression> )
                        { <statement> }*
                        { elif ( <expression> )
                          { <statement> }* }*
                        [ else
                          { <statement> }* ]
                        end if

<DoWhileStatement>     ::= [ do
                        { <statement> }* ]
                        while ( <expression> )
                        { <statement> }*
                        end while

```



```

<ForStatement>      ::= for <variable> = <expression> to <expression> [ by <expression> ]
                        { <statement> }*
                        end for

<CallStatement>     ::= call <identifier> [ ( ( ( <expression> | <variable> ) )
                                                { , ( ( <expression> | <variable> ) ) }* ) ];

<ReturnStatement>   ::= return [ ( <expression> ) ];

<expression>        ::= <conjunction> { ( ( or | nor | xor ) ) <conjunction> }*

<conjunction>       ::= <negation> { ( ( and | nand ) ) <negation> }*

<negation>          ::= [ not ] <comparison>

<comparison>        ::= <comparator> [ ( ( < | <= | == | > | >= | ( ( <> | != ) ) ) ) <comparator> ]

<comparator>        ::= <term> { ( ( + | - ) ) <term> }*

<term>              ::= <factor> { ( ( * | / | % ) ) <factor> }*

<factor>            ::= [ ( ( abs | + | - ) ) ] <secondary>

<secondary>         ::= <prefix> [ ( ( ^ | ** ) ) <prefix> ]

<prefix>            ::= <primary> || ( ( ++ | -- ) ) <variable>

<primary>           ::= <variable> | ( <expression> ) | <literal>
                        | <FunctionReference> | <AssociativeArrayReference>

<variable>          ::= <identifier> [ [ <expression> { , <expression> }* ] ]

<FunctionReference> ::= <identifier> ( [ <expression> { , <expression> }* ] )

<AssociativeArrayReference> ::= <identifier> { <expression> }

<identifier>        ::= <letter> { ( ( <letter> | <digit> | _ ) ) }*

<literal>           ::= <integer> | <boolean> | <character> | <float> | <string> | addr(<variable>)
                        cont(<variable>) | <file>

```

```

<integer>      ::= <digit> { <digit> }*
<boolean>      ::= true | false
<character>    ::= '<ASCIICharacter>' || \ and " must be escaped
<float>        ::= <digit> { <digit> }* . <digit> { <digit> }* [e [-] <digit> { <digit> }* ]
<string>       ::= " { <ASCIICharacter> }* " || \ and " must be escaped
<digit>        ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<comment>      ::= $$ { <ASCIICharacter> } * EOLC           || single-line
                  |$_ { (( <ASCIICharacter> | EOLC )) }* _$    || multi-line
<ASCIICharacter> ::= || Every printable ASCII character in range [ ' ', '~' ]

<file>         ::= <identifier>
<FilePrintStatement> ::= f_print( <file>, (( <string> | <expression> ))
                                   {, (( <string> | <expression> )) }* );
<FileInputStatement> ::= f_input( <file>, <variable> );
<FileOperationStatement> ::= <file> = f_(
                                   create( <string> )
                                   | read( <boolean> )
                                   | write( <boolean> )
                                   | clear()
                                   | open( <string> )
                                   | close()
                                   | delete()
                                   ));

```

Pointers

Rogue allows programmers to declare, dereference, and assign addresses to pointers.

To declare a pointer, use the keyword `ptr` followed by an identifier. In Rogue, the programmer does not have to declare the datatype of the variable that the pointer points to.

To assign an address to a pointer, use the keyword `addr(<variable>)`, where `<variable>` is the variable that the programmer wants to take the address of.

To dereference a pointer, that is, access the contents of the address, use the keyword `cont(<variable>)`, where `<variable>` is the pointer that is to be dereferenced.

The following is Rogue code that declares two `int` variables, `x` and `y`, and declares a pointer named `p`. `p` is then assigned with the address of `x`. Afterwards, `x` is assigned with the value 6. Finally, `p` (which points to `x`) is dereferenced. That is, the contents of the address stored in `p` is assigned to `y` (since `p` stores the **address of** `x`, the **value of** `x` is assigned to `y`, so the value of `y` is now 6).

```
int x, y;
```

```
ptr p;           $$ declares a pointer named p
```

```
p = addr(x);
```

```
x = 6;
```

```
y = cont(p);     $$ y is now equal to 6
```

Enumeration Types

Rogue supports enumeration types, which are user-defined data types in which all possible values are listed (enumerated) in the definition. These values are called *enumeration constants* and are implicitly assigned integer values starting at 0.

To declare an enumeration type, use the keyword `enum` followed by a colon `:` and an identifier for the type's name. Then, declare all the enumeration constants of that type in a comma-separated list within curly brackets `{ }`. End the declaration statement with a semicolon `;`.

For example, the following Rogue statement declares an enumeration type `day` with its values being the seven days of the week:

```
enum : day {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

An enumeration type can be used as a datatype in the scope it has been declared in. Remember that the values of an enumeration type are the set of constants listed in the definition.

The following Rogue code declares a `day` variable `weekStart` and a 5-element `day` array `weekDays`. The variable `weekStart` is assigned with the `day` value `Mon`, and the elements of `weekDays` are assigned with the `day` values `Mon` through `Fri`.

```
$*** Declare a variable called weekStart ***$
day weekStart;
```

```
$*** Declare an array called weekDays ***$
day weekDays[5];
```

```
$*** Assign weekStart with Monday ***$
weekStart = Mon;
```

```
$*** Assign the elements of weekDays with Mon-Fri ***$
weekDays[0] = Mon;
weekDays[1] = Tue;
weekDays[2] = Wed;
weekDays[3] = Thu;
weekDays[4] = Fri;
```

Associative Arrays

Rogue supports associative arrays. In Rogue associative arrays, each element is a **key-value pair** where the key serves as the unique index for its corresponding value. Rogue keys and values can be of type `int`, `bool`, `char`, or `float`.

To declare an associative array, use the keyword `assoc` followed by an identifier with curly braces `{ }` containing an `int` value representing the capacity of the associative array (i.e., the maximum number of key-value pairs the associative array will store). The following Rogue code declares an associative array with a capacity of 6 elements.

```
$***** Define an associative array with 6 (key, value) pairs
*****$

assoc a_arr{6};
```

Adding an associative array element

To add a key-value pair to an associative array that has been declared in the current scope, write an assignment statement in the following format:

```
arrayName{ <key> } = <value> ;
```

where `<key>` and `<value>` are `int`, `bool`, `char`, or `float` expressions. The following Rogue code adds 6 key-value pairs to the associative array `a_arr`

```
$***** Add the hexadecimal digits A-F with their values in
decimal *****$

a_arr{'A'} = 10;
a_arr{'B'} = 11;
a_arr{'C'} = 12;
a_arr{'D'} = 13;
a_arr{'E'} = 14;
a_arr{'F'} = 15;
```

Modifying the value of a key-value pair

To modify the value of an existing element in an associative array, use the key that corresponds to that value when referencing that element on the left-hand side of an assignment statement.

The following Rogue code modifies two values in `a_arr`:

```
$* this will replace the value of key 'B' with a float literal,
3.14 *$
```

```
a_arr{'B'} = 3.14;
```

```
$* this will replace the value of key 'E' with a bool literal,
false *$
```

```
a_arr{'E'} = false;
```

Retrieving the value for a given key

To retrieve the value of an associative array element for a given key, specify the name of the associative array followed by curly braces `{ }` containing the key. The following Rogue code prints the value that corresponds to the key `A` in the associative array `a_arr`:

```
$*** Print the decimal value for the hex digit A ***$
print("The demical value of A is ", a_arr{'A'}, "\n");
```

As another example, the following Rogue code assigns an `int` variable `x` with the value corresponding to the key `D` in `a_arr`. Assume that the variable `x` has been declared.

```
$*** Assign an int variable x with a_arr{'D'} (the int value 13)
***$
x = a_arr{'D'};
print("The demical value of D is ", x, "\n");
```

File Handling

Rogue allows programmers to create, read, update, and delete .txt files on a Windows machine. In Rogue, files are pointed to by file variables, which are declared using the file datatype, as follows:

```
file f;
```

Creating a new file

`f_create()` creates a new file and points a variable to that file. A run-time error occurs when that file already exists. The default path of the file specified is the same path of the STM virtual machine. For example, the statement:

```
f = f_create("fileName.txt");
```

creates a new file named `fileName.txt` and points `f` to that file. The reading and writing permissions of a newly created file are set to **false** for safety. The Rogue programmer can manually toggle these permissions using `f_read()` and `f_write()`, both of which are described in the following sections.

Reading from an existing file

To toggle the reading functionality of a file, use `f_read()`. For example, using the declared file variable `f`, the statement:

```
f = f_read(true);
```

allows reading from the file pointed to by `f`. Conversely,

```
f = f_read(false);
```

deactivates reading from the file pointed to by `f`. A run-time error occurs when `f_read()` is used on a file variable that does not point to an existing file.

A file's contents can be taken as input using a special file input statement. If `var` is a declared variable of `int`, `float`, `char`, or `bool` type, then

```
f_input(f, var);
```

reads a **single-line** of the file pointed to by `f` then stores the type-equivalent of the input into `var`. A run-time error occurs when `f_input()` receives input that cannot be converted to the appropriate data type or when `f_input()` is used on a file with a read status set to `false`.

Writing (appending) to an existing file

To toggle the writing functionality of a file, use `f_write()`. For example, using the declared file variable `f`, the statement:

```
f = f_write(true);
```

allows writing to the file pointed to by `f`. Conversely,

```
f = f_write(false);
```

deactivates writing to the file pointed to by `f`. A run-time error occurs when `f_write()` is used on a file variable that does not point to an existing file.

To print, or append, contents to a file, using the `f_print()` statement. For example, the following statement prints "Hello world!" followed by a newline into the file pointed to by `f`:

```
f_print(f, "Hello world!\n");
```

As another example, the following statement prints " $3 + 5 * 4 = 23$ " into the file pointed to by `f` (the expression is evaluated before it is printed to the file).

```
f_print(f, "3 + 5 * 4 = ", 3 + 5 * 4);
```

When using `f_print()`, all contents are written at the end of the file. A run-time error occurs when `f_print()` is used on a file with a write status set to `false`.

Clearing the contents of a file

To erase all of the contents of a file, use `f_clear()`. For example, the statement:

```
f = f_clear();
```

erases all of the contents of the file pointed to by `f`. This gives the programmer an "empty" file to write to. A run-time error occurs when `f_clear()` is used on a file variable that does not point to an existing file or when `f_clear()` is used on a file with a write status of `false`.

Opening and closing a file

To point a file variable to an existing file, use `f_open()`. For example, the statement:

```
f = f_open("existingFile.txt");
```

points `f` to `existingFile.txt`. When a file is opened (newly pointed to), its read and write statuses are set to `false` by default. A run-time error occurs when `f_open()` is used to open a file that does not exist.

To get a file variable to stop pointing to a file, use `f_close()`. For example, the statement:

```
f = f_close();
```


makes `f` stop pointing to its file. A run-time error occurs when `f_close()` is used on a file variable that does not point to an existing file.

Deleting a file

To delete a file from the directory, use `f_delete()`. For example, the statement:

```
f = f_delete();
```

Deletes the file pointed to by `f`. The deleted file will no longer be found in the directory and `f` will not point to anything. A run-time error occurs when `f_delete()` is used on a file variable that does not point to an existing file.

Plan

This section will discuss the development plan of this project. Each task that must be completed will be described in detail then scheduled on a month-long timeline.

Tasks

This subsection lists the tasks of this project. The project consists of eight phases: Task Planning, Preliminary Study, Design, Develop, Integrate, Test, Documentation (System Manual), and Demonstration (Project Demo). Since the objective is to design and implement new features into the Rogue language, the Design and Develop phases are broken into multiple tasks, with each task focusing on a single Rogue feature. The Test phase is broken into areas of concern regarding the functionality of the Rogue Compiler.

Task Descriptions:

- **Task Planning:** The main objective of the project is broken down into manageable tasks that can be used as a progress benchmark. A task schedule and Gantt chart is produced.
- **Preliminary Study:** Popular languages are surveyed to determine how Rogue will implement new language features. An outline of the history of languages, an analysis of the problem, and a list of functional requirements is produced.
- **Design – Associative Arrays:** The context-free grammar of the Rogue features that will be added in this project is written in a version of Backus-Naur Form. The grammar for Rogue's associative arrays is designed for this comprehensive project.
- **Design – Enumerations:** The context-free grammar of the Rogue features that will be added in this project is written in a version of Backus-Naur Form. The grammar for Rogue's enumeration types is designed for this comprehensive project.

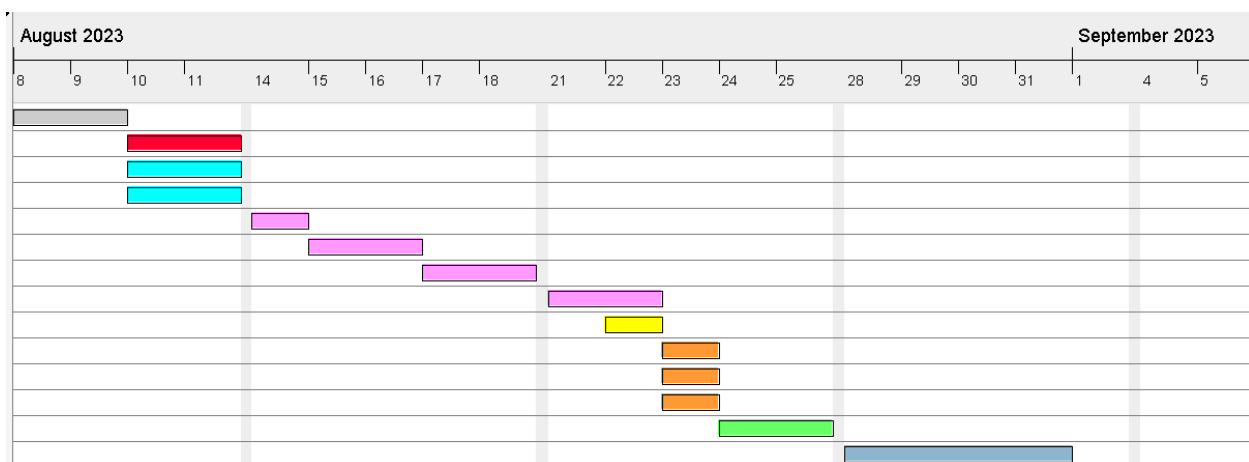
Note that the context-free grammar for Rogue's pointer dereferences, file I/O, and enumeration types, and associative arrays were designed in CS6340 Advanced Software Engineering.

- **Develop – Pointer Dereferences:** The new language features are added to Rogue in a modular manner. Pointer dereferences are implemented in the Rogue Compiler in this stage of development.
- **Develop – File I/O:** The new language features are added to Rogue in a modular manner. File I/O functionality is implemented in the Rogue Compiler in this stage of development.
- **Develop – Associative Arrays:** The new language features are added to Rogue in a modular manner. Associative arrays are implemented in the Rogue Compiler in this stage of development.
- **Develop -- Enumerations:** The new language features are added to Rogue in a modular manner. Enumeration types are implemented in the Rogue Compiler in this stage of development.


- **Integrate**: Each new feature is implemented as a parsing module in the Rogue Compiler. Once all features are implemented, the Rogue Compiler will represent the full Rogue language.
- **Test – Complete Rogue Programs**: Rogue programs that combine and demonstrate multiple features in the Rogue language are compiled to ensure that the Rogue Compiler compiles and runs valid code.
- **Test – Syntax Errors**: Rogue programs that combine and demonstrate multiple features in the Rogue language are compiled to ensure that the Rogue Compiler returns compile-time (syntax) errors when encountered.
- **Test – Run-time Errors**: Rogue programs that combine and demonstrate multiple features in the Rogue language are compiled to ensure that the Rogue Compiler returns run-time errors when encountered.
- **System Manual**: A documentation of the development and functionalities of Rogue. This document contains example programs to help the user learn the language and a guide on how to troubleshoot compile-time and run-time errors. The System Manual also includes an appendix of the Rogue compilers source code as well as the code of the STM virtual machine.
- **Project Demo**: A demonstration of the major features of the Rogue language and compiler in front of a three-person committee.

Schedule

This subsection outlines the timeline on which each task is expected to be completed. The following image is a Gantt Chart of all the tasks, colored to match their descriptions in the previous subsection.



The following is a list of all tasks along with the beginning and end date of each task.



Name	▲ Begin date	End date
• Task Planning	8/8/23	8/9/23
• Preliminary Study	8/10/23	8/11/23
• Design: Associative Arrays	8/10/23	8/11/23
• Design: Enumerations	8/10/23	8/11/23
• Develop: Pointer Dereferences	8/14/23	8/14/23
• Develop: Associative Arrays	8/15/23	8/16/23
• Develop: Enumerations	8/17/23	8/18/23
• Develop: File I/O	8/21/23	8/22/23
• Integrate: Parsing Modules	8/22/23	8/22/23
• Test: Complete Rogue Programs	8/23/23	8/23/23
• Test: Syntax Errors	8/23/23	8/23/23
• Test: Run-time Errors	8/23/23	8/23/23
• System Manual	8/24/23	8/25/23
• Project Demo	8/28/23	8/31/23

Cost

This subsection will cover the cost of Rogue's development as a single-person project.

According to FullStack Labs, freelance, inexperienced software engineers expect an hourly wage from \$50 to \$70 in projects ranging in scope from \$1,000 to \$50,000. Assuming Rogue is developed by a single developer, who works 12 hours a week for 5 weeks (about a month) and is paid the lower end of the expected hourly wage (\$50), the project will have a development budget of \$3,000.

$$\text{Time Cost} = \$50/\text{hr.} * 12 \text{ hr./week} * 5 \text{ weeks of work} = \$3,000$$

It would be good practice to keep the developer's hardware and software up to date, and as such the developer will be in the market for a Windows 11 machine. The most affordable Windows 11 device found in my research is the Microsoft Surface Pro X with the Microsoft SQ1 processor and 8 GB of RAM. Its Best Buy clearance price, as of June 2023, is listed at \$406. Supplying the developer with a Surface Pro X at this price will raise the budget to \$3,406.

$$(\text{Time Cost}) + (\text{Hardware Cost w/ Software Built-in}) = \$3,000 + \$406 = \$3,406$$

This initial cost of development will increase in accordance with updates and maintenance to the Rogue language.

Marketing

This section contains an overview of Rogue’s demographic and marketing campaign.

Rogue is designed for application programmers, regardless of skill-level. Therefore, the marketing of Rogue should appeal to both aspiring programmers and industry professionals. This requires a large breadth of exposure that should highlight how Rogue is just as scalable as the most popular programming languages but is easier to learn and debug. This point can be emphasized by having advertisements that contrast coding in Rogue with the hurdles that are common in developing Java, C#, Python, etc.

This marketing campaign should reach both casual and professional audiences. A social media presence, with accounts on TikTok, Twitter, Reddit, and LinkedIn, will keep a consistent brand identity and generate interest in various markets. This campaign will be bolstered by commercials on YouTube, as well as a YouTube channel that will introduce Rogue and offer tutorials for the language’s features.

A modern, accessible website will go live and outline the strengths of the language as well as offer documentation. Finally, Rogue will be promoted in the professional space through presentations at developer conferences.

Since most language compilers are offered for free, Rogue will be no different. The customer can access current and legacy versions of the Rogue Compiler through the Rogue website.

The following are potential scripts for advertisements that can play on TikTok or before YouTube videos. These ads can also be displayed on Reddit, Twitter, or LinkedIn with a link to Rogue’s website.

Example Ad 1:

Why should we follow the machine’s rules?

Go ***Rogue***

A programming language written by humans, *for* humans.

Example Ad 2:

Tired of being pushed around by your code?

Go ***Rogue***

A new programming language that is easy to learn *and* easy to master.

References

- C language*. cppreference.com. (2022, September 20). Retrieved February 6, 2023, from <https://en.cppreference.com/w/c/language>
- C++ language*. cplusplus.com. (n.d.). Retrieved March 6, 2023, from <https://cplusplus.com/doc/tutorial/>
- IBM Corporation. (2021, March 3). *Enumerations*. IBM Documentation. Retrieved August 4, 2023, from <https://www.ibm.com/docs/en/zos/2.3.0?topic=types-enumerations>
- Pressman R., Maxim B. (2014). *Software Engineering: A Practitioner's Approach* (8th ed.). McGraw-Hill Education, New York, United States
- Python Software Foundation. (2023, February 7). *Python 3.11.1 documentation*. Python . Retrieved February 7, 2023, from <https://docs.python.org/3/>
- Quick Start Guides for Windows 10, Surface Book, and Microsoft Edge*. Microsoft Support. (2023). Retrieved February 7, 2023, from https://support.microsoft.com/en-us/microsoft-edge/quick-start-guides-for-windows-10-surface-book-and-microsoft-edge-4e603411-16ad-73f7-0923-5aa3d327bb59#bkmk_windowsqsg
- Sebesta, R. W. (2016). *Concepts of Programming Languages* (11th ed.). Pearson.
- Software Development Price Guide & Hourly Rate Comparison*. FullStack Labs (n.d.). Retrieved February 7, 2023, from: <https://www.fullstacklabs.co/blog/software-development-price-guide-hourly-rate-comparison>