Document Type	Technical Manual
Name (Student Number)	• Ala Al Din Afana (16395986) • Shanan Lynch(14723071)
Supervisor	David Sinclair

1. Introduction

1.1 overview

The goal of this document is to give the reader a technical overview of the technologies used and architectures implemented to develop the mona programming language platform. For our fourth year project we designed and developed an online first programming language called Mona. Our programming language is accessible through a web application hosted on EC2. The goal of our programming language is to allow novice programmers to learn the fundamentals of programming and help them transition smoothly into more advanced programming languages. To achieve this we researched and broke down two of the biggest and most popular programming languages “python” and “java” and created an understandable **relatable** syntax to help users transition between both. The motivation to develop this project came from the difficulty software developers have from transcending from one programming language to the other. We encountered this issue ourselves when we were moving from python to java in 2nd year. In this document we will discuss in depth our attempt to achieve this and the outcome of those attempts later in this document.

1.2 Glossary

Term	**Meaning**
JAVACC	JAVA Compiler Compiler (JAVACC) is a parser generator, JAVACC could be downloaded from the official Javacc website.
AWS	Amazon Web Services is a subsidiary of Amazon that provides on-demand cloud computing platforms and APIs to individuals

Mona	The name of our programing language inspired by the painting Mona Lisa by Leonardo Di Vinci, We decided on this name because Leonardo Di Vinci was a polymath with work that will last decades,The name is also short and unique, this is similar to how Guido van Rossum derived the name python from “Monty Python's Flying Circus ".
EC2 instance	EC2 Instance is a virtual server in Amazon's Elastic Compute Cloud (EC2) for running applications on the Amazon Web Services (AWS
LLVM IR	LLVM IR is the assembly language we use to produce our Ir code
LL(1) parser	LLparser(Left-to-right,Leftmost derivation) is a top-down parser for a subset of context-free languages. ll(1) means our language uses one lookahead
JJTree	An extremely powerful tree building pre-processor. That we use to build our abstract syntax tree
BNF Notation	Backus–Naur form or Backus normal form is a notation technique for context-free grammars, often used to describe the syntax of languages used

2.Research

In this section we will discuss research we performed before deciding on the grammar we implemented for this language.

According to (C.A.R.Hoare paper “HINTS ON PROGRAMMING LANGUAGE DESIGN” from Stanford University) “A programming language is regarded as a tool to aid the programmer , Meaning it should give assistance in the most difficult aspects of his Art , Namely program design.”. Mona is designed in order to minimize the difficulty developers encounter in their art.

2.1 Program Design

We decided on a less flexible design, restricting programmers from some syntax freedom provided by languages such as python. A good programming language should assist in establishing and enforcing the programming conventions and disciplines which will ensure harmonious co-operation of the parts of a large program when they are developed separately and finally put together. Java does this very well by providing functions and classes to separate blocks of code. It has a strict syntax that limits the creation of bugs throughout code development. However some aspects of java may be seen as too strict for a novice programmer. The no. of lines of a code a user must write in java in order to print hello world can be discouraging , especially that a novice doesn’t know what a class is or what a public static void could mean.

Below is an overview of some programming conventions in our language and rationale behind them:

- Program structure:
    - global variables declarations
    - Function declarations
    - Main block
- This is the structure of a basic mona program the reason our program is designed like this to enforce users to declare functions before their use, as some interpreted languages such as python are executed in a linear fashion (top to bottom) and calling a function before interpreting it’s code will cause an error.
- Functions cannot be used before they are declared a novice would not know the difference between python and java and through restricting him/her to these standards he won't run into these problems when he transitions to python (as a novice)

- Identifier declaration
    - var string a ;
    - To declare an identifier you have to specify if it's a “**var**” or “**const**” followed by the type to educate the user about the difference between constant and variable datatypes.
- Array declaration
    - Array elements have to be of the same type.
    - This gives users an idea on how arrays work in memory

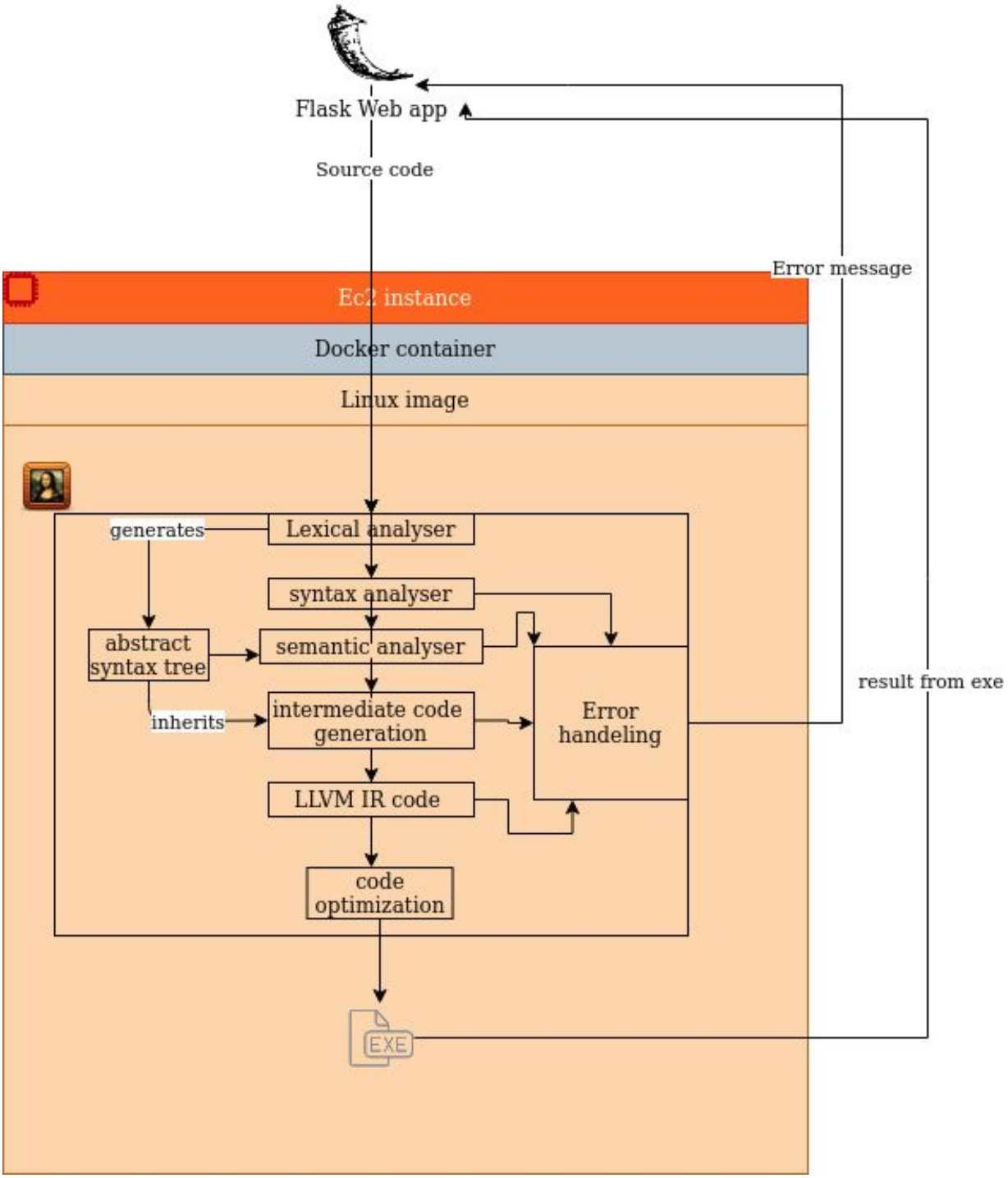
- Python this not limit users and allows them to have multiple types in arrays
- This is because python arrays are a lists store references to memory pointers

This strict format limits simple errors that a novice can make, it allows a beginner programmer to learn about different data types and why these data types should be kept separate; it also allows the users to learn about the underlying data structure. This also enabled us to develop a more complex programming language rather than a simple one where the abstract syntax tree is limited to a small number of nodes.
We also decided on strict syntax for our language because through our research we discovered designing a language that has simple syntax for example python can negatively impact the programmer transitioning into a more complex designed language. It can be argued that a novice who learns to program in c++ will have little difficulty coming into java or any other language whereas transitioning from python can be somewhat challenging. This can be a major issue as C and C++ are two core programming languages used to date.

Note: The grammar of our language can be examined in more dept in the **User Manual** and **BNF document** found in this repository.

3.System architecture

3.1 system architecture diagram



3.2 Flask web application

The flask web application connects the front end to the back end. The UI allows users to input code via forums and click the compile button which will essentially trigger a post request with the data to the python modules in the backend.

The source code is then passed from the backend and sent to the mona programming language and compiled into llvm Ir code **(more detail in section 3.3)**.
LLvm Ir is then executed and the results are displayed on the web app.

- We chose flask because:
    - A great framework to write server side scripts to run the user code on the mona compiler.
    - Flask is a python framework by choosing flask we would of covered three programming languages in our project
        - Java
        - Python
        - Llvm Ir (assembly language)

3.3 Amazon ec2 instance

“An EC2 Instance is a virtual server running on Amazon's Elastic Compute Cloud (EC2) for running applications on the Amazon Web Services (AWS)”. We hosted our compiler and webapp on an Amazon EC2 Linux instance where:

- upon code submission (clicking the “compile” button) a python script is triggered which fetches the data from the front end and makes system calls to compile and run the code
- Once the code is executed it retrieves the output or any errors encountered and returns them to the flask application so they can be formatted and displayed in the front end.

4. Compiler Design

4.1 Lexical analysis

Lexical analysis is the first stage of code compilation. In this stage the source code is broken down into lexemes/tokens. These tokens are then parsed. This process is performed in the mona.jjt.
In the mona.jjt file five important tasks are performed

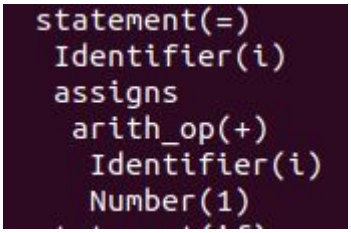
1. The ***code is parsed*** if the tokens are parsed successfully the compilation of the program can continue
2. We generate the ***abstract syntax tree***
    a. This is a very important process and well be discussed in more detail later on
3. We create a ***symbol table***
4. We create and call a ***TypeCheckVisitor*** (semantic checks)
5. We create and call an ***IrCodeVistor*** (IR code generation)

4.2 Abstract syntax tree

4.2.1 Overview of AST

- To generate the abstract syntax tree we created a SimpleNode at the beginning of the program.

- We then created a subsequent node at each stage of the production rules.
- We specify each node to have a number of children.
- Example



- In this scenario statement = has two children Identifier and assigns
- Assigns has one children arith_op(+)
- Arith_op(+) has two children identifier i and 1

4.2.2 Complexity of AST

- The abstract syntax tree is one of the most important and foundational structures in our project as semantic checks and Ir code generation rely on a well structured ast.
- Designing a well structured abstract syntax tree is a difficult task as one error in the abstract syntax tree can lead to dire consequences in the future stages of the programming language.

4.3 Symbol table

4.3.1 Overview

Symbol table is an important data structure we created in order to store information about the occurrence of various entities such as variable names, function names, scope of these entities ect . The symbol table is used in both Semantic checks and Ir code generation.

4.3.2 Complexity of lexical analysis

In the lexical analysis phase in the mona.jjt file there are multiple problems that we solved within our grammar.
We needed to get rid of all left factors to ensure that there isn't infinite recursion of production rules calling themselves directly or indirectly.
Left Factoring is a grammar transformation technique we used, It is composed of "factoring out" prefixes which are common to two or more productions.

We successfully removed all left factors and our grammar has no look aheads.

4.4 Semantic analysis

4.4.1 Overview

- **Semantic analysis:** this phase of the compiler checks if the semantics are correct. We used the abstract syntax tree to derive the nodes and perform semantic analysis

on the code. To help us with our semantic checks we used the symbol table. we used the semantic checks ensure the following:

- Identifiers are declared within the scope they are used
- No multiple declarations of the same identifier.
- Correct types for variable assignments
- Argument of arithmetic operations are of correct type
- Every function call has the correct number of arguments when it is called.
- Correct type for lists.
- Identifier declared before use

- In the typeCheckVisitor we implement the mona visitor and traversed through the abstract syntax three nodes visiting all the nodes in search tree approach.

4.4.2.Complexity of Semantic Analysis

- Semantic analysis is a very difficult task as we need to deal with many dataTypes , It has criticality and if the semantic analysis is not performed well it can lead to wrong compilation in the code and the generation of defected IR code
- Traversing the tree is a very difficult task many semantic checks require us to traverse the tree recursively

4.5 IR code generation

In this section we developed our IR code in llvm IR assembly code. The LLVM compiler infrastructure project is a set of compiler and toolchain technologies, which can be used to develop a front end for any programming language and a back end for any instruction set architecture. Llvm IR supports static single assignment, SSA requires that each variable is assigned exactly once, and every variable is defined before it is used.

4.5.1 Llvm IR background information

Originally implemented for C and C++,

Languages with compilers that use LLVM include ActionScript, Ada, C#, Common Lisp, Crystal, CUDA, D, Delphi, Dylan, Fortran, Graphical G Programming Language, Halide, Haskell, Java bytecode, Julia

4.5.2 Llvm IR code generation

- To generate the Ir code we had to traverse the abstract syntax tree and travel through each node and write to the IR code file.
- This was a very hard task and we had to built multiple infrastructures to assist as with this
- We utilized HashMaps to map list lengths , list type , string declared , ect.
- This allowed us to store and pass down as much information as we needed while traversing the abstract syntax tree.

- Because llvm IR is a Static single assignment
    - you need to allocate memory for each variable.
    - To perform any operation on the value you need to load the value from the memory register
    - After you performed the operations you need to use the store instruction to store the new value in the memory register

- The ***DataTypes*** we used in llvm Ir
    - **Ints** were stored in memory in **i32***
    - **Floats** where stored in memory in **double***
    - **Strings** where a very complex one to implement at the start we stored strings in **I8*** but while developing our IR code we realised this limits Strings from performing multiple operations .
    - For example comparison operators.
        - When llvm IR compares two i8* that were loaded into an i8 variable they do not contain the same values as they point to two different memory pointers
    - To overcome this we changed the data structure to [20 x i8]*
    - This allowed is to compare each ascii pointer , where they are stored in the same location in memory
    - This allowed us to produce accurate results for comparison
    - Changing the infrastructure also allowed us to be able to retrieve single i8 values from the array adding more usability to the user
    - **Array architecture :** Arrays are a collection of similar data type values , that are stored in memory in a continuous sequence.
    - An array of 5 integers are represented like this in llvm [5 x i32]
    - llvm we first where allocating fixed size to the arrays based on the array size during declaration
    - To insert a new element to an array we had to allocate memory to a new array with one extra memory pointer.
    - We then store the value that needs to be stored in that pointer
    - This was a perfect solution but did not work when inserting values in loops
    - We needed to create a more structured array definition.
    - To solve this problem we limited arrays to a size of 256 memory pointers

4.5.3 Built in functions

Some built in functions we made in llvm that allowed us to decrease the size of the llvm Ir code include

- StringCompare
    - This is a built in function written in llvm for the comparison of strings.
    - This function is called when two strings are compared to each other using the comparative operator ==.
    - The function takes in two memory ptr [20 x i8]*
    - It compares the memory ptrs at each index
    - If the ptrs all the ptrs are equal to each other it and returns true
    - If not it returns false

Sample code StringCompare

```
define i1 @BINStringCmp( [20 x i8]* %.a, i32 %.la, [20 x i8]* %.b, i32 %.lb )
 {
%.true = getelementptr [5 x i8], [5 x i8]*@.true, i64 0, i64 0
%.false = getelementptr [6 x i8], [6 x i8]*@.false, i64 0, i64 0
%.newline1098019 = getelementptr [2 x i8], [2 x i8]*@.newline1098019, i64 0, i64 0
%.t1 = icmp ne i32 %.la, %.lb
br i1 %.t1, label %label1, label %label2
label1:
ret i1 0
br label %label3
label2:
br label %label3
label3:
%.i = alloca i32
store i32 0, i32* %.i
%.t4 = load i32, i32* %.i
br label %label4
label4:
%.t6 = load i32, i32* %.i
%.t5 = icmp slt i32 %.t6, %.la
br i1 %.t5, label %label5, label %label6
label5:

%.charA= alloca [20 x i8]
%.t9 = load i32, i32* %.i
%.t8= getelementptr [ 20 x i8 ] , [20 x i8]* %.a, i32 0 , i32 %.t9
%.t10 = load i8 , i8* %.t8
%.t11 = alloca i8
 store i8 %.t10, i8* %.t11
%.t12 = bitcast i8* %.t11 to  [20 x i8 ]*
%.t13 = load [20 x i8 ]  ,[20 x i8 ]* %.t12
 store [20 x i8 ] %.t13, [20 x i8 ]* %.charA

%.charB= alloca [20 x i8]
%.t15 = load i32, i32* %.i
%.t14= getelementptr [ 20 x i8 ] , [20 x i8]* %.b, i32 0 , i32 %.t15
%.t16 = load i8 , i8* %.t14
%.t17 = alloca i8
 store i8 %.t16, i8* %.t17
%.t18 = bitcast i8* %.t17 to  [20 x i8 ]*
%.t19 = load [20 x i8 ]  ,[20 x i8 ]* %.t18
 store [20 x i8 ] %.t19, [20 x i8 ]* %.charB
%.t21 = load [20 x i8], [20 x i8]* %.charA
%.t22 = load [20 x i8], [20 x i8]* %.charB
```

```
%.t23= load i8 , i8* %.t11
%.t24= load i8 , i8* %.t17
%.t20 = icmp ne i8 %.t23, %.t24
br i1 %.t20, label %label7, label %label8
label7:
ret i1 0
br label %label9
label8:
br label %label9
label9:
%.t25 = load i32, i32* %.i
%.t26 = add i32 %.t25, 1
store i32 %.t26, i32* %.i
%.t27 = load i32, i32* %.i
br label %label4
label6:
ret i1 1
}
```

- BuiltinMod this is a function that is used to calculate the modules

4.5.4 Ir Code optimization

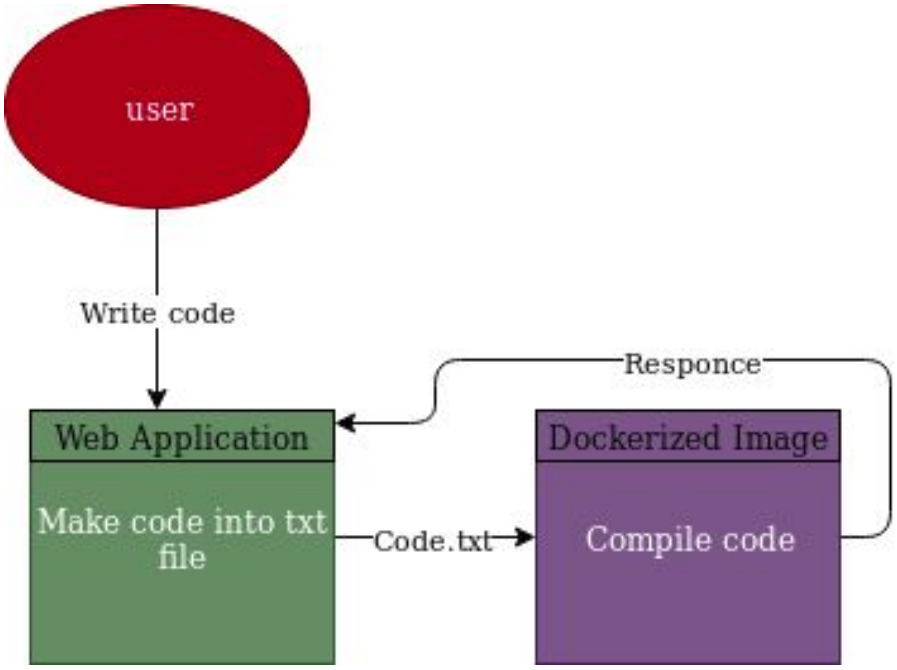
For code optimizer to optimize well we attempted to follow the guidance given to us by llvm documentation

1. Avoid high in-degree basic blocks (e.g. basic blocks with dozens or hundreds of predecessors). Among other issues, the register allocator is known to perform badly with confronted with such structures
2. All allocas at the entry of basic blocks The SROA (Scalar Replacement Of Aggregates) and Mem2Reg passes only attempt to eliminate alloca instructions that are in the entry basic block. Given SSA is the canonical form expected by much of the optimizer; if allocas can not be eliminated by Mem2Reg or SROA, the optimizer is likely to be less effective than it could be.
3. Avoid loads and stores of large aggregate type

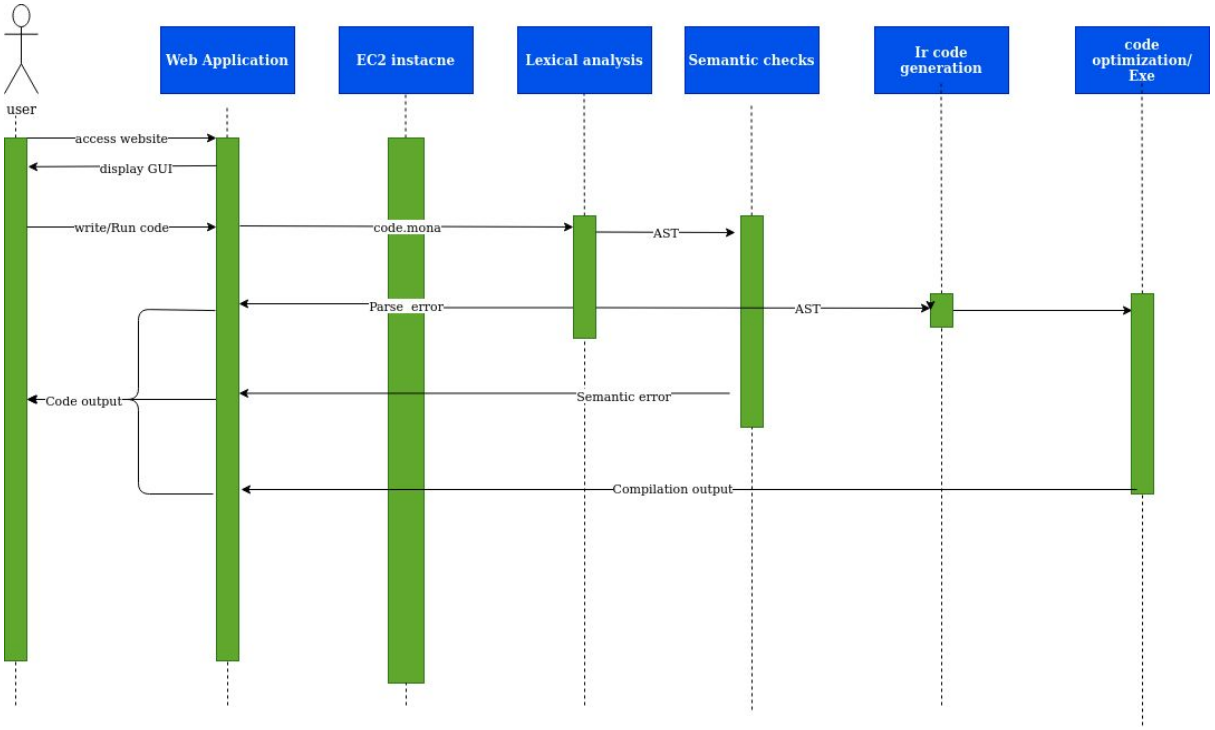
Exe file. We use llvm to run this file the result of the compilation will be collected by the flask web app and displayed for the users

5 High level design

5.1 High level data flow diagram



5.2 Sequence diagram of interaction



Above diagram demonstrates a user interaction with our system

1. User accesses web appl
2. The web application displays the ui
3. The user writes code and presses compile
4. The code file is sent from the webapp to the ec2 instance
5. The code file is then compiled using our compiler
    a. The code file is parsed and an AST is generated if parsing is successful
    b. If parse is not successful the parser will send the parse error to the front end
    c. The semantic checks are performed on the abstract syntax tree.
    d. If errors arise from the semantic check the compiler sends the semantic errors to the web app
    e. The llvm ir code is generated from the abstract syntax tree
    f. IR code file is optimised and compiled
    g. Results are sent back to Web App

6. Problems solved

Below you will find a list of problems solved

1. Created and developed Grammar for a fully working programing language that met user needs
2. Solved multiple ll(1) parse problems on the grammar to create an ll(1) parser with no additional lookaheads
3. Generated a complex abstract syntax tree from a complex grammar that allowed us to traverse through it and perform semantic checks and generate ir code
4. Designed multiple syntax rules
5. Traversed the abstract syntax tree and generated lower level IR code from a high level programming language
    a. Worked directly with memory registers and allocated and stored values in memory.
    b. lvm IR supports static single assignment, SSA requires that each variable is assigned exactly once, and every variable is defined before it is used. Very complex task to translate high level code int such low level code
6. Had to meet a strict llvm code structure for code to be optimised
7. Deployed a Web app on amazon ec2 instance
8. Integrated the flask web application with compiler

7. Results.

1. We received positive feedback from our user testing (tetsting_manual)

2. We have learned so much about programming throughout this project on how programming languages work and on what happens in the background
3. We have successfully built a fully working compiler web application

8.Future work.

We plan on re-engineering our code and developing a more complex compiler and deploy it for the future. The knowledge we have learned from designing and testing Mona will allow us to develop a much better compiler with more grammar and more functionality.

9.Refrences

1. https://crossbrowsertesting.com/blog/development/best-programming-language-to-learn-first/
2. https://www.daxx.com/blog/development-trends/number-software-developers-world
3. https://apps.dtic.mil/dtic/tr/fulltext/u2/773391.pdf
4. https://medium.com/@trungluongquang/java-and-python-which-is-better-to-learn-first-and-why-7f4cf5618a8e
5. https://pythoninstitute.org/what-is-python/
6. https://www.tandfonline.com/doi/full/10.11120/ital.2004.03020004
7. https://www.computing.dcu.ie/~davids/courses/CA4003/CA4003.html
8. https://llvm.org/docs/LangRef.html