# A Comprehensive Guide to Compiler Development: A Hybrid Python-Java Approach with Cloud Accessibility

Abstract

Table of contents

1. Introduction
2. Aims and Objectives
3. Background/Literature review
4. Methodology and Methodology
5. Requirements specification and Design
6. Implementation/Iterations
7. Testing and evaluation
8. Results/Finding and Discussion
9. Conclusions/Future goals
10. References
11. Glossary
12. Appendices

Key Terms:

* Abstract syntax tree (AST) : A tree like representation of which shows the first step of analysis after token generation.
* Lexical analysis: The process of splitting high-level code into predefined functions and character.
* Lexemes: The characters generated during Lexical analysis
* Semantic analysis: The next stage that adds semantic information (in my project checks Node information) to the syntax tree.
* Intermediate code generation/ three address code: An intermediate stage in which the high-level code processing is displayed.
* Assembly code generation: Intermediate code converted into readable machine code.
* Composite design pattern: A programming design pattern that represents class hierarchies.
* Kubernetes: A platform feature that manages containerized programs.
* Docker: A platform that allows creating, deploying and running programs inside “Containers”.

**1. Introduction**

In my proposal, I set out a bridge the educational gap between people with a formal educational background in technology and other users by going through the various stages a compiler takes high-level code through. One of the main goals of this project is increases users access to compilers concepts and functions using a combination of Python’s straightforward syntax and linking the application to a webpage which would utilize cloud practices for traffic management.

To understand the importance of this project, a compiler is crucial for software development and as Krebs and Schmitz(2014) point out in their abstract “Many programmers live in happy ignorance of their compilers’ internal workings”. While this compiler did not have all the features and power of a commercial compiler it follows a similar pipeline referenced in Aho research paper (Compilers: Principles, Techniques, and Tools).

In pursuing the goal of understanding compiler development, I have researched the link between the compiler and the machine. The compiler's last job is Assembly and linking, this process is essential to understanding the compiler’s role in translating high-level into machine executable code. David A. Patterson and John L. Hennessy explain in their book “Computer Organization and Design: The Hardware/Software Interface: Understanding the design of a computer's architecture is essential for understanding how high-level programs are translated into machine code and executed by the hardware.”

This project expands on the assembly code process and pipeline, this explanation will focus on how NASM (Netwide Assembler) will manage the various registers as well as function calls. The biggest challenge was researching the process and the need for converting a numerical value to a string so the machine can understand and print it. To fully explain this process demonstrates my understanding of both high and low-level code as well as the steps in between.

**2. Aims and Objectives**

In this section, I will expand on the goals and objectives section of my proposal. Each objective will be judged for its achievement, challenges, and whether it remained in the final draft. This process will summarise the steps to achieve the goal and a few solutions utilized, as well as a summary of the differences in the iteration regarding that goal. In doing so, I aim to start helping people understand the developmental life cycle of compiler development by highlighting my progress and through the critical reviews establish a successful plan for future iterations.

**Main objectives:**

* **Primary goal**: The primary goal of this project is the development of a toy compiler that shows can parse a single string line, while highlighting the inner layers of a basic compiler. (Completed)
  + **Achievement**:
    - The compiler can successfully parse a single string line and show the process of compilation according to Aho Lam and Sethi Ullman design in Compilers principles, techniques & tools with the final stage being a simplified demonstration of what the code generation, so from tokens to TAC (three-address code format) and finally assembly code as shown below:

Mov R1, world

PRINT R1

here is a snippet of the assembly code as shown the string was split into the address R1 and then finally the function print is used on R1.

* + **Challenges:**
    - One of the problems was my initial implementation of the lexical analysis method as originally my thought (prior to research) was splitting a string and taking out the function before passing the list onto the next stage.
    - One of the biggest challenges for this goal was the parser function and the implementation, I wanted to simply print out what the abstract syntax tree would look like for a simple “print ‘hello world’” call.
  + **Solution:**
    - After extensive research, I changed my implementation to splitting the source code into the official format called lexemes (token-name = LITERAL, value =”hello world”) as Token objects which must adhere to the Enums format to be parsed in the parser class. The original implementation might have satisfied a basic project but for a MSc dissertation project should demonstrate that level of education, so following the structure of a traditional compiler is the first step.
    - The next problem did not allow responsive usage, so I changed the format to composite design pattern this is to enhance the iterative approach this project focuses on. The benefits of the composite design pattern are flexibility and scalability, one of the goals of this project is to encapsulate all the knowledge I have gained throughout my course one of the main practices we focused on in cloud computing are those two features so with each node being individual and not relying on other nodes. The original implementation would not allow users to test the edges of the listed available functions which limits the users learning to most coders first code execution( print “hello world”).

**Iteration differences**

* **Original implementation:** split a string into a basic list for token generation.
* **Revised implementation:** Used design pattern which focuses on scalability as well as proper lexical analysis for token generation.

**Future goals**

* **Complex syntax (16/07/24):** Currently the function calls require set syntax such as “print “- with space required before the string.
  + **Solutions under consideration:** Thinking of using a for loop which cycles through the Enum and then generates the lexemes when the function is used is matches the listed functions.
  + **Solution:** The idea of using regex was correct, using the site regex101 to help practice the correct implementation needed for the correct split so its capable of recognizing print and the quotation mark position does not matter.

**Purpose:**

**Proposal**: The intended future impact of this compiler is that it should be an easy bridge in between both java and python as it will eventually have type casting and other features that align with java’s coding standard.

**Report**: The purpose has not changed but has been expanded as it would give users a basic overview of the stages the source program will go through on the web page using a digestible format which will be expanded upon throughout the report.

**List of Goals Future/Present**

* **Goal 1**: implement functions that allow for the parsing of python like code.

**Objective**: Follow python coding standard when writing conditions for code to be able to compile as well as for function calls.

**Actions**:

* **Research python syntax:** Study python syntax to understand basic functions. {May 2024}
* **Edit the parser file**: currently using Enums and a switch case to hold the function calls. {May 2024}-Complete:{22/07/24}
* **Testing:** test for various scenarios {July 2024}

**Technology needed:** Python, Java, JUnit

**Solution:** This was the first

* **Goal 2:** Build a web application that hosts the compiler that follows DevOps principles.

**Objective:** Build a basic webpage that will host the compiler managed by Kubernetes load balancers as well as a functioning yaml file that will retrieve new Docker images to show case CICD.

**Actions:**

* **Develop a web interface:** design the webpage using node.js for backend and react for frontend where the code can be input {June 2024}
* **Develop a CICD pipeline:** create a yaml file that is connected to docker account that will allow me to change image being used for seamless deployment. {June 2024}

**Technology needed:** Node.js, React

* **Goal 3:** increase functionality to allow for basic error handling that would also give users advice on what is wrong with their commands.

**Objective:** Enhance user experience to allow for error handling messages to be informative to improve learning experience.

**Actions:**

* **Create user friendly interface:** Implement functions that will pick up “bad code”. {August 2024}
* **Goal 4:** conduct user tests using participants to gather feedback on what needs to be improved or removed.

**3. Background/Literature Review**

* Summary of existing research and literature relevant to your project.
* Identification of gaps in the current knowledge.

**4. Methodology and Methods**

The methodology will outline the process taken to develop the compiler and the web application. This project aims to give individuals new to programming access to a platform that will allow them to learn the fundamentals of coding, through demonstrating the process a compiler takes high-level code and transforms it to executable machine language. This process also involves a web application developed as a user interface, deployed using GCP’s scalable cloud infrastructure.

This is achieved through a combination of various developmental and design methodologies. This project has several stages and sections which includes, the compiler development, web development, and the cloud deployment. The sections and design choices were selected to ensure that each feature is robust , maintainable and allows further development.

The methodology provides a detailed description of the various stages in which all components of the project were planned, designed, developed, and eventually deployed. It will also discuss the solutions used to overcome the various design challenges. The aim for this section is to not only provide insight into the developmental procedure but also provide a valuable reference for similar future programs.

**Project Planning**

The project planning section discusses the steps taking that ensured the success of the project as well as how plans developed over time. This process involved using both physically(see fig 1) and electronically drawing the project outline, setting timeline (in the proposal), identifying potential risk and the mitigation strategies (fig 2). The planning was essentially as having the process logged allowed for features to be tracked and all ideas for future iterations to be documented, allowing for adequate time allocation.

**Objectives**

After selecting this project, the first objective was to create a toy compiler that was able to parse a single string line, this eventually developed into integrating the program to a web interface which will be managed by Kubernetes. The present goals are:

1. **Compiler**: The focus of this goal was the implementation of the stages lexical analysis, syntax analysis, semantic analysis, intermediate code generation, and assembly code generation. Bergmann(2007) states, "In compiler design, the systematic progression from lexical analysis to code generation is essential to ensure the correctness and efficiency of the translation process. Each stage builds upon the previous one, transforming the source code step-by-step into a form that can be executed by a machine. This structured approach not only aids in debugging and optimization but also provides a clear framework for understanding and implementing compiler functionality." This was achieved with rigorous testing that focused on the integration of previous stages.
2. **Web applications**: This goal was to create a web application that would use the maven snapshot of the compilers program, allowing users to send high-level code and receive the output demonstrating the compilation process.
3. **Cloud deployments:** The web application is deployed on the scalable cloud infrastructure GCP to increase availability and accessibility while ensuring performance and reliability.

A paper with writing on it

Description automatically generated

Fig 1: A rough sketch of the program after the first sprint had officially started.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Risk Description | Likelihood | Impact | Mitigation strategy | Status |
| High web app traffic (low availability) | High | Medium | use Load balancers make it scalable with Kuberenetees | Not started (01/08.24) |
| Security breach (Data protection) | Medium | High | Use encryption modules to hash users’ data | Not started (01/08/24) |
| User engagement | Medium | Medium | Have a large test group and listen to user feedback | Ongoing (01/08/24) |
| Error in compilers compilation process | Medium | High | Have a comprehensive testing process and code reviews | Ongoing (01/08/24) |

Fig 2 table containing risk, mitigation strategy, likelihood, impact, and status.

**TimeLine**

This project is separated into various sections, each with set timelines, milestones, and deadlines. Which will discuss if they were achieved, met the deadline, or were removed.

**Stage 1 Research and Requirement gathering (1.5 weeks complete)**

* **Understanding compilers theory and architecture**: this was essential as my first lexical analysis method was very wrong, so my research rectified that.
* **Defining functional and non-functional requirements:** allowed logging of various necessities for this project.

**Stage 2 Compiler development (4 weeks original goal “Parse single string line”)**

* **Lexical analysis class:**  The original goal was completed very soon after research and requirement were completed, the first requirement was completed quickly.
* **Syntax analysis and AST generation:** The first class was completed the initial milestone of AST generation was successful, but it was generated in the parser class and not the to string of the node class.
* **Semantic analysis:** The semantic analysis was not in the original implementation but was added later after a single string line function was parsed.
* **Intermediate code generation:** The intermediate code generation method for the first function was completed quickly as well as this project did take inspiration from interpreter from another coursework, so it returned a list of instructions.
* **Assembly code generation:** This function was let out of the first step as the research was separated from the compilers function that I read about until I read Levine, J. R. (2000) paper, which was added after a few iterations.

**Stage 3 Web application development (3 Weeks Original goal “make a simple web interface”)**

* **Backend server**: The first iteration of the backend did not use routes; everything ran through the server.js. Using the Child\_process module, the maven snapshot was linked to the web application.
* **Frontend server:** The first success milestone for the frontend was that it received the high-end code and sent it back to the backend is JSON format to be processed by the compiler application.

**Stage 4 cloud deployment (2 weeks)**

**Development Process**

The development process followed Agile practices (Jira tracking Fig 3), which focused on iterative development adhering to user feedback. This approach was chosen with the goal of catering to people new to programming and their experience with the application.

Technology and Tools stack

* Java: Language used for compiler development.
* Node.js: Backend server primary language.
  + Express: Module management.
  + Nodemon: Allows editing while server is running.
  + Child\_process: Used to link java snapshot.
  + Path: Used to locate target folder.
  + fs: Generate output files and store them.
  + UUID: Used to generate unique IDs for the output files.
* React: Used for frontend of web interface.
  + Axios was used for backend intergration
* GCP: Used for program deployment.
* Docker: For containerizing the program.
* Kubernetes: Used for deployment management
  + Load balancer

A screenshot of a computer

Description automatically generated

Fig 3 Jira board (02/08/24) shows the “Issues” that are either on the To Do list , In progress list, and the Done list

**User feedback**

Due to my current job as a coding teacher at a summer school application I have had access to a range of people, from avid coders to people who are just starting in the technology industry I have logged their feedback along with what change occurred:

|  |  |  |
| --- | --- | --- |
| Review | Status | Change |
| Program must start again if I make a mistake | Complete | In the parser class I removed the throw exception and instead made it return the Unknown node which the Lexeme (token) saying its not a listed function call. |
| Wanted to make notes of progress | Complete | Using the modules fs and UUID I created an output page function which tracks the last 6 compilation results both in the web application to be accessed via links and in the application folder (all compilation results). |
| Not enough space for output/ writing was too small | Complete | In the web application increased the handwriting size but also developed the output page which is dedicated solely to the output. |
| I would like to be able to use it on my phone | Ongoing | This will be resolved in the next iteration. |

 **System Design**

* Overview of the system architecture.
* Design methodologies used (e.g., UML diagrams, flowcharts).
* Design of individual components (e.g., compiler stages, web interface, cloud deployment).

 **Development Process**

* Description of the development environment and tools used.
* Programming languages, frameworks, and libraries.
* Version control practices (e.g., Git workflow).
* Integration and testing strategies.

 **Compiler Development**

* Detailed process of developing the compiler.
  + **Lexical Analysis**: Tools and techniques used (e.g., regex, finite state machines).
  + **Syntax Analysis**: Parsing techniques (e.g., recursive descent, parser generators).
  + **Semantic Analysis**: Methods for semantic checks.
  + **Intermediate Code Generation**: Steps to convert AST to intermediate code.
  + **Assembly Code Generation**: Process for generating assembly code from intermediate code.
  + **Testing and Debugging**: Strategies and tools used.

 **Webpage Development**

* Frontend development process.
  + Frameworks and libraries used (e.g., React, CSS).
  + User interface design and development.
* Backend development process.
  + Server setup and configuration (e.g., Node.js, Express).
  + API development and integration with the compiler.
* Testing and debugging methods for the web application.

 **Cloud Deployment**

* Process of deploying the web application to the cloud.
  + Selection of cloud platform (e.g., GCP).
  + Configuration of virtual machines.
  + Containerization using Docker.
  + Orchestration using Kubernetes.
  + Load balancing setup.
* Security measures implemented.

 **Challenges and Solutions**

* Description of major challenges faced during the project.
* Solutions and strategies used to overcome these challenges.

 **Evaluation and Testing**

* Methods used to evaluate the project.
* Testing strategies and tools.
* Performance testing and results.
* Usability testing and feedback.

 **Future Work**

* Potential improvements and enhancements.
* Plans for future iterations of the project.

 **Conclusion**

* Summary of the methodology.
* Reflection on the effectiveness of the methods used.

 **References**

* Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: Principles, Techniques, and Tools (2nd ed.). Pearson.
* Bergmann, R. (2016). Understanding Compilers and Their Role in Computing. Springer.
* Cooper, K. D., & Torczon, L. (2011). Engineering a Compiler (2nd ed.). Elsevier.
* Levine, J. R. (2000). Linkers and Loaders. Morgan Kaufmann.
* Patterson, D. A., & Hennessy, J. L. (2013). Computer Organization and Design: The Hardware/Software Interface (5th ed.). Morgan Kaufmann.

**5. Requirements Specification and Design**

**Introduction**

This section will focus on the various features and their development stack as well as their necessity to achieving the projects aim. I will focus on the idea behind the feature the developmental resources used for my current iteration and aims for future iterations.

**Functional requirement**

* **Compiler:** The compiler has several stages which will be briefly explored in the following sections along with the respective outputs.
  + **Development Stack:** Java
  + **Lexical analysis:** Using a combinations of regex formulas and if checks generate tokens and links them to their functions. Including their value and identifier:
    - Token{key=VARIABLE, value='dog'}.
  + **Syntax analysis:** Takes the tokens generated in the lexical analysis and parse into an AST (Abstract Syntax Tree) using the toString of each of the nodes. The AST would demonstrate the hierarchical structure of the high-level code:
    - PrintStatement : void
    - |
    - +-- Literal("hello world") : String
  + **Semantic analysis:** This checks the AST for semantic errors using the nodes so for example a print statement function call should be a literal node:
    - Checking expression: Literal("hello world") : String Expression is a LiteralNode.
  + **Intermediate code:** This level takes in the AST from the syntax analysis level and using a series of checks it would return a list of instructions that manipulates the Enum of registers for value manipulation.:
    - LOAD\_STRING R0, ""hello world""
    - CALL print, R0.
  + **Assembly code generation:** Using the AST it generates an assembly code string as well as creating an asm file, for binary operations it uses a series of switch cases on the operator to pick the specific function when generating the string. The method is a series of checks to select the right String generation method:
    - \_start:
    - mov eax, message
    - push eax
    - call print
* **Webpage**: The webpage is an interface that users will be able to send their code to be parsed by the maven snapshot of the latest iteration of the compiler and receive an output with as of(31/07/24) generates runnable assembly code.
  + **User interface**
    - **Compile Page:** The goal of this stage is to not only give users the ability to write code but also shows them the next steps if they wanted to create and run their own executable.
      * **Backend:** Node.js server that has the packages managed by the Express module, the compiler program is linked to the backend using the module Child\_process and sent to the frontend using the module Axios.
      * **Frontend**: A react application that uses some CSS code for styling.
    - **Output page:** Due to the output size an output page was created that displays the output of the parsed code but also using the modules uuid and fs creates a unique ID and weblink that allows users to revisit old outputs.
    - **Glossary page:** The glossary page was created so the main functionality will be known to users. The page contains the functionality and the allowed syntax.
* **Cloud deployment:** To demonstrate cloud principles such as scalability and accessibility, the goal for the webpage is to be deployed on the cloud platform GCP.
  + **Virtual machine:** The virtual machine is used to create and push a docker image to Dockerhub.
  + **Docker:** The web application will be containerized so there is consistency across different environments.
  + **Management:**
    - **Kubernetes:** deploy docker container to be managed by clusters so the project can be easily scaled for demand.
    - **Load balancing:** load balancers ensure that the cloud principle availability even during times of heavy traffic on the web page.

**Non-Functional Requirements**

* **Performance:** The program should be allow users to send high-level code through the webpage to be compiled in the compiler program and returned to the webpage.
* **Scalability**: The program should handle increased demand by simply increasing the nodes in Kubernetes and ensuring availability with load balancers
* **Usability:** The webpage should be user-friendly especially for beginner programmers with the resources (Glossary).
* **Security**: Using modules like bycrypt.js Users sign in data is encrypted before being sent to the database in the event of breach.

**6. Implementation/Iterations**

* Step-by-step explanation of how the project was implemented.
* Details of the iterative development process.
* Challenges faced and how they were overcome.
* Ideas -> for the tokenisation use Nltk module.

##### Lexicon development

Lexical analysis is the first stage the source code goes through, the purpose is to analyse the code and split it into lexemes (tokens) ready to be sent to the parser class. My compiler closely follows Aho Lams implementation at least in appearance "In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation." (Aho, Sethi, & Ullman, 1986).

In this project, my lexical analysis has been organised to take in a static variable or “abstract symbol” which would be a key element in AST (abstract tree generate) generation, followed by the value that is stored for the future stages of compilation. My compiler went through several developmental stages initially, it was a simple string splitter, to using regex and a series of checks for dynamic token generation.

**Supporting classes**

Originally, there was no TokenCheck or Token class. However, after researching lexical analysis my compiler took inspiration from the various projects I have participated in or researched. To properly generate tokens, I decided to use an OOP(object-oriented programming) practice called encapsulation for the Token class which stored the data from the Enum.

Initially, there was only the value variable in the TokenCheck class which would be the second parameter or value in the Token object that generated but after a few iterations it progressed to using regex.

After the introduction of regex, development of the various stages progressed quickly but, there was a problem when it came to the print variable operation as all the regex formulas, I attempted would also result in the value with the last being Token{key=PRINT\_VAR, value='print variable'} – this would be an incorrect return as it would require further processing before the parser so I added a syntax rule for this which is when a process is complete it must end with an “;” before the next function call to enforce this I created two static method, first is a boolean method containsCheck with the parameter String input which just return a check if there is a “;” present this would be used in the lineChecker with the parameter String input as well this splits the input on the “;” cleans the whitespace and then returns the list. After that method was tested it was integrated into the lexicon class its introduction resulted in errors in previously established methods. Eventually, the lexicon was changed to run through a method that generates a list of “lines” (code separated by “;”). Below is the desired output of the print variables token.

dog = "fluffy"; print dog

Token{key=VARIABLE, value='dog'}

Token{key=VAR\_ASSIGN, value='='}

Token{key=LITERAL, value='fluffy'}

Token{key=PRINT, value='print'}

Token{key=LITERAL, value='dog'} –will be edited in future iterations, so it is processed as a VARIABLE token.

A screenshot of a computer

Description automatically generated-Diagram of Unified Modelling Language (UML) with classes:

* Lexicon
* TokenCheck
* Token
* OperationCheck
* VarOperationCheck
* VarOperationCheckExtended

As previously stated, with the introduction of regex and the LineCheck method more complex compiler features have been made possible but a lot of binary functions will only have a different operator meaning the checks will be the same apart from one token so to reduce redundancy external classes that focus on various switch cases like

public static TokenCheck getVarKey(TokenCheck type) {  
 switch (type) {  
 case *NUM\_VAR\_ADD*:  
 return TokenCheck.*NUM\_VAR\_ADD*;  
 case *NUM\_VAR\_SUB*:  
 return TokenCheck.*NUM\_VAR\_SUB*;  
 case *NUM\_VAR\_MUL*:  
 return TokenCheck.*NUM\_VAR\_MUL*;  
 case *NUM\_VAR\_DIV*:  
 return TokenCheck.*NUM\_VAR\_ADD*;  
 default:  
 throw new IllegalArgumentException("Unknown operation type");  
 }  
}

this extract shows one of the classes designed to not only assign the operator based on the regex formula that passes the check but also return the correct tokens, this approach has not only made the Lexicon class more readable but also made it easier to add different binary functions such as modular (will be in future iterations).

**Initial implementation**

The initial implementation was not only wrong but extremely basic, it had no error handling capability as well as it stuck to a very rigid format so more complex syntax would either return an error or something completely different.

public void splitter(){

String[] token = input.split(" ");

Tokens = Arrays.asList(words);

-this method is extremely basic and as shown by the previous information put forward is also incorrect.

This implementation was an idea of what I thought a compiler was when I first picked the kernel. It

**Second Iteration**

My first step in addressing these issues was to create two classes:

* the first class was the TokenCheck class which hosted the Enum containing the functions as well as the data types (as of 19/07/24 only Literal as I am testing Numerical) and only one variable called function which allowed the Enum to have a string value.
* The next step was making the Token class which had the variables key (which is a TokenCheck reference variable) and value as well as setter and getters as the purpose of the two classes was the store lexemes in this format (token-name = LITERAL, value =”hello world”) as Token objects.

This process required multiple steps, firstly the input is split using regex ("\\s+",2) – which splits on whitespace, but it is limited to 1 split (two elements) next there are a series of checks :

boolean flag = false;  
 for (TokenCheck type : TokenCheck.values()) {  
 if (words[0].equals(type.getKeyword()) && StringChecker.quoteCheckLexer(words[1])) {  
 tokens.add(new Token(type, String.valueOf(type)));  
flag = true;  
 break;  
 }  
 }  
 if(flag){  
 tokens.add(new Token(TokenCheck.*LITERAL*,words[1]));

As shown in this code extract an enhanced for loop is done to check if the first value is a function call and that even though semantic checks are the parsers job, I added in a quotation mark checker once all those checks are passed two new token objects are made one for PRINT Enum value and the next for the LITERAL Enum value.

**Third Iteration**

**if (match.matches()) {  
 flag = true;  
 String keyword = match.group(1);  
 tokens.add(new Token(type, keyword));  
 System.*out*.println(keyword);  
 if (match.groupCount() > 1 && match.group(2) != null && type.getKeyword().equals("print")) {  
 String literal = match.group(2);  
 tokens.add(new Token(TokenCheck.*LITERAL*, literal));  
 }  
 break;  
 }  
} catch (Exception e) {  
 System.*out*.println("Failed to tokenize: " + e.getMessage());  
 tokens.add(new Token(TokenCheck.*UNKNOWN*, "Failed to compile due to code: " + input));**

For the print function the regex formula is the correct regex formula ("^(print)[\\s\*\"([^\"]\*)\](file:///\\s*\%22(%5b%5e\%22%5d*)\)"")

**Regex pattern**

* ^(print) checks that print is at the beginning of the command.
* [\\s\*](file:///\\s*) checks for whitespace but also ignores it if there is none.
* \"([^\"]\*)\ checks that the LITERAL is inside the double quotation marks.

This formula was later moved to the Enum to follow scalability practices, by removing the preset print function call it leaves the splitter method can now be expanded to call various function calls using the Enum and regex, so it also is case in sensitive.

**Challenges and solutions**

* challenge:
  + due to being a responsive design it would require extensive checks to stop things such an index out of bounds due to things such as

The next step was moving it to the Enum so using the enhanced for loop to iterate through the Enums pattern until it is matched to a function. The LITERAL Enum was easy enough as it just copied the string regex. This process demonstrates scalability by removing the preset print function inside the lexical analysis method and using regex to make the compiler more responsive to various method calls. This process did cause problems such as having to refactor the entire project but the main problem that arose was that when testing edge cases, the first error was that since I needlessly added the regex formula for the Literal it would pass through as a Literal token instead of being logged as an Unknown token. To amend this, I used a try catch block which would catch the index out bound error as well as any other compilation errors as an unknown token to stop errors from interrupting the web browser. The code below demonstrates the main part of lexical analysis implementation it displays the checks the second one checking for the print function as more functions are added more checks will be added.

**Newest Iteration**

##### Code Generation

This is a note for future me - originally your binary operation between two variables didn’t work so you tried loading the values into a different register by setting it to an object but that didn’t work so instead you settled for using the store instruction and getting the variable value to progress -end of note

##### Assembly code breakdown

Once the compilation process has completed using the node AST a static method would generate the runnable assembly code. As explain in the book Linkers and Loaders John R. Levine states "Generating assembly code is a meticulous process where each high-level instruction is broken down into the specific steps needed to achieve the desired operation on the hardware. The goal is to produce code that is both efficient and correct."

The output below is as of (27/07/24) this is my most advanced functions so the operation “a = 12; b = 55; c = a +b; print c”.

Output for binaryAssignmentNode :

section .data  
a dd 12 ; Define variable a with value 12  
b dd 55 ; Define variable b with value 55  
c dd 0 ; The total variable C is defined here  
buffer db 'Result: ', 0 ; buffer will print the result(printing C value)  
*buffer\_len* equ $ - buffer; Automatically calculate length of the buffer  
num\_str db '0000000000', 0; String storage of number  
*num\_len* equ 10 calculation of string numbers  
  
section .text  
global \_start  
\_start:  
 mov eax, [a] ; Load the value of variable “a” into register eax  
 mov ebx, [b] ; Load the value of variable “b” into register ebx  
 add eax, ebx ; addition (eax = eax + ebx)-function call   
 mov [c], eax ; save result result (eax) into c  
  
 mov ecx, *num\_len* ; Prepare to convert the result to a string  
 mov esi, c ; Load the address of “c”  
 mov eax, [esi] ; move value of c into eax  
 mov ebx, 10 ; Prep to divide by 10 (String conversion)  
  
convert\_loop:  
 xor edx, edx ; empty the edx register  
 div ebx ; Divide eax by 10  
 add dl, '0' ; digit to ASCII (American character encoding) standard  
 dec ecx ; reverse the index  
 mov [num\_str + ecx], dl; Store the ASCII in the string  
 test eax, eax ; check if eax is zero  
 jnz convert\_loop ; loop if check fails   
  
 mov eax, 4 ; to call sys\_write (printer)  
 mov ebx, 1 ;   
 mov edx, *num\_len* ; Num bytes to write  
 mov ecx, num\_str ; Address of the string to print  
 int 0x80 ;   
  
 mov eax, 1 ; to call sys\_exit  
 xor ebx, ebx ; Exit code 0  
 int 0x80 ;

The code is annotated explaining what each instruction does. To reduce code redundancy, a switch case is used for the different operators:

switch (operation) {  
 case "+":  
 assemblyCode.append(" add eax, ").append(((LiteralNode) right).getValue()).append("\n");  
 break;  
 case "-":  
 assemblyCode.append(" sub eax, ").append(((LiteralNode) right).getValue()).append("\n");  
 break;  
 case "\*":  
 assemblyCode.append(" imul eax, ").append(((LiteralNode) right).getValue()).append("\n");  
 break;  
 case "/":  
 assemblyCode.append(" xor edx, edx\n");  
 assemblyCode.append(" mov ebx, ").append(((LiteralNode) right).getValue()).append("\n");  
 assemblyCode.append(" idiv ebx\n");  
 break;  
}

This extract shows the differences between each operator. As shown all the operators are handled in a similar way except division “/” operator which involves a more instructions due to how division is handled by x86 assemblers:

* xor edx, edx - clears register by XORing it by itself
  + since registers work with binary values edx = XOR 1101 | 1101 = 0000
* mov ebx, LiterNode (value)
* idiv ebx
* **Before division**
* eax = 10
* ebx = 3
* edx =0 cleared by xor edx, edx
* **After division**
* eax = 3 result
* edx = 1 remainder

##### Integration Implementation

One of the checkmarks for this project is the maven project is built using the command mvn clean package to create a snapshot which would then be linked to the web application with the goal of receiving a JSON message which would then be transformed to a string and sent to the snapshot for processing.

##### Backend

* Setup (Backend):
  + Development started with the backend since the first iteration did not have any functions for the home and glossary page so the only endpoint in the app.js file was the compile page.
* Compiler events:
  + Using the compiler variable the first operation was the stdout which starts when data is sent, when this event is triggered the data (variable generated) is converted to a string and will append the output variable that has already been initialised. This is verified using the console.log method to show there was a connection made.
  + The next event is stderr this is the error handling event the code is data sent to the frontend is a different from the stdout event as it only returns the standard error message 'Compilation failed'.
  + The compiler.on(‘close’, (code)) event is executed when all events have ended. First, there is a check (code !== 0) for non-zero code (error during compilation process). Once completed the accumulated output string will be sent using res.json. As shown in the method below:

compiler.on('close', (code) => {

if (code !== 0) {

console.error(`Compiler exited with code ${code}`)

res.status(500).json({ error: 'Compilation failed' })

} else {

const formattedOutput = output.replace(/;\s/, ';\n')

res.json({ output: formattedOutput })

}

})-at the end you can see the format is being edited for the user’s sake I used regex (/;\s/, ';\n’)- the function of this is at the occurrence of “;” the output will be pushed to a new line.

* + Finally, once all events have passed the stdin.write event is triggered, the parameters passed are input(data generated) and ‘\nexit\n’ this just sends the exit command to the child\_process to signal the shutdown of the while loop so the processing stops. Ended with the stdin.end which just closes the input stream as no more data is sent.
* For readability routers were used as there was a lot of code for the compiler page functionality it now has its own designated file. Routers were needed before moving onto the next web app iteration as I wanted add log in features to demonstrate features like OAuthv2 (only users would be able to access the compiler features as the compilers features develop more it would be better for users to be able to see their past tokens (generated from the compiler) and the stages it went through.

##### Errors and troubleshooting

* Initial implementation of full stack web page
  + Error:
    - The first implementation was HTML so when I upgraded to React it was difficult to establish connection- originally thought it would work if I ran them both on the same port (first official react project)
  + Solution:
    - Used cors and axios -axios implementation in the frontend - axios.post(<http://localhost:3001/compiler/compile>) this was to receive the response(data generated from compiler) and post it back to the frontend.
* The first error was being unable to connect the server to the snapshot of the maven project.
  + Errors:
    - Only copied the snapshot into the desired folder (researched the procedure and fixed that).
    - While using the entire target folder would improve the percentage of successful compilations there would be the occasional error message – “Compilation Error: Error: Unable to access jarfile WebApp\server\target\CompilerWebCloud-1.0-SNAPSHOT.jar”.
  + Solutions:
    - The first solution was instead of just copying the snapshot was to use the entire target folder.
    - Someone had a similar unable to access jarfile error on StackOverflow and suggested using the path import and splitting the string into a list on the “/” then joining it into a variable – this solved the access problem.
* The next issue was that while the compilation was a success I there was no change to the webpage.
  + Errors:
    - The initial error was that I could not validate whether the JSON data was being or is the problem was with the application integration.
    - After confirming the code was being sent to the java application the output was not being sent to the frontend. Original implementation:

let output = ''

compiler.stdout.on('data', (data) => {

output += data.toString()

console.log(data.toString())

res.send({Output:output})

Since console.log-showed the output was data that had been generated (it did not work as it resulted in header error)

* + Solutions:
    - The first one required some testing (first the code below)

const { text } = req.body

console.log(text)

res.json({ message: `Text is here: ${text}` })

This extract was used to send the requested code back to the frontend and terminal which would remove the idea that the connection between the frontend and backend was the problem.

* Now onto processing the code this took a bit longer- the original implementation would use the stdout event to send the data to the frontend and java application in the same clause this would not work as the stdout event is triggered numerous times which goes against the res.send method triggering the header error. – this was resolved by using the compiler.on method which would only run once all events are completed stopping the header error

**7. Testing and Evaluation**

* Testing methods and results.
* Critical evaluation of the project outcomes.
* Discussion on how the software/system meets the requirements.

**8. Results/Findings and Discussion**

* Presentation of the key results and findings.
* In-depth discussion and analysis of the results.

**9. Conclusions/Future Goals**

* Summary of the findings.
* Recommendations for future work.
* Reflections on what was learned during the project.

**10. References**

Krebs, N. & Schmitz, L. (2014) 'Jaccie: A Java-based compiler–compiler for generating, visualizing and debugging compiler components', Science of Computer Programming, vol. 79, pp. 101-115.

Kundra, D. (2016) 'Application of Case-Based Teaching and Learning in Compiler Design Course', pp. 1-9. (Accessed: 4 May 2024).

Accomazzo, A., Lerner, A., Murray, N., Allsopp, C., Guttman, D., & McGinnis, T. (2017). Fullstack React: The Complete Guide to ReactJS and Friends. Technical advisor: Shoemaker, S. Fullstack.io.

Aho, A. V., Sethi, R. & Ullman, J. D. (1986). Compilers: Principles, Techniques, and Tools. Addison-Wesley.

Cooper, K. D. & Torczon, L. (2022). Engineering a Compiler. 2nd ed. Morgan Kaufmann.

Patterson, D.A. and Hennessy, J.L., 2014. Computer Organization and Design: The Hardware/Software Interface. 5th ed. San Francisco: Morgan Kaufmann.

Levine, J.R., 2000. Linkers and Loaders. San Francisco, CA: Morgan Kaufmann Publishers.

Bergmann, S., 2007. Compiler Design: Theory, Tools, and Examples. New York: Springer.

**11. Glossary**

**12. Appendices**

* User Manual
* Relevant code snippets
* Technical information