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

**1. Introduction**

* Background of the project.
* Importance and relevance of the project.
* Scope and objectives.

**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 short 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 didn’t 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}
* **Testing:** test for various scenarios {July 2024}

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

* **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/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**

* Detailed description of the methods and approaches used in the project.
* Tools and technologies employed.
* Explanation of why these methods were chosen.

**5. Requirements Specification and Design**

* Detailed requirements of the system/software.
* Design specifications and architecture.

**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 initial implementation would simply take in the string and split it by spaces and store it in the array Tokens:

public void splitter(){

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

Tokens = Arrays.asList(words);

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

**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.

**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.

**Newer 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 showcases the main part of lexical analysis implementation it showcases the checks the second one checking for the print function as more functions are added more checks will be added.

##### Integration Implementation

One of the checkmarks for this project is the maven project will be 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’s already been initialised. This would be 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.

**References**

* Complete list of references cited in the report, following the Harvard referencing system.

**Appendices**

* User Manual
* Relevant code snippets
* Technical information