UNIVERSITY OF HARGEISA

College of Computing and Information Technology

Department of Information Technology

A Somali Interpreted Multi-paradigm Programming Language

A project submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Information Technology

By

1. **Yasir Esse Ahmed Omer (2022959)**
2. **Mohamed Mohamoud Mohamed (2023009)**
3. **Hamse Abdillahi Muhumed Badil (2023013)**
4. **Hamda Herzi Nour (2023108)**

Supervised by

Dr. Aidarus Ibrahim

June 2024

DECLARATION

We declare that the thesis titled "A Somali Interpreted Multi-paradigm Programming Language" and all accompanying work are our own original efforts. We confirm that all sources consulted are properly acknowledged. This thesis has not been submitted for any other academic qualification, nor is it being published elsewhere. We adhere to the regulations on academic integrity and plagiarism. Additionally, we declare that we have written this thesis by ourselves without using any sources or methods without proper citation. All ideas from others and direct quotations are clearly marked.

1.	Yasir Esse Ahmed Omer	____________________
2.	Mohamed Mohamoud Mohamed	____________________
3.	Hamse Abdillahi Muhumed	____________________
4.	Hamda Herzi Nour	____________________

As the supervisor of the candidates' thesis, I certify that the above statements are true to the best of my knowledge.

Dr. Aidarus Ibrahim ____________________

Date: 11/07/2024

DEDICATION

To our dear parents, whose unwavering support, sacrifices, and endless encouragement have been our guiding light throughout our challenging academic journey. Your belief in us has fueled our determination to succeed, and we dedicate this achievement to your unwavering love and support.

ACKNOWLEDGEMENT

We extend our heartfelt gratitude to Dr. Aidarus Ibrahim, whose exceptional guidance, unwavering encouragement, and insightful feedback profoundly shaped every aspect of our thesis, making an indelible impact on our work. We are also deeply grateful to the University of Hargeisa for their continuous support and for providing us with the necessary resources and environment to pursue our research effectively. The invaluable contributions of the IT and computing faculty and staff played a pivotal role in our success, for which we are sincerely thankful.

CERTIFICATE OF COMPLETION

This is to certify that the following students.

2022959	Yasir Esse Ahmed Omer	____________________
2023009	Mohamed Mohamoud Mohamed	____________________
2023013	Hamse Abdillahi Muhumed	____________________
2023108	Hamda Herzi Nour	____________________

Have successfully completed their final year project titled:

A Somali Interpreted Multiparadigm Programming Language

In the partial fulfillment of the requirements of the Degree of Bachelor of Science in Information Technology for the academic year 2023-2024.

____

Dr. Aidarus Ibrahim

ABSTRACT

Non-English speakers often need help learning programming due to the dominance of English in programming languages and educational resources. This thesis addresses these challenges by developing a Somali-interpreted multi-paradigm programming language, referred to as Gob. it is Designed to provide a native-language programming environment for Somali speakers; this language facilitates more accessible access to programming education by reducing linguistic barriers.

This programming language evolves with user feedback. It integrates features from both Object-Oriented Programming (OOP) and Functional Programming (FP) paradigms, offering a flexible and comprehensive learning and development environment. Developed following Agile methodologies, Gob was iteratively refined based on user feedback, ensuring its adaptability and responsiveness to the needs of Somali speakers. Rigorous testing, including black-box and white-box methods, validated its functionality and usability, demonstrating its effectiveness in meeting the educational needs of Somali speakers.

The development of this Somali-interpreted language highlights the potential for localized programming languages to enhance learning outcomes for non-English speakers. By enabling Somali speakers to learn and practice programming concepts in their native language, the language contributes to a more inclusive and diverse global technology landscape. This project sets a precedent for similar initiatives aimed at linguistic inclusivity in technology, fostering greater accessibility and diversity.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Introduction

In the rapidly evolving landscape of technology, access to education and resources plays a pivotal role in shaping opportunities for individuals and communities (Ahmed, 2023). However, for the Somali community, like many others, language barriers present formidable obstacles on the path to technological advancement. The prevalence of programming languages predominantly written in foreign languages, coupled with limited resources available in Somali, creates a significant disparity in access to technology education.

Non-English programming languages have historically played a crucial role in making technology accessible to non-English speaking communities. For example, the Arabic programming language Kalimat was developed to cater to Arabic-speaking programmers (Al-Ajlan, 2015). Himawari, a Japanese language-based programming environment, allows for wider accessibility and adoption among Japanese speakers. Another example is the Chinese programming language Yabasic, which was developed to cater to Chinese-speaking programmers. These examples demonstrate the importance and impact of developing programming languages that cater to native languages.

As a response to this challenge, this study endeavors to bridge the gap by developing a programming language tailored to the Somali language. By doing so, it aims to empower individuals within the Somali community to overcome linguistic barriers, access technology education, and participate more fully in the digital era. Through this initiative, we seek to foster inclusivity, promote local innovation, and contribute to the global diversity of technological resources.

1.2 Problem Background

Language barriers present a significant obstacle for the non-English community's integration into the technological landscape. For non-English speaking youth with limited English proficiency (LEP), mastering programming languages such as Python, Java, and C++

poses a formidable challenge (Becker, 2019). He mentions that most LEP students face unique difficulties in understanding the technical vocabulary and syntax predominantly rooted in English. These linguistic hurdles, compounded by cultural differences in learning approaches and educational frameworks, further hindered their comprehension of programming concepts and limited their access to educational resources and effective engagement with technological learning tools (Adewusi, 2024).

1.3 Problem Statement

Within the Somali community, a significant hurdle exists for individuals seeking to enter the field of programming. This obstacle stems from the prevalence of English-based programming languages, creating a substantial language barrier for those who are not fluent in English. This barrier manifests in the difficulty of grasping technical terminology and core coding concepts presented in a foreign language. Furthermore, unlike initiatives undertaken for languages like Yoruba (Olatunji, E. K., Oladosu, J. B., Odejobi, O. A., & Olabiyisi, S. O., 2020), there is a critical absence of established programming resources specifically designed for the Somali language. This lack of Somali-tailored learning materials compounds the existing challenges faced by Somali speakers aspiring to become programmers. The combined effect of the language barrier and the dearth of Somali-specific resources creates a significant roadblock, potentially limiting access to valuable educational and professional opportunities within the tech industry for Somali speakers.

1.4 Purpose of The Project

1.4.1 General Objective

The general objective of this project is to develop a programming language that makes technology education accessible to the Somali community by addressing language barriers, thus empowering individuals to learn, apply, and innovate using modern programming techniques.

1.4.2 Specific Objectives

1. To study and analyze the historical development and impact of non-English programming languages.

2. To design and implement a Somali-interpreted multiparadigm programming language, incorporating functional programming principles and object-oriented design.
3. To test the developed Somali programming language for functionality, usability, and accuracy in computations.

1.5 Scope of The Project

1.5.1 Geographical Scope

The geographical scope of this project is global, targeting all Somali speakers who are interested in learning logic or are new to programming. The aim is to introduce them to basic to intermediate concepts, such as object-oriented programming.

1.5.2 Content Scope

This project will cover the development of a Turing-complete programming language tailored to the Somali language, ensuring inclusivity and accessibility for Somali speakers. The language design will encompass features that support both Object-Oriented Programming (OOP) and Functional Programming (FP) paradigms, providing versatility and flexibility for developers to choose their preferred programming style.

1.5.3 Time Scope

Within a tight two-month timeframe, the project will focus on crafting a programming language tailored to the Somali community. It will support Object-oriented programming (OOP) and Functional programming (FP) paradigms while producing essential documentation and educational materials.

1.6 Importance of The Project

- The study addresses the significant challenge of language barriers within the Somali community, enabling access to technology education by creating a programming language in Somali. This breakthrough opens doors for individuals who face linguistic obstacles, empowering them to pursue careers and opportunities in technology.
- By providing the Somali community with tools and resources to engage with technology in their native language, the study fosters a culture of local innovation. Also,

it opens the doors for the youth to understand the core fundamentals that make up programming. This localized approach has the potential to stimulate the development of technology solutions tailored to the specific needs and contexts of the Somali community, contributing to broader advancements in technology and innovation.

1.7 Organization of The Project

Chapter 1 introduces the context and motivation for the project, highlighting the technological barriers faced by the Somali community due to language constraints. It outlines the problem background, problem statement, the purpose of the project, objectives, scope, and the importance of the project.

Chapter 2 reviews existing literature relevant to the development of non-English programming languages. It explores challenges faced by non-English speakers in learning programming, previous efforts in creating non-English programming languages, and theoretical frameworks that support the project.

Chapter 3 describes the methodologies used in the development of the Somali Interpreted Multiparadigm Programming Language. It details the software engineering approaches, including Agile methodology and object-oriented design, as well as the development process, including lexical analysis, parsing, semantic analysis, and the use of the Visitor pattern.

Chapter 4 outlines the requirements for the development of the programming language, including functional and non-functional requirements. It describes the design process, focusing on the architecture of the language interpreter and the design of key components such as the lexical analyzer, parser, semantic analyzer, and code generator/executor.

Chapter 5 provides an in-depth look at the implementation of the programming language. It discusses the coding of the interpreter, the integration of components, and the testing procedures used to ensure functionality, usability, and accuracy. This chapter also covers the evaluation of the language against the defined requirements and objectives.

Chapter 6 summarizes the findings and contributions of the project. It reflects on the achievement of the objectives, the impact of the developed programming language on the Somali community, and potential areas for future research and development.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

This chapter provides an in-depth literature review surrounding the development of a Somali-specific programming language. It reviews the challenges non-English speakers face in learning programming, examines past efforts in developing non-English programming languages, and discusses theoretical frameworks relevant to this project.

2.2 Challenges Faced by Non-English Speakers in Learning Programming

Individuals who are not native English speakers encounter numerous difficulties when learning programming languages written in English. The main challenge arises from the language barrier since most programming languages adopt English as their instructional language and incorporate its syntactical structure as a fundamental element of coding practice. (Guo, 2018). This intrinsic reliance on English complicates the learning experience for those unfamiliar with the language.

Moreover, Lui et al. (2020) highlight that the complexity extends into accessing and understanding the available resources, such as documentation, guides, and tutorials. Non-native speakers often require additional resources or assistance to interpret these materials correctly, owing to their limited knowledge of specialized English terminology used in technical contexts. This dependence not only hampers the speed of learning but also increases the likelihood of frustration and a sense of discouragement among learners whose first language is not English.

It is crucial to recognize that the hurdles faced by these individuals go beyond mere linguistic issues. Cultural differences also significantly influence the learning process. The interpretation of programming concepts may vary across different languages, and this variation can lead to misunderstandings when abstract ideas, typically explained in English, need to be translated. The interaction between language and cultural background can thus add another layer of complexity to the educational challenges faced by non-native English speakers in the field of programming. (Hadgraft, Roger, 2022).

2.3 Past Efforts in Creating Non-English Programming Languages

In recent years, there has been a growing recognition of the importance of making programming accessible to individuals who face language barriers (Smith, 2021). As a result, several initiatives have emerged with the goal of developing programming languages tailored to non-English speakers (Jones, 2020). This literature review aims to provide an overview of these endeavors, examining their approaches and achievements.

2.3.1 Kalimat

Kalimat is an Arabic-based programming language developed in 2010 by Dr. Mohammad Samy. As highlighted by Nabbout (2012), Its primary goal was to help kids learn programming, but it has since evolved. The language adopts Arabic grammar and syntax conventions, making code written in Kalimat closely resemble natural Arabic language structures. While suitable for professional software development, Kalimat strongly emphasizes educational use, providing a beginner-friendly environment for learners to grasp fundamental programming concepts. Additionally, Kalimat fosters a community of Arabic-speaking programmers and educators, offering support forums, tutorials, and resources in Arabic to assist users in their programming journey.

2.3.2 Himawari

Himawari means "sunflower" in Japanese. The language was created for use on Windows 98 through XP. The goal was to make it easier for beginner programmers in Japan to learn how to program by removing the need for them to learn English keywords simultaneously, and it is the most reliable Japanese programming language (jeffcarp, 2013).

Moreover, it provides a Japanese language interface for easy navigation and comprehension by Japanese-speaking users. With a focus on simplicity and accessibility, it offers a beginner-friendly learning experience. Himawari's interactive environment allows users to experiment with coding concepts and receive immediate feedback. It may include visual programming tools to aid in understanding programming logic and align with educational curricula in Japanese-speaking regions to support effective teaching of programming concepts.

2.4 Theoretical Frameworks

The theoretical framework for developing a Somali Interpreted Multiparadigm Programming Language involves several key theories and models from linguistics, computer science, and education. These theories provide a foundation for understanding the challenges and guiding the design of a programming language tailored to the Somali-speaking community.

2.4.1 Linguistic Relativity Hypothesis

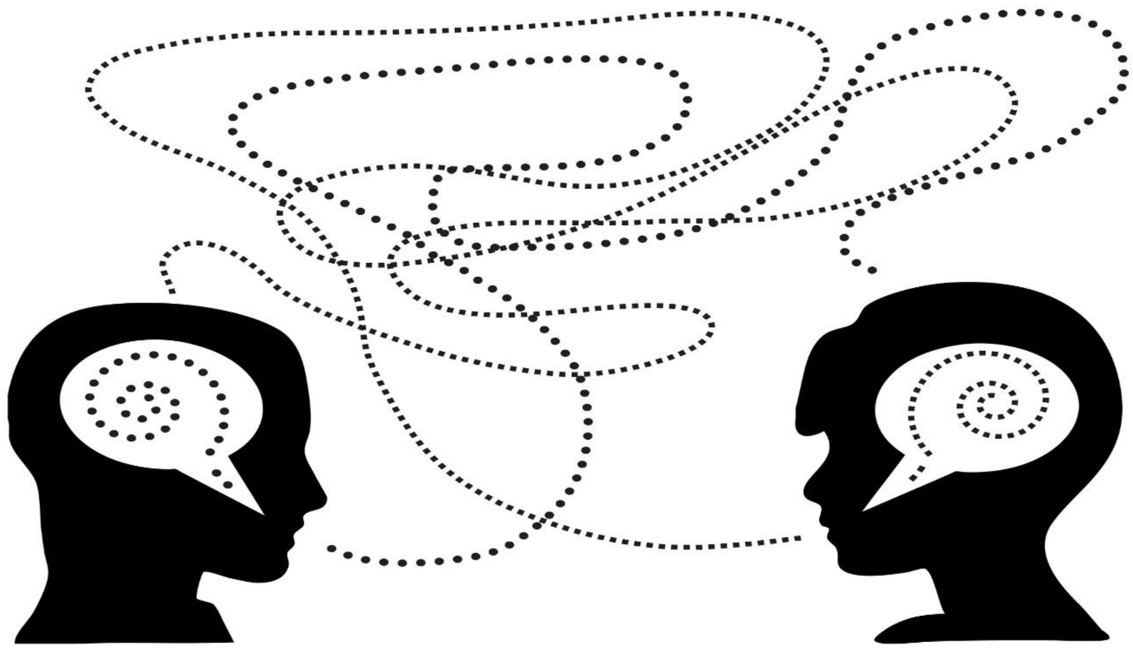


Figure 2. 1: Linguistic Relativity Hypothesis

The Sapir-Whorf Hypothesis, also known as the Linguistic Relativity Hypothesis, proposed by Edward Sapir and Benjamin Lee Whorf in the early 20th century, asserts that a language's structure can significantly influence the cognition and worldview of its speakers. When applying this hypothesis to programming education, it strongly suggests that a programming language in Somali could indeed impact how Somali speakers conceptualize and solve problems. By aligning the syntax and semantics of a programming language with the natural linguistic patterns of Somali, learners are likely to find it considerably easier to comprehend programming concepts and apply them creatively.

2.4.2 Constructivist Learning Theory

Constructivist learning theory, as proposed by Jean Piaget in the 1930s and further developed by Lev Vygotsky in the 1960s, emphasizes that learners construct their own understanding and knowledge of the world through experiences and reflecting on those experiences. In the context of programming education, this theory supports the idea that learners will better understand programming concepts if they can relate them to their existing knowledge and language. A programming language in Somali allows learners to connect new programming concepts directly to their native language, facilitating a deeper and more intuitive understanding.

2.4.3 Cognitive Load Theory

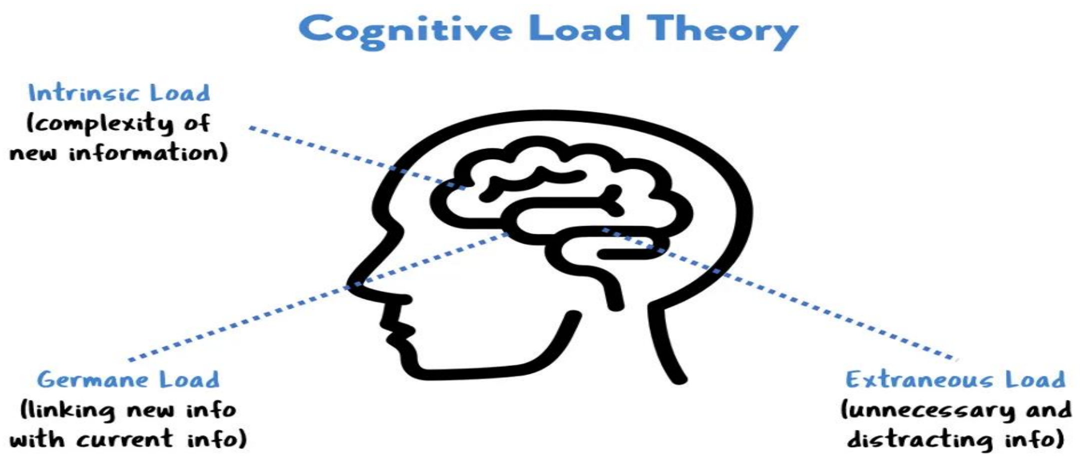

Cognitive Load Theory
Intrinsic Load
(complexity of new information)
Germane Load
(linking new info with current info)
Extraneous Load
(unnecessary and distracting info)

Figure 2. 2: Cognitive Load Theory

John Sweller's Cognitive Load Theory, developed in the late 1980s, posits that learning is most effective when the cognitive load is optimized. Cognitive load can be intrinsic, extraneous, or germane. For non-English speakers learning programming, the intrinsic load is already high due to the complexity of the subject matter. Using a programming language in Somali can reduce extraneous cognitive load by eliminating the need to translate terms and concepts from English, thus allowing learners to focus more on the germane cognitive load, which is the actual learning of programming concepts.

2.5 Summary

The literature This literature review explores the adaptation of programming languages for non-English speakers, focusing on creating a Somali-specific version. It examines challenges like significant language barriers and cultural differences that complicate learning for non-native English speakers.

- Non-English speakers need help with English-centric programming instructions, resources, and cultural nuances in traditional programming education.
- Successful past endeavors, such as the creation of Kalimat in Arabic and Himawari in Japanese, have demonstrated the feasibility and effectiveness of developing programming environments that align with local languages and cultures. This success story inspires confidence in the potential of a Somali-specific programming language.
- Theories such as the Linguistic Relativity Hypothesis, Constructivist Learning Theory, and Cognitive Load Theory support developing a Somali programming language, suggesting that integrating native linguistic patterns could reduce cognitive load and enhance learning.

This review emphasizes the importance of culturally and linguistically inclusive tools in programming education to improve accessibility and learning outcomes for non-English speakers.

CHAPTER 3

METHODOLOGY

3.1 Introduction

This chapter details the methodologies employed in the development and evaluation of a Somali Interpreted Multiparadigm Programming Language. This includes the software engineering methodologies followed during the design, implementation, and testing phases, as well as the evaluation methods used to assess the effectiveness and usability of the developed programming language.

3.2 Software Engineering Methodology

The development of the Somali programming language adhered to established software engineering principles to ensure a robust, maintainable, and scalable solution. The methodologies utilized include Agile development and object-oriented design.

3.2.1 Agile Methodology

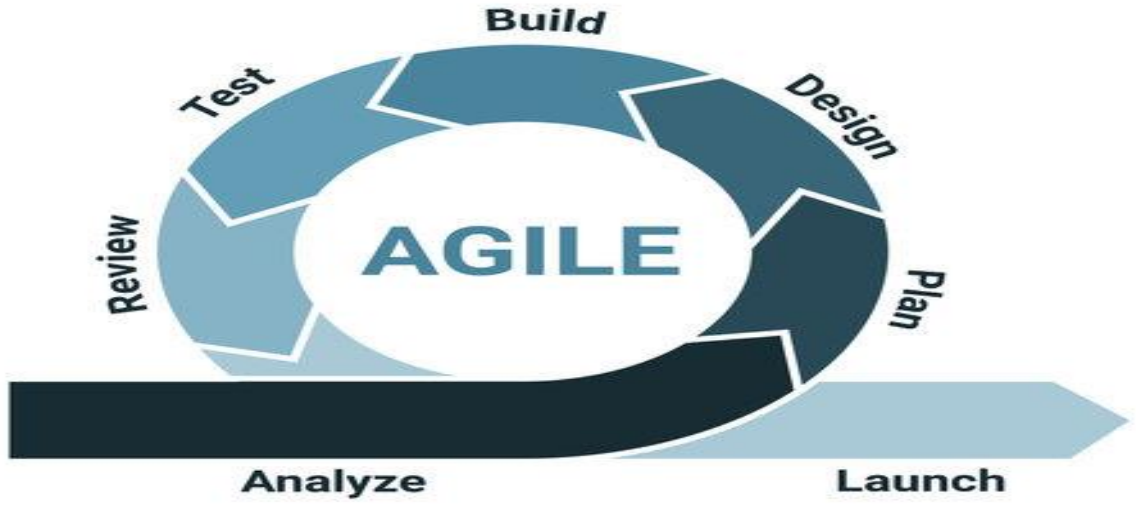


Figure 3. 1: Agile methodology process

Agile development was chosen for its iterative and incremental approach, allowing for continuous feedback and improvement. The key aspects of the Agile methodology applied were:

- **Iterations**: The project was divided into several iterations or sprints, each focusing on specific features and improvements. The phases in the iterations are:
    - ✓ **Requirement Analysis:** Gather detailed requirements and prioritize them based on importance and urgency. Create user stories to capture requirements from the end-user perspective.
    - ✓ **Planning:** Define the project's vision, objectives, and deliverables. Create a product backlog that lists all features, functions, requirements, enhancements, and fixes required for the project.
    - ✓ **Design:** Outline the system architecture and design the components necessary to meet the requirements. Plan the iterations or sprints, determining what will be delivered at the end of each sprint.
    - ✓ **Build:** Implement the planned features and functions. This phase is divided into several iterations or sprints, each focusing on specific features and improvements. Continuous feedback is obtained from stakeholders and potential users to refine requirements and improve functionality.
    - ✓ **Test:** Conduct testing at the end of each sprint to identify and fix bugs. Testing ensures that the features developed in the sprint meet the specified requirements and function correctly.
    - ✓ **Review:** Review the work completed in the sprint with stakeholders to ensure it meets their expectations. This phase involves demonstrating the completed work, gathering feedback, and incorporating any necessary changes or adjustments.

- **Continuous Feedback**: Regular feedback was obtained from stakeholders and potential users to refine requirements and improve functionality.
- **Collaboration**: Frequent communication and collaboration among team members ensured alignment with project goals and timely resolution of issues.

3.2.2 Object-Oriented Design

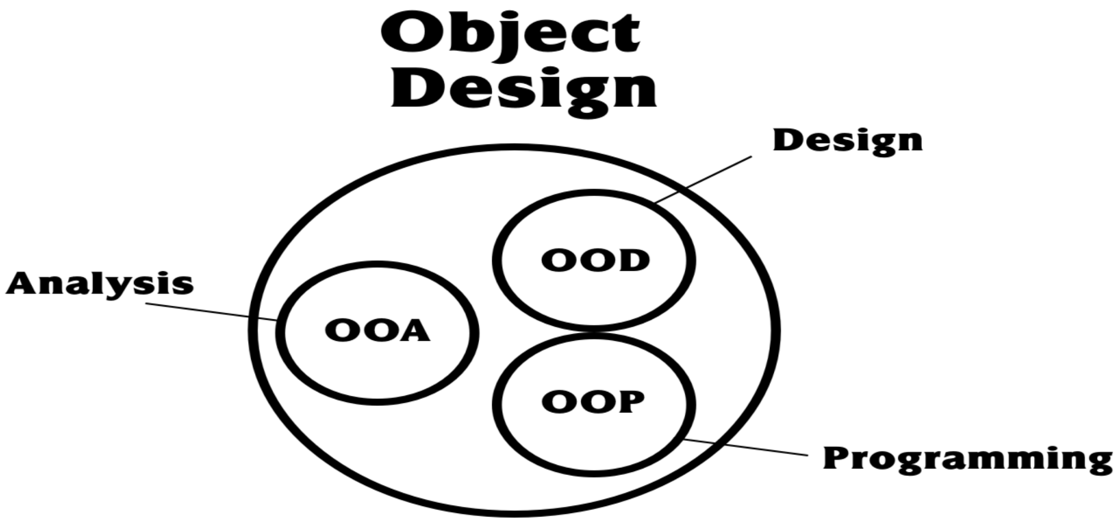


Figure 3. 2: Object-oriented Design

Object-oriented design principles were employed to create a modular and reusable codebase. Key principles included:

- **Encapsulation**: Encapsulating data and functions within classes to promote modularity and reduce complexity.
- **Inheritance**: Using inheritance to create a hierarchy of classes that share common behaviors, enhancing code reusability.
- **Polymorphism**: Implementing polymorphism to allow objects to be treated as instances of their parent class, enabling flexible and dynamic code.

3.3 Visitor Pattern

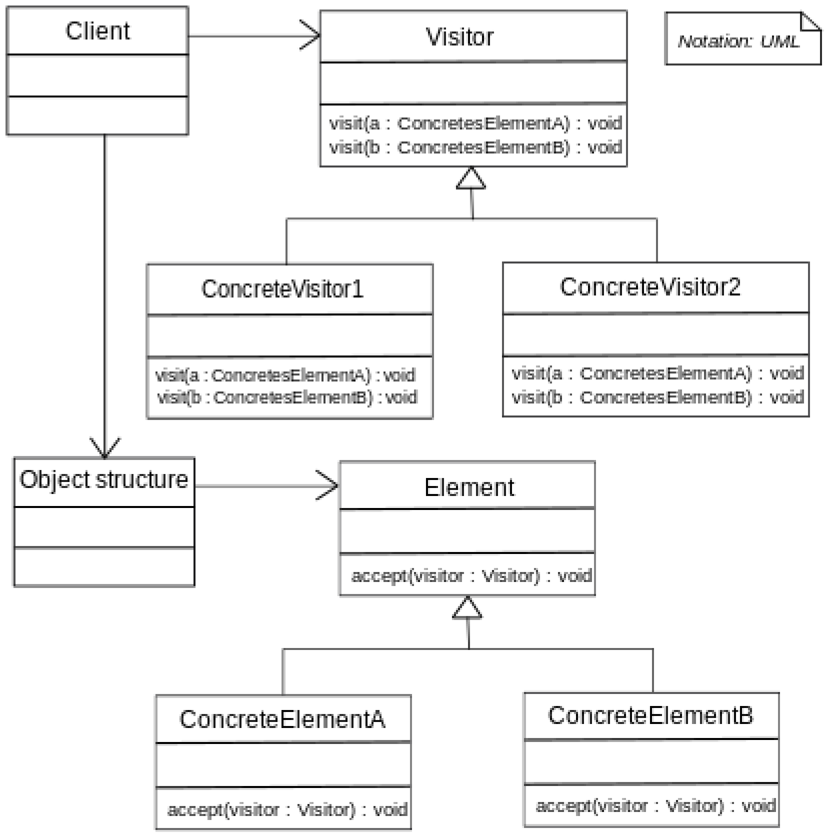

Client
Visitor
Notation: UML
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
ConcreteVisitor1
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
ConcreteVisitor2
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
Object structure
Element
accept(visitor : Visitor) : void
ConcreteElementA
accept(visitor : Visitor) : void
ConcreteElementB
accept(visitor : Visitor) : void

Figure 3. 3: Visitor Pattern UML

The Visitor design pattern was utilized to separate algorithm implementation from the objects on which it operates, promoting code modularity and extensibility. The Visitor pattern was applied in the following ways:

- **AST Traversal**: The Visitor pattern facilitated the traversal of the Abstract Syntax Tree (AST) by defining a set of visiting operations for each node type. This separation allowed for clean and maintainable code.
- **Operation Encapsulation**: By encapsulating operations in visitor classes, the language implementation could easily extend or modify behavior without altering the AST node classes.

- **Flexibility**: The pattern provided flexibility to add new operations (such as optimization passes or additional semantic checks) without modifying the existing node structure.

Implementation steps

- ✓ **Visitor Interface**: Defined an interface with visit methods for each type of AST node (e.g., visitBinaryExpression, visitVariableDeclaration).
- ✓ **Concrete Visitors**: Implemented concrete visitor classes that encapsulate specific operations like interpretation, optimization, and code generation.
- ✓ **Node Acceptance**: Modified AST node classes to accept a visitor object and call the appropriate visit method.

3.4 Technology and Tools Used to Develop System

Building a system needs technology and tools. Each system can use different types of technology. There are two main parts of the technology used: software and hardware. Software is necessary for making the system. Hardware is important for putting the system into action.

3.4.1 Hardware

The hardware requirements outline the minimum specifications needed for hosting and serving a Spring Boot backend with a React frontend application.

Table 3. 1: Hardware requirements

Hardware Component	Minimum Requirement
Processor	Dual-core processor (e.g., Intel Core i3/i5)
RAM	2 GB (4 GB recommended for better performance)
Disk Space	10 GB (HDD or SSD; SSD recommended for better performance)
Network	Stable internet connection with sufficient bandwidth

3.4.2 Software

The software requirements encompass the essential components necessary for developing and deploying a React frontend application integrated with a Spring Boot backend.

Table 3. 2: Software requirements

Software Component	**Minimum Requirement**
Operating System	Windows, Linux, MacOS, etc.
Java Development Kit (JDK)	Version 17 and above
Node.js	Latest stable version
npm (Node Package Manager)	Latest stable version

3.5 Summary

This chapter delves into our systematic approach, a fusion of Agile development for iterative improvement and object-oriented design for modularity. The strategic use of the Visitor pattern significantly enhanced our ability to traverse and manipulate the Abstract Syntax Tree (AST) with unparalleled flexibility and precision. This chapter vividly captures our journey from concept to execution, setting the stage for the impactful results and insights detailed in the subsequent chapters.

CHAPTER 4

REQUIREMENT ANALYSIS AND DESIGN

4.1 Introduction

This chapter provides a brief overview of the requirement analysis and system design. It outlines how the system architecture and related techniques fulfill user requirements. This phase also encompasses the interface application design and discusses the data flow within the system. The chapter explains the architectural design, illustrating how the system is expected to function effectively and ensuring that use cases accurately reflect the specified requirements.

4.2 System Architecture Design

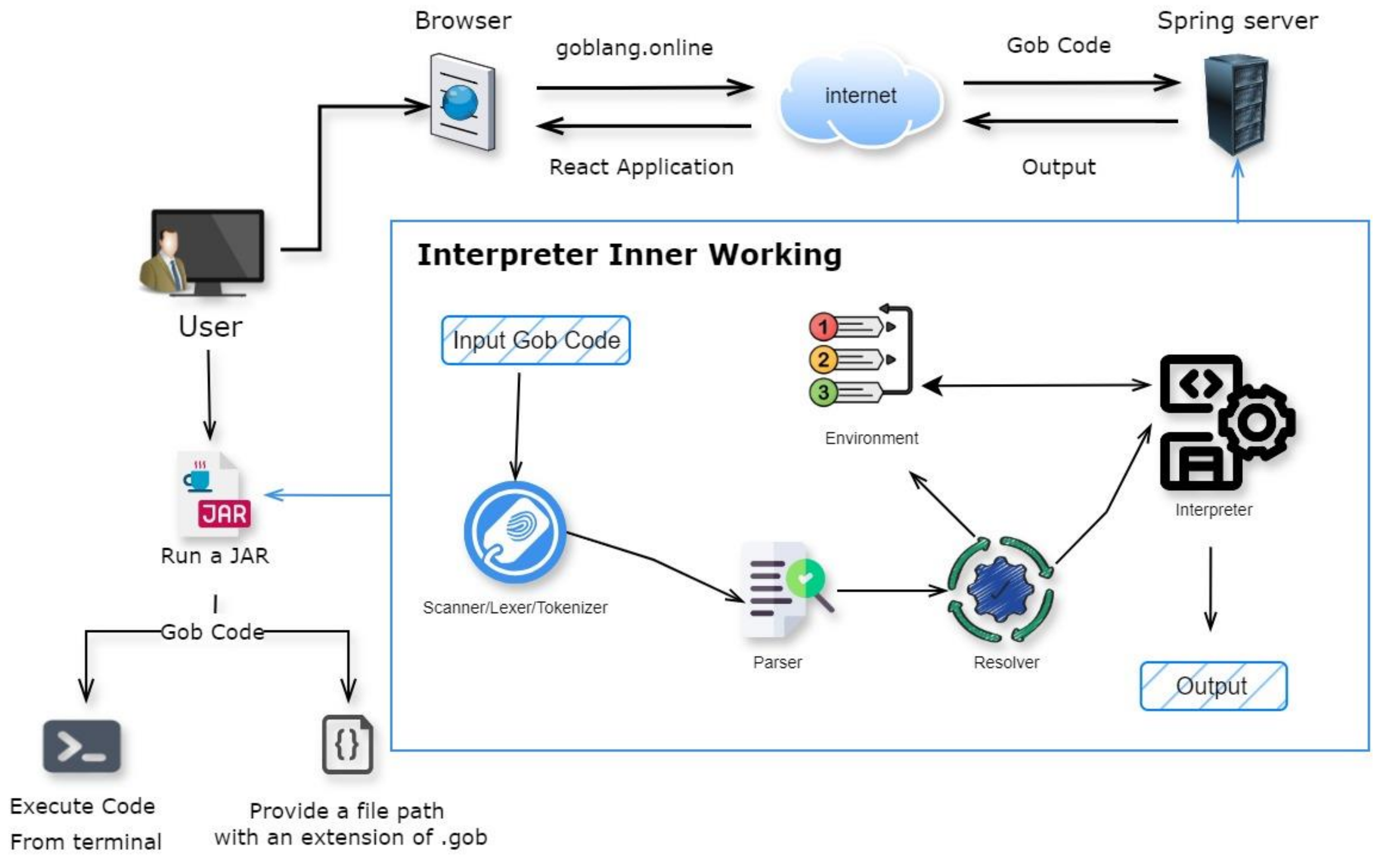


Figure 4. 1: System Architecture

Figure 4.1 illustrates the structure and main functionalities of the Somali Interpreted Multiparadigm Programming Language System. Users access the system through a web interface or through running a Java JAR, where they can input their code written in Somali.

The system's interpreter processes the code by first performing lexical analysis, breaking the input into tokens. It then carries out parsing, constructing an Abstract Syntax Tree (AST) to represent the code structure. Semantic analysis follows, ensuring that the code adheres to the rules and logical constraints of the language. If any errors occur during these phases, such as syntax or semantic issues, the interpreter immediately returns detailed error messages to the user through the interface. This real-time error feedback helps users identify and correct issues in their code.

The interpreter executes the code following successful analysis according to the defined instructions. The output or results from this execution are displayed to the user via the interface, providing immediate feedback on the code's functionality.

4.3 UML Diagrams

UML is short for Unified Modeling Language and is an ISO (International Standard) specification language for modeling objects. It is a refinement of earlier object-oriented design and object-oriented analysis methodologies.

4.3.1 Use case diagrams

The use case diagram illustrates the various services and functionalities provided by the Somali Interpreted Multiparadigm Programming Language System to developers. Within this system, one primary actor interacts with the system: The user or the learner, as depicted in Figure 4.2.

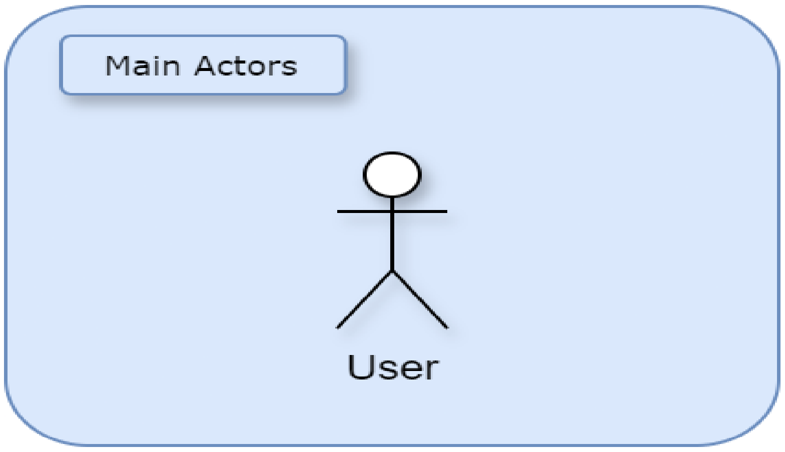


Figure 4. 2: Main actors

The user can perform any operation that he would normally do with any other Turing complete programming language. Figure 4.3 demonstrates the use cases that the user can use the system for:

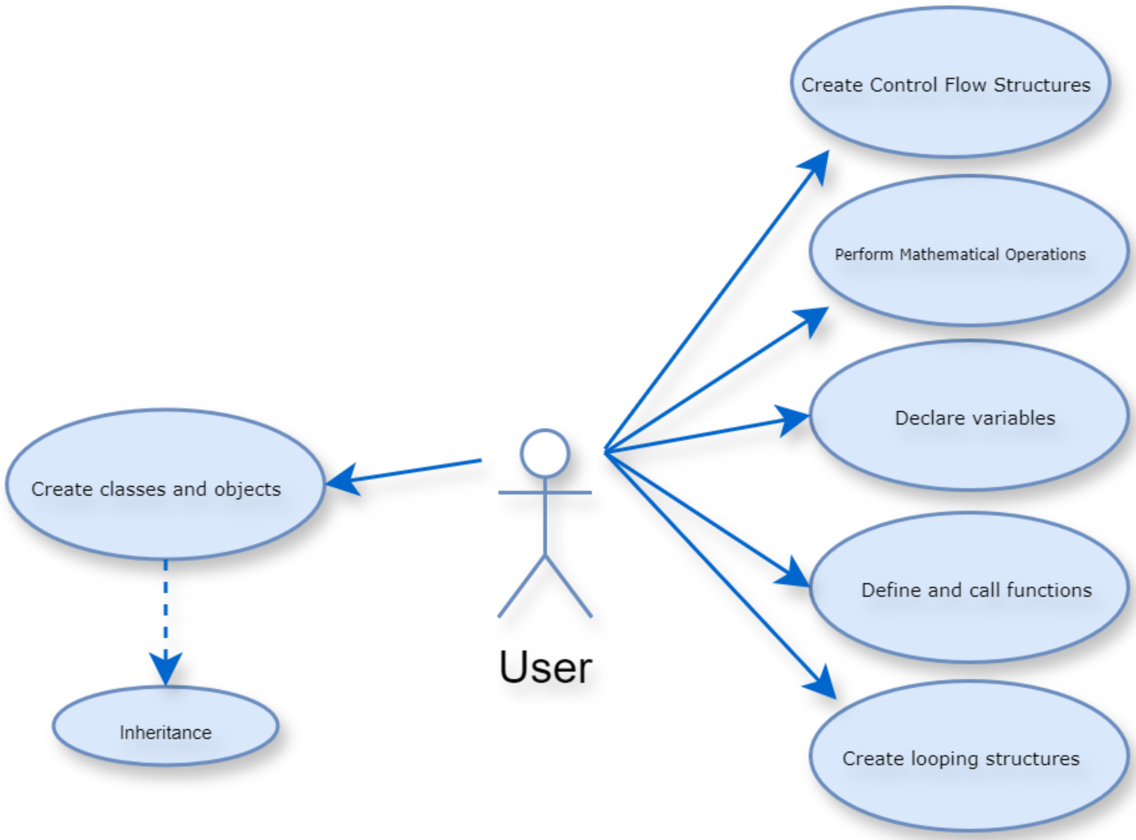

Create Control Flow Structures
Perform Mathematical Operations
Declare variables
Create classes and objects
User
Define and call functions
Inheritance
Create looping structures

Figure 4. 3: The Gob Language Use Case Diagram

Figure 4.3 demonstrates different functionalities that the user can take advantage of using the Interpreter

Table 4. 1: Use Case Description

Use Case	Functionality Description
Declare Variables	○ Allows the developer to create and initialize variables with specific values. This is fundamental for storing data that can be used and manipulated within the program.
Perform Mathematical Operations	○ Enables the developer to execute various mathematical operations such as addition, subtraction, multiplication, and division. This is essential for any computational tasks.
Define and Call Functions	○ Provides the capability to create reusable code blocks (functions) and invoke them as needed, which supports modularity and code reusability.
Create Control Flow Structures	○ Supports the implementation of control flow mechanisms such as conditional statements (kol-kale) to control the execution flow based on specific conditions.
Create Looping Structures	○ Facilitates the creation of loops (wareeg, intay) that repeat a block of code multiple times, essential for tasks requiring iteration.
Create Classes and Objects	○ Allows for the definition of classes and instantiation of objects, supporting object-oriented programming principles, which is crucial for modeling real-world entities.
Inheritance	○ Provides functionality to create new classes that inherit attributes and methods from existing ones, promoting code reuse and hierarchical class relationships.

4.4 Sequence Diagrams

Each Use case diagram typically correlates to a sequence diagram detailing the interactions within the system. In this system, the developer can perform specific tasks such as variable declaration, mathematical operations, function definition and invocation, control flow creation, loop creation, and object-oriented programming through classes and objects.

4.4.1 Sequence Diagram for JAR Execution with Terminal

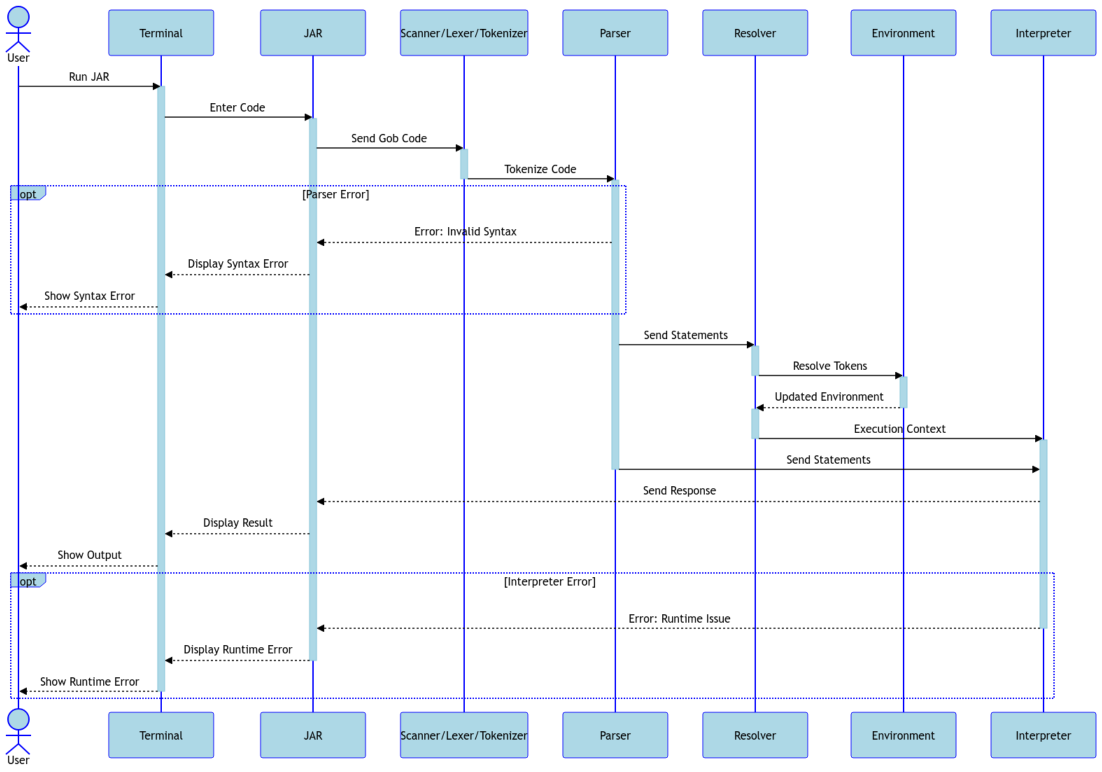


Figure 4. 4: Sequence Diagram for JAR Execution with Terminal

In Figure 4.4, the user initiates the execution of the JAR file by providing a file path to a code file. The JAR reads the code from the file through its interaction with the terminal. Subsequently, the JAR processes this input using its internal components: the Scanner for tokenization, the Parser for syntax parsing, the Resolver for resolving statements, and the Interpreter for executing the code in a controlled environment. If there are syntax errors during parsing, the JAR communicates a syntax error back to the terminal, which then displays it to the user. Similarly, if there are runtime issues during interpretation, the JAR communicates a runtime error back to the terminal for display. Finally, the terminal presents the output generated by the JAR back to the user.

4.4.2 Sequence Diagram of Browser Interaction

Figure 4.5 illustrates the sequence of interactions that occur when a user accesses the Gob language interpreter through a web browser. The user navigates to the Gob interpreter's online platform, initiating a series of communications between the browser, the React application, the Spring server, and the interpreter.

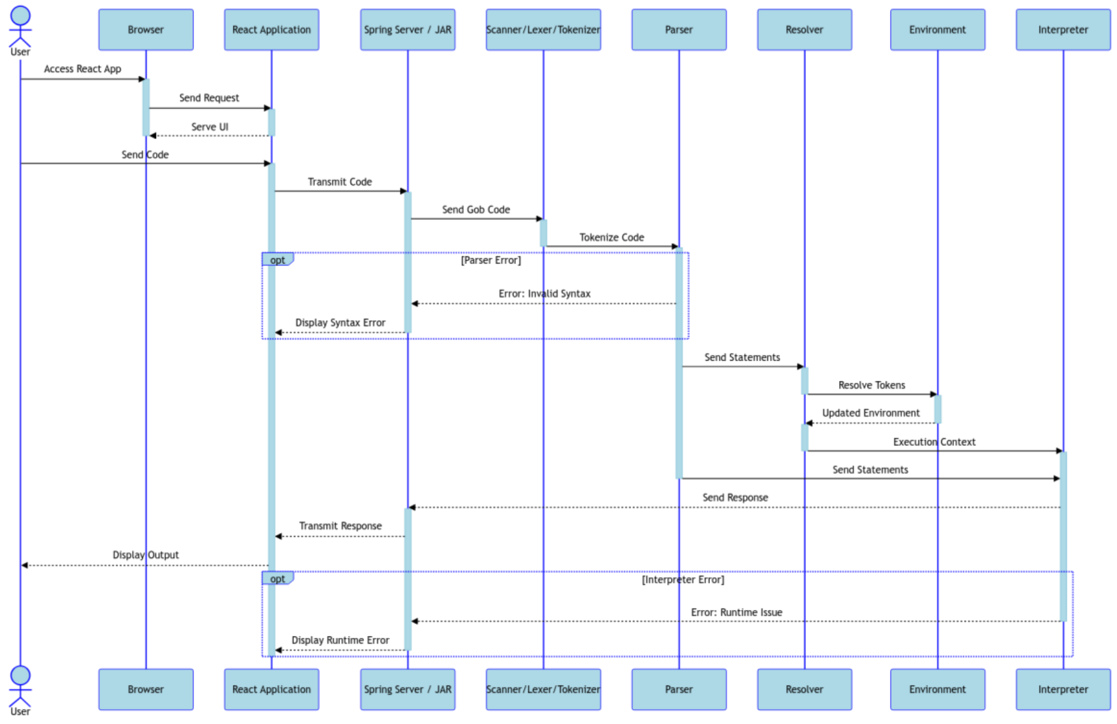


Figure 4. 5: Sequence Diagram of Browser Interaction

4.4.3 Sequence Diagram for JAR Execution with File

Figure 4.6 illustrates the process flow when a user runs the Gob interpreter from a Java Archive (JAR) file, providing a file path containing Gob code. The JAR retrieves and processes the code, starting with scanning to identify tokens. Errors during scanning result in a scanning error. Valid tokens are parsed for syntax correctness, forming an abstract syntax tree (AST). Syntax errors during parsing result in a syntax error. Resolved statements then undergo execution, with runtime errors, such as type mismatches, reported back to the user. The terminal displays any output generated by the interpreter.

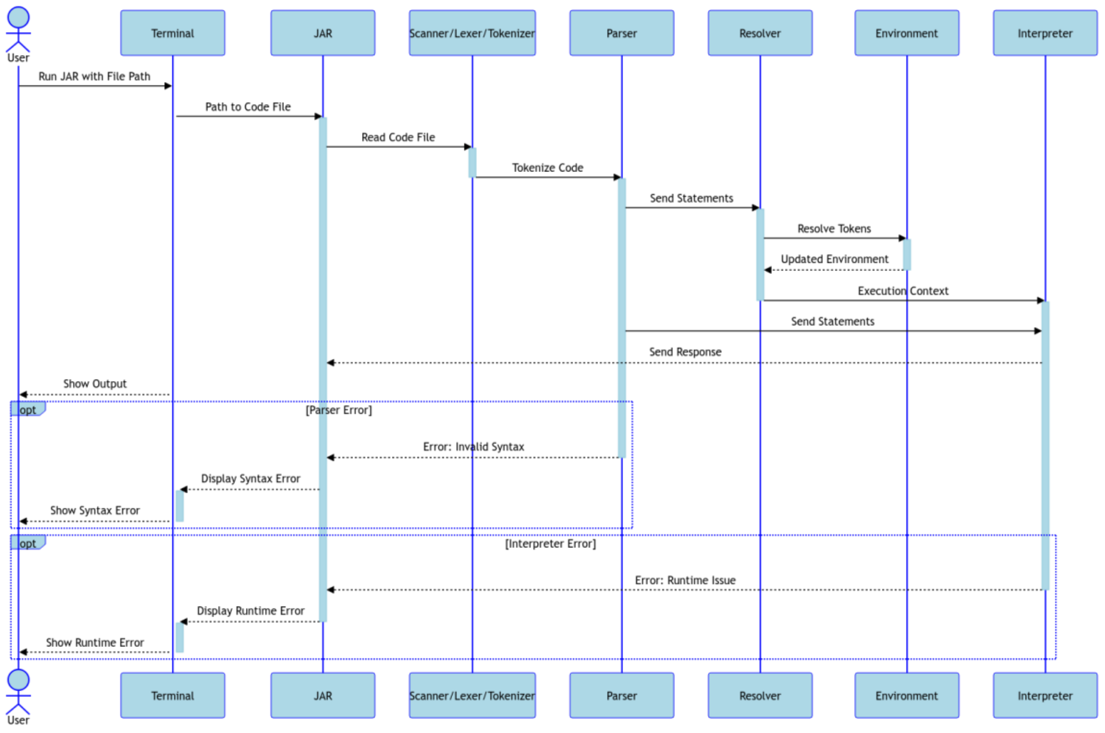


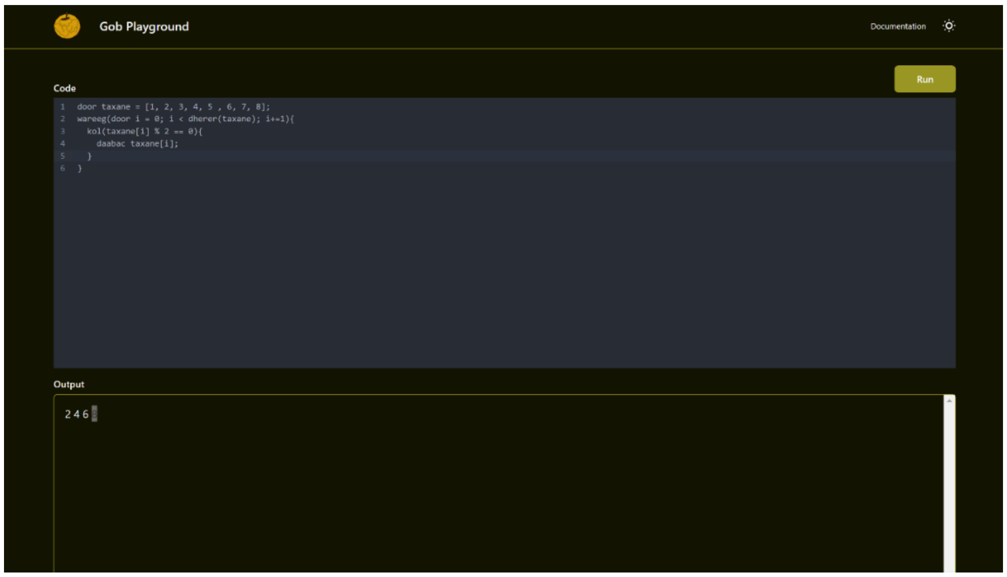
Figure 4. 6: Sequence Diagram for JAR Execution with File

4.4.4 Interface Design

The interface design of the Gob language interpreter is tailored to meet the needs of diverse user interactions while ensuring a seamless and intuitive experience across different platforms. The design encompasses various components facilitating code execution and interaction, including a web-based platform and a command-line interface provided through a JAR file.

4.4.4.1 Web Interface

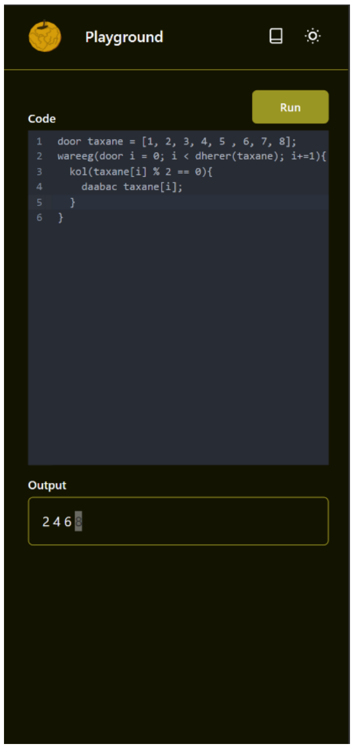
The web-based interface is hosted on Gob's online platform and utilizes modern web technologies such as React for the front end and Spring for the back end.



```
door taxane = [1, 2, 3, 4, 5 , 6, 7, 8];
wareeg(door i = 0; i < dherer(taxane); i+=1){
  kol(taxane[i] % 2 == 0){
    daabac taxane[i];
  }
}
```

```
2 4 6
```

Figure 4. 7: Desktop Web Interface



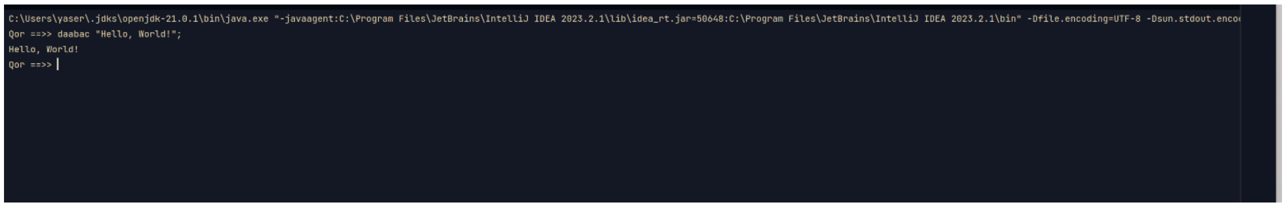
```
door taxane = [1, 2, 3, 4, 5 , 6, 7, 8];
wareeg(door i = 0; i < dherer(taxane); i+=1){
  kol(taxane[i] % 2 == 0){
    daabac taxane[i];
  }
}
```

```
2 4 6
```

Figure 4. 8: Mobile Web Interface

4.4.4.2 Command-Line Interface (CLI) via Terminal Input

The command-line interface supports direct Gob code execution through terminal input, making it suitable for quick, interactive coding sessions.

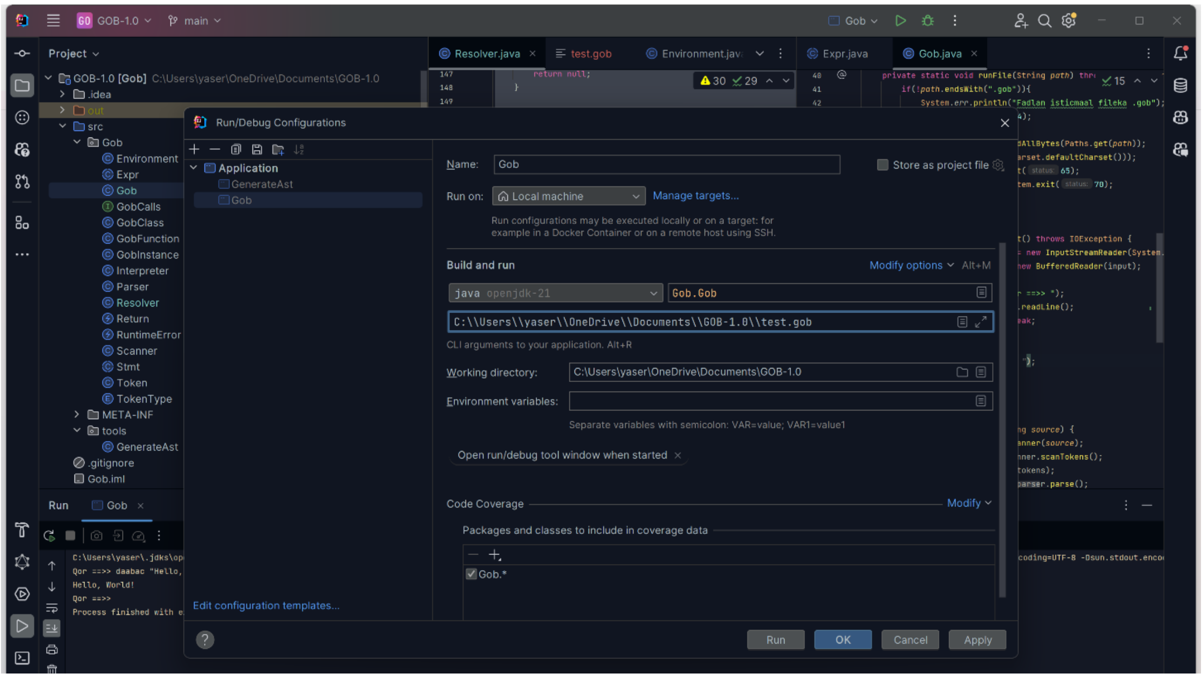


```
C:\Users\yaser\.jdks\openjdk-21.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.1\lib\idea_rt.jar=50648:C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.1\bin" -Dfile.encoding=UTF-8 -Dsun.stdout.enco
Qor ==>> daabac "Hello, World!";
Hello, World!
Qor ==>> |
```

Figure 4. 9: Terminal Input

4.4.4.3 Command-Line Interface (CLI) via File Input

The CLI also supports file-based execution of Gob code, allowing users to run scripts stored in files by passing their paths as an arguments, which is beneficial for handling larger projects.



Run/Debug Configurations

Name: Gob

Run on: Local machine

Run configurations may be executed locally or on a target: for example in a Docker Container or on a remote host using SSH.

Build and run

java openjdk-21 | Gob.Gob

```
C:\\Users\\yaser\\OneDrive\\Documents\\GOB-1.0\\test.gob
```

CLI arguments to your application. Alt+R

Working directory: C:\Users\yaser\OneDrive\Documents\GOB-1.0

Environment variables:

Separate variables with semicolon: VAR=value; VAR1=value1

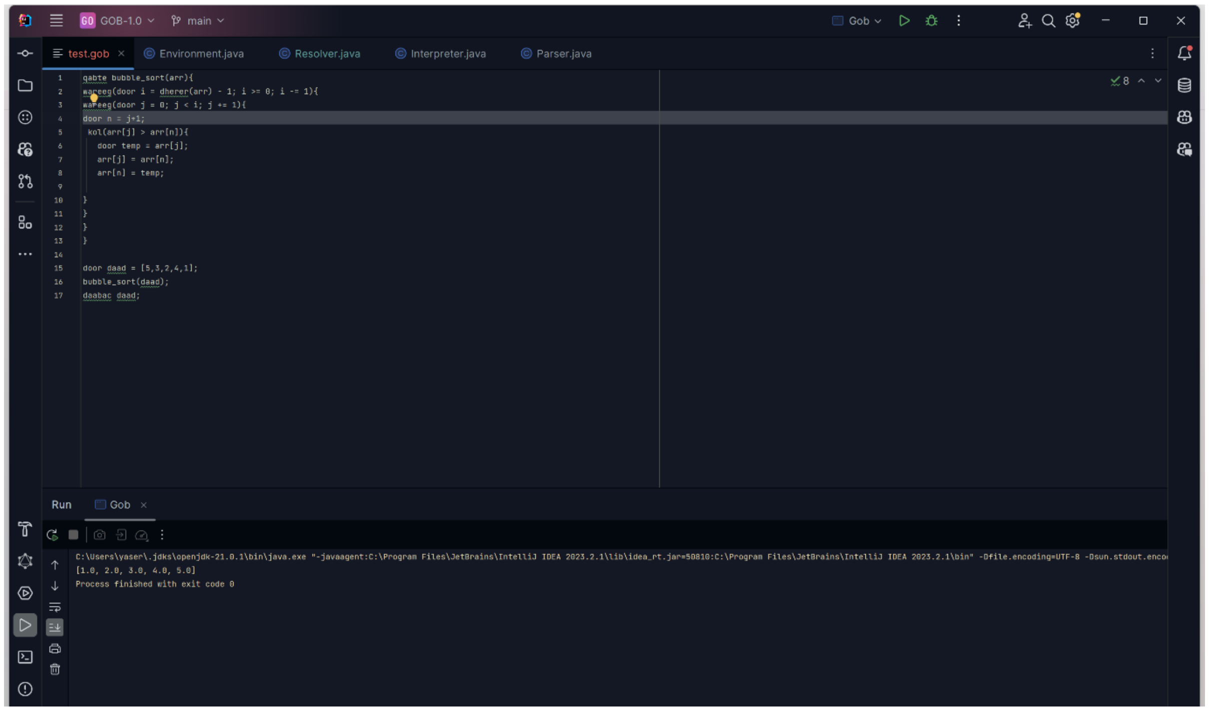
Open run/debug tool window when started

Code Coverage

Packages and classes to include in coverage data

✓ Gob.*

Figure 4. 10: passing the path as an argument to the program



```
qabte bubble_sort(arr){
wareeg(door i = dherer(arr) - 1; i >= 0; i -= 1){
wareeg(door j = 0; j < i; j += 1){
door n = j+1;
 kol(arr[j] > arr[n]){
   door temp = arr[j];
   arr[j] = arr[n];
   arr[n] = temp;

}
}
}
}

door daad = [5,3,2,4,1];
bubble_sort(daad);
daabac daad;
```

```
C:\Users\yaser\.jdks\openjdk-21.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.1\lib\idea_rt.jar=50810:C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.1\bin" -Dfile.encoding=UTF-8 -Dsun.stdout.enco
[1.0, 2.0, 3.0, 4.0, 5.0]
Process finished with exit code 0
```

Figure 4. 11: reading and executing a file

CHAPTER 5

SYSTEM IMPLEMENTATION AND TESTING

5.1 Introduction

This chapter outlines the design and implementation of an interpreter for a language system using Java that has characteristics similar to Somalia. The development environment includes IntelliJ IDEA for integrated development and Maven for build and dependency management. The chapter details the system's architecture, including lexical analysis, parsing, and execution, and discusses the testing approach using Maven.

5.2 System Development Environment

The development environment for the Somali-interpreted language system consists of the following components:

1. IDEs: Visual Studio Code (VS Code) and IntelliJ IDEA for coding and debugging.
2. Programming Language: Java for backend development.
3. Build Tool: Maven for project management and building.
4. Testing Framework: JUnit for unit tests.
5. Frontend Technologies: React, JavaScript, Tailwind CSS.
6. Backend Framework: Spring Boot for robust and scalable services.

5.3 System Architecture

The architecture of the interpreter includes three main components: the scanner/lexer, parser, and interpreter. Each component is implemented as a separate Java class and tested using Maven.

5.3.1 Scanner / Lexer

The Scanner processes the input code to produce a list of tokens, which the parser then uses.

```java
public class Scanner {
    private static final Map<String, TokenType> keywords;
    static {
        keywords = new HashMap<>();
        keywords.put("iyo", AND);
        keywords.put("cayn", CLASS);
        keywords.put("kale", ELSE);
        keywords.put("been", FALSE);
        keywords.put("wareeg", FOR);
        keywords.put("qabte", FUN);
        keywords.put("kol", IF);
        keywords.put("kolkale", ELSE_IF);
        keywords.put("ban", NIL);
        keywords.put("ama", OR);
        keywords.put("dhaxal", EXTENDS);
        keywords.put("daabac", PRINT);
        keywords.put("daabacLn", PRINTLN);
        keywords.put("celi", RETURN);
        keywords.put("ab", SUPER);
        keywords.put("kan", THIS);
        keywords.put("run", TRUE);
        keywords.put("door", VAR);
        keywords.put("intay", WHILE);
        keywords.put("dherer", LENGTH);
    }
    private final List<Token> tokens = new ArrayList<>();
    private final String source;
    private int start = 0;
    private int current = 0;
    private int line = 1;

    public Scanner(String source) {
        this.source = source;
    }

    private char advance() {

        return source.charAt(current++);
    }
    private void addToken(TokenType type) {
        addToken(type, null);
    }
    private void addToken(TokenType type, Object literal) {
        String text = source.substring(start, current);
        tokens.add(new Token(type, text, literal, line));
    }
    private boolean match(char expected) {
        if (isAtEnd()) return false;
        if (source.charAt(current) != expected) return false;
        current++;
        return true;
    }
    private char peek() {
        if (isAtEnd()) return '\0';
        return source.charAt(current);
    }
    private char peekNext() {
        if (current + 1 >= source.length()) return '\0';
        return source.charAt(current + 1);
    }
    private void string(char s) {
        while ((s == '"' && peek() != '"') || (s == '\'' && peek() != '\'') && !isAtEnd())
{
            if (peek() == '\n') line++;
            advance();
        }
        if (isAtEnd()) {
            Gob.error(line, "string aan dhamaystirnayn.");
            return;
        }
        advance();
        String value = source.substring(start + 1, current - 1);
        addToken(STRING, value);
    }
    private boolean isDigit(char c) {
        return c >= '0' && c <= '9';
    }
    private void number() {
        while (isDigit(peek())) advance();
        if (peek() == '.' && isDigit(peekNext())) {
            advance();
            while (isDigit(peek())) advance();
        }
        addToken(NUMBER, Double.parseDouble(source.substring(start, current)));
    }
    private boolean isAlphaNumeric(char c) { return isAlpha(c) || isDigit(c);}
    private void identifier() {
        while (isAlphaNumeric(peek())) advance();
        String text = source.substring(start, current);
        TokenType type = keywords.get(text);
        if (type == null) type = IDENTIFIER;
        addToken(type);
    }
    private boolean isAlpha(char c) {return (c >= 'a' && c <= 'z') || (c >= 'A' && c <=
'Z') || c == '_';}
    private void scanToken() {
        char c = advance();
        switch (c) {
            case '(':
                addToken(TokenType.LEFT_PAREN);
                break;
            case '%':
                addToken(match('=') ? COMPOUND_PERCENT: PERCENT);
                break;
            case ')':
                addToken(TokenType.RIGHT_PAREN);
                break;
            case '{':
                addToken(TokenType.LEFT_BRACE);
                break;
            case '}':
                addToken(TokenType.RIGHT_BRACE);
                break;
            case '[':
                addToken(LEFT_SQUARE);
                break;
            case ']':
                addToken(RIGHT_SQUARE);
                break;
            case ',':
                addToken(TokenType.COMMA);
                break;
            case '.':
                addToken(TokenType.DOT);
                break;
            case '-':
                addToken(match('=') ? COMPOUND_MINUS : MINUS);
                break;
            case '+':
                addToken(match('=') ? COMPOUND_PLUS: PLUS);
                break;
            case ';':
                addToken(TokenType.SEMICOLON);
                break;
            case '*':
                addToken(match('=') ? COMPOUND_STAR : STAR);
                break;
            case '!':
                addToken(match('=') ? BANG_EQUAL : BANG);
                break;
            case '=':
                addToken(match('=') ? EQUAL_EQUAL : EQUAL);
                break;
            case '<':
                addToken(match('=') ? LESS_EQUAL : LESS);
                break;
            case '>':
                addToken(match('=') ? GREATER_EQUAL : GREATER);
                break;
            case '/':
                if (match('/')) while (peek() != '\n' && !isAtEnd()) advance();
                else addToken(match('=') ? COMPOUND_SLASH: SLASH);
                break;
            case ' ': case '\r': case '\t': break; case '\n':
                line++;
                break;
            case '"':
                string('"');
                break;
            case '\'':
                string('\'');
                break;
            default:
                if (isDigit(c)) {
                    number();
                }else if (isAlpha(c)) {
                    identifier();
                } else {
                    Gob.error(line, "Xaraf lama filaan ah.");
                }
                break;
        }
    }

    public List<Token> scanTokens() {
        while (!isAtEnd()) {start = current; scanToken();}
        tokens.add(new Token(TokenType.EOF, "", null, line));
        return tokens;
    }
    private boolean isAtEnd() {
        return current >= source.length();
    }
}
```

5.3.2 Parser

The parser constructs an Abstract Syntax Tree (AST) from the list of tokens generated by the scanner.

Statements

```
    private Stmt statement() {
        if(match(TokenType.IF)) return ifStatement();
        if(match(TokenType.FOR)) return forStatement();
        if(match(TokenType.WHILE)) return whileStatement();
        if (match(TokenType.PRINT)) return printStatement();
        if (match(TokenType.PRINTLN)) return printLNStatement();
        if (match(TokenType.LEFT_BRACE)) return new Stmt.Block(block());
        if (match(TokenType.FUN)) return function("qabte");
        if (match(TokenType.RETURN)) return returnStatement();
        if (match(TokenType.CLASS)) return classDeclaration();
        return expressionStatement();
    }
```

Function Statement

```
private Stmt.Function function(String kind) {
        Token name = consume(TokenType.IDENTIFIER, "La Filayey " + kind + " Magacii.");
        consume(TokenType.LEFT_PAREN, "La Filayey '(' magaca " + kind + " kadib.");
        List<Token> parameters = new ArrayList<>();
        if (!check(TokenType.RIGHT_PAREN)) {
            do {
                parameters.add(
                        consume(TokenType.IDENTIFIER, "La Filayey magaca halbeeg"));
            } while (match(TokenType.COMMA));
         }
         consume(TokenType.RIGHT_PAREN, "La Filayey ')' halbeeg Kadib");
         consume(TokenType.LEFT_BRACE, "La Filayey '{' Kahor jidhka " + kind);
         List<Stmt> body = block();
         return new Stmt.Function(name, parameters, body);
    }
```

Class Statement

```
    private Stmt classDeclaration() {
        Token name = consume(TokenType.IDENTIFIER, "La Filayey cayn Magacii.");
        Expr.Variable superclass = null;
        if (match(TokenType.EXTENDS)) {
            consume(TokenType.IDENTIFIER, "La Filayey Magac Waalid.");
            superclass = new Expr.Variable(previous());
        }
        consume(TokenType.LEFT_BRACE, "La Filayey '(' magaca cayn kadib.");
        List<Object> methods = new ArrayList<>();
        while (!check(TokenType.RIGHT_BRACE) && !isAtEnd()) {
                methods.add(method("method"));

        }
        consume(TokenType.RIGHT_BRACE, "La Filayey '}' cayn dhamaadkii");
        return new Stmt.Class(name, superclass, methods);
    }

```

Expressions Parsing

```
 private Expr primary(){
        if(match(TokenType.FALSE)) return new Expr.Literal("been");
        if (match(TokenType.THIS)) return new Expr.This(previous());
        if (match(TokenType.SUPER)) {
            Token keyword = previous();
            consume(TokenType.DOT, "La Filayey '.'  'ab' Kadib.");
            Token method = consume(TokenType.IDENTIFIER,
                    "La Filayey qabte Uu Waalid Leyahay.");
            return new Expr.Super(keyword, method);
        }
        if(match(TokenType.TRUE)) return new Expr.Literal("run");
        if(match(TokenType.NIL)) return new Expr.Literal("ban");
        if (match(TokenType.LENGTH)) return lengthStatement();
        if (match(TokenType.IDENTIFIER)) {
            var variable = previous();
            if(match(TokenType.LEFT_SQUARE)){
                Expr idx = null;
                while(match(TokenType.NUMBER)){
                    idx = new Expr.Literal(Integer.parseInt(previous().literal.toString().split("[.]")[0]));

                }
                if(match(TokenType.IDENTIFIER)){
                    idx = new Expr.Variable(new Token(TokenType.VAR,previous().lexeme.toString(), null, previous().line));

                }
                if(match(TokenType.MINUS)){
                    consume(TokenType.MINUS, "godku negative manoqon karo [" + previous().lexeme.toString() + "]");
                }
                consume(TokenType.RIGHT_SQUARE, "La Filayey ']' god Kadib");
                return new Expr.ListCall(new Expr.Variable(variable), idx);

            }
            return new Expr.Variable(variable);
        }
        if(match(TokenType.NUMBER, TokenType.STRING)){
            return new Expr.Literal(previous().literal);
        }
        if(match(TokenType.LEFT_BRACE)){
            Expr expr = expression();
            consume(TokenType.RIGHT_PAREN, "La Filayey ')' Tacbiir Kadib.");
            return new Expr.Grouping(expr);
        }
        throw error(peek(), "La Filayey Tacbiir ");
    }
    private Expr unary() {
        if (match(TokenType.BANG, TokenType.MINUS)) {
            Token operator = previous();
            return new Expr.Unary(operator, unary());
        }
        return call();

    }
    private Expr call() {
        Expr expr = primary();
        while (true) {
            if (match(TokenType.LEFT_PAREN)) {
                expr = finishCall(expr);
            }else if (match(TokenType.DOT)) {
                Token name = consume(TokenType.IDENTIFIER, "La Filayay Magac sifo Kadib '.'.");
                expr = new Expr.Get(expr, name);
            }else {
                break;
            }
        }
```

```java
        return expr;

    }

    private Expr factor() {
        Expr expr = parenthesis();
        while (match(TokenType.SLASH, TokenType.STAR, TokenType.PERCENT)) {
            Token operator = previous();
            Expr right = unary();
            expr = new Expr.Binary(expr, operator, right);
        }
        while (match(TokenType.COMPOUND_SLASH, TokenType.COMPOUND_STAR, TokenType.COMPOUND_PERCENT)) {
            Token operator = previous();
            Expr right = factor();
            if (expr instanceof Expr.Variable) {
                Token name = ((Expr.Variable) expr).name;
                expr = new Expr.CompAssign(name, operator, right);
            }
        }
        return expr;
    }

    private Expr parenthesis() {
        if (match(TokenType.LEFT_PAREN)) {
            Expr expr = expression();
            consume(TokenType.RIGHT_PAREN, "La Filayey ')' Tacbiir Kadib.");
            return new Expr.Grouping(expr);
        }
        return unary();
    }
    private Expr term() {
        Expr expr = factor();
        while (match(TokenType.MINUS, TokenType.PLUS)) {
            Token operator = previous();
            Expr right = factor();
            expr = new Expr.Binary(expr, operator, right);

        }
        while (match(TokenType.COMPOUND_MINUS, TokenType.COMPOUND_PLUS)) {
            Token operator = previous();
            Expr right = factor();
            if (expr instanceof Expr.Variable) {
                Token name = ((Expr.Variable) expr).name;
                expr = new Expr.CompAssign(name, operator, right);
            }
        }
        return expr;
    }

    private Expr comparison() {
        Expr expr = term();
        while (match(TokenType.GREATER, TokenType.GREATER_EQUAL, TokenType.LESS, TokenType.LESS_EQUAL)) {
            Token operator = previous();
            Expr right = term();
            expr = new Expr.Binary(expr, operator, right);
        }
        return expr;
    }

    private Expr equality() {
        Expr expr = comparison();
        while (match(TokenType.BANG_EQUAL, TokenType.EQUAL_EQUAL)) {
            Token operator = previous();
            Expr right = comparison();
            expr = new Expr.Binary(expr, operator, right);
        }
        return expr;
    }

```

Expr Class and its Visitor Interface

```
package com.kq.gob;

import java.util.List;

abstract class Expr {
 interface Visitor<R> {
 R visit(Assign expr);
 R visit(CompAssign expr);
 R visit(Binary expr);
 R visit(Call expr);
 R visit(Get expr);
 R visit(Grouping expr);
 R visit(Literal expr);
 R visit(Logical expr);
 R visit(Set expr);
 R visit(Super expr);
 R visit(This expr);
 R visit(Unary expr);
 R visit(Variable expr);
 R visit(ListCall expr);
 R visit(ListUpdate expr);
 R visit(Length expr);
 }
 static class Assign extends Expr {
 Assign(Token name, Expr value) {
 this.name = name;
 this.value = value;
 }

 @Override
 <R> R accept(Visitor<R> visitor) {
 return visitor.visit(this);
 }

 final Token name;
 final Expr value;
 }
 static class CompAssign extends Expr {
 CompAssign(Token name, Token operator, Expr value) {
 this.name = name;
 this.operator = operator;
 this.value = value;
 }

 @Override
 <R> R accept(Visitor<R> visitor) {
 return visitor.visit(this);
 }

 final Token name;
 final Token operator;
 final Expr value;
```

```
}
static class Binary extends Expr {
Binary(Expr left, Token operator, Expr right) {
this.left = left;
this.operator = operator;
this.right = right;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Expr left;
final Token operator;
final Expr right;
}
static class Call extends Expr {
Call(Expr callee, Token paren, List<Expr> arguments) {
this.callee = callee;
this.paren = paren;
this.arguments = arguments;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Expr callee;
final Token paren;
final List<Expr> arguments;
}
static class Get extends Expr {
Get(Expr object, Token name) {
this.object = object;
this.name = name;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Expr object;
final Token name;
}
static class Grouping extends Expr {
Grouping(Expr expression) {
this.expression = expression;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Expr expression;
}
static class Literal extends Expr {
Literal(Object value) {
this.value = value;
}

```

```
@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Object value;
}
static class Logical extends Expr {
Logical(Expr left, Token operator, Expr right) {
this.left = left;
this.operator = operator;
this.right = right;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Expr left;
final Token operator;
final Expr right;
}
static class Set extends Expr {
Set(Expr object, Token name, Expr value) {
this.object = object;
this.name = name;
this.value = value;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Expr object;
final Token name;
final Expr value;
}
static class Super extends Expr {
Super(Token keyword, Token method) {
this.keyword = keyword;
this.method = method;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Token keyword;
final Token method;
}
static class This extends Expr {
This(Token keyword) {
this.keyword = keyword;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Token keyword;
```

```
}
static class Unary extends Expr {
Unary(Token operator, Expr right) {
this.operator = operator;
this.right = right;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Token operator;
final Expr right;
}
static class Variable extends Expr {
Variable(Token name) {
this.name = name;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Token name;
}
static class ListCall extends Expr {
ListCall(Variable name, Expr index) {
this.name = name;
this.index = index;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Variable name;
final Expr index;
}
static class ListUpdate extends Expr {
ListUpdate(Variable name, Expr index, Expr value) {
this.name = name;
this.index = index;
this.value = value;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Variable name;
final Expr index;
final Expr value;
}
static class Length extends Expr {
Length(Expr expression) {
this.expression = expression;
}

@Override
<R> R accept(Visitor<R> visitor) {
 return visitor.visit(this);
 }

 final Expr expression;
 }

 abstract <R> R accept(Visitor<R> visitor);
}

```

Stmt Class and its Visitor Interface

```
abstract class Stmt {
 interface Visitor<R> {
 R visit(Block stmt);
 R visit(Class stmt);
 R visit(Expression stmt);
 R visit(PrintLN stmt);
 R visit(Print stmt);
 R visit(Var stmt);
 R visit(Listing stmt);
 R visit(Function stmt);
 R visit(Return stmt);
 R visit(If stmt);
 R visit(Else stmt);
 R visit(While stmt);
 }
 static class Block extends Stmt {
 Block(List<Stmt> statements) {
 this.statements = statements;
 }

 @Override
 <R> R accept(Visitor<R> visitor) {
 return visitor.visit(this);
 }

 final List<Stmt> statements;
 }
 static class Class extends Stmt {
 Class(Token name, Expr.Variable superclass, List<Object> methods) {
 this.name = name;
 this.superclass = superclass;
 this.methods = methods;
 }

 @Override
 <R> R accept(Visitor<R> visitor) {
 return visitor.visit(this);
 }

 final Token name;
 final Expr.Variable superclass;
 final List<Object> methods;
 }
 static class Expression extends Stmt {
 Expression(Expr expression) {
 this.expression = expression;
 }

 @Override
 <R> R accept(Visitor<R> visitor) {
```

```
return visitor.visit(this);
}

final Expr expression;
}
static class PrintLN extends Stmt {
PrintLN(Expr expression) {
this.expression = expression;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Expr expression;
}
static class Print extends Stmt {
Print(Expr expression) {
this.expression = expression;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Expr expression;
}
static class Var extends Stmt {
Var(Token name, Expr initializer) {
this.name = name;
this.initializer = initializer;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Token name;
final Expr initializer;
}
static class Listing extends Stmt {
Listing(Token name, ArrayList<Expr> initializer) {
this.name = name;
this.initializer = initializer;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Token name;
final ArrayList<Expr> initializer;
}
static class Function extends Stmt {
Function(Token name, List<Token> params, List<Stmt> body) {
this.name = name;
this.params = params;
this.body = body;
}

```

```
@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Token name;
final List<Token> params;
final List<Stmt> body;
}
static class Return extends Stmt {
Return(Token keyword, Expr value) {
this.keyword = keyword;
this.value = value;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Token keyword;
final Expr value;
}
static class If extends Stmt {
If(Expr condition, Stmt thenBranch, Stmt elseBranch, Stmt elseIF) {
this.condition = condition;
this.thenBranch = thenBranch;
this.elseBranch = elseBranch;
this.elseIF = elseIF;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Expr condition;
final Stmt thenBranch;
final Stmt elseBranch;
final Stmt elseIF;
}
static class Else extends Stmt {
Else(Stmt statement) {
this.statement = statement;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

final Stmt statement;
}
static class While extends Stmt {
While(Expr condition, Stmt body) {
this.condition = condition;
this.body = body;
}

@Override
<R> R accept(Visitor<R> visitor) {
return visitor.visit(this);
}

```

```
  final Expr condition;
  final Stmt body;
  }

  abstract <R> R accept(Visitor<R> visitor);
}



```

5.4 Black Box and White Box Testing:

White box and black box testing are fundamental methodologies in software testing. White box testing involves examining the internal code structure to ensure correct control flows, data processing, and logic. It detects and fixes logical and syntactical errors early in development. Black box testing assesses the system externally by inputting data and checking outputs against expected behavior, ensuring the system functions as intended under various conditions.

In developing the Somali-interpreted language system, both white box and black box testing were crucial. White box tests, implemented with JUnit, rigorously validated components like the lexer, parser, resolver, and interpreter. Maven facilitated build management and test automation, ensuring consistent testing across environments. Black box testing validated overall system functionality by inputting Somali language scripts and verifying correct syntax and semantic processing. This comprehensive approach ensured the system met all functional requirements reliably.

5.5 Black Box Testing:

Black box testing evaluates the external behavior of the Somali-interpreted language system by focusing on input and output without considering the internal workings. This approach tests how well the system handles various inputs and whether the outputs match expected results. Specific testing scenarios include:

5.5.1 Interpreter Behavior Testing

This section tests the interpreter's behavior by providing different input scenarios and verifying the output. It ensures the interpreter processes inputs correctly and handles errors appropriately.

Table 5. 1: Interpreter Behavior Testing

Input	Output	Expected Outcome
Valid Gob language code	Successful execution	Interpreter processes code correctly
Invalid syntax or semantics	Error message	Interpreter identifies and reports errors
Empty input	No output	Interpreter handles empty input gracefully

5.5.2 Syntax and Semantic Testing

This section verifies the system's ability to interpret Somali language syntax and semantics accurately. It ensures that correct code is parsed successfully and errors are handled properly.

Table 5. 2: Syntax and Semantic Testing

Input	Output	Expected Outcome
Correct Gob syntax	Successful parsing	Interpreter parses code correctly
Incorrect syntax	Parsing error message	Interpreter reports syntax errors
Semantic error	Execution error message	Interpreter identifies semantic errors

5.5.3 Error Handling and Recovery Testing

This section tests the interpreter's error handling and recovery mechanisms, ensuring the system can gracefully handle errors and continue functioning or exit safely.

Table 5. 3: *Error Handling and Recovery Test*

Input	**Output**	**Expected Outcome**
Code with syntax errors	Error message	Interpreter reports syntax errors
Unexpected runtime errors	Graceful error handling	Interpreter recovers or exits gracefully

5.6 White Box Testing

White box testing involves detailed examination of the internal code and logic of the Somali-interpreted language system. This testing approach ensures that each component works correctly, code is executed as expected, and interactions between different parts of the system function seamlessly. Specialized tools and techniques are used to detect and fix any issues within the codebase, ensuring that the system behaves as intended and prevents potential data loss or failures.

5.7 Summary

This chapter details the implementation of a Somali-interpreted language system in Java, covering its core components and testing methodologies. Using IntelliJ IDEA for development, Maven for project management, and JUnit for testing, the system underwent rigorous validation through white box and black box testing. The chapter concludes with the system's successful implementation, ensuring accurate interpretation of Somali-like syntax and readiness for deployment.

REFERENCES

Adewusi, O. A. (2024). Educational approaches in African social work: Implications for U.S. social work training. *International Journal of Science and Research Archive*, 1178–1194.

Ahmed, R. (2023, 5 30). *Importance of Education*. Retrieved from Medium: https://medium.com/@roohan.ahmed.142004/impo

Al-Ajlan, A. (2015). Kalimat: An Arabic programming language for beginners. *International Journal of Computer Applications*, 1-5.

Becker. (2019). Parlez-vous Java? Bonjour La Monde!= Hello World: Barriers to Programming Language Acquisition for Non-Native English Speakers. *Proceedings of the Psychology of Programming Interest Group Annual Conference (PPIG)*.

Guo, P. (2018). Non-Native English Speakers Learning Computer Programming: Barriers, Desires, and Design Opportunities.

Hadgraft, Roger. (2022). Identifying the Difficulties of Learning Programming for Non-English Speakers at CQUniversity and Sebha University. *SEBHA UNIVERSITY JOURNAL OF PURE &APPLIED SCIENCES*.

jeffcarp. (2013, February 25). *Japanese Programming*. Retrieved from jeffcarp: https://www.jeffcarp.com/posts/2013/japanese-programming/

Jones, R. e. (2020). Programming Languages for Non-English Speakers: A Review of Recent Initiatives. *Journal of Global Technology Education*, 56-72.

Liu, Y. H. (2020). A Study on the Influence of English Language Barriers on Programming Education for Chinese Students. *Journal of Educational Technology & Society*, 21-34.

Nabbout, W. (2012, May 30). *Kalimat: First Arabic Programming Language for Kids*. Retrieved from Arabnet: https://www.arabnet.me/english/editorials/business/industry/kalimat-first-arabic-programming-language-for-kids

Olatunji, E. K., Oladosu, J. B., Odejobi, O. A., & Olabiyisi, S. O. (2020). Design and implementation of an African native language-based programming language. *International Journal of Advances in Applied Sciences*.

Smith, J. (2021). Speakers., Bridging the Language Gap: The Role of Programming Languages for Non-English. *Journal of Multilingual Computing*, 211-226.