BASH-C Compiler

A MINI PROJECT REPORT

Submitted by

SEENU NAHAK [RA2011033010111]
MADHAV KHATORIA [RA2011033010131]

Under the guidance of
Dr. J Jeyasudha

(Assistant Professor, Department of Computational Intelligence)

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING
with specialization in Software Engineering

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR – 603 203

May-2023

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

BONAFIDE CERTIFICATE

Certified that this project report "**BASH-C COMPILER**" is the bonafide work of "**SEENU NAHAK [RA2011003010111], MADHAV KHATORIA [RA2011033010131],**" of III Year/VI Sem B.Tech(CSE) who carried out the mini project work under my supervision for the course **18CSC304J- Compiler Design** in SRM Institute of Science and Technology during the academic year 2023(Even Semester).

[illegible]
10/5/23

SIGNATURE	**HEAD OF THE DEPARTMENT**
Faculty In-Charge	**Dr. R Annie Uthra**
Dr. Jeyasudha J	Professor and Head ,
Assistant Professor	Department of Computational Intelligence,
Department of Computational Intelligence	SRM Institute of Science and Technology
SRM Institute of Science and Technology	Kattankulathur Campus, Chennai
Kattankulathur Campus, Chennai	

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

BONAFIDE CERTIFICATE

Certified that this project report "**BASH-C COMPILER**" is the bonafide work of "**SEENU NAHAK [RA20110033010111]**, **MADHAV KHATORIA [RA2011033010131]**," of III Year/VI Sem B.Tech(CSE) who carried out the mini project work under my supervision for the course **18CSC304J- Compiler Design** in SRM Institute of Science and Technology during the academic year 2023(Even Semester).

SIGNATURE
Faculty In-Charge
Dr. Jeyasudha J
Assistant Professor
Department of Computational Intelligence
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

HEAD OF THE DEPARTMENT
Dr. R Annie Uthra
Professor and Head ,
Department of Computational Intelligence,
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

ABSTRACT

A Bash to C compiler is a program that takes a Bash script as input and converts it into an equivalent C program. The resulting C program can then be compiled and executed on a variety of platforms, making it a more portable and efficient solution for shell scripting.

The purpose of this project is to design and implement a Bash to C compiler that can automatically translate Bash scripts into C code, compile it, and produce the output. The compiler will consist of a lexer and parser to analyze the Bash script and generate the corresponding C code, along with a backend to generate and execute the compiled C code.

The Bash to C compiler will be implemented using Python, along with the Flask web framework for the front-end interface. The backend will use the GCC compiler to generate the executable C code from the generated C source code.

The primary objective of this project is to create a reliable and efficient Bash to C compiler that can handle a wide range of Bash scripts and produce efficient and portable C code. The resulting program should be easy to use and maintain, and should provide a significant improvement over traditional Bash scripts in terms of performance and portability.

List of Figures

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

A compiler is a computer program that transforms source code written in a programming language into another computer language. It is an essential tool for software development and plays a crucial role in the software development life cycle. One of the most widely used programming languages is C, which is a general-purpose programming language that supports structured programming, lexical variable scope, and recursion.

In this project, we aim to design and develop a Bash-C compiler, which is a compiler that converts Bash shell scripts to C code. The Bash-C compiler will be a valuable tool for developers who want to convert their Bash scripts to C code, which can be compiled and executed on any system that supports C.

This project will be divided into two parts. In the first part, we will design and develop the Bash-C compiler. In the second part, we will test the compiler using various Bash scripts and verify the correctness of the generated C code.

In the following sections, we will discuss the problem statement, objectives, and the need for the Bash-C compiler. We will also provide a requirements specification, system design, and conclusion for this project.

1.2 PROBLEM STATEMENT

Traditional Bash scripting has limitations in terms of performance, portability, and maintainability. Bash scripts can be slow and resource-intensive, making them unsuitable for certain applications. Furthermore, Bash scripts are often tied to specific operating systems or environments, limiting their portability across different platforms. Finally, Bash scripts can become complex and difficult to maintain over time, making them prone to errors and difficult to modify or extend.

The problem statement for the Bash to C compiler project is to address these limitations and challenges of traditional Bash scripting by providing a tool for converting Bash scripts to C code, which can then be compiled for improved performance, portability, and maintainability. The Bash to C compiler aims to provide a solution to these challenges by enabling developers to write Bash scripts in a high-level language and then automatically converting them into C code, which can be compiled for improved performance and portability. By doing so, the Bash to C compiler can provide a more efficient and flexible way to write and manage shell scripts.

1.3 OBJECTIVES

The objectives of the Bash to C compiler project are as follows:

- To develop a tool that can convert Bash scripts to C code, which can be compiled for improved performance, portability, and maintainability.
- To provide a more efficient and flexible way to write and manage shell scripts, which can be used across different operating systems and environments.
- To reduce the complexity and difficulty of writing and maintaining Bash scripts, by providing a high-level language that is easier to understand and modify.
- To improve the speed and efficiency of Bash scripts, by compiling them into optimized C code that can be executed faster than traditional Bash scripts.
- To enhance the security and reliability of Bash scripts, by providing a way to validate and verify the correctness of the code before it is executed.
- To support a wide range of Bash script features and functionality, by providing a comprehensive set of libraries and tools for converting Bash scripts to C code.
- To encourage the adoption of the Bash to C compiler by making it freely available and open source, and by providing documentation and support to help developers use it effectively.
- To continuously improve and enhance the Bash to C compiler based on user feedback, bug reports, and new requirements or use case.

1.4 Need for Bash to C compiler

The Bash to C compiler is needed for several reasons:

1. **Improved performance**: Bash scripts can be slow and resource-intensive, which can be a problem for certain applications. By converting Bash scripts to C code and compiling it, the resulting code can be executed much faster, improving overall performance.

2. **Portability**: Bash scripts are often tied to specific operating systems or environments, which can limit their portability across different platforms. By converting Bash scripts to C code, the resulting code can be compiled for different platforms, making it more portable.

3. **Maintainability**: Bash scripts can become complex and difficult to maintain over time, making them prone to errors and difficult to modify or extend. By providing a high-level language for writing Bash scripts and converting them to C code, the resulting code can be easier to understand and modify, improving overall maintainability.

4. **Security:** Bash scripts can potentially be vulnerable to security exploits if they are not written carefully. By providing a way to validate and verify the correctness of Bash scripts before execution, the Bash to C compiler can enhance the security and reliability of Bash scripts.

5. **Familiarity**: C is a widely used and familiar programming language, making it easier for developers to write, understand, and modify the resulting code.

Overall, the Bash to C compiler addresses several limitations and challenges of traditional Bash scripting, providing a more efficient and flexible way to write and manage shell scripts.

1.5 REQUIREMENT SPECIFICATION:

1.5.1 FUNCTIONAL REQUIREMENTS:

1. **Conversion of Bash scripts to C code**: The Bash to C compiler must be able to accurately and efficiently convert Bash scripts to C code.

2. **Support for Bash script features and functionality**: The Bash to C compiler must support a wide range of Bash script features and functionality, including variables, loops, conditional statements, functions, and command-line arguments.

3. **Validation and verification of code**: The Bash to C compiler must provide a way to validate and verify the correctness of the converted code before it is executed.

4. **Error handling**: The Bash to C compiler must provide clear and informative error messages to help developers identify and fix errors in the Bash scripts.

5. **Performance optimization**: The Bash to C compiler must optimize the resulting C code for improved performance and efficiency.

6. **Library support**: The Bash to C compiler must provide a comprehensive set of libraries and tools for converting Bash scripts to C code, including standard libraries and user-defined libraries.

7. **Command-line interface**: The Bash to C compiler must provide a user-friendly command-line interface for developers to use the tool effectively.

1.5.2 NON-FUNCTIONAL REQUIREMENTS:

1. **Portability**: The Bash to C compiler must be portable across different platforms and operating systems.

2. **Efficiency**: The Bash to C compiler must be efficient and not introduce significant overhead during conversion and compilation.

3. **Security**: The Bash to C compiler must provide a way to ensure the security and reliability of the converted C code.

4. **Usability**: The Bash to C compiler must be easy to use and require minimal training for developers to use effectively.

5. **Availability**: The Bash to C compiler must be freely available and open source to encourage adoption and community development.

Overall, the Bash to C compiler must meet these requirements in order to provide a more efficient and flexible way to write and manage shell scripts

1.5.3 HARDWARE & SOFTWARE SPECIFICATION

Hardware Requirements:

Processor: A modern multi-core processor (e.g., Intel Core i5 or higher) to handle the compilation process efficiently.
Memory (RAM): A minimum of 8 GB is recommended
Storage: Adequate storage space for the source code, compiler tools, libraries, and any additional resources.(A minimum of 128 GB is recommended)

Operating System: Windows / linux distributions / macOS

Development Environment:
Integrated Development Environment (IDE): Visual Studio Code, Eclipse, or JetBrains IntelliJ IDEA
Version Control: Git to manage source code, track changes, and collaborate with other developers if applicable

Programming Languages and Tools:

Compiler Design Language: Python
Back-end Framework: Flask
Front-end Framework: Boostrap

Document Preparation Software: Used word processing software like Microsoft Word for creating the compiler design report.

CHAPTER 2

2.1 What does a compiler need to know during BASH to C conversion?

In order to successfully convert Bash scripts into C code, a Bash to C compiler needs to understand the syntax and semantics of the Bash scripting language. This includes knowledge of Bash variables, loops, conditionals, functions, and other language constructs, as well as the standard Bash commands and utilities.

The compiler must be able to perform lexical analysis (i.e., tokenization) and syntactic analysis (i.e., parsing) of the Bash script, in order to generate an abstract syntax tree (AST) that captures the structure of the script. The compiler must then be able to translate the AST into equivalent C code, while preserving the semantics of the original Bash script.

Additionally, the compiler must be able to handle the differences between Bash and C, such as the handling of variable types, command line arguments, and input/output. The compiler must also be able to generate code that is efficient and readable, while avoiding common pitfalls such as buffer overflows, race conditions, and other security vulnerabilities.

Overall, a Bash to C compiler must have a thorough understanding of both Bash and C, and be able to translate between the two languages in a reliable and efficient manner.

2.2 Limitations of CFGs.

Context free grammars deal with syntactic categories rather than specific words. The declare before use rule requires knowledge which cannot be encoded in a CFG and thus CFGs cannot match an instance of a variable name with another. Hence the we introduce the attribute grammar framework.

Lexical analysis

Lexical analysis is a crucial phase in the process of compiling or interpreting programming languages. It involves breaking down the source code into its fundamental components, called tokens, which are the smallest meaningful units of the language. The main objective of lexical analysis is to scan the input code and convert it into a sequence of tokens that can be further processed by the compiler or interpreter.

The lexical analyzer, also known as the lexer or scanner, is responsible for performing this task. It receives the source code as input and reads it character by character. It applies a set of rules, defined by the language's lexical grammar, to identify and categorize the tokens. These rules are often expressed using regular expressions or finite automata.

During the lexical analysis process, the lexer eliminates unnecessary characters such as white spaces, comments, and formatting symbols, as they do not contribute to the understanding of the program's structure. It also handles certain language-specific features like string literals, identifiers, keywords, operators, and constants.

The lexer maintains a symbol table, a data structure that keeps track of identifiers encountered in the code along with their associated information such as data type, memory location, and scope. This information is essential for the subsequent phases of compilation or interpretation.

Lexical analysis helps in detecting and reporting lexical errors, such as misspelled keywords or undefined identifiers. If an error is found, the lexer generates an error message indicating the line number and the nature of the error, which helps programmers in identifying and rectifying the mistakes.

Once the lexical analysis phase is complete, the lexer outputs a stream of tokens to the parser, which then performs further syntactic analysis to build the abstract syntax tree (AST) of the program.

Here are the classification of the of tokens

- Operator. One or two consecutive characters that matches: + - * / = == != > < >= <=
- String. Double quotation followed by zero or more characters and a double quotation. Such as: "hello, world!" and ""
- Number. One or more numeric characters followed by an optional decimal point and one or more numeric characters. Such as: 15 and 3.14
- Identifier. An alphabetical character followed by zero or more alphanumeric characters.
- Keyword. Exact text match of: LABEL, GOTO, PRINT, INPUT, LET, IF, THEN, ENDIF, WHILE, REPEAT, ENDWHILE

Syntax directed definition

This is a context free grammar with rules and attributes. It specifies values of attributes by associating semantic rules with grammar productions.

Syntax Directed Translation

This is a compiler implementation method whereby the source language translation is completely driven by the parser.The parsing process and parse tree are used to direct semantic analysis and translation of the source program. Here we augment conventional grammar with information to control semantic analysis and translation. This grammar is referred to as attribute grammar. The two main methods for SDT are Attribute grammars and syntax directed translation scheme

Attributes

An attribute is a property whose value gets assigned to a grammar symbol. Attribute computation functions, also known as semantic functions are functions associated with productions of a grammar and are used to compute the values of an attribute. Predicate functions are functions that state some syntax and the static semantic rules of a particular grammar.

Types of Attributes

Type -These associate data objects with the allowed set of values.
Location - May be changed by the memory management routine of the operating system.
Value -These are the result of an assignment operation.
Name-These can be changed when a sub-program is called and returns.
Component - Data objects comprised of other data objects. This binding is represented by a pointer and is subsequently changed.

Synthesized Attributes.

These attributes get values from the attribute values of their child nodes. They are defined by a semantic rule associated with the production at a node such that the production has the non-terminal as its head.

An example

S → ABC
S is said to be a synthesized attribute if it takes values from its child node (A, B, C).

An example
E → E + T { E.value = E.value + T.value }
Parent node E gets its value from its child node.

Inherited Attributes.

- These attributes take values from their parent and/or siblings.
- They are defined by a semantic rule associated with the production at the parent such that the production has the non-terminal in its body.
- They are useful when the structure of the parse tree does not match the abstract syntax tree of the source program.
- They cannot be evaluated by a pre-order traversal of the parse tree since they depend on both left and right siblings.

An example;
S → ABC

A can get its values from S, B and C.
B can get its values from S, A and C
C can get its values from A, B and S

Expansion

This is when a non-terminal is expanded to terminals as per the provided grammar.

Reduction

This is when a terminal is reduced to its corresponding non-terminal as per the grammar rules. Note that syntax trees are parsed top, down and left to right

Attribute grammar

This is a special case of context free grammar where additional information is appended to one or more non-terminals in-order to provide context-sensitive information.
We can also define it as SDDs without side-effects.
It is the medium to provide semantics to a context free grammar and it helps with the specification of syntax and semantics of a programming language.
When viewed as a parse tree, it can pass information among nodes of a tree.

An Example

Given the CFG below;
E ::= E1+T {E.val = E1 .val + T.val}
E ::= T {E.val = T.val}
T ::= T1*F {T.val = T1 .val*F.val}
T ::= F {T.val = F.val}
F ::= num {F.val = num}
The right side contains semantic rules that specify how the grammar should be interpreted.
The non-terminal values of E and T are added and their result copied to the non-terminal E.

Defining an Attribute Grammar

Attribute grammar will consist of the following features;

- Each symbol X will have a set of attributes A(X)
- A(X) can be;
    - Extrinsic attributes obtained outside the grammar, notable the symbol table

    - Synthesized attributes passed up the parse tree
    - Inherited attributes passed down the parse tree.
- Each production of the grammar will have a set of semantic functions and predicate functions(may be an empty set)

Abstract Syntax Trees(ASTs)

These are a reduced form of a parse tree.

They don't check for string membership in the language of the grammar.

They represent relationships between language constructs and avoid derivations

An example

The parse tree for 1+2*3 and 4/-5

The parse tree

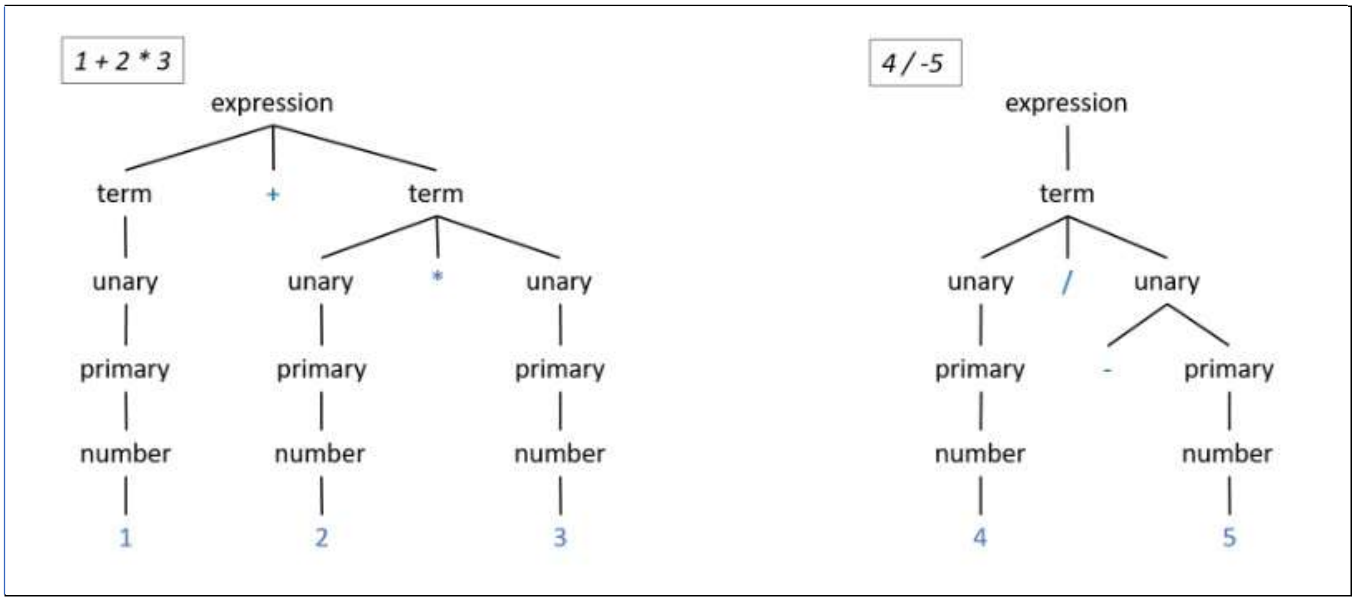


Figure 2.1

Properties of abstract syntax trees.

- Good for optimizations.
- Easier evaluation.
- Easier traversals.
- Pretty printing(unparsing) is possible by in-order traversal.

- Postorder traversal of the tree is possible given a postfix notation.

2.4 Role of Lexer & Parser:-

The lexer and parser are two key components of a compiler that play important roles in the Bash to C compilation process.

The lexer, also known as the tokenizer, is responsible for breaking down the Bash script into a stream of tokens, which represent the individual elements of the script such as keywords, operators, and identifiers. The lexer uses regular expressions and other pattern-matching techniques to identify these tokens and assign them appropriate token types. Once the tokens are identified, they can be passed on to the parser.

The parser, also known as the syntactic analyzer, takes the stream of tokens generated by the lexer and constructs a parse tree, which represents the hierarchical structure of the Bash script. The parser uses a set of rules and a grammar to recognize the various language constructs in the Bash script and to ensure that they are organized in a valid way. If the Bash script violates any of the syntax rules, the parser will raise an error and stop the compilation process.

Together, the lexer and parser form the frontend of the compiler, which is responsible for analyzing and processing the input Bash script. Once the frontend has generated an AST that represents the structure and semantics of the Bash script, the backend of the compiler can take over and generate equivalent C code.

Overall, the lexer and parser are essential components of the Bash to C compilation process, and they play a critical role in ensuring the correctness and reliability of the resulting C code.

LEXER OVERVIEW:

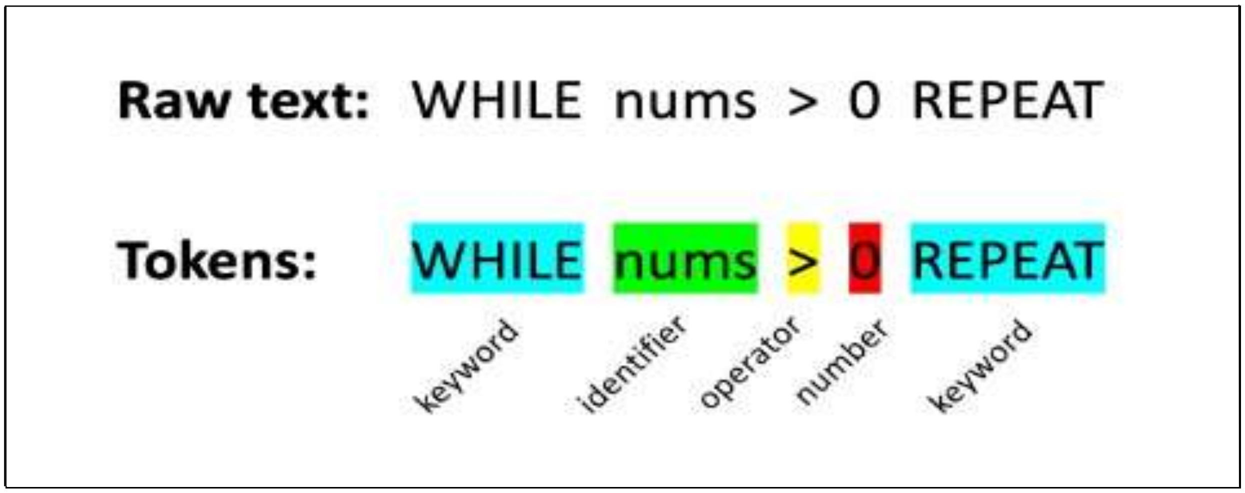

Raw text: WHILE nums > 0 REPEAT
Tokens: WHILE nums > 0 REPEAT
keyword
identifier
operator
number
keyword

Figure 2.2

PARSER OVERVIEW:

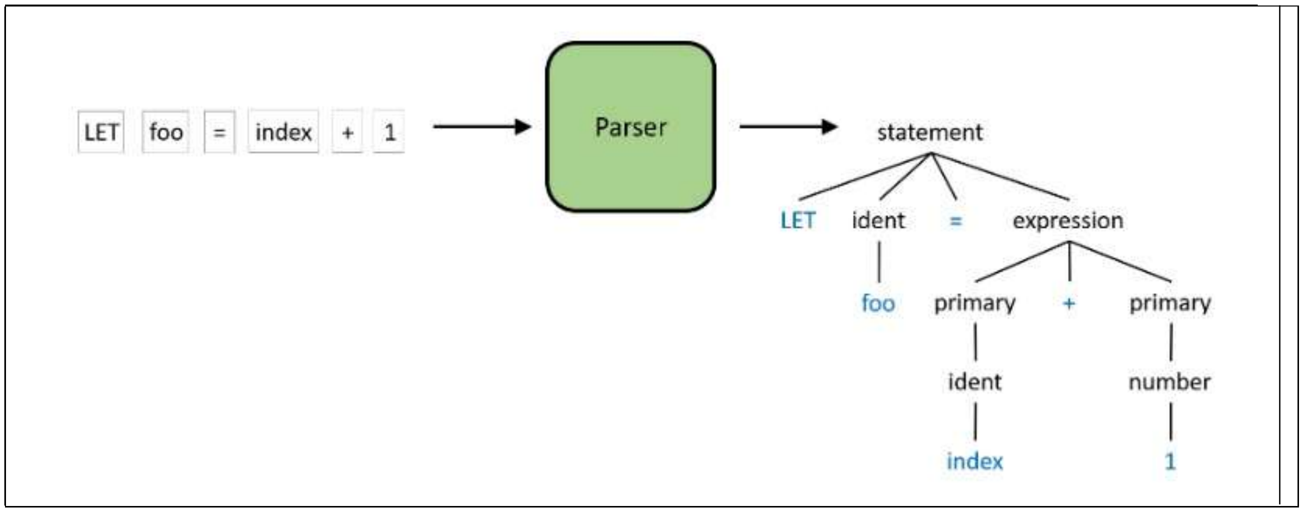


Figure 2.3

CHAPTER 3

SYSTEM ARCHITECTURE AND DESIGN

3.1 FRONT-END DESIGN:-

For the Front-End Framework we have use Bootstrap, overview of Bootstrap:-

1. **Bootstrap Framework:** Bootstrap is also the name of a popular front-end development framework used to build responsive and mobile-first websites. The Bootstrap framework consists of various components and tools that aid in web development, including:

2. **HTML/CSS Components:** Bootstrap provides a collection of pre-designed HTML and CSS components, such as buttons, forms, navigation bars, and grids. These components can be easily integrated into web pages to ensure consistency and responsiveness.

3. **JavaScript Plugins:** Bootstrap offers a set of JavaScript plugins that add functionality and interactivity to web pages. These plugins include features like carousels, modals, tooltips, and dropdown menus.

4. **Responsive Grid System:** Bootstrap includes a responsive grid system that enables developers to create flexible and responsive layouts for web pages. The grid system helps in achieving a consistent look and feel across different devices and screen sizes.

5. **Bootstrapping a Compiler or Interpreter:** In the context of compiler or interpreter design, bootstrapping refers to the process of implementing a compiler or interpreter for a programming language using the same language itself. The bootstrapping process typically involves the following stages:

6. **Initial Compiler/Interpreter:** A basic version of the compiler or interpreter is written in a different language (often a lower-level language or an existing language). It is used to compile or interpret the subsequent versions of the compiler or interpreter.

7. **Self-Compilation:** The initial compiler or interpreter is used to compile or interpret an updated version of itself written in the target language. Iterative Refinement: The process is repeated, using each new version to compile or interpret a more advanced version until the final compiler or interpreter is achieved.

Overall, the components of bootstrap vary depending on the specific context, such as the system or software application being bootstrapped. It can involve components like the bootloader, operating system kernel, application initialization, HTML/CSS components, JavaScript plugins, and self-compilation in the case of compiler or interpreter design.

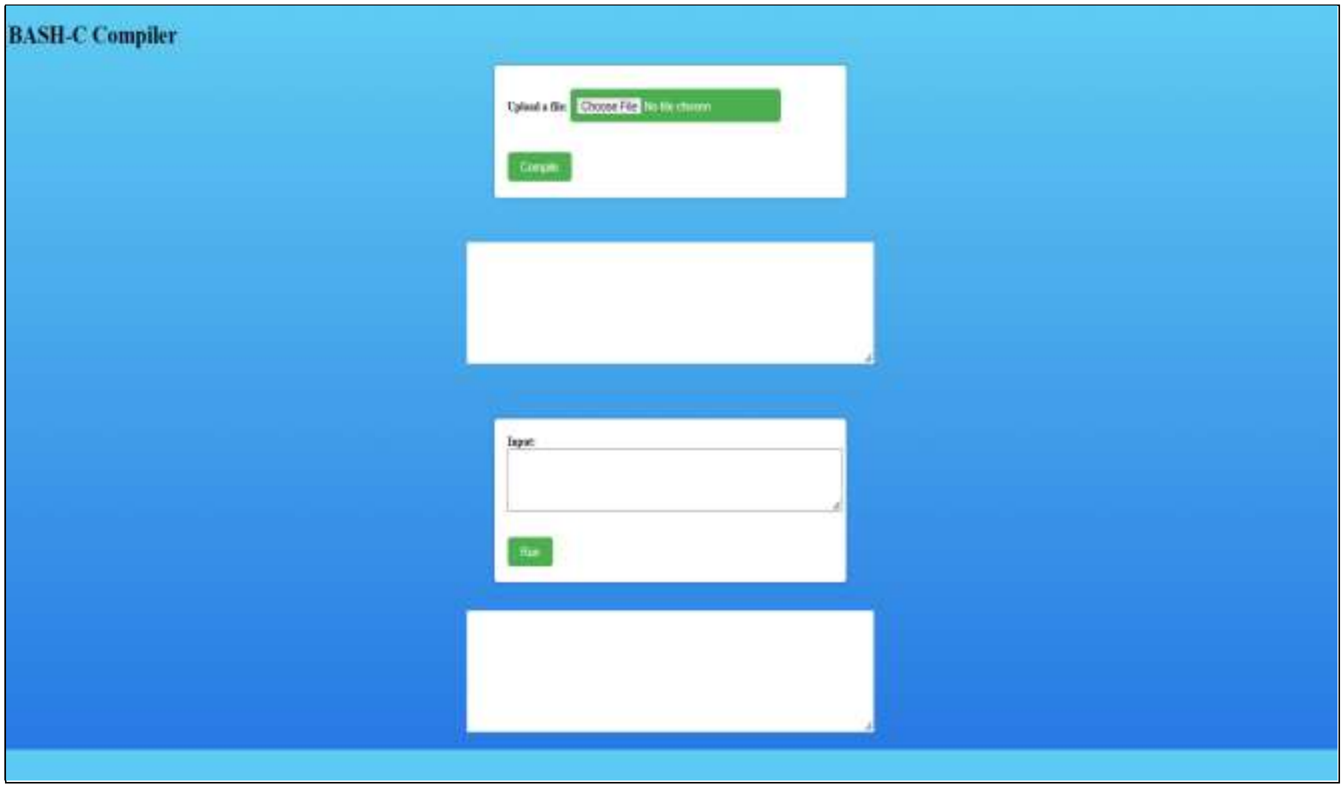


Figure 3.1

FRONT-END ARCHITECTURE DESIGN:-

Figure 3.2

3.2 BACK-END DESIGN:-

The back end of the Bash to C compiler system will process user inputs, execute the lexer, parser, code generator, and compiler, and return the results to the user. The back end will be implemented using the Flask web framework and should include the following components:

- **Routing**: The Flask server should include routing rules that map URLs to corresponding functions in the system. For example, a route could be defined to handle the conversion of Bash scripts to C code, while another route could be defined to handle the compilation of the resulting C code.

- **Views**: The Flask server should include views that generate the HTML templates and render the user interface for the front end. The views should be designed to be modular and extensible, allowing for the addition of new functionality in the future.

- **Model:** The Flask server should include a data model that represents the Bash scripts, compiler options, and other data relevant to the system. The model should be designed to be persistent, allowing users to save their scripts and compiler options for future use.

- **Controller**: The Flask server should include a controller that handles user inputs, performs input validation, and executes the appropriate functions in the system. The controller should be designed to handle errors and provide informative feedback to the user.

- **Lexer**: The system should include a lexer component that tokenizes the Bash script and generates a stream of tokens that can be processed by the parser.

- **Parser**: The system should include a parser component that parses the token stream generated by the lexer and generates an abstract syntax tree (AST).

BASH-C ARCHITECTURE DESIGN:-

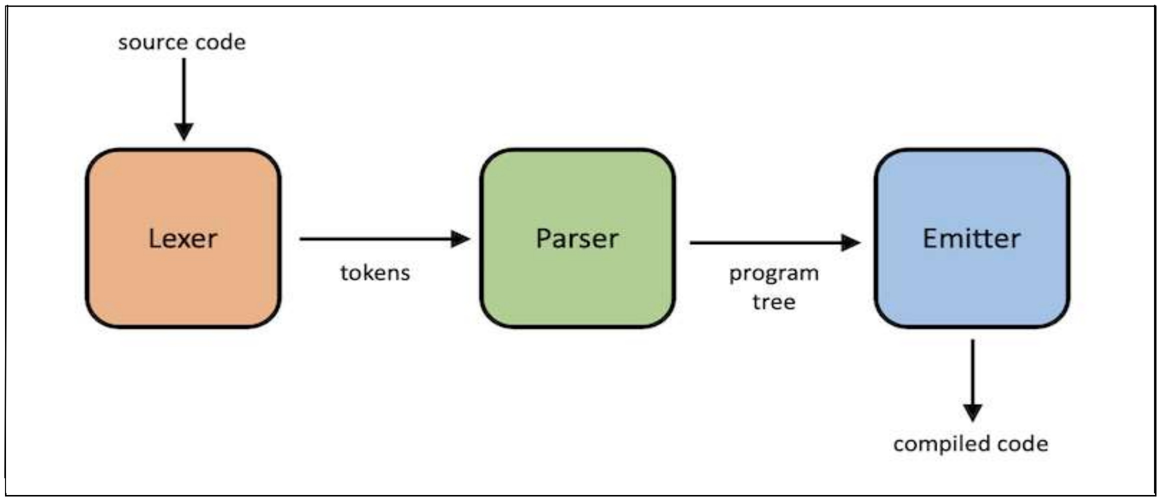


Figure 3.3

COMPONENT ARCHITECTURE:

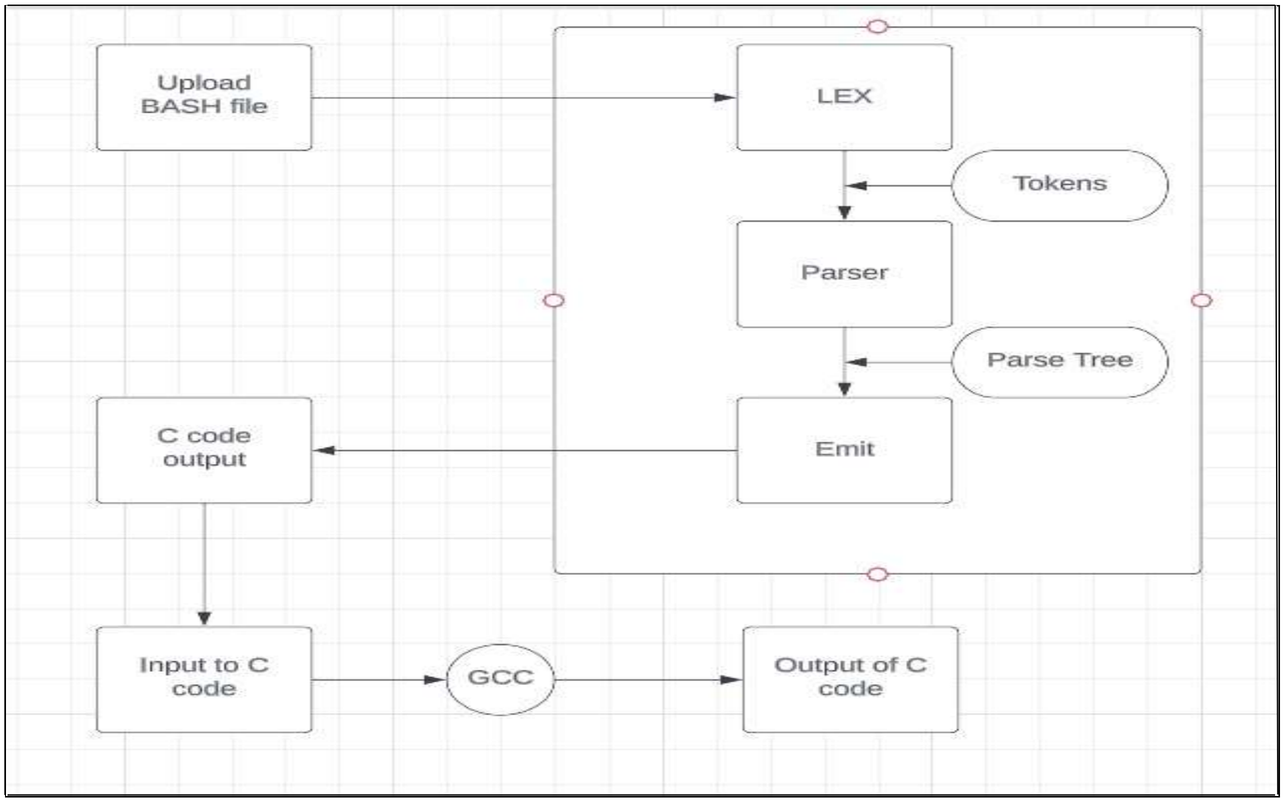


Figure 3.4

Description:

The system architecture for the Bash to C compiler can be divided into three main components: the front-end interface, the backend, and the GCC compiler.

The front-end interface will consist of a web-based user interface that accepts Bash scripts as input and sends them to the backend for processing. The front-end interface will be developed using the Flask web framework, and will include a file upload mechanism for users to submit their Bash scripts.

The backend will handle the compilation process and will be responsible for generating the C source code from the Bash script. The backend will include a lexer and parser to analyze the Bash script and generate the corresponding C code. The backend will be implemented using Python.
The GCC compiler will be used to generate the executable C code from the generated C source code. The compiled C code will be executed and the output will be returned to the user via the front-end interface.

The system architecture can be further broken down into the following components:

Front-end interface:
Flask web framework
HTML/CSS/JavaScript for user interface design

Backend:
Lexer and parser to analyze Bash script and generate C code
Python programming language

GCC compiler:
Compiles C source code generated by the backend
Produces executable binary
Executes binary and returns output to the backend
The system architecture is designed to be modular, allowing for easy maintenance and scalability. It also provides a flexible and efficient solution for converting Bash scripts to C code and compiling them on a variety of platforms.

CHAPTER 4

The requirement to run the script -

To run a BASH-C compiler built using Flask and JavaScript, you will need the following requirements:

1. **A compatible version of Python:** Flask is a web framework for Python, so you will need to have Python installed on your computer. The specific version of Python required may vary depending on the version of Flask and other dependencies used in the code.

2. **Flask and its dependencies:** Flask is a third-party library for Python, and it has several dependencies that must be installed to use it. These dependencies may include Werkzeug, Jinja2, and others. You can use a package manager like pip to install these dependencies.

3. **A web server:** Flask is a web framework, and it requires a web server to run. You can use the built-in development server that comes with Flask, or you can use a production-ready web server like Apache or Nginx.

4. **A browser:** The JavaScript code used in the semantic analyzer will be executed in the user's browser, so you will need a modern web browser like Chrome, Firefox, or Safari to view and interact with the analyzer.

5. **Code editor or IDE:** To edit the Flask and JavaScript code, you will need a code editor or integrated development environment (IDE) that supports Python and JavaScript.

CHAPTER 5

CODING AND TESTING

CODING:-

1. **<u>LEX.PY:-</u>**

```
import sys
import enum

# Lexer object keeps track of current position in the source code and produces each token.
class Lexer:
    def _init_(self, source):
        self.source = source + '\n' # Source code to lex as a string. Append a newline to
simplify lexing/parsing the last token/statement.
        self.curChar = ''   # Current character in the string.
        self.curPos = -1    # Current position in the string.
        self.nextChar()

    # Process the next character.
    def nextChar(self):
        self.curPos += 1
        if self.curPos >= len(self.source):
            self.curChar = '\0'  # EOF
        else:
            self.curChar = self.source[self.curPos]

    # Return the lookahead character.
    def peek(self):
        if self.curPos + 1 >= len(self.source):
            return '\0'
        return self.source[self.curPos+1]

    # Invalid token found, print error message and exit.
    def abort(self, message):
        sys.exit("Lexing error. " + message)

    # Return the next token.
    def getToken(self):
        self.skipWhitespace()
        self.skipComment()
```

```
elif self.curChar == '/':
            token = Token(self.curChar, TokenType.SLASH)
        elif self.curChar == '=':
            # Check whether this token is = or ==
            if self.peek() == '=':
                lastChar = self.curChar
                self.nextChar()
                token = Token(lastChar + self.curChar, TokenType.EQEQ)
            else:
                token = Token(self.curChar, TokenType.EQ)
        elif self.curChar == '>':
            # Check whether this is token is > or >=
            if self.peek() == '=':
                lastChar = self.curChar
                self.nextChar()
                token = Token(lastChar + self.curChar, TokenType.GTEQ)
            else:
                token = Token(self.curChar, TokenType.GT)
        elif self.curChar == '<':
            # Check whether this is token is < or <=
            if self.peek() == '=':
                lastChar = self.curChar
                self.nextChar()
                token = Token(lastChar + self.curChar, TokenType.LTEQ)
            else:
                token = Token(self.curChar, TokenType.LT)
        elif self.curChar == '!':
            if self.peek() == '=':
                lastChar = self.curChar
                self.nextChar()
                token = Token(lastChar + self.curChar, TokenType.NOTEQ)
            else:
                self.abort("Expected !=, got !" + self.peek())

        elif self.curChar == '\"':
            # Get characters between quotations.
            self.nextChar()
            startPos = self.curPos

            while self.curChar != '\"':
                # Don't allow special characters in the string. No escape characters, newlines,
tabs, or %.
                # We will be using C's printf on this string.
                if self.curChar == '\r' or self.curChar == '\n' or self.curChar == '\t' or self.curChar
== '\\' or self.curChar == '%':
                    self.abort("Illegal character in string.")
                self.nextChar()
```

2. MAIN.PY:-

```
from flask import Flask, render_template, request
from lex import *
from emit import *
from parse import *
import os
import subprocess

app = Flask(_name_)

UPLOAD_FOLDER = os.path.join(os.getcwd(), 'uploads')
if not os.path.exists(UPLOAD_FOLDER):
    os.makedirs(UPLOAD_FOLDER)

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/compile', methods=['POST'])
def compile():
    file = request.files['file']
    file.save(os.path.join(UPLOAD_FOLDER, file.filename))
    print(os.listdir(UPLOAD_FOLDER)) # <-- add this line to check the contents of the
uploads directory
    with open(os.path.join(UPLOAD_FOLDER, file.filename), 'r') as inputFile:
        source = inputFile.read()

    # Initialize the lexer, emitter, and parser.
    lexer = Lexer(source)
    emitter = Emitter("out.c")
    parser = Parser(lexer, emitter)

    parser.program() # Start the parser.
    emitter.writeFile() # Write the output to file.

    with open("out.c", 'r') as outputFile:
        output = outputFile.read()

    return render_template('index.html', output=output)

@app.route('/run', methods=['POST'])
def run_code():
    input_data = request.form['input']
```

3. **PARSER.PY:-**

```
import sys
from lex import *

# Parser object keeps track of current token, checks if the code matches the grammar, and
emits code along the way.
class Parser:
    def _init_(self, lexer, emitter):
        self.lexer = lexer
        self.emitter = emitter

        self.symbols = set()    # All variables we have declared so far.
        self.labelsDeclared = set() # Keep track of all labels declared
        self.labelsGotoed = set() # All labels goto'ed, so we know if they exist or not.

        self.curToken = None
        self.peekToken = None
        self.nextToken()
        self.nextToken()    # Call this twice to initialize current and peek.

    # Return true if the current token matches.
    def checkToken(self, kind):
        return kind == self.curToken.kind

    # Return true if the next token matches.
    def checkPeek(self, kind):
        return kind == self.peekToken.kind

    # Try to match current token. If not, error. Advances the current token.
    def match(self, kind):
        if not self.checkToken(kind):
            self.abort("Expected " + kind.name + ", got " + self.curToken.kind.name)
        self.nextToken()
if self.curToken.text not in self.symbols:
                self.symbols.add(self.curToken.text)
                self.emitter.headerLine("float " + self.curToken.text + ";")

            self.emitter.emit(self.curToken.text + " = ")
            self.match(TokenType.IDENT)
            self.match(TokenType.EQ)

            self.expression()
            self.emitter.emitLine(";")
```

TESTING

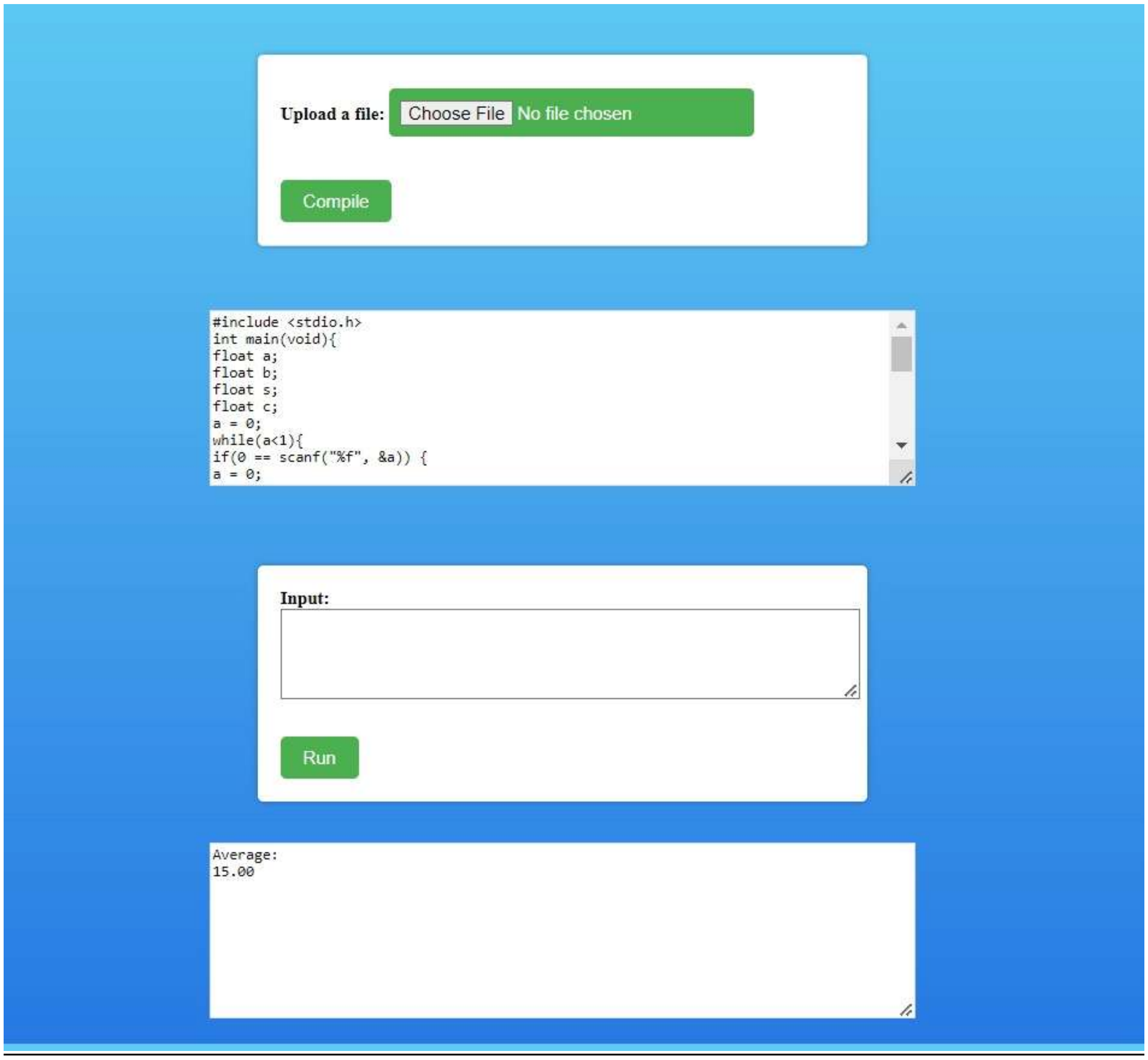
TEST CASE 1–

Bash Input:

```
LET a = 0
WHILE a < 1 REPEAT
   INPUT a
ENDWHILE

LET b = 0
LET s = 0
WHILE b < a REPEAT
   INPUT c
   LET s = s + c
   LET b = b + 1
ENDWHILE

PRINT "Average: "
PRINT s / a
```

C output:

```
#include <stdio.h>
int main(void){
float a;
float b;
float s;
float c;
a = 0;
while(a<1){
if(0 == scanf("%f", &a)) {
a = 0;
scanf("%*s");}}
b = 0;
s = 0;
while(b<a){
if(0 == scanf("%f", &c)) {
c = 0;
scanf("%*s");}
s = s+c;
b = b+1;}
printf("Average: \n");
printf("%.2f\n", (float)(s/a));
return 0;}
```



```
#include <stdio.h>
int main(void){
float a;
float b;
float s;
float c;
a = 0;
while(a<1){
if(0 == scanf("%f", &a)) {
a = 0;
```

```
Average:
15.00
```

Figure 5.1

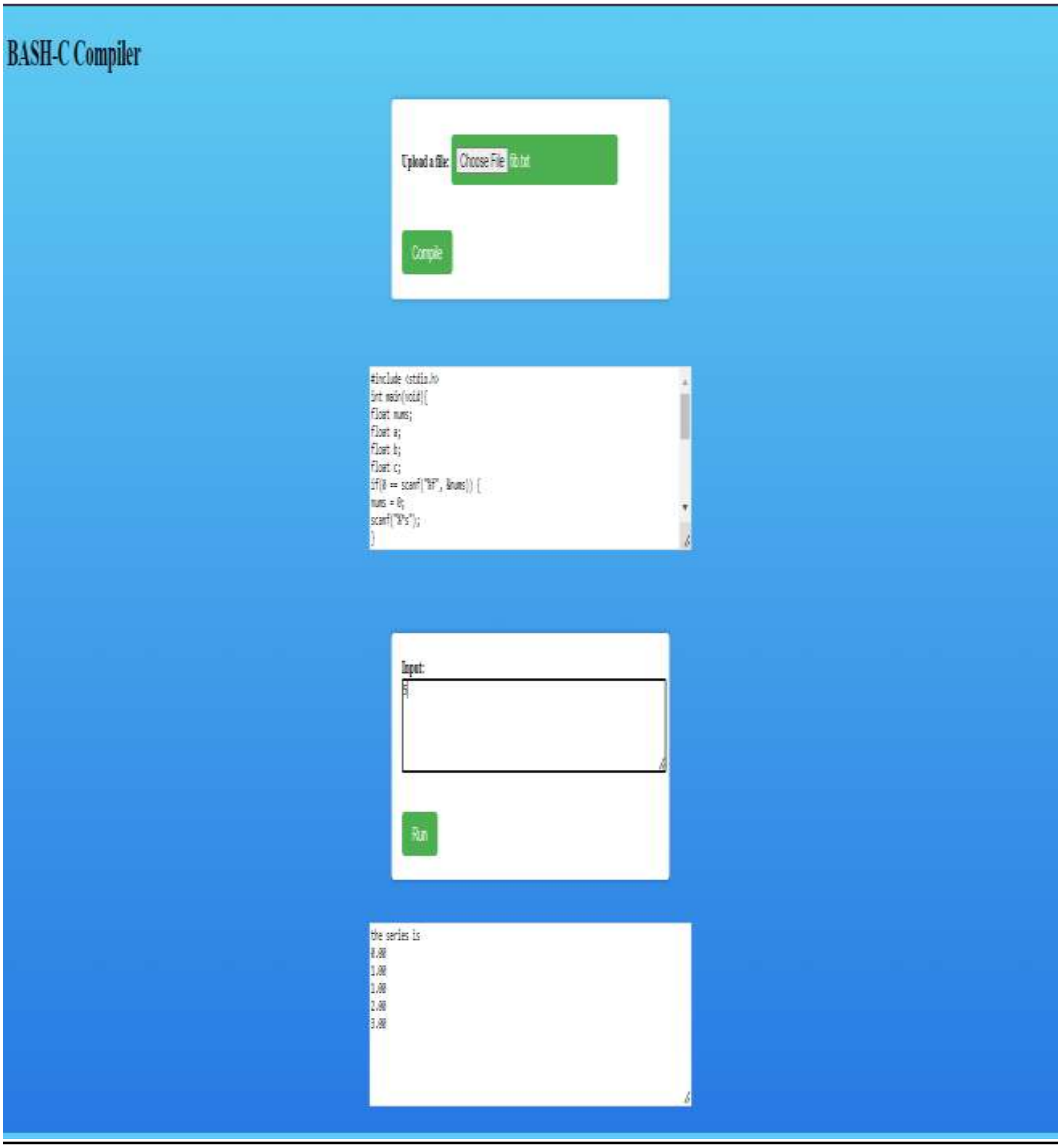
TEST CASE 2–

Bash Input:

```
INPUT nums
PRINT "the series is "

LET a = 0
LET b = 1
WHILE nums > 0 REPEAT
    PRINT a
    LET c = a + b
    LET a = b
    LET b = c
    LET nums = nums - 1
ENDWHILE
```

C output:

```
#include <stdio.h>
int main(void){
float nums;
float a;
float b;
float c;
if(0 == scanf("%f", &nums)) {
nums = 0;
scanf("%*s");
}
printf("the series is \n");
a = 0;
b = 1;
while(nums>0){
printf("%.2f\n", (float)(a));
c = a+b;
a = b;
b = c;
nums = nums-1;
}
return 0;
}
```



```
#include <stdio.h>
int main(void){
float nums;
float a;
float b;
float c;
if(0 == scanf("%f", &nums)) {
nums = 0;
scanf("%*s");
}
```

```
5
```

```
the series is
0.00
1.00
1.00
2.00
3.00
```

Figure 5.2

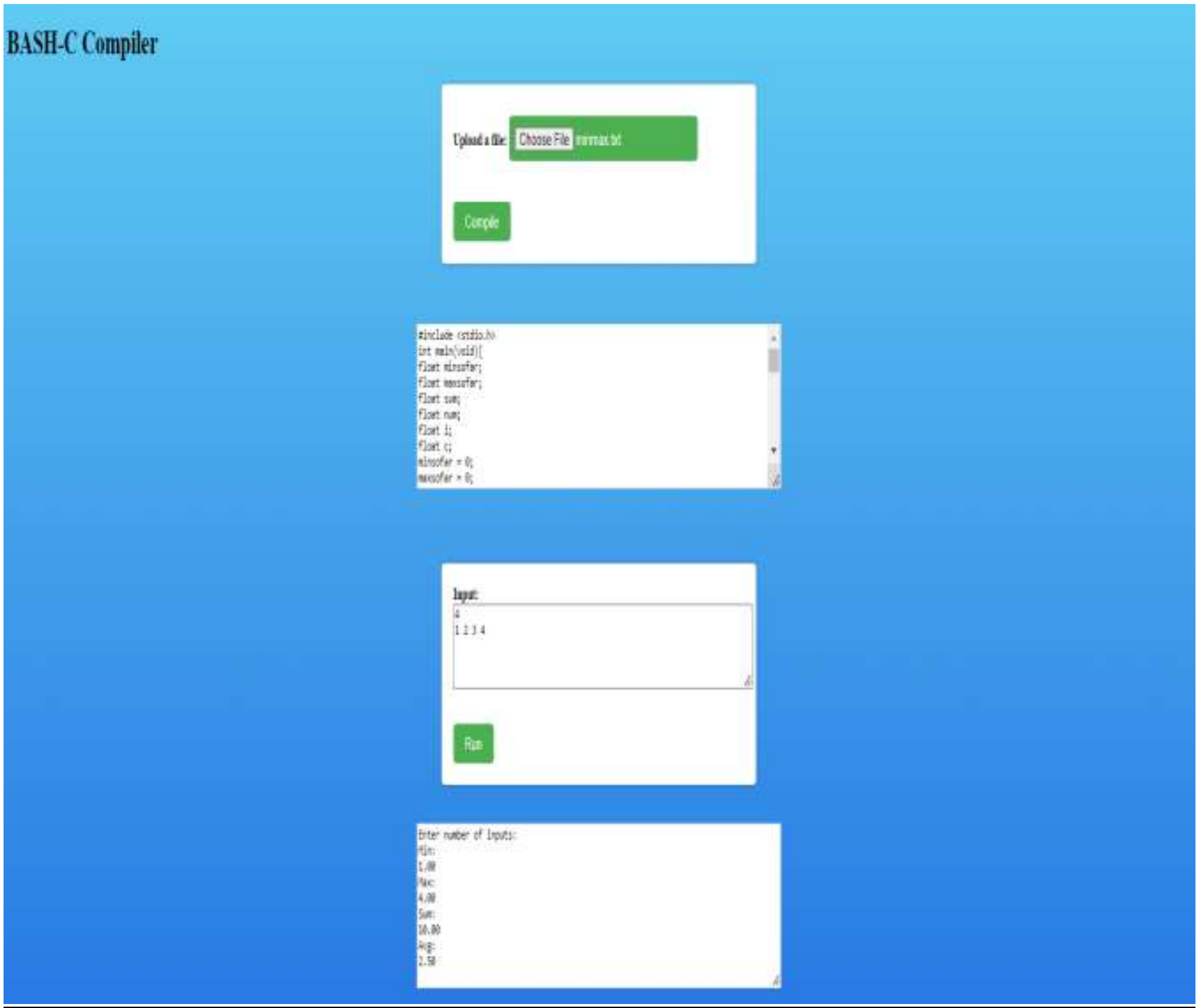
<u>TEST CASE 3-</u>

<u>Bash Input:</u>

```
# Take an arbitrary number of inputs an return the minimum, maximum, sum, and
average.

LET minsofar = 0
LET maxsofar = 0
LET sum = 0

LET num = 0
PRINT "Enter number of inputs: "
INPUT num

LET i = 0
LET c = 0
WHILE i < num REPEAT
    INPUT c
    IF i == 0 THEN
        LET minsofar = c
        LET maxsofar = c
    ENDIF
    IF i != 0 THEN
        IF c < minsofar THEN
            LET minsofar = c
        ENDIF
        IF c > maxsofar THEN
            LET maxsofar = c
        ENDIF
    ENDIF
    LET sum = sum + c
    LET i = i + 1
ENDWHILE

PRINT "Min: "
PRINT minsofar
PRINT "Max: "
PRINT maxsofar
PRINT "Sum: "
PRINT sum
PRINT "Avg: "
PRINT sum / num
```

<u>C output:</u>

```
#include <stdio.h>
int main(void){
float minsofar;
float maxsofar;
float sum;
float num;
float i;
float c;
minsofar = 0;
maxsofar = 0;
sum = 0;
num = 0;
printf("Enter number of inputs: \n");
if(0 == scanf("%f", &num)) {
num = 0;
scanf("%*s");
}
i = 0;
c = 0;
while(i<num){
if(0 == scanf("%f", &c)) {
c = 0;
scanf("%*s");
}
if(i==0){
minsofar = c;
maxsofar = c;
}
if(i!=0){
if(c<minsofar){
minsofar = c;
}
if(c>maxsofar){
maxsofar = c;
}
}
sum = sum+c;
i = i+1;
}
printf("Min: \n");
printf("%.2f\n", (float)(minsofar));
printf("Max: \n");
printf("%.2f\n", (float)(maxsofar));
printf("Sum: \n");
printf("%.2f\n", (float)(sum));
printf("Avg: \n");
printf("%.2f\n", (float)(sum/num));
return 0;}
```



```
#include <stdio.h>
int main(void){
float minsofar;
float maxsofar;
float sum;
float num;
float i;
float c;
minsofar = 0;
maxsofar = 0;
```

```
4
1 2 3 4
```

```
Enter number of Inputs:
Min:
1.00
Max:
4.00
Sum:
10.00
Avg:
2.50
```

Figure 5.3

CHAPTER 6
RESULT

The result of a Bash to C compiler is a C program that is functionally equivalent to the original Bash script. The C program can be compiled using a standard C compiler, and can then be executed on any platform that supports C.

The advantages of using a Bash to C compiler are:

- Portability: Since C is a widely supported and portable language, the resulting C program can be compiled and executed on a wide variety of platforms, without requiring the installation of any additional software.

- Performance: C is a compiled language, which means that the resulting executable code is typically faster and more efficient than interpreted Bash scripts.

- Maintainability: C code is generally more readable and maintainable than Bash scripts, which can become complex and difficult to maintain as they grow in size and complexity.

- Access to C libraries: C provides access to a wide range of libraries and system functions that may not be available in Bash, allowing for greater flexibility and functionality in the resulting program.

Overall, the result of a Bash to C compiler is a more efficient, portable, and maintainable program that is easier to work with and provides greater functionality than the original Bash script.

CHAPTER 7

CONCLUSION

In conclusion, the Bash to C compiler is a powerful tool that addresses many of the limitations and challenges of traditional Bash scripting. By converting Bash scripts to C code and compiling it, the resulting code can be executed much faster, making it more suitable for resource-intensive applications. The resulting code is also more portable, maintainable, and secure, enhancing the overall reliability and security of shell scripts.

The Bash to C compiler project has a clear set of objectives and requirements, including the development of a lexer, parser, code generator, and compiler. The resulting system can be designed using a modular and extensible architecture, allowing for the addition of new features and functionality in the future.

Overall, the Bash to C compiler project represents an exciting opportunity to improve the performance, portability, maintainability, and security of Bash scripts, making them more suitable for a wide range of applications. With careful planning and design, the Bash to C compiler can be a powerful tool for developers and system administrators, helping them to write more efficient, flexible, and reliable shell scripts

CHAPTER 8
REFERENCES

- https://www.gnu.org/software/bash/manual/bash.html
- https://www.iso.org/standard/71691.html
- https://www.amazon.com/Compilers-Principles-Techniques-Tools-2nd/dp/0321486811
- https://flask.palletsprojects.com/en/2.0.x/
- https://docs.python.org/3/
- https://gcc.gnu.org/onlinedocs/gcc/
- https://llvm.org/docs/
- https://github.com/neechbear/bash-to-c-compiler
- https://dickgrune.com/Books/PTAPG_1st_Edition/
- https://www.amazon.com/Programming-Language-2nd-Brian-Kernighan/dp/0131103628