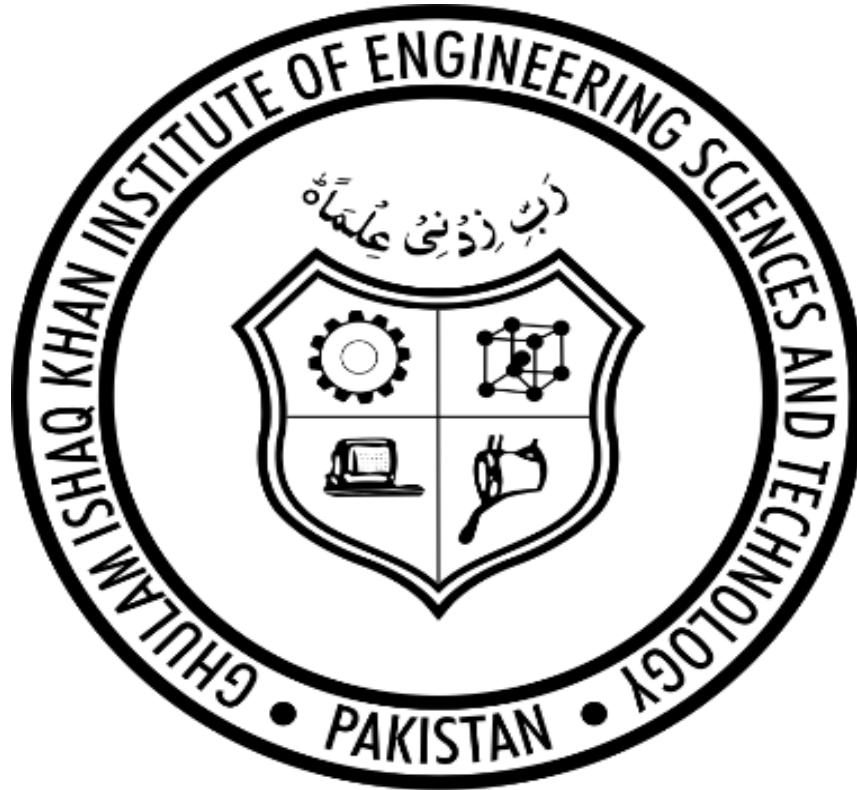


Ghulam Ishaq Khan Institute of Engineering Sciences and
Technology, Topi



CS 424 (Compiler Construction)

Course Project

**Project Title: Design and Implementation of a Compiler for a C-like Programming
Language**

Submitted by
Zartaj Asim 2020526

Table of Contents

1. Introduction
 - 1.1. Language Specification
 - 1.2. Grammar
 - 1.3. Supported Features
 - 1.4. Limitations
2. Conversion to automata
 - 2.1 Grammar Breakdown and FA Conversion
 - 2.1.1. Keywords
 - 2.1.2. Identifiers
 - 2.1.3. Integers and Floating-Point Numbers
 - 2.1.4. String Literals
 - 2.1.5. Comments
 - 2.1.6. Operators
3. Compiler Design
 - 3.1. Lexical Analysis
 - 3.2. Syntax Analysis
 - 3.3. Test Cases
 - 3.4. Lexer and Parser Testing Results
4. Semantic Analysis
 - 4.1. Type Checking
 - 4.2. Variable Declaration
5. Intermediate Code Generation
 - 5.1. Code Generation Functions
 - 5.2. Testing
6. Code Optimization
 - 6.1. Intermediate Code Optimization
 - 6.2. Code Generation Optimization
7. Code Generation
 - 7.1. Outputs for Code Generation
8. Challenges

1. Introduction

This report documents the design and implementation of a compiler for a C-like programming language. The primary goal of this project is to develop a robust compiler capable of translating source code written in the specified language into executable machine code. Throughout this report, we detail the various stages of the compiler's development, including language specification, design choices, implementation strategies, and testing methodologies.

1.1. Language Specification

The grammar rules provided encapsulate the structure of the language, ranging from the top-level program definition to the smallest atomic units like identifiers and numerical constants. Additionally, the specification delineates supported features such as variable and function declarations, control flow statements (e.g., if-else, while), expressions, and function calls.

1.2. Grammar

program	-> declaration_list
declaration_list	-> declaration declaration_list declaration
declaration	-> variable_declaration function_declaration
variable_declaration	-> type_specifier ID ;
function_declaration	-> type_specifier ID (parameters) compound_statement
parameters	-> parameter_list VOID
parameter_list	-> parameter parameter_list , parameter
parameter	-> type_specifier ID
compound_statement	-> { statement_list }
statement_list	-> statement statement_list statement
statement	-> expression_statement compound_statement selection_statement iteration_statement return_statement
expression_statement	-> expression ;

selection_statement -> IF (expression) statement | IF (expression) statement
ELSE statement

iteration_statement -> WHILE (expression) statement

return_statement -> RETURN expression ;

expression -> variable = expression | simple_expression

variable -> ID

simple_expression -> additive_expression relop additive_expression | additive_expression

additive_expression -> additive_expression addop term | term

term -> term mulop factor | factor

factor -> (expression) | variable | call | NUM

call -> ID (args)

args -> arg_list | ϵ

arg_list -> expression | arg_list , expression

1.3. Supported Features

- **Variable and Function Declarations:** The language supports the declaration of variables and functions.
- **Control Flow Statements:** Control flow statements such as if-else and while are supported.
- **Expressions:** The language allows for the formation of expressions, including arithmetic operations, assignments, and function calls.
- **Compound Statements:** Compound statements, enclosed within curly braces, enable the grouping of multiple statements into a single block.
- **Return Statements:** Functions can return values using the return statement.

1.4. Limitations

- **Simplified Syntax:** To facilitate easier implementation, the language adopts a simplified syntax compared to full-fledged C.
- **Limited Data Types:** The language may support a limited set of data types compared to C, restricting the range of data that can be manipulated within the program.
- **No Support for Advanced Features:** Advanced language features such as object-oriented programming, dynamic memory allocation, or advanced data structures may not be supported.

2. Conversion to Automata

We transformed a context-free grammar (CFG) for a C-like language into finite automata (FAs) to facilitate lexical analysis. By identifying the terminal symbols defined in the grammar, such as keywords, identifiers, integers, floating-point numbers, and operators, we developed specific FAs to recognize these patterns. Each FA was designed to handle the sequences of characters corresponding to these terminals, ensuring accurate token recognition.

2.1. Grammar Breakdown and FA Conversion

1. Keywords (e.g., if, else, while, return)

Grammar Rules

selection_statement -> IF (expression) statement | IF (expression) statement ELSE statement

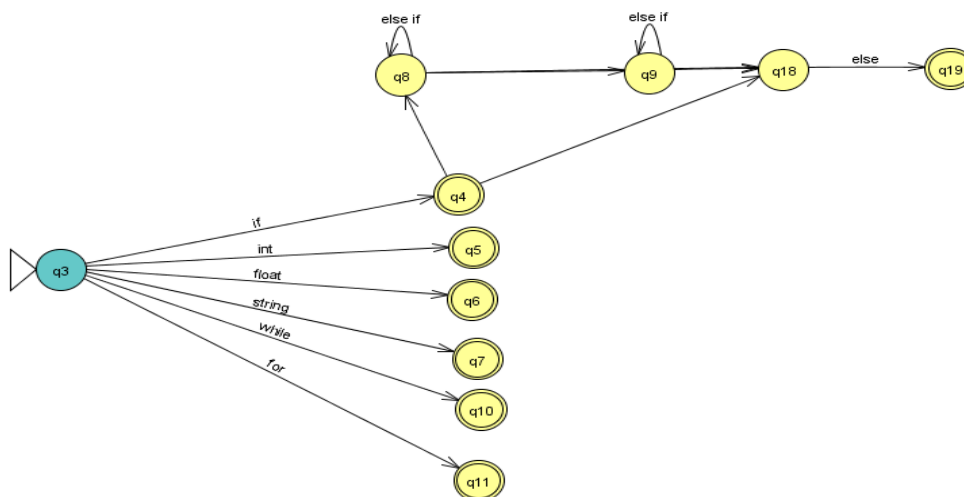
iteration_statement -> WHILE (expression) statement

return_statement -> RETURN expression ;

FA Conversion

For each keyword, we create an FA that recognizes the specific string (e.g., "if", "else", "while", "return"). The process involves transitioning through states corresponding to each character in the keyword.

Finite Automata



Test Inputs

Table Text Size	
Input	Result
int	Accept
float	Accept
string	Accept
while	Accept
for	Accept
if	Accept
if else	Accept
if else if	Reject
if else if else if	Reject
if else if else if else	Accept
if else else	Reject

int: The FA transitions through states corresponding to each character ('i', 'n', 't') and reaches an accepting state after processing the entire input, indicating that "int" is recognized as a keyword. Thus, the input is accepted. Similarly, for float and string.

if else if else if: The FA would transition to an accepting state after recognizing "if" and remain there, unable to recognize subsequent "else if" or "else" sequences. Thus, it would reject the input.

2. Identifiers

Grammar Rules

variable_declaration \rightarrow type_specifier ID ;

parameter \rightarrow type_specifier ID

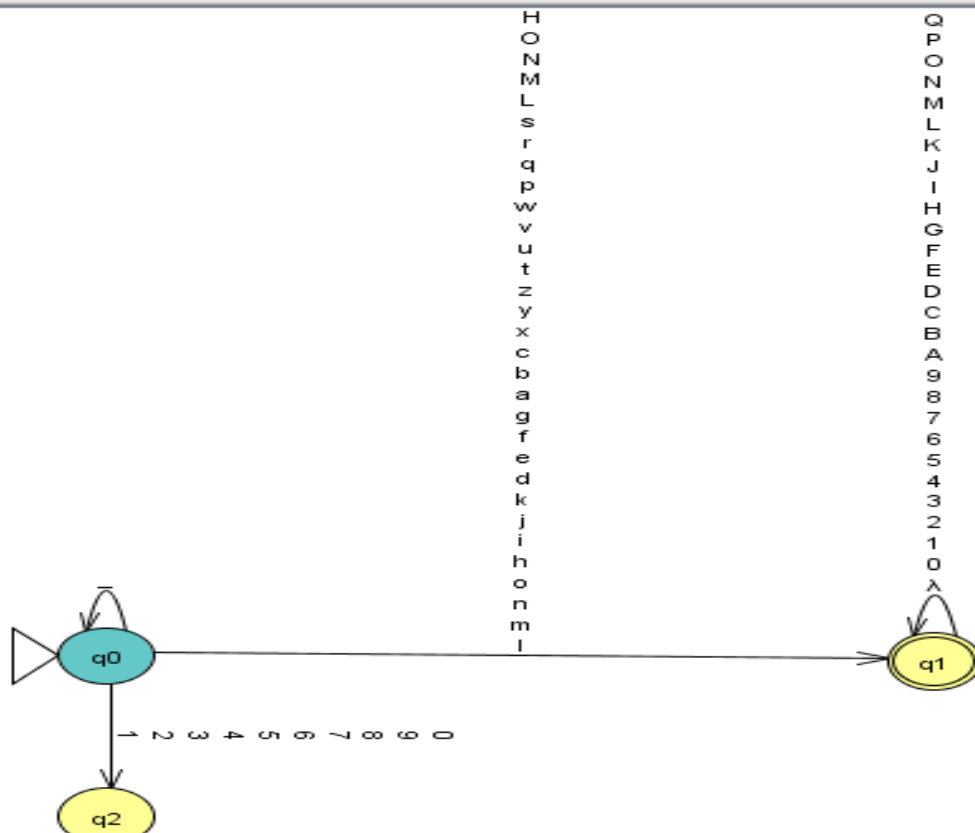
variable \rightarrow ID

call \rightarrow ID (args)

FA Conversion

The FA for identifiers starts in an initial state, accepts an underscore or letter, transitions to an accepting state, and remains there for any combination of letters, digits, and underscores.

Finite Automata



Test Inputs

identifier	Accept
IDENTIFIER	Accept
cOmPlLeR	Accept
compiler123	Accept
_compiler123	Accept
_Compiler	Accept
123Compiler	Reject

In this case, the FA is likely constructed to accept identifiers that start with a letter (either uppercase or lowercase) or an underscore, followed by any combination of letters (uppercase or lowercase), digits, or underscores.

123Compiler: Starts with a digit ('1'), which is not allowed as the initial character for an identifier. The FA rejects it.

3. Integers and Floating-point Numbers (NUM)

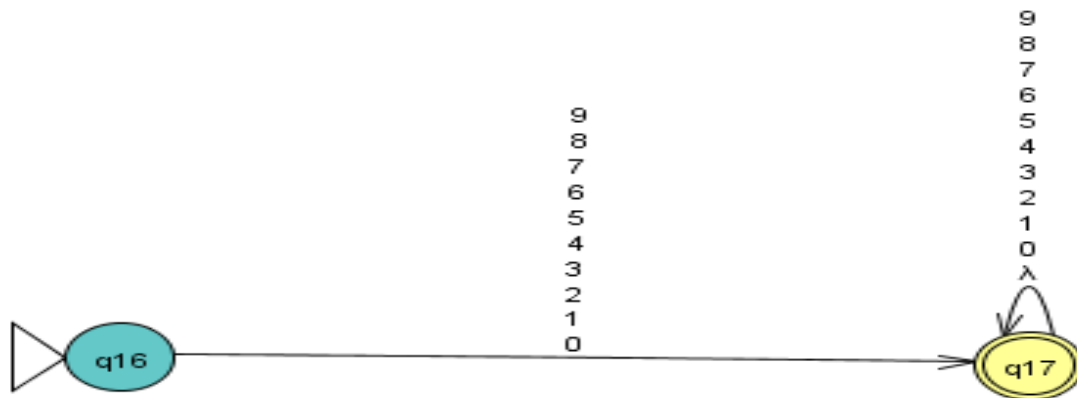
Grammar Rules

factor \rightarrow (expression) | variable | call | NUM

FA Conversion for integers

The FA for integers starts in an initial state, transitions to an accepting state on digits (0-9) and stays there on further digits.

Finite Automata



Test Inputs

1	Accept
123	Accept
1.0	Reject
11.23	Reject
11.	Reject
0.123456789	Reject

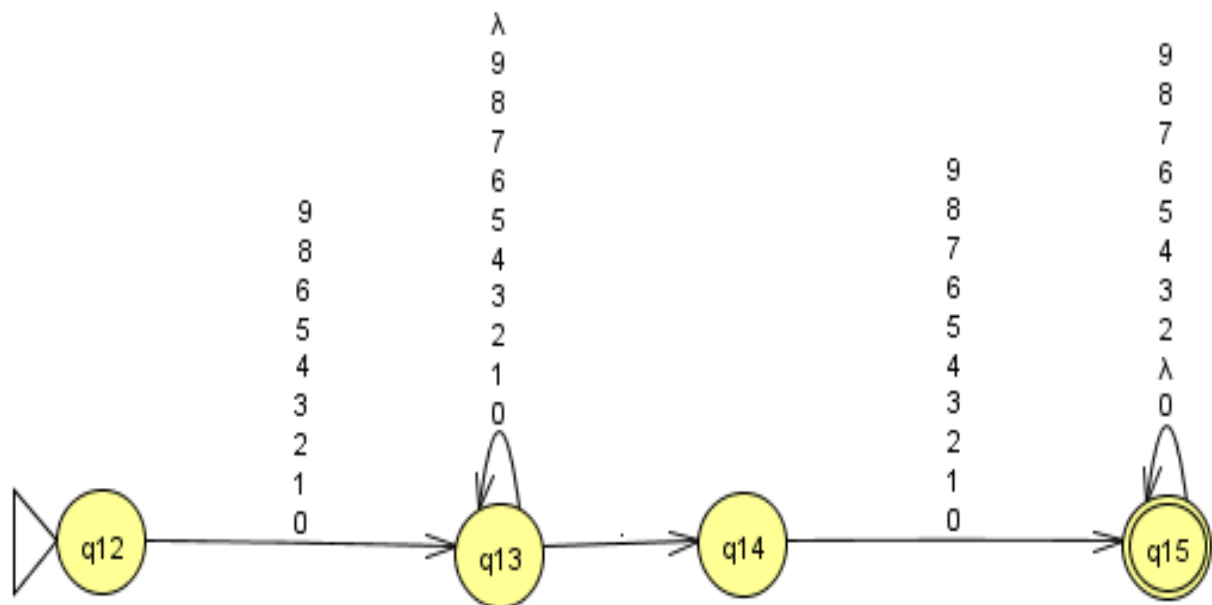
"1" Accepted: The FA for integers is likely designed to accept sequences of digits. The FA transitions through states for each digit and reaches an accepting state, correctly accepting "1" as an integer.

"1.0" Rejected: In contrast, the input "1.0" contains a decimal point ('.'), which does not conform to the rules defined by the FA for integers. The FA is designed to recognize sequences of digits without any fractional parts or decimal points. Upon encountering the decimal point in "1.0", the FA is unable to proceed according to its defined transitions, leading to rejection.

FA Conversion for Floating-point numbers

The FA for floating-point numbers recognizes sequences like "123.456". It involves an initial state, transitions on digits, then a transition on a dot, followed by more digits.

Finite Automata



Test Inputs

1	Reject
123	Reject
1.0	Accept
11.23	Accept
11.	Reject
0.123456789	Accept

"1.0" Accepted: The FA for floating-point numbers is likely designed to accept sequences of digits optionally followed by a decimal point and more digits. Since "1.0" conforms to this pattern, with "1" preceding a decimal point ('.') and "0" following it, the FA transitions through states for each digit and the decimal point, reaching an accepting state and correctly accepting "1.0" as a floating-point number.

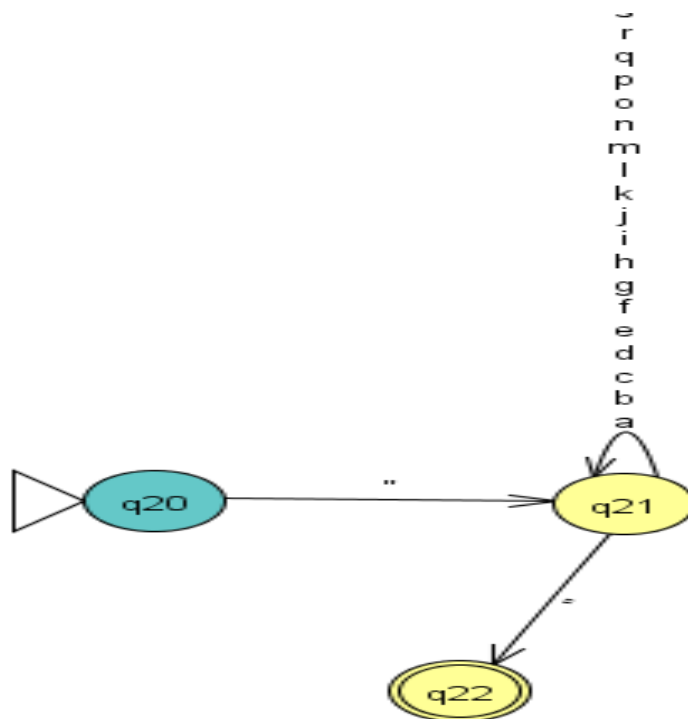
"123" Rejected: In contrast, "123" contains only digits without a decimal point. This input does not match the pattern recognized by the FA for floating-point numbers, which expects at least one digit before and after the decimal point.

4. Strings Literals

FA Conversion

The FA starts in an initial state, transitions on the opening double-quote, moves to a state that accepts any character except a double-quote, and transitions to an accepting state on the closing double-quote.

Finite Automata



Test Inputs

"Hello This is a string "	Accept
---------------------------	--------

The FA for string literals is designed to recognize inputs enclosed within double quotation marks. It includes transitions to handle any printable characters.

5. Comments

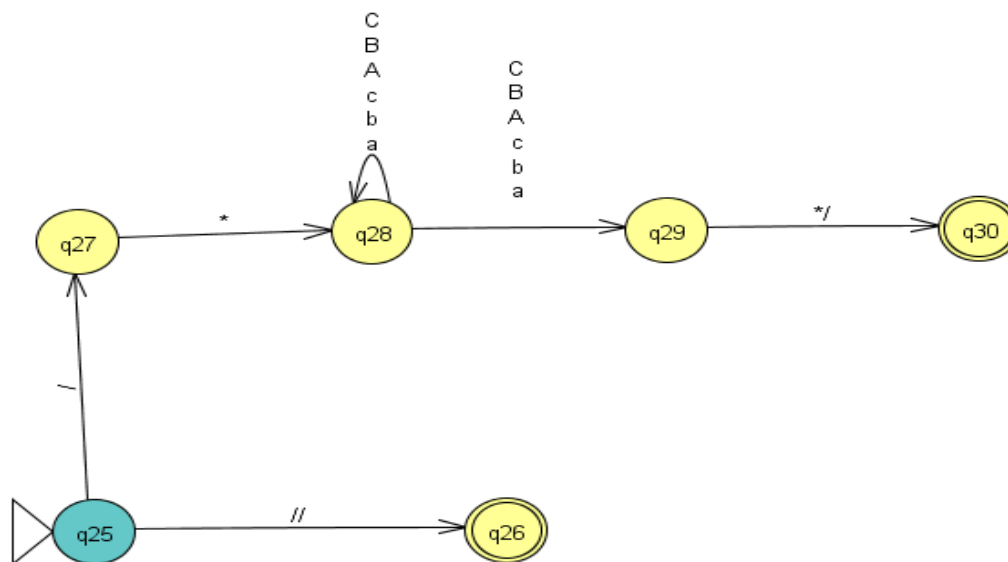
Grammar Rules

Comments are typically ignored during parsing and are not represented in the grammar.

FA Conversion

Single-line comments (`//...`) and multi-line comments (`/*...*/`) are recognized by FAs designed to skip these sequences.

Finite Automata



Test Inputs

//	Accept
/*ABC*/	Accept
/	Reject
/*	Reject
/*abc	Reject

Single-Line Comments ("//"): The FA for comments typically recognizes single-line comments that start with "//" and extend to the end of the line.

Incomplete character / : The FA may not be designed to accept single characters like "/", as they do not form valid comments according to the syntax rules.

Incomplete Multi-Line Comments: Inputs like "/*" and "/*abc" are rejected because they do not include the necessary closing "/" to form a valid multi-line comment.

6. Operators

Grammar Rules

expression \rightarrow variable = expression | simple_expression

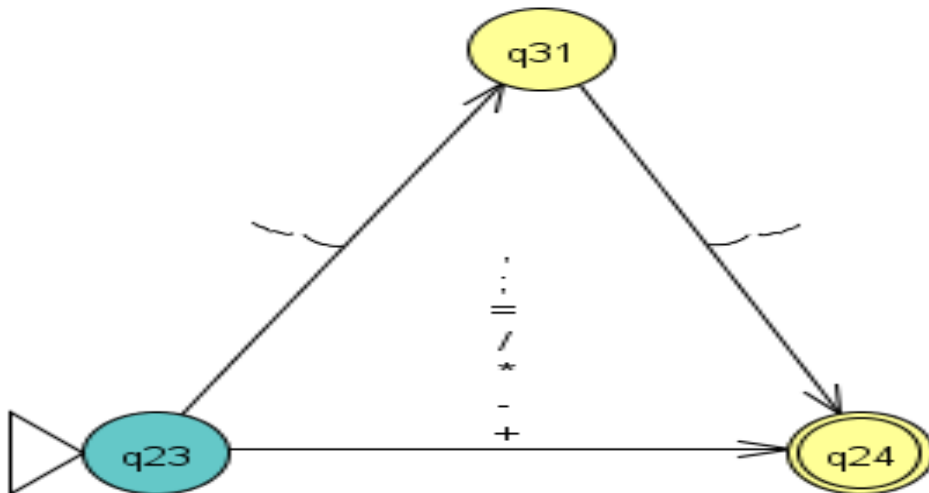
additive_expression \rightarrow additive_expression addop term | term

term \rightarrow term mulop factor | factor

FA Conversion

Operators like =, +, -, *, /, and punctuation like “; , () { }” are handled by simple FAs where each character directly transitions from the start state to an accepting state.

Finite Automata



Test Inputs

+	Accept
,	Accept
-	Accept
++	Reject
*	Accept
.	Accept
()	Accept
{	Reject
}	Accept

The FA for operators and punctuations is designed to recognize individual characters as operators or punctuations. Each character has its own dedicated transition in the FA.

3. Compiler design

3.1. Lexical Analysis

The first phase of the compilation process is lexical analysis, where a lexer is implemented to transform a stream of characters into a sequence of tokens. Lexical analysis is achieved through the implementation of a lexer using regular expressions. Each token type, such as identifiers, keywords, operators, and literals, is defined along with its corresponding regular expression pattern in the `TOKEN_TYPES` list. The lexer iterates over the input string, matching patterns to identify tokens and generating a token stream.

3.2. Syntax Analysis

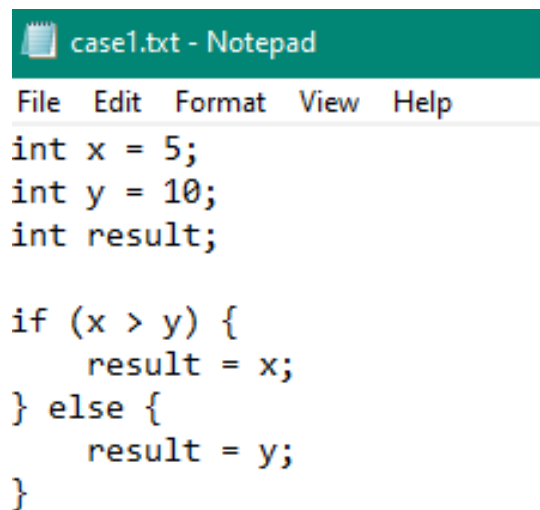
Following lexical analysis, the next phase is syntax analysis, where a parser is developed to construct a parse tree or an abstract syntax tree (AST) from the token stream generated by the lexer. In the provided code, syntax analysis is performed through the development of a recursive descent parser. This parser traverses the token stream, identifying language constructs such as variable declarations, control flow statements, and expressions. For each recognized construct, the parser generates corresponding nodes in the parse tree, organizing them hierarchically based on the language's grammar rules. Specific parsing functions are defined for different language constructs, such as if-else statements, variable declarations, and while loops.

Additionally, the parser employs helper functions to handle specific tasks, such as parsing expressions and blocks of statements enclosed within braces. These functions collaborate to parse the entire input code and generate a structured representation of its syntactic structure in the form of a parse tree.

3.3. Test Cases

To ensure the correctness of the compiler, we have tested it with three different code snippets representing distinct language constructs. Each test case will evaluate the compiler's ability to tokenize input strings, parse them into a parse tree, and produce the expected output.

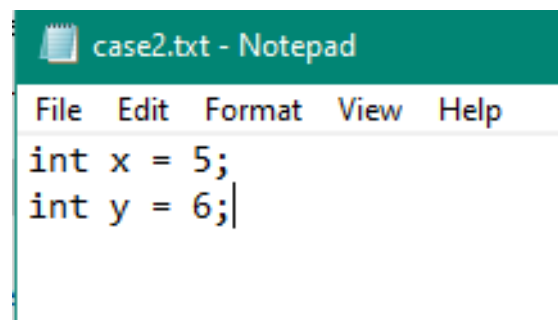
Test Case 1



```
case1.txt - Notepad
File Edit Format View Help
int x = 5;
int y = 10;
int result;

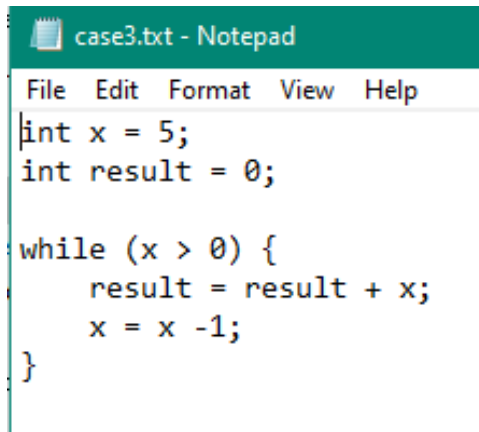
if (x > y) {
    result = x;
} else {
    result = y;
}
```

Test case 2



```
case2.txt - Notepad
File Edit Format View Help
int x = 5;
int y = 6;
```

Test case 3



```
case3.txt - Notepad
File Edit Format View Help
int x = 5;
int result = 0;

while (x > 0) {
    result = result + x;
    x = x - 1;
}
```

3.4. Lexer and Parser Testing Results

Test Case 1

In this test case, the lexer identifies tokens representing variable declarations ('int', 'x', 'y', 'result'), numerical literals ('5', '10'), assignment operators ('='), and control flow constructs ('if', 'else'). The parser then constructs a parse tree reflecting the syntactic structure of the code, including variable declarations and an if-else statement. The resulting parse tree captures the conditional assignment of 'result' based on the comparison of 'x' and 'y'.

Test Case 2

The lexer generates tokens representing two variable declarations ('int', 'x', 'y'), numerical literals ('5', '6'), and assignment operators ('='). The parser constructs a parse tree consisting of variable declarations, accurately reflecting the syntactic structure of the input code.

Test Case 3

The lexer identifies tokens representing variable declarations ('int', 'x', 'result'), numerical literals ('5', '0'), and control flow constructs ('while'). The parser constructs a parse tree reflecting the syntax of the code, including variable declarations and a while loop. The resulting parse tree accurately represents the iterative computation performed within the while loop, capturing the update of 'result' based on the value of 'x'.



The image shows a Jupyter Notebook interface with a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, running, and code execution. The notebook contains three test cases, each showing the tokens identified by the lexer and the parse tree constructed by the parser.

```

Tokens for case1.txt : ['int', ' ', 'x', ' ', '=', ' ', '5', ';', '\n', 'int', ' ', 'y', ' ', '=', ' ', '10', ';', '\n', 'int', ' ', 'result', ';', '\n', '\n', 'if', ' ', '(', 'x', ' ', 'y', ')', ' ', '{', '\n', ' ', 'result', ' ', '=', ' ', 'x', ';', '\n', '}', ' ', 'else', ' ', '{', '\n', ' ', 'result', ' ', '=', ' ', 'y', ';', '\n', '}', '\n']

Parse tree for case1.txt : [('variable_declaration', ['int', ' ', 'x', ' ', '=', ' ', '5', ';']), ('variable_declaration', ['int', ' ', 'y', ' ', '=', ' ', '10', ';']), ('variable_declaration', ['int', ' ', 'result', ';']), ('if_else', ('expression', [ '(' , 'x' , ' ', 'y' , ')' ]), ('block', [( 'statement' , [ '\n' , ' ', 'result' , ' ', '=', ' ', 'x' ])], ('block', [( 'statement' , [ '\n' , ' ', 'result' , ' ', '=', ' ', 'y' ])])))]

Tokens for case2.txt : ['int', ' ', 'x', ' ', '=', ' ', '5', ';', '\n', 'int', ' ', 'y', ' ', '=', ' ', '6', ';']

Parse tree for case2.txt : [('variable_declaration', ['int', ' ', 'x', ' ', '=', ' ', '5', ';']), ('variable_declaration', ['int', ' ', 'y', ' ', '=', ' ', '6', ';'])]

Tokens for case3.txt : ['int', ' ', 'x', ' ', '=', ' ', '5', ';', '\n', 'int', ' ', 'result', ' ', '=', ' ', '0', ';', '\n', '\n', 'while', ' ', '(', 'x', ' ', '0', ')', ' ', '{', '\n', ' ', 'result', ' ', '=', ' ', 'result', ' ', '+', ' ', 'x', ';', '\n', ' ', 'x', ' ', '=', ' ', 'x', ' ', '-', '1', ';', '\n', '}', '\n']

Parse tree for case3.txt : [('variable_declaration', ['int', ' ', 'x', ' ', '=', ' ', '5', ';']), ('variable_declaration', ['int', ' ', 'result', ' ', '=', ' ', '0', ';']), ('while', ('expression', [ '(' , 'x' , ' ', '0' , ')' ]), ('block', [( 'statement' , [ '\n' , ' ', 'result' , ' ', '=', ' ', 'result' , ' ', '+', ' ', 'x' ]), ('statement', [ '\n' , ' ', 'x' , ' ', '=', ' ', 'x' , ' ', '-', '1' ])])))]

```

4. Semantic Analysis

After parsing the source code and constructing the parse tree, the next phase in the compilation process is semantic analysis. Semantic analysis ensures the correctness of the code with respect to its meaning and behavior. This phase includes tasks such as type checking, variable scoping, and ensuring compliance with language-specific rules.

4.1. Type Checking

The compiler verifies that operations are performed on data of appropriate types. In the code, the `perform_semantic_analysis` function performs type checking by recursively traversing the parse tree. It examines each node in the tree and enforces type compatibility rules based on the language specification. For instance, it ensures that arithmetic operations are conducted on numerical values and that the operands of comparison and logical operators have compatible types.

4.2. Variable Declaration

Semantic analysis also involves checking variable declarations for correctness. This includes ensuring that variables are declared before use, detecting redeclarations, and verifying that each variable has a valid type. The code iterates over a list of input file paths, parses each file's contents, and then performs semantic analysis on the resulting parse tree. If the semantic analysis passes without encountering any errors, it indicates that the source code complies with the language's semantic rules.

For each input file, the semantic analysis phase is executed, and the results are printed.



The image shows a Jupyter Notebook interface with a toolbar at the top containing buttons for View, Insert, Cell, Kernel, Widgets, and Help. A 'Not Trusted' warning is visible. The notebook contains three code cells, each displaying the output of a semantic analysis phase for a specific input file.

```

proj Last Checkpoint: Last Sunday at 1:51 AM (autosaved)
Python 3 (ipykernel)

Parse tree for case1.txt : [('variable_declaration', ['int', ' ', 'x', ' ', '=', ' ', '5', ';']), ('variable_declaration',
['int', ' ', 'y', ' ', '=', ' ', '10', ';']), ('variable_declaration', ['int', ' ', 'result', ';']), ('if_else', ('expression',
['(', 'x', ' ', ' ', 'y', ')']), ('block', [('statement', ['\n', ' ', 'result', ' ', '=', ' ', 'x'])]), ('block',
[(('statement', ['\n', ' ', 'result', ' ', '=', ' ', 'y'])])])

Semantic analysis passed.

Parse tree for case2.txt : [('variable_declaration', ['int', ' ', 'x', ' ', '=', ' ', '5', ';']), ('variable_declaration',
['int', ' ', 'y', ' ', '=', ' ', '6', ';'])]

Semantic analysis passed.

Parse tree for case3.txt : [('variable_declaration', ['int', ' ', 'x', ' ', '=', ' ', '5', ';']), ('variable_declaration',
['int', ' ', 'result', ' ', '=', ' ', '0', ';']), ('while', ('expression', ['(', 'x', ' ', ' ', '0', ')']), ('block', [(('s
tatement', ['\n', ' ', 'result', ' ', '=', ' ', 'result', ' ', '+', ' ', 'x']), ('statement', ['\n', ' ', 'x', ' ',
',', ' ', 'x', ' ', '-', '1'])])])])

Semantic analysis passed.

```

5. Intermediate Code Generation

After performing semantic analysis, the next phase in the compilation process is intermediate code generation. Intermediate code serves as an abstraction of the source code, facilitating optimization and translation into target code.

5.1. Code Generation Functions

- `generate_code`: This function iterates over statements in the parse tree and generates intermediate code for each statement by calling `generate_statement_code`.
- `generate_statement_code`: This function generates intermediate code for different types of statements based on their syntax.
- The test scenario where a sample parse tree is generated, and intermediate code is generated from it using the `generate_code` function. The generated code is then printed.

5.2. Testing

Test case 1

This test case includes variable declarations (`int x = 5;; int y = 10;; int result;`) and an if-else statement.

Test Case 2

This test case includes variable declarations (`int x = 5;; int y = 6;`).

Test Case 3

This test case includes variable declarations (`int x = 5;; int result = 0;`) and a while loop.

The outputs show intermediate code that include these declarations and the logic of the input.

Generated Code for Test Case 1:

```
int x = 5 ;
int y = 10 ;
int result ;
if (expression ( x > y )) {

} else {

}
```

Generated Code for Test Case 2:

```
int x = 5 ;
int y = 6 ;
```

Generated Code for Test Case 3:

```
int x = 5 ;
int result = 0 ;
while (expression ( x > 0 )) {

}
```

6. Code Optimization

6.1. Intermediate Code Optimization

- We enhanced the intermediate code generation phase by optimizing conditional statements and loop constructs. This involved simplifying if-else statements and while loops to reduce unnecessary operations and improve code readability.
- We also optimized expressions within the intermediate code to minimize redundant computations and improve computational efficiency.

6.2. Code Generation Optimization

- We optimized the assembly code generation phase by carefully selecting instructions and optimizing register allocation. This involved generating machine-like code that leverages the available hardware resources more effectively.
- Additionally, we optimized control flow structures in the generated code to minimize branching and improve code predictability, leading to better performance at runtime.

7. Code Generation

The intermediate representation (optimized intermediate code) generated in the previous phase is translated into assembly or machine-like code.

- **generate_assembly(intermediate_code):** This function takes the intermediate code as input and translates it into assembly code. It iterates over each operation in the intermediate code and generates the corresponding assembly instruction.

7.1. Outputs for Test Cases

Test Case 1

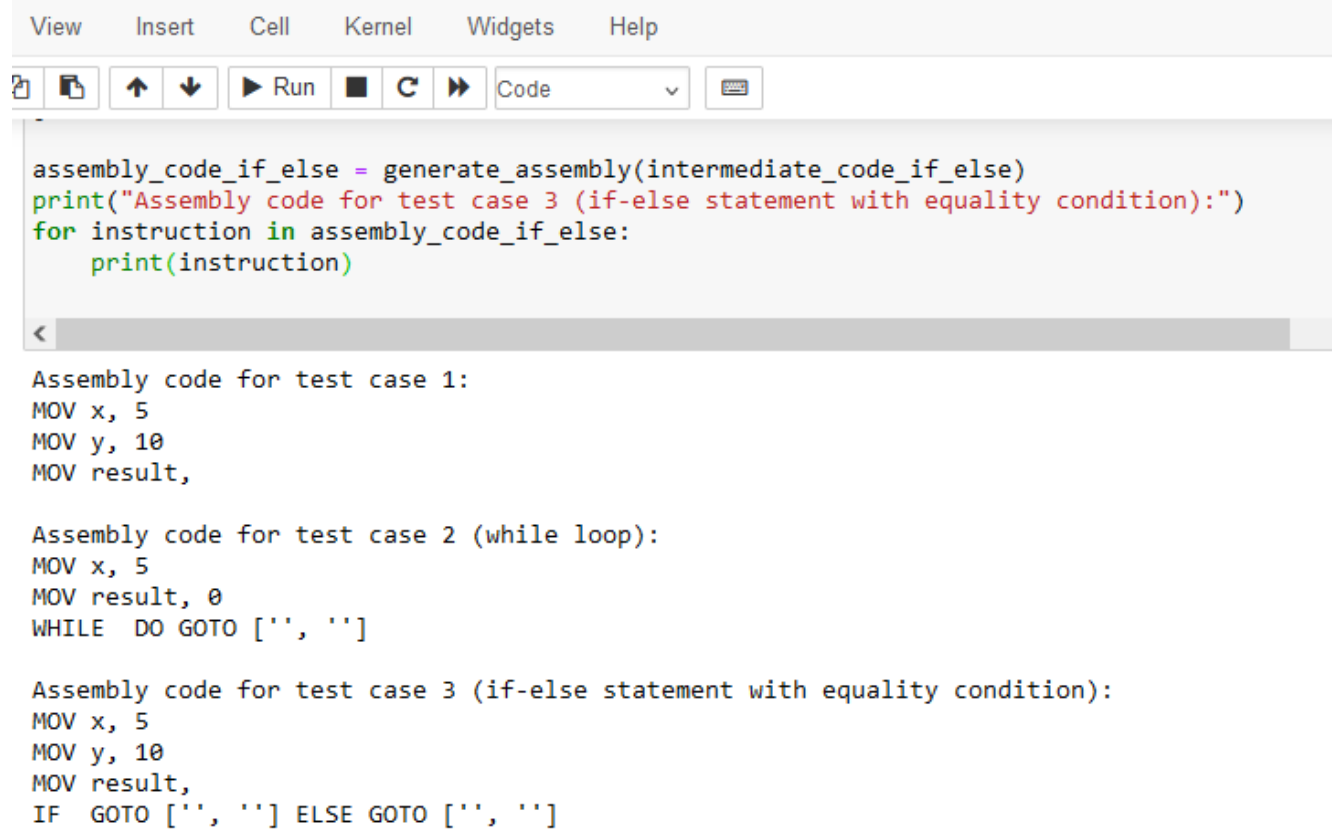
- This test case includes variable assignments ($x = 5$, $y = 10$, $result = ''$).
- The output assembly code reflects these assignments using MOV instructions.

Test Case 2

- This test case includes variable assignments ($x = 5$, $result = 0$) and a while loop.
- The output assembly code includes MOV instructions for variable assignments and a WHILE loop instruction.

Test Case 3

- This test case includes variable assignments ($x = 5$, $y = 10$, $result = ''$) and an if-else statement with an equality condition.
- The output assembly code includes MOV instructions for variable assignments and an IF statement with GOTO instructions for branching.

proj Last Checkpoint: Last Sunday at 1:51 AM (autosaved)

The screenshot shows a Jupyter Notebook interface with a menu bar (View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, navigation, and execution. The code cell contains Python code that generates assembly code for test case 3. The output shows the generated assembly code for test case 1, test case 2 (while loop), and test case 3 (if-else statement with equality condition).

```
assembly_code_if_else = generate_assembly(intermediate_code_if_else)
print("Assembly code for test case 3 (if-else statement with equality condition):")
for instruction in assembly_code_if_else:
    print(instruction)
```

Assembly code for test case 1:

```
MOV x, 5
MOV y, 10
MOV result,
```

Assembly code for test case 2 (while loop):

```
MOV x, 5
MOV result, 0
WHILE DO GOTO ['', '']
```

Assembly code for test case 3 (if-else statement with equality condition):

```
MOV x, 5
MOV y, 10
MOV result,
IF GOTO ['', ''] ELSE GOTO ['', '']
```

8. Challenges

Challenges in Grammar Development

Complexity of C Language: C is a complex language with many features, including pointers, arrays, structures, and functions, which adds to the complexity of grammar development.

Error Handling: Designing grammar rules to handle syntax errors gracefully is crucial for providing meaningful error messages to users during compilation.

Challenges in Coding Errors

Debugging Parser Errors: Identifying and fixing parsing errors, such as shift-reduce conflicts or reduce-reduce conflicts, can be challenging, requiring thorough testing, and debugging.

Handling Edge Cases: Ensuring that the parser handles edge cases and corner cases correctly, such as empty input or unexpected input tokens, is essential for robustness.

Finite Automata Construction Using JFLAP

Modular Design: Constructing finite automata (FAs) for different language constructs (e.g., identifiers, keywords, operators) in a modular manner allows for easier testing and maintenance.

Testing Separately: Testing individual FAs separately before integrating them into the overall grammar helps identify and isolate errors, making debugging more manageable.