Group ID: 26

CSN-352 Project-2 Final Report

Himani Panwar H_PANWAR@CS.IITR.AC.IN
Department of Computer Science and Engineering,
IIT Roorkee,
21114041

Sahil Safy S_SAFY@CS.IITR.AC.IN
Department of Computer Science and Engineering,
IIT Roorkee,
21114087

Sanidhya Bhatia S_BHATIA@CS.IITR.AC.IN
Department of Computer Science and Engineering,
IIT Roorkee,
21114090

1 Problem Statement

The objective of this project is to develop a compiler tailored for the LOGO programming language, facilitating the creation of 3D figures using OpenGL. Comprising distinct phases—tokenization, parsing, and semantic analysis—the compiler empowers users to craft complex 3D structures through LOGO programming.
By using OpenGL's rendering capabilities, the compiler transforms LOGO code into visually compelling three-dimensional renderings. The tokenization phase involves breaking down LOGO code into meaningful components, while parsing analyzes the structure of the code to ensure syntactic correctness. Semantic analysis verifies the logic of the program, ensuring that it adheres to the rules and conventions of the LOGO language.

2 Introduction

Logo is an educational programming language, designed in 1967 by Wally Feurzeig, Seymour Papert, and Cynthia Solomon. It is a general purpose programming language known for its use of turtle graphics, in which commands for movement and drawing produced line or vector graphics, either on screen or with a small robot termed as turtle. The language was conceived to teach concepts of programming related to Lisp and foster logical thinking in learners.

There are certain predefined commands used to produce the expected output in LOGO, the commands used in the project and their output have been mentioned in table 1.

Command	Abbreviation	Output
Forward	FD X	Move Turtle forward by X units
Backward	BK X	Move Turtle Backward by X units
Right	RT X	Turn Turtle right by X degrees
Left	LT X	Turn Turtle left by X degrees
Upper	UT D	Turn Turtle up by X degrees
Down	DT X	Turn Turtle Down by X degrees
Pen up	PENUP X	Lift pen
Pen down	PENDOWN X	Drop pen
Home	HOME X	Return turtle to (0,0,0)
Loop	REPEAT N[ ]	Repeat set of commands N times

Table 1.

Note : UT and DT commands in above table are not present in official documents of LOGO, as it is implemented in 2D Plane while this project is implemented in a 3D Space.

3 Libraries used

- **OpenGL :** OpenGL (Open Graphics Library) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D graphics. It also provides a simple platform-independent interface for creating graphical user interfaces. In this project, it has been used for creating windows, setting up the view matrix, handling input events, and managing the rendering context.

- **Pygame :** Pygame is a set of Python modules designed for writing video games. It provides functions and classes for handling input, graphics, sound, and other multimedia functionalities. In this project, it has been used for drawing graphics on the screen.

- **sys :** The sys module provides access to some variables used or maintained by the Python interpreter and to functions that interact strongly with the interpreter. In this project, it has been used for accessing command-line arguments.

- **copy :** The copy library provides features for creating shallow and deep copies of objects from any type of class. It was used in parsing when printing the Stack elements after each iteration of the parsing process.

4 Implementation

4.1 Tokenizer

The tokenizer serves as the initial stage in the compilation process, responsible for breaking down the input LOGO file into individual tokens, each representing a specific lexical unit. Implemented in Python, our tokenizer scans the input file, identifying and extracting tokens based on predefined patterns and rules. Upon execution, the tokenizer seamlessly processes the input, generating a structured output comprising tokens along with their corresponding types. Longest prefix matching has been used for Lexical Analysis. There are two classes that are used in tokenizing, the Token Class is type class for tokens. Every Token that belongs to this class has a type and a value. The Lexer class performs lexical analysis on an input text and converts it into a sequence of valid tokens. Entire text is converted to lowercase for uniformity, and the Lexer also identifies invalid texts.
The information about valid tokens and their types is mentioned in Table 2.

Pattern	**Token type**
fd	FORWARD
bk	BACKWARD
rt	RIGHT_ROTATION
lt	LEFT_ROTATION
ut	UP_ROTATION
dt	DOWN_ROTATION
penup	PEN_UP
pendown	PEN_DOWN
home	HOME
repeat	REPEAT
`[`	LEFT_SQUARE_BRACKET
`]`	RIGHT_SQUARE_BRACKET
$[0\text{-}9]^+$ or $[0\text{-}9]^+$ $[.][0\text{-}9]^+$	FLOATING_POINT

Table 2.

4.2 Parser

Before implementation of the parser, a grammar is defined which is like a guidebook for using the LOGO programming language. The grammar is given below.

0. S → repeat K [S] S
1. S → fd K S
2. S → rt K S
3. S → lt K S
4. S → ut K S
5. S → dt K S
6. S → bk K S
7. S → home S
8. S → penup S
9. S → pendown S
10. S → #
11. K → id

NOTE: Here commands have been written in lower case to differentiate it from non-terminals, these are usually used in upper case. Also, # represents Epsilon and id represents non-negative integers and floating point numbers.
LL(1) parsing is a top-down parsing method in the syntax analysis phase of compiler design. Required components for LL(1) parsing are input string, a stack, parsing table for given grammar, and parser.
The parser is implemented in python referring to the above grammar.Tokens which were generated using tokenization are given as input to the parser, which are then parsed using the LL(1) Parsing technique. For the purpose of LL(1) parsing, parsing table is generated manually which is given in Table 3.

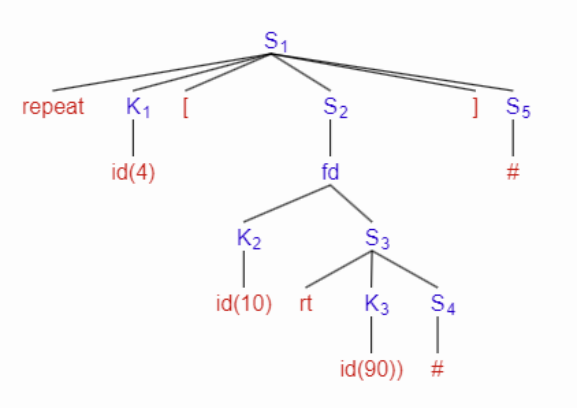
	repeat	fd	bk	rt	ut	lt	dt	home	penup	pendown	id	[	]	$
S	0	1	6	2	4	3	5	7	8	9	-	-	10	10
K	-	-	-	-	-	-	-	-	-	-	11	-	-	-

Table 3.

Parsing Example for the following Test Case in Table 4:
Repeat 4 [
fd 10
rt 90
]

Stack	Input	Action
$S	repeat 4[fd 10 rt 90] $	Push (rule 0)
$S] S [K repeat	repeat 4[fd 10 rt 90] $	Pop
$S] S [K	4[fd 10 rt 90] $	Push(rule 11)
$S] S [id	4[fd 10 rt 90] $	Pop
$S] S [	[fd 10 rt 90] $	Pop
$S] S	fd 10 rt 90] $	Push(rule 1)
$S] S K fd	fd 10 rt 90] $	Pop
$S] S K	10 rt 90] $	Push(rule 11)
$S] S id	10 rt 90] $	Pop
$S] S	rt 90] $	Push(rule 2)
$S] S K rt	rt 90] $	Pop
$S] S K	90] $	Push(rule 11)
$S] S id	90] $	Pop
$S] S	] $	Push(rule 10)
$S]	] $	Pop
$S	$	Parsing Successful!

Table 4: Parsing Example for a simple testcase.



Parse Tree for a simple testcase.

4.3 Semantic Analysis

The generate_moves and generate functions, defined in the python file named Generator, perform semantic analysis by interpreting Logo commands and generating geometric sequences based on the semantic meaning of each command.
It invovles the use of a Head class for geometric Operations. The Head class represents the virtual pen (or a turtle) and implements geometric operations such as forward and backward movement, rotations in different directions, and pen control (up or down).

The following Functions in the Head Class define the motion of the pointer:

```
def home(self):
    self.x = 0
    self.y = 0
    self.z = 0
    self.horizontal_angle = 0
    self.vertical_angle = 0
    self.pendown = True

def move(self, d):
    r1 = math.radians(self.horizontal_angle)
    r2 = math.radians(self.vertical_angle)
    self.x += d * (math.cos(r2) * math.cos(r1))
    self.y += d * (math.cos(r2) * math.sin(r1))
    self.z += d * (math.sin(r2))

def horizontal_turn(self, r):
    self.horizontal_angle += r
    self.horizontal_angle %= 360

def vertical_turn(self, r):
    self.vertical_angle += r
    self.vertical_angle %= 360
```

home(self): This method sets the initial state of the head. It initializes the coordinates (x, y, z) to 0 and the horizontal and vertical angles to 0. It also sets pendown to True, which might indicate that something is ready to be drawn or recorded.

move(self, d): This method moves the head in the direction it's facing by a distance d. It calculates the new coordinates based on the current angles using trigonometric functions (cos and sin). The horizontal_angle affects the movement in the x and y directions, while the vertical_angle affects the z direction.

horizontal_turn(self, r): This method rotates the head horizontally by an angle r. It updates the horizontal_angle accordingly and ensures that the angle stays within the range of 0 to 360 degrees using the modulo operator.

vertical_turn(self, r): Similarly, this method rotates the head vertically by an angle r. It updates the vertical_angle and ensures it stays within the range of 0 to 360 degrees.

The processing of Turtle or Head moves takes place by storing all the positions taken by the head in a list and then Passing that list to the Pygame library object for creating visuals using the coordinates. Only the moves FORWARD and BACKWARD are responsible for the movement of the object. The other moves UP_ROTATION, DOWN_ROTATION, LEFT_ROTATION and RIGHT_ROTATION are responsible for turning the Turtle in any Direction in the 3D-plane while the moves PENUP, PENDOWN and HOME are moves that

control the Drawing part.
Recursion is used to handle the special case of REPEAT. The approach is to create a separate lexer for the statements inside the REPEAT block, and pass this to the generator with same list of positions, and do this the number of times the REPEAT is required to be executed. A counter is maintained to count the difference of number of Opening and Closing Square Brackets, the block is complete when the counter reaches zero.

Semantic Rules for the Compiler using these functions are:
Here "head" is the Instance of class Head and "id" is a non-negative floating point number or integer.

0. S → repeat K [S_1] S_2 { Repeat S_1 K.val times }
1. S → fd K S { head.move(K.val) }
2. S → rt K S { head.horizontal_turn(K.val) }
3. S → lt K S { head.horizontal_turn(- K.val) }
4. S → ut K S { head.vertical_turn(K.val) }
5. S → dt K S { head.vertical_turn(- K.val) }
6. S → bk K S { head.move(- K.val) }
7. S → home S { head.x, head.y, head.z, head.horizontal_angle, head.vertical_angle = {0, 0, 0, 0, 0}}
8. S → penup S { head.pendown = False }
9. S → pendown S { head.pendown = True }
10. S → # {}
11. K → id { K.val = id.val }

4.4 Execution

The program accepts command-line arguments specifying the source file containing Logo commands and the optional parameter rotation settings. The content of the source file is tokenized, parsed, and generated into geometric sequences using the tokeniser, parser, and generator modules. Then, the program utilizes Pygame and OpenGL libraries to render 3D graphics based on these sequences.
Pygame is initialized and a display window is set up with specified dimensions and flags for double buffering and OpenGL rendering. The perspective projection matrix is set to create a perspective view within the display window. Variables for translation (x, y, z) and rotation (rotate) are initialized to control the movement and rotation of objects in the 3D space.
Upon Execution the Program prints a list of Valid Tokens, if an invalid input is found, a Lexical Error is generated and the program ends. After Successful Tokenization, Syntax is checked based on the production rules and the LL(1) Parsing Table. If the Syntax is incorrect, a Syntax Error is generated and the program ends. The program enters a continuous loop to handle user events such as keyboard inputs and window close events.

- Arrow keys adjust translation along the corresponding axis (x, y) by a fixed increment.
- Plus and minus keys adjust translation along the z-axis.

- Spacebar toggles rotation mode, enabling/disabling object rotation based on rotation settings.
- If rotation is enabled, the scene is rotated by the specified angle around the axis defined by (rot_x, rot_y, rot_z).

5 Test Cases

- **REPEAT 36 [**
  FD 0.3
  UT 10
  ]
  REPEAT 36 [
  FD 0.3
  RT 10
  ]
  REPEAT 36 [
  FD 0.3
  DT 10
  ]
  REPEAT 36 [
  FD 0.3
  DT 10
  ]

```
Tokens are as follows:
Token(REPEAT, 'repeat')
Token(FLOATING-POINT, 36.0)
Token(LEFT-SQUARE-BRACKET, '[')
Token(FORWARD, 'fd')
Token(FLOATING-POINT, 0.3)
Token(UP-ROTATION, 'ut')
Token(FLOATING-POINT, 10.0)
Token(RIGHT-SQUARE-BRACKET, ']')
Token(REPEAT, 'repeat')
Token(FLOATING-POINT, 36.0)
Token(LEFT-SQUARE-BRACKET, '[')
Token(FORWARD, 'fd')
Token(FLOATING-POINT, 0.3)
Token(RIGHT-ROTATION, 'rt')
Token(FLOATING-POINT, 10.0)
Token(RIGHT-SQUARE-BRACKET, ']')
Token(REPEAT, 'repeat')
Token(FLOATING-POINT, 36.0)
Token(LEFT-SQUARE-BRACKET, '[')
Token(FORWARD, 'fd')
Token(FLOATING-POINT, 0.3)
Token(DOWN-ROTATION, 'dt')
Token(FLOATING-POINT, 10.0)
Token(RIGHT-SQUARE-BRACKET, ']')
Token(REPEAT, 'repeat')
Token(FLOATING-POINT, 36.0)
Token(LEFT-SQUARE-BRACKET, '[')
Token(FORWARD, 'fd')
Token(FLOATING-POINT, 0.3)
Token(DOWN-ROTATION, 'dt')
Token(FLOATING-POINT, 10.0)
Token(RIGHT-SQUARE-BRACKET, ']')
Syntax is correct!
```

Figure 1: Tokenizer output for Test Case-1 Circles.

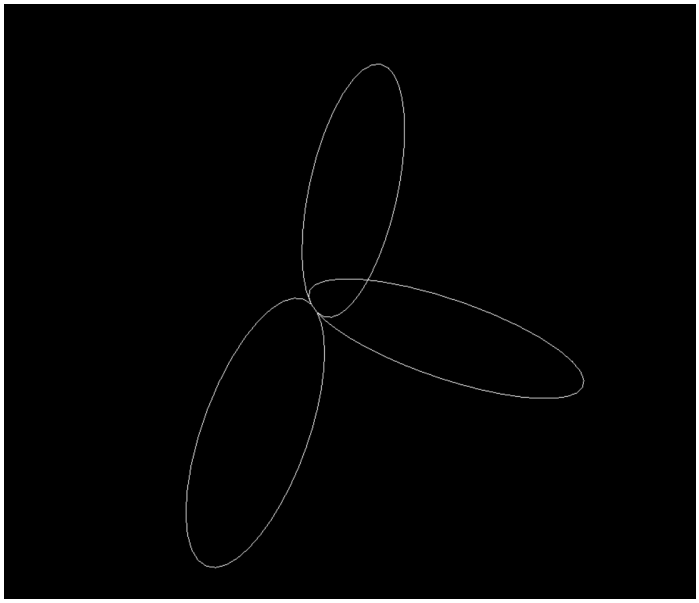


Figure 2: Final output for Test Case-1 Circles.

- **FD 2**
  UT 90
  FD 2
  UT 90
  FD 2
  LT 90
  FD 2
  LT 90
  FD 2
  UT 90
  FD 2
  UT 90
  FD 2
  RT 90
  FD 2

```
Tokens are as follows:
Token(FORWARD, 'fd')
Token(FLOATING-POINT, 2.0)
Token(UP-ROTATION, 'ut')
Token(FLOATING-POINT, 90.0)
Token(FORWARD, 'fd')
Token(FLOATING-POINT, 2.0)
Token(UP-ROTATION, 'ut')
Token(FLOATING-POINT, 90.0)
Token(FORWARD, 'fd')
Token(FLOATING-POINT, 2.0)
Token(LEFT-ROTATION, 'lt')
Token(FLOATING-POINT, 90.0)
Token(FORWARD, 'fd')
Token(FLOATING-POINT, 2.0)
Token(LEFT-ROTATION, 'lt')
Token(FLOATING-POINT, 90.0)
Token(FORWARD, 'fd')
Token(FLOATING-POINT, 2.0)
Token(UP-ROTATION, 'ut')
Token(FLOATING-POINT, 90.0)
Token(FORWARD, 'fd')
Token(FLOATING-POINT, 2.0)
Token(UP-ROTATION, 'ut')
Token(FLOATING-POINT, 90.0)
Token(FORWARD, 'fd')
Token(FLOATING-POINT, 2.0)
Token(RIGHT-ROTATION, 'rt')
Token(FLOATING-POINT, 90.0)
Token(FORWARD, 'fd')
Token(FLOATING-POINT, 2.0)
Syntax is correct!
```

Figure 3: Tokenizer output for Test Case-2 Cube.

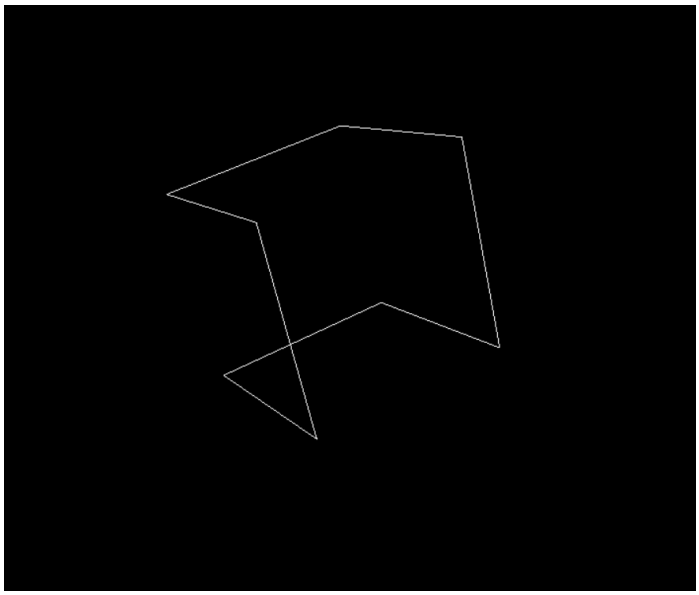


Figure 4: Final output for Test Case-2 Cube.

- **REPEAT 37 [**
  REPEAT 37 [
  FD 0.2
  UT 10
  ]
  RT 10
  ]

```
Tokens are as follows:
Token(REPEAT, 'repeat')
Token(FLOATING-POINT, 37.0)
Token(LEFT-SQUARE-BRACKET, '[')
Token(REPEAT, 'repeat')
Token(FLOATING-POINT, 37.0)
Token(LEFT-SQUARE-BRACKET, '[')
Token(FORWARD, 'fd')
Token(FLOATING-POINT, 0.2)
Token(UP-ROTATION, 'ut')
Token(FLOATING-POINT, 10.0)
Token(RIGHT-SQUARE-BRACKET, ']')
Token(RIGHT-ROTATION, 'rt')
Token(FLOATING-POINT, 10.0)
Token(RIGHT-SQUARE-BRACKET, ']')
Syntax is correct!
```

Figure 5: Tokenizer output for Test Case-3 Loops.

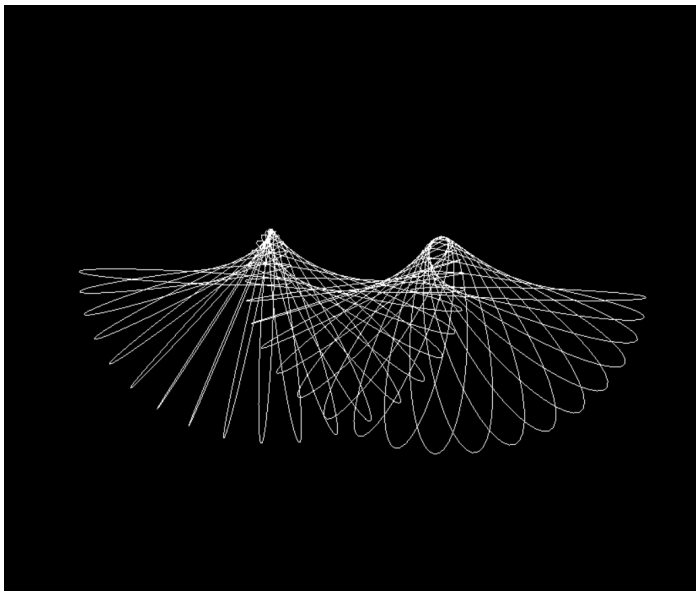


Figure 6: Final output for Test Case-3 Loops.

6 Compiler Design Concepts Involved

This project encompasses the core principles of compiler design, employing a systematic approach to transform LOGO code into manageable units known as tokens through tokenization. Utilizing the Longest Prefix Matching method, the LOGO code is dissected into smaller entities, such as words or symbols, facilitating comprehension and analysis.
Following tokenization, an LL(1) parser is constructed to systematically structure the code, ensuring adherence to grammatical rules and syntax. Moreover, semantic analysis is employed to decipher the intended meaning of commands, facilitating the evaluation of the turtle's motion through specifically defined functions. By comprehensively examining the context and purpose of each instruction, semantic analysis aids in discerning the logical flow of the program.
In essence, this undertaking can be likened to the development of a compiler tailored for the LOGO programming language. Through tokenization, parsing, and semantic analysis, the project endeavors to compile and execute LOGO commands, ultimately producing intricate 3D structures.