SER 502: Project Milestone 1 (Team 22)

Team Number: 22

Team Members:

1. Devanshu Amitkumar Desai
2. Aditya Gupta
3. Nimit Pragnesh Patel
4. Neel Nirajkumar Shah

Language Name: LingoScript

URL to GitHub: https://github.com/DevanshuDesai15/SER502-LingoScript-Team22

Introduction

LingoScript is a simple yet versatile programming language designed to empower developers with an intuitive syntax, rich feature set, and ease of use. It aims to provide a comprehensive solution for various programming needs, from basic arithmetic operations to complex control flow and data manipulation.

Lexical Analysis (Tokenizer)

In this phase, the source code can be processed to generate tokens using Python or ANTLR. Each token represents a specific lexical element like keywords, identifiers, operators, etc. The new line character is used as a separator. This step ensures accurate representation of individual tokens, crucial for further processing.

Parsing (Parser)

The parser can validate the syntactic correctness of the program by utilizing SWI-Prolog or ANTLR and Definite Clause Grammar (DCG). It constructs a parse tree from the tokens generated by the lexer. The parse tree represents the hierarchical structure of the program, ensuring adherence to the language's syntax rules.

Semantic Evaluation/Interpretation (Evaluator)

This phase evaluates the semantics of the program, including operator correctness, expression evaluation, and variable scoping. SWI-Prolog can be utilized for semantic evaluation. The evaluator operates on the parse tree, executing the program's instructions

according to its layout, bridging the gap between human-readable code and machine-executable instructions.

Parsing Technique and Tools

We will employ ANTLR (Another Tool for Language Recognition) or SWI-Prolog for parsing LingoScript code. ANTLR is a powerful parser generator that allows us to define the grammar rules and automatically generate a parser for the language.

Tools To Be Used:

- Python or ANTLR: Used for lexical analysis to generate tokens from the input stream.
- ANTLR or SWI-Prolog: Used for syntax analysis to generate a parse tree from the token stream.
- SWI-Prolog: Used for evaluation of semantics within each line by processing the parse tree.

Data Structures:

- Token: Represents a lexical token generated by the lexer, containing information such as token type and value.
- Abstract Syntax Trees (ASTs): Represents the hierarchical structure of the parsed input, facilitating interpretation and execution.

Language Design

LingoScript supports the following features:

1. **Primitive Types:**
    - Boolean values (**true**, **false**) with support for **and**, **or**, and **not** operators.
    - Numeric types at least one (integers, floats, doubles) supporting basic arithmetic operations: addition, subtraction, multiplication, and division.
    - Strings: Ability to assign string values to variables. Operations on strings are optional and can be added in scope in future.
2. **Variable Assignment:**
    - The language allows associating values with identifiers using an assignment operator (**=**).

3. **Conditional Constructs:**
    - Ternary Operator: Similar to the **? :** operator in Java, allowing for concise conditional expressions.
    - Traditional if-then-else construct for more complex branching logic.
4. **Looping Structures:**
    - Traditional **for** loop.
    - Traditional **while** loop.
    - **for i in range(start, end)**: Equivalent to **for (i=start; i<end; i++)**.
5. **Printing:**
    - A built-in **print** construct to display identifier values, supporting all data types including strings and booleans.

Language Grammar

The grammar will be used to define the syntax for the lexical analyzer and parser. Here's a detailed grammar for the language:

P for Program

M for Main Block

D for Declaration

C for Command

CD for Constant Declaration

VD for Variable Declaration

E for Expression

T for Term

F for Factor

B for Boolean Expression

RE for Relational Expression

T for Ternary Operation

ELIF for Else-if

A for Assignment

N for Number

ID for Identifier

LL for Lower Case Letter

UL for Upper Case Letter

PI for Positive Integer

NI for Negative Integer

INT for Integer

S for String

P ::= main () { M }

M ::= D ; C

D ::= D ; D | CD | VD

CD ::= const int ID = INT | const int ID = E | const bool ID = B | const string

ID = S

VD ::= var int ID = INT | var int ID = E | var bool ID = B | var string ID = S

B ::= true | false | B and B | B or B | not B | RE

E ::= E + T | E - T | T

T ::= T / F | T * F | F

F ::= (E) | ID | INT

RE ::= E > E | E < E | E >= E | E <= E | E == E

T ::= B ? C : C

ID ::= LL ID | UL ID | LL | UL

S ::= ‘LL S’ | ‘UL S’ | ‘N S’ | ‘N’ | ‘LL’ | ‘UL’ | ‘’

PI ::= N N | N

NI ::= - PI

INT ::= PI | NI

C ::= C C | if (B) { C } else { C } | if (B) { C } ELIF | if (B) { C } ELIF else { C } |

for (D ; B ; A) { C } | for (A ; B ; A) { C } | for ID in range (INT, INT) { C } |

while (B) { C } | A; | T; | print (E); | print (S); | print (B); | print (INT);

ELIF ::= else if (B) { C } ELIF | else if (B) { C }

A ::= ID = E

N ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

LL ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

UL ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z