SER 502 - Milestone 1 (Project Description)

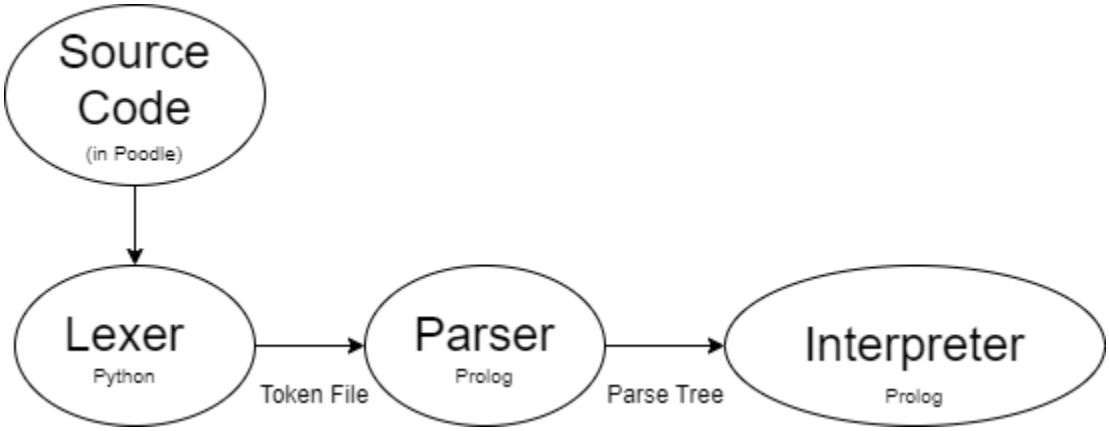
Team 23

Members:
Shloka Manish Pandya (1231997574)
Vipsa Kamani (1231818590)
Malavika Anand (1231722364)

SECTION 1: Programming Language Description

Language Name: Poodle
Programming Paradigm: Imperative
Extension: .poo
Github Link: https://github.com/Malavika-Anand/SER502-Poodle-Team23

SECTION 2: Process Flow



Lexer: The lexer, also known as the lexical analyzer or scanner, is responsible for breaking down the source code into a sequence of tokens. In our case, we will write the code in “Poodle” and then provide the filename to the lexer as input, which will read the source code and tokenize it. The output will be a token file.
Parser: The parser will read the token file generated by the lexer and construct a parse tree for it using the grammar rules defined for the language.

Interpreter: The interpreter is a component that executes the instructions represented by the parse tree. This interpreter will receive a parse tree generated by the parser as input. Thereafter, it will traverse through each node of the tree and generate the semantics of it by executing the instructions in each node.

SECTION 3: Programming Languages used:

Lexer : PLY (Python Lex-Yacc)
Parser : Prolog
Interpreter : Prolog

SECTION 4: Implementation of compiler and parse tree interpreter

4.1 Parsing technique

The parser will be using the top-down approach for parsing. This approach ensures that the syntax is validated from the highest-level rule down to the lowest. We plan to parse the token file via writing DCG(Definite Clause Grammar) to generate a parse tree using PROLOG.

4.2 Data Structure used by parser or interpreter

Both parsers and interpreters will be utilizing lists as a fundamental data structure. Lists provide flexibility and efficiency in storing and manipulating parsed or interpreted data. The Python lexer utilizes a list data structure to store tokens, which are then saved into a file. This file is subsequently read by the Prolog parser and transformed into a list format. The parser proceeds to generate a parse tree represented as a list, which serves as the output. This output is fed into the evaluator component and executed directly on the machine. Additionally, all variables involved in the program execution are maintained as a list within the Prolog environment.

SECTION 5: Design

Syntax and Grammar

Our programming language aims to create an imperative language which is a mixture of C and Python. This includes the curly braces syntax, other control statements, operational and data syntaxes of C. Whereas taking the concept of lists from Python. The following subsections outline the various levels of grammar details.

1.Program Structure

A program in our language consists of a series of commands or blocks, organized hierarchically.

2. Command List

A command list is a sequence of one or more commands or blocks, executed in the order they appear.

3. Block

In Poodle, block is a collection of commands enclosed within braces { }. We have borrowed this concept from the C language, as indentation in the Python language can sometimes be prone to errors and also Deeply nested indentations can lead to excessive complexity.

In python Language :

```
if condition1:
   if condition2:
      if condition3:
         # Code block
```

Instead, using curly braces "{ }" in languages like C allows for more complex structures to be implemented, and code can be written in a single line:

In C Language :

```
if (condition1 && condition2 && condition3) {// Code block}
```

4.Commands

Our language supports various types of commands, including:

4.1 Loop Statements

4.1.1 For Loop Statement

```
for(int i=2; i<5; i++) {
        //code blocks
}
```

4.1.2 While Loop Statement

```
while(true) {
        //command
}
```

4.1.3 Range Loop Statement

```
for i in range(2,5) {
        //code block
}
```

4.2 Conditional Statement

4.2.1 If Statement

```
if(i==10) { //code block }
```

4.2.2 If-Else Statement

```
if(i==10){
//code block
} else {
//code block
}
```

4.2.3 else if Statement

```
if(i==10){ //code block}
elseif(i==3){//code block}
else { //code block }
```

5. Data Types

5.1 Integer

```
int x;
x = 2;
```

5.2 String

```
x = "Hello";
```

5.3 Boolean

```
bool x;
x=true;
```

5.4 Decimal

```
float x;
x = 2.222222
```

6. Operators

6.1 Arithmetic Operators:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)

```
// Addition
int sum = 5 + 3; // sum is assigned the value 8

// Subtraction
```

```
int difference = 10 - 4; // difference is assigned the value 6

// Multiplication
int product = 3 * 6; // product is assigned the value 18

// Division
int quotient = 20 / 5; // quotient is assigned the value 4
```

6.2 Relational Operators:

- Equal to (==)
- Not equal to (!!=)
- Greater than (>)
- Less than (<)
- Greater than or equal to (>=)
- Less than or equal to (<=)

```
// Equal to
int a = 5, b = 5;
if (a == b) {
   // Perform action if a is equal to b
}

// Not equal to
if (a !!= b) {
   // Perform action if a is not equal to b
}

// Greater than
if (a > b) {
   // Perform action if a is greater than b
}

// Less than
if (a < b) {
   // Perform action if a is less than b
}

// Greater than or equal to
if (a >= b) {
   // Perform action if a is greater than or equal to b
}
```

```
// Less than or equal to
if (a <= b) {
   // Perform action if a is less than or equal to b
}
```

6.3 Logical Operators:

- Logical AND (&&)
- Logical OR (||)
- Logical NOT (!!)

```
// Logical AND
if (a > 0 && b < 10) {
   // Perform action if both conditions are true
}

// Logical OR
if (a == 0 || b == 0) {
   // Perform action if either condition is true
}

// Logical NOT
if (!!(a == 0)) {
   // Perform action if the condition is not true
}
```

6.4 Assignment Operators:

Assignment (=)

```
a=b=c=10;
```

6.5 Conditional operator (?:)

```
int result = (a > b) ? a : b;
```

Here, the result is assigned the value of a if a is greater than b, otherwise it is assigned the value of b.

7. Variable Declaration

```
int x;
float y;
```

8. Variable Initialization

```
x = 2;
y = 2.2222;
```

9. Guidelines for Variable Naming Conventions

9.1 Variables can contain upper case letters and lower case letters and digits.

```
mySampleVariable (Correct)
MySampleVariable (Correct)
MYSAMPLEVARIABLE (Correct)
My Sample        (Incorrect)
```

9.2 Variable names should not start or end with a digit or underscore.

```
9Sample (Incorrect)
Sample9 (Incorrect)
_sample_variable  (Incorrect)
Sample_variable          (Correct)
```

9.3 No special characters are allowed.

```
%SampleVariable  (Incorrect)
SampleVariable*  (Incorrect)
```

10. Print Statement

```
print>>'This is yellow world';
print>>"This double is also allowed";
print>>"{Variable_name} is my name.";
```

11. Reserved Words

print>> - To print the statement
&& - Logical AND
|| - Logical OR
!! - Logical NOT
true - Boolean True value
false - Boolean False value
int - Integer data type
float - Floating-point data type
bool - Boolean data type
string - String data type
if - Conditional statement
else - Alternative condition
elseif - Additional Condition
for - Loop iteration
while - Conditional loop

```
function - Function without name0
return - Returns value from a function
```

12. Other Enhanced Features

12.1 Variable declaration and initialization:

```
int num = 5;
int a=b=c=1;
```

12.2 Syntactic Sugar:

Increment (++)
Decrement (--)

```
// Increment
int num = 5;
num++; //num = num+1
++num;

// Decrement
num--; // num = num-1
--num; //
```

12.3 List Data structure

```
List = [1,2,4,a,"ef"];
```

SECTION 6: Contribution

Milestone 1 (Initial Phase Submission)

- GitHub Repository created with project structure.
- Project Description Document: Written by all the team members through various in-person brainstorming meetings for language design.

Milestone 2 (Final Phase Submission)

- Malavika: Lexer
- Vipsa: Parser
- Shloka: Interpreter
- All members: Test scripts

Link to Contribution.txt file :
https://github.com/Malavika-Anand/SER502-Poodle-Team23/blob/main/doc/Contribution.txt

SECTION 7: Grammar

```
%PROGRAM
Program ::= Command_List
Block ::= '{' Command_List '}'

%COMMAND LIST AND COMMANDS
Command_List ::= Command; Command_List
Command ::= Declration_Assignment_Command
            | Print_Command
            | If_Command
            | If_Else_Command
            | If_Elseif_Else_Command
            | Forloop_Command
            | Compact_Forloop_Command
            | Whileloop_Command
            | List_Command


%PRINT COMMAND
Print_Command ::= print >> Expression;

%IF COMMAND
If_Command ::= if(Condition) Block.

%IF_ELSE COMMAND
If_Else_Command ::= if(Condition) Block else Block.

%IF_ELSEIF_ELSE COMMAND
If_Elseif_Else_Command ::= if(Condition) Block ElseIf_Block else
Block.
ElseIf_Block ::= elseif(Condition) Block ElseIf_Block | Epsilon;

%FOR LOOP COMMAND
Forloop_Command ::= for (Variable_initialization ; Condition ;
increment_expression ) Block
                  | for (Variable_initialization ; Condition ;
decrement_expression ) Block
```

```
                | for (Variable_initialization ; Condition ;
                Variable_Name = Expression ) Block

%WHILE LOOP COMMAND
Whileloop_Command ::= while (Condition) Block.


%COMPACT FOR LOOP COMMAND

Compact_Forloop_Command ::=
                for Variable_Name in range(integer, integer) block
                | for Variable_Name in range(Variable_Name,
            Variable_Name) block.


%LIST COMMAND
List_Command ::= Variable_Name = [ Values ];
Values ::= Value , Values | Value

%ALL DECLARATION COMMAND
Declration_Assignment_Command ::= Data_Type Variable_Name;
                              | Variable_Name = Value;
                              | Data_Type Variable_Name = Value
                              | Variable_Name = Function_Command;

%ALL EXPRESSIONS
Expression ::= Expression Arithmetic_operator Expression
            | Expression boolean_operator Expression
            | Value
            | Ternary_Expression
            | ( Expression )

%TERNARY COMMAND
Ternary_command ::= (Condition) ? Expression : Expression.

%CONDITION
Condition ::= Expression Comparison_operator Expression.

%VALUES FOR RHS OF ASSIGNMENT OPERATOR
Value ::= float | integer | boolean_value | String | Variable_Name.

%VARIABLE NAME RULES
Variable_Name ::=  Alphabet_Lower_Case Variable_Name
                | Alphabet_Upper_Case Variable_Name
```

```
                | Variable_Name Integer Variable_Name
                | Variable_Name _ Variable_Name
                | Alphabet_Lower_Case
                | Alphabet_Upper_Case


%RULES FOR SYNTACTIC SUGAR
Increment_expression ::= Variable_Name ++ | ++ Variable_Name;
Decrement _expression ::= Variable_Name -- | -- Variable_Name;

%DATA TYPES RULES
Integer ::= Digit Integer| Digit
Float ::= Integer . Integer | Integer
String ::= `Char_List ` | "Char_List".
Char_List ::=  Char String  |  Char
Char ::= Alphabet_Lower_Case | Alphabet_Upper_Case | Symbol | Digit.

Digit ::= `0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

Alphabet_Lower_Case ::= `a'|'b'|'c'|....|'z'
Alphabet_Upper_Case ::= `A'|'B'|'C'|....|'Z'

Symbol ::=  ['{'] | ['|'] | ['}'] | ['!'] | ['\"'] | ['#'] | ['$'] |
['%'] | ['&'] | ['@'] | ['('] | [')'] | ['*'] | ['+'] | [',] | ['-']
| ['.'] | ['/'] | [':'] | [';'] | ['<'] | ['='] | ['>'] | ['?'] |
['['] | ['>>'] | [']'] | ['^'] | ['_'] | [' ']'

Data_Type ::= `int' | `bool' | `string' | `float'
Boolean_value ::= true | false
Boolean_operator ::= `&&' | `||' | `!!'.
Comparison_operator ::= `>' | `<' | `==' | `!!=' | `>=' | `<=.
Arithmetic_operator ::= `+' | `-' | `/' | `*'
```