SER 502 Spring 2023 - Group 35 - Milestone 1

Team members:
Akshat Nambiar (ASU ID: 1225484000)
Manideep Nalluri (ASU ID: 1225915641)
Ryan Collins (ASU ID: 1225687957)
Sai Prakash Ravichandran (ASU ID: 1225761147)
Sai Viswas Nirukonda (ASU ID: 1225421353)

GitHub Link: https://github.com/saiprakashasu/SER502-Spring2023-Team35

Name of the language: Spectra

Design of the language:
The code written in Spectra will go through a lexical analyzer, a parser and an interpreter to generate the output

A lexical analyzer reads the program's source code and breaks it down into a stream of tokens which are individual units of meaning such as keywords, identifiers, and operators.

The parser then takes this stream of tokens and uses a grammar to determine whether the code is syntactically correct. If it is, the parser builds a data structure called a parse tree that represents the program's structure.

Finally, the interpreter reads the parse tree and executes the program by following the instructions specified by the code. This involves evaluating expressions, performing operations, and making decisions based on conditional statements. The interpreter translates the high-level code into machine code that the computer can understand, allowing the program to be executed.

Parsing technique used:
We have planned to generate the parse tree by parsing the list of tokens using Definite clause grammar (DCG). A list of tokens (words) as input will be provided to the DCG parser, and the parser would generate a parse tree that represents the structure of the program.

Data structure used by parser and interpreter:

We will use an abstract syntax tree data structure for the parser, as it allows parsers to more efficiently and effectively analyze and manipulate the structure of programs and expressions.

We plan to use a stack or a stack-based data structure for the interpreter as it can be used to keep track of the state of the program, including the order of execution, intermediate computations, and variable values.

Data types and operators:
The data types supported by the language: integer, float, string, boolean.

Supported operations on all the data types: Assignment
Supported operations on boolean values: and, or, not
Supported operations on int: Addition, subtraction, multiplication, division

These are the operators supported by the language:
1. Assignment operator(=)
2. Logical operators(<, <=, >, >=, ==, !=)
3. Boolean operators(and, or, not)
4. Arithmetic Operators(+, -, *, /)
5. ternary operator(?:)

Conditional constructs: if-then-else and ternary operator(?:)

Loops:
1. Normal for loop - for (i=2; i<5; i++)
2. while loop - while(b==True)
3. Enhanced for loop - for i in range(2,5)

print Construct
Uses the keyword print. It prints the identifier values of all the data types supported by Spectra (integer, float, string, boolean)

Grammar:

```
/*
Mapping
P --> Program
K --> Block
D --> Declaration
C --> Command Lines (Statements)
DT --> Data Type
Bl --> Boolean Value
B --> Boolean Expression
E --> Expression
T --> Term
F --> Form   (Form is used to assign Identifier or Number to Term)
I --> Identifier
N --> Number
DG --> Digit
S --> Symbols
ID --> increment/Decrement
Data --> Literals
ST --> String
CH --> Characters
*/


P ::=    K
K ::=    start C stop.

D ::=  DT I = data;
       | Dt I;
       | I = I;
       | I = data;

DT ::=   int | float | string | boolean
data:= BI | N | I | ST


C ::=   D, C;
```

```
            | I = E; C
            |print_statement; C
            |If_then_else; C
            |while_loop; C
            |for_loop; C
            |ternary_expression; C
            | K
            | ε


If_then_else::=  if B { C }
                 | if B { C } else { C }
                 | if B { C } elif { C } Elif else { C }

Elif ::= elif { C } Elif | ε

while_loop::=  while B { C }

for_loop::=  for I in range (N,N){ C }
             | for(I = N; I S N; ID){ C }

ternary_expression::= ( C ) ? E : E

print_statement::= print(E)
                   |print(I)
                   |print(ST)
                   |print(B)

B ::=    Bl | E S E | not B | B and B| B or B

E ::=    T + E | T - E | T
T ::=    ( E ) T
T ::=    F * T | F / T | F
F ::=    I | N
Bl ::=   true | false
N ::=    DG, N | ε
ST ::=  CH, ST | ε
```

```
DG ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
CH ::=  Upper_case
        | Lower_case
        | special_char

Upper_case ::= 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

Lower_case ::= 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

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

S ::=    < | > | >= | <= | == | !=
ID ::=   I++ | I--
```

Choice of tools for Language:

Lexer: We are planning to implement using Python
Parser: We are planning to implement using Python
Interpreter: We are planning to implement using Python

Potential future works:

1) Adding provisions for data structure like lists and arrays. (Have to study and analyze similar implementations in popular grammar).

2) Adding more commands for better functionality.

3) Making the grammar more similar towards front-end languages to keep consistency and allow for seamless programming between back-end and front-end. (To be discussed later with Professor Bansal).

4) Any more optimizations to grammar will be considered over the course of lexical and parser development.