SER 502: LANGUAGES AND PROGRAMMING PARADIGMS

MILESTONE 2

Design and Grammar of the Spectra Programming Language

1. Introduction

The **Spectra programming language** is designed to offer a simple and intuitive way to perform computations and operations based on a color spectrum theme. This language supports various constructs found in typical programming languages, including variable declarations, assignments, conditional statements, loops, and arithmetic expressions. The aim is to provide a learning tool that helps new programmers grasp fundamental programming concepts while enjoying a visually appealing theme.

2. Grammar

The grammar of the Spectra language is defined using Extended Backus-Naur Form (EBNF). The grammar defines a program as a sequence of statements ending with an end-of-file (EOF) marker. This allows for multiple statements to be processed in a single program. Below is the complete grammar specification:

CODE:

```
grammar Spectra;

program
    : statement* EOF;

statement
    : declaration                  # DeclarationStmt
    | assignment                   # AssignmentStmt
    | printStatement               # PrintStmt
    | ifStatement                  # IfStmt
    | whileLoop                    # WhileLoopStmt
    | forLoop                      # ForLoopStmt
    | expression                   # ExpressionStmt
    | breakStatement               # BreakStmt
    | continueStatement            # ContinueStmt
    ;

declaration
    : type IDENTIFIER ('=' expression)? ';'  # VarDeclaration
    ;

assignment
    : IDENTIFIER '=' expression ';'         # VarAssignment
    ;

printStatement
    : SPECTRUM_DISPLAY '(' expression ')' ';'  # Print
    ;

ifStatement
    : TRANSPARENT_IF '(' condition ')' '{' statement* '}'
```

```
    (TRANSLUCENT_ELSEIF '(' condition ')' '{' statement* '}')*
    (OPAQUE_ELSE '{' statement* '}')?       # IfElse
  ;

whileLoop
  : VIOLET_WHILE '(' condition ')' '{' statement* '}'   # WhileStmt
  ;

forLoop
  : BLUE_FOR '(' assignment condition ';' assignment ')' '{' statement* '}'   # ForStmt
  ;

breakStatement
  : BREAK_COLOR ';'                    # Break
  ;

continueStatement
  : CONTINUE_COLOR ';'                   # Continue
  ;

expression
      : expressionPrimary ((SHADE_CHECK expression CONTRAST_DO expression)? ) ; //
Ternary operator

expressionPrimary
  : expressionOr
  ;

expressionOr
  : expressionAnd (MAGENTA_OR expressionAnd)*  // Logical OR
  ;

expressionAnd
  : equalityExpression (CYAN_AND equalityExpression)*  // Logical AND
  ;

equalityExpression
          : relationalExpression ((BRIGHTDARK_EQUAL | BRIGHTDARK_NOTEQUAL)
relationalExpression)*  // Equality
  ;
```

```
relationalExpression
    : additionExpression ((LIGHT_LESS_THAN | DARK_GREATER_THAN | LIGHT_LESS_EQUAL | DARK_GREATER_EQUAL) additionExpression)*  // Relational
    ;

additionExpression
    : multiplicationExpression ((ADD_COLOR | SUBTRACT_COLOR) multiplicationExpression)*  // Additive
    ;

multiplicationExpression
    : unaryExpression ((MULTIPLY_COLOR | DIVIDE_COLOR) unaryExpression)*  // Multiplicative
    ;

unaryExpression
    : GREY_NOT unaryExpression          # NotExpr
    | primary                           # PrimaryExpr
    ;

primary
    : '(' expression ')'                # ParenExpr
    | IDENTIFIER                        # IdentifierExpr
    | NUMBER                            # NumberExpr
    | STRING                            # StringExpr
    | BOOLEAN                           # BooleanExpr
    ;

condition
    : expression                        # ExpressionCondition
    ;

type
    : 'red_int'
    | 'blue_float'
    | 'purple_bool'
    | 'green_string'
    | 'yellow_char'
    ;
```

```
// Lexer rules
CYAN_AND: 'cyan_and';
MAGENTA_OR: 'magenta_or';
GREY_NOT: 'grey_not';
ADD_COLOR: 'add_color';
SUBTRACT_COLOR: 'subtract_color';
MULTIPLY_COLOR: 'multiply_color';
DIVIDE_COLOR: 'divide_color';
LIGHT_LESS_THAN: 'light_less_than';
LIGHT_LESS_EQUAL: 'light_less_equal';
DARK_GREATER_THAN: 'dark_greater_than';
DARK_GREATER_EQUAL: 'dark_greater_equal';
BRIGHTDARK_EQUAL: 'brightdark_equal';
BRIGHTDARK_NOTEQUAL: 'brightdark_notequal';
TRANSPARENT_IF: 'transparent_if';
OPAQUE_ELSE: 'opaque_else';
TRANSLUCENT_ELSEIF: 'translucent_elseif';
VIOLET_WHILE: 'violet_while';
BLUE_FOR: 'blue_for';
SHADE_CHECK: 'shade_check';
CONTRAST_DO: 'contrast_do';
SPECTRUM_DISPLAY: 'spectrum_display';
BREAK_COLOR: 'break_color';
CONTINUE_COLOR: 'continue_color';

BOOLEAN: 'white' | 'black';
NUMBER: [0-9]+ ('.' [0-9]+)?;
STRING: '"' .*? '"';
IDENTIFIER: [a-zA-Z_][a-zA-Z_0-9]*;

WS: [ \t\r\n]+ -> skip;
COMMENT: '//' ~[\r\n]* -> skip;
BLOCK_COMMENT: '/' .? '*/' -> skip;
```

3. Token Definitions

The tokens in the Spectra programming language are defined as follows:

- **Boolean Operators:**
  - cyan_and: Logical AND operation
  - magenta_or: Logical OR operation
  - grey_not: Logical NOT operation
- **Data Types:**
  - red_int: Integer data type
  - blue_float: Floating-point data type
  - purple_bool: Boolean data type
  - green_string: String data type
  - yellow_char: Character data type
- **Arithmetic Operators:**
  - add_color: Addition (+)
  - subtract_color: Subtraction (-)
  - multiply_color: Multiplication (*)
  - divide_color: Division (/)
- **Relational Operators:**
  - light_less_than: Less than (<)
  - light_less_equal: Less than or equal (<=)
  - dark_greater_than: Greater than (>)
  - dark_greater_equal: Greater than or equal (>=)
  - brightdark_equal: Equal (==)
  - brightdark_notequal: Not equal (!=)
- **Conditional Statements:**
  - transparent_if: if statement
  - opaque_else: else statement
  - translucent_elseif: else if statement
- **Loops:**
  - violet_while: while loop
  - blue_for: for loop

- **Ternary Operator:**
  - shade_check: ? in ternary
  - contrast_do: : in ternary
- **Print Statement:**
  - spectrum_display: print function
- **Additional Tokens:**
  - Assignment: equals (=) for assignment.
  - Separators: semicolon (;) to end statements, comma (,) for list separation.
  - Boolean Literals: bright_true (true), dark_false (false).
  - Grouping: open_paren ((), close_paren ()), open_brace ({), close_brace (}).