SKYLOAD TEAM 18: LANGUAGE DESIGN AND GRAMMAR

1. OVERVIEW:

The main aim of this project is to create a simple, minimalist scripting language that can perform arithmetic and logical operations and provide a simple syntax, allowing the user to write and execute programs efficiently. A few key features include:

- **Arithmetic Operations:** SkyLoad supports basic arithmetic operations (+, -, *, /, %).
- **Boolean Logic:** and, or, not, from operations for conditional statements and loops.
- **Relational Comparisons:** Allows the use of comparison operators like >, <, ==, >=, <=, != etc.

· **Control Structures:** Conditional statements like ‘if-else’ statements and support for ‘for’ and ‘while’ loops for repeated executions.

2. GRAMMAR:

SkyLoad language’s grammar is defined using the Extended Backus-Naur Form (EBNF). is a syntax notation used to formally define the grammar of programming languages, document formats, and data structures. It is an extension of **BNF (Backus-Naur Form)**, offering more expressive options for specifying complex syntax rules.. EBNF is written and saved with the extension of ‘.g4’.

SkyLoad Grammar:

```
grammar SkyLoad;

// Lexer rules (tokens)
YES         : 'yes';
NO          : 'no';
DO          : 'do';
PERFORM          : 'perform';
```

```
WITH        : 'with';
SHOW        : 'show';
DECIDE      : 'decide';
ALTERNATIVE   : 'otherwise';
AND         : 'and';
OR          : 'or';
NOT         : 'not';
PLUS        : '+';
MINUS       : '-';
MULTIPLY          : '*';
DIVIDE      : '/';
MODULO            : '%';
LESS_THAN         : '<';
LESS_EQUAL    : '<=';
GREATER_THAN  : '>';
GREATER_EQUAL:  '>=';
EQUALS            : 'is';
NOT_EQUALS    : '!=';
ASSIGN    : ':=';
QUESTION          : '?';
COLON     : ':';
SEMI        : ';';
COMMA             : ',';
LPAREN            : '(';
RPAREN            : ')';
LBRACE            : '{';
RBRACE            : '}';
INT           : [0-9]+;
ID            : [a-zA-Z_][a-zA-Z_0-9]*;
STRING    : '"' .*? '"';
WS            : [ \t\r\n]+ -> skip;
COMMENT         : '#' ~[\r\n]* -> skip;

// Parser rules

program     : instruction+ ;

instruction
     : declareVar
     | assignment
```

```
    | decision
    | forLoop
    | whileLoop
    | showStatement
    ;

declareVar
    : 'define' ID ASSIGN (expression | ternary | condition) SEMI
    ;

assignment
    : ID ASSIGN (expression | condition ) SEMI
    ;

decision
    : DECIDE LPAREN condition RPAREN LBRACE instruction* RBRACE
(ALTERNATIVE LBRACE instruction* RBRACE)?
    ;

ternary:
    | condition QUESTION expression COLON expression  // Ternary operator
    ;

forLoop
    : PERFORM LPAREN declareVar WITH condition SEMI WITH assignment RPAREN
LBRACE instruction* RBRACE
    ;

whileLoop
    : DO LPAREN condition RPAREN LBRACE instruction* RBRACE
    ;

showStatement
    : SHOW LPAREN expression (COMMA expression)* RPAREN SEMI
    ;

condition
    : expression (LESS_THAN | LESS_EQUAL | GREATER_THAN | GREATER_EQUAL
| EQUALS | NOT_EQUALS | AND | NOT | OR) expression
    | condition AND condition
```

```
    | condition OR condition
    | NOT condition
    | YES
    | NO
    ;

expression
    : expression (PLUS | MINUS | MULTIPLY | DIVIDE| MODULO | OR | NOT | AND )
expression
    | ID
    | INT
    | STRING
    | LPAREN expression RPAREN
    ;
```

2.1. Generating the Tokenizer and Parser:

To generate the Tokenizer and Parser we can use ANTLR (for SkyLoad we used ANTLR with code generation target as Java). For this, we need to have the ANTLR jar file. This file can be downloaded from the ANTLR website. The downloaded file is saved as 'antlr-4.13.2-complete.jar'.

Parser and Lexer for the language grammar can be generated by running this command in the command prompt:

java -jar antlr-4.13.2-complete.jar -Dlanguage=Java SkyLoad.g4

Make sure to have the grammar specification already defined in Skyload.g4 file in the same location as the antlr…jar file.

This command generates the necessary java files. After generating the Lexer and Parser files, clear Previous Builds and Package the Project: Use the Maven clean command to delete any old build files. Then, compile the project and generate a JAR file by running: mvn clean package. The resulting JAR file, usually named SKYLOAD-1.0-SNAPSHOT.jar, will be located in the target directory.

Set Up Your Skyload Source File: Place the ".sky" file you intend to test in the designated directory: src/main/java/data/yourfilename.sky. Run the Skyload File: Execute the following command from the command line or terminal:

java -jar <.jar file location> <skyload .sky file location>.

This generates a parsed output of our program code. It should look as below:

```
#PS C:\Users\Acer\IdeaProjects\SKYLOAD> java -jar target/SKYLOAD-1.0-
SNAPSHOT.jar src/main/java/data/Factorial.sky

Parse tree: (program (instruction (declareVar define n := (expression 5) ;))
(instruction (declareVar define factorial := (expression 1) ;)) (instruction
(declareVar define i := (expression 1) ;)) (instruction (forLoop perform (
(declareVar define i := (expression 1) ;) with (condition (expression i) <=
(expression n)) ; with (assignment i := (expression (expression i) +
(expression 1)) ;) ) { (instruction (assignment factorial := (expression
(expression factorial) * (expression i)) ;)) })) (instruction (showStatement
show ( (expression "Factorial of ") , (expression n) , (expression " is: ") ,
(expression factorial) ) ;)))
```

This parse tree is generated for the below code:

```
# Define the number for which we want the factorial

define n := 5;



# Initialize the factorial result to 1

define factorial := 1;



# Initialize a counter variable

define i := 1;



# Perform loop to calculate factorial (factorial = factorial * i for i = 1 to
n)

perform (define i := 1; with i <= n; with i := i + 1;) {

   factorial := factorial * i;
```

```
}



# Display the result

show("Factorial of ", n, " is: ", factorial);
```