Design Document

Team-20
Team Members-
Ajinkya Abhinay Pise, Manmeet Singh, Debesh Kumar Pradhan,
Rahul Balulmath, Jay Sachin Chiddarwar
Git Link-[**jaychiddarwar/SER502-Spring2023-Team20 (github.com)**]

1. **LANGUAGE NAME** - DevLingo

2. **Language Design/Constraints** :
    - Primary Data Types-
        a) int (integer)
        b) boolean (boolean)
        c) chars (character)
        d) float (floating point)
    - Arithmetic Operations for Binary Numbers -
        - "+" - Addition
        - "-" - Subtraction
        - "*" - Multiplication
        - "/" - Division
        - "%" - Modulus
    - Assignment Operator - "="
    - Comparison Operators -
        - "=="
        - "<"
        - ">"
        - "<="
        - ">="
    - Logical Connectives-
        - and
        - or
        - not

    - Conditional Operator -
        - if(condition){

```
                    "Statement"
                            }
    [optional]Else{
                    "Statement"
```

}

- **Iteration Operators -**
  - for(data type ; condition; assignmentExpression){
    “Statement”
    }
  - for (data type) in range(condition){
    “Statement”
    }

  - while(condition){
    “Statement”
    }

- Reserved Keywords -
  - int
  - boolean
  - chars
  - float
  - if
  - else
  - while
  - range
  - for
  - true
  - false
  - printf

- Symbols Used -
  - =
  - ==
  - <
  - >
  - <=
  - >=
  - +
  - -
  - *

- /
- {
- }
- ;
- ,
- “
- (
-)

3. **Grammar Of DevLingo** -

Characters

<primitiveDataType> ::= float | int | boolean | string | chars
<chars> ::= 'a' | 'b' | 'c' | 'd' | ...
<digits> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<comparisonOperator> ::= '<' | '>' | '<=' | '>=' | '==' | '!='
<arithmeticOperator> ::= '+' | '-' | '*' | '/' | ‘%’
<assignmentOperator> ::= '='
<booleanType> ::= False | True
<number> ::= <number> <digits> | <digits>

Conditional Expression

<statement> ::= <expression> ';' | <whileLoop> ';' | <ifBlock> ';' | <declarationST> ';' | <forLoop> ';' | <printStatement> ';'
<statements> ::= <statement> <statements> | ε

Assignment Expression

<assignmentExpression> ::= <iden> <assignmentOperator> <mathExpression>
<declarationST> ::= <primitiveDataType> <iden> | <primitiveDataType> <assignmentExpression>

Comparison Expression

<comparisonTr> ::= <iden> | <number> | '(' <mathExpression> ')'
<comparisonExpression> ::= <comparisonTr> <comparisonOperator> <comparisonTr> | <logicalExpression>

Expression

<expression> ::= <assignmentExpression> | <mathExpression> | <comparisonExpression> | <ternaryOperator> | <logicalExpression> | <stringExpression>

Identifier

<iden> ::= <chars> | <iden> <chars> | <iden> <digits>

Mathematical Expressions

<mathExpression> ::= <mathematicalTerm> '+' <mathExpression>
| <mathematicalTerm> '-' <mathExpression>
| <mathematicalTerm>
<mathematicalTerm> ::= <mathematicalFactor> '*' <mathematicalTerm>
| <mathematicalFactor> '/' <mathematicalTerm>
| <mathematicalFactor> '%' <mathematicalTerm>
| <mathematicalFactor>
<mathematicalFactor> ::= <number> | '(' <mathExpression> ')' | <mathematicalFactor> '%' <mathematicalFactor>

While Loop

<whileLoop> ::= 'while' '(' <expression> ')' <statement>

For Loop

<forLoop> ::= 'for' '(' <declarationST> ';' <comparisonExpression> ';' <assignmentExpression> ')' <statement>

Ranged For Loop

<rangedForLoop> ::= 'for' '(' <declarationST> 'in' <iden> ')' <statement>

If block

<ifBlock> ::= 'if' '(' <expression> ')' <statement> | 'else if' '(' <expression> ')' <statement> 'else' <statement>

Print Statement

<printStatement> ::= 'printf' '(' <expression> ')'

Logical statements

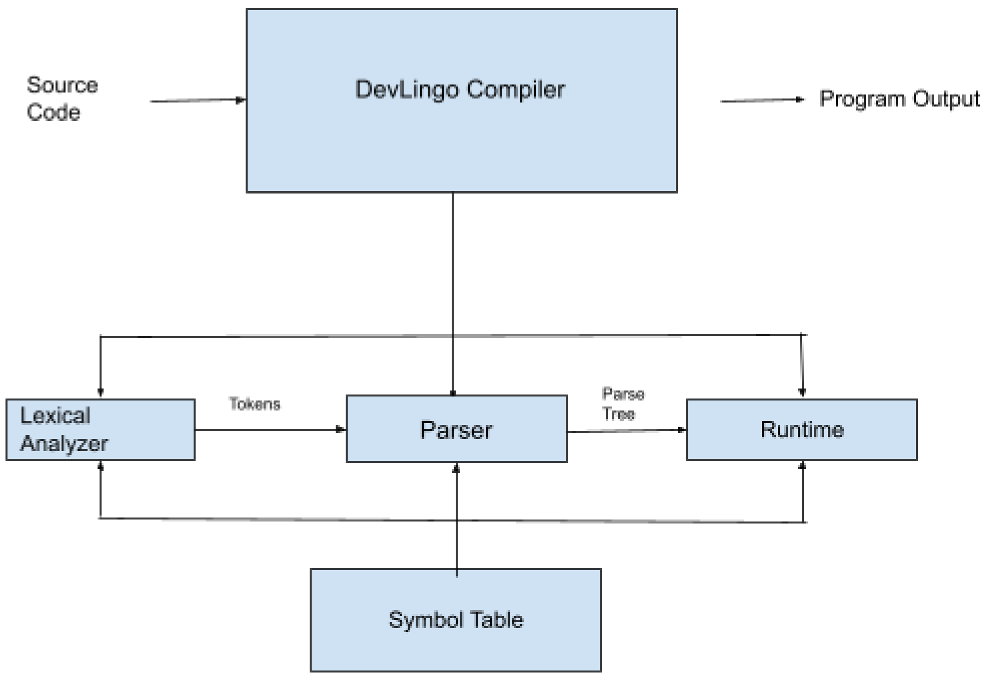
<logicalExpression> ::= <booleanType> | <comparisonExpression> <logicalOperator> <logicalExpression> | '(' <logicalExpression> ')' <logicalOperator> <logicalExpression> | <unaryLogicalOperator> <logicalExpression>

<logicalOperator> ::= 'and' | 'or'
<unaryLogicalOperator> ::= 'not'

Ternary Operator
<ternaryOperator> ::= <comparisonExpression> '?' <expression> ':' <expression>

<stringExpression> ::= '"' <chars>* '"'

4. COMPILER ARCHITECTURE



5.
The use case requirement involves ANTLR utilizing two implementation aspects:

1. Regulating the lexical review of the DevLingo program.

2. Producing the parse tree.

To attain this, ANTLR will be given the already-existing grammar of DevLingo, which will let us know, how the input program should be parsed. ANTLR then divides the grammar file into two parts, Lexer and Parser. The Lexer studies the DevLingo program as an input stream from a file and transforms it into tokens for additional development. These tokens are then passed to the Parser, which relates them to the assumed grammar of the language. The Parser creates a parse tree from the input, which is used as a midway code to produce the required output in the runtime environment implemented in Python. The instructions are accessed based on the data structure format in which the code is stored. Finally, the output generated in the runtime environment is sent back to Python and printed out to the console.

6. The following Data Structures will be used in future

Trees - The parser will be using Trees as a data structure to produce parse trees for the origin code. The parse trees will be made up of tokens, which will be stored in Inorder tree format for each expression. The Interpreter will utilize these parse trees to generate meaning to the expressions and keep their outputs.
Dictionary -To execute the symbol table, a Dictionary data structure will be utilized. Both the parser and Interpreter will employ dictionaries to convert the origin code to intermediate code. Variables will be kept as keys in the symbol table, which will have their corresponding data values.
Lists - In addition, various Lists will be used to keep keywords, identifiers, and mathematical symbols. The whole program will consist of a list of statements. Lists will also be helpful in the execution and examination of Grammar during the parsing and interpretation stage.