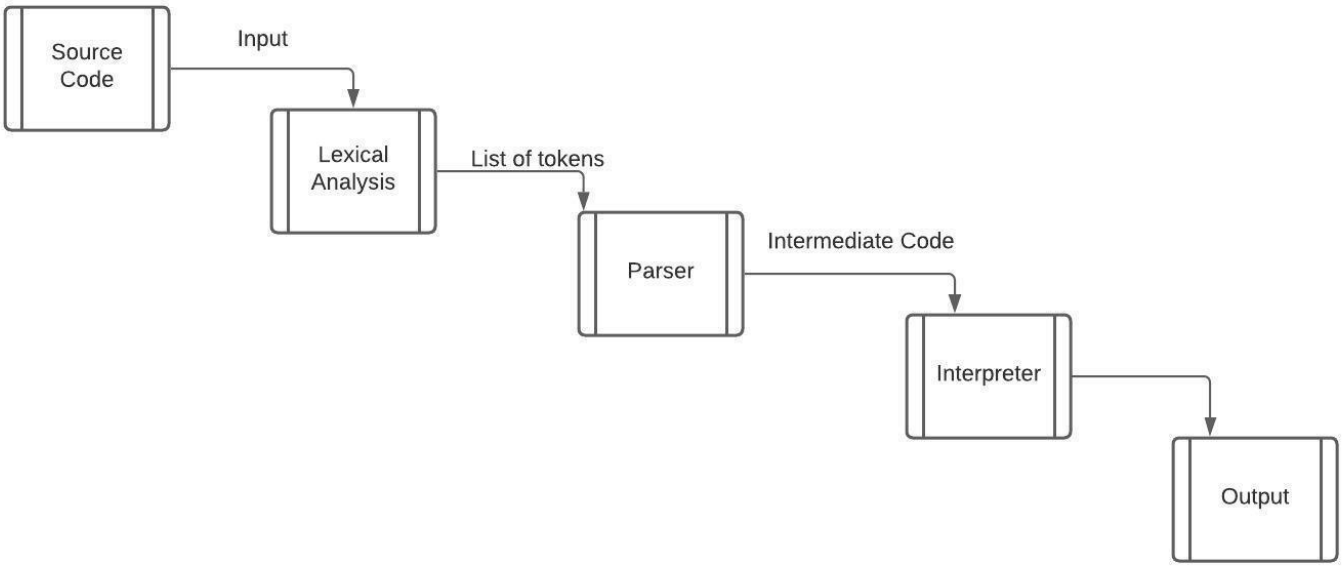
Amulya Bodla (1219477220)
Niharika Pothana (1219596937)
Ritesh Reddy (1220005293)
Piyush Reddy (1219456537)

VERTEX (Team-2)

Programming Language Name: Vertex
Language extension: .tex

Programming paradigm: Imperative
Design/Structure:



1. Source Code: The source code is a file that has to be executed. It has a .tex extension.
2. Lexical Analyzer: The lexical analyzer takes the input source code file and reads this file thereby generating a list of tokens. The tokens are generated as per the grammar rules of Vertex language. This list of tokens is stored in a List data structure. Prolog is used to generate the tokens list.
3. Parser: Once the list of tokens is generated, it is passed to the Parser. The parser reads each token at a time as per the grammar rules and generates a Parse tree using Prolog. This step is completed successfully after all the tokens are parsed. The predicates for grammar are written as per DCG (Definite clause grammar). The generated parse tree would be the intermediate code which is passed to the next stage.
4. Interpreter: Interpreter reads the intermediate code (i.e., parse tree) by traversing through each of the nodes in the parse tree. Each of the nodes would be evaluated using the semantic rules and updated in the list accordingly. Interpreter would be implemented in Prolog.

Tools used:
We would be using Prolog to design our Lexical analyzer, Parser and Interpreter. The grammar rules are written using a BNF notation.
Data structure used: List

Datatypes:
Vertex supports 3 data types: int, bool, string

- int: supports all positive numbers.
- bool: supports values, 'true' and 'false' to variables.
- string: supports string value assignments to variables.

Declaration:

- A syntax must be followed while declaring any variable.
  Ex: int i; (to declare a variable i of type int)
- Identifier names should start with a lowercase alphabet and can consist numbers, uppercase alphabets., etc.

Operators:
Vertex supports the below operators:

- Arithmetic operators: Addition, subtraction, multiplication, division denoted using '+', '-', '*' and '/'.
- Boolean operators: 'and', 'or', 'not' which are used to check the validity in loops and conditional statements.
- Comparison operators: Less than, Greater than, Less than or equal to, Greater than or equal to, equals, not equals denoted using '<', '>', '<=', '>=' '==', '!='.

Conditional statements:
Vertex supports the below conditional statements:

1. if-then
   A traditional 'if' loop to check for a statement which when evaluated to true, the 'then' block is executed.
   Ex:

```
if(a!=0)
then
{a=2;}
;
```

2. if-then-else
   Here 'if' condition is evaluated and when true, the 'then' statements are executed. Otherwise, 'else' statements are executed.
   Ex:

```
if (x==2)
then
{x=x+2;}
else
```

```
{x=x+3;}
;
```

3. Ternery operator
   This is a shorthand notation for if-else implementation and denoted by
   (conditional expression)?(if the previous expression returns true, evaluate expression 1):(else evaluate expression 2);
   Ex: (x==2)?(x=x+2):(x=x+3);
   This statement works the same as below:

```
if (x==2)
then
{x=x+2;}
else
{x=x+3;}
;
```

Iterative statements:

1. for
   This is a traditional 'for' statement which has three parameters, the first one handles initialization, the second one checks for the condition, the third expression handles the looping.
   Ex: for(i=0;i<2;i=i+1)

```
{
    print i;
};
```

2. for i in range(a,b)
   This is a modified for loop which acts in the same way as a traditional 'for-each' loop.
   Ex: for i in range(1,5) behaves the same as for(i=1;i<5;i++)
3. while
   'While' loop condition when evaluated to true, executes the enclosed block of statements and the loop continues until condition is evaluated to be false.
   Ex:

```
while(x<2)
{
    x=x+1;
};
```

Print statement:

This displays the value of identifiers entered after 'print' onto the console.

Terminate statements:

We use 'start' and 'end' keywords to begin and end a given block of code.
Delimiter: We use a semicolon (;) in Vertex to terminate a particular line.

GRAMMAR:

program → START block END.
block → { commands_list }.
commands_list → commands ; commands_list | commands ; .
commands → print | declare | assign | if-else | while | for | ternary |expr .

declare → DATATYPE VARCHAR .
assign → VARCHAR ASSIGNOP expr | VARCHAR ASSIGNOP ALPHANUMERIC.
if-else → IF (compositeBool) THEN block | IF (compositeBool) THEN block ELSE block .
for → FOR (loopScope) block| FOR VARCHAR IN RANGE (NUMBERS,NUMBERS) block.
loopScope →VARCHAR ASSIGNOP expr DELIMITER VARCHAR COMPAREOP compositebool ; commands.
while → WHILE (compositebool) block.
ternary → (compositebool) TERNARYOP expr TERNARYOP1 expr.
print → PRINT expr|PRINT member.
compositeBool → bool_expr COMPOSITEBOOLOP compositeBool | bool_expr.
bool_expr → expr COMPAREOP expr | NOT expr | BOOL.
expr → component ADDSUB expr | component.
component → member MULDIV component | member .
member → NUMBERS|VARCHAR.

NUMBERS → ^[0-9]+$
VARCHAR → ^[a-zA-Z_$][a-zA-Z_$0-9]*$
ALPHANUMERIC→ ^[a-zA-Z_$0-9]*$
PRINT → print
ASSIGNOP → ‘=’
COMPAREOP → ‘<=’ | ‘<’ | ‘>’ | ‘>=’ | ‘==’ | ‘! =’
BOOL → true | false.
ADDSUB → '+'|'-'.
MULDIV → '*'|'/'.
START → start.
END → end.
DATATYPE → int | bool | string .
COMPOSITEBOOLOP → AND | OR
TERNARYOP → ?
TERNARYOP1 → :
IF → if.
ELSE → else.
THEN → then.
FOR → for.
WHILE → while.
IN → in.
RANGE → range.