KVS

SER 502 – Project – Team 28 – Spring 2022

Members: Kavya Alla - 1225990508 – kalla1.

Vedasree Bodavula - 1225885273 – vbodavul.

Satvik Chemudupati – 1225665038 – schemudu.

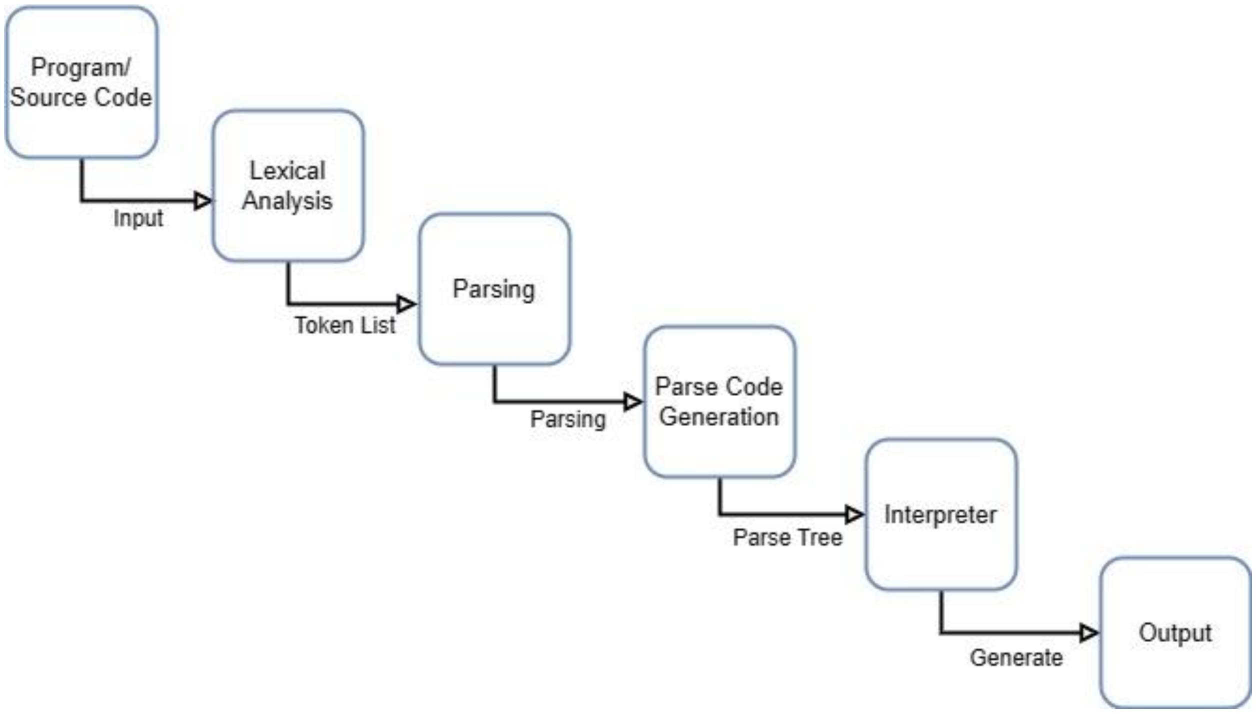
Sai Sunil Neralla – 1225718832 - snerall1.

GitHub Repository Link: - https://github.com/SatvikChemudupati/SER502-Spring2023-Team-28.git.

Programming Language Name - KVS

Programming Language Extension - .kvs.

The following stages of execution are used to demonstrate the language compilation:



The Flow Structure of Language and Execution Process:

The role of the lexical analyzer is to scan the input program and recognize the tokens present in it. When requested by the parser, the scanner produces these tokens. On the other hand, the interpreter uses the parse tree generated by the parser to verify if the code adheres to the semantic rules of the language. Finally, after evaluating the program, the interpreter generates the output.

- **Input Program**: The program files with a. kvs extension are given as input into the system.

- **Lexical Analyzer**: The lexical analyzer scans the source code and reads the characters from the file. It then converts these characters into tokens recognized by the KVS language. These tokens are stored in a token list in the parsed sequence. We adhere to Prolog and Python as the main languages to design the proposed language.

- **Parser**: The parser's job is to read the tokens from the token list and check if the syntax rules of the Kvs language are followed in the source code. The parser generates the parse tree from the tokens if the syntax is correct or returns an error message if it has any syntactical errors. Definite Clause Grammar (DCG) is the grammar technique used.

- **Intermediate Code**: The parser creates an intermediate code file containing the parse tree.

- **Interpreter**: The interpreter reads the intermediate code file and evaluates each node in the parse tree using syntax-based semantics using a top-down approach. We implement List data structure in Python as the standard data structure.

- **Output**: The output of the program is generated after its evaluation.

Language Design:

1. Fundamental Parameters:

   - **Data type** – num, bool, string
   - **Identifier** – [a - z], lowercase, uppercase, [0-9], special characters like $.
   - **Defined keywords** – num, bool, nLesser, nGreater, Lesserthan, Greaterthan, equalto, nequalto, while, for, if, else.

2. Arithmetic Operators:

   The arithmetic operators that are standard norms are implemented.

- **Multiplication** – a*b – Multiply a value to b.
- **Addition** – a+b – Add the value of a to b.
- **Subtraction** – a – b - Subtract the value of b from a.
- **Modulus** - a//b - The output value would be the remainder of a divided by b.
- **Division** – a / b – The quotient value of a divided by b will be the output.

3. Comparison Operators:

- **Greater than operator** – A Greaterthan B – Checks if A is greater than B.
- **Less than operator** – A Lessthan B – Checks if A is less than B.
- **Greater than or equal to** – A nLesser B – Checks if A is greater than or equal to B.
- **Less than or equal to** – A nGreater B – Checks if A is less than or equal to B.
- **Equal to operator** – A equalto B – Checks if A is equal to B.
- **Not Equal to operator** – A nequalto B – Checks if A is not equal to B.

4. Logical Operators:

- **AND** Operator – A && B – Checks (A and B) logical condition
- **OR** Operator – A || B – Checks (A or B) logical condition.
- **NOT** Operator – A !! B – Checks Not A operator.

5. Iterative loops:
   It supports '**for'**, '**if-else'**, '**while'** and '**for in range'** loops and block conditions. Like all programming languages, the compiler will loop the by the defined number of times and executes the block.

➢ Comments can be added to the code by using a special symbol "#".
➢ The only limitation in comments is the language only the single line of comments.
➢ The "disp" command is used to display the output requested by the user.

KVS Language Grammar:

Fundamental Definitions:

NUM: = /^[0-9]+$/

BOOL: = / ‘True’ | ‘False’/

STR: = /\” [\x00-\x7F]*\”/

CHAR: = /'[\x00-\x7F]'/

DATATYPE: = ‘int’ | ‘bool’ | ‘string’

IDENTIFIER: = /^[a-z, A-Z_$][a-zA-Z_$0-9]*$/

ADDITION: = ‘+’

SUBTRACTION: = ‘-’

MULTIPLICATION: = ‘*’

DIVISION: = ‘/’

AND OPERATOR: = ‘and’

OR OPERATOR: = ‘or’

NOT OPERATOR: = ‘not’

ASSIGNING VALUE: = ‘=’

COMPARING VALUES: = ‘>’ | ‘<’ | ‘<=’ | ‘>=’ | ‘==’ | ‘!=’

CONDITIONAL OPERATORS: = and | or | not

IF-BLOCK: = ‘if’

ELSE-BLOCK: = ‘else’

ELSEIF-BLOCK: = ‘elseif’

WHILE-BLOCK: = ‘while’

FOR-BLOCK: = ‘for’

IN-LOOP: = ‘in’

RANGE-CONDITION: = ‘range’

COMMENTING: = ‘#’

START BLOCK: = ‘{‘

END BLOCK: = ‘}’

DLR: = ‘;’

COMMA: = ‘,’

STARTING QUOTE: = ‘”’

ENDING QUOTE: = ‘”’

BEGIN: = ‘start’

END: = ‘terminate’

OPAR: = ‘(‘

CPAR: = ‘)’

DISPLAY:= ‘disp’

Program Rules:

program: = START statements END with TERMINATE | comment statements with START statements and end with TERMINATE.

statement: = all statements DLR statements | statements

every statement: = disp | declaration | assign | if-else | while | for

declaration: = DATATYPE SPACE IDENTIFIER ASSIGN data | DATATYPE SPACE IDENTIFIER

assign: = IDENTIFIER ASSIGN expression

print: = DISP SPACE STARTING QUOTE STRING ENDING QUOTE | DISP SPACE IDENTIFIER | DISP

STARTING QUOTE STRING ENDING QUOTE | DISP IDENTIFIER

if else: = IF OPAR condition CPAR STARTING BLOCK statements

DLR END BLOCK | IF OPAR condition CPAR STARTING BLOCK

statements DLR END BLOCK DLR, else if Loop DLR ELSE

STARTING BLOCK statements DLR END BLOCK | IF OPAR condition

CPAR STARTING BLOCK statements DLR END BLOCK DLR ELSE

OPAR statements DLR CPAR

else if Loop:= elseifLoop1 DLR else if Loop | elseifLoop1 elseifLoop1: = ELSEIF OPAR condition CPAR STARTING BLOCK

statements DLR END BLOCK

while: = WHILE OPAR condition CPAR STARTING BLOCK statements

END BLOCK

for: = FOR for Range STARTING BLOCK statements END BLOCK

forRange: = OPAR IDENTIFIER ASSIGN expression DLR IDENTIFIER

COMPARING VALUES expression DLR CLOSEPARANTHESIS | OPAR IDENTIFIER

ASSIGN expression DLR IDENTIFIER COMPARING VALUES expression DLR expression.

CPAR | IDENTIFIER IN RANGE OPAR expression COMMA expression CPAR

condition: = IDENTIFIER SPACE COMPARING VALUES SPACE expression | IDENTIFIER SPACE

COMPARING VALUES expression CONDITIONAL OPERATORS condition | BOOL

comment: = COMMENT STRING

expression: = value ADDITION expression | value SUBTRACTION expression | value

value: = factor MULTIPLY value | factor DIVISON value | factor

factor: = OPAR expression CPAR | data | IDENTIFIER

data: = INTEGER | BOOL | STRING