Compile Design Project Report

Project submitted to the
SRM University – AP, Andhra Pradesh
for the partial fulfillment of the requirements to award the degree of

Bachelor of Technology
In
Computer Science and Engineering
School of Engineering and Sciences

Submitted by

Goutham Krishna Yarra	**AP20110010007**
Srinivasarao Botla	**AP20110010014**
Mannath Shaik	**AP20110010048**
Siva Chandra Prasad Panguluri	**AP20110010051**
Uday Kiran Nathani	**AP20110010055**
Avinash Nagandla	**AP20110010066**
Nikhil Chowdary Yamani	**AP20110010067**
T. Bhavesh Kalki Saibabu	**AP20110010705**

SRM University – AP
Neerukonda, Mangalagiri, Guntur
Andhra Pradesh – 522240

[December, 2022]

ACKNOWLEDGEMENT

In the accomplishment of completion of our project on Compiler Design for Javascript using c language, we would like to convey our special gratitude towards our professor Mrs. Jaya Lakshmi, of Computer Science and Engineering. Your valuable guidance and suggestions helped us in various phases of the completion of this project. We will always be thankful to you in this regard.

We would like to appreciate our work that we have done in our project and thank each other for continuous support and coordination.

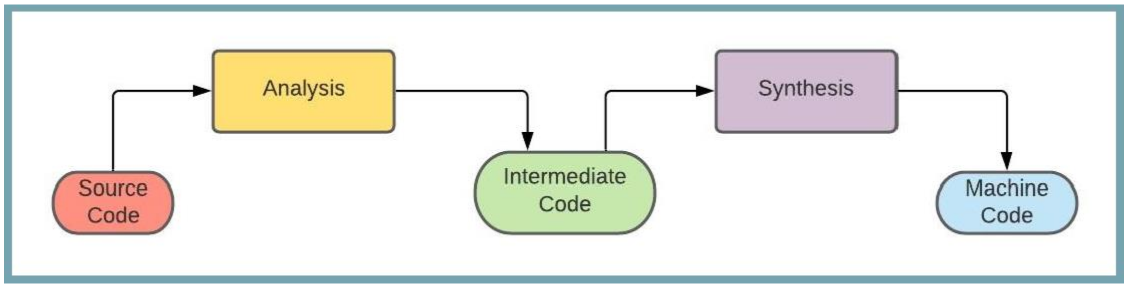
Project Report

→ Language Selected: JavaScript.

INTRODUCTION:

In this project we are designing a compiler which outputs if the given syntax is correct.

A compiler is something that translates the code written in one language to some other language without changing the actual meaning of the program. When executing, the compiler first parses (or analyzes) all of the language statements syntactically one after the other and then, in one or more successive stages or "passes", builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. The compilation process is a sequence of various phases. So here, each and every phase takes input from its previous stage, has its own representation of the source program, and sends its output to the next phase of the compiler. The analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase generates an intermediate code which is also referred as Assembly Language Code.



DEFINITIONS:

- **Lexical analyzer**

  Lexical analysis is the process of converting a sequence of characters from a source program into a sequence of tokens. A program which performs lexical analysis is termed as a lexical analyzer (lexer), tokenizer or scanner.

  Lexical analysis consists of two stages of processing which are as follows:

  • Scanning

  • Tokenization

  Token is a valid sequence of characters which are given by lexeme. In a programming language,

  • keywords,

  • constant,

  • identifiers,

  • numbers,

  • operators and

  • punctuations symbolsare possible tokens to be identified.

  For example : c=a+b;

  In this c,a and b are identifiers and ‘=’ and ‘+’ are mathematical operators.

- **Syntax analyser**

  Syntax analysis is the second phase of compiler. Syntax analysis is also known as parsing.
  Parsing is the process of determining whether a string of tokens can be generated by a grammar.
  It is performed by a syntax analyzer which can also be termed as a parser.
  In addition to construction of the parse tree, syntax analysis also checks and reports syntax errors accurately. Parser is a program that obtains tokens from lexical analyzer and constructs the parse tree which is passed to the next phase of the compiler for further processing.
  Parser implements context free grammar for performing error checks.

  Here we are implementing Top down parsers that construct a parse tree from root to leaves.

● Role of Parser

  - Once a token is generated by the lexical analyzer, it is passed to the parser.
  - On receiving a token, the parser verifies the string of token names that can be generated by the grammar of the source language.
  - It calls the function getNextToken(), to notify the lexical analyzer to yield another token.
  - It scans the token one at a time from left to right to construct the parse tree.
  - It also checks the syntactic constructs of the grammar.

❖ Data types available in JavaScript Language:

- Data types using in compiler design are:

▪ Number:

Number type stands for both integers and floating points. Many mathematical operations are carried out using these numbers.

- Example:

var number=30; console.log("Your integer

number is : " + a) let length = 16;

console.log(length);

- ## String:

  Strings are used to store data that involves characters, like names or addresses. You can perform operations like string concatenation, in JavaScript.

  - <u>Example:</u>

    const str = "abc";

    console.log(str); let

    Name = "abc";

    console.log(Name)

    ;

- ## Boolean:

  Boolean type only has 2 types of return values, true and false. This type is usually used to check if something is correct or incorrect.

  - <u>Example:</u>

    let x = 10;
    Boolean(x); // returns true

  - <u>Example2:</u>

    let x = -0;
    Boolean(x); // returns false

❖ Data structures in JavaScript Language:

- Array:

  An array is defined as the collection of similar types of data items stored at contiguous memory locations.

  - Syntax:

    const (<id>) = new Array <val>

  - Example: const(a)= new Array(1 ,2 ,3)

❖ Syntax of variable declaration:

- There are 3 ways to declare a JavaScript variable:

  - var:

    keyword var to declare a variable in JavaScript.

    - Syntax: var <variable-name> = <value>;

  - Let:

    let allows you to declare variables that are limited to the scope of a block statement

    - Syntax: let <variable-name> = <value>;

  - Const:

    Const is another variable type assigned to data whose value cannot and will not change throughout the script.

    - Syntax: const <variable-name> = <value>;

❖ Decision-making statement:

➢ If condition:

If statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

- Syntax:

```
if(condition)

{

   // Statements to execute if

   // condition is true

}
```

- Example:

```
<script type = "text/javaScript"> var i

= 10; if (i < 15) document.write("10

is less than 15");

< /script>
```

➢ If-else:

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with the if statement to execute a block of code when the condition is false.

- Syntax:

```
if (<expr>) {console.log(<id>)} : else {console.log(<id>)}
```

- Example:

```
if (a>b) {console.log(a)} : else {console.log(b)}
```

❖ Iterative statements:

➢ For:

A for loop repeats until a specified condition evaluates to false.

- Syntax:

```
for ([initialExpression]; [conditionExpression]; [incrementExpression])
  statement
```

- Example:

```
for (let i = 0; i < 5; i++) { text += "The
  number is " + i + "<br>";
}
```

➢ While loop:

A while statement executes its statements as long as a specified condition evaluates to true.

- Syntax:

```
while (condition){statement}
```

- Example:

```
while (a > b){console.log(a)}
```

➢ do-while loop:

The do-while statement creates a loop that executes a specified statement until the test condition evaluates to false. The condition is evaluated after executing the statement, resulting in the specified statement executing at least once.

- Syntax:

```
do{console.log()}while (condition)
```

- Example:

```
do{console.log(a)}while (a > b)
```

❖ Functions:

A JavaScript function is executed when "something" invokes it (calls it). A JavaScript function is defined with the function keyword, followed by a name, followed by parentheses ().

- Syntax:

  function[<name>]{<fun_val>}

- Example:

  function [a(a, b)]{return a}

❖ Context Free Grammar

➢CFG for Number:

Integer : signed/unsigned integer

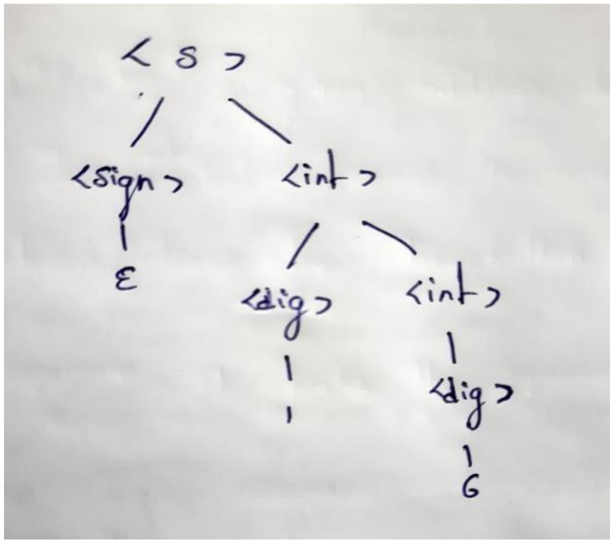
S → <sign><Int>;

<sign> → + | - | ε

<int> → <dig>|<dig><int>

<dig> → 0 | 1 | 2 | 3…. | 9

Parse tree:



➢CFG for Operators :

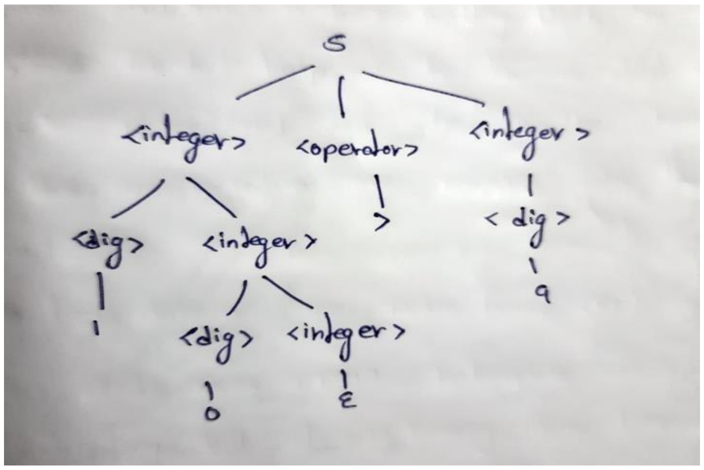
<S> → <integer><operator><integer>

<integer> → <dig> | <dig><integer>

<operator> → > | < | ==

<dig> → 0 | 1 | 2 | 3…. | 9

ParseTree:



- CFG for arithmetic operations :

<exp> → <term><operator><term>

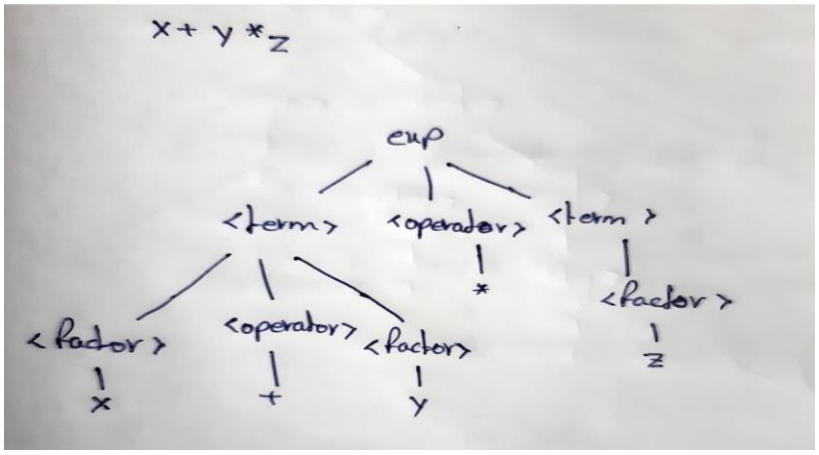
<term> → <factor><operator><factor>|<factor>

<operator> → + | - |* |/ |%

<factor> → a-z | A-Z

Parse Tree:

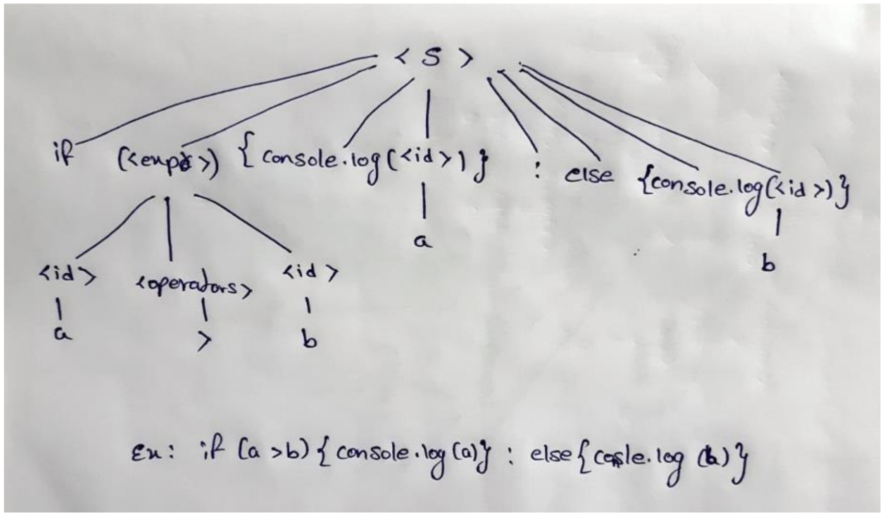
Example: X+Y*Z

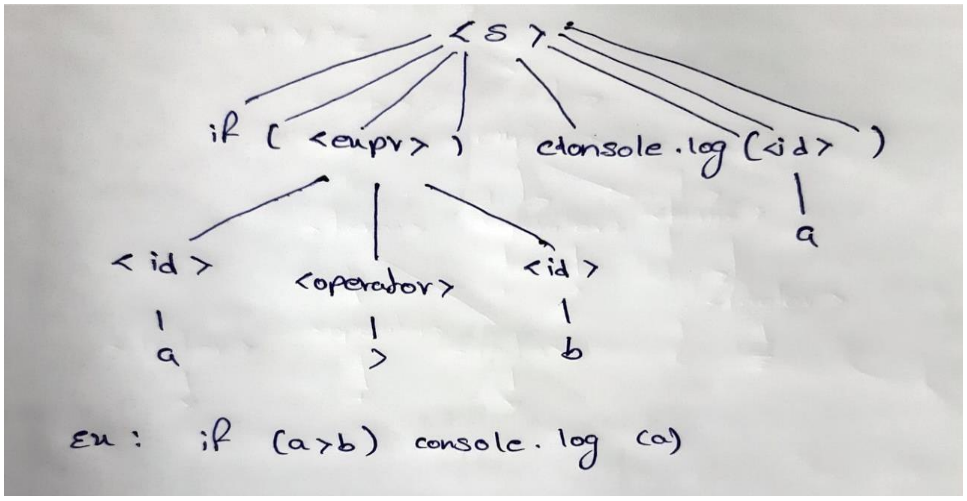


➢CFG for if-else :

<S> → if (<expr>) console.log(<id>)
          | if (<expr>) {console.log(<id>)} : else {console.log(<id>)}
<expr> → <id><operator><id>|<id>
<operator> → > |< | == | >= |<=
<id> → a-z | A-Z | 0-9 | ε

Parse Tree:





➢CFG for for loop :

<S> → for X { <code block> }

X → (<Initialization>; <Condition>; <Increment>)

<Initialization> → <letter> = <num>

<Condition> → <letter><relop><num>

<Increment> → <letter><Y> | <Y><letter>

<relop> → >= | <= | = | > | <

<Y> → ++ | --

<letter> → a-z | a-z<letter>

<num> → 0-9 | 0-9<num>

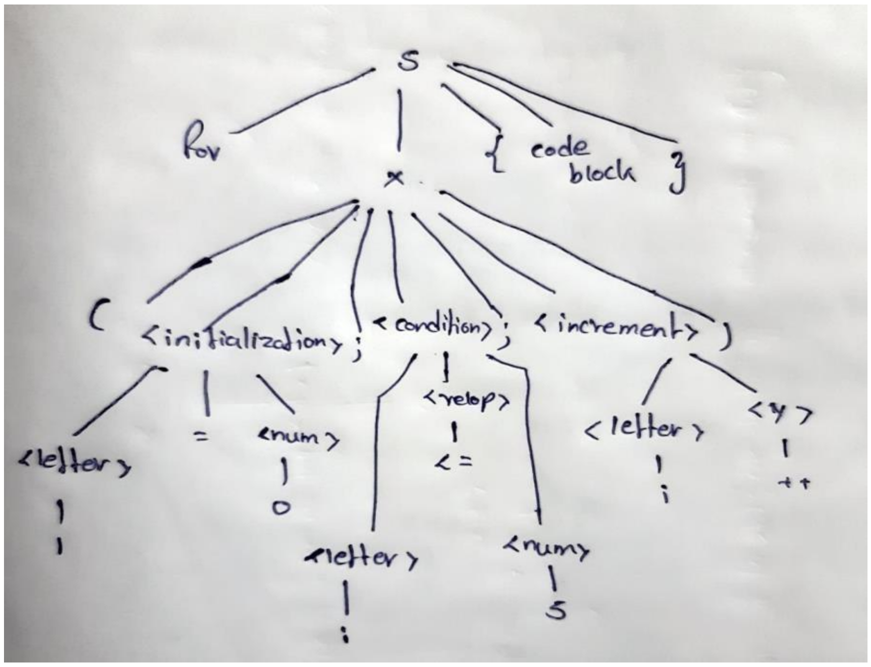
Parse Tree:

Example-

for(i=0; i<=5; i++){

console.log(i);

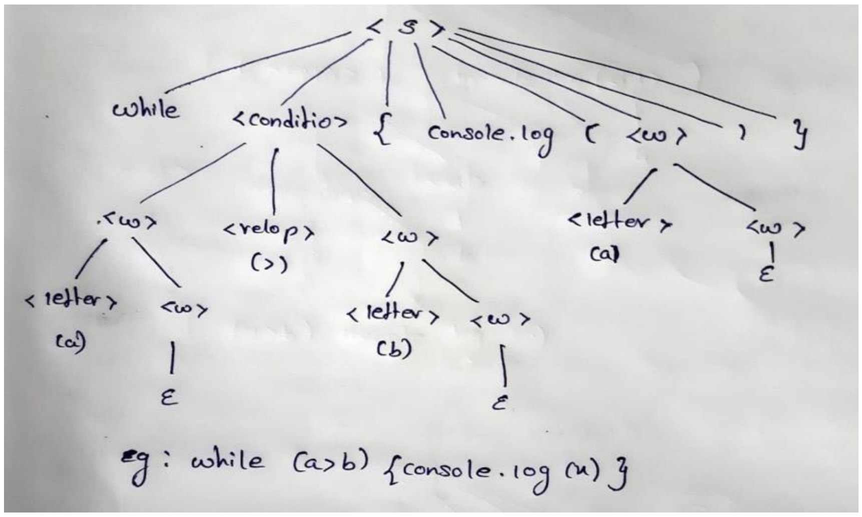
}



➢CFG for While loop :

<S> → while (<condition>){console.log(<w>)}
<condition> → <w><relop><w>
<w> → <letter><w>|<num><w>|ε
<letter> → a-z
<num> → 0-9
<relop> → >|<|==|>=|<=

Parse Tree:



➢CFG for do while loop :

<S> → do {console.log(<id>)} while (<condition>)

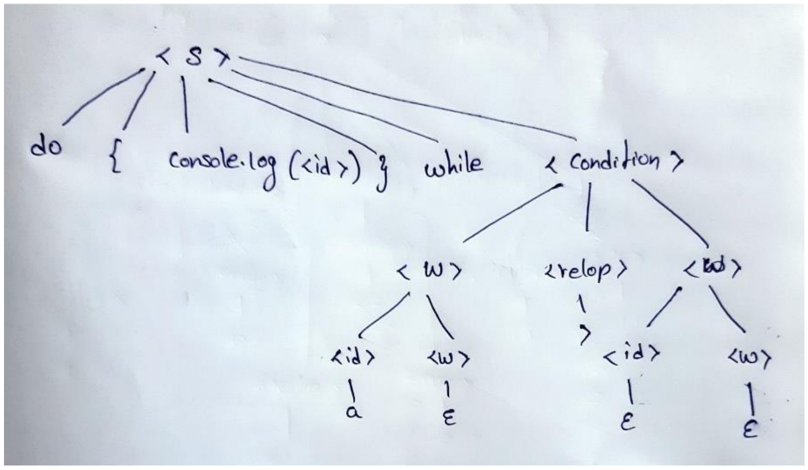
<condition> → <w><relop><w>

<w> → <id><w> | ε

<id> → a-z | 0-9

<relop> → > |< |==|>=|<=

Parse Tree:



Example:

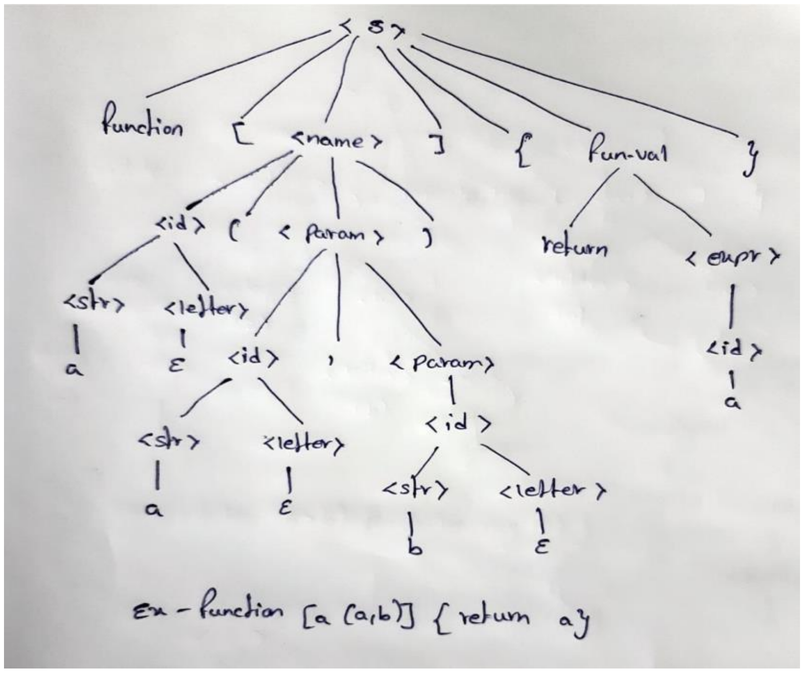
do{console.log(a)}while(a>b)

➢CFG for functions :

<S> → function[<name>]{<fun_val>}
<name> → <id>(<param>)
<id> → <str><letters>
<letters> → <str><letters> | <digit><letters>|ε
<param> → <id>,<param>|<id>|ε
<fun_val> → return <expr>
<expr> → <id>|<id><op><id>
<op> → +|-|*|%|/
<digit> → 0-9

<str> → a-z

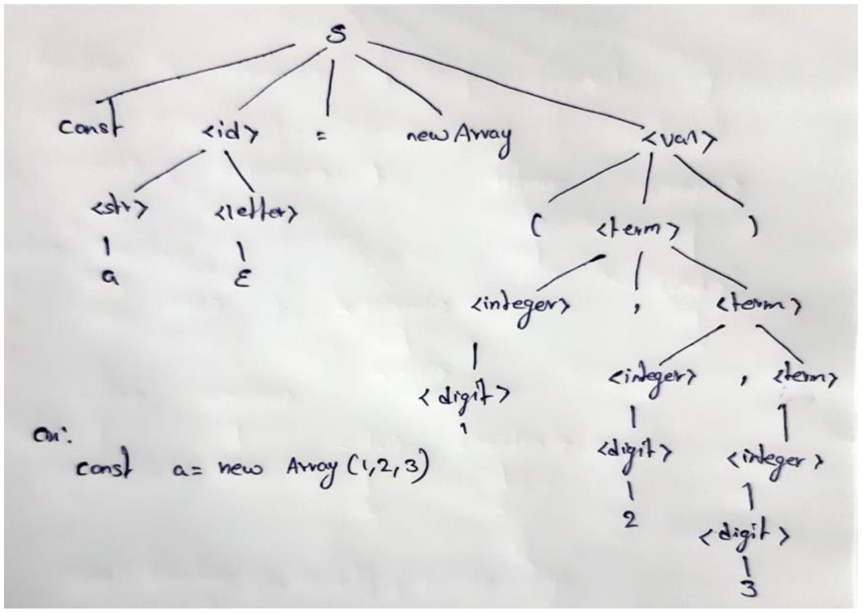
ParseTree:



➢CFG for array :

<S> → const (<id>) = new Array <val>
<id> → <str><letter>
<letter> → <str><letter>|<digit><letter>| ε
<str> → a-z | A-Z
<val> → (<term>| ε)
<term> → <strings>,<term> | <integer>,<term> | <float>,<term> | <strings> |<integer> | <float>
<digit> → 0-9
<strings> → “ <id> “
<integer> → <digit> | <digit><integer> | ε
<float> →<integer> . <integer>

Parse Tree:



❖ Syntax for the target language:

➢ The syntax for Data Structure:

- The syntax for Data Types:

  Syntax: data_type

  variable_name;

  Example:

  ```
  int a=5;
  ```

- The syntax for Array:

  Syntax: data_type array_name[size];

  Example:

  ```
  int marks[5];
  ```

➢ The syntax for Functions:

In Python a function is defined using the def keyword

Syntax: return_type function_name(data_type parameter...){

}

Example:

```
void hello(){
printf("hello c");
}
```

➢Syntax for for loop:

Syntax:

```
for ( init; condition; increment ) {
   statement(s);
}
```

Example: #include

```
<stdio.h> int main

() {

   int a; for( a = 1; a < 10;
   a++ ){
      printf(a);
   }
   return 0;
}
```

➢The syntax for while loop:

Syntax:

```
while(condition) {
   statement(s);
}
```

Example:

```
int main () {
   int a = 1;

   while( a < 10 ) {
      printf(a)
   ; a++; }

   return 0;
}
```

➢ The syntax for do-while loop:

Python doesn't have a do-while loop. Below program is executed similarly to the do while loop.

Syntax:

```
do {
   statement(s);
} while( condition );
```

Example:

```
#include <stdio.h>

int main () {
   int a = 1;

   do {
      printf(a); a = a
   + 1; }while( a <
   10 );

   return 0;
}
```

➢The syntax for if condition:

Syntax:

```
if (condition)
{
  // code
}
```

Example:

```
#include <stdio.h>
int main() {
    int number=10; if
    (number < 20) {
        printf("number is less than 20", number);
    }
    return 0;
}
```

➢The syntax for if-else condition:

Syntax:

```
if(condition){

       //cod

e } else{

      //code

}
```

Example:

```
#include <stdio.h>
int main() {
    int number=10; if
    (number < 20) {
        printf("number is less than 20", number);
    }
   else{
```

```
            printf("number is greater than 20", number);
        }
        return 0;
    }
```

❖ Lexical Analyzer

LEX CODE:

```
%{
#include "project.tab.h"
%}
%%

"if" {return IF;}
"else" {return ELSE;}
"while" {return WHILE;}
"do" {return DO;}
"console.log" {return PRINT;}
"const" {return CONST;}
"new Array" {return NA;}
"function" {return FUN;}
"return" {return RETURN;}
"=" {return EQ;}
[0-9]+ {return NUMBER;}
[0-9]+\.[0-9]* {return FLOAT;}
\'[A-Za-z]*\' {return STRING;}
[A-Za-z][A-Za-z0-9]* {return ID;}
["+"|"-"|"*"|"%"|"/"|"<="|"<"|">="|">"|"=="] {return OPERATOR;}

[\t] ;
\n return 0;
. return yytext[0];
%%
```

❖ Parser

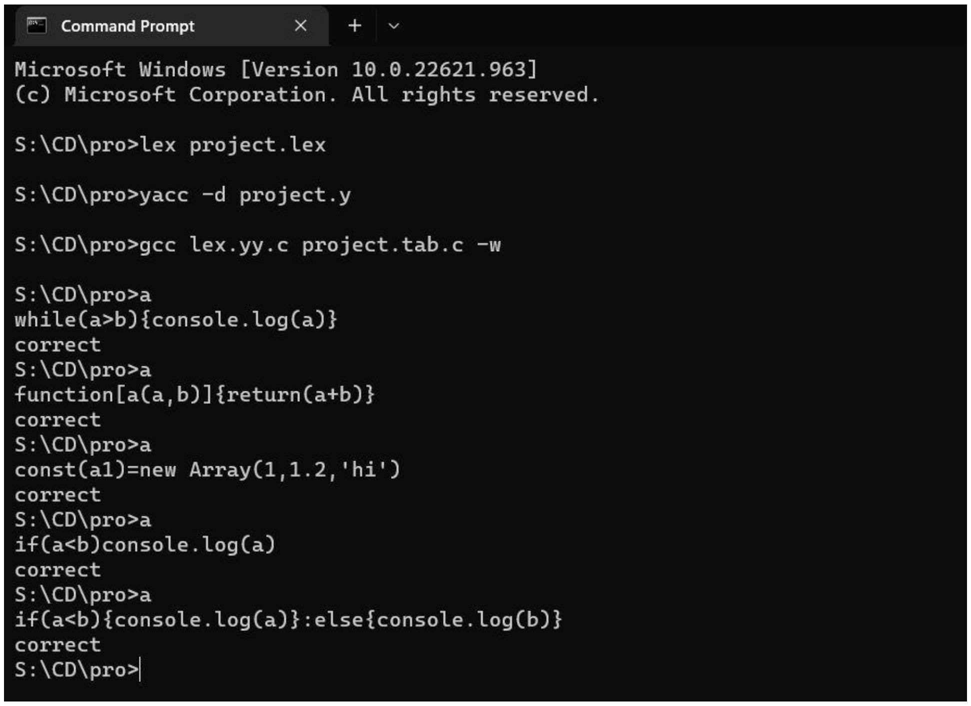
YACC CODE :

```
%{
#include<stdio.h>
%}
%token NUMBER
%token FLOAT
%token STRING
%token ID
%token OPERATOR
%token WHILE
%token DO
%token PRINT
%token CONST
%token NA
%token FUN
%token RETURN
%token EQ
%token IF
%token ELSE
%%
S: WHILE'('C')' '{'PRINT'('ID')' '}'  { printf("correct");}
| DO'{'PRINT'('ID')' '}' WHILE'('C')' { printf("correct");}
| FUN'['N']' '{'FV'}'  { printf("correct");}
| X { printf("correct");}
| IF'(' C ')' PRINT'('ID')'           { printf("correct");}
| IF'(' C ')' '{'PRINT'('ID')' '}' ':' ELSE '{'PRINT'('ID')' '}' { printf("correct");}
;
X: CONST'('ID')'EQ NA V
;
V: '('T')'
;
C: ID OPERATOR ID
;
T: STRING','T
|NUMBER','T
|FLOAT','T
|STRING
```

```
|NUMBER
|FLOAT
;
N: ID'('P')'
;
P: ID','P
| ID
|
;
FV: RETURN'('E')'
;
E: ID
| ID OPERATOR ID
;

%%
int main(){
yyparse();
}
int yywrap(){
return 1;
}
void yyerror(char *s){
printf("Error %s",s);
}
```

OUTPUT :



```
Microsoft Windows [Version 10.0.22621.963]
(c) Microsoft Corporation. All rights reserved.

S:\CD\pro>lex project.lex

S:\CD\pro>yacc -d project.y

S:\CD\pro>gcc lex.yy.c project.tab.c -w

S:\CD\pro>a
while(a>b){console.log(a)}
correct
S:\CD\pro>a
function[a(a,b)]{return(a+b)}
correct
S:\CD\pro>a
const(a1)=new Array(1,1.2,'hi')
correct
S:\CD\pro>a
if(a<b)console.log(a)
correct
S:\CD\pro>a
if(a<b){console.log(a)}:else{console.log(b)}
correct
S:\CD\pro>
```

❖ Issues Faced

We faced issues while generating tokens according to the declaration used in the lex file. While compiling the yacc file we were not able to take a few keywords. This problem is solved by declaring those as keywords and generating them as tokens in the yacc file. Few other errors like giving operators in a particular order.