LE - TRANSPILER

A toy compiler to transpile a custom language to C code

PROJECT REPORT

Eklavya Mentorship Program

At

SOCIETY OF ROBOTICS AND AUTOMATION,

VEERMATA JIJABAI TECHNOLOGICAL INSTITUTE, MUMBAI

SEPTEMBER 2022

ACKNOWLEDGMENT

We would like to thank all the members of **SRA VJTI** for organizing the **Eklavya 2022** and allowing us to explore new domains and the learnings are unparalleled.

We are extremely grateful to our mentors

Krishna Narayanan and Himanshu Chougule

for their constant patience, motivation, enthusiasm, and immense knowledge, which they have shared with us throughout the duration of the project.

Our team:

Khushi Balia

khhbalia@gmail.com

Rajat Kaushik

rrkaushik_b21@et.vjti.ac.in

TABLE OF CONTENTS

1. PROJECT OVERVIEW

1.1 Objective of the project

Our project is to build a toy compiler that converts our custom language - PYLOX into an equivalent C code. Flex and bison are used as the lexer and parser generators, which form the basis of the compiler design.

Write a code in our language PYLOX and it emits a C code that can be compiled.

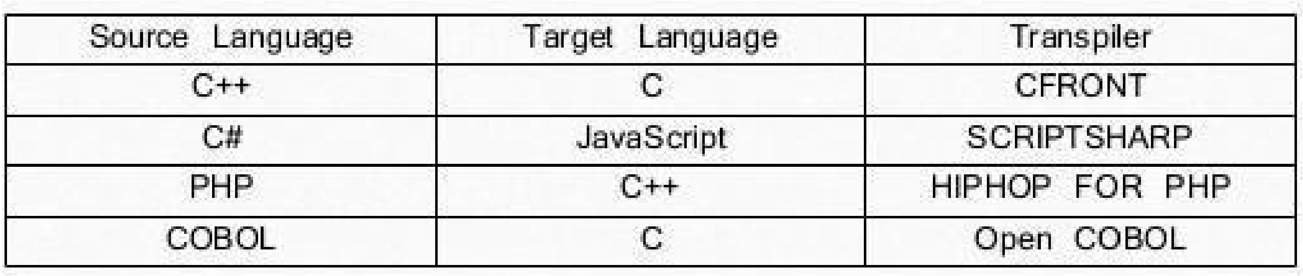
PYLOX code⟶C code

1.2 What is a transpiler?

A transpiler (also called source-to-source compiler) is a program that translates a source code from one language to another, at the same level of abstraction, that is, it converts a high-level language to another high-level language. The transpiler can either use an existing language C++ or C as input language or a new one like ours - Pylox which is transpiled to an existing language.

Transpilers bring in the convenience of instantly converting a code from one language to another and are thus beneficial.

Examples of some existing transpilers:

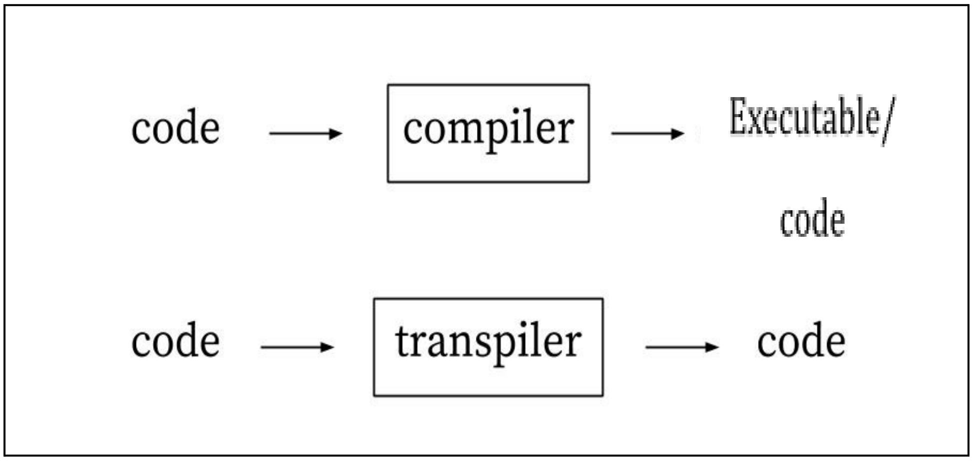


Source Language	Target Language	Transpiler
C++	C	CFRONT
C#	JavaScript	SCRIPTSHARP
PHP	C++	HIPHOP FOR PHP
COBOL	C	Open COBOL

1.3 Difference between a compiler and transpiler

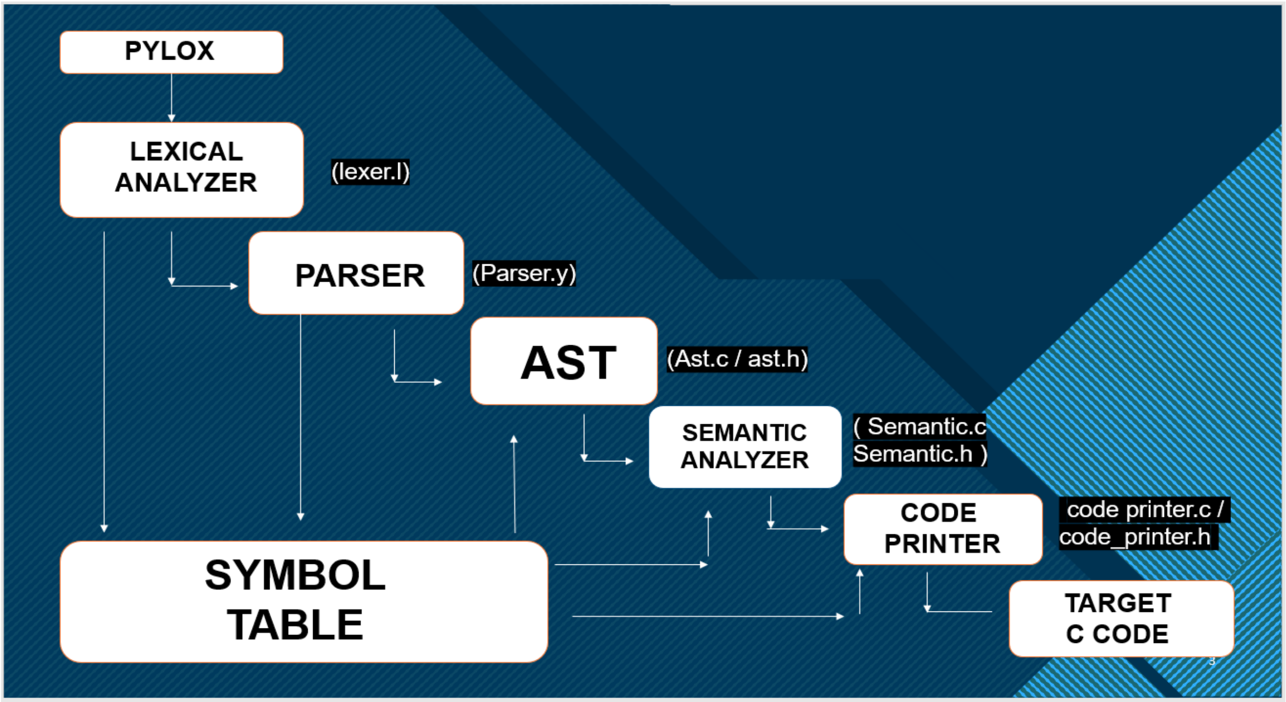
Compiler	**Transpiler**
It converts a source code written in a high-level language to an output code in	It converts a source code written in a high-level language to an output code in

a low-level language.	a different high-level language.
The source code has a higher level of abstraction than the output code.	The source code and the output code generated are of the same level of abstraction.
Output code is in assembly language and is readily executable after linking and decoding into machine language.	Output code is still in high-level programming language and requires a compiler to convert into low-abstraction assembly language.
In a compiler, the source code is scanned, parsed, transformed into an abstract syntax tree semantically analyzed, then converted into an intermediate code, and finally into the assembly language.	In a transpiler, the source code is parsed, and transformed into an abstract syntax tree, which is then converted to an intermediate model. This then transforms into an abstract syntax tree of the target language and code is generated.
Converting Java code into assembly language instructions is an example of compilation.	Converting Java code into C++ code is an example of transpilation.



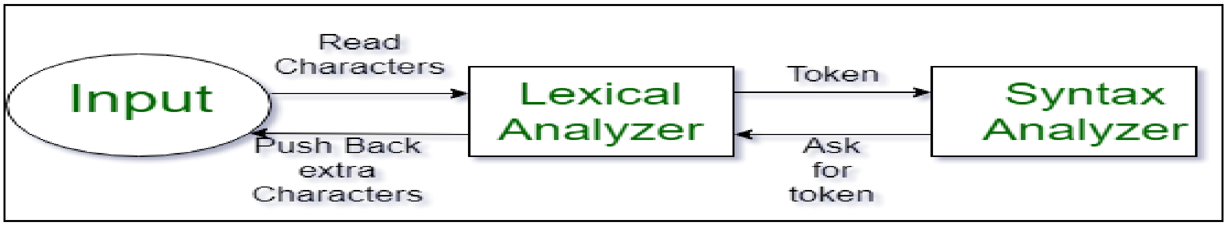
2. INTRODUCTION

2.1 Phases of a transpiler



i) Lexical analyser :

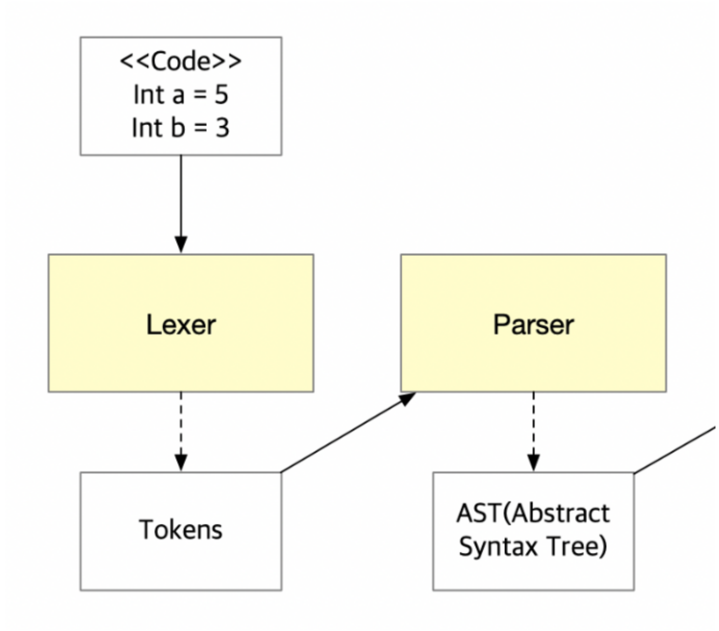
- Lexical analysis is the first phase of a compiler.It converts our input code, a sequence of characters (string) to lexemes.
- Lexemes are said to be a sequence of characters (alphanumeric) in a token.
- There are some predefined rules for every lexeme to be identified as a valid token.
- These lexemes pass through the lexer and it gives us tokens.
- In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuation symbols can be considered as tokens.
- If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer(parser).
- These tokens are then sent forward to use in parsing.
- Removal of white space(blanks, tab, new line, etc) and comments.



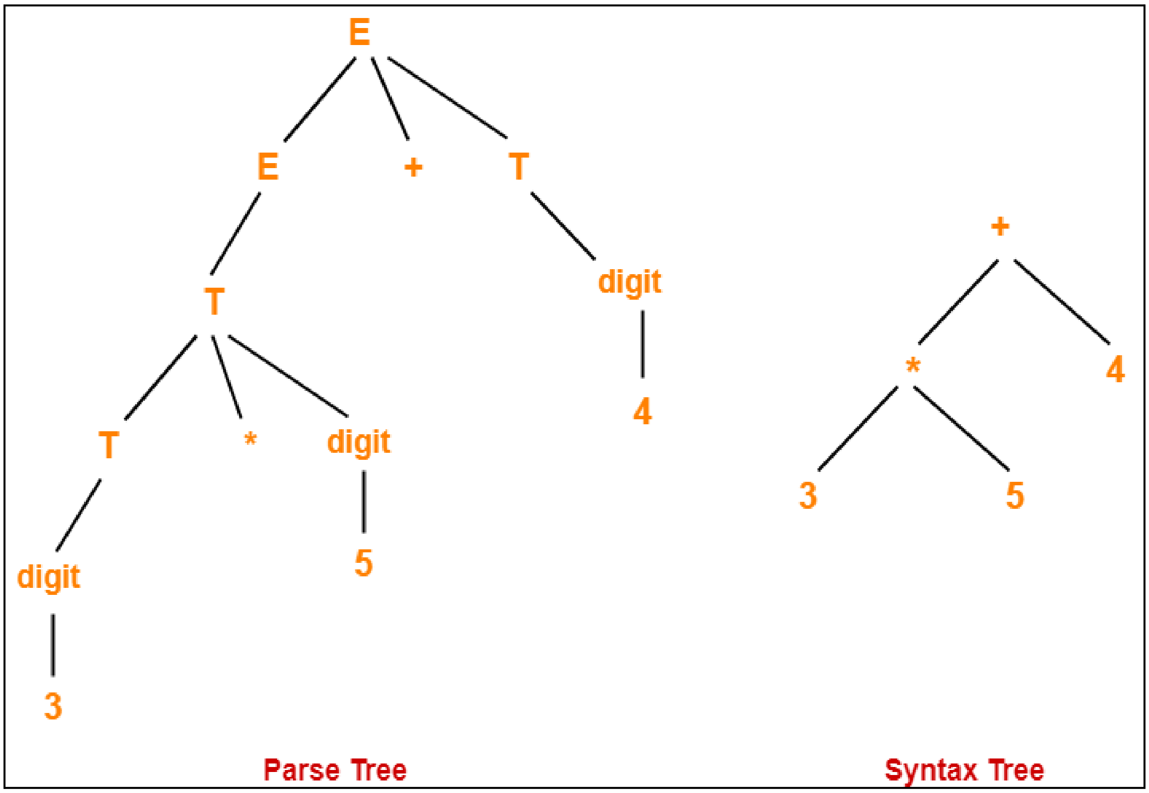
- We can conclude that the lexical analyzer phase scans the source code as a stream of characters and converts them into meaningful lexemes.

ii) Parsing :

- Syntax analysis or parsing is the second phase of a compiler.
- In this phase expressions formal grammar of the programming language is defined, statements, declarations, and so on, using the tokens received after lexical analysis.
- The parser accomplishes three tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.



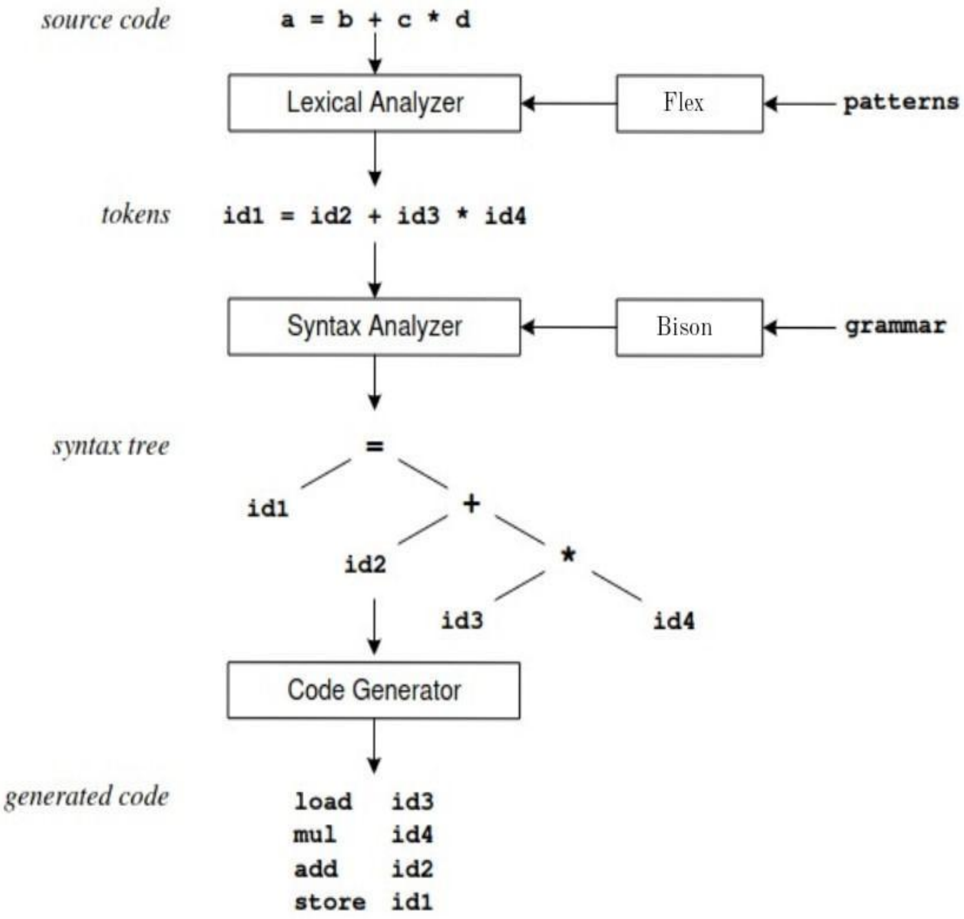
- The start symbol of the derivation becomes the root of the parse tree.
- In a parse tree:
    1. All leaf nodes are terminals.
    2. All interior nodes are non-terminals.



- Parser takes the tokens produced by lexical analysis as input and generates a parse tree.
- The parser checks if the expression made by the tokens is syntactically correct.

iii) Code printer :

- The final phase is code generation(code printer).
- It traverses through each semantic node and produces an equivalent target program (C code in this case) as output.



- The expression a = b + c * d is given as the input to the lexer generated by flex, this expression is broken down into tokens id1, =, id2, +, id3, *, id4 where id stands for the identifier. This was the lexical analysis phase.
- These tokens are then passed through the parser generated by bison, which defines the grammar and forms the AST using the tokens as seen in the figure above. This is the syntax analysis phase.
- The code generator traverses through every node of the AST, and converts it to the required output, in our case it would be a C code.

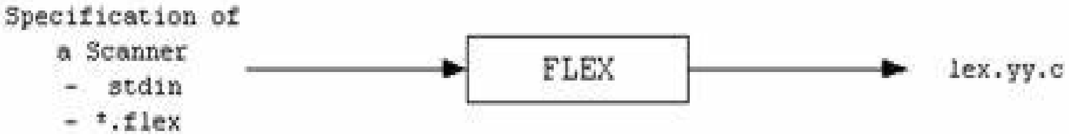
2.2 Flex and Bison

Introduction

- Flex and Bison will generate a parser that is virtually guaranteed to be faster than anything you could write manually in a reasonable amount of time.
- Updating and fixing Flex and Bison source files is a lot easier than updating and fixing custom parser code.
- Flex and Bison have mechanisms for error handling and recovery, which is something you don't want to try to bolt onto a custom parser.

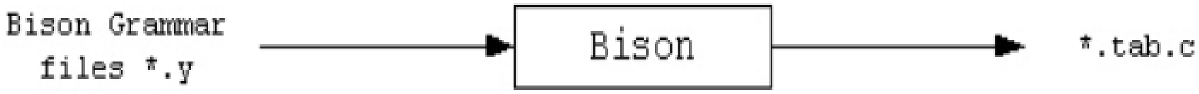
i) Flex

- Fast lexical analyzer generator (flex) is a program for generating lexical analyzers(lexers/scanners).
- **yylex()** is automatically generated by the flex when it is provided with a file extension of .l and this yylex()function is expected by the parser to call to collect tokens from token streams.



ii) Bison

- Bison is a parser generator, it warns about any parsing ambiguities and generates a parser that reads sequences of tokens.
- It is a grammar specification file with a .y extension.
- **yyparse()** returns a value of 0 if the input it parses is valid according to the given grammar rules. It returns a 1 if the input is incorrect and error recovery is impossible.

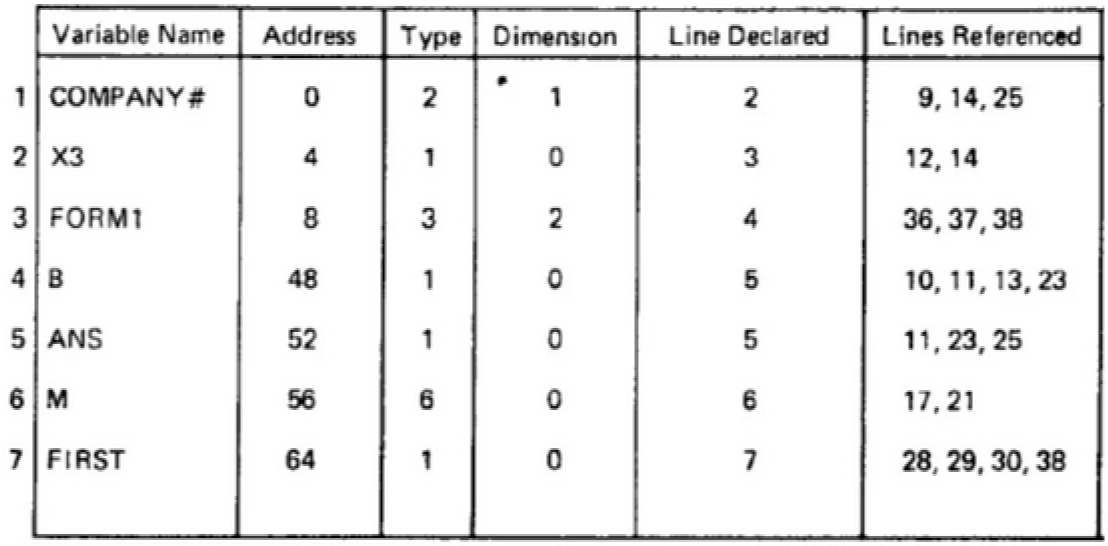


2.3 Symbol table

Introduction

- Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc.
- Function of symbol table are as follow :-
  1. To store the names of all entities in a structured form at one place.
  2. To verify if a variable has been declared.
  3. To implement type checking, by verifying assignments and expressions in the source code.
- Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

Symbol table



	Variable Name	Address	Type	Dimension	Line Declared	Lines Referenced
1	COMPANY#	0	2	1	2	9, 14, 25
2	X3	4	1	0	3	12, 14
3	FORM1	8	3	2	4	36, 37, 38
4	B	48	1	0	5	10, 11, 13, 23
5	ANS	52	1	0	5	11, 23, 25
6	M	56	6	0	6	17, 21
7	FIRST	64	1	0	7	28, 29, 30, 38

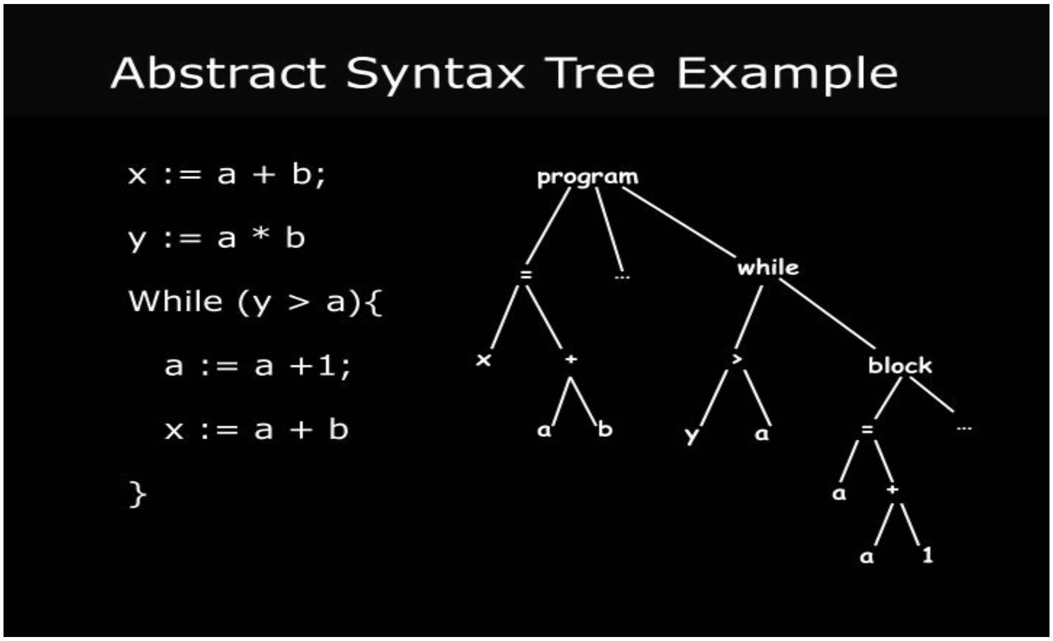
Typical view of a symbol table.

Usage of symbol table by various phase of compiler

- **Lexical Analysis:** Creates new entries in the table about tokens.
- **Parser/Syntax Analysis:-** Adds information regarding attribute type,scope,dimension,line of reference , etc in the table.
- **Code printer:-** Generates code by using address information of identifier present in the table.

2.4 Abstract Syntax tree

- Abstract syntax tree(AST) is a compact version of the parse tree.
- AST is a tree data structure that stores various tokens as its nodes, such that it can abstractly represent the code in memory, and each node of the tree denotes a construct occurring in the source code.
- The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details.



Abstract Syntax Tree Example

```
x := a + b;
y := a * b
While (y > a){
  a := a +1;
  x := a + b
}
```

2.4 Third-Party Apps

i) Klib :

- Klib is a standalone and lightweight C library. Most components are independent of external libraries, except the standard C library and independent of each other.
- Klib strives for **efficiency and a small memory footprint**
- To use a component of this library, you only need to copy a couple of files to your source code tree without worrying about library dependencies
- macros and the C preprocessor play a key role in klib. Klib is fast partly because the compiler knows the key-value type at the compile time and can optimize the code to the same level as type-specific code.
- We are only using the khash.h component.

Khash.h

- □ Its function is to generate a hash table with open addressing.
- □ It doesn't have any dependencies.
- □ A hash table uses a hash function to compute an *index*, also called a *hash code*.

ii) Vec :

- A type-safe dynamic array implementation for C.

- Installation can be done by dropping the vec. c and vec. h files into an existing C project and compiled along with it.

- All vector functions are macro functions. The parameter “v “ in each function should be a pointer to the vec struct on which the operation is to be performed on.

- Functions used:-

  1. **vec_init(v)**:- Initializes the vector, this must be called before the vector can be used.

  2. **vec_foreach(v, var, iter**):- Iterates the values of the vector from the first to the last. var should be a variable of the vector's contained type where the value will be stored with each iteration. iter should be an int used to store the index during iteration.

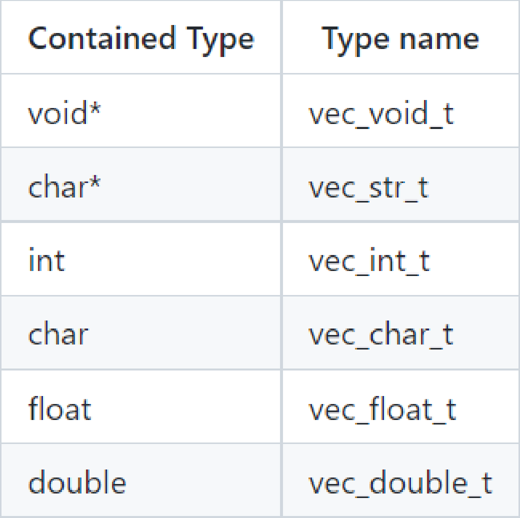
  3. **vec_pusharr(v, arr, count):-** Pushes the contents of the array arr to the end of the vector. The count should be the number of elements in the array.

- Types:-

  To define a new vector type the vec_t()

```
typedef vec_t(FILE*) fp_vec_t;
```

Vec. h provides the following predefined vector types:



Contained Type	Type name
void*	vec_void_t
char*	vec_str_t
int	vec_int_t
char	vec_char_t
float	vec_float_t
double	vec_double_t

3. CUSTOM LANGUAGE - PYLOX

Inspired by Python + Lox

Rules:-

1. **There is no need to use a semicolon at the end of any Pylox statement.**

2.

1) Variables:

Data Types: - int, float, bool, char

- bool has two values - true and false

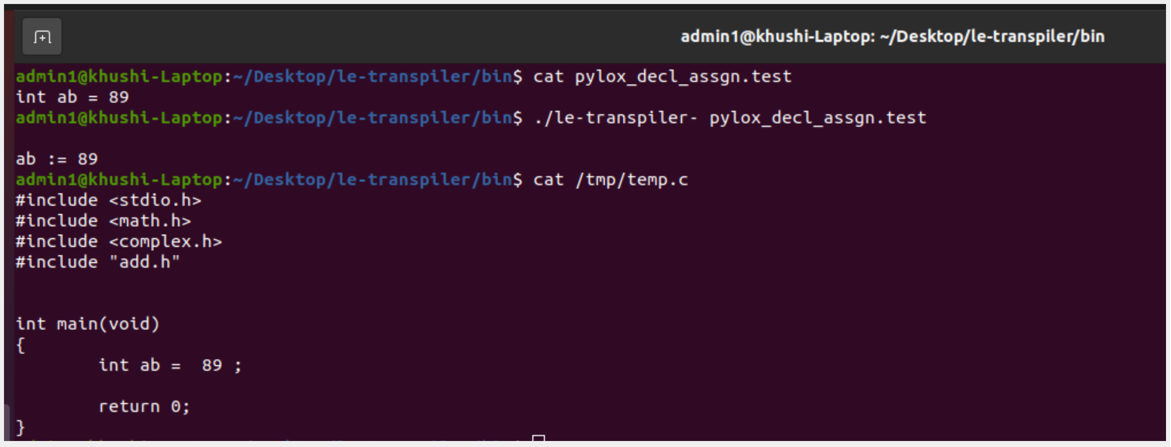
Syntax:

Declaration:

```
data_type var_name
```

Declaration and Assignment:

```
data_type var_name = value
```



```
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat pylox_decl_assgn.test
int ab = 89
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ ./le-transpiler- pylox_decl_assgn.test

ab := 89
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat /tmp/temp.c
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include "add.h"


int main(void)
{
        int ab =  89 ;

        return 0;
}
```

2) Operators:

{,} - Braces

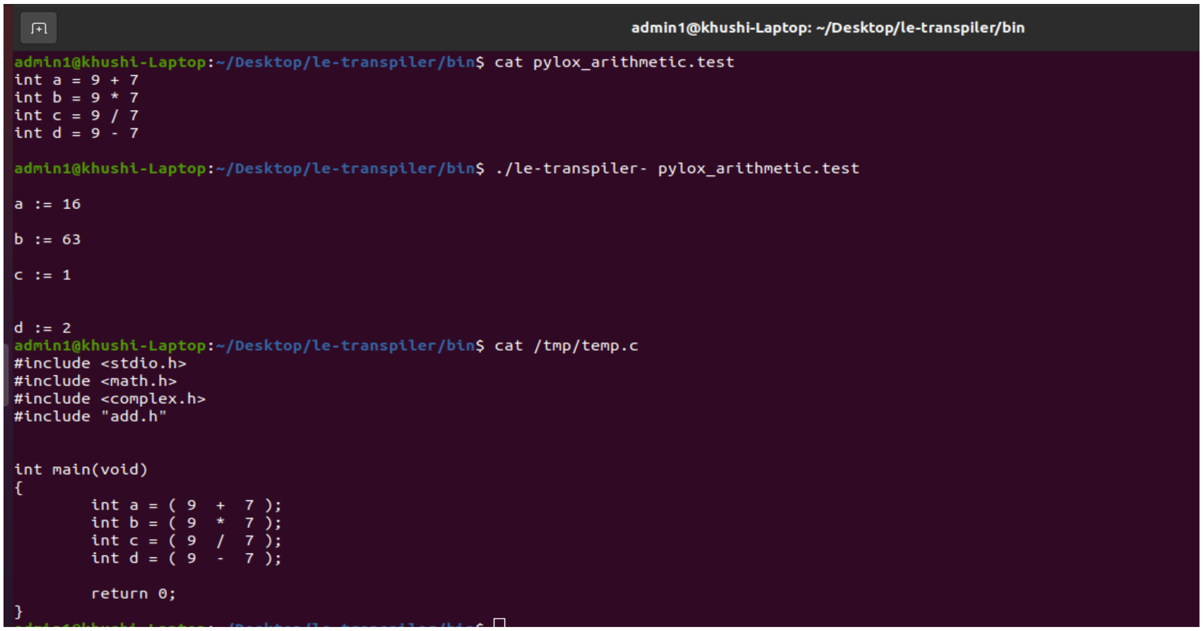
(,) - Parenthesis

/,*,+,-,% - Arithmetic operators

>,<,==,!=,>=,<= - Comparison operators

not, and, or - Logical operators: not, and, or4

= - Assignment operator



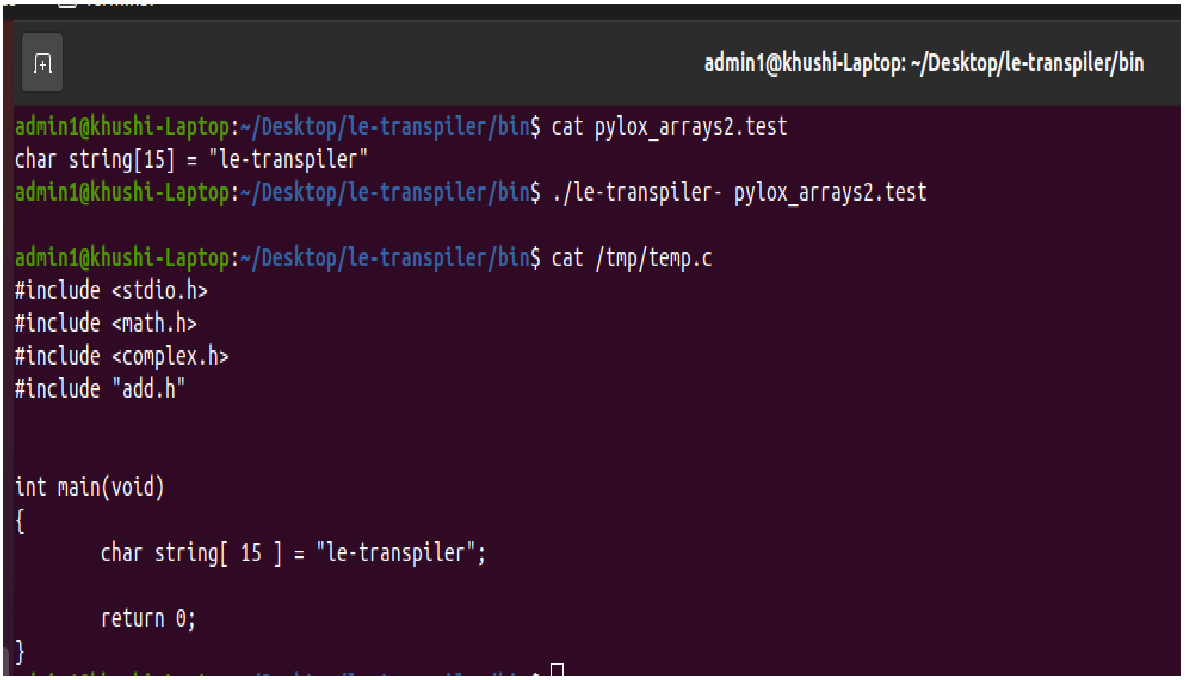
```
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat pylox_arithmetic.test
int a = 9 + 7
int b = 9 * 7
int c = 9 / 7
int d = 9 - 7

admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ ./le-transpiler- pylox_arithmetic.test

a := 16

b := 63

c := 1


d := 2
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat /tmp/temp.c
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include "add.h"


int main(void)
{
        int a = ( 9  +  7 );
        int b = ( 9  *  7 );
        int c = ( 9  /  7 );
        int d = ( 9  -  7 );

        return 0;
}
```

3) Arrays:

- Arrays are zero-indexed
- can be used with bool, int, and char.

Syntax:

```
data_type array_name[size]
```



```
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat pylox_arrays2.test
char string[15] = "le-transpiler"
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ ./le-transpiler- pylox_arrays2.test

admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat /tmp/temp.c
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include "add.h"


int main(void)
{
        char string[ 15 ] = "le-transpiler";

        return 0;
}
```

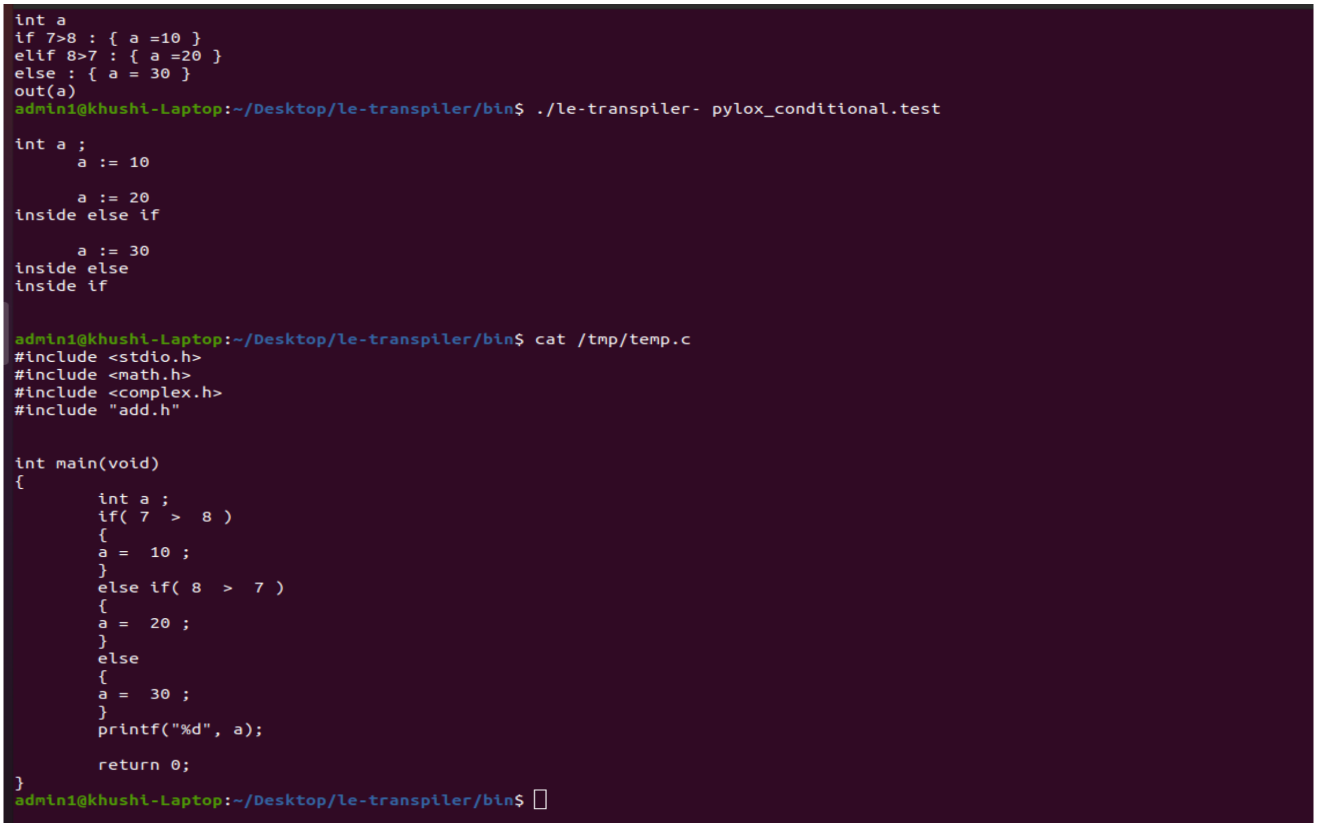
4) If -elif - else statement:

- Conditional statement that executes a code depending on whether the specified condition is true or false.

Syntax:

```
if boolean_expression : {
  statement 1
  ...
}
```

```
elif Boolean expression: {
   statement 2
   ...
}
else: {
   statement 3
   ...}
```



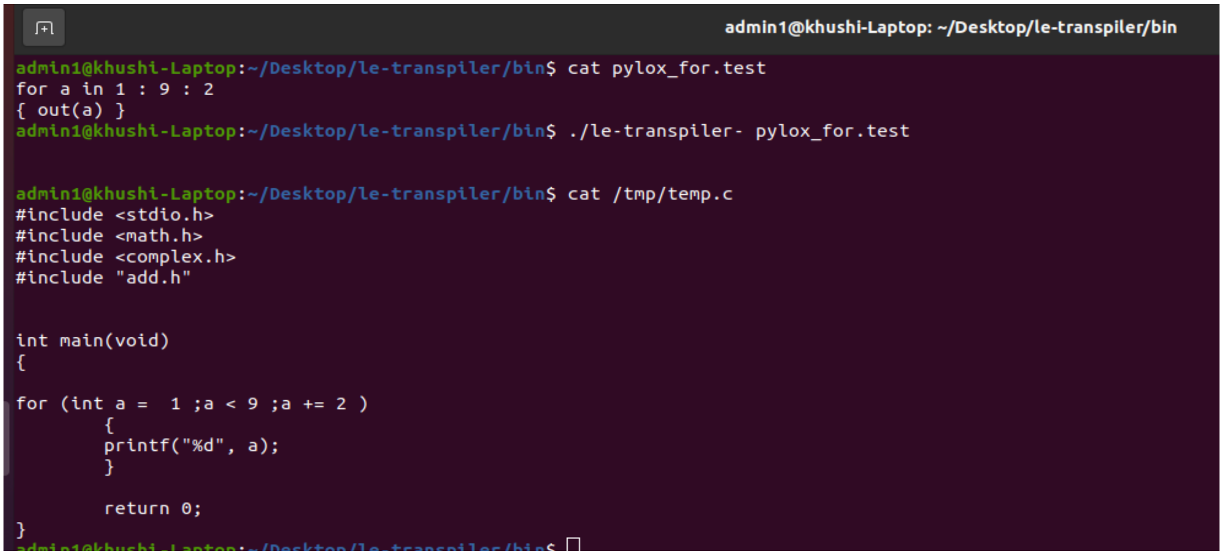
```
int a
if 7>8 : { a =10 }
elif 8>7 : { a =20 }
else : { a = 30 }
out(a)
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ ./le-transpiler- pylox_conditional.test

int a ;
      a := 10

      a := 20
inside else if

      a := 30
inside else
inside if


admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat /tmp/temp.c
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include "add.h"


int main(void)
{
        int a ;
        if( 7  >  8 )
        {
        a =  10 ;
        }
        else if( 8  >  7 )
        {
        a =  20 ;
        }
        else
        {
        a =  30 ;
        }
        printf("%d", a);

        return 0;
}
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ 
```

5) For loop:

- Value of any integer variable considered, will vary from start to stop - 1.

Syntax:

```
for var in start: stop: increment {
    statement 1
     ....
}
```



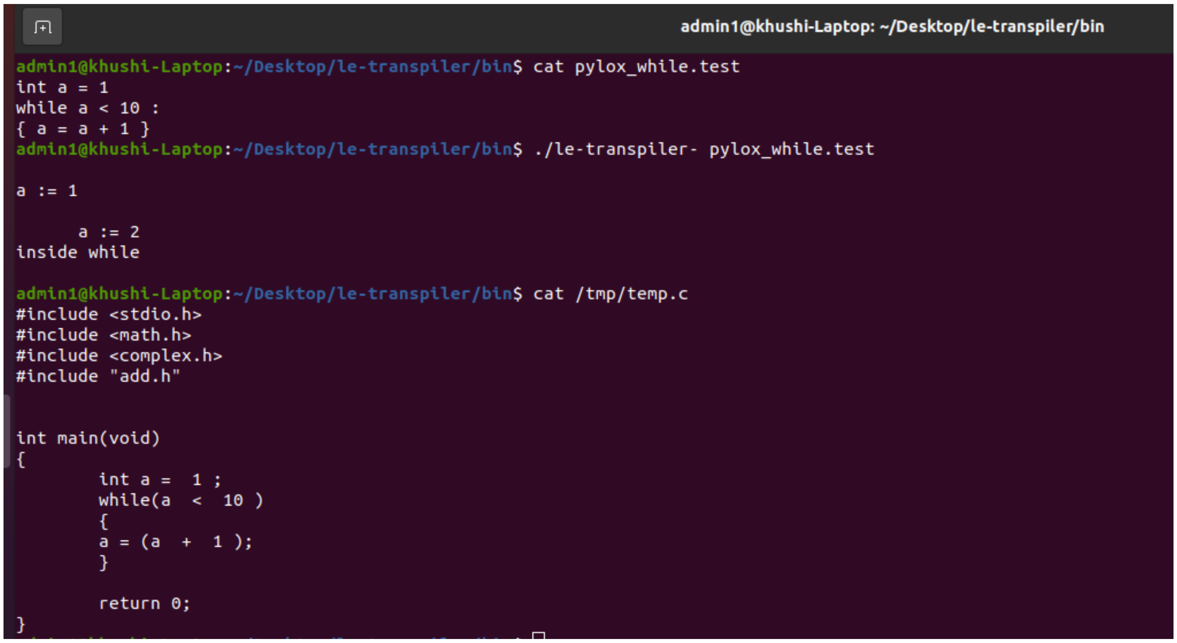
```
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat pylox_for.test
for a in 1 : 9 : 2
{ out(a) }
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ ./le-transpiler- pylox_for.test


admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat /tmp/temp.c
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include "add.h"


int main(void)
{

for (int a =  1 ;a < 9 ;a += 2 )
        {
        printf("%d", a);
        }

        return 0;
}
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$
```

6) While loop:

- repeatedly executes a target statement as long as a given condition is true.

Syntax:

```
while Boolean expression: {
    statement 1

    ...

}
```



```
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat pylox_while.test
int a = 1
while a < 10 :
{ a = a + 1 }
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ ./le-transpiler- pylox_while.test

a := 1

      a := 2
inside while

admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat /tmp/temp.c
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include "add.h"


int main(void)
{
        int a =  1 ;
        while(a  <  10 )
        {
        a = (a  +  1 );
        }

        return 0;
}
```

7) Control Statements:

- **break** is used to break execution in a loop statement, either for loop or while loop. It exits the loop upon calling.
- **continue** is used to continue execution in a loop statement, either for loop or while loop.

8) Functions:

- The keyword func is used.
- The function must be defined before calling it

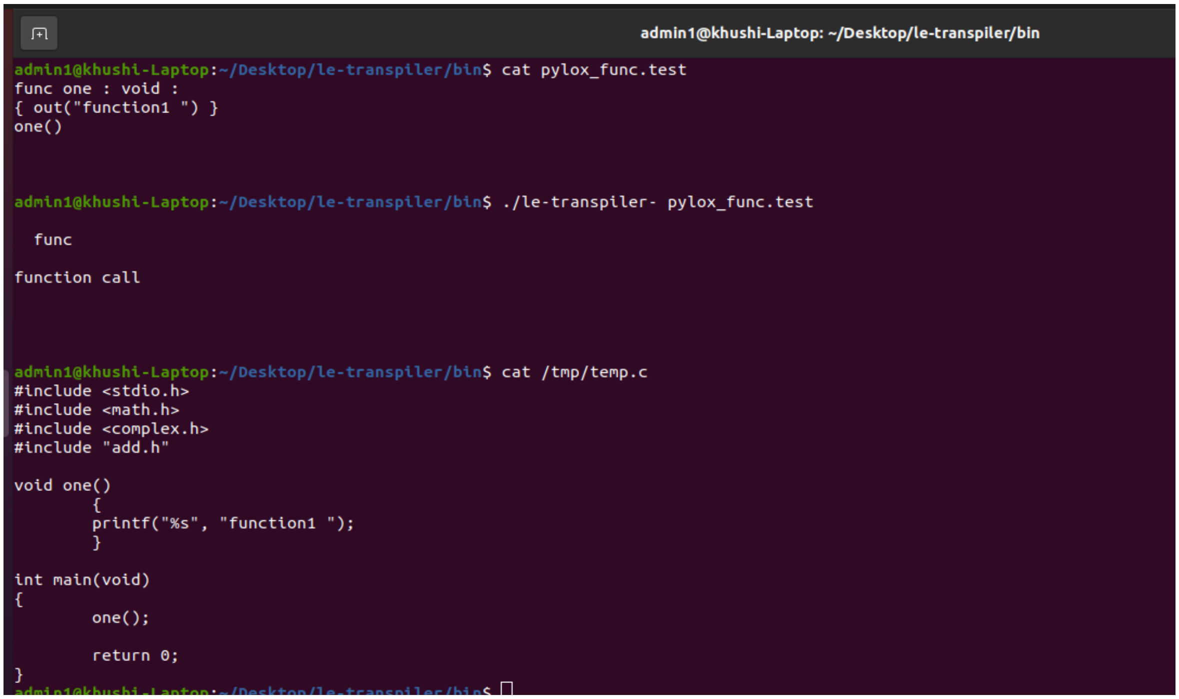
Syntax:

Function definition:

```
func <function name>: <return type>: <datatype> <Param name>, datatype> <Param name>. {
  statement_1
  ...
  return <variable/value>
}
```

Function call:

```
function name (var1, var2,)
```



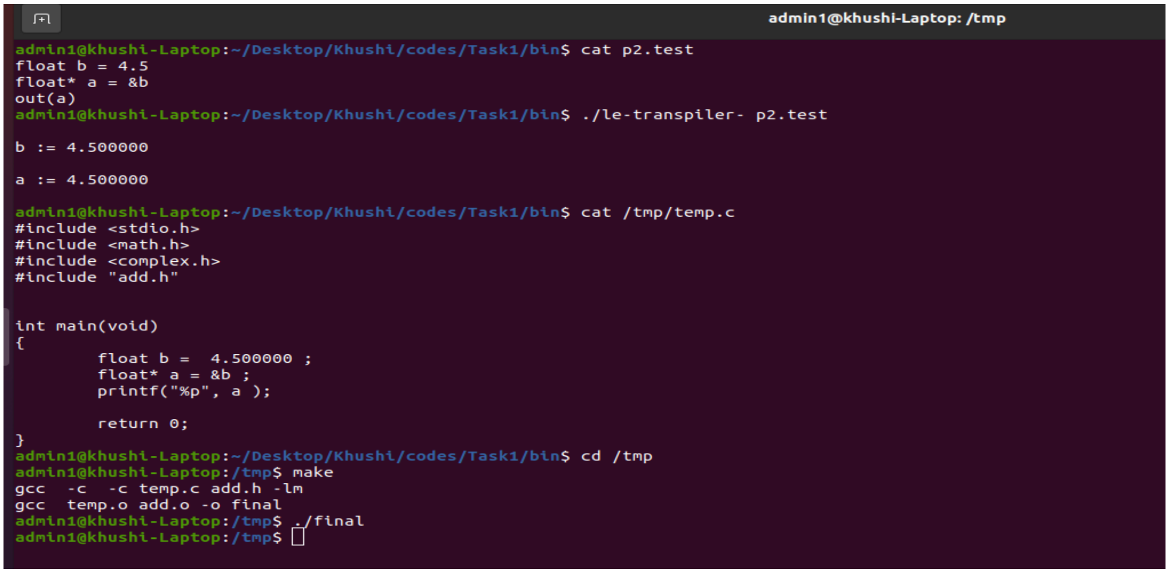
```
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat pylox_func.test
func one : void :
{ out("function1 ") }
one()



admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ ./le-transpiler- pylox_func.test

  func

function call




admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat /tmp/temp.c
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include "add.h"

void one()
        {
        printf("%s", "function1 ");
        }

int main(void)
{
        one();

        return 0;
}
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$
```

9) Pointers

- Pointers store addresses of variables or a memory location.

Syntax:

datatype* variable name = &variable name



```
admin1@khushi-Laptop:~/Desktop/Khushi/codes/Task1/bin$ cat p2.test
float b = 4.5
float* a = &b
out(a)
admin1@khushi-Laptop:~/Desktop/Khushi/codes/Task1/bin$ ./le-transpiler- p2.test

b := 4.500000

a := 4.500000

admin1@khushi-Laptop:~/Desktop/Khushi/codes/Task1/bin$ cat /tmp/temp.c
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include "add.h"


int main(void)
{
        float b =  4.500000 ;
        float* a = &b ;
        printf("%p", a );

        return 0;
}
admin1@khushi-Laptop:~/Desktop/Khushi/codes/Task1/bin$ cd /tmp
admin1@khushi-Laptop:/tmp$ make
gcc  -c  -c temp.c add.h -lm
gcc  temp.o add.o -o final
admin1@khushi-Laptop:/tmp$ ./final
admin1@khushi-Laptop:/tmp$ 
```

- To include the functions of the math. the h header file, the syntax of the various functions is as follows

I) To find the log:

float variable name = log (float/integer value)

ii) To find sin:

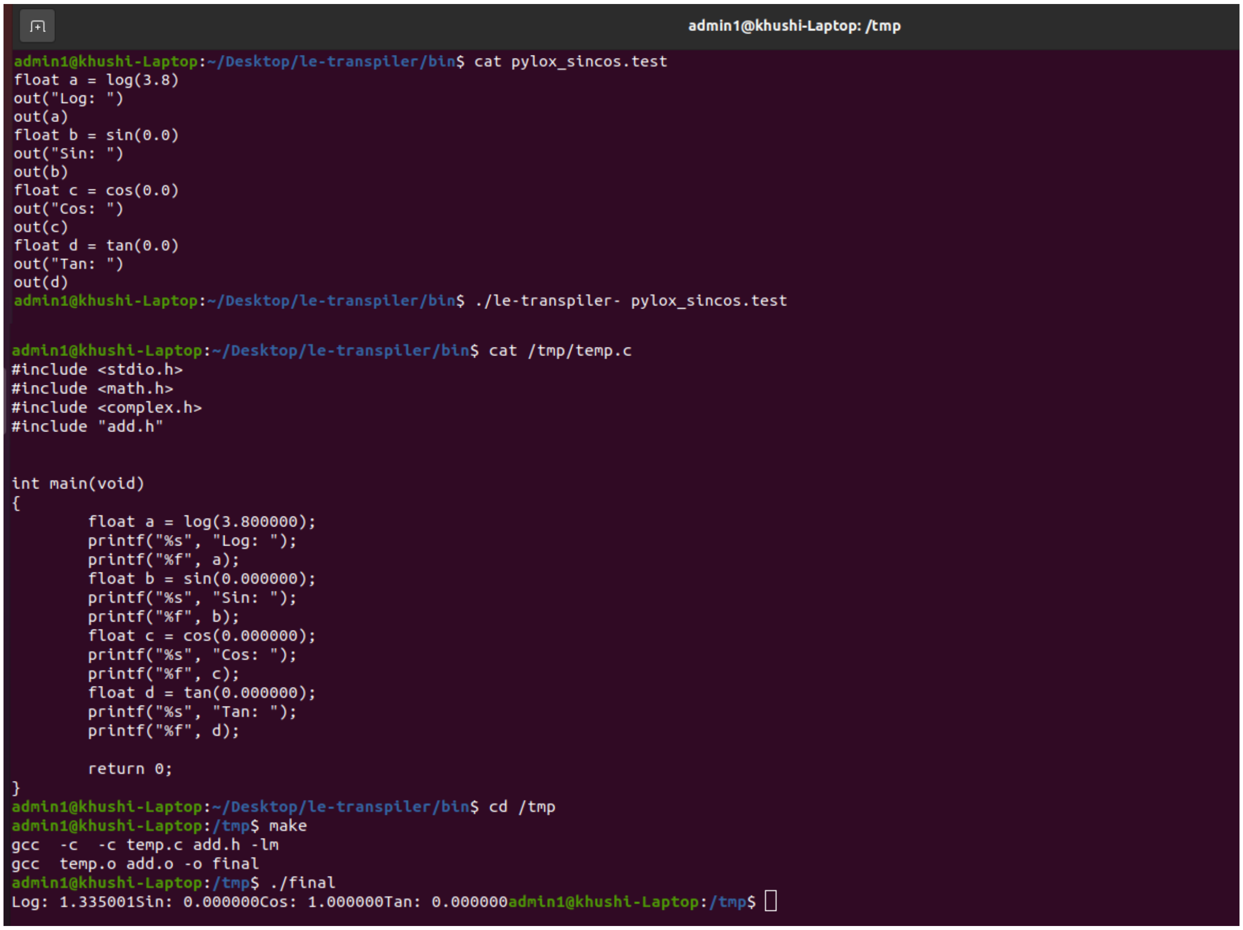
float variable name = sin (float/integer value)

iii) To find cos:

float variable name = cos (float/integer value)

iv) To find tan:

float variable name = tan (float/integer value)

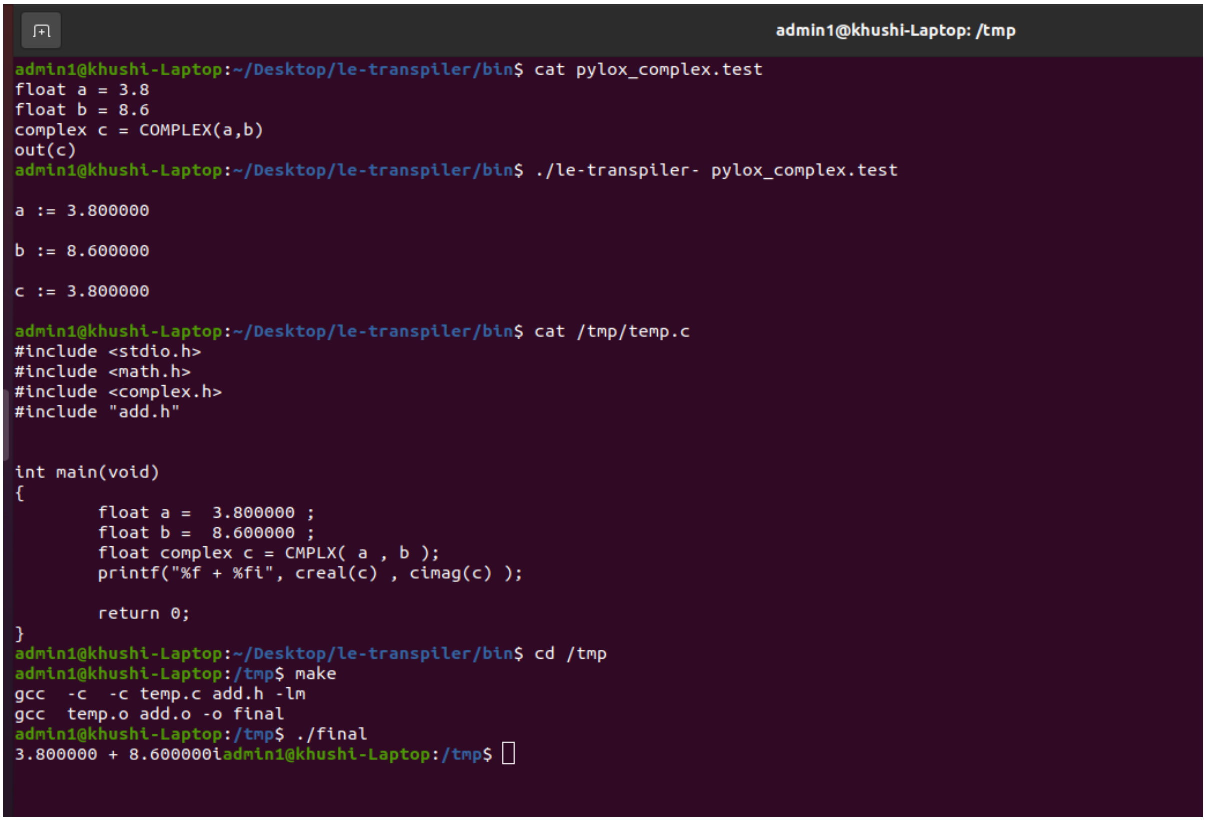


```
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat pylox_sincos.test
float a = log(3.8)
out("Log: ")
out(a)
float b = sin(0.0)
out("Sin: ")
out(b)
float c = cos(0.0)
out("Cos: ")
out(c)
float d = tan(0.0)
out("Tan: ")
out(d)
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ ./le-transpiler- pylox_sincos.test

admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat /tmp/temp.c
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include "add.h"


int main(void)
{
        float a = log(3.800000);
        printf("%s", "Log: ");
        printf("%f", a);
        float b = sin(0.000000);
        printf("%s", "Sin: ");
        printf("%f", b);
        float c = cos(0.000000);
        printf("%s", "Cos: ");
        printf("%f", c);
        float d = tan(0.000000);
        printf("%s", "Tan: ");
        printf("%f", d);

        return 0;
}
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cd /tmp
admin1@khushi-Laptop:/tmp$ make
gcc  -c  -c temp.c add.h -lm
gcc  temp.o add.o -o final
admin1@khushi-Laptop:/tmp$ ./final
Log: 1.335001Sin: 0.000000Cos: 1.000000Tan: 0.000000admin1@khushi-Laptop:/tmp$ 
```

- To add the function of the complex. the h header file, the syntax of the function is as follows:

Syntax:

complex variable name = COMPLEX (variable name, variable name)



```
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat pylox_complex.test
float a = 3.8
float b = 8.6
complex c = COMPLEX(a,b)
out(c)
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ ./le-transpiler- pylox_complex.test

a := 3.800000

b := 8.600000

c := 3.800000

admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cat /tmp/temp.c
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include "add.h"


int main(void)
{
        float a =  3.800000 ;
        float b =  8.600000 ;
        float complex c = CMPLX( a , b );
        printf("%f + %fi", creal(c) , cimag(c) );

        return 0;
}
admin1@khushi-Laptop:~/Desktop/le-transpiler/bin$ cd /tmp
admin1@khushi-Laptop:/tmp$ make
gcc  -c  -c temp.c add.h -lm
gcc  temp.o add.o -o final
admin1@khushi-Laptop:/tmp$ ./final
3.800000 + 8.600000iadmin1@khushi-Laptop:/tmp$ 
```

4. IMPLEMENTATION

4.1 File structure

Lets look at the Eklavya - - le-transpiler package:

```
📦le-transpiler
| ├📦build
| ├📂CMakeFiles
| ├📂CMakeTmp
| | ├📂le-transpiler-.dir
|  |  ├📂src
| | | ├📜ast.c.o
| | | ├📜code_printer.c.o
| | | ├📜lexer.c.o
| | | ├📜main1.c.o
| | | ├📜parser.c.o
| | | ├📜semantic.c.o
| | | └📜symbol_table.c.o
|  |  ├📦third-party
| | | └📂vec
| | | |  └📜vec.c.o
| | ├📜CMakeCache.txt
| | ├📜Makefile
| | └📜cmake_install.cmake
| ├ 📦include
| | ├📜ast.h
| | ├📜code_printer.h
| | ├📜parser.h
| | ├📜semantic.h
| | └📜symbol_table.h
|  ├📦src
| | ├📜ast.c
| | ├📜code_printer.c
| | ├📜lexer.c
| | ├📜lexer.l
| | ├📜main1.c
| | ├📜parser.c
| | ├📜parser.h
| | ├📜parser.y
| | ├📜semantic.c
| | └📜symbol_table.c
|  ├ ⓂCMakelist.txt
```

1.build:- CMake can generate a native build environment that will compile source code, create libraries, generate wrappers and build executables in arbitrary combinations

2.src :- Contains all the necessary sources files for transpiling the programs.

3.include:- Contains all the necessary header files.

4.**third-party:-** Contains the third party app source code files.

5. **CMakelist.txt:-** Used to link all these files according to their dependencies when the package is built.

4.2 Understanding flex and bison, coding lexer. l and parser. y

(I) Program structure of lexical analysis [lexer. l] (flex)

There are 3 sections:

1. **Definition section:**

   This section contains the declaration of regular definitions, variables, constants, etc. In the definition section, the text is enclosed in % {.....%} brackets.

   Anything written in these brackets is copied directly to the file lex.yy.c

   Syntax:

   ```
   % {// Definitions   %}
   ```

2. **Rules section:**

   This section contains a series of rules, which are enclosed in %% …. %%

   Syntax:

   ```
   %% pattern action %%
   ```

3. **Code section:**

This section contains C statements and additional functions. We can also compile these functions separately and load them with the lexical analyzer.

Basic Program Structure:

```
%{
// Definitions
%}

%%
Rules
%%

User code section
```

- The flex specification will be compiled by the command

```
flex   <filename>. l
```

(II) Program structure of parser [parser. y] (bison)

Syntax:

```
%{
C declarations
%}
Bison declarations like tokens, type, operator precedence
%%
Grammar rules
```

```
%%
Additional C code
```

Example:

```
%{
#include <stdio.h>
Int yylex(void); /* type of yylex () */
Void yyerror (char const *s);
#define YYDEBUG 1
%}
%union {/* type of ‘yylval’ (value stack stype) */
int integer;
Float real;
}

%token NAME NUMBER
%type <real> exp        /* nonterminal and its type */
%left
%start s                /* start symbol */

%%
statement: NAME '=' expression
```

```
| expression {print ("= %d\n", $1);}
;
expression: expression '+' NUMBER {$$ = $1 + $3;}
| expression '-' NUMBER {$$ = $1 - $3;}
| NUMBER {$$ = $1;}
;
%%
int main(void)
{
yyparse ();
return 0;
}
Void yyerror(char const *s)
```

- The bison specification will be compiled by the command
  `bison  -d  -t  <filename>.`

4.3 Milestone 1 - Connecting all the files for a single part of the program (Declaration and assignment)

- Connecting all the files shown above in the file structure using CMake is the most integral part of the project. So, starting with a small segment of the code, and transpile it to C code, is a wise decision, which makes it easy to debug the errors.
- We decided to link all the files just for the declaration and the assignment part, and after all the files got linked successfully, then we added the other parts of the code one by one, ensuring each of them got transpiled without any errors.

4.4 Cmake - an integral part of our project

CMake is a meta-build system that uses scripts called CMakeLists to generate build files for a specific environment. CMake is a cross-platform free and open-source software for building automation and testing software. CMake is used to control the software compilation process using a simple platform and generate native makefiles, and workspaces that can be used in the compiler environment.

CMake commands

Cmake commands are case-insensitive, some commonly used commands-

- message: prints given message
- cmake_minimum_required: sets the minimum version of CMake to be used
- add executable: adds executable target with the given name
- add library: adds a library target to be built from listed source files
- add subdirectory: adds a subdirectory to build.

5. CONCLUSION AND FUTURE WORK

5.1 Conclusion

- To conclude, the aim was to transpile a custom language (Pylox language) to C code. We were successful in transpiling and adding the followings:-

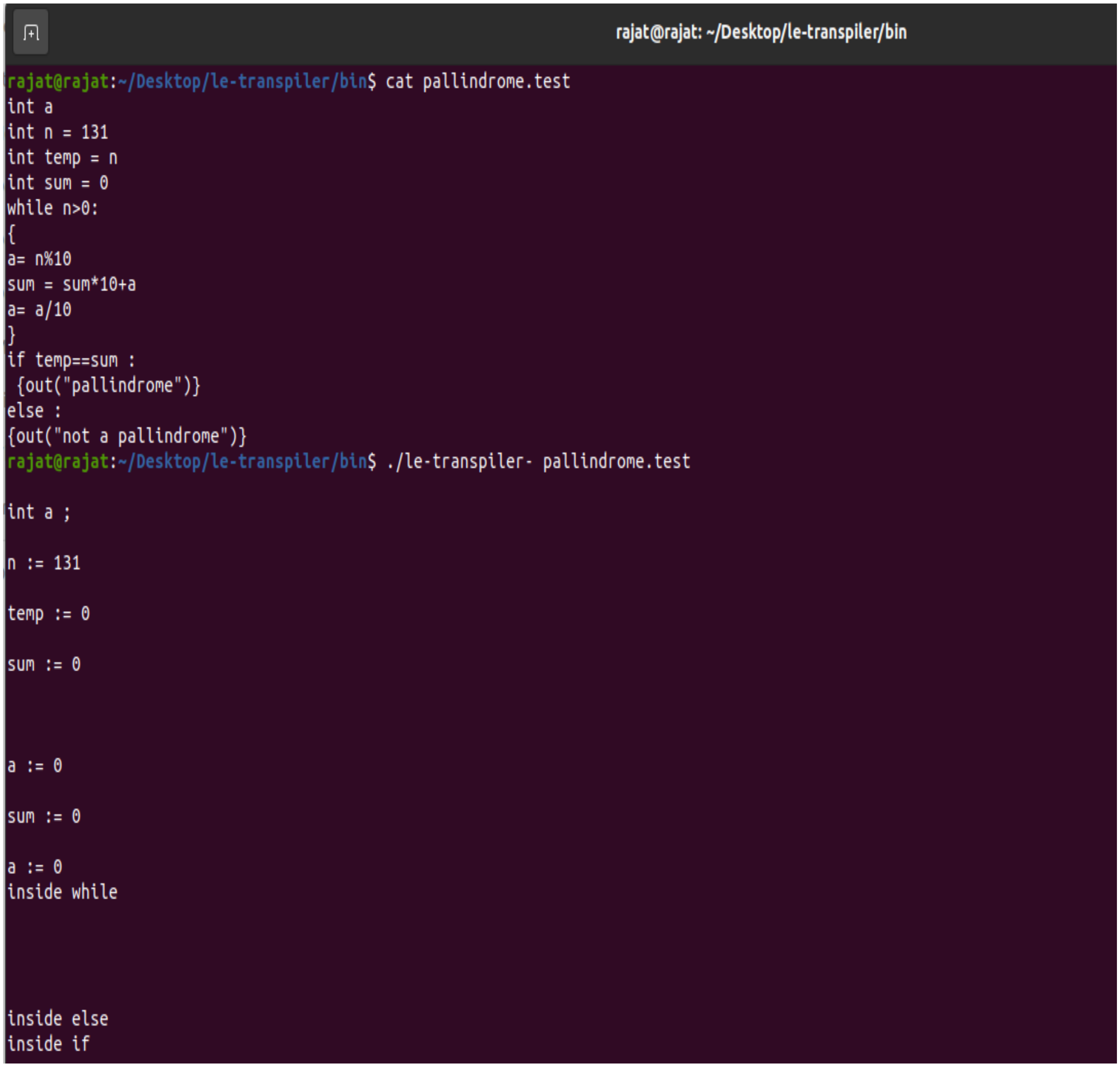
  a) Declaration statement
  b) Assignment statement
  c) Array declaration
  d) Array declaration assignment
  e) If, elif, else condition
  f) For and while loop
  g) Print statement
  h) Logical operators
  i) Arithmetic operators
  j) Relational operators
  k) Functions
  l) Math functions
  m) Complex function

- We have not only learned how to make our syntax language, but we have also learned how to transpile them into another language and compile the file generated

• Output

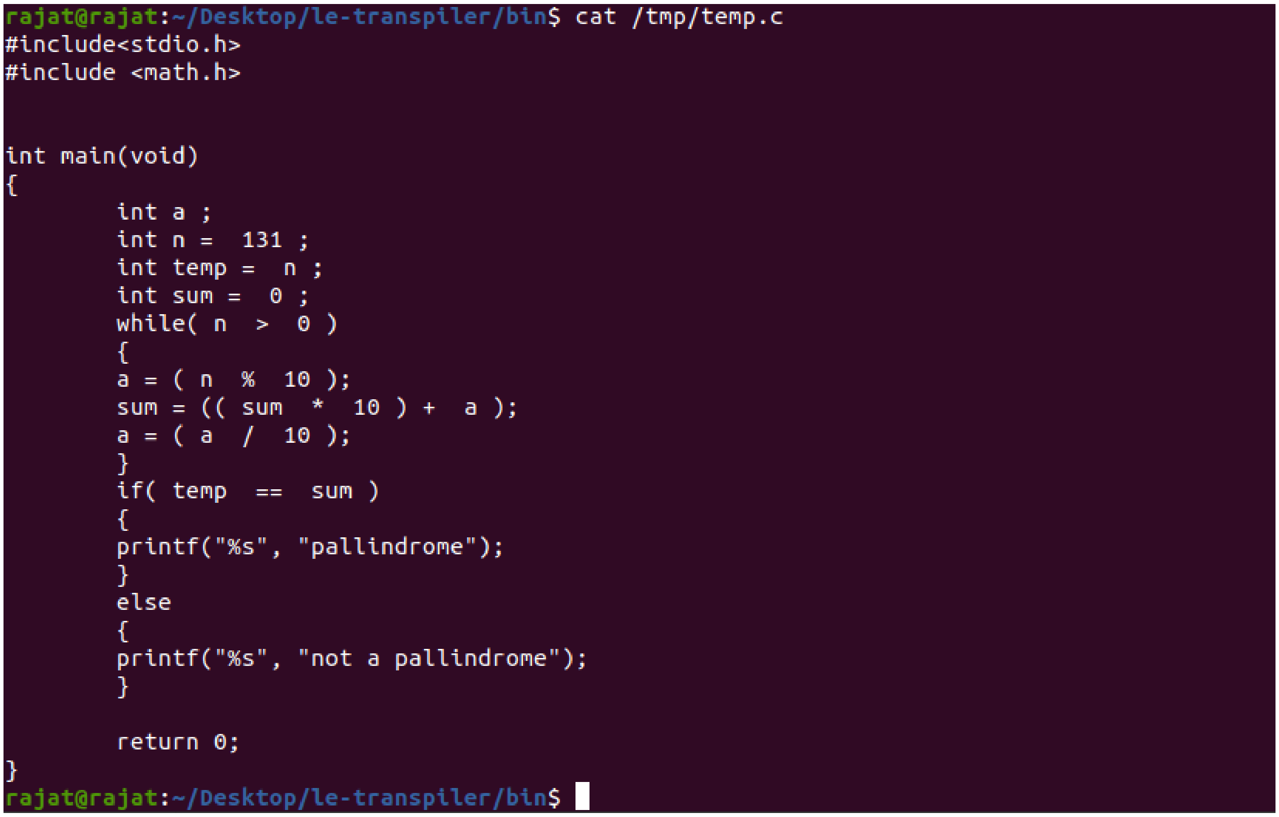
Transpiling and running the palindrome program in Pylox syntax to C:-

Here we wrote our palindrome program in a text editor and then transpiled this program to C .



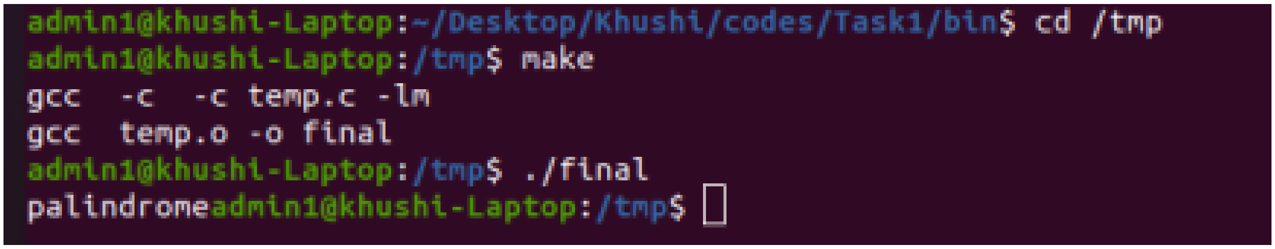
```
rajat@rajat:~/Desktop/le-transpiler/bin$ cat pallindrome.test
int a
int n = 131
int temp = n
int sum = 0
while n>0:
{
a= n%10
sum = sum*10+a
a= a/10
}
if temp==sum :
 {out("pallindrome")}
else :
{out("not a pallindrome")}
rajat@rajat:~/Desktop/le-transpiler/bin$ ./le-transpiler- pallindrome.test

int a ;

n := 131

temp := 0

sum := 0



a := 0

sum := 0

a := 0
inside while




inside else
inside if
```

This is how it looks after getting transpiled into generated C code



```
rajat@rajat:~/Desktop/le-transpiler/bin$ cat /tmp/temp.c
#include<stdio.h>
#include <math.h>


int main(void)
{
        int a ;
        int n =  131 ;
        int temp =  n ;
        int sum =  0 ;
        while( n  >  0 )
        {
        a = ( n  %  10 );
        sum = (( sum  *  10 ) +  a );
        a = ( a  /  10 );
        }
        if( temp  ==  sum )
        {
        printf("%s", "pallindrome");
        }
        else
        {
        printf("%s", "not a pallindrome");
        }

        return 0;
}
rajat@rajat:~/Desktop/le-transpiler/bin$ 
```

Running the program in C and getting the output as required :-



```
admin1@khushi-Laptop:~/Desktop/Khushi/codes/Task1/bin$ cd /tmp
admin1@khushi-Laptop:/tmp$ make
gcc  -c  -c temp.c -lm
gcc  temp.o -o final
admin1@khushi-Laptop:/tmp$ ./final
palindromeadmin1@khushi-Laptop:/tmp$ 
```

5.2 Future scope

- We enjoyed making our syntax language and transpiling it during our project and plan to continue exploring the field for further applications. Some of the points that we think this project can grow are listed below.

1. Adding code optimization :- Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

2. Adding Malloc:- MALLOC(size) allocates size bytes of dynamic memory and returns the address of the allocated memory.

3. Adding struct variables :- Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a **member** of the structure.

   Unlike an array,, a structure can contain many different data types (int, float, char, etc.).

5.3 References

1. Lexer - Parser code :

   https://www.youtube.com/watch?v=eF9qWbuQLuw

2. How to write a toy compiler:

   https://cop3402fall20.github.io/lectures/03_toy_compiler.html#org0eeeb2d

3. Abstract syntax tree:

   Leveling Up One's Parsing Game With ASTs | by Vaidehi Joshi | basics | Medium

4. How to write a programming language:

   I wrote a programming language. Here's how you can, too. (freecodecamp.org)

5. How to program your CMakelist:

   Writing CMakeLists Files — Mastering CMake

6. How to program your Makefile:

   Makefile - Quick Guide (tutorialspoint.com