LANG - CRAFT

AIM : Making our own programming language and building it's tree-walk interpreter from scratch

PROJECT REPORT

Eklavya Mentorship

Program

SOCIETY OF ROBOTICS AND AUTOMATION,

VEERMATA JIJABAI TECHNOLOGICAL INSTITUTE,

MUMBAI SEPTEMBER 2023

ACKNOWLEDGMENT

We would like to thank all the members of **SRA VJTI** for organizing the **Eklavya 2023** and allowing us to explore new domains and the learnings are unparalleled.

We are extremely grateful to our mentors

Khushi Balia and Lakshaya Singhal

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

Our Team :

- **Nishat Patil**

Email: ndpatil_b22@el.vjti.ac.in

Github: https://github.com/nishatp9

- **Siddhi Parekh**

Email: shparekh_b22@ce.vjti.ac.in

Github: https://github.com/siddhip2004

TABLE OF CONTENTS

1. PROJECT OVERVIEW

1.1 Objective of the project

- The project is to create a custom programming language from scratch.
- It will involve defining the syntax of the language, implementing a lexer, developing a parser to construct an AST, and building a tree-walk interpreter to execute the code written in the custom language.

LangCraft
CyPy

This involves writing a program according to the defined syntax and getting the required output according to the structure and logic of code.

1.2 What is an Interpreter ?

- An interpreter is a program that directly executes the instructions in a high-level language, without converting it into machine code.
- It does not generate any intermediate object code. Hence it is memory efficient.

It can work in three ways:

- Execute the source code directly and produce the output.
- Translate the source code in some intermediate code and then execute this code.
- Using an internal compiler to produce a precompiled code. Then, execute this precompiled code.

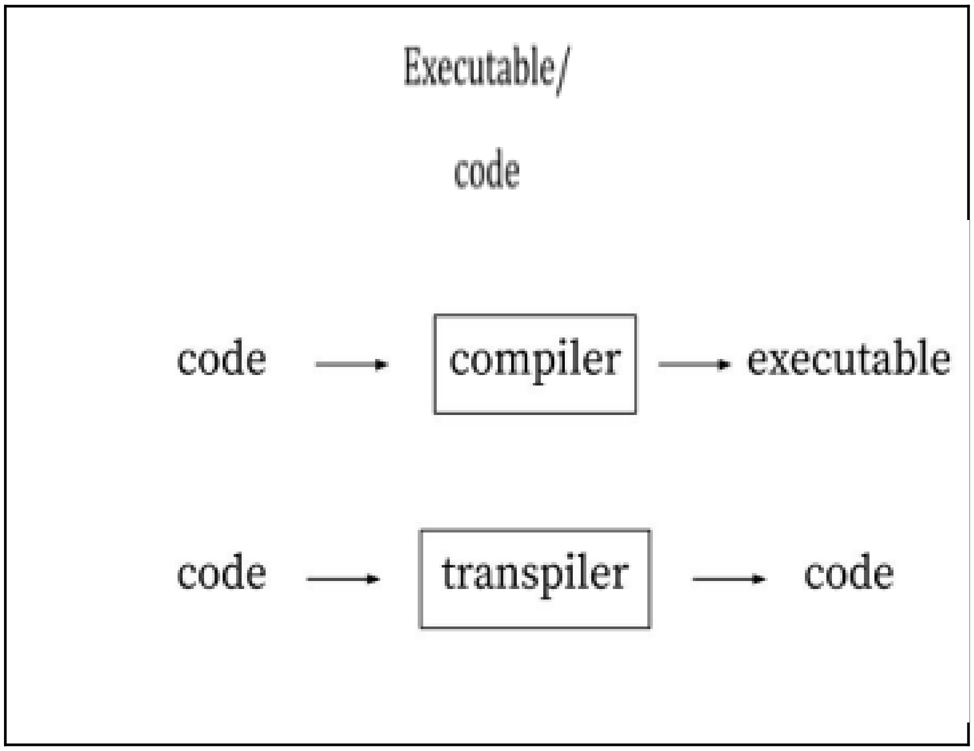
Types of Interpretation :

1. Byte Code
2. Threaded Code
3. Self
4. AST

1.3 Difference between a compiler and Interpreter

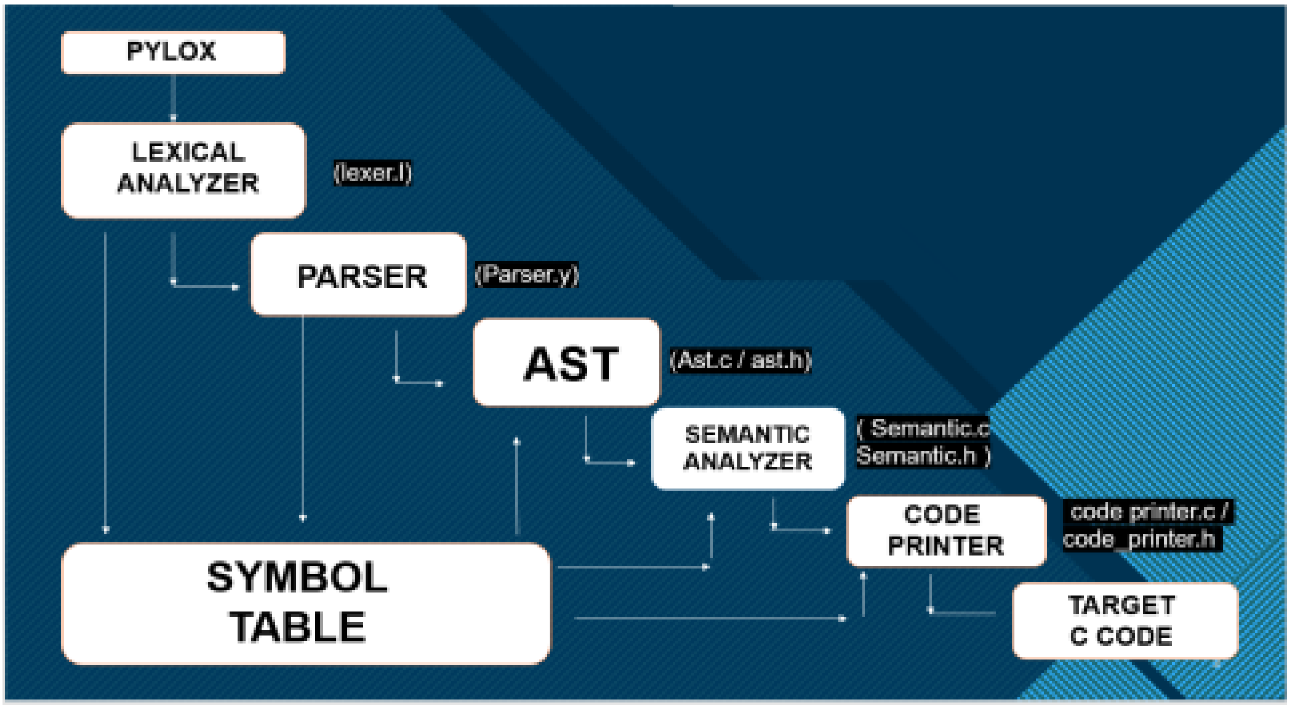
Compiler	Interpreter
It converts a source code written in a high-level language to an output code in	It converts a source code written in a high-level language to an output code in
a low-level language. a different high-level language.	
The source code has a higher level of abstraction than the output 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	Converting Java code into C++ code is

language instructions is an example of compilation.	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.

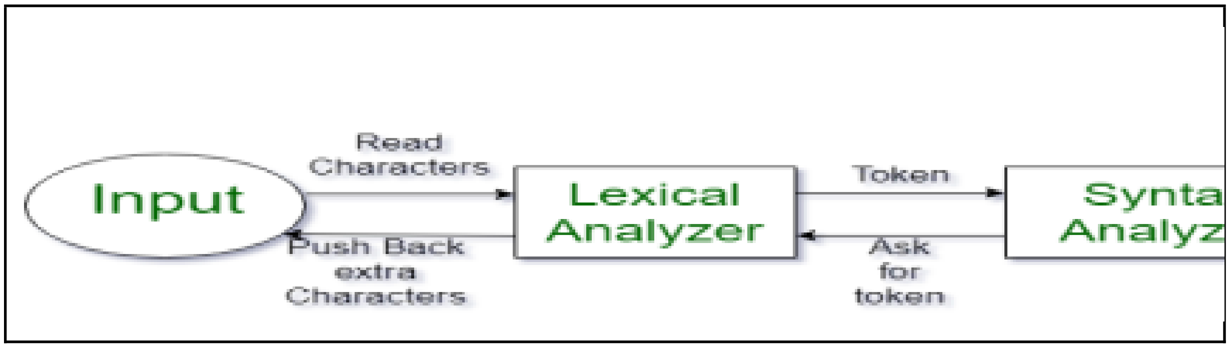
- Lexemes are said to be a sequence of characters (alphanumeric) in a token.
- There are some predefined rules for every lexeme to be identified as a valid token.
- These lexemes pass through the lexer and it gives us tokens.

- 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.



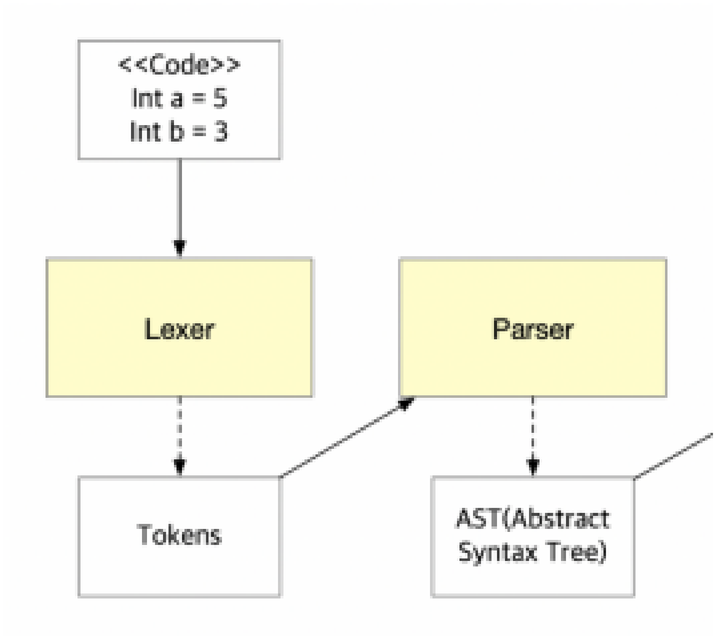
● We can conclude that the lexical analyzer phase scans the source code as a stream of characters and converts them into meaningful lexemes.

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.

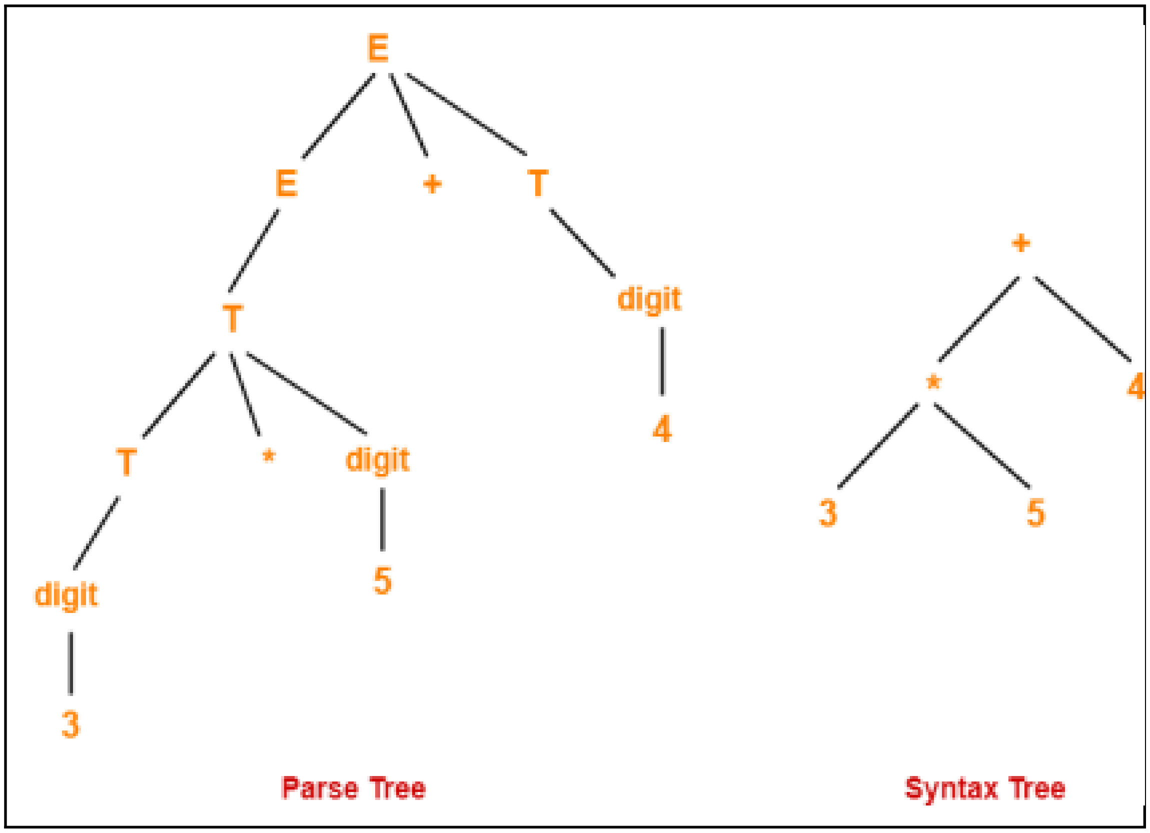
● The parser accomplishes three tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.



● The start symbol of the derivation becomes the root of the parse tree.

● In a parse tree:

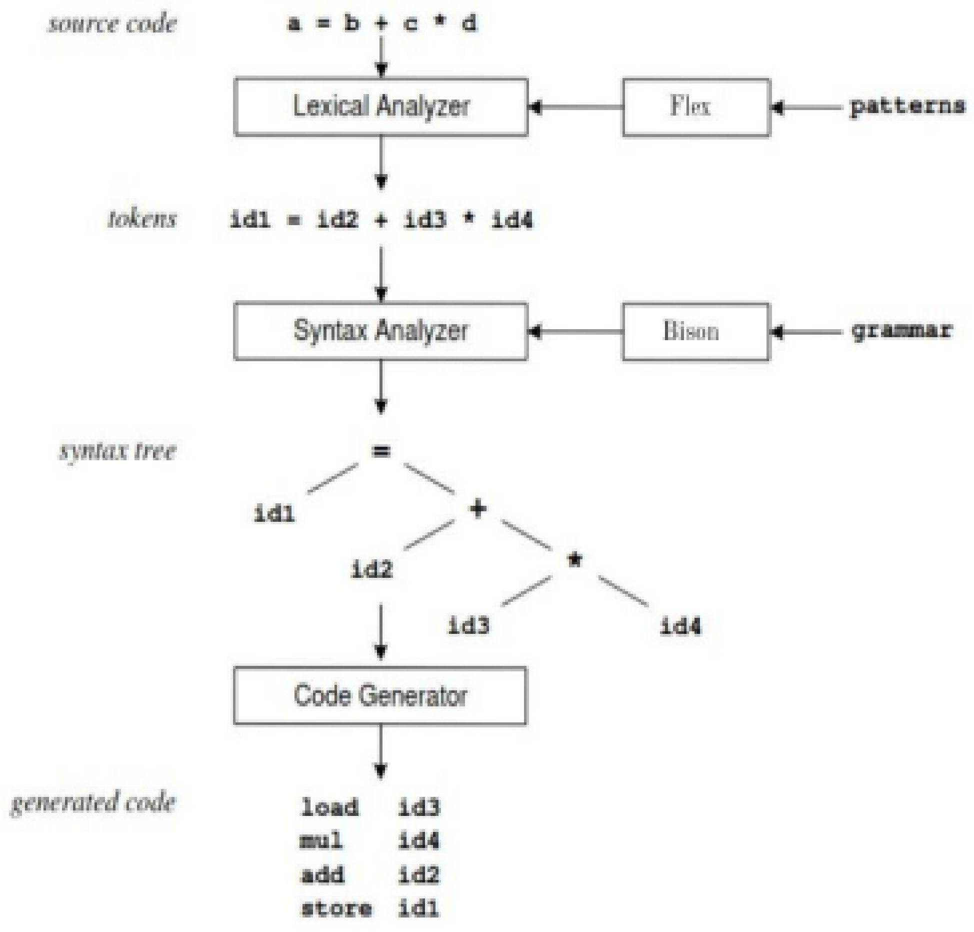
1. All leaf nodes are terminals.
2. All interior nodes are non-terminals.



- 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.

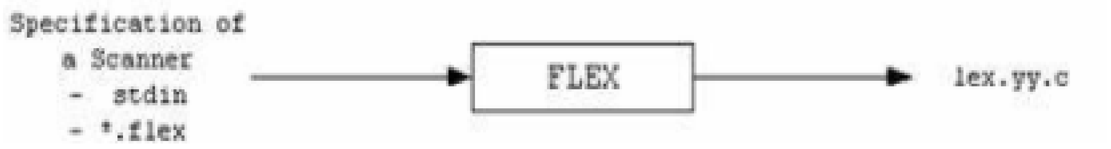
● Updating and fixing Flex and Bison source files is a lot easier than updating and fixing custom parser code.

● 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.

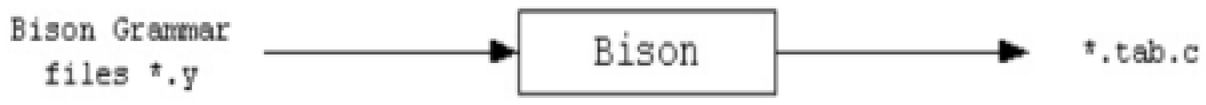


ii) Bison

● 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

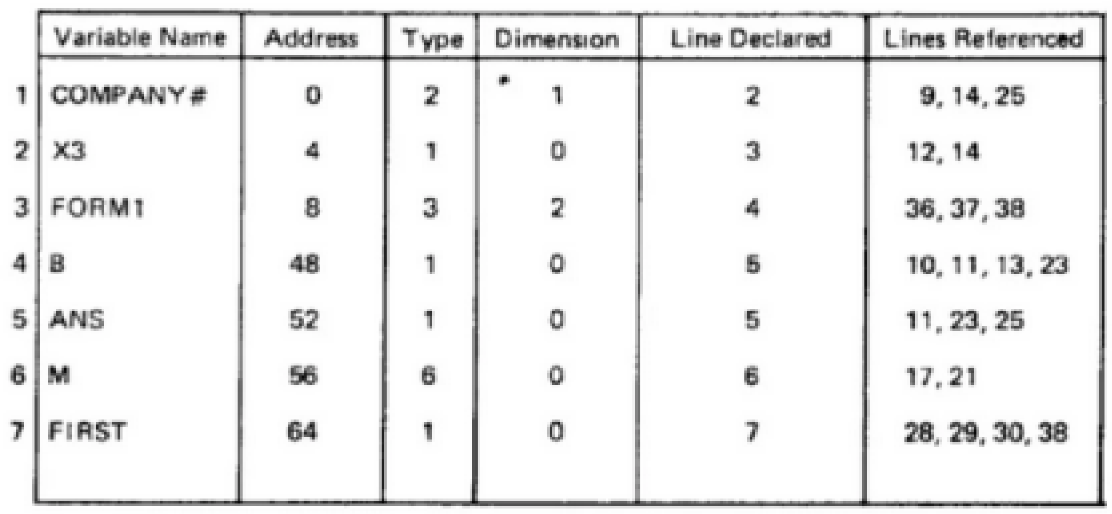
Introduction

● 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

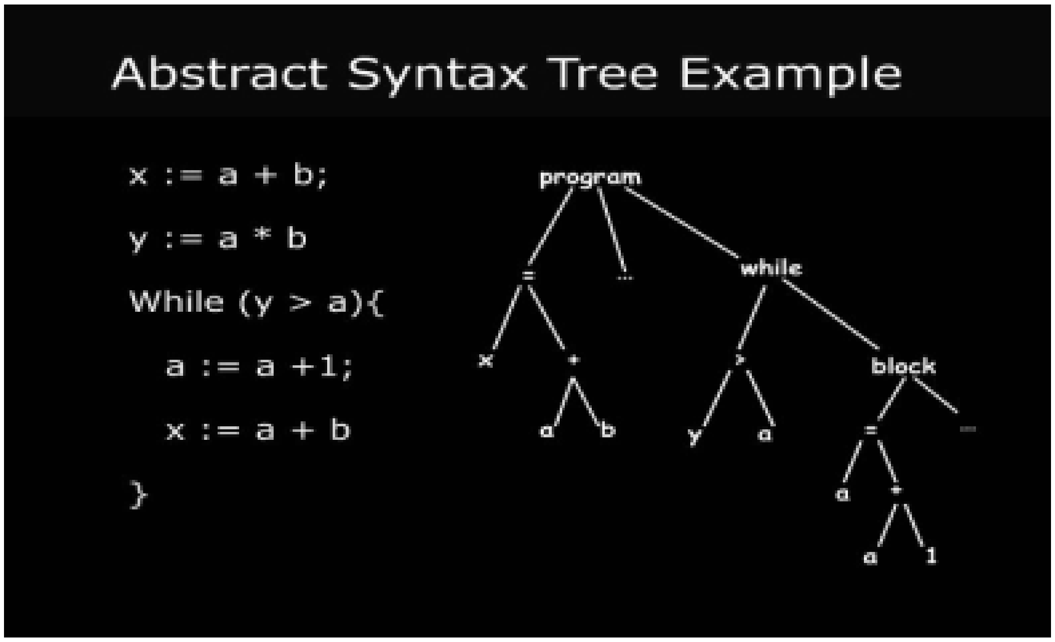
● **Lexical Analysis:** Creates new entries in the table about tokens.
● **Parser/Syntax Analysis:-** Adds information regarding attribute type,scope,dimension,line of reference , etc in the table. ● **Code printer:-** Generates code by using address information of identifier present in the table.

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
}
```

3. CUSTOM LANGUAGE - CyPy

Inspired by C++ & Python

1. Semicolon:

NO semicolon required at the end of the line.

2. Variables, Keywords:

Variables ~

No data types required before variable names to be defined.

Example:

a=10	declare integer
b=”string”	declare a string name inside double inverted commas
c= 5.765	declare float
D=’c’	declare character

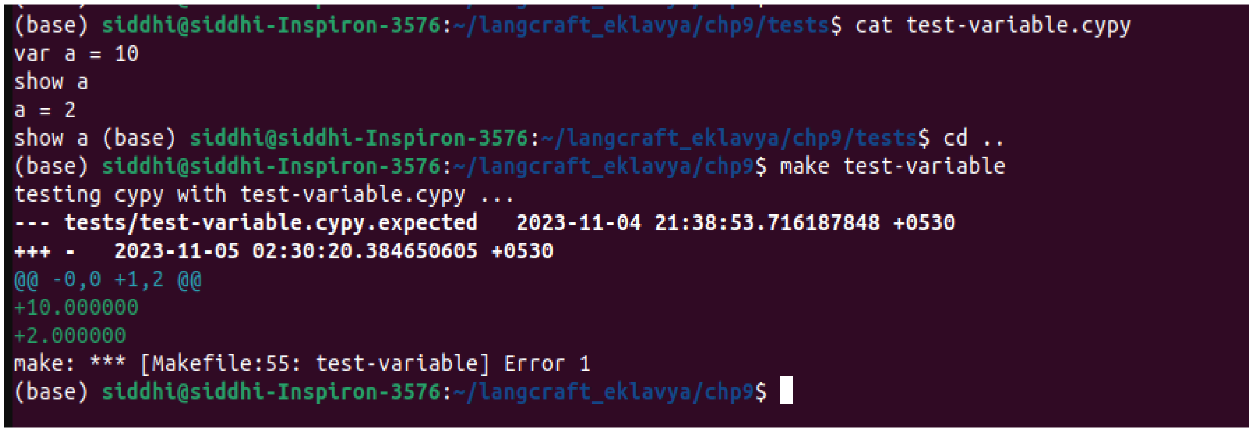
Data Types ~

bool a = t	declare a as true
bool b = f	declare b as false
bool c = !a	c is false
show f“(a)”	output is 1
show f“(c)”	output is 0

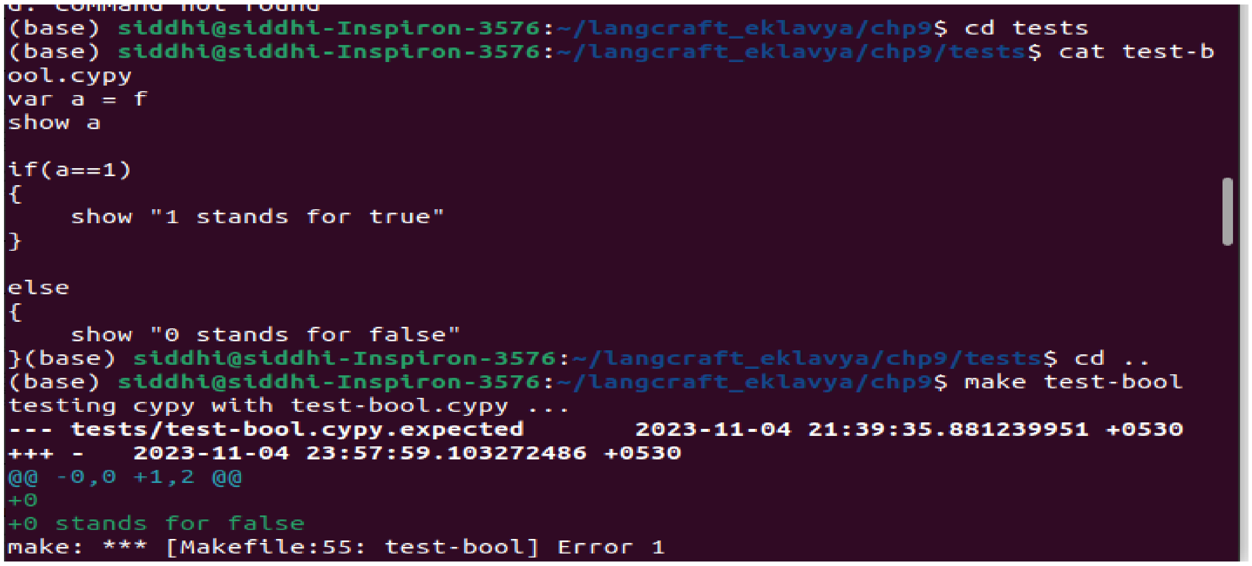
Keywords ~

int, float, double, default, main, if, elif, else, for, while, do, def, break, continue are all keywords

NOTE: for type casting data types are required.
Example : int(5.77) gives 5 or (int)5.77 gives 5



```
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9/tests$ cat test-variable.cypy
var a = 10
show a
a = 2
show a (base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9/tests$ cd ..
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9$ make test-variable
testing cypy with test-variable.cypy ...
--- tests/test-variable.cypy.expected   2023-11-04 21:38:53.716187848 +0530
+++ -   2023-11-05 02:30:20.384650605 +0530
@@ -0,0 +1,2 @@
+10.000000
+2.000000
make: *** [Makefile:55: test-variable] Error 1
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9$
```



```
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9$ cd tests
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9/tests$ cat test-b
ool.cypy
var a = f
show a

if(a==1)
{
    show "1 stands for true"
}

else
{
    show "0 stands for false"
}(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9/tests$ cd ..
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9$ make test-bool
testing cypy with test-bool.cypy ...
--- tests/test-bool.cypy.expected       2023-11-04 21:39:35.881239951 +0530
+++ -   2023-11-04 23:57:59.103272486 +0530
@@ -0,0 +1,2 @@
+0
+0 stands for false
make: *** [Makefile:55: test-bool] Error 1
```

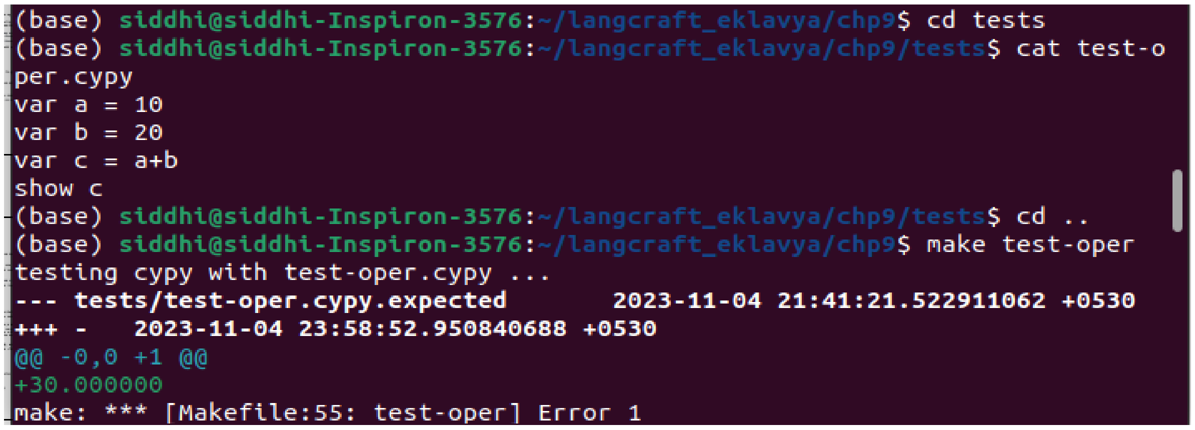
3. Operators:

1.Arithmetic operator : + Addition
- Subtraction
* multiplication
/ division
% modulo operator (remainder after division can only be used with integers)

2.Assignment operator:	=	a=b	a=b
	+=	a+=b	a=a+b
	-=	a-=b	a=a-bg

	*=	a*=b	a=a*b
	/=	a/=b	a=a/b
	%=	a%=b	a=a%b

3.Relational operators:	==	is equal to	
	!=	is not equal to	
	>	Greater than	
	<	Less than	
	>=	Greater than or equal to	
	<=	Less than or equal to	
4.Logical operators:	&&	logical AND	
	||	logical OR	
	!	logical NOT	
5.Bitwise operator:	&	binary AND	
	|	binary OR	
	~	binary one's complement	
	<<	binary left shift	
	>>	binary right shift	
6.Other operators:	sizeof	returns size of the data type	sizeof(int)=4
	?:	returns value based on condition	string result = (5>0)? "Even" : "odd"
	$	stores memory address of The operator	$num
	.	access the members of Structure or class	s1.marks=92
	->	used with pointers to access Class or struct members	ptr-> marks =92
7.Increment and Decrement operator	++	increases by 1	a=10
	–	decreases by 1	a++ gives 11
			a-- gives 9



```
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9$ cd tests
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9/tests$ cat test-o
per.cypy
var a = 10
var b = 20
var c = a+b
show c
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9/tests$ cd ..
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9$ make test-oper
testing cypy with test-oper.cypy ...
--- tests/test-oper.cypy.expected       2023-11-04 21:41:21.522911062 +0530
+++ -   2023-11-04 23:58:52.950840688 +0530
@@ -0,0 +1 @@
+30.000000
make: *** [Makefile:55: test-oper] Error 1
```

4.Scanning, Printing and Comments :

1.Scanning ~

```
a = read “”                                   when a is any data type.
  Here this read”” function will simply store the value in a

a = read “Enter the value of a”               when a is a string.
 Here read”something written here”, will  print the string inside” “ and then
 Store the value in a .
```

2.Printing ~

```
show “Things to be printed”
show “\n”

a=10
show f”Value of a is (a)\n”        output is:  Value of a is 10
```

3. Single line comment ~ ^Comments

   Multi line comment ~ ^^Comments

   Comments^^

5.Arrays, Pointers, Structures :

1.Arrays ~

```
@array_name[5] = {1,2,3,4,5}                       whether string or integer,
@array_name[ ] = {“name1”, “name2”, “name3”}       declaration style remains the
                                                   same.

array_name[ ]        [ ] used to access array elements with the help of the index no.
```

2.Pointers ~

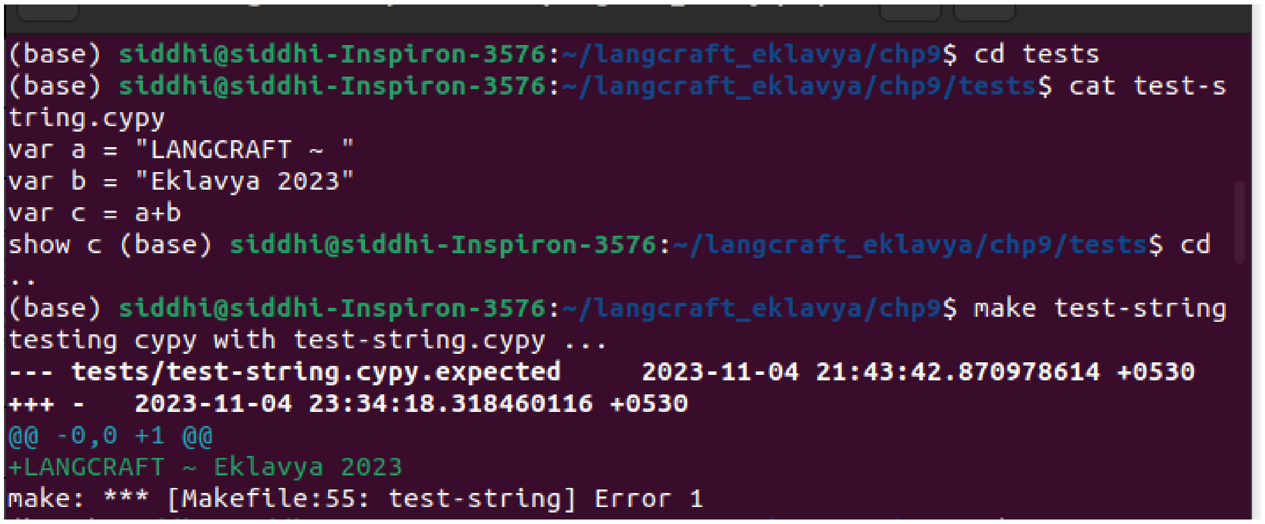
```
int #ptr = &(var_name)
```

A pointer named ptr that points to an integer variable. # is used to make a pointer

Dereferencing,increment, decrement can also be done with the help of #.

3.Structures~

```
struct (struct_name) -
     Indentation code
```



```
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9$ cd tests
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9/tests$ cat test-s
tring.cypy
var a = "LANGCRAFT ~ "
var b = "Eklavya 2023"
var c = a+b
show c (base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9/tests$ cd
..
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9$ make test-string
testing cypy with test-string.cypy ...
--- tests/test-string.cypy.expected     2023-11-04 21:43:42.870978614 +0530
+++ -   2023-11-04 23:34:18.318460116 +0530
@@ -0,0 +1 @@
+LANGCRAFT ~ Eklavya 2023
make: *** [Makefile:55: test-string] Error 1
```

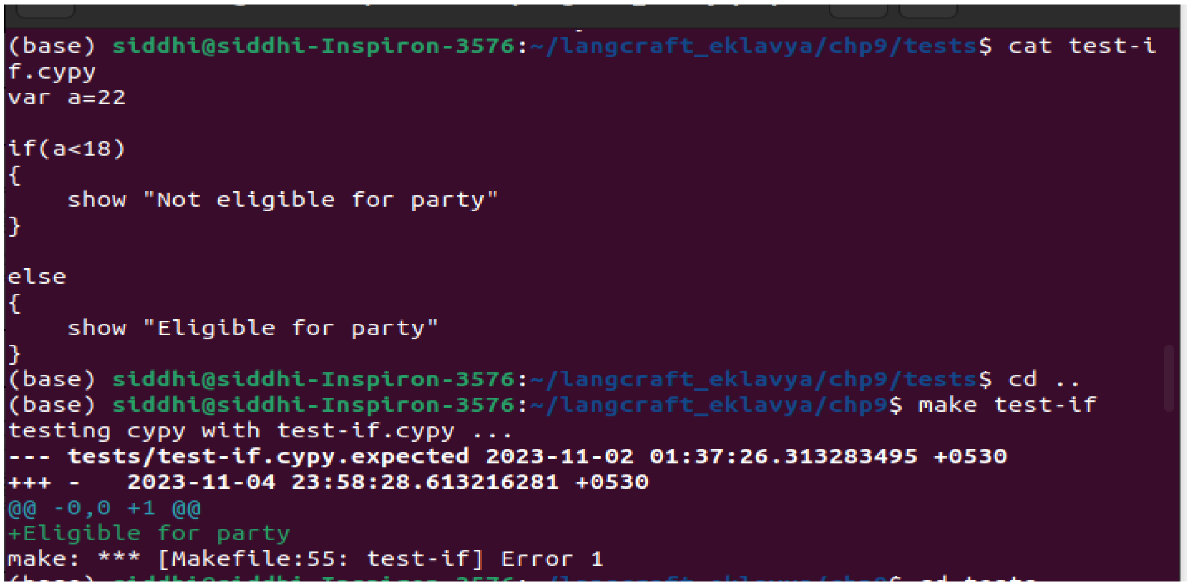
6.Control structures :

1.If-else-elseif ~

```
    if(condition)
    {
        Code
    }
    elif(condition)
    {
        Code
    }
    else
    {
        Code
    }
```

2.Switch Case ~

```
    switch(variable_name)-
        case 1-
         Indentation code
        default-
         Indentation code
```



```
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9/tests$ cat test-i
f.cypy
var a=22

if(a<18)
{
    show "Not eligible for party"
}

else
{
    show "Eligible for party"
}
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9/tests$ cd ..
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9$ make test-if
testing cypy with test-if.cypy ...
--- tests/test-if.cypy.expected 2023-11-02 01:37:26.313283495 +0530
+++ -   2023-11-04 23:58:28.613216281 +0530
@@ -0,0 +1 @@
+Eligible for party
make: *** [Makefile:55: test-if] Error 1
```

7.Loops :

1.While loop ~

```
while(condition)-
    Indentation code
```

2.Do-While loop ~

```
do-
    Indentation code
while(condition)
```

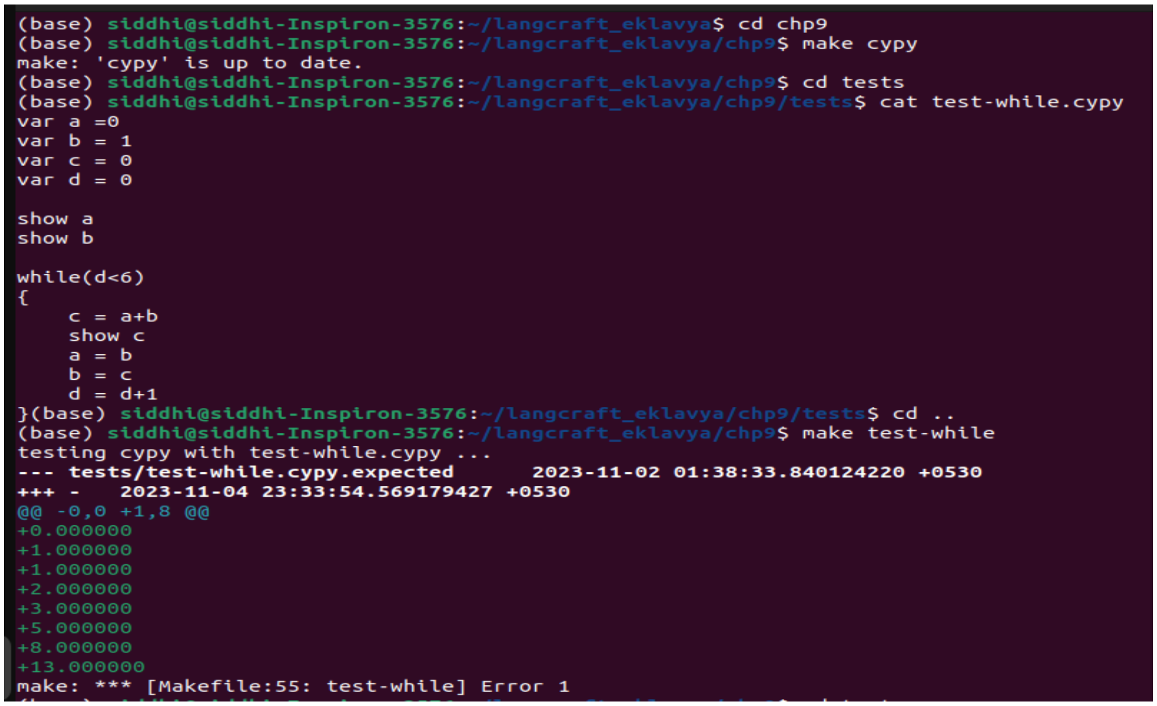
Executed at least once, even if the condition is satisfied or not.

3.For loop ~

```
for(i=0; i<n; i++) -
    Indentation code
  break
  continue
```

Semicolons required to separate 3 parts.

‘break’ and ‘continue’ are used inside the conditional statements which are either separate or inside any of above 3 loops



```
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya$ cd chp9
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9$ make cypy
make: 'cypy' is up to date.
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9$ cd tests
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9/tests$ cat test-while.cypy
var a =0
var b = 1
var c = 0
var d = 0

show a
show b

while(d<6)
{
    c = a+b
    show c
    a = b
    b = c
    d = d+1
}(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9/tests$ cd ..
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9$ make test-while
testing cypy with test-while.cypy ...
--- tests/test-while.cypy.expected      2023-11-02 01:38:33.840124220 +0530
+++ -   2023-11-04 23:33:54.569179427 +0530
@@ -0,0 +1,8 @@
+0.000000
+1.000000
+1.000000
+2.000000
+3.000000
+5.000000
+8.000000
+13.000000
make: *** [Makefile:55: test-while] Error 1
```

8.Functions :

1. Main Function ~

        def main()-
           Indentation code

   Main function contains objects of structure or class, and these objects with the help of Dot operator access the members of structure or class.

2.Any other function ~

        def function_name (parameters)-
             Indentation code

9.Space Sensitivity :

1.The language is not space sensitive, except for the indentation (4 spaces) used in control structures and loops.
2. The language is case sensitive.

4. IMPLEMENTATION

4.1 File structure

Let's look at the Eklavya - - Lang-Craft package:

```
├── cypy.cpp

├── generated

│   ├── AstPrinterDriver.d

│   ├── cypy

│   ├── cypy.d

│   ├── cypy.o

│   ├── generate_ast

│   ├── GenerateAst.d

│   └── GenerateAst.o

├── includes

│   ├── AstPrinterDriver.cpp
│   ├── AstPrinter.h
│   ├── Environment.h
│   ├── Error.h
│   ├── Expr.h
│   ├── GenerateAst.cpp
│   ├── interpreter.h
│   ├── parser.h
│   ├── RunTimeError.h
│   ├── scanner.h
│   ├── Stmt.h
│   ├── Token.h
│   └── TokenType.h
├── Makefile
```

1. **CyPy.cpp** : This file contains required functions to run the code either on prompt or a file.
2. **Includes:** This directory includes all the necessary

header files that we need during writing and compiling out syntax.

3. **Generated:** This directory contains all the files that are generated when the syntax is compiled.
4. **Makefile:** This file is the heart of the interpreter. All the .o and .d files are generated by the makefile as a result of compiling their main files. All the work of compilation is done by the makefile.

4.2 **Milestone 1- Running all files for a code:**

~ Makefile does all the work of checking for the existing files, compiling them and generating the required output files, required for the code to run.

~ So we made test-cases where every part of syntax can be tested and errors if any can be resolved. All the test-cases were provided in the Makefile

5. CONCLUSION AND FUTURE WORK

5.1 Conclusion

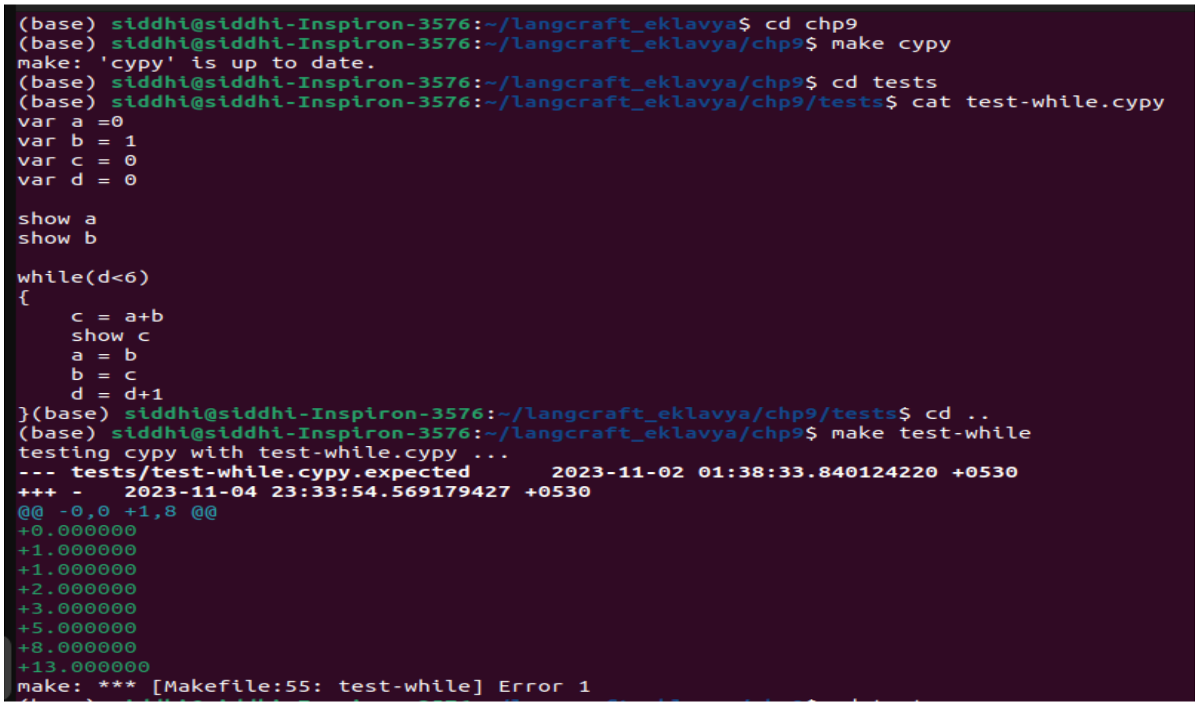
- To conclude, the aim was to interpret a custom language (CyPy language) . We were successful in interpreting the followings:-

  a) Declaration statement
  b) Assignment statement
  c) If, else condition
  d) while loop
  e) Print statement
  f) Arithmetic operators

- We have not only learned how to make our syntax language, but we have also learned how the interpreter actually reads them and how we get the output.

Output:

We wrote a code to implement the fibonacci series using a while loop.



```
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya$ cd chp9
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9$ make cypy
make: 'cypy' is up to date.
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9$ cd tests
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9/tests$ cat test-while.cypy
var a =0
var b = 1
var c = 0
var d = 0

show a
show b

while(d<6)
{
    c = a+b
    show c
    a = b
    b = c
    d = d+1
}(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9/tests$ cd ..
(base) siddhi@siddhi-Inspiron-3576:~/langcraft_eklavya/chp9$ make test-while
testing cypy with test-while.cypy ...
--- tests/test-while.cypy.expected      2023-11-02 01:38:33.840124220 +0530
+++ -   2023-11-04 23:33:54.569179427 +0530
@@ -0,0 +1,8 @@
+0.000000
+1.000000
+1.000000
+2.000000
+3.000000
+5.000000
+8.000000
+13.000000
make: *** [Makefile:55: test-while] Error 1
```

5.2 Future scope

- We enjoyed making our syntax language and interpreting 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 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.).

4. Implementation of arrays, pointers, functions and OOP.

5.3 References

1. Lexer - Parser code :

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

2. How to write an interpreter:

   https://www.toptal.com/scala/writing-an-interpreter

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)

6. How to program your Makefile:

   Makefile - Quick Guide (tutorialspoint.com