

Compiler Design Report

Team Members :

Nallanagula Anjana (AP19110010340)

Lasya Nagamalla (AP19110010365)

Chandana Kovelamudi (AP19110010369)

Kota Hema Sai (AP19110010454)

Tondupalli Bhavana (AP19110010482)

Language :

Python + HTML

Data Types Used In The Project :

Python :

- a. **Int:** It contains negative or positive whole numbers without fractions or decimals. In Python, there's no restriction on how lengthy an integer value can be.

Eg: `x = 20`

- b. **Float:** It is written with a decimal point dividing the integer and fractional parts. The character e or E followed by positive or negative integers is also floating-point numbers.

Eg: `>>> 8.4e7`
`>>> type(.4e7)`
`<class 'float'>`

- c. **Str:** A string is a set of one or more characters put in a single quote, double-quote, or triple quote.

Eg: `s = "This is a string"`
`print(type(s))`
Output: str

HTML :

- a. **Text String:** A text string can take any characters from the document character set and may include character entities. Stores a string with a maximum length of 2,147,483,647 printable characters.

Eg: Compiler design (can be any name or text or description).

- b. **Colour:** A colour value may either be a hexadecimal number (prefixed by a hash mark) or certainly considered one among 16 colours. The colour names are case-sensitive.

Eg: Black = "#000000", Blue = "#0000FF"
`<print style="color : blue">4</print>`

- c. **Length:** Pixels: The value is an integer that represents the number of pixels on the canvas.

Eg: `<print style="font-size : 50px">2</print>`

Assumptions:

Python doesn't need any pre-declarations for variables before its use. The assumptions for variables used are :

1. **Int** - For Int datatype, Integer value ranges from **-2,147,483,647** to **2,147,483,647** for 9 or 10 digits of precision.

Syntax : For i in range(-2,147,483,647, 2,147,483,647)

2. **Float** - For float datatype, values are decimals representing real numbers.

3. **Str**- Set of characters, from a-z to A-Z.

Tokens:

Token	Pattern	Lexeme
id	(letter)(letter digit)*	A1, ls
either	characters e, i, t, h, e, r	either
<print>	characters <,p, r, i, n, t, >	<print>
list	[(int float string)*]	[1, "s", 4.2]
relational operators	<, >, <=, >=, !=	<=,>=
number	int - (sign)?(digits)+ float - (sign)?(digits)+.(digits)+	0, 5.432, 3.14159, 8.32e45
style attribute	style="attr:val"	style="color:blue"

Example:

for i in fruits:

 <print style="color:green">i</print>

Token	Lexeme
Keyword	for
identifier	i
Keyword	in
identifier	fruits
delimiter	:
Start tag	<print>
style attribute	style="color:green"
End tag	</print>

Decision-Making Statement: Decision-making allows us to run a particular block of code for a particular decision and based on the validity of a particular condition, the decisions are made accordingly.

In this compiler, Either - or else - or statement works for the decision-making.

Either Statement: Either statement is used to check a specific condition. If the condition is true, the block of code will be executed.

Or Statement: It is combined with either statement and it contains a block of code that executes if the condition expression in either block is false.

Or else Statement: This statement is used to check multiple expressions for true and executes when one of the conditions is true.

Syntax :

```
<python>
either expression:
    statement(s)
orelse expression:
    statement(s)
orelse expression:
    statement(s)
or:
    statement (s)
</python>
```

Example :

```
<python>
a=23
either a>=5:
<print>a is greater than 5</print>
orelse a<=10:
< print> a is less than 10</print>
or:
<print>a</print>
</python>
```

Iterative Statements:

- 1) For Loop
- 2) While Loop

For loop: It is used for iterating over a sequence and can execute a set of statements. It can be a list, tuple, strings, etc.

Syntax:

```
<python>
for variable_name in sequence:
    statement(s)
</python>
```

Example:

```
<python>
for i in "lasya":
    <print>(i)</print>
</python>
```

While Loop: It is used to execute a set of statements as long as the condition is true. Else statement is used to execute a block of code when the condition is no longer true.

Syntax:

```
<python>
while(condition):
    statement(s)
else:
    statement (s)
</python>
```

Example:

```
<python>
int i=2;
float j=3.3
while i < 4:
```

```

while j< 8.5:
    <print>(i,"")</print>;
else:
    <print>Loop is finished</print>
</python>

```

List: The syntax of the list is similar to that in python and we can perform all the basic operations like list indexing, slicing, and use built-in methods to manipulate the list. The syntax for declaring a list -

- a. `ls = [1, 2, 3, 4]`
- b. `ls = [1, 's', 2.3]`
- c. `ls = list()`

Function: The syntax of the function is similar to that in python and we can print or return the value of expressions from the function. We can pass arguments and evaluate the results using a function. The function can return int, float, string, and boolean values. The syntax of the function is -

- a. `def fun_name(param1, param2):`
`<print>param1 + param2</print>`
- b. `def fun_name(param1, param2):`
`return param1 + param2`

CFG's Used For Parsing :

a. Integer :

```

<sign> → + | - | ε
<int> → <sign> <digits>
<digits> → <digit><digits> | ε
<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

b. Float :

```

<digits> → <digit><digits> | ε
<digit> → 0 | 1 | ... | 9
<sign> → + | - | ε
<fraction> → .<digits> | ε
<exponent> → E <sign> <digits> | ε
<float> → <sign> <digits> <fraction> <exponent>

```

c. String :

$\langle \text{strings} \rangle \rightarrow \langle \text{str} \rangle \mid \langle \text{int} \rangle \mid \langle \text{float} \rangle \mid \langle \text{str} \rangle \langle \text{strings} \rangle \mid \epsilon$
 $\langle \text{str} \rangle \rightarrow a \mid b \mid c \mid \dots \mid z$

d. Decision - Making Statement :

$\langle \text{either_stmt} \rangle \rightarrow \text{either } \langle \text{expression} \rangle : (\langle \text{print_stmt} \rangle \mid \langle \text{either_stmt} \rangle)$
 $\mid \text{either } \langle \text{expression} \rangle : (\langle \text{print_stmt} \rangle \mid \langle \text{either_stmt} \rangle) \text{ or } : (\langle \text{print_stmt} \rangle \mid \langle \text{either_stmt} \rangle)$
 $\mid \text{either } \langle \text{expression} \rangle : (\langle \text{print_stmt} \rangle \mid \langle \text{either_stmt} \rangle) (\text{orelse } \langle \text{expression} \rangle : (\langle \text{print_stmt} \rangle \mid \langle \text{either_stmt} \rangle)) \text{ or } : (\langle \text{print_stmt} \rangle \mid \langle \text{either_stmt} \rangle) \mid \langle \text{either_stmt} \rangle \mid \epsilon$
 $\langle \text{print_stmt} \rangle \rightarrow \langle \text{print} \rangle (\langle \text{expression} \rangle \mid \langle \text{stmt} \rangle) \langle \text{/print} \rangle$
 $\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle \langle \text{oper} \rangle \langle \text{term} \rangle \mid \langle \text{term} \rangle$
 $\langle \text{term} \rangle \rightarrow \langle \text{id} \rangle \mid \langle \text{number} \rangle$
 $\langle \text{id} \rangle \rightarrow \langle \text{letter} \rangle \langle \text{id} \rangle \mid \langle \text{digit} \rangle \langle \text{id} \rangle \mid \epsilon$
 $\langle \text{number} \rangle \rightarrow \langle \text{int} \rangle \mid \langle \text{float} \rangle \mid \langle \text{fraction} \rangle \mid \langle \text{exponent} \rangle$
 $\langle \text{int} \rangle \rightarrow \langle \text{sign} \rangle \langle \text{digits} \rangle$
 $\langle \text{sign} \rangle \rightarrow + \mid - \mid \epsilon$
 $\langle \text{float} \rangle \rightarrow \langle \text{sign} \rangle \langle \text{digits} \rangle \langle \text{fraction} \rangle \langle \text{exponent} \rangle$
 $\langle \text{fraction} \rangle \rightarrow . \langle \text{digits} \rangle \mid \epsilon$
 $\langle \text{exponent} \rangle \rightarrow E \langle \text{sign} \rangle \langle \text{digits} \rangle \mid \epsilon$
 $\langle \text{oper} \rangle \rightarrow \langle \text{>} \rangle \mid \langle \text{<=} \rangle \mid \langle \text{>=} \rangle \mid \langle \text{<} \rangle \mid \langle \text{>} \rangle \mid \langle \text{*} \rangle \mid \langle \text{-} \rangle \mid \langle \text{\%} \rangle \mid //$
 $\langle \text{letter} \rangle \rightarrow [A-Za-z]$
 $\langle \text{digit} \rangle \rightarrow [0-9]$
 $\langle \text{digits} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{digits} \rangle \mid \epsilon$

e. For Loop :

$\langle \text{s} \rangle \rightarrow \text{for } \langle \text{var_name} \rangle \text{ in } \langle \text{sequence} \rangle : (\langle \text{print_stmt} \rangle \mid \langle \text{s} \rangle) \mid \epsilon$
 $\langle \text{var_name} \rangle \rightarrow \langle \text{letter} \rangle \mid \langle \text{letters} \rangle$
 $\langle \text{sequence} \rangle \rightarrow \langle \text{id} \rangle \mid \langle \text{val} \rangle$
 $\langle \text{print_stmt} \rangle \rightarrow \langle \text{print} \rangle (\langle \text{expression} \rangle \mid \langle \text{stmt} \rangle) \langle \text{/print} \rangle$
 $\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle \langle \text{oper} \rangle \langle \text{term} \rangle \mid \langle \text{term} \rangle$
 $\langle \text{term} \rangle \rightarrow \langle \text{id} \rangle \mid \langle \text{number} \rangle$
 $\langle \text{id} \rangle \rightarrow \langle \text{letter} \rangle \langle \text{id} \rangle \mid \langle \text{digit} \rangle \langle \text{id} \rangle \mid \epsilon$
 $\langle \text{val} \rangle \rightarrow \langle \text{strings} \rangle, \langle \text{val} \rangle \mid \langle \text{int} \rangle, \langle \text{val} \rangle \mid \langle \text{float} \rangle, \langle \text{val} \rangle \mid \langle \text{strings} \rangle \mid \langle \text{int} \rangle \mid \langle \text{float} \rangle \mid \epsilon$
 $\langle \text{strings} \rangle \rightarrow \langle \text{letter} \rangle \mid \langle \text{int} \rangle \mid \langle \text{float} \rangle \mid \langle \text{letter} \rangle \langle \text{strings} \rangle \mid \epsilon$
 $\langle \text{number} \rangle \rightarrow \langle \text{int} \rangle \mid \langle \text{float} \rangle \mid \langle \text{fraction} \rangle \mid \langle \text{exponent} \rangle$

$\langle \text{int} \rangle \rightarrow \langle \text{sign} \rangle \langle \text{digits} \rangle$
 $\langle \text{sign} \rangle \rightarrow + \mid - \mid \epsilon$
 $\langle \text{float} \rangle \rightarrow \langle \text{sign} \rangle \langle \text{digits} \rangle \langle \text{fraction} \rangle \langle \text{exponent} \rangle$
 $\langle \text{fraction} \rangle \rightarrow . \langle \text{digits} \rangle \mid \epsilon$
 $\langle \text{exponent} \rangle \rightarrow E \langle \text{sign} \rangle \langle \text{digits} \rangle \mid \epsilon$
 $\langle \text{oper} \rangle \rightarrow \langle \mid \rangle \mid \langle = \rangle \mid \langle > \rangle \mid \langle < \rangle \mid \langle + \rangle \mid \langle * \rangle \mid \langle - \rangle \mid \langle \% \rangle \mid \langle // \rangle$
 $\langle \text{letter} \rangle \rightarrow [A-Za-z]$
 $\langle \text{letters} \rangle \rightarrow \langle \text{letter} \rangle \langle \text{letters} \rangle \mid \epsilon$
 $\langle \text{digit} \rangle \rightarrow [0-9]$
 $\langle \text{digits} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{digits} \rangle \mid \epsilon$

f. While Loop :

$S \rightarrow \text{while} \langle \text{condition} \rangle : \langle \text{print_stmt} \rangle \langle \text{else_stmt} \rangle$
 $\langle \text{else_stmt} \rangle \rightarrow \text{else} : \langle \text{print_stmt} \rangle \mid \epsilon$
 $\langle \text{print_stmt} \rangle \rightarrow \langle \text{print} \rangle (\langle \text{term} \rangle) \langle / \text{print} \rangle$
 $\langle \text{condition} \rangle \rightarrow \langle \text{term} \rangle \mid \langle \text{term} \rangle \langle \text{rel_oper} \rangle \langle \text{condition} \rangle$
 $\langle \text{term} \rangle \rightarrow \langle \text{id} \rangle \mid \langle \text{number} \rangle$
 $\langle \text{id} \rangle \rightarrow \langle \text{letter} \rangle \langle \text{id} \rangle \mid \langle \text{digit} \rangle \langle \text{id} \rangle \mid \epsilon$
 $\langle \text{number} \rangle \rightarrow \langle \text{int} \rangle \mid \langle \text{float} \rangle \mid \langle \text{fraction} \rangle \mid \langle \text{exponent} \rangle$
 $\langle \text{int} \rangle \rightarrow \langle \text{sign} \rangle \langle \text{digits} \rangle$
 $\langle \text{sign} \rangle \rightarrow + \mid - \mid \epsilon$
 $\langle \text{float} \rangle \rightarrow \langle \text{sign} \rangle \langle \text{digits} \rangle \langle \text{fraction} \rangle \langle \text{exponent} \rangle$
 $\langle \text{fraction} \rangle \rightarrow . \langle \text{digits} \rangle \mid \epsilon$
 $\langle \text{exponent} \rangle \rightarrow E \langle \text{sign} \rangle \langle \text{digits} \rangle \mid \epsilon$
 $\langle \text{rel_ope} \rangle \rightarrow \langle \mid \rangle \mid \langle = \rangle \mid \langle > \rangle \mid \langle < \rangle \mid \langle + \rangle \mid \langle * \rangle \mid \langle - \rangle \mid \langle \% \rangle \mid \langle // \rangle$
 $\langle \text{letter} \rangle \rightarrow [A-Za-z]$
 $\langle \text{digit} \rangle \rightarrow [0-9]$
 $\langle \text{digits} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{digits} \rangle \mid \epsilon$

g. List :

$\langle \text{list_stmt} \rangle \rightarrow \langle \text{id} \rangle = [\langle \text{list_val} \rangle] \mid \text{list}()$
 $\langle \text{list_val} \rangle \rightarrow \langle \text{val} \rangle \mid \epsilon$
 $\langle \text{val} \rangle \rightarrow \langle \text{strings} \rangle , \langle \text{val} \rangle \mid \langle \text{int} \rangle , \langle \text{val} \rangle \mid \langle \text{float} \rangle , \langle \text{val} \rangle \mid \langle \text{strings} \rangle \mid \langle \text{int} \rangle \mid \langle \text{float} \rangle \mid \epsilon$
 $\langle \text{strings} \rangle \rightarrow \langle \text{str} \rangle \mid \langle \text{int} \rangle \mid \langle \text{float} \rangle \mid \langle \text{str} \rangle \langle \text{strings} \rangle \mid \epsilon$
 $\langle \text{str} \rangle \rightarrow a \mid b \mid \dots \mid z$
 $\langle \text{int} \rangle \rightarrow \langle \text{sign} \rangle \langle \text{digits} \rangle$
 $\langle \text{digits} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{digits} \rangle \mid \epsilon$

$\langle \text{digit} \rangle \rightarrow 0 \mid 1 \mid \dots \mid 9$
 $\langle \text{sign} \rangle \rightarrow + \mid - \mid \varepsilon$
 $\langle \text{float} \rangle \rightarrow \langle \text{sign} \rangle \langle \text{digits} \rangle \langle \text{fraction} \rangle \langle \text{exponent} \rangle$
 $\langle \text{fraction} \rangle \rightarrow . \langle \text{digits} \rangle \mid \varepsilon$
 $\langle \text{exponent} \rangle \rightarrow E \langle \text{sign} \rangle \langle \text{digits} \rangle \mid \varepsilon$
 $\langle \text{id} \rangle \rightarrow \langle \text{str} \rangle \langle \text{letters} \rangle$
 $\langle \text{letters} \rangle \rightarrow \langle \text{str} \rangle \langle \text{letters} \rangle \mid \langle \text{digit} \rangle \langle \text{letters} \rangle \mid \varepsilon$

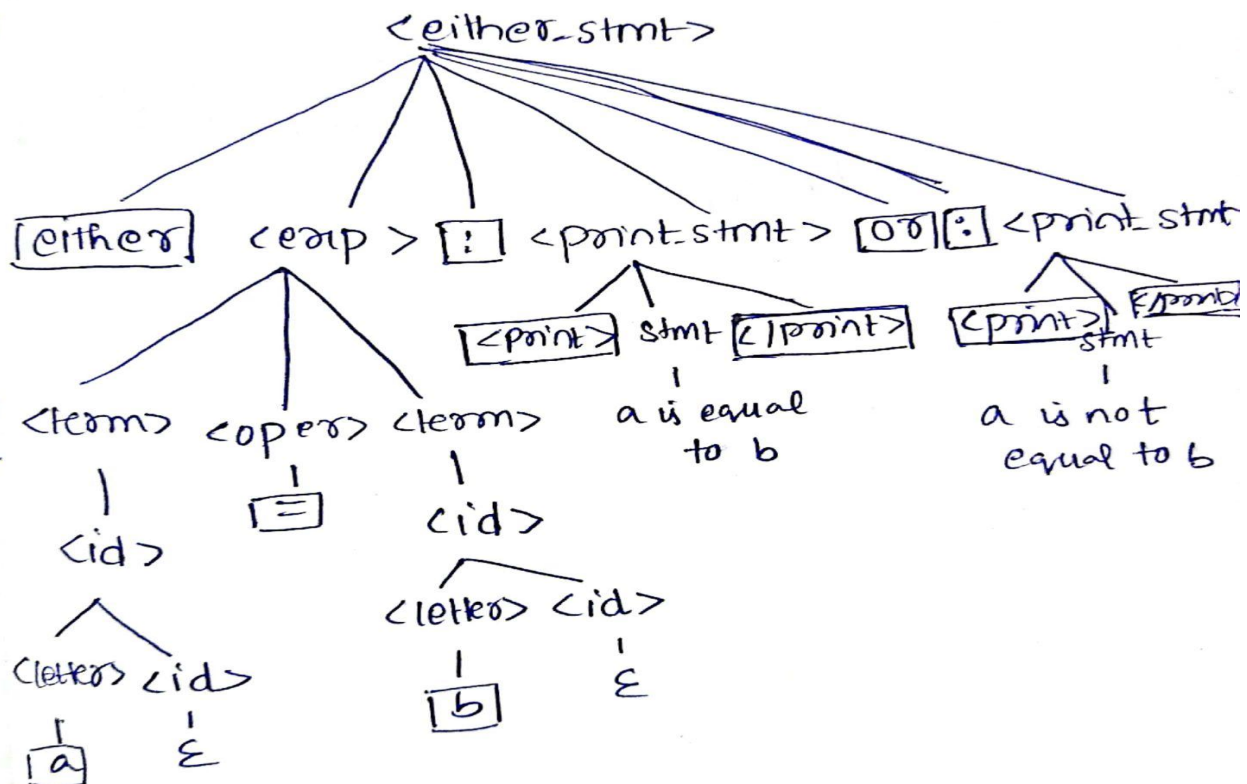
h. Function :

$\langle \text{fun_stmt} \rangle \rightarrow \text{def } \langle \text{fun_name} \rangle : \langle \text{fun_val} \rangle$
 $\langle \text{fun_name} \rangle \rightarrow \langle \text{id} \rangle \backslash (\langle \text{param} \rangle \backslash)$
 $\langle \text{id} \rangle \rightarrow \langle \text{str} \rangle \langle \text{letters} \rangle$
 $\langle \text{letters} \rangle \rightarrow \langle \text{str} \rangle \langle \text{letters} \rangle \mid \langle \text{digit} \rangle \langle \text{letters} \rangle \mid \varepsilon$
 $\langle \text{str} \rangle \rightarrow a \mid b \mid \dots \mid z$
 $\langle \text{digit} \rangle \rightarrow 0 \mid 1 \mid \dots \mid 9$
 $\langle \text{param} \rangle \rightarrow \langle \text{strings} \rangle , \langle \text{param} \rangle \mid \langle \text{int} \rangle , \langle \text{param} \rangle \mid \langle \text{float} \rangle , \langle \text{param} \rangle \mid \langle \text{id} \rangle , \langle \text{param} \rangle \mid$
 $\langle \text{strings} \rangle \mid \langle \text{int} \rangle \mid \langle \text{float} \rangle \mid \langle \text{id} \rangle \mid \varepsilon$
 $\langle \text{strings} \rangle \rightarrow \langle \text{str} \rangle \mid \langle \text{int} \rangle \mid \langle \text{float} \rangle \mid \langle \text{str} \rangle \langle \text{strings} \rangle \mid \varepsilon$
 $\langle \text{int} \rangle \rightarrow \langle \text{sign} \rangle \langle \text{digits} \rangle$
 $\langle \text{digits} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{digits} \rangle \mid \varepsilon$
 $\langle \text{digit} \rangle \rightarrow 0 \mid 1 \mid \dots \mid 9$
 $\langle \text{sign} \rangle \rightarrow + \mid - \mid \varepsilon$
 $\langle \text{float} \rangle \rightarrow \langle \text{sign} \rangle \langle \text{digits} \rangle \langle \text{fraction} \rangle \langle \text{exponent} \rangle$
 $\langle \text{fraction} \rangle \rightarrow . \langle \text{digits} \rangle \mid \varepsilon$
 $\langle \text{exponent} \rangle \rightarrow E \langle \text{sign} \rangle \langle \text{digits} \rangle \mid \varepsilon$
 $\langle \text{fun_val} \rangle \rightarrow \backslash \text{t } \langle \text{print_stmt} \rangle \mid \backslash \text{t return } \langle \text{expr} \rangle$
 $\langle \text{print_stmt} \rangle \rightarrow \backslash \langle \text{print} \rangle \langle \text{expr} \rangle \backslash \langle \text{print} \rangle$
 $\langle \text{expr} \rangle \rightarrow \langle \text{id} \rangle \mid \langle \text{id} \rangle \langle \text{op} \rangle \langle \text{id} \rangle$
 $\langle \text{op} \rangle \rightarrow + \mid - \mid * \mid \% \mid // \mid < \mid > \mid ==$

Design of Parser: LALR parser is used to parse the tokens to generate a parse tree in the syntax analysis phase. It is a bottom-up parser, which builds the parse tree in the bottom-up method by constructing closure and goto for each grammar symbol. The rightmost derivation is used to build the parse tree. The parse trees for a few constructs in the language -

a. Parser tree for decision-making statement

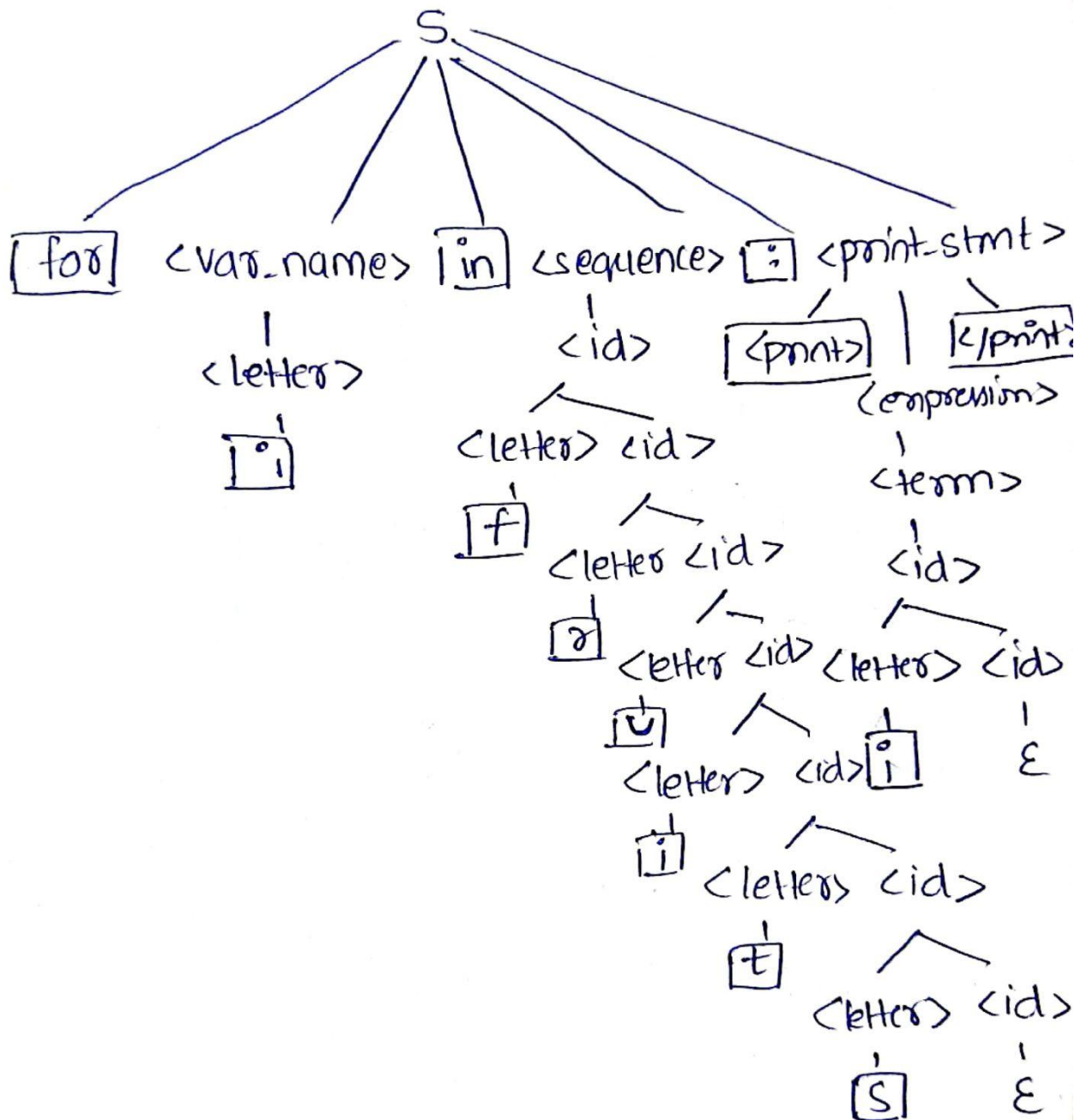
Ex:- Either $a=b$;
 $\langle \text{Print} \rangle$ a is equal to b $\langle / \text{Print} \rangle$
 or;
 $\langle \text{Print} \rangle$ a is not equal to b $\langle / \text{Print} \rangle$



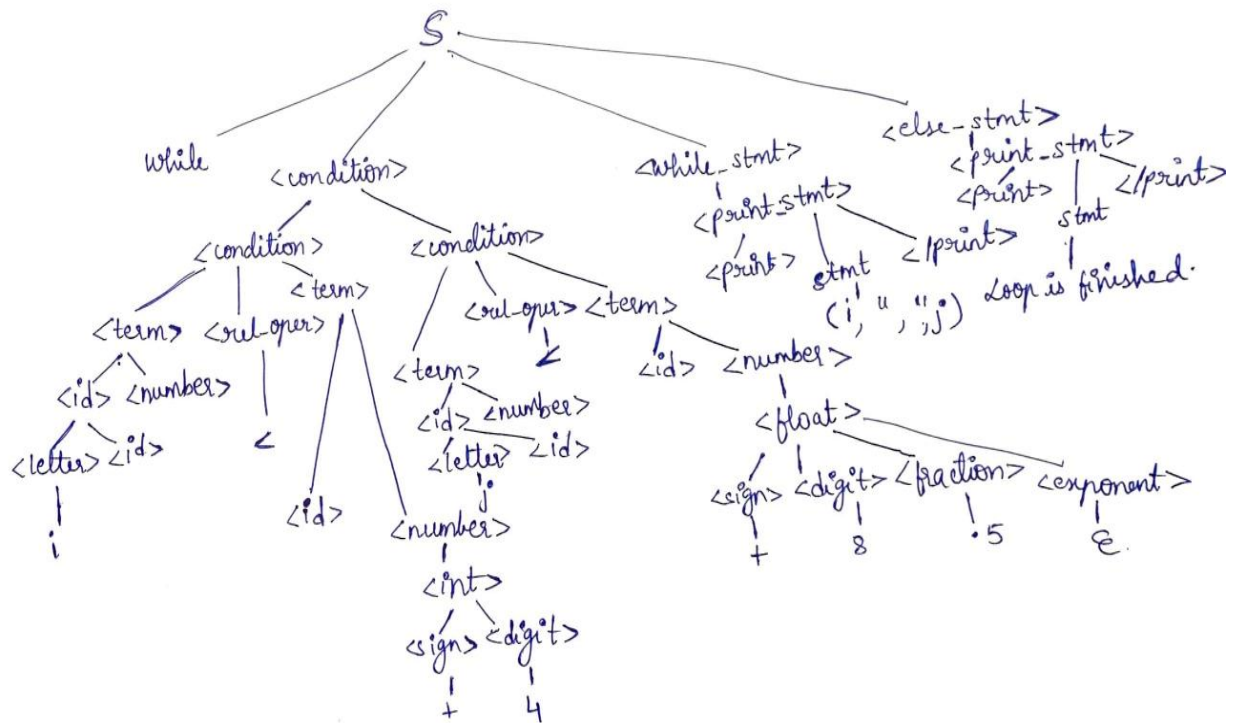
b. Parse tree for a for loop :

Eat for i in fruits:

```
<print> i </print>
```

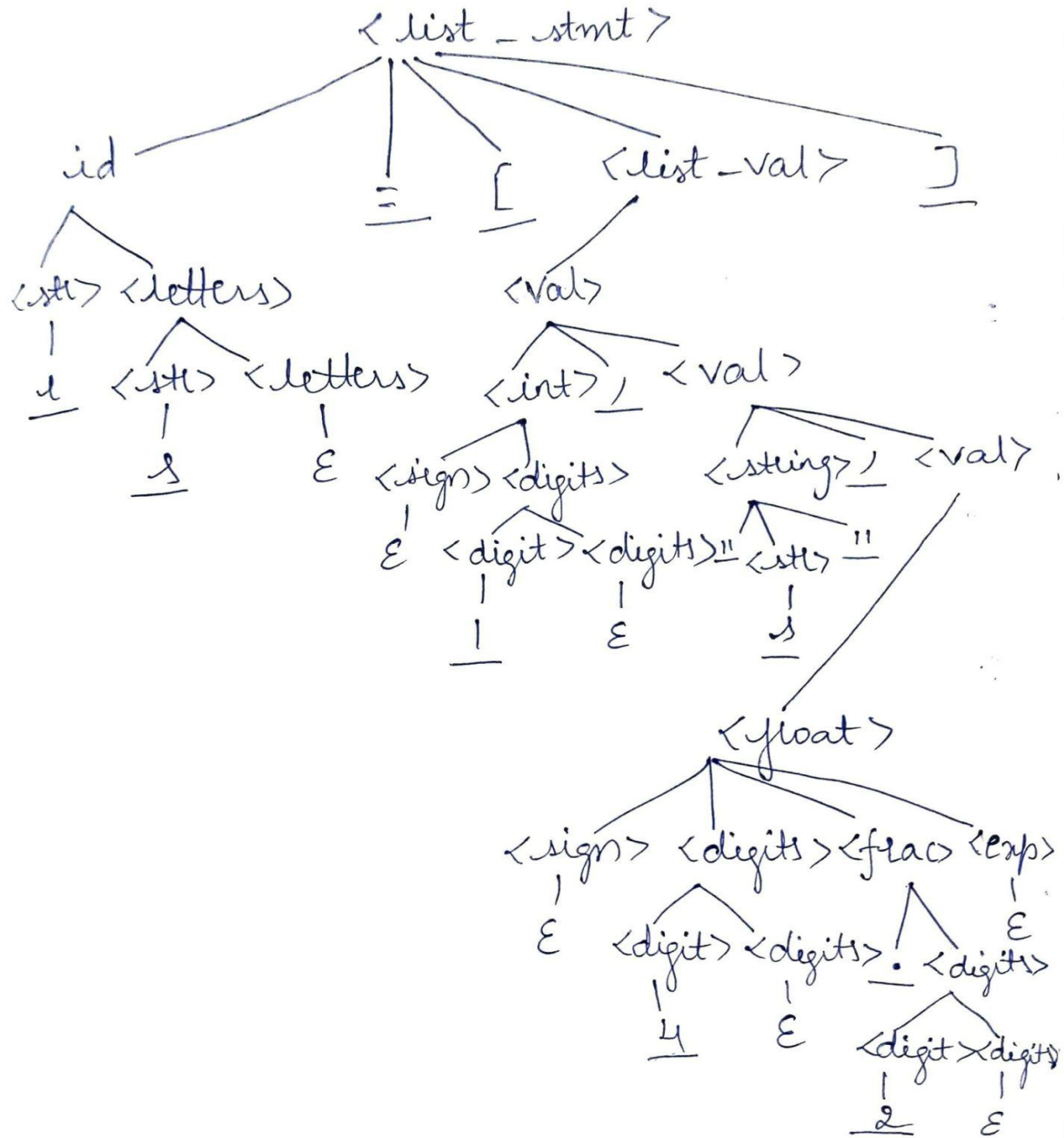


c. Parse tree for a while loop :



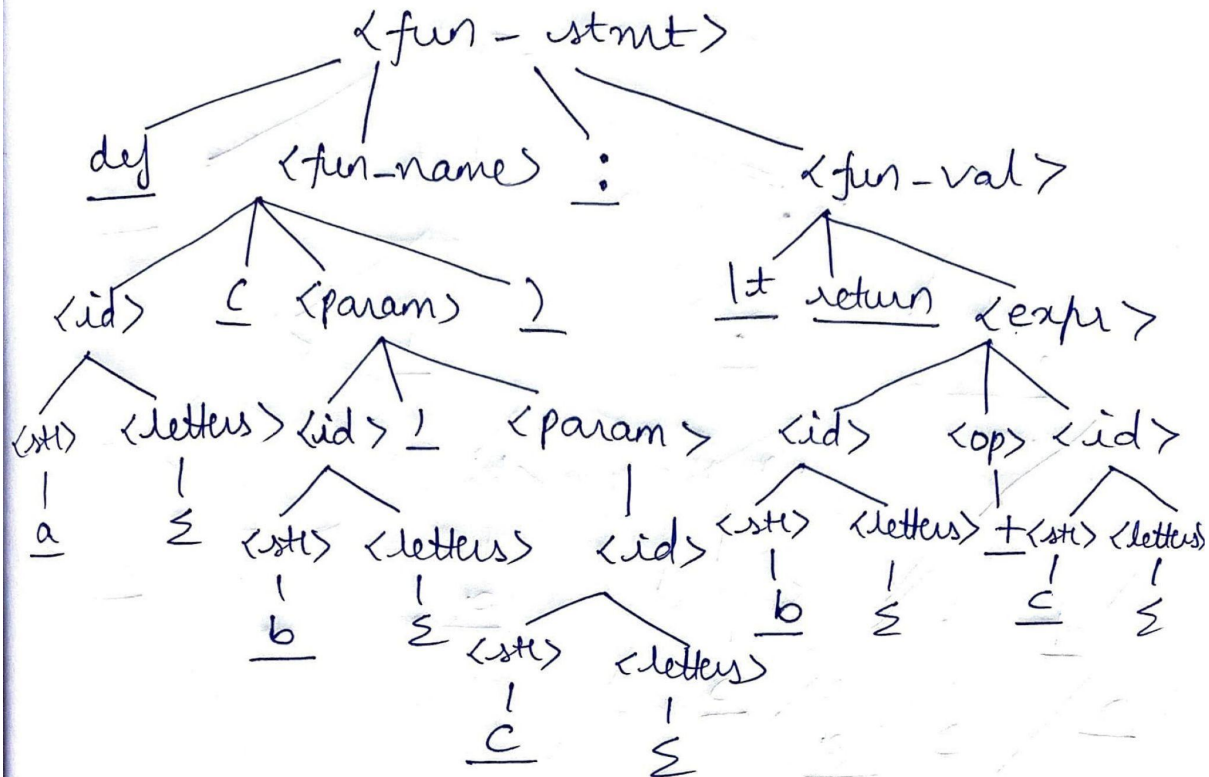
d. Parse tree for a list :

$ls = [1, "s", 4.2]$



e. Parse tree for a function :

```
def a(b, c):  
    return b + c
```



Semantics analysis:-

Semantic analysis is the task of making sure that the declarations and statements of a program are semantically correct, i.e, that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.

Semantics for Python Language:

- **Indentation:** For python, whitespace matters i.e. python blocks of code are denoted by indentation mostly four spaces. It will give you an error if there is no space before the statements. Python blocks of code are always preceded by a colon (:) on the previous line.
- Whitespace within lines doesn't matter.
Ex : x = 1+2, x = 1 + 2 - Both ways are the same, it doesn't matter.

- Python has types; however, the types are linked not to the variable names but to the objects themselves.
- In python, we cannot concatenate int and string values in the same print statement.
- In python, there is no need to declare 'i' in - for i in range.
- There is no need to pre-define a variable in python. Variables are created at the moment you first assign a value to them.
- Python has so many libraries that you can use to select and save time on the initial cycle of development.

Ex: matplotlib for plotting charts and graphs

BeautifulSoup for HTML parsing and XML

Semantics for HTML Language:

- There are many semantic elements in HTML, used to define different parts of a web page.
Ex : <article>, <table>, <form>
- In HTML, we have to define open and close tags for proper execution, but it is an exception for image tags.
- We can use any tags in the HTML format.
- The <section> element defines a section in a document.
- Many web sites contain HTML code like: <div id="nav"> <div class="header"> <div id="footer"> to indicate navigation, header, and footer.
- We can style our page by using classes in HTML tags.

Ex:

```
<body style="background-color:powderblue;">
<h1>This is a heading</h1>
<p>This is a paragraph.</p>
</body>
```


Syntax of Target Language :

Example Program :

```
<python>  
    a = b*c+d  
<print> a </print>  
</python>
```

Intermediate Code Generation :

The program is converted into simple machine-independent instructions. This is similar to the target machine code. It specifies the no. of registers to store the variables. This code can be optimized to avoid unreachable parts of code, parts that are not modified inside the loop, multiplications by 1, and addition by 0. The intermediate code for the above program is :

```
t1 = b*c  
t2 = d + t1  
a = t2
```

Target Language :

The equivalent target code is generated by choosing the suitable instructions and memory locations for the variables. The target code for the above program is :

```
LD R1, b  
MUL R1, c  
LD R2, d  
ADD R2, R1  
MOV a, R2
```

LEX and YACC Programs :

a. List :

i. LEX :

```
%{  
#include "list.tab.h"  
%}  
  
%%  
  
[0-9]* {yylval=atoi(yytext);  
return NUMBER;  
}  
  
[0-9]*\.[0-9]* {yylval=atof(yytext);  
return DECIMAL;  
}  
  
\'[a-zA-Z]*\' {yylval=yytext;  
return STRING;  
}  
  
[a-zA-Z][a-zA-Z0-9]* {yylval=yytext;  
return IDENTIFIER;  
}  
  
\n return 0;  
  
. return yytext[0];  
  
%%
```

ii. YACC

```
%{  
#include<stdio.h>  
%}  
  
%token NUMBER
```

%token DECIMAL
%token STRING
%token IDENTIFIER

%%

```
S: L { printf("correct"); }  
;
```

```
L: IDENTIFIER='V' { $$ = $1 = $3; }  
;
```

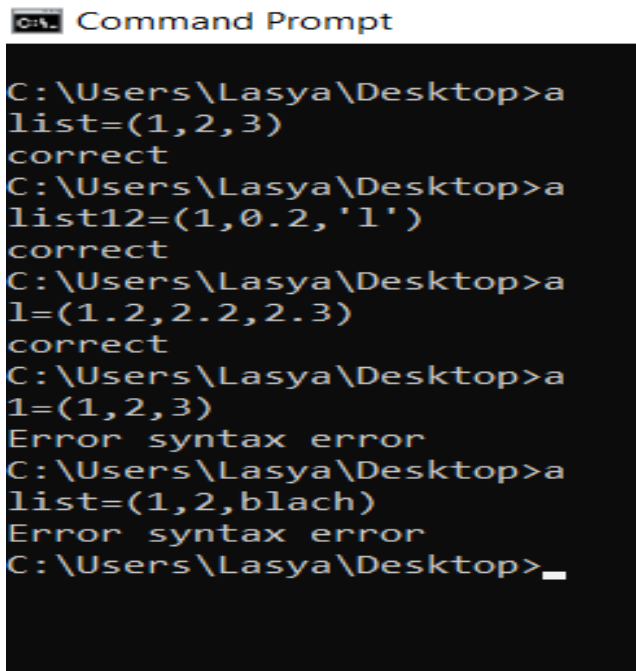
```
V: '('A')' { $$ = ( $2 ); }  
;
```

```
A :  
| STRING,'A' { $$ = $1,$3; }  
| NUMBER,'A' { $$ = $1,$3; }  
| DECIMAL,'A' { $$ = $1,$3; }  
| STRING { $$ = $1; }  
| NUMBER { $$ = $1; }  
| DECIMAL { $$ = $1; }  
;
```

%%

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

Output :



```
C:\> Command Prompt
C:\Users\Lasya\Desktop>a
list=(1,2,3)
correct
C:\Users\Lasya\Desktop>a
list12=(1,0.2,'1')
correct
C:\Users\Lasya\Desktop>a
l=(1.2,2.2,2.3)
correct
C:\Users\Lasya\Desktop>a
1=(1,2,3)
Error syntax error
C:\Users\Lasya\Desktop>a
list=(1,2,blach)
Error syntax error
C:\Users\Lasya\Desktop>_
```

b. While :

i. LEX :

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

%%

"while" { return WHILE; }

"else" { return ELSE; }

"<print>" { return STARTPRINT; }

"</print>" { return ENDPRINT; }

[0-9]* { return NUMBER; }

[0-9]\.[0-9] { return DECIMAL; }
```

```

\[a-zA-Z]*\ { return STRING; }

[a-zA-Z][a-zA-Z0-9]* { return IDENTIFIER; }

["<" ">" "<=" ">=" "=" "==" "<>"] { return RELOP; }

\n return 0;

. return yytext[0];

%%

```

ii. YACC :

```

%{
#include<stdio.h>
%}

%token NUMBER
%token DECIMAL
%token STRING
%token IDENTIFIER
%token RELOP
%token WHILE
%token STARTPRINT
%token ENDPRINT
%token ELSE

%%

L: WHILE '(' C ')' ':' P E { printf("correct"); }
;

C: T
| T RELOP C
;

T: IDENTIFIER
| N
;

```

```

N: NUMBER
| DECIMAL
;

E:
| ELSE ':' P
;
P: STARTPRINT '(' T ')' ENDPRINT
;

%%

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

```

Output :

```

C:\Users\Lasya\Desktop>a
while(i<j):<print>(i)</print>
correct
C:\Users\Lasya\Desktop>a
while(i<j):<print>(i)</print>else:<
correct
C:\Users\Lasya\Desktop>a
while(i):<print>(i)</print>
correct
C:\Users\Lasya\Desktop>a
ehile(a):
Error syntax error
C:\Users\Lasya\Desktop>a
while(1<2):<print>(2)</print>
correct
C:\Users\Lasya\Desktop>a
while (a):<print>1</print>
Error syntax error
C:\Users\Lasya\Desktop>

```

c. Function :

i. LEX :

```
%{  
#include "function.tab.h"  
%}  
  
%%  
  
def {yylval=yytext;  
return DEF;  
}  
  
print {yylval=yytext;  
return PRINT;  
}  
  
"return" {yylval=yytext;  
return RETURN;  
}  
  
[0-9]* {yylval=atoi(yytext);  
return NUMBER;  
}  
  
[0-9]\.[0-9] {yylval=atof(yytext);  
return DECIMAL;  
}  
  
\[a-zA-Z]*\ {yylval=yytext;  
return STRING;  
}  
  
[a-zA-Z][a-zA-Z0-9]* {yylval=yytext;  
return IDENTIFIER;  
}  
  
[+ | - | * | % | // | < | > | ==] {yylval=yytext;
```

```

return OP;
}

\n return 0;

. return yytext[0];

%%

```

ii. YACC :

```

%{
#include<stdio.h>
#include<string.h>
}%

%token NUMBER
%token DECIMAL
%token STRING
%token IDENTIFIER
%token OP
%token DEF
%token PRINT
%token RETURN

%%

S: F { printf("correct"); }
;

F: DEF N V { $$ = strcat(strcat($1,$2),$3); }
;

N: IDENTIFIER '(' P ')' { $$ = strcat(strcat($1,'('),strcat($3,')')); }
;

P : {}
| STRING 'P' { $$ = $1,$3; }
| NUMBER 'P' { $$ = $1,$3; }
| DECIMAL 'P' { $$ = $1,$3; }

```



```

| IDENTIFIER','P { $$ = $1,$3; }
| STRING { $$ = $1; }
| NUMBER { $$ = $1; }
| DECIMAL { $$ = $1; }
| IDENTIFIER { $$ = $1; }
;

V : PRINT '(' E ')' { $$ = strcat(strcat($1,'('),strcat($3,')')); }
| RETURN E { $$ = strcat($1,$2); }
;

E : IDENTIFIER { $$ = $1; }
| IDENTIFIER OP IDENTIFIER { $$ = strcat(strcat($1,$2),$3); }
;

%%

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

```

Insights :

The designing of the compiler for this language includes a series of implementation methods. The implementation is similar to that of other compiler-based languages. In this, we combined the features of python and HTML to provide the benefit of the easy to learn nature of python and website constructing language like HTML. First, we designed the language constructs like tokens, patterns, and other functional units. Second, we constructed the CFG's of these constructs along with a parse tree for various instances. We worked on the semantics of the language and generated the intermediate code and target code for an expression program.

Issues Faced and Methods used to resolve them :

Initially, we tried constructing the parse trees using the Recursive Decent Parser. But after analysing the difficulties like left most recursion, left factorization while constructing a CFG for a top-down parser, and the difficulty to write code for large CFG's we substituted RDP with LALR parser. We constructed LALR parsers for the list, while, and function components in our language using YACC. We faced a few challenges while writing code for these constructs. We succeeded in constructing the parse tree for list, and while loop.