BCSE-III, 2023

JADAVPUR UNIVERSITY

COMPILER DESIGN PROJECT

PROJECT-3

GROUP-A2

ROLL	NAME
002010501030	Md Mufassir Azam
002010501039	Amartya Chakroborty
002010501051	Anannyo Dey
002010501052	Kabir Raj Singh

Construct a simple C-like language with

Data Type: Basic data types are character(char), integer(int), and floating point(float).

There is **no declaration statement** and the datatype of a variable is determined at the time of assignment of a value to it (similar to Python).

Input and Output statements are in the form **get x** and **put x**.

Loop Constructs: **do-while**, **nested loops** are supported.

Relational operators supported in the loop construct are {.gt.,.lt.,.ge.,.le.}, i.e. greater than, less than, greater than equal to, and less than equal to.

Arithmetic operators supported are {+,-,++ and --} i.e. addition, subtraction, increment, and decrement. Operations can only be done on integer variables.

Assignment operator '=' is supported.

The only function is **main()**, there is no other function. The **main()** function may contain arguments, but no return statements.

PART -I: Construct a CFG for this language.

PART -II: Write a lexical analyzer to scan the stream of characters from a program written in the above language and generate a stream of tokens.

PART -III: Maintain a symbol table with appropriate data structures.

PART -IV: Write a top-down parser for this language (modules include left recursion removal, FIRST,FOLLOW, parsing table construction and parsing).

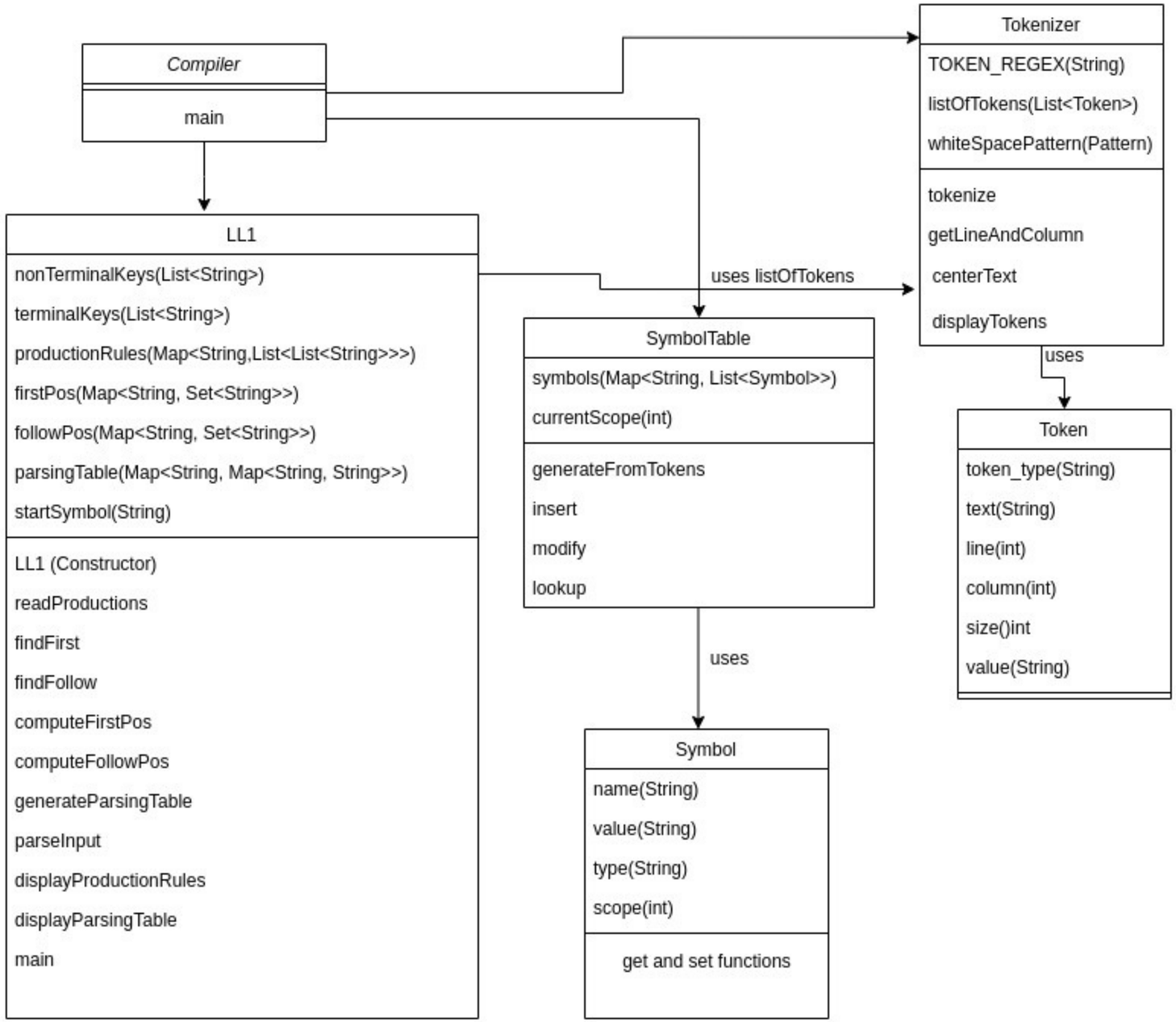
CHALLENGES

CFG CONSTRUCTION FOR LL(1)

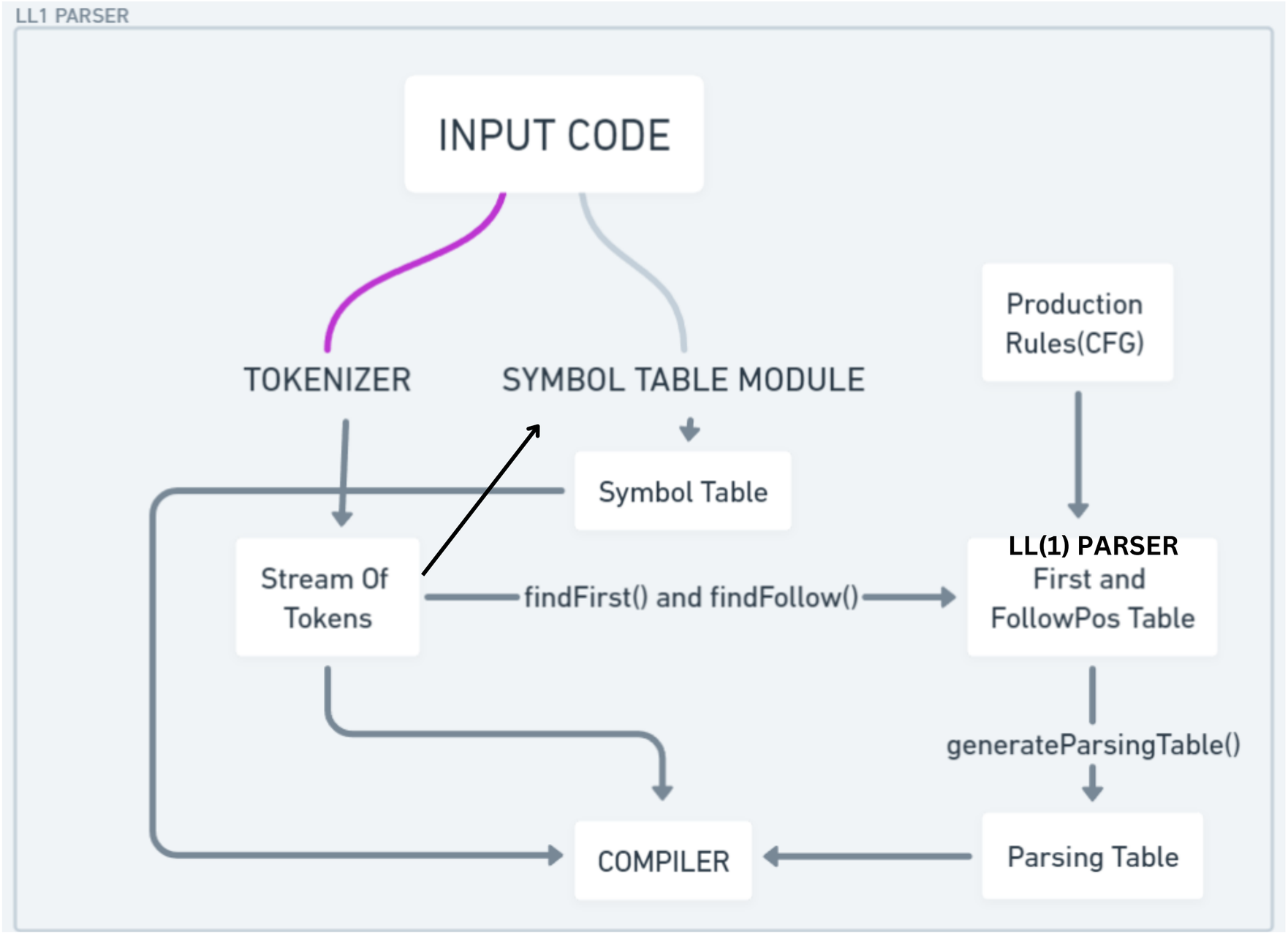
LL(1) parser itself is very simple, but this makes constructing CFG for it very difficult. Special care needs to be taken to use left factoring and remove left recursion.

IMPLICIT TYPE ASSIGNMENT

Inferring the data type of variables simply from their assigned values is very difficult without explicitly knowing it at time of declaration. We had to find the mapping between identifiers and assigned values


Compiler
main
Tokenizer
TOKEN_REGEX(String)
listOfTokens(List<Token>)
whiteSpacePattern(Pattern)
tokenize
getLineAndColumn
centerText
displayTokens
LL1
nonTerminalKeys(List<String>)
terminalKeys(List<String>)
productionRules(Map<String,List<List<String>>>)
firstPos(Map<String, Set<String>>)
followPos(Map<String, Set<String>>)
parsingTable(Map<String, Map<String, String>>)
startSymbol(String)
LL1 (Constructor)
readProductions
findFirst
findFollow
computeFirstPos
computeFollowPos
generateParsingTable
parseInput
displayProductionRules
displayParsingTable
main
uses listOfTokens
SymbolTable
symbols(Map<String, List<Symbol>>)
currentScope(int)
generateFromTokens
insert
modify
lookup
uses
Token
token_type(String)
text(String)
line(int)
column(int)
size()int
value(String)
uses
Symbol
name(String)
value(String)
type(String)
scope(int)
get and set functions

FLOW DIAGRAM



FORGOT TO ASK?

We added it without asking for it...

1. Accepts string type
2. Relational operators like == as .eq, != as .ne
3. Arithmetic operators like *, / and %
4. Other assignment operators like +=, -=, *=, /=
5. If and If-else blocks
6. Nested while loops
7. Scope and value detection for variables in the symbol table
8. Error checks for input CFG if LL1 parsable, reassignment in same scope of variables, etc.
9. List of command line parameters passed to main()
10. Compound arithmetic operations like a + (6 * c - b)

PART –I: Construct a CFG for this language.

program -> function
function -> def main (params) block
params -> , params | param params | EPSILON
param -> id
block -> { stmtlist }
stmtlist -> stmt stmtlist | EPSILON
stmt -> assignstmt | getStmt | putStmt | whilestmt | dowhilestmt | ifstmt | unaryexpr
assignstmt -> id eval
eval -> assignop expr delim | unaryop delim
getStmt -> get id delim
putStmt -> put expr delim
dowhilestmt -> do block while (expr) delim
whilestmt -> while (expr) block
ifstmt -> if (expr) block ifstmt2
ifstmt2 -> else block | EPSILON
expr -> addexpr expr2

expr2 -> relop addexpr | EPSILON
addexpr -> term addexpr2
addexpr2 -> addop term addexpr2 | EPSILON
term -> factor term2
term2 -> mulop factor term2 | EPSILON
factor -> id | integer_constant | char_constant | string_constant | float_constant | (expr)
relexpr -> addexpr relop addexpr
unaryexpr -> unaryop id delim
addop -> + | -
mulop -> * | / | %
assignop -> = | += | -= | *= | /=
relop -> .lt | .gt | .le | .ge | .eq | .ne
unaryop -> ++ | --
delim -> ;

PART –II: Write a lexical analyzer to scan the stream of characters from a program written in the above language and generate a stream of tokens.

Different Type of Tokens

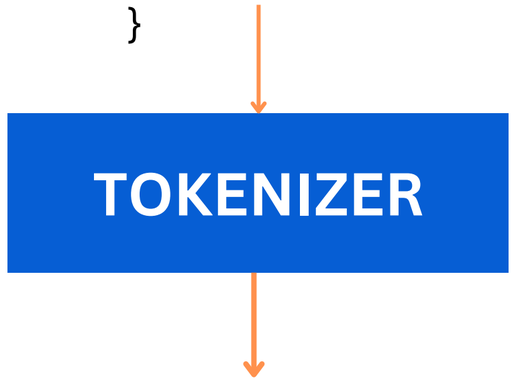
TYPE	REGEX	EXAMPLE
Operator	(\\.eq|=|\\.le|\\.ge|\\.ne|\\.lt|\\.gt|\\+=|-=|*=|/=|\\+\\+|--|\\+|-|*|\\/|%)	.gt.,.eq.,.le., +., .+=.,./=,++,--
Punctuator	(\\(|\\)|\\{|\\}|\\,)	(,) , { , } , ,
Delimiter	;	;
Constant	(\\d+\\.\\d+|\\d+|\'.\'|\".*\")	1, 23.2 , "hello" , 'a'
Keyword	(get|put|def|do|if|else|main|while)	get, put, return, int, char , string, main
Identifier	[a-zA-Z_]\\w*	a , b , var_a , var_A

Structure of Token

PARAMETER	DESCRIPTION
Token Type	Information regarding type of token [identifier , constant , operator , delimiter, keyword, etc.]
Lexeme	Textual value of object. In case of constant , it will be overrided with [integer_constant, char_constant, string_constant]
Line	Line no where the token appears
Column	Column no [position] where the token found in specific line
Size	Size of the token
Value	This field used to hold value of constants. In other case it is null

Tokenizer

```
def main( x ) {
    put x ;
}
```



TYPE	LEXEME	LINE	COLUMN	SIZE	VALUE
keyword	def	1	2	3	null
keyword	main	1	6	4	null
punctuator	(	1	10	1	null
identifier	id	1	12	1	x
punctuator	)	1	14	1	null
punctuator	{	1	16	1	null
keyword	put	2	6	3	null
identifier	id	2	10	1	x
delimiter	;	2	12	1	null
punctuator	}	2	2	1	null

PART –III: Maintain a symbol table with appropriate data structures.

Implemented using HashMap

Map<String, List<Symbol>> symbols

In the map we have-> NAME -> Name,Value,Type,Scope

Symbol is defined as

```
private String name;
private String value;
private String type;
private int scope;
```

```
def main(){
   a = 456;
   b = 456.3;
   {
      a = 'a';
       {
          a ="hello";
       }
   }
}
```

```
a = ( type : int, value : 456, scope: 1)
b = ( type : float, value : 456.3, scope: 1)
a = ( type : char, value : 'a', scope: 2)
a = ( type : string, value : "hello" , scope: 3)
```

PART -IV: Write a top-down parser for this language (modules include left recursion removal, FIRST,FOLLOW, parsing table construction and parsing)

DEMO OF LL(1) ITEM SET CONSTRUCTION

For this purpose, we are taking a small grammar

EXAMPLE GRAMMAR

S -> E
E -> T E'
T -> F T'
F -> (E) | id
E' -> + T E' | EPSILON
T' -> * F T' | EPSILON

FIRST & FOLLOW SETS

Symbol	FIRST	FOLLOW
S	[(, id]	[$]
E	[(, id]	[$,)]
T	[(, id]	[$,), +]
F	[(, id]	[$,), *, +]
E'	[+, EPSILON]	[$,)]
T'	[*, EPSILON]	[$,), +]

PARSING `id * id` USING THE GRAMMAR DEFINED PREVIOUSLY

[id, *, id, $] → INPUT

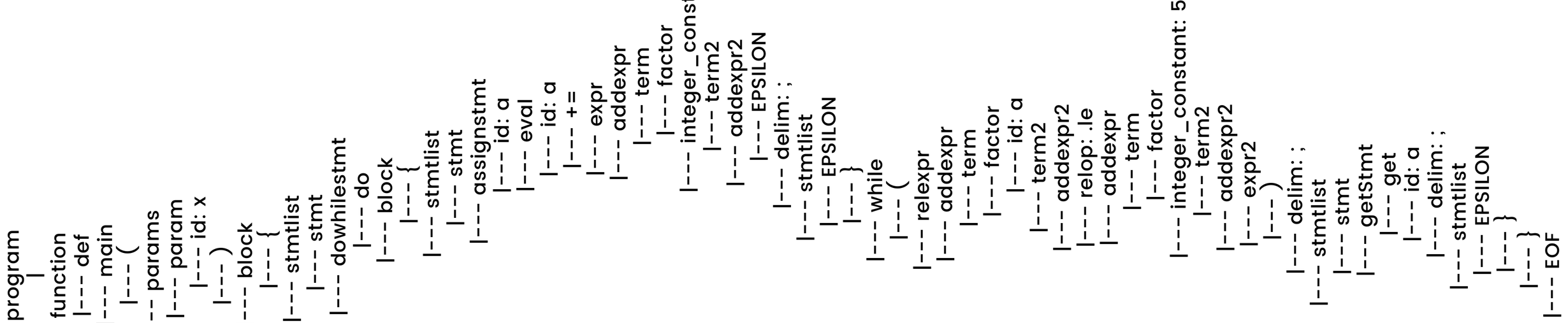
ACTION	STACK
PUSHED S#0	[$, E]
PUSHED E#0	[$, E', T]
PUSHED T#0	[$, E', T', F]
PUSHED F#1	[$, E', T', id]
MATCHED id	[$, E', T']
PUSHED T'#0	[$, E', T', F, *]
MATCHED *	[$, E', T', F]
PUSHED F#1	[$, E', T', id]
MATCHED id	[$, E', T']
PUSHED EPSILON	[$, E']
PUSHED EPSILON	[$]
MATCHED $	[]

PARSING

Parsing Successful ! Code is Accepted → RESULT

```
def main( x )
{    a = 0 ;
     do
     { a += 2; } while( a .le 5 ) ;
     get a ;
}
```

PARSE TREE FOR EXAMPLE CODE FOR OUR CFG



Tokenizer.tokenize(code)

Converts the code into stream of tokens using Regex Pattern Matching from slide #11. It also checks for illegal keywords. Replaces variables by id.

SymbolTable.generateFromTokens(tokens)

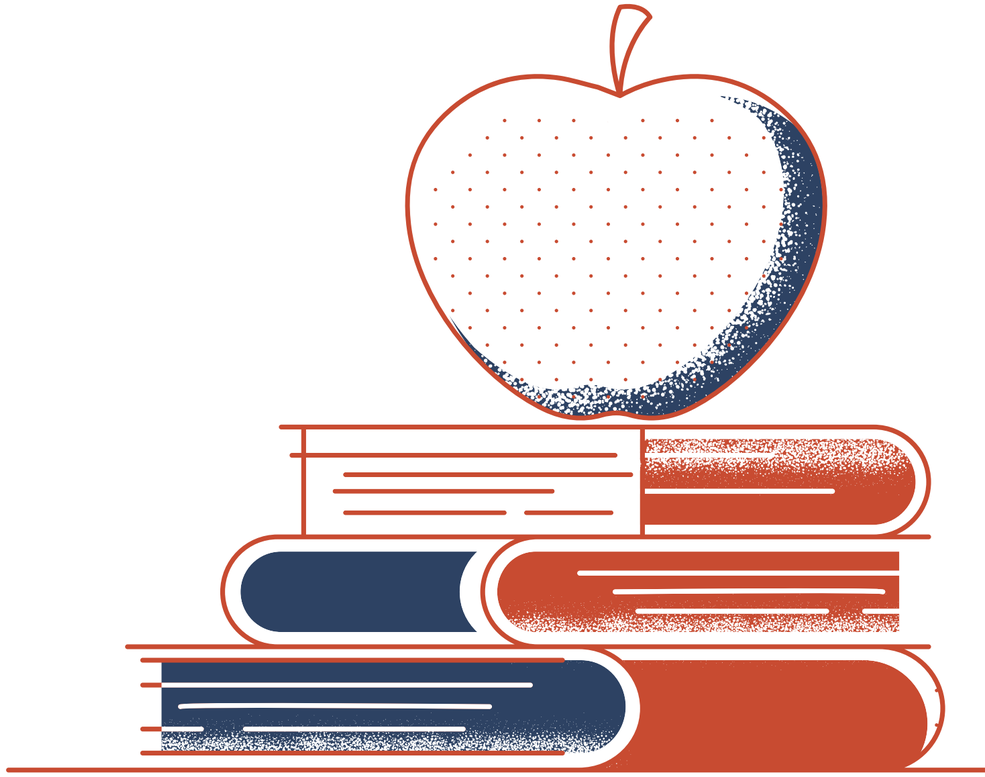
From stream of tokens generates symbol table, also assigns type to the variables implicitly inferred from the value assigned. Also maintains scope, and makes multiple entries for same variable in case of reassignments in different scopes

ll1.parseInput(output.toString())

Parses the stream of tokens(InputQueue) using the ParsingTable generated by generateparsingTable(). Displays the actions taken by the parser and the contents of the ParseStack, at every step.

INPUT CODE

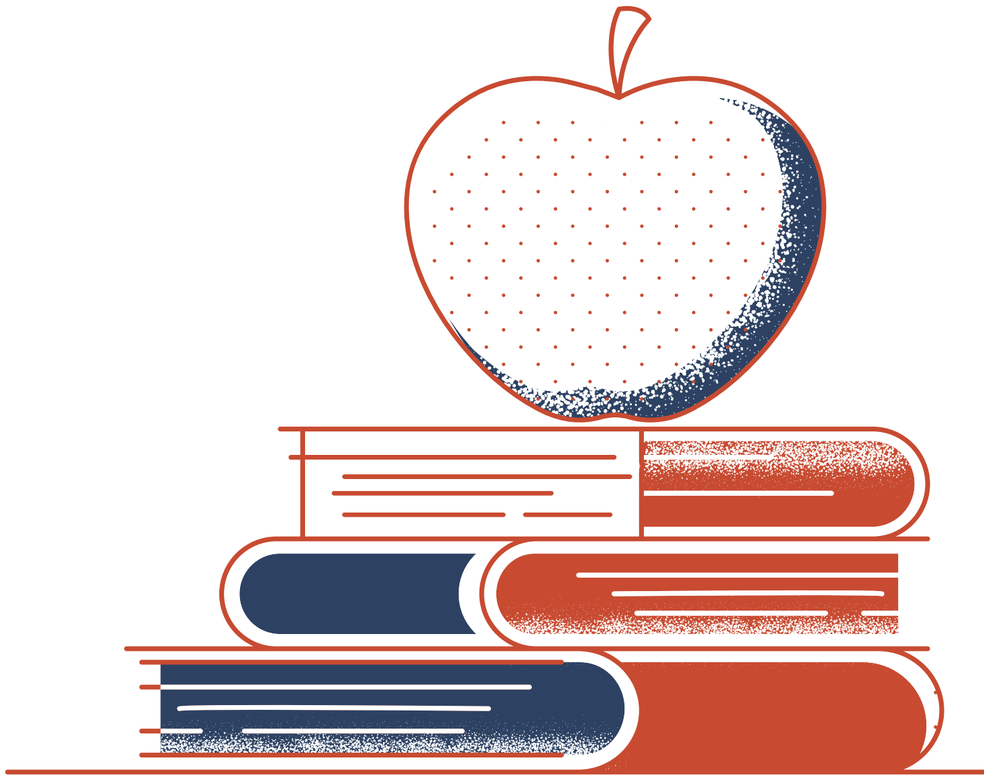
```
def main( x , y ){
    a = 0 ;
    c = 'a' ;
    b = 6.5 ;
    do
    {
        if (a .ne 5)
        {
            a = b + (6 * c - a) ;
            b += 6;
        }
        else
        {
            a++;
        }

        while( b .eq 6)
        {
            a *= b - 7;
        }
    } while( a .le 5 ) ;
    get a ;
    put "hello world" ;
}
```



OUTPUT

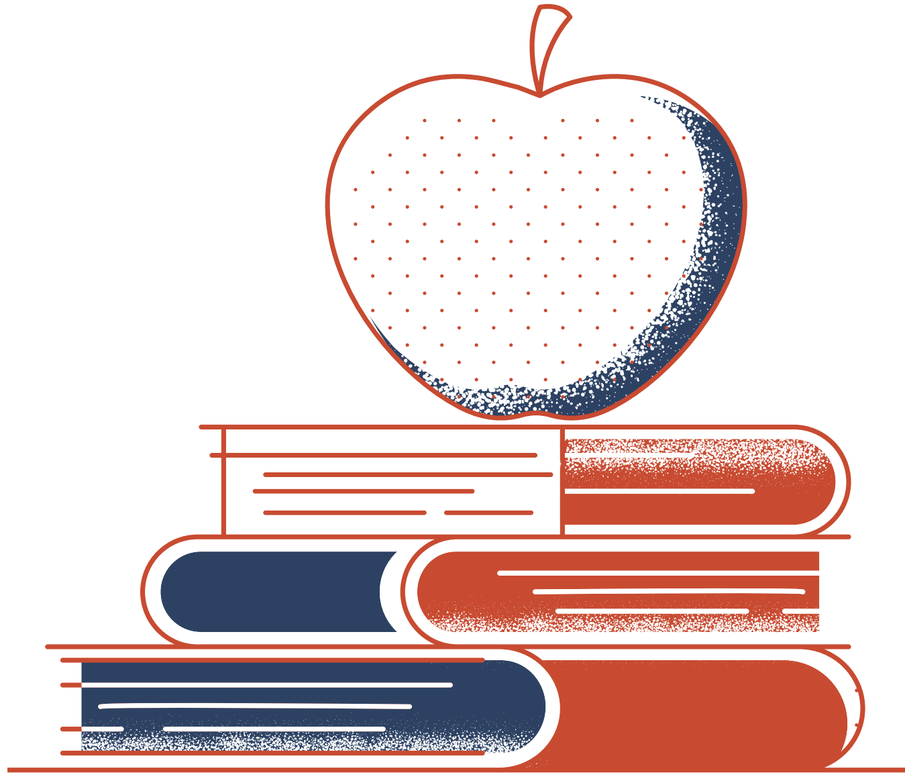
> Tokens:

TYPE	TEXT	LINE	COLUMN	SIZE	VALUE
keyword	def	1	2	3	null
keyword	main	1	6	4	null
punctuator	(	1	10	1	null
identifier	id	1	12	1	x
punctuator	,	1	14	1	null
identifier	id	1	16	1	y
punctuator	)	1	18	1	null
punctuator	{	1	20	1	null
identifier	id	2	6	1	a
operator	=	2	8	1	null
constant	integer_constant	2	10	1	0
delimiter	;	2	12	1	null
identifier	id	3	6	1	c
operator	=	3	8	1	null
constant	char_constant	3	10	3	'a'
delimiter	;	3	14	1	null
identifier	id	4	6	1	b
operator	=	4	8	1	null
constant	float_constant	4	10	3	6.5
delimiter	;	4	14	1	null
keyword	do	5	6	2	null
punctuator	{	6	6	1	null
keyword	if	7	10	2	null
punctuator	(	7	13	1	null
identifier	id	7	14	1	a
operator	.ne	7	16	3	null
constant	integer_constant	7	20	1	5
punctuator	)	7	21	1	null
punctuator	{	8	10	1	null
identifier	id	9	14	1	a
operator	=	9	16	1	null
identifier	id	9	18	1	b
operator	+	9	20	1	null



TOKENS GENERATED

punctuator	(	9	22	1	null
constant	integer_constant	9	23	1	6
operator	*	9	25	1	null
identifier	id	9	27	1	c
operator	-	9	29	1	null
identifier	id	9	31	1	a
punctuator	)	9	32	1	null
delimiter	;	9	34	1	null
identifier	id	10	14	1	b
operator	+=	10	16	2	null
constant	integer_constant	10	19	1	6
delimiter	;	10	20	1	null
punctuator	}	11	10	1	null
keyword	else	12	10	4	null
punctuator	{	13	10	1	null
identifier	id	14	14	1	d
operator	=	14	16	1	null
constant	string_constant	14	18	17	"compiler_design"
delimiter	;	14	36	1	null
identifier	id	15	14	1	a
operator	++	15	15	2	null
delimiter	;	15	18	1	null
punctuator	}	16	10	1	null
keyword	while	18	10	5	null
punctuator	(	18	15	1	null
identifier	id	18	17	1	b
operator	.eq	18	19	3	null
constant	integer_constant	18	23	1	6
punctuator	)	18	24	1	null
punctuator	{	19	10	1	null
identifier	id	20	14	1	a
operator	*=	20	16	2	null
identifier	id	20	19	1	b
operator	-	20	21	1	null
constant	integer_constant	20	23	1	7
delimiter	;	20	25	1	null

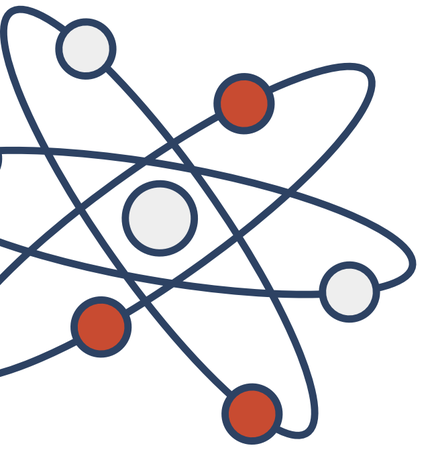


SYMBOL TABLE

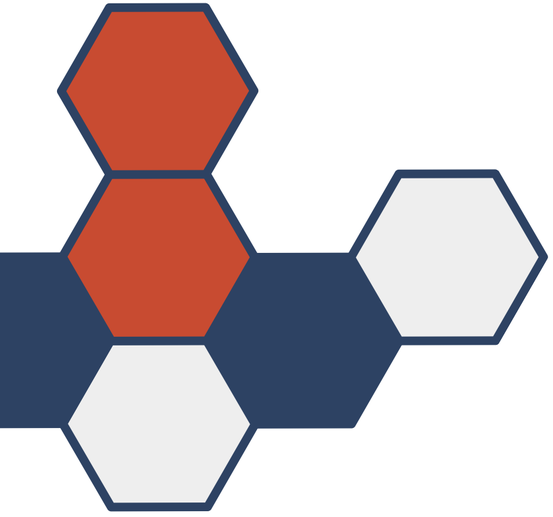
```
Symbol Table:
a = ( type : Integer, value : 0, scope: 1)
b = ( type : Float, value : 6.5, scope: 1)
c = ( type : Character, value : 'a', scope: 1)
d = ( type : String, value : "compiler_design", scope: 3)
x = ( type : _PARAMS_, value : _PARAMS_, scope: 0)
y = ( type : _PARAMS_, value : _PARAMS_, scope: 0)
--------------------
```

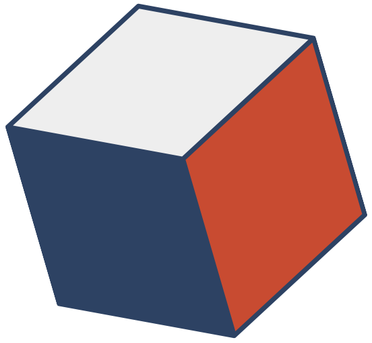
FIRST/FOLLOW SET

Symbol	FIRST	FOLLOW
program	[def]	[$]
function	[def]	[$]
params	[id, ,, EPSILON]	[)]
param	[id]	[), id, ,]
block	[{]	[$, else, get, id, do, while, if, }, put]
stmtlist	[get, id, do, while, if, put, EPSILON]	[}]
stmt	[get, id, do, while, if, put]	[get, id, do, while, if, }, put]
assignstmt	[id]	[get, id, do, while, if, }, put]
eval	[-=, ++, --, +=, *=, =, /=]	[get, id, do, while, if, }, put]
getStmt	[get]	[get, id, do, while, if, }, put]
putStmt	[put]	[get, id, do, while, if, }, put]
dowhilestmt	[do]	[get, id, do, while, if, }, put]
whilestmt	[while]	[get, id, do, while, if, }, put]
ifstmt	[if]	[get, id, do, while, if, }, put]
ifstmt2	[else, EPSILON]	[get, id, do, while, if, }, put]
expr	constant, char_constant, string_constant, integer_constant,	[), ;]
expr2	[.ne, .lt, .le, .eq, .gt, .ge, EPSILON]	[), ;]
addexpr	constant, char_constant, string_constant, integer_constant,	[++, --, .ne, .lt, .le,), .eq, .gt, ;, .ge]
addexpr2	[+, -, EPSILON]	[++, --, .ne, .lt, .le,), .eq, .gt, ;, .ge]
term	constant, char_constant, string_constant, integer_constant,	[++, --, .ne, .lt, .le,), .eq, +, .gt, ;, .ge, -]
term2	[*, /, EPSILON]	[++, --, .ne, .lt, .le,), .eq, +, .gt, ;, .ge, -]
factor	constant, char_constant, string_constant, integer_constant,	[++, --, .ne, .le,), *, +, .ge, -, /, .lt, .eq, .gt, ;]
relexpr	constant, char_constant, string_constant, integer_constant,	[)]
unaryexpr	constant, char_constant, string_constant, integer_constant,	[]
addop	[+, -]	[float_constant, char_constant, string_constant, integer_constant, (, id]
mulop	[*, /]	[float_constant, char_constant, string_constant, integer_constant, (, id]
assignop	[-=, +=, *=, =, /=]	[float_constant, char_constant, string_constant, integer_constant, (, id]
relop	[.ne, .lt, .le, .eq, .gt, .ge]	[float_constant, char_constant, string_constant, integer_constant, (, id]
unaryop	[++, --]	float_constant, char_constant, string_constant, integer_constant, (, id, ;]
delim	[;]	[get, id, do, while, if, }, put]



Parse Stack Tree

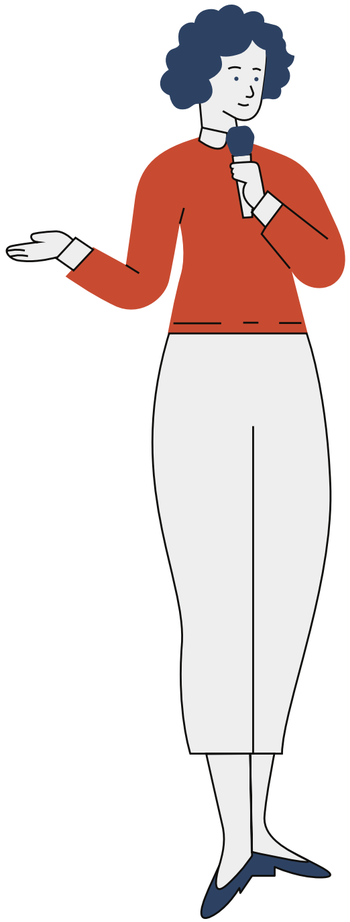




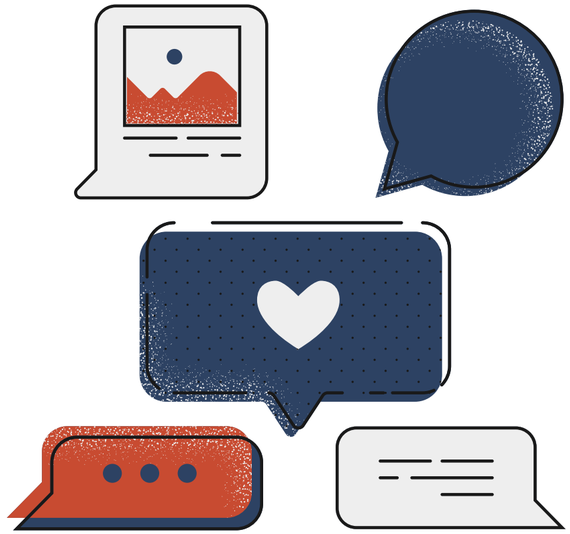
```
> Parsing Tokens :
ACTION          STACK
PUSHED program_0        [$, function]
PUSHED function_0   [$, block, ), params, (, main, def]
MATCHED def     [$, block, ), params, (, main]
MATCHED main        [$, block, ), params, (]
MATCHED (       [$, block, ), params]
PUSHED params_1     [$, block, ), params, param]
PUSHED param_0      [$, block, ), params, id]
MATCHED id      [$, block, ), params]
PUSHED params_0     [$, block, ), params, ,]
MATCHED ,       [$, block, ), params]
PUSHED params_1     [$, block, ), params, param]
PUSHED param_0      [$, block, ), params, id]
MATCHED id      [$, block, ), params]
PUSHED EPSILON      [$, block, )]
MATCHED )       [$, block]
PUSHED block_0      [$, }, stmtlist, {]
MATCHED {       [$, }, stmtlist]
PUSHED stmtlist_0       [$, }, stmtlist, st
                .
                .
                .
MATCHED put     [$, }, stmtlist, delim, expr]
PUSHED expr_0   [$, }, stmtlist, delim, expr2, addexpr]
PUSHED addexpr_0    [$, }, stmtlist, delim, expr2, addexpr2, term]
PUSHED term_0   [$, }, stmtlist, delim, expr2, addexpr2, term2, factor]
PUSHED factor_3     [$, }, stmtlist, delim, expr2, addexpr2, term2, string_constant]
MATCHED string_constant  [$, }, stmtlist, delim, expr2, addexpr2, term2]
PUSHED EPSILON      [$, }, stmtlist, delim, expr2, addexpr2]
PUSHED EPSILON      [$, }, stmtlist, delim, expr2]
PUSHED EPSILON      [$, }, stmtlist, delim]
PUSHED delim_0      [$, }, stmtlist, ;]
MATCHED ;       [$, }, stmtlist]
PUSHED EPSILON      [$, }]
MATCHED }       [$]
MATCHED $       []
Parsing Successful ! Code is Accepted
```

But what happens when there is erroneous code?

Let's find out by trying to parse this code with error->



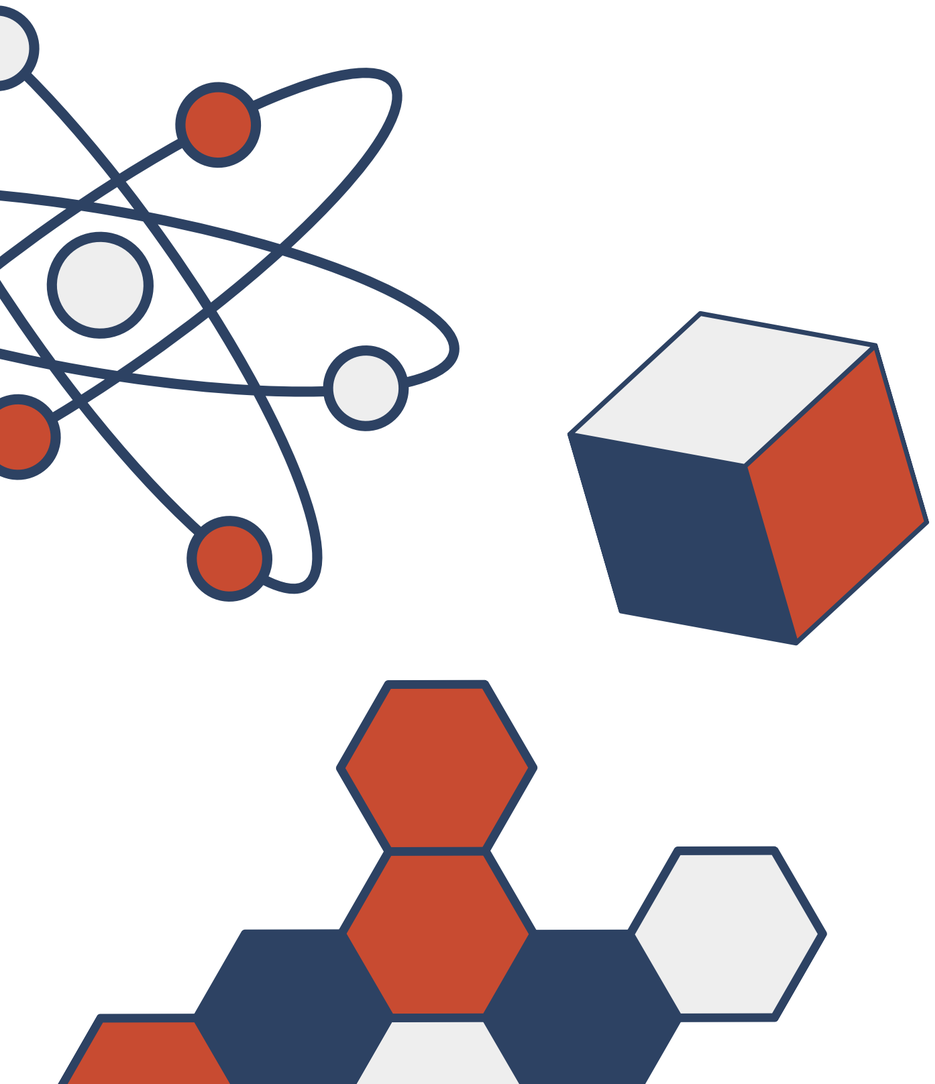
```
def main( ){
    a = 0 ;
    .le ;
    put "hello world" ;
}
```



THE PARSING FAILS AND WE GET THE ERROR AS FOLLOWS-

```
        ACTION                          STACK
    PUSHED program_0                [$, function]
   PUSHED function_0          [$, block, ), params, (, main, def]
    MATCHED def               [$, block, ), params, (, main]
    MATCHED main                 [$, block, ), params, (]
     MATCHED (                    [$, block, ), params]
    PUSHED EPSILON                   [$, block, )]
     MATCHED )                         [$, block]
    PUSHED block_0                [$, }, stmtlist, {]
     MATCHED {                      [$, }, stmtlist]
   PUSHED stmtlist_0               [$, }, stmtlist, stmt]
    PUSHED stmt_0               [$, }, stmtlist, assignstmt]
  PUSHED assignstmt_0            [$, }, stmtlist, eval, id]
     MATCHED id                    [$, }, stmtlist, eval]
    PUSHED eval_0              [$, }, stmtlist, delim, expr, assignop]
   PUSHED assignop_0           [$, }, stmtlist, delim, expr, =]
     MATCHED =                 [$, }, stmtlist, delim, expr]
    PUSHED expr_0              [$, }, stmtlist, delim, expr2, addexpr]
   PUSHED addexpr_0            [$, }, stmtlist, delim, expr2, addexpr2, term]
    PUSHED term_0              [$, }, stmtlist, delim, expr2, addexpr2, term2, factor]
   PUSHED factor_1             [$, }, stmtlist, delim, expr2, addexpr2, term2, integer_constant]
MATCHED integer_constant       [$, }, stmtlist, delim, expr2, addexpr2, term2]
    PUSHED EPSILON             [$, }, stmtlist, delim, expr2, addexpr2]
    PUSHED EPSILON             [$, }, stmtlist, delim, expr2]
    PUSHED EPSILON                [$, }, stmtlist, delim]
    PUSHED delim_0                 [$, }, stmtlist, ;]
     MATCHED ;                        [$, }, stmtlist]
Parsing Failed ! No production rule found for this conversion stmtlist -> .le
```

WE WERE ABLE TO GENERATE A PARSER WITH THE REQUIRED AND EXTRA ADDED FEATURES TO MIMIC C AND PYTHON LIKE CODE, AND ERROR CHECKS



- No variable definition of the same name at the same scope. The same variable cannot be defined twice.
- If any token is found in code that is not valid, it reports that too.
- For error checking, if a grammar which is not LL(1) parser is passed then it reports that.
- If the CFG is ambiguous or has left recursion, appropriate error messages are shown.

The End

Thank you!

WE REALLY PUT IN OUR BEST EFFORTS FOR THIS!