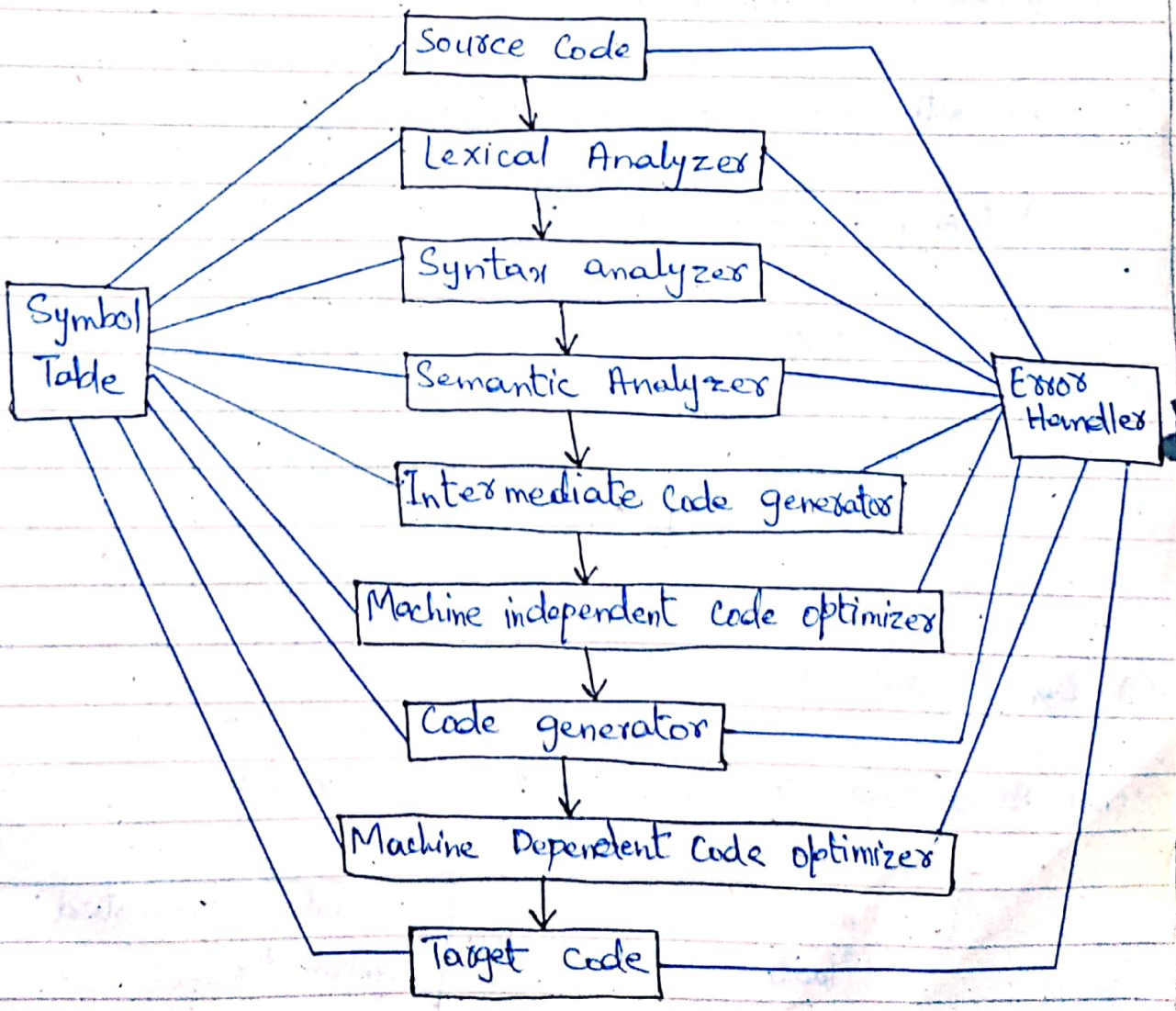
ISA

1. Study
2. Design
3. Construct (Programming)
4. Maintain

Forward Engineering

COMPILER

Phases of compiler:-



① lexical analyzer /Scanner/ Tokenizer:

It represents lexemes in the form of tokens.

(~~Tokens~~ identifier, Token-type)

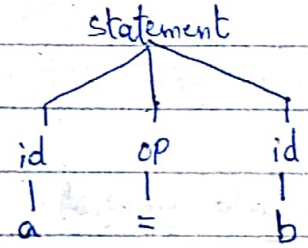
Example: a = b

('a', id)
('=', op)
('b', id)

② Syntax Analyzer / Parser:

Generate parse tree from tokens.

statement ⟶ id op id



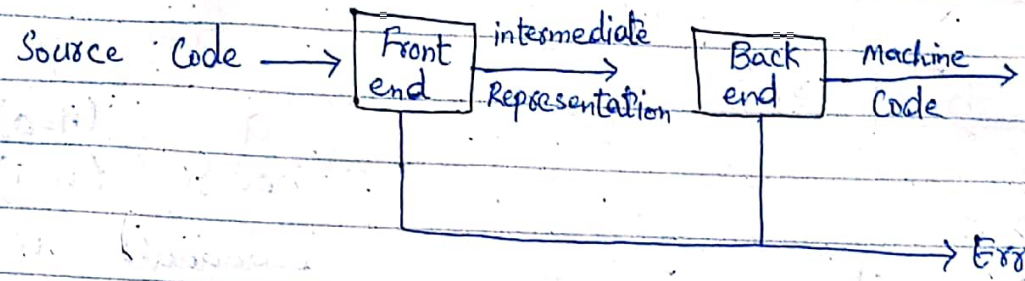
③ ~~Syntax~~ Semantic Analyzer:

Check Parse tree is according to rules or not

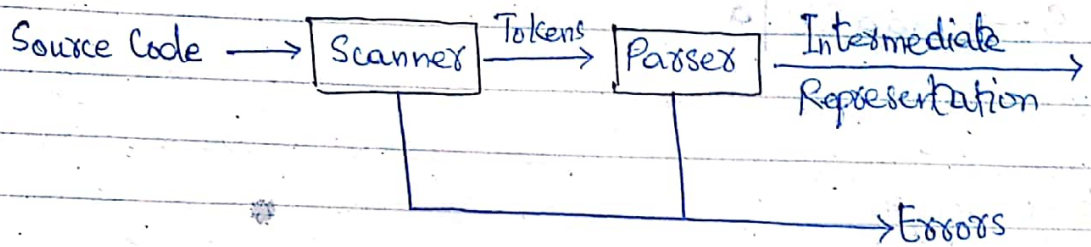
```
int a;
float b;
a = b
```

Generates annoted syntax tree.

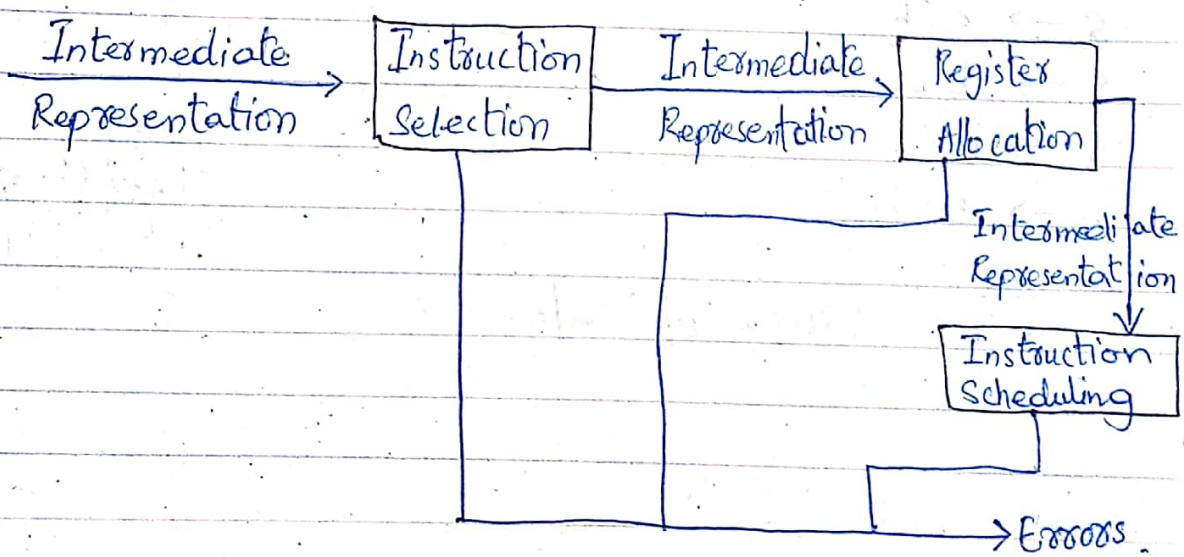
Two Pass Compiler:



Front End:



Back End:



Dec - 2011

CFG

① $\{a^{3n+1} b^n \mid n \geq 0\}$

$S \rightarrow a.$

$S \rightarrow aaaSb$

OR

$S \rightarrow aaaSb \mid a$

a (n=0)
aaab (n=1)
aaaaaaabb (n=2)

② Even length

$S \rightarrow \wedge$

$S \rightarrow aSb$

$S \rightarrow bSa$

$S \rightarrow aSa$

$S \rightarrow bSb$

OR

$S \rightarrow aSa \mid bSb \mid aSb \mid bSa \mid \wedge$

∧
aa
bb
ab
ba
baab
abaabbbaab

③ Odd length

$S \rightarrow aSa \mid bSb \mid aSb \mid bSa \mid a \mid b$

④ $\{x^n y^{2n} \mid n \geq 0\}$

$S \rightarrow xSyy \mid \wedge$

⑤ $\{a^n c^m b^n \mid m \geq 0, n \geq 0\}$

$S \longrightarrow aSb \mid T$

$T \longrightarrow cT \mid \Lambda$

⑥ $\{a^i b^{2i} c^i \mid i \geq 0\}$

$S \longrightarrow abSbc \mid \Lambda$

CFG's to Study through internet : (Assignment)

1. Variable declaration:
2. Arithmetic statement
3. Array declaration
4. Multi dimensional Array Declaration
5. if statement
6. switch case statement
7. for loop
8. while loop
9. Function declaration / Prototype
10. Function definition
11. Class declaration
12. Class instantiation
13. Unions
14. Structures
15. Filing

June-2015

Q) Generate Token set for only underline segments for following C code;

```
VOID main(void)
{
    float num1 = 1.0.1, num2 = 2.0;
    integer fs@fe = 'a', #asd =="afesdsa", *ptr;
    switch (op)
    { case '+':
        num1 += num2;
        break;
      case '-':
        num1 -= num2;
        break;
      // return 3; while (1) { while { } do (I <=5); }
      default:
        num1 +++;
        num3 = "Computer Science";
    }
}
```

Class Part	value Part	Line no.	Class Part	Value Part	Line no.
Character String	VOID	1	Separator	,	3
id	num1	2	invalid id	#asd	3
assignment operator	=	2	Relational operator	==	3
invalid numbers	1.0.1	2	Character String	afesdsa	3
character string	integer	3	Separater	,	3
invalid id	fs@fe	3	Pointer id	*ptr	3
assignment op.	=	3	line termi	;	3
character constant	a	3	Keyword	case	5

Class Part	Value Part	line no.
Character Constant	+	5
arithmetic OP	+=	6
id	num2	9
Keyword	default	12
separator	:	12
invalid Token	+++	13
invalid token	"Computer Science	14

Symbol Table:

S:No.	Values	Data type
1	VOID	character Str.
2	num1	float
3	integer	character str
4	fs@fe	invalied id
5	a	character constant
6	#asd	invalid id
7	afesdsa	character string
8	*ptr	pointer
9	num2	float
10	"Computer Science	invalid token

LL(1) Grammar :

L : Scanning the input from left to right.
L : Producing leftmost derivation.
(1) : One input symbol at each step.

Definition:
"A CFG is called LL(1) if and only if production start with same lefthand side have disjoint selection sets."

Steps :
- Left factored grammar
- Eliminate left recursion.
- First sets
- Follow sets
- Selection sets.
- Parsing Table
- Control Table
- Parsing a string
- Recursive - Descent Parser / Pseudo Code.
- Run a string on a Recursive - Descent Parser.

cont.

- Left Factoring:

$A \rightarrow \alpha\beta_1$
$A \rightarrow \alpha\beta_2$
$\vdots$
$A \rightarrow \alpha\beta_n$
$A \rightarrow \wedge$

starting of right handside (Production) should/must be unique.

Same left hand's production (right hand) should start with unique value.

—OR—

$A \rightarrow \alpha A'$
$A \rightarrow \wedge$
$A' \rightarrow \beta_1$
$A' \rightarrow \beta_2$
$\vdots$
$A' \rightarrow \beta_n$

Example:

$S \rightarrow$ if E then S
$S \rightarrow$ if E then S else S
$S \rightarrow a$
$E \rightarrow b$

Solution:

$S \rightarrow$ if E then S S'
$S \rightarrow a$
$S' \rightarrow$ else S'
$S' \rightarrow \wedge$
$E \rightarrow b$

- Left Recursion Elimination:

$A \xrightarrow{*} A\alpha$

$A \longrightarrow A\alpha$
$A \longrightarrow \beta$ } first step [left factoring]

$A \longrightarrow \beta A'$
$A' \longrightarrow \alpha A'$
$A' \longrightarrow \lambda$

Example:

$E \longrightarrow E+T \mid T$
$T \longrightarrow T*F \mid F$
$F \longrightarrow (E) \mid id$

(OR)

$E \longrightarrow E+T$
$E \longrightarrow T$
$T \longrightarrow T*F$
$T \longrightarrow F$
$F \longrightarrow (E)$
$F \longrightarrow id$

Solution:

$E \longrightarrow TE'$
$E' \longrightarrow +TE' \mid \lambda$
$T \longrightarrow FT'$
$T' \longrightarrow *FT' \mid \lambda$
$F \longrightarrow (E) \mid id$

Indirect left recursion:

Example :

$S \longrightarrow Aa \mid b$

$A \longrightarrow Ac \mid Sd \mid \Lambda$

Solution:

$S \longrightarrow Aa \mid b$

$A \longrightarrow Ac \mid Aad \mid bd \mid \Lambda$

$S \longrightarrow Aa \mid b$

$A \longrightarrow bd\ A' \mid \Lambda$

$A' \longrightarrow c\ A' \mid ad\ A' \mid \Lambda$

June 2018 Q. 2 (b)	left factored and eliminate left recursion

$E \longrightarrow aba$

$E \longrightarrow abba$

$E \longrightarrow Ea$

$E \longrightarrow EbE$

Solution,

Left factored:

$E \longrightarrow ab\,A$

$A \longrightarrow a$

$A \longrightarrow ba$

$E \longrightarrow EB$

$B \longrightarrow a$

$B \longrightarrow bE$

Removing left recursion

$E \longrightarrow ab\,AB$

$A \longrightarrow a \mid ba$

$B \longrightarrow a \mid bE \mid \Lambda$

June–2016

Example:

$S \rightarrow EtS \mid EtSeS \mid atS \mid SbE \mid Ex$

$E \rightarrow EEb \mid Ea \mid DaD$

$D \rightarrow b \mid bxc$

Solution:

$S \rightarrow EtSS' \mid atS \mid SbE \mid Ex$

$S' \rightarrow eS \mid \lambda$

$E \rightarrow EE' \mid DaD'$

$E' \rightarrow Eb \mid a$

$D \rightarrow bD'$

$D' \rightarrow xc \mid \lambda$

$S \rightarrow EtSS'T \mid atST \mid ExT$

$T \rightarrow bET \mid \lambda$

$E \rightarrow DaDF$

$F \rightarrow EbF \mid aF \mid \lambda$

$D \rightarrow bD'$

$D' \rightarrow xc \mid \lambda$

$S' \rightarrow eS \mid \lambda$

Question:

① $E \longrightarrow TE'$
② $E' \longrightarrow +TE'$
③ $E' \longrightarrow \wedge$
④ $T \longrightarrow FT'$
⑤ $T' \longrightarrow *FT'$
⑥ $T' \longrightarrow \wedge$
⑦ $F \longrightarrow (E)$
⑧ $F \longrightarrow id$

~~Production Sets~~

- First Set

First (E) = { (, id }
First (E') = { +, ∧ }
First (T) = { (, id }
First (T') = { *, ∧ }
First (F) = { (, id }

- Follow Set

Follow (E) = {), $ }
Follow (E') = {), $ }
Follow (T) = { +,), $ }
Follow (T') = { +,), $ }
Follow (F) = { *, +,), $ }

- Selection Set

Select (1) = First (TE') = { (, id }
Select (2) = First (+TE') = { + }
Select (3) = Follow (E') = {), $ }
Select (4) = First (FT') = { (, id }
Select (5) = First (*FT') = { * }
Select (6) = Follow (T') = { +,), $ }
Select (7) = First (E) = { (}
Select (8) = First (id) = { id }

- Parser Table:

Non-terminals	id	+	*	(	)	$
E	1			1		
E'		2			3	3
T	4			4		
T'		6	5		6	6
F	8			7		

- Control Table:

Non-terminals	id	+	*	(	)	$
E	POP PUSH(E', T) RETAIN			POP PUSH(E', T) RETAIN		
E'		POP PUSH(E', T) ADVANCE			POP RETAIN	POP RETAIN
T	POP PUSH(T', F) RETAIN			POP PUSH(T', F) RETAIN		
T'		POP RETAIN	POP PUSH(T', F) ADVANCE		POP RETAIN	POP RETAIN
F	POP ADVANCE			POP PUSH(), E) ADVANCE		
$						ACCEPT

- Parsing a string

stack	Input
$E	id + id * id $
$E'T'	id + id * id $
$E'T'F	id + id * id $
$ E'T'	+ id * id $
$ E'	+ id * id $
$E'T	id * id $
$ E'T'F	id * id $
$ E'T'	* id $
$ E'T'F	id $
$ E'T'	$
$ E'	$
$	$
Accept	

- Recursive Descent Parser / Pseudo Code

```
E( )
{
    if lookahead ∈ { id, '(' }
    {
        T( );
        E'( );
    }
    else
        Error;
}
```

```
E' ()
{
    if lookahead ∈ { '+' }
    {
        match ('+');
        T();
        E'();
    }
    elseif lookahead ∈ { ')', $ }
    {
        // do nothing because of null;
    }
    else
        Error;
}

T()
{
    if lookahead ∈ { id, '(' }
    {
        F();
        T'();
    }
    else
        Error;
}
```

```
T'()
{
   if lookahead ∈ {'+', ')', $}
    {      }
   else if lookahead ∈ {'*'}
    {
        match ('*');
        F();
        T'();
    }
     else
         Error;
}

F()
{
    if lookahead ∈ {'('}
     {
         match ('(');
         E();
         match (')');
     }
      else if lookahead ∈ { id}
      {
         match(id);
      }
      else
           Error;
     }
```

```
match (char t)
{
    If ( lookahead == t)
            lookahead = getchar( );
    else
        Error;
}
```

June-2018
Q4a.

$E \longrightarrow E+E$
$E \longrightarrow E-E$
$E \longrightarrow E*E$
$E \longrightarrow E/E$
$E \longrightarrow a \mid b$

writ leftmost and rightmost derivation for a+b * a+b

Leftmost derivation:

$E \rightarrow E*E \longrightarrow E+E*E \longrightarrow a+E*E \longrightarrow a+b*E$

$\longrightarrow a+b*E+E \longrightarrow a+b*a+E \longrightarrow a+b*a+b$

Rightmost Derivation:

$E \rightarrow E*E \rightarrow E*E+E \longrightarrow E*E+b \longrightarrow E*a+b$

$\longrightarrow E+b*a+b \longrightarrow a+b*a+b$

Top-down and Bottom-up parsing:

- In top-down parsing, we begin with starting non-terminal and reach till given string by applying leftmost or ~~and~~ rightmost (or none of them derivation).

- In bottom-up parsing, we begin with given string apply production rules in reverse order to reach at single starting non-terminal.

$a+b*a+b \longrightarrow E+b*a+b \longrightarrow E+E*a+b \longrightarrow E+E*E+b$

$\longrightarrow E+E*E+E \longrightarrow E+E*E \longrightarrow E+E \longrightarrow E$

June-2015 Q8 b

Write Pseudo-code for:

1- $S \rightarrow ZY$

2- $Z \rightarrow dZ$

3- $Z \rightarrow XY$

4- $Y \rightarrow cY$

5- $Y \rightarrow \wedge$

6- $X \rightarrow Y$

7- $X \rightarrow aX$

Symbol	First Set	Follow Set	Selection Set
S:	{d, a, c, ∧}	{$}	Select(1): First(ZY) U Follow(S) = {d, a, c, $}
Z:	{d, a, c, ∧}	{c, $}	Select(2): First(dZ) = {d}
Y:	{c, ∧}	{c, $}	Select(3): First(XY) U Follow(Z) = {a, c, $}
X:	{a, c, ∧}	{c, $}	Select(4): First(cY) = {c}
			Select(5): Follow(Y) = {c, $}
			Select(6): First(Y) U Follow(X) = {c, $}
			Select(7): First(aX) = {a}

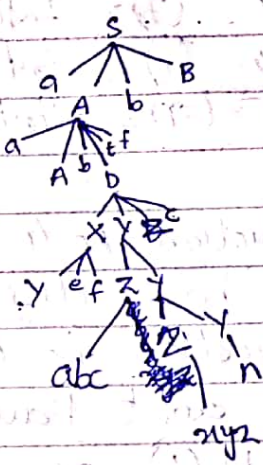
Recursive - Descent Parser / Pseudo code:

```
S()
{
    if lookahead ∈ {'a', 'c', 'd', '$'}
    {
        Z();
        Y();
    }
    else {Error;}
}

Z()
{
    if lookahead ∈ {'d'}
    {
        match('d')
        Z();
    }
    else if lookahead ∈ {'a', 'c', '$'}
    {
        X();
        Y();
    }
}

Y()
{   if lookahead ∈ {'c'}
    {   match('c');
        Y();
    }
    else if lookahead ∈ {'c', '$'}
    {   }
}
```

```
X()
{
    if lookahead ∈ {'c', '$'}
    {
        match {'c', '$'};
        Y();
    }
    else if lookahead ∈ {'a'}
    { match ('a');
      X();
    }
    else { Error;}
}

match (char t)
 {
    if lookahead == t
            lookahead = getchar();
    else
        Error;
}
```

Dec-2014
Q2 Is it LL(1) Grammar:

1- $S \rightarrow aAbB$
2- $A \rightarrow aAbDtf$
3- $D \rightarrow XYZc$
4- $X \rightarrow Yef$
5- $X \rightarrow \wedge$
6- $Y \rightarrow ZY$
7- $Y \rightarrow \wedge$
8- $Z \rightarrow abc$
9- $Z \rightarrow xyz$



Solution:

The given CFG is already left factored and there is no left recursion.

Symbol	First Set	Follow Set
S	{a}	{$}
A	{a}	{b, $}
D	{a, x, ∧}	{t, }
X	{a, x, ∧}	{a, x, }
Y	{a, x, ∧}	{a, x, e}
Z	{a, x}	{$} {c, a, x, e}

Selection Set:

Select(1) = First(aAbB) = {a}
// (2) = first(aAbDtf) = {a}
// (3) = first(XYDc) = {a, x, }
// (4) = first(Yef) = follow(X) = {a, x, }

Select(5) = Follow(X) = {a, x, $}

" (6) = first(ZY) = {a, x}

" (7) = follow(Y) = {a, x, $}

" (8) = first(abc) = {a}

" (9) = first(x.yz) = {x}

- Production no. 4 and 5 have same lefthand side but don't have disjoint selection sets
- Production no. 6 and 7 have same lefthand side but don't have disjoint selection sets. Hence it is not LL(1) grammar.

July-2015 Part A Q9 For the following C-Language statements, find what is the error and classify the error into lexical, syntax and symantic.

	Statement	Error Description	Type of Error
①	int 9abc=89;	Invalid token: id cannot be started with digit	Lexical Error
②	for(a<=5){ }	Error in expression: incomplete for	Syntax Error
③	int *ptr, arr[10]; ptr = arr; arr ++;	Error in expression: Array can't be incremented.	Semantic Error.
④	struct Acc{ int num; string id="EECO"; }	"string" is not a data type of C	Syntax Error
⑤	public void func(int x){ float f=99.99; ..... f+=x ..... return f; }	float and integer variables can't be added.	Semantic Error
⑥	string str = "comp"+"iler";	(i) string is not a data type in C (ii) strings can not be concatenated with '+' sign.	(i) Syntax error (ii) Semantic Error.

Attributed Grammar:

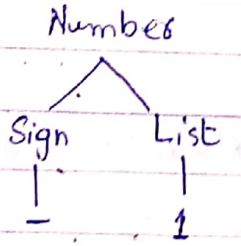
- A CFG is augmented with the set of rules.
- Each symbol in derivation has a set of values or attributes.
- Rules specify how to compute a value for each attribute.

CFG for Sign binary numbers:

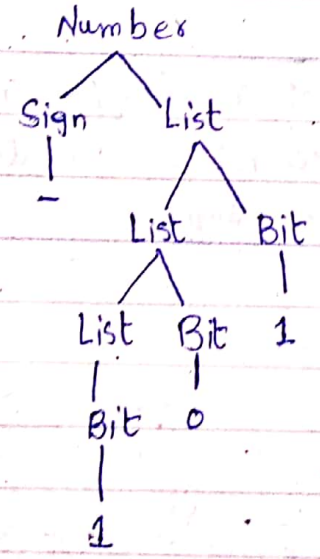
<Number> ⟶ <Sign> <List>
<Sign> ⟶ + | -
<List> ⟶ <List> <Bit> | <Bit>
<Bit> ⟶ 0 | 1

Parse Trees:

① for -1



② for -101



Leftmost derivation for -101 :

<Number> ⟶ <Sign><List> ⟶ - <List> ⟶ - <List><Bit>

⟶ - <List><Bit><Bit> ⟶ - <Bit><Bit><Bit>

⟶ -1 <Bit><Bit> ⟶ -10 <Bit> ⟶ -101

Rightmost derivation for -101 :

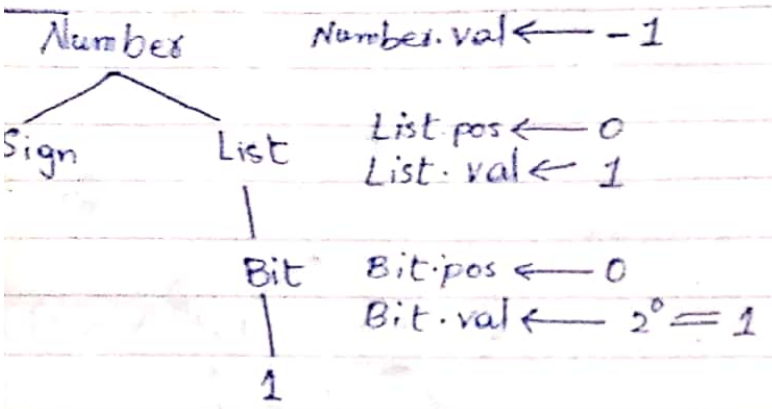
<Number> ⟶ <Sign><List> ⟶ <Sign><List><Bit> ⟶

<Sign> <List> 1 ⟶ <Sign> <List> <Bit> 1 ⟶ <Sign><List> 01

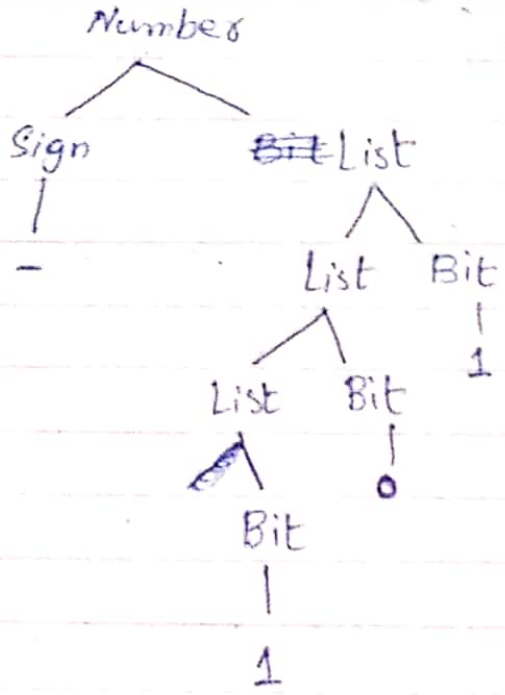
⟶ <Sign><List> 01 ⟶ <Sign> <Bit> 01 ⟶

<Sign> 101 ⟶ -101

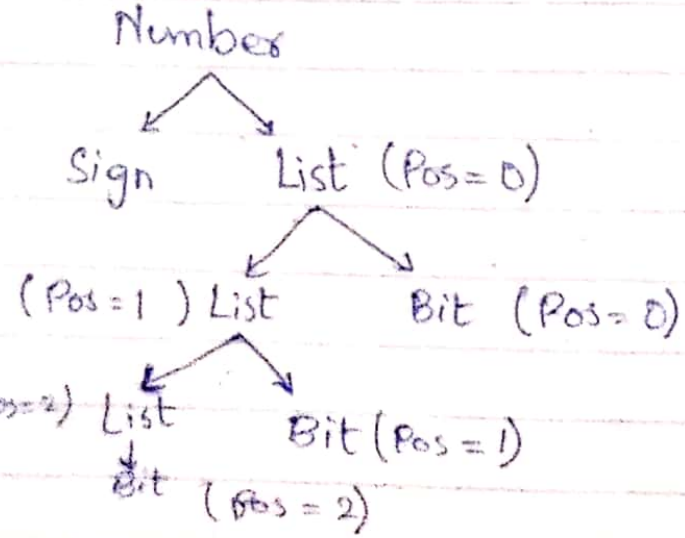
-1


Number
Number.val ← -1
Sign
List
List.pos ← 0
List.val ← 1
Bit
Bit.pos ← 0
Bit.val ← 2^0 = 1
1

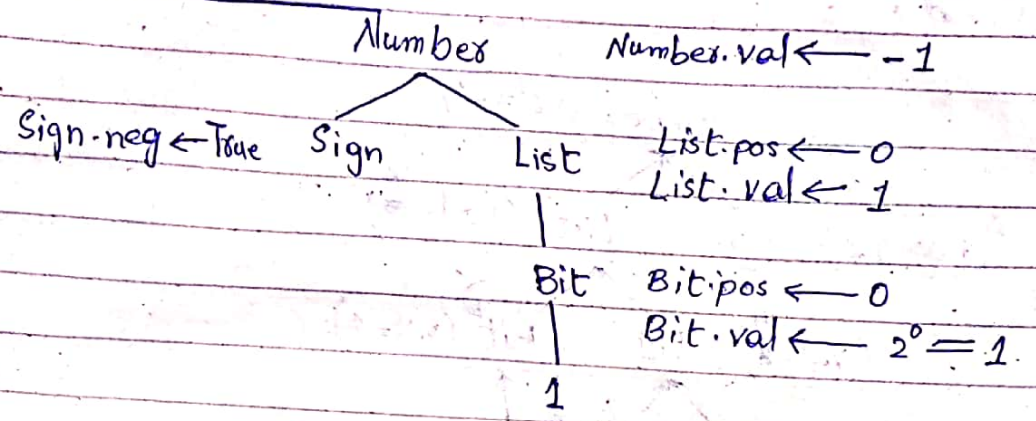
-101


Number
Sign
~~Bit~~ List
-
List
Bit
1
List
Bit
0
Bit
1

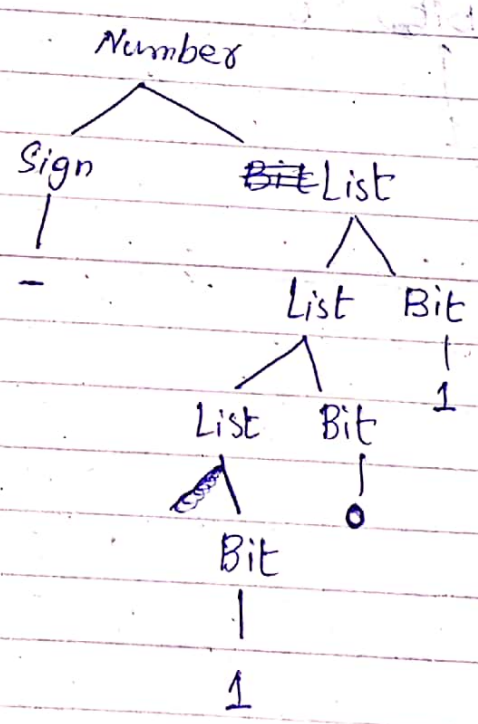
1:


Number
Sign
List (Pos=0)
(Pos=1) List
Bit (Pos=0)
(Pos=2) List
Bit (Pos=1)
Bit (Pos=2)

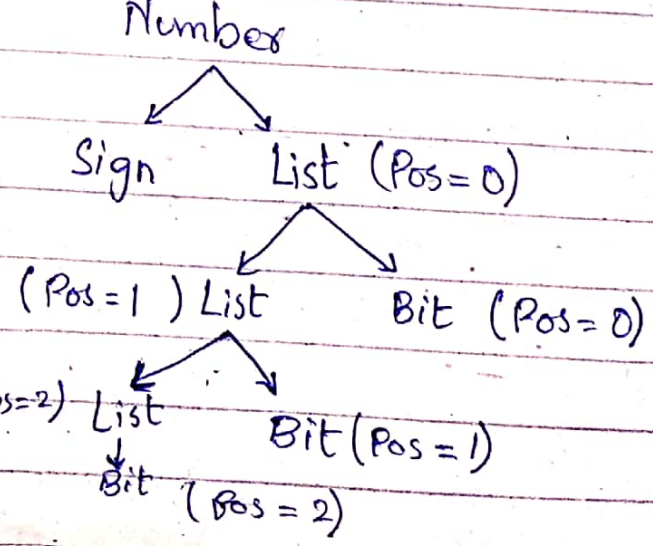
Parse Tree for -1



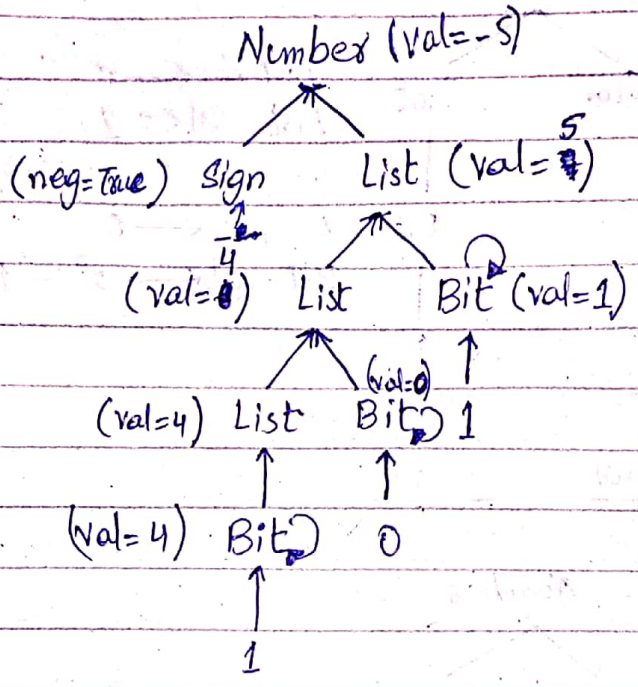
Parse Tree for -101



Inherited:



Synthesized :



June 2005

Q1. Write recursive descent parser.

$E_v \longrightarrow F_q\ T_{c,d}$; $v = d$, $c = q$

$T_{q,d} \longrightarrow F_q\ T_{c_1,d_1}$; $c_1 = d + q$, $d = d_1$

$T_{c,d} \longrightarrow \wedge$; $d = 0$

$F_q \longrightarrow num_r$; $q = r$

$F_q \longrightarrow (E_v)$; $q = v$

v is synthesized attribute of E
q is synthesized attribute of F
d is synthesized attribute of T
c is inherited attribute of T

Solution:

We simplifying the given CFG

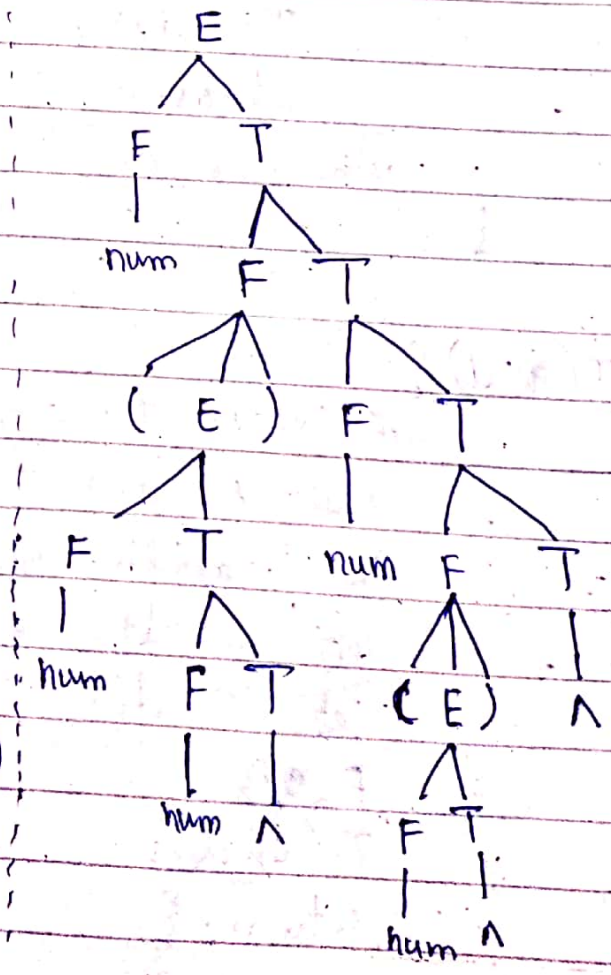
$E \longrightarrow FT$

$T \longrightarrow FT$

$T \longrightarrow \wedge$

$F \longrightarrow num$

$F \longrightarrow (E)$



Derived Equation:

num (num num) = num (num)

Non-terminal	First set	Follow set
E	{num, (}	{$,)}
T	{num, (, ∧}	{$,)}
F	{num, (}	{num, (, $,)}

Now recursive descent:

```
E(v) {
    if (lookahead == 'num' || lookahead == 'c') {
        local variable a, c, d;
        let v = d;
        let c = a;
        F(a);
        T(c,d);
        return; }
    else { error; }
}

T(a, d) {
    if (lookahead == '+') {
    call ADVANCE
    local variable a, c1, d1;
    let c1 = d+a ;
    let d = d1;
    F(a);
    T(c1, d1);
    return; }
```

```
    else {error('Syntax error');}
}

T(c,d) {
    if (lookahead == '$' || lookahead == ')') {
        let d=c;
        return;}
    else {error("Syntax error");}
}

F(q) {
    if (lookahead == 'num') {
        local variable r;
        let r = INATT;
        call ADVANCE;
        let q=r;
        return;}
    else if (lookahead == '(') {
        local variable v;
        call ADVANCE;
        E(v);
        call ADVANCE;
        let q=v;
        return;}
    else {error('Syntax Error');}

main() {
    let INCLASS = class part of first input symbol;
    let INATT = value part of first input symbol;
    if (INCLASS == '$') {error('syntax error')}
    else {ACCEPT;}
}
```

① CFG if else statement

<S> → if <E> then <S> | if <E> then <S> else <S>
E → 0 | 1

② CFG for variable declaration

<Var Dec> → <Type> <Identifier>
<Type> → int | float | double | char
<Identifier> → <Alpha> <Alphanumeric>
<Alpha> → a | b | .... | z | A | B | .... | Z
<Alphanumeric> → <Alpha><Alphanumeric> | <Numeric> <Alphanumeric> | ∧
<Numeric> → 0 | 1 | 2 | .... | 9

③ CFG for Arithmetic statement / Expression

E → E op E | (E) | num
op → + | * | / | -
num → 0 | 1 | 2 | .... | 9 | .... | nth

July 2018

Q 1(b) Token Set

Class part	Value Part	Line #	class Part	Value Part	Line#
data type	int	01	Assignement OP.	=	04
id	k	01	valid number	0	04
Assignment OP.	=	01	Line Terminator	;	04
valid number	2	01	id	j	04
Arithmetic OP.	*	01	Relational OP.	<	04
id	n	01	id	k	04
Arithmetic OP.	-	01	Line Terminator	;	04
valid number	2	01	id	j	04
line Terminator	;	01	increment OP.	++	04
Keyword	for	02	Right Parenthesis	)	04
left Parenthesis	(	02		cout <<	05
data type	int	02	character Constant	" "	05
id	i	02	Line Terminator	;	05
Assingment OP.	=	02	id	k	06
valid number	0	02	Assignement OP	=	06
line Terminator	;	02	id	k	06
id	i	02	Arithmetic DP.	-	06
Relational OP.	<	02	valid number	2	06
id	n	02	Line Terminator	;	06
Line Terminator	;	02	Keyword	for	07
id	i	02	left Parenthesis	(	07
increment OP	++	02	data type	int	07
Right Parenthesis	)	02	id	j	07
	{	03	Assignment OP	=	07
Keyword	for	04	valid number	0	07
left Parenthesis	(	04	Line Terminator	;	07
data type	int	04	id	j	07
Id	j	04	Relational OP	<=	07

Class Part	Value Part	line #
id	i	07
Line Terminator	;	07
id	j	07
Increment OP.	++	07
Right Parenthesis	)	07
	{	08
cout	Cout<<	09
character Constant	" * "	09
Line Terminator	;	09
	}	10
	Cout<<	11
invalid Token	endl	11
Line Terminator	;	11
	}	12
	}	13

Q) 2(b) - July 2018

Eliminate left factor & left recursion of the grammar.

$E \longrightarrow aba$

$E \longrightarrow abba$

$E \longrightarrow Ea$

$E \longrightarrow EbE$

Solution:

Left factor:

$E \longrightarrow abE'$

$E' \longrightarrow a/ba$

$E \longrightarrow ES'$

$S' \longrightarrow a/bE$

Left Recursion:

$E \longrightarrow abE'S'$

$E' \longrightarrow a/ba$

$S' \longrightarrow a/bE/\Lambda$

Q3(a) Write R.E for a language containing strings which end abb over {a, b}.

$R.E = (a+b)^*.abb$

Q3(c) Consider the following LL(1) parsing table and prove the string a+b+a$:

Solution:

Control Table

Parser Table

	a	b	+	*	$
E	E→TE'	E→TE'			
E'			E'→+TE'		E'→∧
T	T→F	T→F			
T'			T→∧	T'→*FT'	T→∧
F	F→a	F→b			

Control Table

	a	b	+	*	$
E	POP PUSH(E',T) RETAIN	POP PUSH(E',T) RETAIN			
E'			POP PUSH(E',T) ADVANCE		POP RETAIN
T	POP PUSH(F) RETAIN	POP PUSH(F) RETAIN			
T'			POP RETAIN	POP PUSH(T',F) ADVANCE	POP RETAIN
F	POP ADVANCE	POP ADVANCE			
$					ACCEPT

Parsing a String:

Stack	Input
$E	a+b+a$
$E'T	a+b+a$
$E'F	a+b+a$
$E'	+b+a$
$E'T	b+a$
$E'F	b+a$
$E'	+a$
$E'T	a$
$E'F	a$
$E'	$
$	$
ACCEPT	

Q4(a) Consider the grammar given below:

$E \longrightarrow E+E$

$E \longrightarrow E-E$

$E \longrightarrow E*E$

$E \longrightarrow E/E$

$E \longrightarrow a/b$

Solution:

Left-most Derivation:

$E \rightarrow E*E \rightarrow E+E*E \rightarrow a+E*E$

$\rightarrow a+b*E \rightarrow a+b*E+E \rightarrow a+b*a+E$

$\rightarrow a+b*a+b$

Right most Derivation:

$E \rightarrow E*E \rightarrow E*E+E \rightarrow E*E+b \rightarrow E*a+b$
$\rightarrow E+E*a+b \rightarrow E+b*a+b \rightarrow a+b*a+b$

Pseudo Code:

```
E( ) {
    if lookahead ∈ {'a', '+', 'b', '*', 'a', '+', 'b'}
```

Jan-2017

Q4(a)

$S \rightarrow X\,S\,A$

$X \rightarrow a\,A$

$Y \rightarrow b\,A\,x \mid \wedge$

$A \rightarrow c\,A \mid b\,A \mid \wedge$

Non-Term	First Set	Follow Set	Selection Set
S	{a}	{c, b, $}	① first(XSA) = {a}
X	{a}	{a}	② first(aA) = {a}
Y	{b, ∧}	{c, b, $}	③ first(bAx) = {b}
A	{c, b, ∧}	{x, c, b, $}	④ follow(Y) = {c, b, $}
			⑤ first(cA) = {c}
			⑥ first(bA) = {b}
			⑦ follow(A) = {x, c, b, $}

Passing Table

Non-termi	a	b	c	x	$
S	1				
X	2				
Y		3, 4	4		4
A		6, 7	5, 7	7	7

Done:

July - 2016

Q1(b) Remove left recursion and perform left factoring in the following grammar.

$S \rightarrow EtS$
$S \rightarrow EtSeS$
$S \rightarrow atS$
$S \rightarrow SbE$
$S \rightarrow Ex$
$E \rightarrow EEb$
$E \rightarrow Ea$
$E \rightarrow DaD$
$D \rightarrow b$
$D \rightarrow bxc$

Left factoring:

$S \rightarrow EX \mid atS \mid SbE$
$X \rightarrow Y \mid x$
$Y \rightarrow teSZ$
$Z \rightarrow eS \mid \wedge$
$E \rightarrow EE' \mid DaD$
$E' \rightarrow Eb \mid a$
$D \rightarrow bD'$
$D' \rightarrow xc \mid \wedge$

$A \rightarrow A\alpha \mid \beta$
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A'$

Left Recursion:

$S \rightarrow EXS' \mid atSS'$
$S' \rightarrow bES' \mid \wedge$
$X \rightarrow Y \mid x$
$Y \rightarrow tSZ$
$Z \rightarrow eS \mid \wedge$
$E \rightarrow DADT$
$T \rightarrow E'T$
$E' \rightarrow Eb \mid a \mid \wedge$
$D \rightarrow bD'$
$D' \rightarrow xc \mid \wedge$