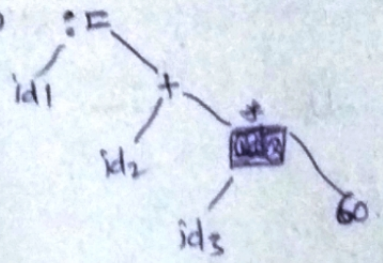
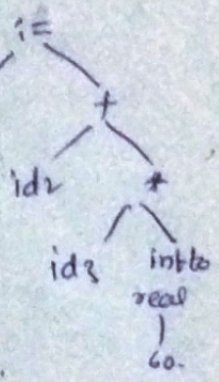
Example: position := initial + rate * 60.

lexical ⇒ id1 := id2 + id3 * 60

Syntax ⇒



Semantic ⇒



Intemediate code generator ⇒

$t_2 = id3 * 60.0$

$t_1 = id2 + t_2$

$t_3 = t_1 + t_2$

$id1 = t_3$.

code optimizes.

$t_2 = id3 * 60.0$.

$id1 = id2 + t_2$.

code generator: MOV R1, id3
MUL R1
⋮

Cousins of the compiler: PALoders, Linkers, debuggers.

preprocessors: #include, # define

Assemblers: helps compiler in generating relocatable machine code.

Loaders and linkers: loader copies code and data into memory, allocates storage
linkers handles relocation and resolves symbol references.

Debuggers: locating the errors by the use of break-points.

compiler construction tools:

example $S \to iEtS \mid iEtSeS \mid a$
$E \to b$

$S \to iEtS$
$S \to iEtSeS$
$S \to a$
$E \to b$

$A \to A\alpha \mid \beta$
$A \to \beta A'$
$A' \to \alpha A' \mid \varepsilon$

$S \to iEt\underset{A\ \alpha}{S}eS \mid \underset{\beta}{a}$

$S \to aS'$
$S' \to iEt$

Algorithm for predictive parsing

M → table, w → string, X → stack top symbol, a input symbol

if X = a
pop X from stack and advance a else error()

else
push x production
$X - Y_1 Y_2 \cdots Y_k$ such that Y_1
pop x and push x production

until X = $

$A \to A\alpha \mid \beta$
$A \to \beta A'$
$A' \to \alpha A' \mid \varepsilon$

Left factoring

$A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 \ldots | \alpha\beta_n | \gamma$

$\alpha \to$ is the common prefix

$A \to \alpha A'$

$A' \to \beta_1 | \beta_2 | \beta_3 \ldots | \beta_n$

$S \to \underline{iEtS} | \underline{iEtSeS} | \underline{a}$
$\quad\quad \alpha\ \beta_1 \quad\quad \alpha\ \beta_2 \quad \gamma$

$S \to iEtSS' | a$

$S' \to \varepsilon | eS$

$E \to b$

Left Recursion

$A \to A\alpha | \beta$

$A \to \beta A'$

$A' \to \alpha A' | \varepsilon$

$S' \to eS$
follow(S')

first(S) = {i, a}

first(S') = {e, ε}

first(E) = {b}

follow(S) = {$, e}
↙
first(S')

follow(S') = {$, e}

follow(E) = {t}

$S' \to \varepsilon$

	a	b	i	t	e	$
S	S→a		S→iEtSS'		Sync	Sync
S'					S'→eS S'→ε	S'→ε
E		E→b		Sync		

since more than one production grammar is not LL(1) grammar

```
%{
 int nc=0;
 int nl=0;
 int nw=0;
%}
%%
\n  {nl++;}
[^ \t\n]+ {nl++= yyleng;
           nc++ = yyleng;}
.  {nc++;}
%%
int main(){
  yylex();
```

Canonical set.

By finding the closure of.

21/02/2024
Wednesday.

⇒ • is followed by non-Terminal then add its (non-terminals) one next step.

example $A \rightarrow aBB$

$B \rightarrow ba$

String ababa.

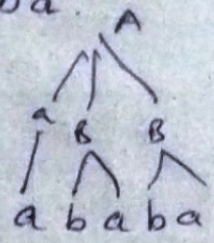
$A \rightarrow aBB$

$\downarrow$

ba

aBba

ababa.



LR(0) → If we are getting multiple Actions, then we cannot perform LR(0).

* do not keep E!

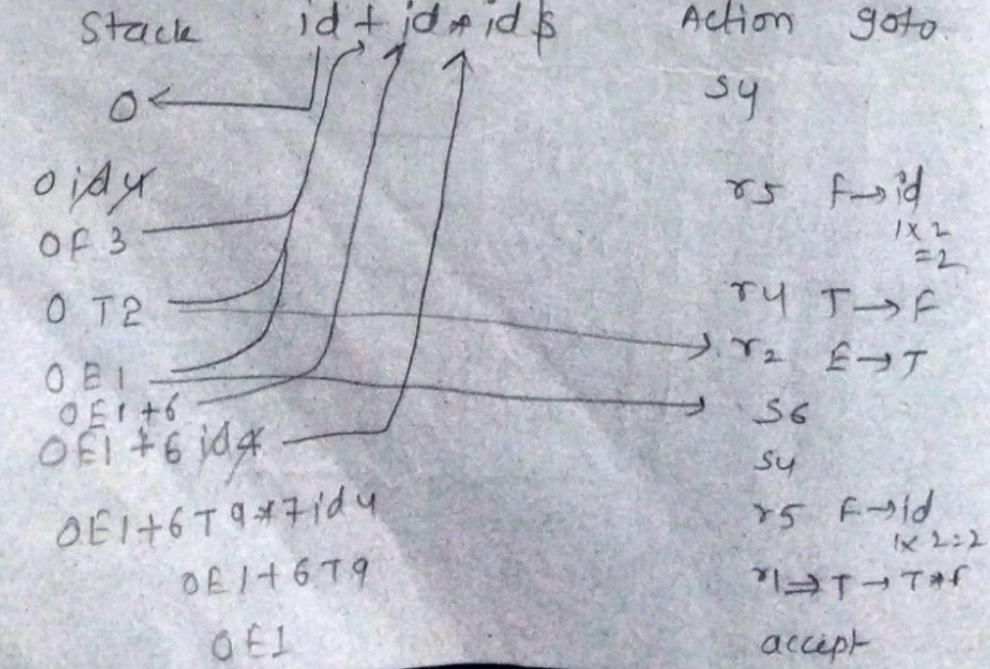
	Action						GO TO		
	+	*	id	(	)	$	E	T	F
0			S4	S5			1	2	3
1	S6								
2	r2	S7			r2	r2			
3	r4	r4			r4	r4			
4	r5	r5			r5	r5			
5			S4	S5			8	2	3
6			S4	S5				9	3
7			S4	S5					10
8	S6				S11				
9	r1	S7			r1	r1			
10	r3	r3			r3	r3			
11	r6	r6			r6	r6			

Reduce actions, r4, r5, r10, r11, r2, r3

In the previous. We get two actions in one
So → SLR is preferred in those kind of
case.

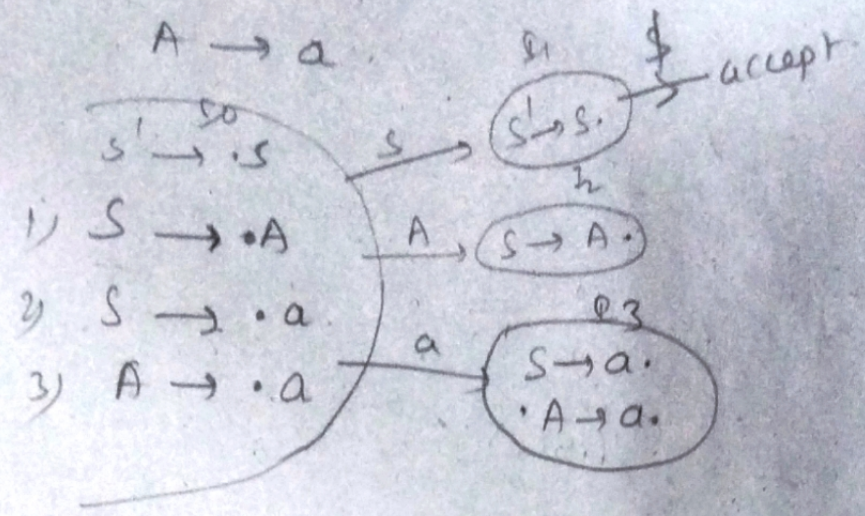
⇒ We have to keep reduce actions whenever
after finding the follow (Non-terminal)
at those respective symbols. Note: In SLR no need of removing left recursion

	Action						GOTO		
	+	*	id	(	)	$	E	T	F
0			S4	S5			1	2	3
1	S6					accept			
2	r2	S7			r2	r2			
3	r4	r4			r4	r4			
4	r5	r5			r5	r5			
5			S4	S5			8	2	3
6			S4	S5				9	3
7			S4 (see one)	S5					10
8	S6				S11				
9	r1	S7			r1	r1			
10	r3	r3			r3	r3			
11	r6	r6			r6	r6			



$S \rightarrow A/a$

$A \rightarrow a$

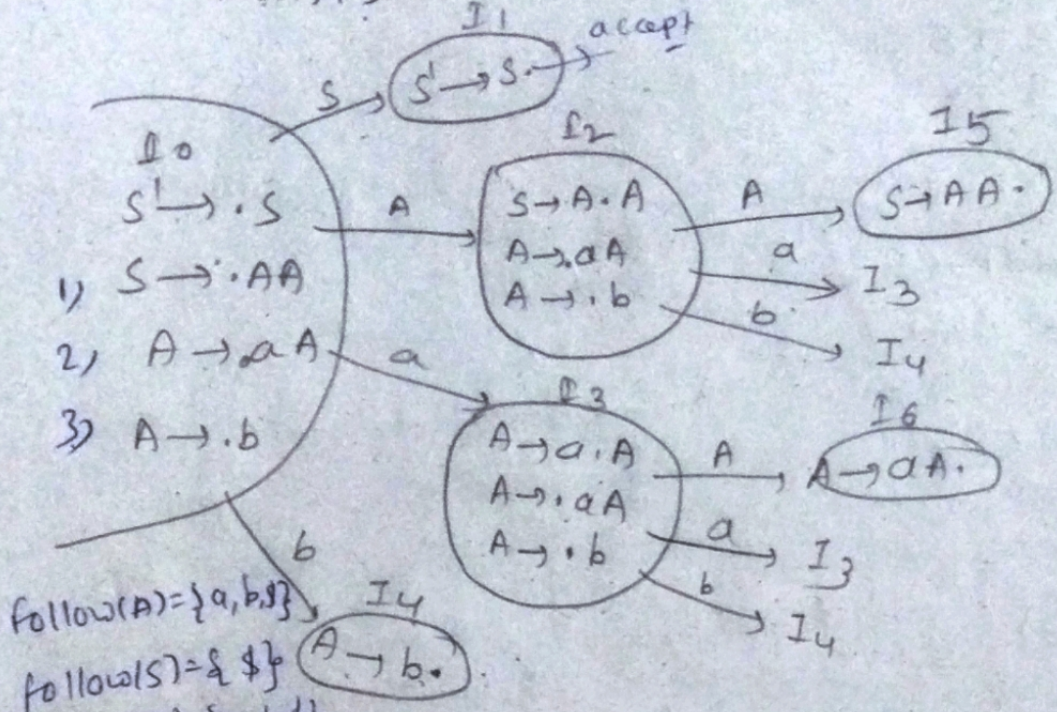


Action			goto	
	a	b	S	A
0			1	2
1		accept		
2				
3	r2/r3			
4				

reduce action reduce-reduce conflict, so → not SLR

Ex:- $S \to AA$
$A \to aA \mid b$

Steps: ① Augmented grammar
② Explore DFA.



may we get Shift reduce conflict
Reduce reduce conflict.
r/r or r/s

follow(A) = {a, b, $}
follow(S) = {$}
follow(A) = {a, b, $}

	Action			goto	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

SS
Sr
rs
rr
SLR

Stack	i/p	Action
0	a b b $	
0a3		S3
0a3b4		S4
0a3A6		r3 A → b
0A2		r2 A → aA
0A2b4		S4
		r3 A → b

0 on a = S3
3 on b = S4

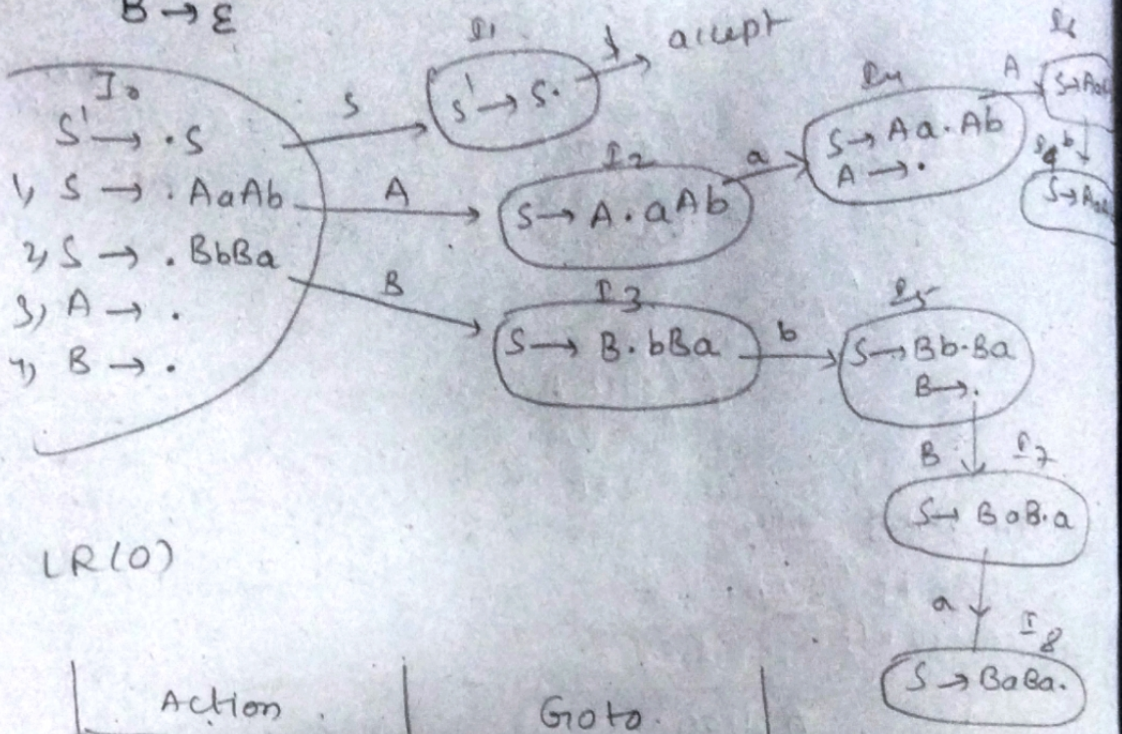
Example

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

Feb 22 / now
Thursday



LR(0)

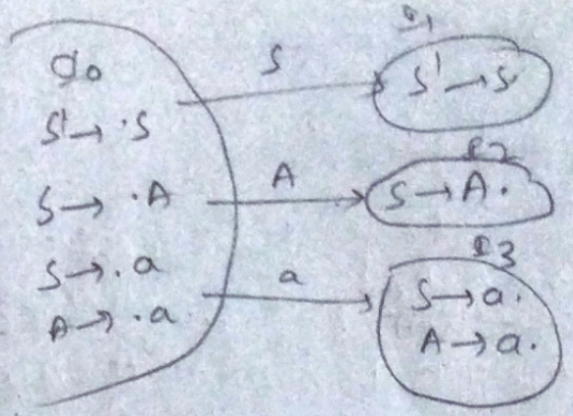
	Action			Goto		
	a	b	$	S	A	B
0	r3/r4	r3/r4		1	2	3
1			accept			
2	S4					
3		S5				
4	r3	r3	r3	6		
5	r4	r4	r4		7	
6		S9				
7	S8					
8			r2			
9			r1			

I_0 on S = 1
on A = 2
on B = 3

Reduce actions
$A \rightarrow .$ I_0
$B \rightarrow .$
r3/r4 → reduce-reduce conflict

In LR(0) we got
$I_0 \rightarrow$ r3/r4 reduce-reduce conflict
LR(0) passing not possible

In SLR
$A \rightarrow .$ Follow(A) = {a, b}
$B \rightarrow .$
Follow(B) = {a, b}
r3/r4 r3/r4 again r-r conflict
So not possible



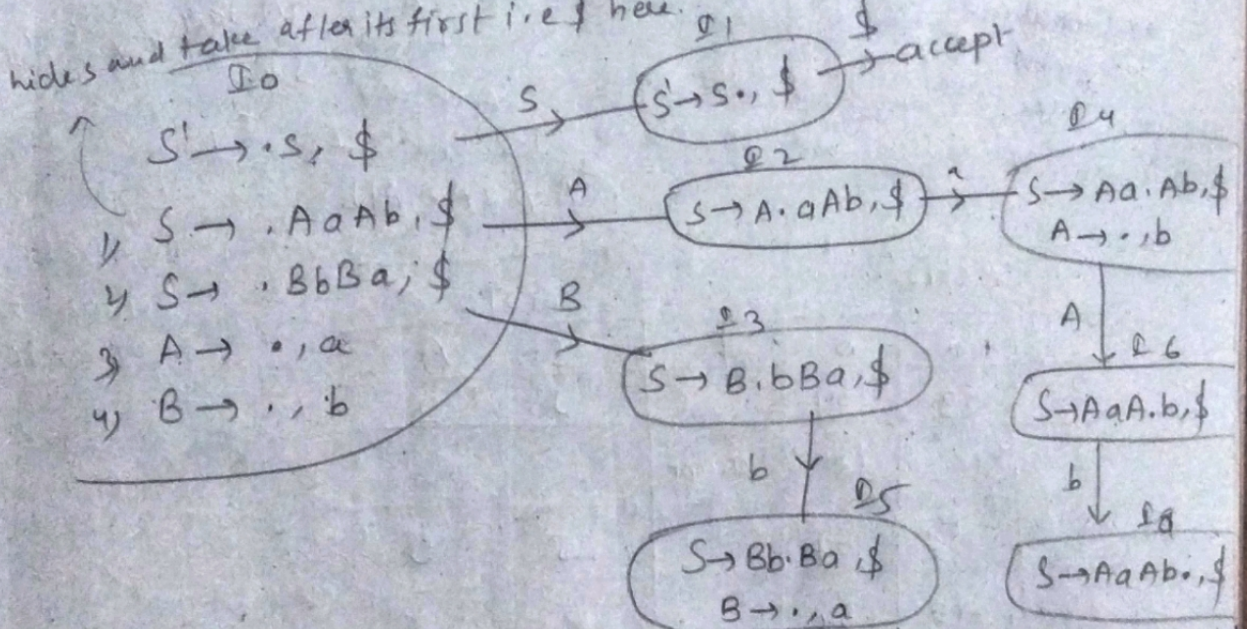
	a	$	S	A
0	S_3		1	2
1				
2		r_1		
3		r_2/r_3		

↳ SLR not possible.

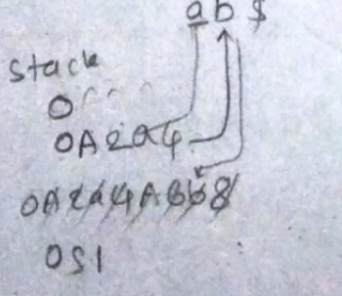
CLR → canonical LR.

Add look-ahead symbol.

hides and take aflo its first i.e $ here.



	Action			Goto		
	a	b	$	S	A	B
0	r_3	r_4		1	2	3
1			accept			
2	S_4					
3		S_5				
4		r_3			6	
5	r_4					7
6		S_8				
7	S_9					
8			r_1			
9			r_2			

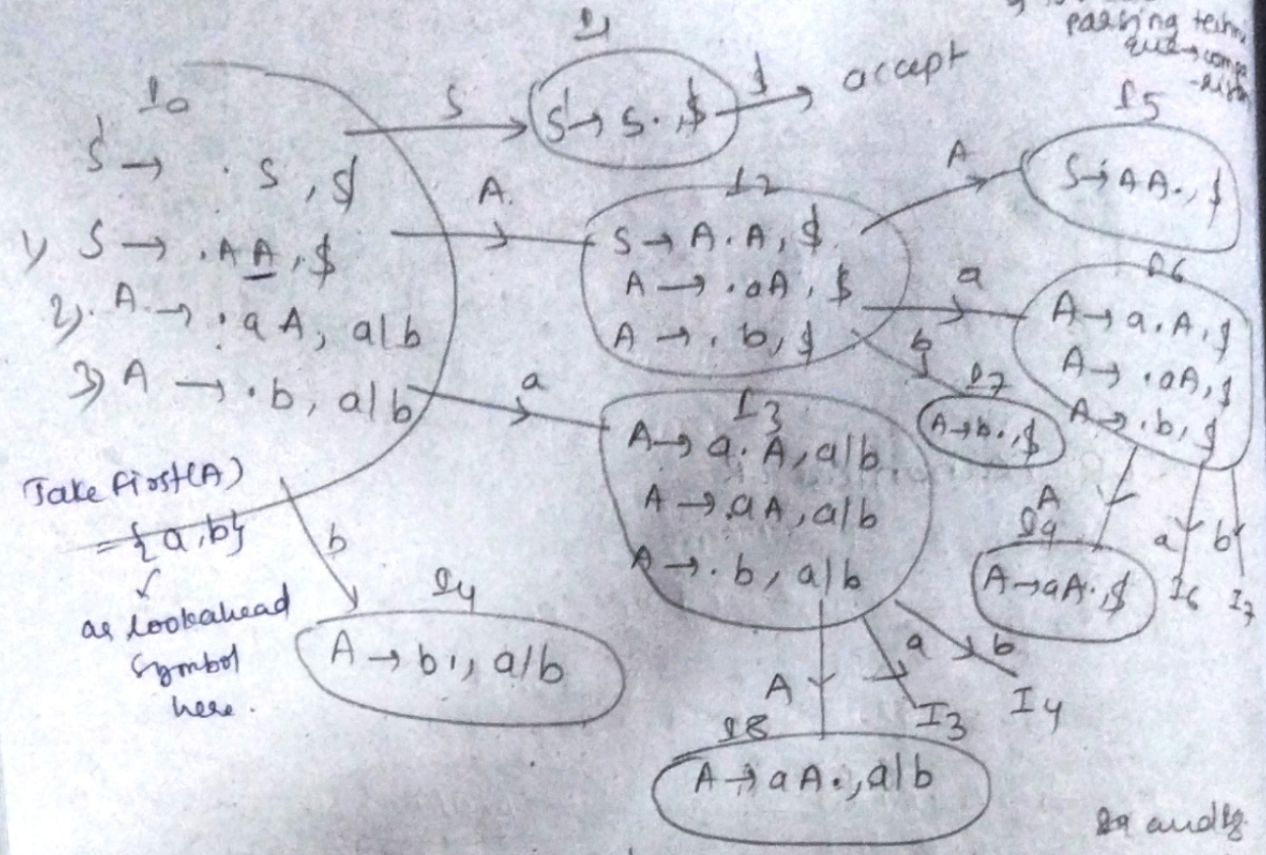


Action

r_3
A →
S4
a on b
r_3 A →·
6 on b = S8
8 on $ = r_1
S → Aa Ab
4×2=8
0S1 pop
1 on $

CLR → is most powerful one

LALR(1) → 3 ex
for each passing term, rule → compare with



I3 and I6
I4 and I7
I8 and I9

	Action			GoTo	
	a	b	$	S	A
0	S_3	S_4		1	2
1			accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

reduce actions

YACC: yet Another compiler compiler

LALR: for better space management it is prefered.

merging rows here.

If shift and reduce are to be merged then it is not possible with LALR

	Action			GOTO	
	a	b	$	S	A
0	S3	S4		1	2
1			accept		
2	S6	S7			5
3	S36	S47			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

LALR table.

	Action			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S6	S7		5	
36	S36	S47		89	
47	r3	r3			
5			r1		
36	S36	S47		89	
47			r3		
89	r2	r2			
89			r2		

4) number of lines, words and characters.
3)

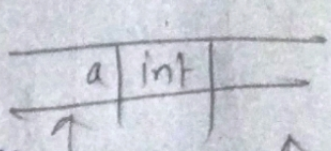
NOTE: All LALR are CLR.

28 Feb, 2024
wednesday

SDT: Syntax Directed Translation.
- ↳ Store & retrieve info from symbol Table
- ↳ Intermediate forms.
- ↳ Semantic checks.

⇒ here we are checking uniqueness of the identifiers.

⇒ with the help of SDT only instructions are given to the system.



int a;
float a;

checks symbol table if found then throws error. if not then it will create

$A \rightarrow XYZ + \{ \}$ (action)

SDT.

Semantic rule
Translation action rule } same.

order postfix form

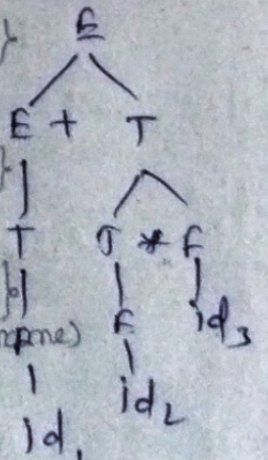
$a + b * c \rightarrow abc*+$

$id_1 + id_2 * id_3 \rightarrow id_1 id_2 id_3 * +$

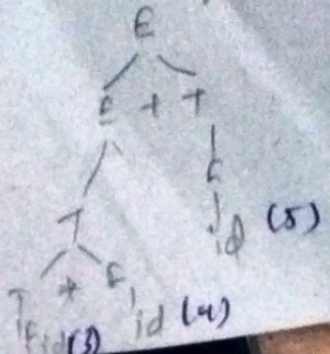
Example:

$E \rightarrow E+T$ { printf("+"); }
$E \rightarrow T$ { }
$T \rightarrow T*F$ { printf("*"); }
$T \rightarrow F$
$F \rightarrow id$ { printf(id.name); }

Example:



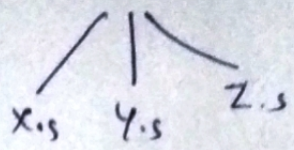
Eg: 3*4+5 ⇒ o/p



Attributes < Synthesized
Inherited

eg: $A \to XYZ$

A.s → attribute value



$A.s = f(X.s + Y.s)$ — its children attributes

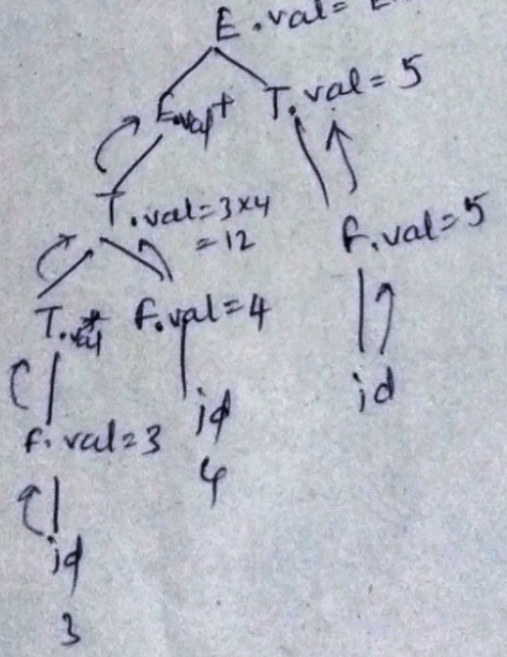
• → synthesized attribute. (evaluated from)

$Y.s = f(A.s, X.s)$

parent, sibling

Inherited attributed.

⇒ If any nodes attributed value is evaluated from its parent or sibling is called Inherited.



$E \to E+T$ { E.val = E.val + T.val }

$E \to T$ { E.val = T.val }

$T \to T*F$ { T.val = T.val * F.val }

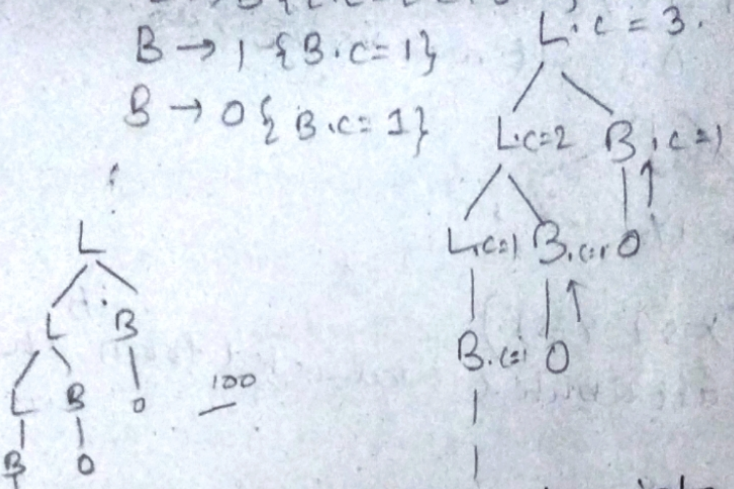
$T \to F$ { T.val = F.val }

$F \to id$ { F.val = Lookup(id.val) }

⇒ Procedure:
Whenever you are going with the actions you attach it.

example: i/p: 100. o/p: 3
count no of digits.

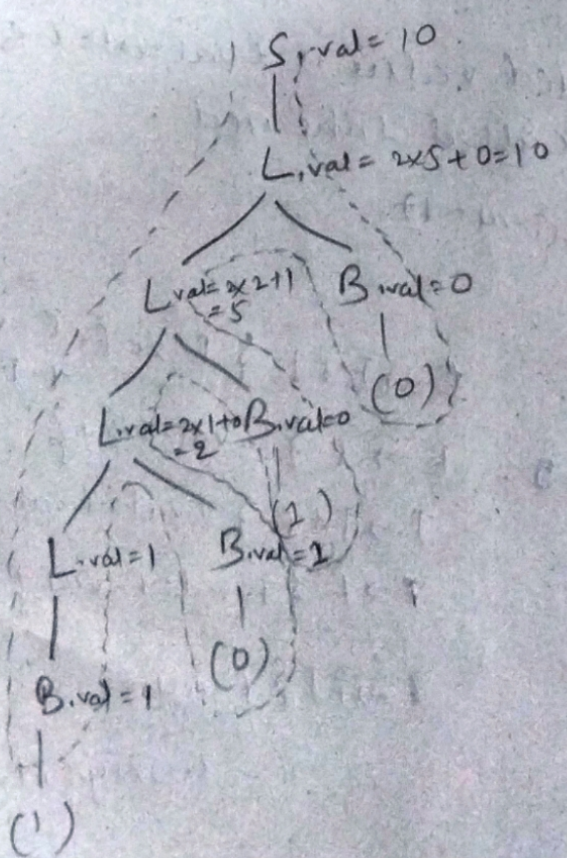
$L \rightarrow LB \{L.c = L.c + B.c\}$
$L \rightarrow B \{L.c = L.c + B.c\}$
$B \rightarrow 1 \{B.c = 1\}$
$B \rightarrow 0 \{B.c = 1\}$



ex:- convert binary number into its equivalent decimal number.

Input
for 1010 $2^3 2^2 2^1 2^0 = 8+2=10$

$S \rightarrow L \{S.val = L.val\}$
$L \rightarrow LB = \{L.val = 2 \times L.val + B.val\}$
$L \rightarrow B \{L.val = B.val\}$
$B \rightarrow 0 \{B.val = 0\}$
$B \rightarrow 1 \{B.val = 1\}$



06/03/2024
Wednesday.

```
%% %
E' : E { printf("accepted"); }
E : E '+' E { }
E : E '*' E { }
E : NUMBER { }
%token NUMBER
%left '+' '*'
%%
main(){
    yyparse();
}
```

→ here we are just verifying so { }.

⇒ tokens belonging

⇒ parallelly we can add many tasks

- l file.
- h file.

Semantic checks:

1) uniqueness:
2) flow of control → eg:- goto L (upto 2nd phase okay). → during 3rd phase whether L is there (declared or not).
3) Type checking

Type Expressions: to implement the type checking.

1) basic type } int char
2) array int a[5];
   (type expression for array.
   a : array(0,..4, int)
   array(IndexRange, type)
3) $T_1 \times T_2$ is also a Type expression
4) function.

```
int add(int a, int b){
    return(a+b);
}
```

5) Domain → Range.

   int x int ⟶ int

6) int *p. void x ⟶ void

06/03/2024
Wednesday.

```
%% printf("accepted");
E : E '+' E { }
E : E '*' E { }
E : NUMBER { }
%token NUMBER
%left '+' '*'
%%
main(){
    yyparse();
}
```

→ here we are just verifying so { }

⇒ tokens belonging

⇒ parallelly we can add many tasks

- .l file
- .h file

Semantic checks:

1) uniqueness: eg:- goto L (upto 2nd phase okay).
2) flow of control → during 3rd phase whether L is there (declared or not).
3) Type checking

Type Expressions: to implement the type checking.

1) basic type → int char
2) array int a[5];
   (type expression for array.
   a : array(0,..4, int)
   array(IndexRange, type)
3) $T_1 \times T_2$ is also a Type expression
4) function.

```
int add(int a, int b){
    return(a+b);
}
```

5) Domain → Range.
   int x int ⟶ int
   void x ⟶ void
6) int *p;

```
Struct book {
    int price;
char author[5];
}
```

record (price x int, author x array (0..4, char))

example: E : int → { E.type = int }

E : E mod E

E : E1 [E2] → { if E1.type == E2.type and E1.type == int then E.type = int else E.type = typeerror }

E : *E2

{ if E2.type = int and E1.type = array(s,t) then E.type }

Intermediate Code Representation:

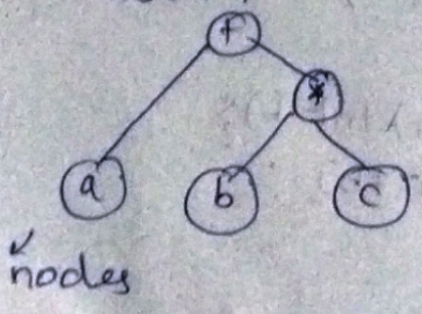
1) Syntax tree
2) DAG
3) Three-address-code
4) postfix notation → examples.

a+b*c
abc*+

(a+b)*(b2+d)
ab+ b2d+*

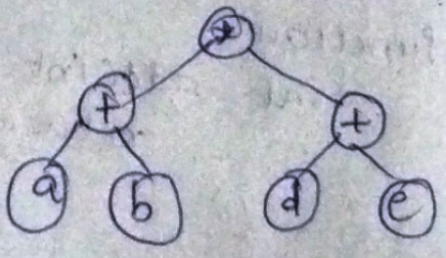
a+b+c
ab+c+

example a+b*c
abc*+



nodes

(a+b)*(d+e)
ab+ de+*



Assignment: write SDT to construct Syntax trees.

$E \rightarrow E + T$

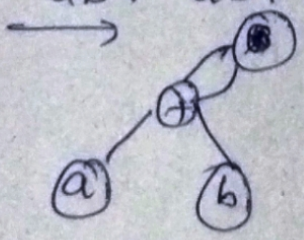
$E \rightarrow T$

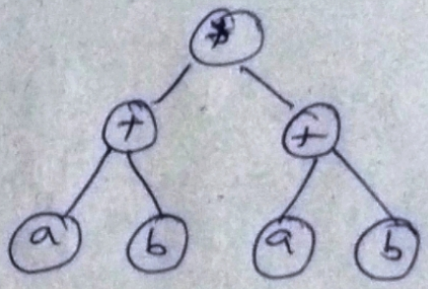
$T \rightarrow T * F$

DAG Notation.

$(a+b) * (a+b)$

$\longrightarrow$ ab+ ab+ *





If common sub expression is there, if it is already there then it refers otherwise creates it.

Three-address-code.

$x = a+b$

$t1 = a+b$

$x = t1$

If any expression.

$A = B \ OP \ C$

For instance if (x==0)

printf ("mam");

if x==0 then goto statement;

$E \rightarrow E + T$

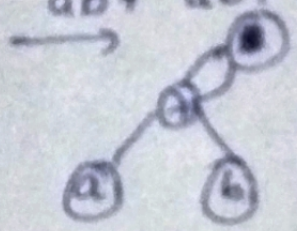
$E \rightarrow T$

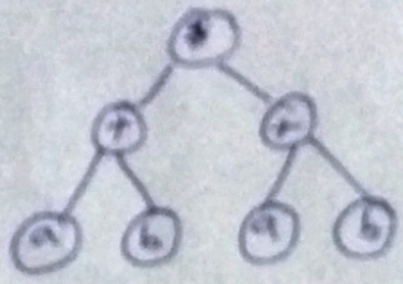
$T \rightarrow T * f$

DAG notation

$(a+b) * (a+b)$

ab+ ab+ *





If common sub expression is there, if it is already there then it refers otherwise creates it.

Three-address-code

x = a+b

t1 = a+b

z = t1

If any expression

A = B op C

For instance if (x==0)
printf("mam");

if x==0 then goto statement1
else/otherwise statement2

Relational expression.

3) while(exp) while(exp)
do
:
done.

example: a+b*c/d+e*f/h → postorder from syntax tree
DAG notation

ex: a+b*-(c-d)+a+b*-(c-d)

3 address code, unary minus

ab cd - - * + → a b c d - - * + +

t1 = c-d
t2 = -t1
t3 = b*t2
t4 = a+t3

t5 = c-d
t6 = -t5
t7 = b*t6
t8 = a+t7
t9 = t4+t8

↘ If repeated then they are optimized during code optimization

ex: (a+b) * (c-d)

ab+ → ab+ cd- *

t1 = a+b
t2 = c-d
t3 = t1*t2

ex: x = a * -(b+c)

x a bc+ - * =

t1 = b+c
t2 = -t1
t3 = a*t2
x = t3

ex: ~~for~~ (i=1; i<5; i++){
a[i] = i;
}

base address + i * size

1) i=1
2) if i<5 then goto 4
3) else goto 11
4) t1 = &a → base address
5) t2 = i * size(int)
6) t3 = t1+t2
7) *t3 = i
8) t4 = i+1
9) i = t4
10) goto 2
11) exit

Three Address Statement Representation.

1) Quadraples
2) Tripples
3) Indirect Tripples

operator	op1	op2	result	
-	c	d	t1	
-	t1		t2	
*	b	t2	t3	
+	a	t3	t4	t4 = a+t3
-	c	d	t5	t5 = c-d
-	t5		t6	t6 = -t5
*	b	t6	t7	
+	a	t7	t8	
+	t4	t8	t9	

Tripples:

Sequence number	op	op1	op2
00 (0)	-	c	d
11 (1)	-	(0)	
22 (2)	*	b	(1)
(3)	+	a	(2)
(4)	-	c	d
(5)	-	(4)	
(6)	*	b	(5)
(7)	+	a	(6)
(8)	+	(3)	(8)

Assignment: Explain all Intermediate code forms with example

Drawback:- If we shuffle, then we have to alter the op1 and op2 references also.

3) Indirect Tripples.

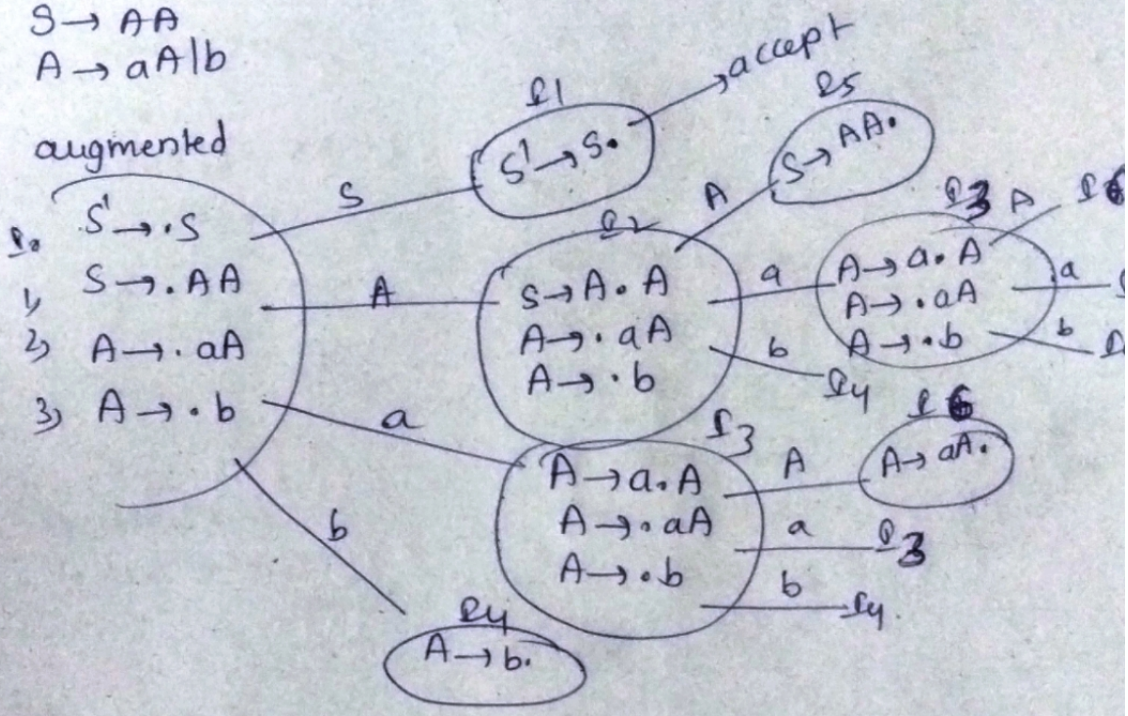
Stmt	pointer
(0)	00
(1)	11
(2)	22

LR(0) → Example:

$S \to AA$

$A \to aA \mid b$

augmented



	Action			GoTo	
	a	b	$	S	A
0	S_3	S_4		1	2
1			accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

I_0 on $S \Rightarrow 1$ I_0 on $A \Rightarrow 2$ I_0 on $a \Rightarrow I_3$ I_0 on $b \Rightarrow S_4$

I_1 on $\$ \Rightarrow$ accept I_2 on $A \Rightarrow 5$ I_2 on $a \Rightarrow S_3$ I_2 on $b = S_4$

I_3 on $A \Rightarrow 6$ I_3 on $a \Rightarrow S_3$

Reduce actions: For I_4, I_5, I_6 on all terminals keep reduce actio

SDT: A technique of compiler execution, where the source code translation is totally conducted by the parser, is known as syntax-directed translation.

The parser primarily uses a context-free-Grammar to check the input sequence and deliver output for the compiler's next stage. o/p could be either a parse tree or an abstract syntax tree

⇒ to interleave semantic analysis phase of the compiler, we use SDT with the syntax anysis

eg:- x or y and not z

15/03/2024
Thursday.

SDT
examples
S-
L-
Type checking
Semantic analysis
Intermediate

eg:- while(i<5)
{
a[i] = x*x;
i = i+1;
}

1. t1 = i<5 then goto 4
   else
   goto 11
4. t2 = x*x
5. t3 = &a
6. t4 = i*4
7. t3[t4] = t2
8. t5 = i+1
9. i = t5
10. goto 3
11. exit.

switch(x){
case '1' := S1
break;
=
=

eg:- L1 : S1
break
L2 : S2
break
test: if(x == 1) then goto L1
if(x == 2) then goto L2.