Bottom Up parser

→ Recursive parser, Reduction

→ RMD in reverse.

Root

$S \rightarrow aABe$

$A \rightarrow Abc/b$

$B \rightarrow d$

$\Rightarrow$ abbcde (leaf)

$\Rightarrow$ aAbcde [A → b] → handle

$\Rightarrow$ aAde [A → Abc]

$\Rightarrow$ aABe [B → d]

$\Rightarrow S$ // (root)

2m

Bottomup

- LR(k)
  - LR(0)
  - SLR
  - CLR
  - LALR
- operator precedence (operator parser)

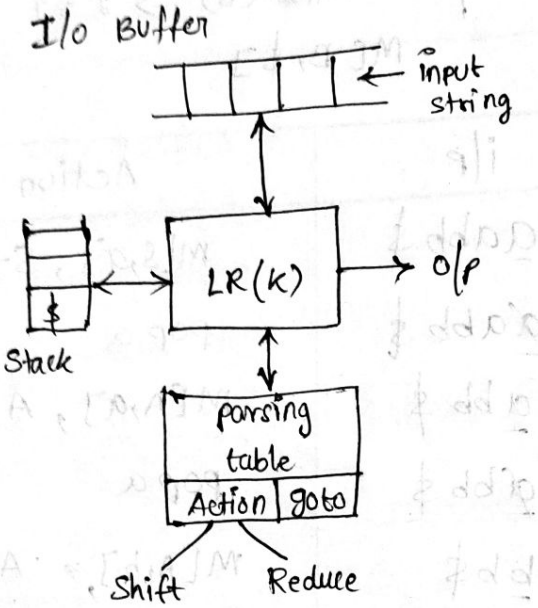
LR(k)

↓

L - Left to Right scanning

R - Reduction / derivation in reverse order.

K - No of lookahead symbols to pare the grammar



Aug

LR(0)

K=0

1) $S \rightarrow AA$

$A \rightarrow aA/b$

S1) I_0: $S' \rightarrow .S$ ← Augmented grammar

$S \rightarrow .AA$

$A \rightarrow .aA$

$A \rightarrow .b$

Canonical items (. should be added)

goto (I_0, S)

I_1: $S' \rightarrow S.$

goto (I_0, A)

I_2: $S \rightarrow A.A$

$A \rightarrow .aA$

$A \rightarrow .b$

(if non terminal after . write all production of the next symbol)

goto (I_0, a)

I_3 : $A \to a.A$
$A \to .aA$
$A \to .b$

goto (I_0, b)

I_4 : $A \to b.$

goto (I_2, A)

I_5 : $S \to AA.$

goto (I_2, a)

I_3 : $A \to a.A$
$A \to .aA$
$A \to .b$

goto (I_2, b)

I_4 : $A \to b.$

goto (I_3, A)

I_6 : $A \to aA.$

goto (I_3, a)

I_3 : $A \to a.A$
$A \to .aA$
$A \to .b$

goto (I_3, b)

I_4 : $A \to b.$

	Terminals (Action)			NT (goto)	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			Accepted		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

~~$S' \to S$~~
$S \to AA$
$A \to aA$
$A \to b$

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

0 I_1 : $S' \to S.$ $\Rightarrow r_0$
1 I_5 : $S \to AA.$ $\Rightarrow r_1$
2 I_6 : $A \to aA.$ $\Rightarrow r_2$
3 I_4 : $A \to b.$ $\Rightarrow r_3$

AUG

SLR(1) or LR(1)

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id.$

(1)
I_0 : $E' \to E$
$E \to E+T$
$E \to T$
$T \to T*F$
$T \to F$
$F \to (E)$
$F \to id.$

(2)
I_0 : $E' \to .E$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

goto (I_0, E)

I_1 : $E' \to E.$
$E \to E. + T$

goto (I_0, T)

I_2 : $E \to T.$
$T \to T. * F$

goto (I_0, F)

I_3 : $T \to F.$

goto $(I_0, ()$

I_4 : $F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

goto (I_0, id)

I_5 : $F \to id.$

goto $(I_1, +)$

I_6 : $E \to E+.T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

goto $(I_2, *)$

I_7 : $T \to T*.F$
$F \to .(E)$
$F \to .id$

goto (I_4, E)

I_8 : $F \to (E.)$
$E \to E. + T$

goto (I_4, T)

I_2: $E \to T.$
$T \to T. * F$

goto (I_4, F)

I_3: $T \to F.$

goto $(I_4, ()$

I_4: $F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

goto (I_4, id)

I_5 : $F \to id.$

goto (I_6, T)

I_9: $E \to E+T.$
$T \to T. * F$

goto (I_6, F)

I_3: $T \to F.$

goto $(I_6, ()$

I_4: $F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

goto (I_6, id)

I_5: $F \to id.$

goto (I_7, F)

I_{10}: $T \to T*F.$

goto $(I_7, ()$

I_4: $F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

goto (I_7, id)

I_5: $F \to id.$

goto $(I_8,))$

I_{11}: $F \to (E).$

goto $(I_8, +)$

I_6: $E \to E+.T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

goto $(I_9, *)$

I_7: $T \to T*.F$
$F \to .(E)$
$F \to .id$

0 I_1 $E' \to E.$ r_0
2 I_2 $E \to T.$ r_2
1 I_9 $E \to E+T.$ r_1
4 I_3 $T \to F.$ r_4
3 I_{10} $T \to T*F.$ r_3
5 I_{11} $F \to (E).$ r_5
6 I_5 $F \to id.$ r_6

follow(E) = {+,), $}
follow(T) = {*, +,), $}
follow(F) = {*, +,), $}

first(E) = first(T) = first(F) = {(, id}
first(T) = first(F) = {(, id}
first(F) = {(, id}

	terminals (Action)						NT (goto		
	+	*	(	)	id	$	E	T	F
I_0			S4		S5		1	2	3
I_1	S6					Accepted			
I_2		S7		r_2		r_2			
I_3	r_4			r_4		r_4			
I_4			S4	:	S5		8	2	3
I_5	r_6	r_6		r_6		r_6			
I_6			S4		S5			9	3
I_7			S4		S5				10
I_8	S6			S11					
I_9	r_1	S7		r_1		r_1			
I_{10}	r_3	r_3		r_3		r_3			
I_{11}	r_5	r_5		r_5		r_5			

Stack	i/p string	Action
$0	id * id + id $	M[0,id], shift 5
$0id5	* id + id $	M[5,*], r_6 F → id, RHS 1 2 symbol from stack apply 0 on F M[0,F] = 3
$0		
$0F3	* id + id $	M[3,*], r=4 T → F pop 2 symbol from stack T => T2 + => T2
$0T2	* id + id $	M[2,*] S7
$0T2*7	id + id $	M[7,id] S5
$0T2*7id5	+ id $	M[5,+] r_6 → F → id pop 2 symbol F => F10
$0T2*7F10	+ id $	M[10,+], r_3 → T → T*F There is 3 symbol. So apply pop 3 T → T2
~~$0T2~~	~~+ id $~~	

r_6 : F → id
1 symbol then
pop 1 symbol

$0T2	+id$	M[2,+], r2 → E→T E → E1
$0E1	+id$	M[1,+], s6
$0E1+6	id$	M[6,id], s5
$0E1+6id5	$	M[5,$], r6 → F→id F=F3
$0E1+6F3	$	M[3,$], r4 → T → F T → T4
$0E1+6T9	$	M[9,$], r1 → E → E+T E → E1
$0E1	$	M[1,$] accepted.

String cannot be empty do process until it is accepted

(2) SLR

$S \to L=R \mid R$

$L \to *R \mid id$

$R \to L$

i) $S' \to S$ ~~augmented~~ augmented grammar

$S \to L=R \mid R$

$L \to *R \mid id$

$R \to L$

ii) I_0: $S' \to .S$

$S \to .L=R$

$S \to .R$

$L \to .*R$

$L \to .id$

$R \to .L$

goto (I_0, S)

I_1: $S' \to S.$

goto (I_0, L)

I_2: $S \to L. =R$

$R \to L.$

goto (I_0, R)

I_3: ~~$S \to L. =R$~~

$S \to R.$

~~$L \to *. R$~~

~~$R \to .L$~~

goto $(I_0, *)$

I_4: $L \to *. R$

$R \to .L$

goto (I_0, id)

~~I_5: $S \to L. =R$~~

~~goto (I_0, id)~~

I_5: $L \to id.$

goto $(I_2, =)$

I_6: $S \to L= .R$

$R \to .L$

$L \to .*R$

$L \to .id$

goto (I_4, R)

~~I_7: $S \to L= .R$~~

~~$R \to .L$~~

~~goto (I_3, R)~~

I_8: $L \to *R.$

goto (I_4, L)

I_8: ~~$S \to L= .R$~~

$R \to L.$

goto $(I_4, *)$

I_9: $S \to L=R.$

$L \to *R$

$R \to .L$

$L \to .*R$

$L \to .id$

goto (I_4, id)

I_5: $L \to id$

goto (I_6, R)

I_7: $S \to L=R.$

goto (I_6, L)

I_8: $R \to L.$

goto $(I_6, *)$

I_4: $L \to *.R$

$R \to .L$

$L \to .*R$

$L \to .id$

goto (I_6, id)

I_5: $L \to id.$

0	$S' \to S.$	— I_1	— $r0$
1	$S \to L=R.$	— I_9	— $r1$
2	$S \to R.$	— I_3	— $r2$
3	$L \to *R.$	— I_8	— $r3$
4	$L \to id.$	— I_5	— $r4$
5	$R \to L.$	— (I_8 and I_2)	— $r5$

	=	*	id	$	S	L	R
I_0		S4	S5		1	2	3
I_1	S6/r5			Accepted			
I_2				r5			
I_3				r2			
I_4		S4	S5			8	7
I_5	r4						
I_6		S4	S5			8	9
I_7	r3			r3			
I_8	r5			r5			
I_9				r9			

follow (S) = { $ }
follow (L) = { =, $ }
follow (R) = { =, $ }

S6, r5 → shift reduce conflict

① Construct canonical LR parser (CLR) for the given string grammar and parse the input string

$S \to CC$
$C \to CC \mid d$
dd - input string

LR(0)
↳ SLR(1) - follow
↳ CLR → LR(0) + lookaheads = LR(1)

S1: $S' \to S$
$S \to CC$
$C \to CC$
$C \to d$

S2: canonical collection of LR(1) items

$I_0: S' \to .S, \$$ (LR(0) → lookahead)
first($) = {$}
$S \to .CC, \$$
first(C$) = first(C) = {c, d}
$C \to .CC, c/d$
$C \to .d, c/d$

goto (I_0, S)
$I_1: S' \to S., \$$

goto (I_0, C)
$I_2: S \to C.C, \$$
$C \to .CC, \$$
$C \to .d, \$$

goto (I_0, c)
$I_3: C \to C.C, c/d$
$C \to .CC, c/d$
$C \to .d, c/d$

goto (I_0, d)
$I_4: C \to d., c/d$

goto (I_2, C)

I_5 : $S \to CC.,\$$

goto (I_2, C)

I_6 : $C \to c.C, \$$

$C \to .cC, \$$

$C \to .d, \$$

goto (I_2, d)

I_7 : $C \to d., \$$

goto (I_3, C)

I_8 : $C \to cC., c/d$

goto (I_3, C)

I_3 : $C \to c.C, c/d$

$C \to .d, c/d$

goto (I_3, d)

I_4 : $C \to d., c/d$

goto (I_6, C)

I_9 : $C \to CC.\$$

goto (I_6, C)

I_6 : $C \to c.C, \$$

$C \to .cC, \$$

$C \to .d, \$$

goto (I_6, d)

I_7 : $C \to d., \$$

	c	d	$	S	C
I_0	S_3	S_4		1	2
I_1			Accepted		
I_2	S_6	S_7		5	
I_3	S_3	S_4		8	
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7		9	
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		

I_1 : $S' \to S., \$$

I_5 : $S \to CC., \$$

I_7 : $C \to dd., \$$

I_4 : $C \to d., c/d$

I_8 : $C \to cC., c/d$

I_9 : $C \to cC., \$$

$S' \to .S$ 0

$S \to CC.$ - 1 - r_1

$C \to cC$ - 2 - r_2

$C \to d$ - 3 - r_3

Stack	I/p string	Action
$0	dd$	M[0,d], S_4
$0d4	d$	M[4,d], r_3, C→d
$0C2	d$	M[2,d], S_7
$0C2d7	$	M[7,$], r_3, C→d
$0C2C5	$	M[5,$], r_1, S→CC
$0S1	$	Accepted

LALR (similar to CLR) — smaller number of states, saving

Look Ahead LR

LR(1) = LR(0) + LA

I_1 $S' \to S_0$, \$ c/d

I_2 , \$ c/d

I_3 & I_6 , \$ c/d I_{36} → combine states

I_4 & I_7 , \$ c/d I_{47} → combine states

I_5

I_8 & I_9 , \$ c/d I_{89}

	c	d	\$	S	C
I_0	S_{36}	S_{47}		1	2
I_1			Accepted		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r3	r3	r3		
I_5			r1		
I_{89}	r2	r2	r2		

I_1 : $S' \to S, \$$

I_5 : $S \to CC.\$$

I_{47} { I_7 : $C \to d, \$$ } \$/c/d ; I_4 : $C \to d., c/d$ } c/d

I_{89} { I_8 : $CC., c/d$ } \$/c/d ; I_9 : $CC., \$$

I_{36} → c/d/\$

I_3 : $C \to c.C, c/d$
$C \to .CC, c/d$
$C \to .d, c/d$

I_6 : $C \to c.C, \$$
$C \to .CC, \$$
$C \to .d, \$$

0 $S' \to .S$
1 $S \to .CC$ r1
2 $C \to CC$ r2
3 $C \to d$ r3

M[1,\$] is always accept

Stack	i/p string	Action
\$0	dd\$	M[0,d] S47
\$0d47	d\$	M[47,d] r3, C→d
\$0C2	d\$	M[2,d] S47
\$0C2d47	\$	M[47,\$] r3, C→d
\$0C2C5	\$	M[2,\$] r1, S→CC
\$0S1	\$	Accepted

(1) Construct CLR and LALR for grammar

$S \to L=R \mid R$
$L \to *R \mid id$
$R \to L$
i/p string: id = id

S1:
$S' \to S$
$S \to L=R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

I_0 : $S' \to .S, \$$
$S \to .L=R, \$$
$S \to .R, \$$
$L \to .*R, =/\$$
$L \to .id, =/\$$
$R \to .L, \$$

goto (I_0, S)
I_1: $S' \to S., \$$

goto (I_0, L)
I_2: $S \to L.=R, \$$
$R \to L., \$$

goto (I_0, R)
I_3: $S \to R., \$$

goto (I_0, *)
I_4: $L \to *.R, =/\$$
$R \to .L, =/\$$
$L \to .*R, =/\$$
$L \to .id, =/\$$
goto (I_0, id)

I_5: $L \to id., =/\$$

goto (I_2, =)
I_6: $S \to L=.R, \$$
$R \to .L, \$$
$L \to .*R, \$$
$L \to .id, \$$
goto (I_4, R)

I_7: $L \to *R., =/\$$

goto (I_4, L)
I_8: $R \to L., =/\$$

goto (I_4, *)
I_4: $L \to *.R, =/\$$
$R \to .L, =/\$$
$L \to .*R, =/\$$
$L \to .id, =/\$$
goto (I_4, id)

I_5: $L \to id., =/\$$

goto (I_6, R)
I_9: $S \to L=R., \$$

goto (I_6, L)
I_{10}: $R \to L., \$$

goto (I_6, *)
I_{11}: $L \to *.R, \$$
$R \to .L, \$$
$L \to .*R, \$$
$L \to .id, \$$
goto (I_6, id)

I_{12}: $L \to id., \$$

goto (I_{11}, R)
I_{13}: $L \to *R., \$$

goto (I_{11}, L)
I_{10}: $R \to L., \$$

goto (I_{11}, *)
I_{11}: $L \to *.R, \$$
$R \to .L, \$$
$L \to .*R, \$$
$L \to id, \$$
goto (I_{11}, id)
I_{12} $L \to id., \$$

	=	*	id	$	S	L	R
0		S4	S5		1	2	3
1				Accepted			
2	S6			r5			
3				r2			
4		S4	S5			8	7
5	r4			r4			
6		S11	S12			10	9
7	r3			r3			
8	r5			r5			
9				r1			
10				r5			
11		S11	S12			10	13
12				r4			
13				r3			

$0	id = id $	S5
$0id5	= id $	r4 , L→id
$0L2	= id $	S6
$0L2=6	id $	S12
$0L2=6id12	$	r4, L→id
$0L2=6R9	$	r1, S→L=R
$0S1	$	Accepted.

0 $S' \to S$
1 $S \to L=R$ r1
2 $S \to R$ r2
3 $L \to *R$ r3
4 $L \to id$ r4
5 $R \to L$ r5

I_0 : $S' \to .S, \$$
I_1 $S' \to S., \$$
I_3 $S \to R., \$$
I_5 $L \to id., \$$
I_7 $L \to *R., =/\$$
I_8 $R \to L., =/\$$
I_9 $S \to L=R., \$$
I_{10} $R \to L., \$$
I_{12} $L \to id, \$$

Conflicts

(2) 2M

1) Shift - Reduce Conflict (S1, R2) → precedence level
2) Reduce - Reduce conflict (R3, R5)

} Can be solved by
1) Gold parser
2) Yacc parser

Error handling

* Good compiler - helps in identifying and locating errors.
* Errors may be:
  - Lexical : misspelling of identifiers, keywords
  - Syntatic : expression with incorrect format.

Syntax Error handling

* Error detection and recovery - Syntax analysis phase.
* Accuracy of parsing methods
* Nature of errors.
* **Goals of the error handler in a parser**

* Must report the presence of errors
* Recovery quickly
* Fast
* LL and LR methods detect error as early as possible - viable prefix property.
* **Common errors**
  1) punctuation errors (, and ;)
  2) operator and operand errors
  3) keyword errors
* Recovery should not lead to spurious (meaningless) errors.
  - Skipping declarations.

(2) 2M

* **Error Recovery Strategies**

1) **panic Mode**
   * Discards input symbols until tokens in the synchronizing set are encountered
   * skips input
   * synchronizing set must be chosen carefully
   * Simple.
2) **phrase level**
   * Local correction, replaces prefix
   * should not lead to infinite loops

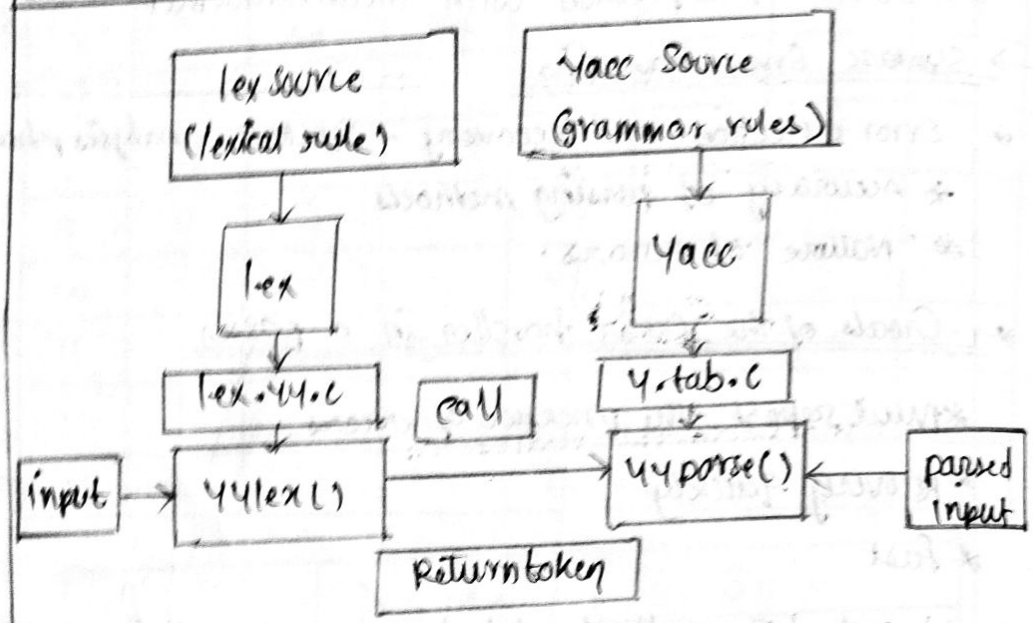
3) **Error productions**

* Augment grammar
* can generate error diagonstics

2) **Global Correction**

* minimal sequence of changes needed to transform x to y
* costly.

* **YACC – Yet Another Compiler Compiler**




```
%{
  #include <ctype.h>
%}
%%
line : expr '\n' { printf("%d\n", $1); }
  ;
expr : expr '+' term { $$ = $1 + $3; }
  | term
  ;
term : term '*' factor { $$ = $1 * $3; }
  | factor
factor : '(' Expr ')'  { $$ = $2; }
  | DIGIT
%%
```

```
yylex()
{
  int c;
  c = getchar();
  if (isdigit(c))
  {
    yylval = c - '48';
    return DIGIT;
  }
  return c;
}
```

E=1
T=2
F=3

E → E+T | T
T → T*F | F
F → (E) | digit.

→ Bottom up

LR(k)

LR(0) SLR CLR LALR Operator precedence

* operator precedence parser.

operator grammer :-

↳ mathematical Exp

No two NT should be adjacent - RHS

No symbol at RHS

E → E+E | E*E
E → EAE | id
A → + | *
E → E (+) E
E → E (*) E

→ Steps for operator precedence parser

i) Check the grammar is operated grammar or not
ii) Construct operator precedence relation table based on terminals
iii) parse the input string
iv) Construct parse tree.

Eg: E → E+E | E*E | id , id+id*id$

i/p

stack	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	-	>
$	<	<	<	Accepted

id, a, b → high
$ → low
+ > +
* > *
* > +

stack high priority

i/p string + has low priority

Stack	Relation	i/p string	Action
$	<	id+id*id$	push i/p, shift id
$id	>	+id*id$	pop, Reduce E→id
$E	<	+id*id$	shift +
$E+	<	id*id$	shift id
$E+id	>	*id$	Reduce E→id
$E+E	<	*id$	shift *
$E+E*	<	id$	Shift id
$E+E*id	>	$	Reduce E→id
$E+E*E	>	$	Reduce E→E*E
$E+E	>	$	Reduce E→E*E
$E		$	Accepted,

with parenthesis

Stack \ i/p	+	*	(	)	id	$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	<	<	<	=	<	—
)	>	>	—	>	—	>
id	>	>	—	>	—	—
$	<	<	<	—	—	Accepted

UNIT - 3

SYNTAX DIRECTED TRANSLATION AND INTERMEDIATE CODE GENERATION

01/10/24

Syntax directed definitions - construction of syntax Tree - Bottom-up - Evaluation of S-Attribute definitions - Design of predictive translator - Type systems - specification of a simple type checker - Equivalence of type expressions - type conversions. Intermediate languages: Syntax Tree, Three Address code, types and declarations, Translation of Expressions, Type checking, Backpatching.

→ **Syntax directed (grammar) definition (SDD).**

$E \to E+T$ } → Gramar
$E \to T$ } CFG
+
Semantic Rules - meaning of production

CFG

$E.val = E.val + T.val$
$E.val = T.val$

Grammar Symbols
+
Attributes
(reference no string)

SDD = CFG + SR

If X is a grammar symbol, a be any string X.a is SDD

→ **Types of attributes**

1) **synthesized attribute**

$S \to ABC$
$S.syn = A.syn$
$S.syn = B.syn$ $S.Syn = C.Syn$

2) **Inherited attributes**

$S \to ABC$
$B.i = S.i$
$B.i = A.i$
$B.i = C.i$

SDD

* SDD is a Context free grammar with Semantic rules the attributes are associated with grammar rules and Semantic rules are associated with productions,
* every symbol contains an attributes and
* every production contains a symmentic rule
* symantic rule provides meaning for the production
* attributes may be numbers, strings, data types and references.
* Types of attributes.

(1) synthesized attribute.

if a node takes value from its children the parent node is called the synthesized attribute.

(2) Inherited attributes.

if a node takes the value from its value or siblings Then the child node is called inherited attribute.

(1) synthesized attributes.

→ (1) S-attributed SDD (or) S-attributed definition (or) S-attributed grammar

→ (2) L-attributed SDD (or) L-attributed definition (or) L-attributed grammar.

S-attributed SDD	L-attributed SDD
* SDD That uses only synthesized attribute is called S-attributed.	A SDD that uses both synthesized and inherited attributes but each inherited attribute is restricted to inherit from parent or left sibling only is called L-attributed SDD.
Eg:- S → ABC S.syn = A.syn S.syn = B.syn S.syn = C.syn	A → XYZ Y.i = A.i ✓ Y.i = X.i ✓ Y.i = Z.i ✗
* Symantic actions are always placed at right end	Symentic action are placed at any place in RHS.

of the production and it is called postfix SDD.	
attributes are evaluated with bottom-up-parsing.	attributes are inherited by traversing parse tree depth first left to right order.

Construction of syntax trees

1. operations for given symbol / expressions
2. SDD (production + Sematic Rules)
3. syntax tree Construction

operator → mknode(operator(op), left, right)
identifier → mkleaf(id, entry_id)
number → mkleaf(num, value)

operators → mknode
identifiers, number → mkleaf

Eg:- x * y + 5 - z

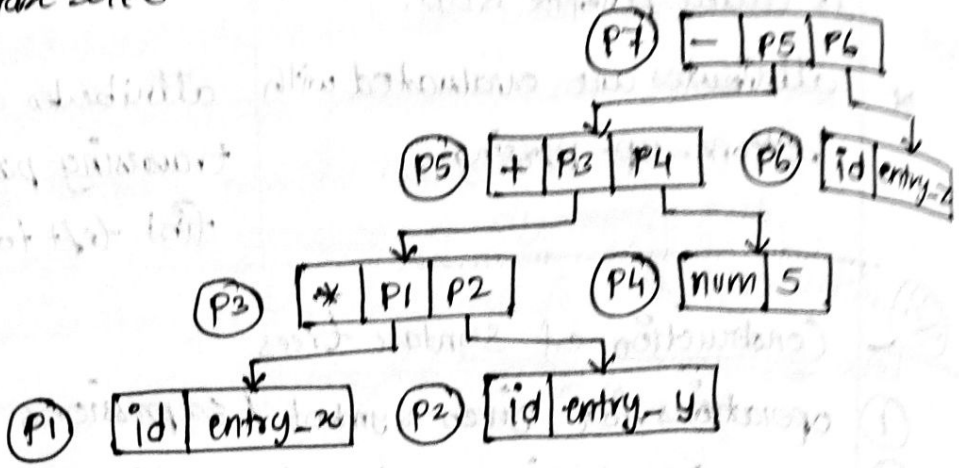
(1)

Symbol	operations
x	p1 = mkleaf(id, entry_x)
y	p2 = mkleaf(id, entry_y)
*	p3 = mknode('*', p1, p2)
5	p4 = mkleaf(num, 5)
+	p5 = mknode('+', p3, p4)
z	p6 = mkleaf(id, entry_z)
-	p7 = mknode('-', p5, p6)

(2) SDD (production + Sematic rules) E.attribute → grammar symbol, attribute → any attribute, dot operator

production	Sematic rules
E = E1 - T	E.node = E1.node - T.node (or) mknode(-, E1.node, T.node)
E = E1 + T	E.node = E1.node + T.node (or) mknode(+, E1.node, T.node)
E = E1 * T	E.node = E1.node * T.node (or) mknode(*, E1.node, T.node)
E = T	E.node = T.node
T = id	T.node = mkleaf(id, entry_id)
T = num	T.node = mkleaf(num, num.value)

(3) Syntax tree



2) Construct the Syntax for a-4+c

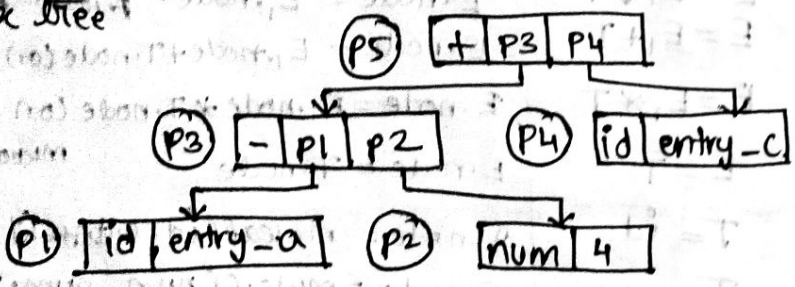
(1)

Symbol	operations.
a	p1 = mkleaf (id, entry_a)
4	p2 = mkleaf (num, 4)
-	p3 = mknode ('-', p1, p2)
c	p4 = mkleaf (id, entry_c)
+	p5 = mknode ('+', p3, p4)

(2) SDD

production	Semantic rule
E = E1 + T	E.node = E1.node + T.node = mknode (+, E1.node, T.node)
E = E1 - T	E.node = E1.node - T.node = mknode (-, E1.node, T.node)
E = T	E.node = T.node
T = id	T.node = mkleaf (id, entry_id)
T = num	T.node = mkleaf (num, num.value)

(3) Syntax tree



SDT - Syntax Directed Translation Schemes.

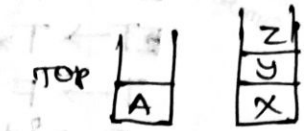
↓

Semantic action

↓

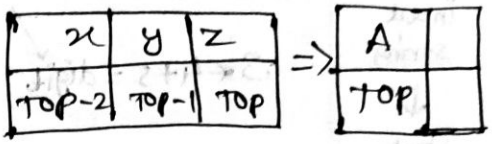
SDD

$A \rightarrow xyz$



⊛ Stack implementation

Evaluated → Bottom up / top down



SDT

It is the complementary notation of SDD - Syntax directed definition can be implemented using SDT - Syntax directed translation schemes.

* SDT can be defined by the combination of CFG with program segments emperded in production bodies and it is called semantic actions
* Semantic actions can appear anywhere in the production body
* Semantic actions are represented by ~~eper~~ { }
* CFG: $E \rightarrow E+T$ — SDD: $E.node = E_1.node + T.node$ — SDT: $\{E.val = E_1.val + T.val\}$

Evaluation of Translation Schemes

* SDD can be evaluated by Bottom up evaluation and top down evaluation for S attributed and L attributed definitions.

Grammar
- ~~top down~~ (Bottom up) LR — S attributed (parent can take the value of child node)
- LL (top down) — L attributed (child can take the value of sibling along with parent)

* Bottom up evaluation for S-attributed deffintion
* Bottom up evaluation is done by stack implementation.
* When there is a reduction semantic action is carried out by using a double stack

CFG: $L \to E_n$ → assignment statement (SDD need for SDT)

$E \to E+T$

$E \to T$

$T \to T*F$

$T \to F$

$F \to id$

Start symbol

SDD

1) $L.val = E_n.val$
2) $E.val = E.val + T.val$
3) $E.val = T.val$
4) $T.val = T.val * F.val$
5) $T.val = F.val$
~~$F.val = id.$~~
6) $F.val = digit.lexval$

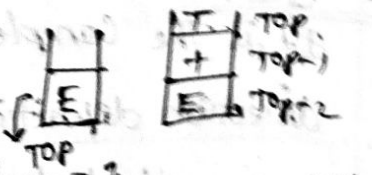
Bottom UP

take ↓ input string ↓ Start symbol

$3*4+5 = digit$

SDT

1) {print (val[TOP,]};



2) { val[nTOP,] = val[TOP-2] + val[TOP] }

4) { val[nTOP] = val[TOP-2] * val[TOP]

S-attributed $3*4+5 = digit$

$id_1 * id_2 * id_3$

T*F E=E+T

12 | 12+5 = 17

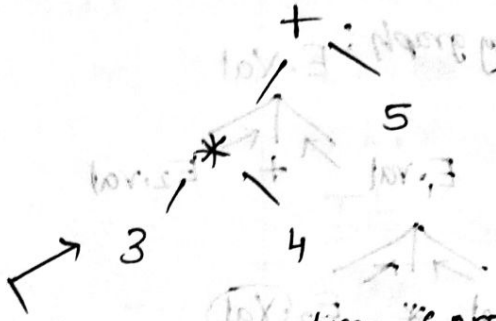
E=12

E→T, ~~F→12~~

Stack implementation.

input	Stack	value	Action.
3*4+5$	–	–	–
*4+5$	3	3	Reduce F→3 (F→id)
*4+5$	F	3	–
*4+5$	T	3	Reduce T→F (F→id)
4+5$	T*	3	–
+5$	T*4	3,4	Reduce F→4 (F→id)
+5$	T*F	12	–
+5$	T	12	Reduce T→T*F
+5$	E	12	Reduce E→T
5$	E+	12	–
$	E+5	12	–

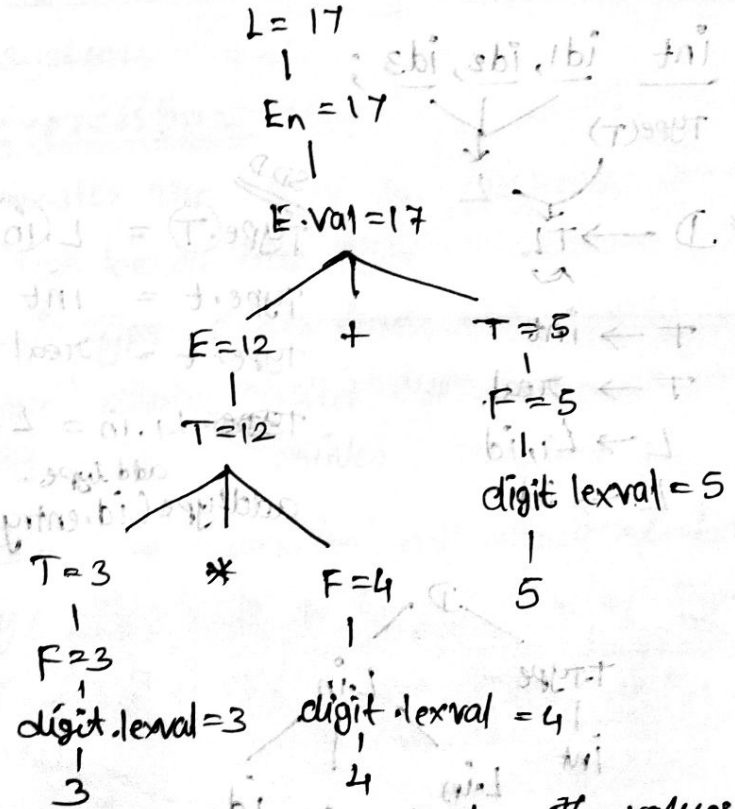
$	E + F	12, 5	F → 5 (F → id)
$	E + T	12, 5	T → F
$	E	17	E → E + T
$	E	17	—
$	E_n	17	L → E_n



Abstract parse tree (APT)

(2)

2m * **Annotated parse tree**



** It is a parse tree that displays the values of attributes at each node.

* The process of computing attributes values at the nodes is called annotating or decorating the parse tree.

* information for annotated parse tree.

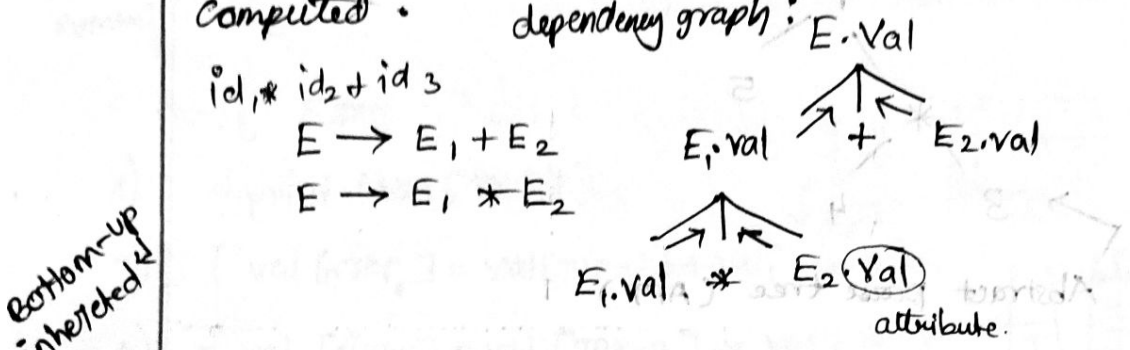
1) attributes, inherited attribute.
2) Semantic phase - deals with SDD, SDT.

3) Dependency graph - a dependency graph determines how the values of the attributes can be computed.

* a) The order of computations that depends on dependency graph is induced by Semantic rules.

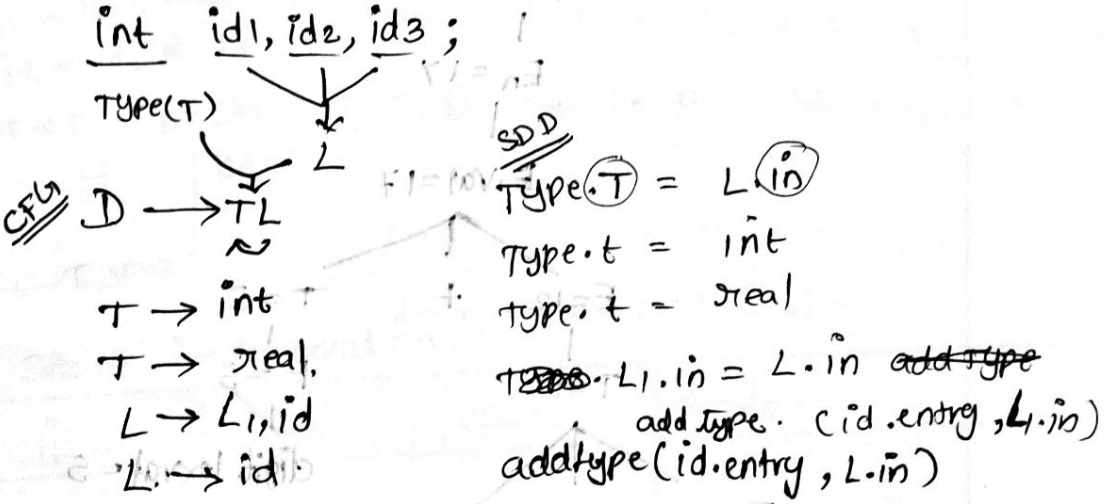
* b) Annotated phase tree shows the values of the attributes where as dependency graph determine how the values are computed.

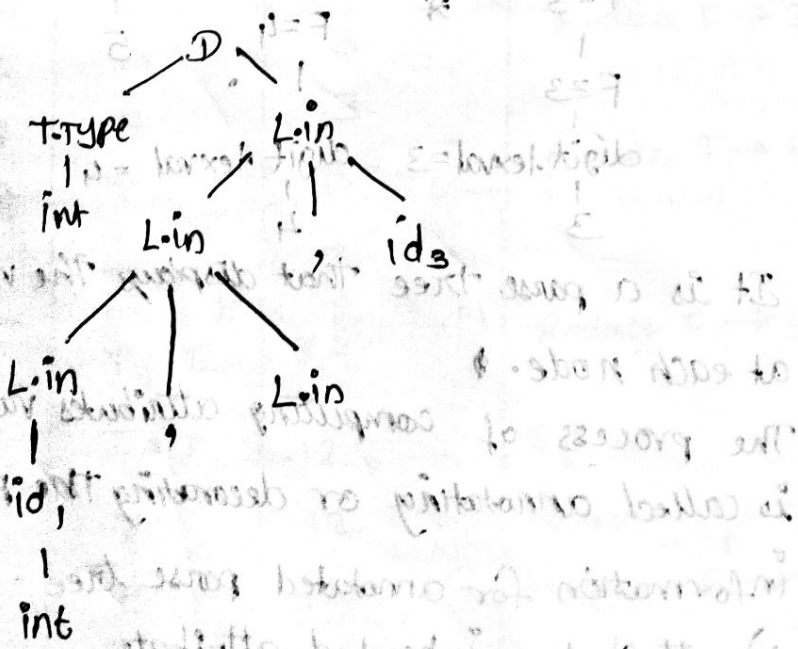
dependency graph :



Bottom-up inherited

→ Inherited attribute





I/p string	stack	Action
int p,q,r	—	—
pq,r	int	Reduce T → int
p,q,r	T	—
,q,r	TP	Reduce L → id → p
.q,r	TL,	—
,r	TL,q	Reduce ~~L → id → q~~ L → L1,id (2)
r	TL,	—
—	TL,r	reduce L → L1,id (2)
—	TL	D → TL
—	D	start symbol.

(2) 2m

Type expressions and type systems

~~Type systems~~

(1) Type expressions

* it denotes the type of language construction and it can be of two type
  1) basic type (int, float, char, boolean)
  2) Type name, called type construction.
     eg:- Array, pointer, structure :
* if T is a type expression then array of I,t ~~(array)~~ (array[I,T)) is a type system of type expression where ~~I~~ I is the index and T is the type expression
* notation for type expression.

(array (1, ..., 10), T)

TE ⇐ (array[1,...,10], int)

TE → TS

T

(2) Type systems

* it is the collection of rules for assigning type expressions

Components of type systems.

1) Basic types
2) Type construction
3) Type equivalence < structured equivalence (eg: struct a,b)
   name equivalence (char a; a='A'; b=a;)
4) Type checker - it is a implementation of type system.

→ Arrays int [2][3]

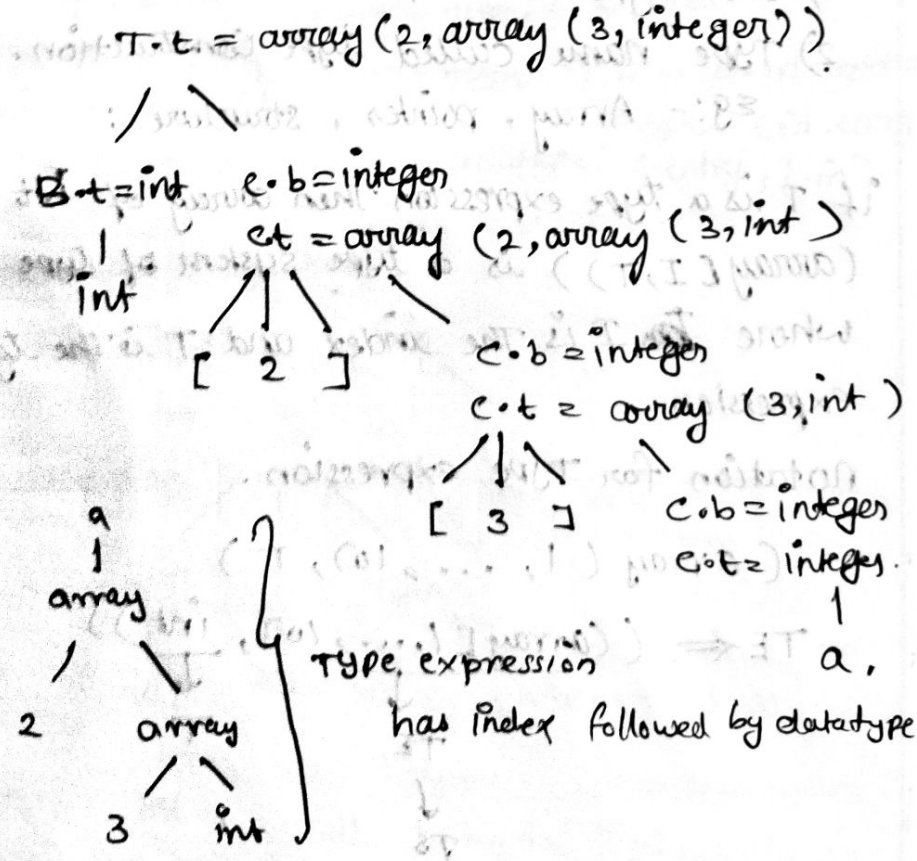
production	Semantic Rules
T → Bc	T.t = C.t
B → int	B.t = integer
B → float	B.t = float
C → [NUM] C_1	C.t = array (num.val, C_1.t)
C → a	C_1.b = C.b e.t = C.b

SDD

CFG

Synthesised, inherited attribute

→ Annotated parse tree.



has index followed by datatype.

EQUIVALENCE OF TYPE EXPRESSIONS

* "If two type expressions are equal then return a certain type else return type_error".
* Therefore we should have a precise definition of when two type expressions are equivalent.
* Ambiguities ....
* two types of type expression
  1. STRUCTURAL EQUIVALENCE OF TYPE EXPRESSIONS.
  2. NAME EQUIVALENCE

(1) Structural equivalence

* as long as type expressions are built from the basic structure equivalence → structural equivalence

Eg:
```
1. Function sequiv (s,t) : boolean ;
   begin
2. if s and t are the same basic type then
3. Return true .
4. Else if s = array (s1,s2) and t = array (t1,t2) then
5. Return sequiv (s1,t1) and sequiv (s2,t2)
6. Else if s = s1 * s2 and t = t1 * t2 then
7. Return sequiv (s1,t1) and sequiv (s2,t2)
8. Else if s = pointer (s1) and t = pointer (t1) then
9. Return sequiv (s1,t1)
10. Elseif s = s1 → s2 and t = t1 → t2 then
11. Return sequiv (s1,t1) and sequiv (s2,t2)
    else
12. Return false
    end;
```

(index, datatype)

* Example

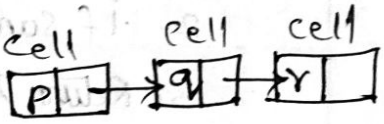
Type constructors	encoding
pointer	01
array	10
freturns	11

(2) Name EQUIVALENCE

VARIABLE	TYPE EXPRESSION
NEXT	link
last	link
p	pointer (cell)
q	pointer (cell)
r	pointer (cell)

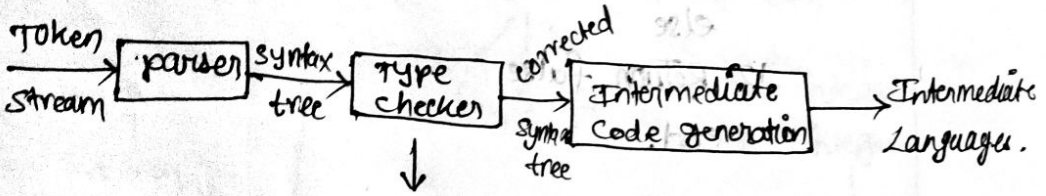
* Confusion arise in pascal from the fact that many implementation associate an implicit type name with each declared identifier. If the declaration contains a type expression that is not a name, a fresh implicit name is created every time a type expression appears in a variable declaration.

Ex:- type link = ↑cell;
np = ↑cell;
nqr = ↑cell;
var next : link;
last : link;
p : np;
q : nqr;
r : nqr;



→ Type checking

Implementation of type systems.



⊗ verifies types of constructs & matches
↓
ptr, array, fund, records.

* One of the most important semantic aspects of compilation.
* Allows the programmer to limit what types may be used in certain circumstances.

* Assigns types to values.
* Determine whether these values are used in an appropriate manner.
* simplest situation: check types of objects and report a type-error in case of a violation.
* Complex situation: incorrect type of objects

Static checking	Dynamic checking
* type checking done at compile time	performed during program execution
* properties can be verified before program run	permits programmers to be less concerned with types.
* can catch many common errors	mandatory in some situations such as, array bounds check.
* Desirable when faster execution is important	more robust and cleaner code.

① rules for type checking.
② type conversion.
③ overloading of functions and operators.
④ type interface and polymorphic functions.
⑤ An Algorithm for unification.

① **Rules for type checking.**

type checking can take on two forms:
1) synthesis
2) inference.

1) synthesis. builds up the type of an expression from the type of its subexpressions. It requires names to be declared before they are used. The type of $E_1 + E_2$ is defined in terms of the type of E_1 and E_2. A typical rule for synthesis

eg:- $E \rightarrow E_1 + E_2$; $E \rightarrow E_1(E_2)$

int ← s ← t → int

rule:

if f has type $s \rightarrow t$ and x has type s, then expression $f(x)$ has type t.

Here, f and x denote expression, and $s \rightarrow t$ denotes a function from s to t.

This rule for function with one argument carries the function.

* A typical rule for type inference has the form

rule:

if $f(x)$ is an expression,

then for some α and β, f has type $\alpha \rightarrow \beta$ and x has type α.

eg:- int id1, id2, id3 ;

type $\alpha \rightarrow \beta$

int ↓ x — int

* type inference is needed for language ML.

Type checking of Expressions.

* use synthesized attribute 'type' for the non-terminal E representing an expression

Expression	Action (~~semantic action~~)
$E \rightarrow id$	E.type ← lookup(id.entry)
$E \rightarrow E_1 \text{ op } E_2$	E.type ← if E_1.type = E_2.type then E_1.type else type.error
$E \rightarrow E_1 \text{ relop } E_2$	E.type ← if E_1.type = E_2.type then boolean else type.error.
$E \rightarrow E_1[E_2]$	E.type ← if E_2.type = integer and E_1.type = array(s,t) then t else type.error
$E \rightarrow E_1 t$	E.type ← if E_1.type = pointer(t). then t else type.error.

* Statements normally do not have any value, hence of type void.
* for propagating type error occurring in some statement nested deep inside a block, a set of rules needed.

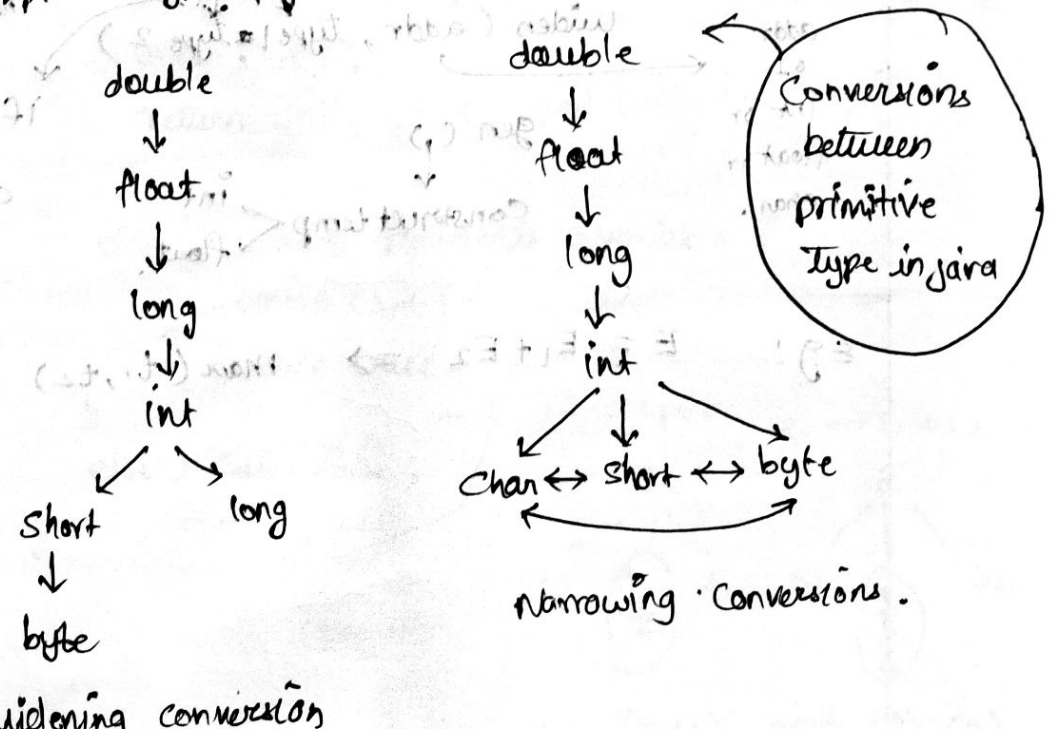
$S \rightarrow id = E$	S.type ← if id.type = E.type then void else type.error.
$S \rightarrow$ if E then S_1	S.type ← if E.type = boolean then S_1.type else type.error.
$S \rightarrow$ while E do S	S.type ← if E.type = boolean then S1.type else type.error.
$S \rightarrow S_1 ; S_2$	S.type ← if S_1.type = void and S_2.type = void then void else type.error.

* A function call is equivalent to the application of one expression to another.

$E \rightarrow E_1(E_2)$	E.type ← if E_2.type = s and E_1.type = s → t then t else type.error.

(2) Type conversion. (specification of Expression

* Conversion from one type to another is said to be implicit if it is done automatically by the compiler. Implicit type conversions, also called coercions.



* It is also called type casting
* Type casting is basically the conversion of one data type to another.

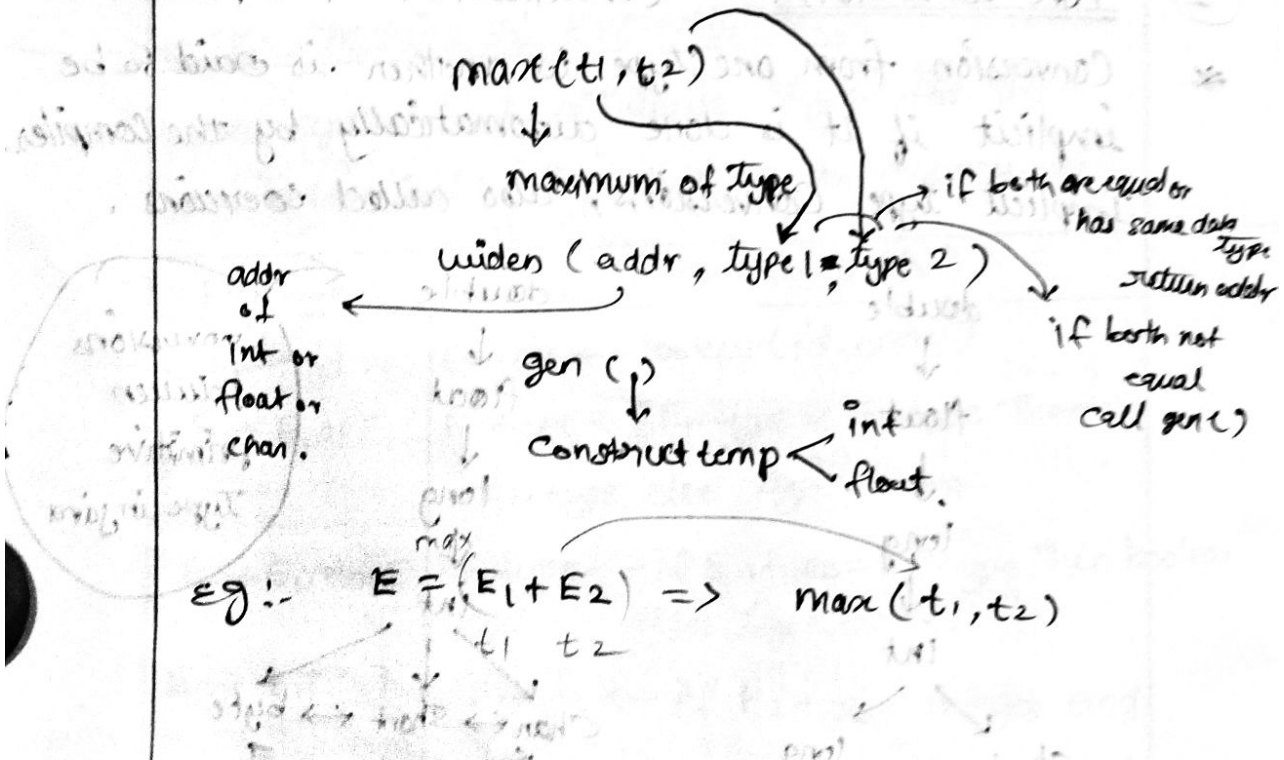
→ Types: ① implicit type conversion (type coercion)

If the compiler converts the one type of data to another automatically with no data lost.

eg:- char c='A'
printf("%d", c)

② Explicit type conversion

* When data of one type is converted explicitly to another type with the help of programming constraints is called explicit type conversion it has data loss.
* higher data type cannot be converted to the lower data type.



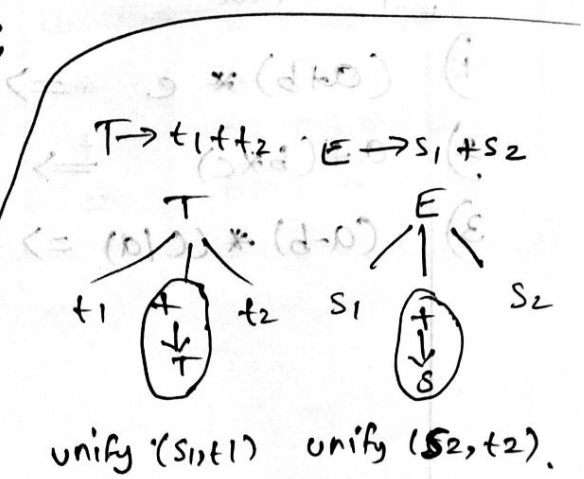
```
Addr widen (Addr a, Type t, Type w)
if (t = w) return a;
else if (t = integer and w = float) {
    temp = new Temp();
    gen (temp '=' '(float)' a);
    return temp;
}
else error;
}
```

Semantic action

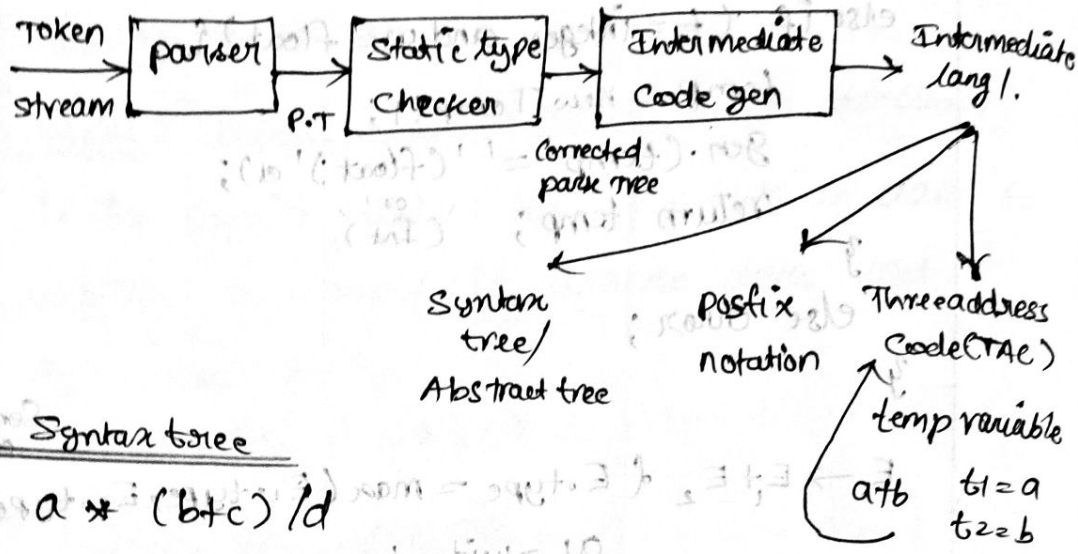
```
E → E1 + E2 { E.type = max (E1.type, E2.type);
    a1 = widen (E1.addr, E1.type, E.type);
    a2 = widen (E2.addr, E2.type, E.type);
    E.addr = new Temp();
    gen (E.addr '=' a1 '+' a2); }
```

(X) Algorithm : unification Algorithm.

```
boolean unify (Node m, Node n) {
    s = find (m); t = find (n);
    if (s = t) return true;
    else if (nodes s and t represent the same basic type)
        return true;
    elseif (s is an op-node with children s1 and s2
        and t is an op-node with children t1 and t2) {
        union (s, t);
        return unify (s1, t1) and unify (s2, t2);
    }
    else if s or t represents a variable {
        union (s, t);
        return true;
    }
    else return false;
}
```



Intermediate code generation

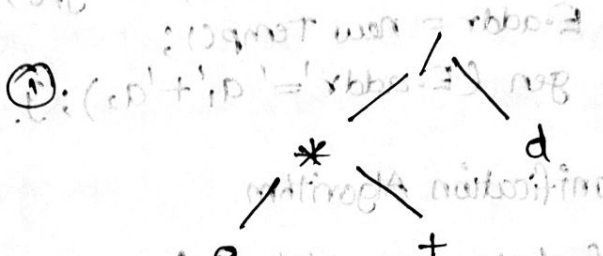


Syntax tree

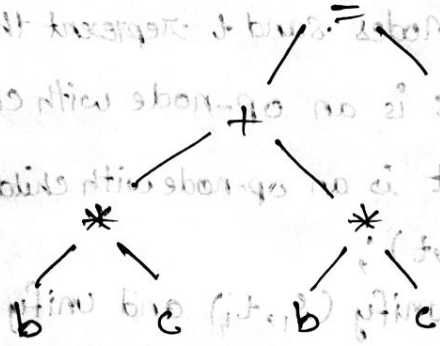
1) a * (b+c) /d

2) a=b * c + b * - c

(1)



(2)



postfix

1) (a+b) * c => ab+c*

2) a+(b*c) => abc*+

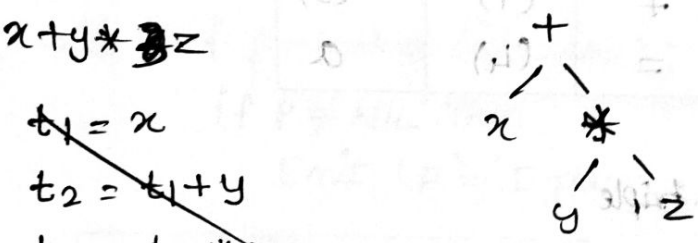
3) (a-b) * (c/d) => ab-cd/*

* Three address code should contain atmost 3 address and atmost one operator on RHS

* Types of Three address code (3 types)

(1) quadrauple
(2) Triple
(3) Indirect Triple

1) $a = x + y * z$



$t_1 = y * z$
$t_2 = x + t_1$
$a = t_2$

$t_1 = x$
$t_2 = t_1 + y$
$t_3 = t_2 * z$

2) $a = b * -c + b * -c$

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = -c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

(1) quadriple has four fields: 1) op
2) arg1
3) arg2
4) result

Forms of Intermediate Language

$a = b * -c + b * -c$

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = -c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$.

	op	arg1	arg2	result
(0)	uminus	c	–	t_1
(1)	*	b	t_1	t_2
(2)	uminus	c	–	t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5	–	a

2) Triple

	OP	arg1	arg2
(0)	uminus	c	–
(1)	*	b	(0)
(2)	uminus	c	–
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	(4)	a

3) Indirect triple

ptr table

100	(0)
101	(1)
102	(2)
103	(3)
104	(4)
105	(5)

ptr	OP	arg1	arg2
100	uminus	c	–
101	*	b	(100)
102	uminus	c	–
103	*	b	(102)
104	+	(101)	(103)
105	=	(104)	a

→ TRANSLATION EXPRESSION (OR) THREE ADDRESS CODE FOR EXPRESSION (OR) SDD - SYNTAX DIRECTED DEFINITION OF THREE ADDRESS CODE FOR EXPRESSION

(-X) AUG

(1) Assignment statement
(2) Boolean statement
(3) Array reference statement

(1) Assignment Statement

$S \rightarrow id := E$
$E \rightarrow E_1 + E_2$
$E \rightarrow E_1 * E_2$
$E \rightarrow (E)$
$E \rightarrow id.$

production :- $S \rightarrow id := E$

{ p = lookup (id_name);
if p ≠ NIL then
Emit (p '=' E.place);
else
error; }

$E \rightarrow E_1 + E_2$ { E.place := newTemp();
Emit (E.place := E1.place + E2.place);

$E \rightarrow E_1 * E_2$ { E.place := newTemp();
Emit (E.place := E1.place * E2.place); }

$E \rightarrow (E)$ { E.place := (E1.place); }

$E \rightarrow id$ { p = lookup (id_name);
if p ≠ NIL then
Emit (p := E.place);
else
Error; }

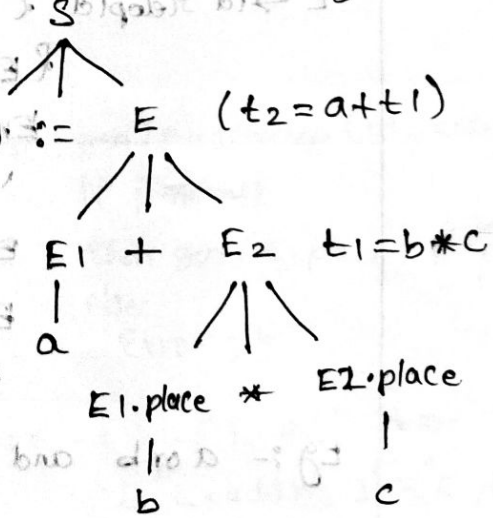
Eg: a+b*c <=> x=id+id*id

* Where p returns the entry for id_name in symbol table
* Emit is a function appends 3 address code to be output file. or it ~~a~~ reports error.
* newTemp : construction that generate new temporary variable
* E.place gives the value of E in symbol table.

Eg: a+b*c <=> x=id+id*id

$t_1 = b * c$

$t_2 = a + t_1$



(2) Boolean Expression.

$E \rightarrow E_1 \text{ OR } E_2$

$E \rightarrow E_1 \text{ AND } E_2$

$E \rightarrow \text{NOT } E$

$E \rightarrow (E)$

$E \rightarrow \text{id relop id}$

$E \rightarrow \text{True}$

$E \rightarrow \text{False}$

$E \rightarrow E \text{ OR } E$ { E.place = new Temp();
Emit (E.place = 'E1.place OR E2.place'); }

$E \rightarrow E_1 \text{ AND } E_2$ { E.place = new Temp();
Emit (E.place = 'E1.place AND E2.place'); }

$E \rightarrow \text{NOT } E$ { E.place = new Temp();
Emit (E.place = 'NOT E1.place'); }

$E \rightarrow (E_1)$ { E.place = E1.place }

$E \rightarrow \text{True}$ { E.place = new Temp();
Emit (E.place = '1'); }

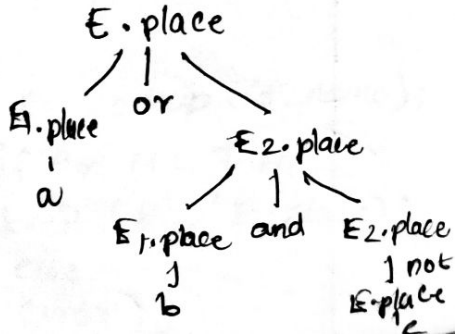
$E \rightarrow \text{False}$ { E.place = new Temp();
Emit (E.place = '0'); }

$E \rightarrow \text{id relop id}$ { id1 relop id2

{ E.place = new Temp();
Emit (if (id1.place relop.op id2.place
'goto' nextstate 3));
Emit (t.place = '0');
Emit ('goto' nextstate 2);
Emit (E.place = '1'); }

Eg:- a or b and not c

t1 = not c
t2 = b and t1
t3 = a or t2

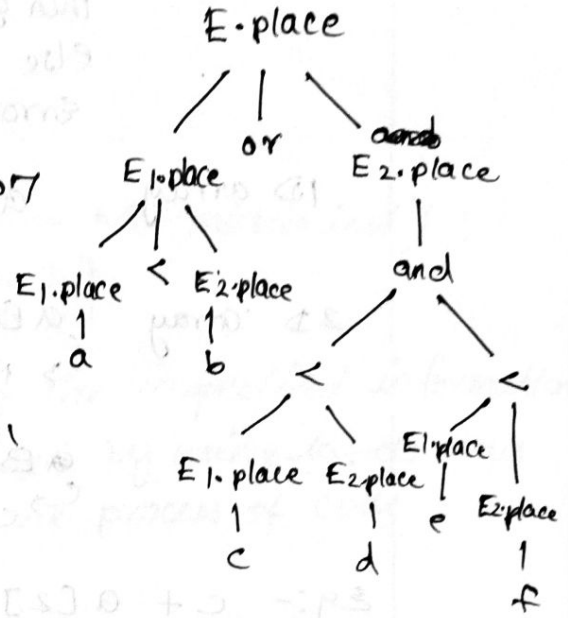


eg if a<b then 1 else 0.

```
100 : if a<b goto 103
101 : t1 = 0
102 : goto 104
103 : t1 = 1
104 :
```

eg a < b or c < d and e < f

```
100: if a<b goto 103
101 : t1=0
102 : goto 104
103 : t1=1
104 : if c<d goto 107
105 : t2=0
106 : goto 108
107 : t2=1
108 : if e<f goto 111
109 : t3=0
110 : goto 112
111 : t3=1
112 : t4=t2 and t3
113 : t5= t1 or t4
```



assignment (3) Array references

↑ Emit() gen()

$S \rightarrow id = E$

$S \rightarrow \underline{L} = E$

$L \rightarrow L[E]$

array → $L \rightarrow id[E]$

$E \rightarrow L$

$E \rightarrow E_1 + E_2$

$E \rightarrow id$

$S \rightarrow id = E$ { gen { p = lookup (id_entry)
if p ≠ nil
then gen (E.place)
else
Error ; }

$S \rightarrow L = E$ { gen { L.addr.base
'[' 'L.addr' ']' '=' E.place }

$L \rightarrow L[E]$ { L.array = L1.array
L.type = L1.type ;
t = newTemp ();
L.addr = newTemp();

gen (t = 't.addr' * L.type.width) ;
gen (L.addr = 'L1.addr' + 't' t) ; }

E → L { E.addr = new Temp ( ) ;
gen (E.addr = ' L.array.base '[' L.addr ']') ; }

E → t1 + t2 { t.place = new Temp ( ) ;
gen (E.place = ' E1.place + ' E2.place) ; }

E → id gen { p = lookup (id_entry)
if p ≠ nil
then gen (E.place =)
else
error ;

1D array $a[i] = B + w * (i - LB)$ (Eg:- 1000 + 4 * (1 - 0); base, width, initial value, lower bound)

2D array $a[i][j] = B + w * (N * (i - Lr)) + (j - Lc)$
↳ Row major

$a[B + w * ((i - Lr) + (M * (j - Lc)))]$
↳ Column major.

Eg:- c + a[2][3]
i j

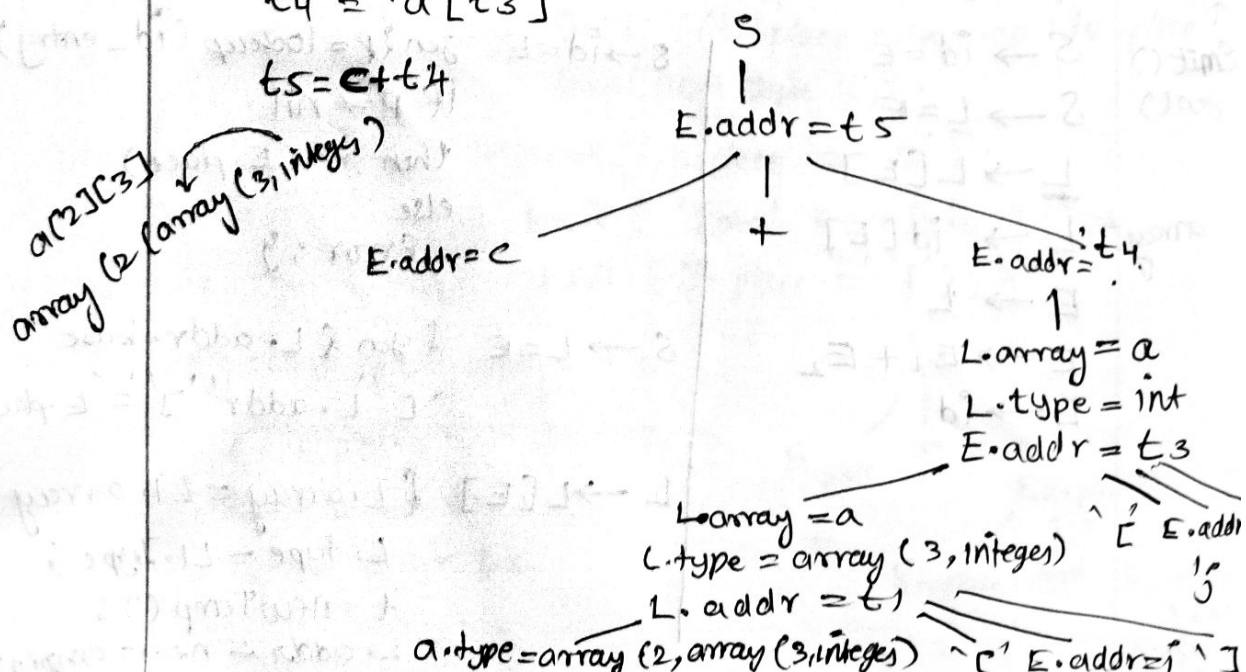
	C_0	C_1	C_2	
R_0:	10	11	12	=> 4+4+4 = 12
R_1:	13	14	15	

(11 ↓ 4) int => 4 bits.

3 addr Code

$t_1 = i * 12$
$t_2 = j * 4$ (4 because 1 included in row).
$t_3 = t_1 + t_2$
$t_4 = a[t_3]$
$t_5 = c + t_4$

a[2][3] array (2, array (3, integer))



BACKPATCHING

(X) Not

$x < 100 \;||\; y > 200 \;\&\&\; x != y$

TAC (1st pass)

Labels

100 : if $x < 100$ then 106
101 : goto 102
102 : $y > 200$ then 104
103 : goto 107
104 : if $x != y$ then 106
105 : goto 107
106 : true
107 : false.

II pass (Backpatch)

M (marker inst)

backpatch merge() truelist falselist

* The process of fullfilling the unspecified information by TAC (Three addr code) by using labels and semantic actions during the process of code generation.
* It is also defined as by the process of filling the graph in an incomplete transformations and information
* Backpatching is applied on ~~boolean expression, flow~~ boolean expression, flow of control labels
  1. boolean expression
  2. flow of control statement.
  3. Label and goto
* in the given example if $x < 100$ is true. Then the entire expression is True.
  It is called short circuit evaluation.

Eg:- $x < 100 \;||\; y > 200 \;\&\&\; x != y$

TAC (I st pass)

```
100 : if x<100 then ____
101 : goto ____
102 : if y > 200 then ____
103 : goto ____
104 : if x != y then ____
105 : goto ____
106 : true
107 : false
```

(II pass)

marker – address of next instruction

```
100 : if x <100 then 106
101 : goto 102
102 : if y>200 then 104
103 : goto 107
104 : if x != y then 106
105 : goto 107
106 : true
107 : false
```

fl – falselist
tl – truelist
marker → stores address of B2.

1) $B \rightarrow B_1 \;||\; MB_2$ { Backpatch (B1.fl, M.inst);
B.tl = merge (B1.tl, B2.tl);
B.fl = B2.fl; }

(M → marker)

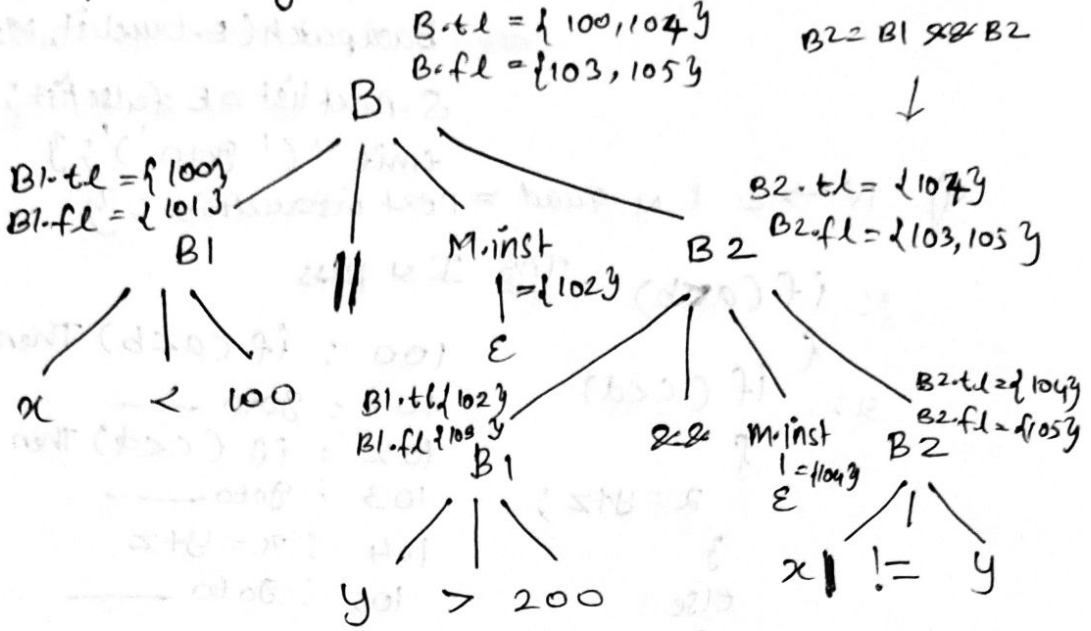
2) $B \rightarrow B_1 \;\&\&\; MB_2$ { Backpatch (B1.tl, M.inst);
B.tl = B2.tl;
B.fl = merge (B1.fl, B2.fl); }

3) $B \rightarrow !B_1$ { B.tl = B1.fl;
B.fl = B1.tl; }

4) $B \rightarrow (B_1)$ { B.tl = B1.tl;
B.fl = B1.fl; }

5) $M \rightarrow \varepsilon$ { M.inst = nextinst; }

1) Where M = non terminal marker and address of B2 is specified by M.



→ Flow of control structures.

S → if E Then S
| if E Then S else S
| while E do S
| begin L end
| A

L → L ; S
| S

1) S → A { S.nextlist = nil }
2) S → begin L end { S.nextlist = L.nextlist }
3) S → if E Then M S1 { backpatch (E.truelist, M.Quad)
S.nextlist = merge (E.falselist, S1.nextlist); }
4) L → L1 ; M S { backpath (L1.nextlist, M.Quad);
L.nextlist := S.nextlist ; }
5) L → S { L.nextlist = S.nextlist ; }
6) M → ε . { M.quad = next instruction ; }
7) S → if E Then . M1 S1 N else M2 S2
{ backpatch . (E.truelist, M1.Quad);
balkpatch . (E.falselist, M2.Quad);
S.nextlist = merge . (S1.nextlist, merge (
N.nextlist, S2.nextlist)); }

8) $S \rightarrow$ while $M_1 E$ do $M_2 S_1$

{ backpatch (S1.nextlist, M_1.quad);
backpatch(E.truelist, M_2.quad);
S.nextlist = E.falselist;
emit 'C' goto ')'; }

9) $N \rightarrow \varepsilon$ { N.quad = nextinstruction; }

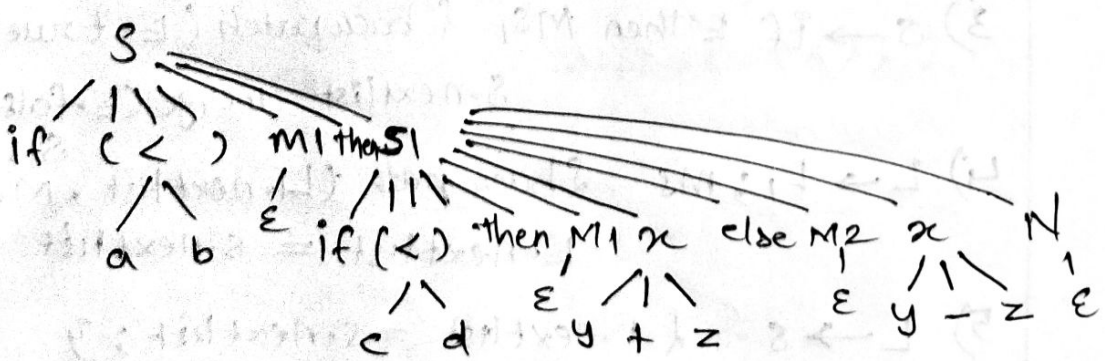
S: if (a<b)
{
S1: if (c<d)
{
x = y+z;
}
else
{
x = y-z;
}
}

TAC I st pass

100 : if (a<b) Then ——
101 : goto ——
102 : if (c<d) Then ——
103 : goto ——
104 : x = y+z
105 : goto ——
106 : x = y-z
107 : goto ——

II pass

100 : if (a<b) Then 102
101 : goto 108
102 : if (c<d) Then 104
103 : goto 106
104 : x = y+z
105 : goto 108
106 : x = y-z
107 : goto 108



→ Types and declaration.

* Type expressions (TE)
* Type equivalence (2 types)
* Type declaration.