Introduction

- Compiler is a program which converts High-level program to Low-level program

- There are multiple phases of Compilation:
  - Lexical Analysis
  - Syntax Analysis
  - Semantic Analysis

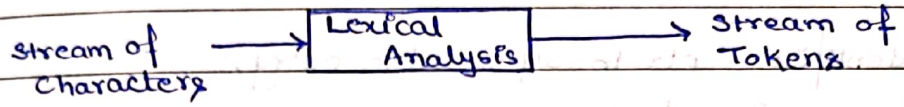
  → Analysis Phase

  - Intermidiatte Code Generation
  - IC Optimization
  - Code Generation
  - Code Optimization

  → Synthesis Phase

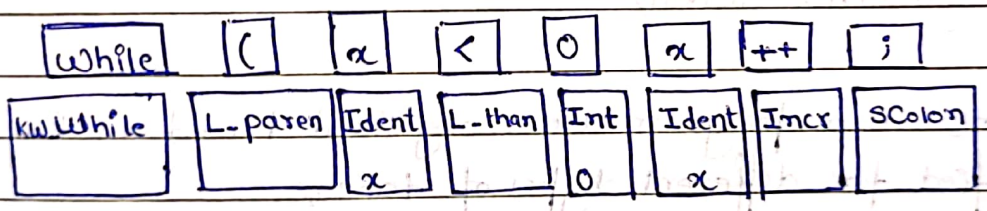
Lexical Analysis

- Token: A logical unit of a program.
- The source program is a sequence of characters.
- The job of Lexical Analysis phase is to convert the stream of characters to stream of tokens:



- A token mainly consists of Token Name, but it can additionally contain attributes. Ex: Name of identifier, value of numeral, etc.

Ex. `while ( x <0) x++;`



- Lexeme: The peice of original program from which the token is made.

Ex. In above program x is Lexeme, Identifier is Token.

- Tokens are considered as the terminal of Grammar (Useful in Syntax Analysis).

* Symbol Table

A data structured maintained by the Compiler which is used to store & query information about various identifiers (variables, functions etc), through different phases of compilation. In the lexical analysis phase a lexeme can be identified as identifier, and its

entry can be created in the symbol table, but its type can only be determined in later phases.

Ex. x += y + 3;

<identifier, ptr to sym. table entry>
<incr_assign_op, >
<identifier, ptr to symbol table entry>
<add_op>
<integer_const, 3>

* Recognizing Tokens

- Tokens in a program are recognized based on RE rules specified for different types of tokens.

Ex. KW_FOR for
Ident [A-Za-z_] [A-Za-z0-9_]*

So, based on these rules, matching is done.

- Longest Match (Maximal Munch): The Lexical Analyzer matches the longest matching lexeme with a rule from start.

Ex. [In C] a++;
0 1 2 3

- Starting from pos 0, the maximal matching lexeme is 'a' which is a identifier. Move to pos 1.
- Starting from pos 1, one match is '+' w/ add_op & other match is '++' which is incr_op. So, since "++" is

longer, that's matched. Move to pos 3

- Only ';' matches.

∴

a	+ +	;

- There can be other problem. A lexeme can match w/ more than one rule.

Ex. KW-DOUBLE double

Identifier [A-Za-z_][A-Za-z0-9_]*

double x;

↓

matches w/ both rules.

In such situations, match w/ the rule defined first (priority by position).

- In case Lexical Analyzer starting from some position, can't match the lexemes with any rule, then there is a lexical error.

Ex. x="apdb;

x is matched.

= is matched.

"apdb; isn't matched w/ any token type. This throws error.

Ex. int 3abc;

int matches KW-INT

3abc; doesn't matches any token type.

- Lexical Analyzer ignores comments.

* Token Counting For C programs.

- LAnalyzer / scanner removes / ignores comments before converting to tokens.

- Before LA runs, preprocessing happens, hence any substitution by #define, any inclusion (copy-paste) by #include, etc must be done before converting to tokens.

- String literals : Anything b/w starting " & ending " is counted as one token.

  Ex. `x = "abc";` ⇒ [x] [=] ["abc"] [;]

Ex.
```
#define OP +
#define M 100

int main() {
    x = M + OP;
}
```
⇒ [int] [main] [(] [)] [{] [x] [=] [100] [+] [;] [}]

∴ 11 tokens.

- Possible Lexical Errors:
  (1) Unterminated Comment
  (2) Unterminated String
  (3) Unidentified token
  etc.

Syntax Analysis

- After Lexical Analysis, we have a stream of tokens, so this is the input for Syntax Analyzer.

- Syntax Analysis phase tries to structure the stream of tokens into "Parse Tree", by following the Grammar rules of the language. While converting to Parse Tree, it also has to check if there's any token(s) which don't follow the Grammatical rules & don't properly convert to the structure & reports the Syntax Errors.

* Grammar (Context Free)

- A Grammar is used to define a language (set of strings) finitely. It is a four tuple: (V, T, P, S)
  - V: Set of Variables (Non-terminals) {Non empty, finite}
  - T: Language Alphabet (Terminals) {Nonempty, finite}
  - P: Set of production rules {finite}
  - S: Start symbol (variable). $S \in V$

- CFG is used to describe CFLs. Each production of a CFG is of the form:

$$V \longrightarrow (V+T)^*$$

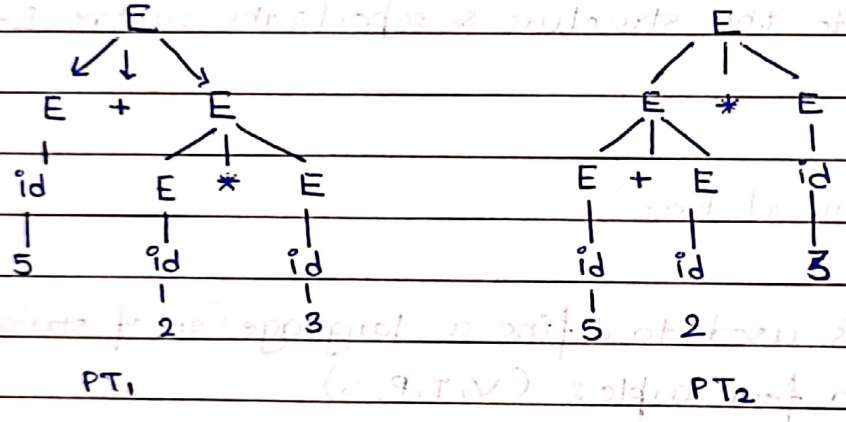
- Left Most Derivation: At any step of derivation, resolve the leftmost NTerm.
  Right Most Derivation: At any step of derivation, resolve the rightmost NTerm.

- For a string $w \in$ CFG $G \overset{\text{Lang}(G)}{\uparrow}$, the no. of LMDs = no. of RMDs = no. of Parse Trees.

* Ambiguous Grammars

- A CFG G is ambiguous iff there exists some string $w \in L(G)$, such that there ~~are~~ is more than one Parse Tree for it w.r.t G.

Ex. $E \rightarrow E+E \mid E*E \mid id$. $\quad w = 5+2*3$



So, PT_1 results in the op^n: $5+(2*3) = 5+6 = 11$

PT_2 results in the op^n: $(5+2)*3 = 7*3 = 21$.

- So, Ambiguous grammars can create serious problems, as the meaning of the program becomes unpredictible & may differ from what it was intended to.

- The major problems w/ Ambiguous grammars:
  (1) There is no algorithm to detect if a given arbitrary grammar is ambiguous or not.

  (2) There is no algorithm to convert a given ambiguous grammar to unambiguous grammar.

Ex. Try to analyze if the following grammar can be made unambiguous.

$E \to E+E \mid E*E \mid id.$

① For a given string id + id * id, we'd want to give higher priority to *.

② For id + id + id, we'd want to have left association for +. Similarly for *.

We can modify the grammar as follows:

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id.$

- So, since we want left associativity, so we have left recursion, for the + & * op^{n}s.
- Since we want higher priority for *, we keep it below the level of +.

- Recursion: For a non-terminal X in CFG G, X has a recursion iff $X \xrightarrow{*} \ldots X \ldots$, where "$\ldots X \ldots$" is some sentential form w/ X. $\{\alpha X \beta;\ \alpha, \beta \in (V+T)^*\}$

Leftmost Recursion: X has leftmost rec. iff $X \xrightarrow{*} X \ldots$ (or $X \xrightarrow{*} X\alpha$)

Rightmost Recursion: X has rightmost rec. iff $X \xrightarrow{*} \ldots X$ (or $X \xrightarrow{*} \alpha X$)

Ex. Analyze: $S \to$ if C then S ① | if C then S else S ② | other ③

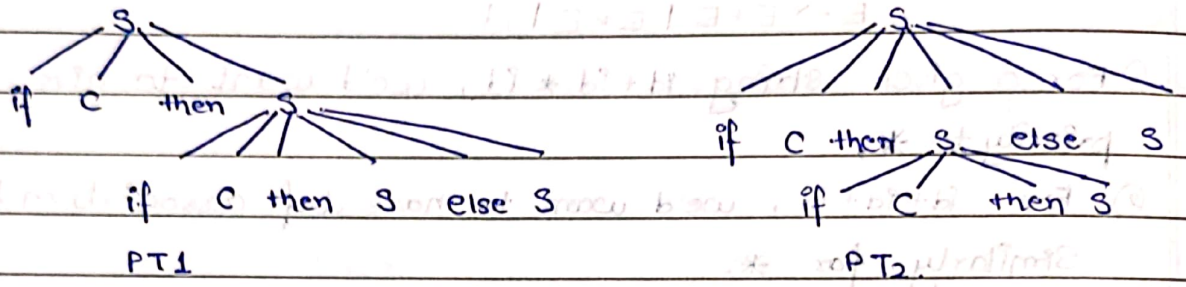
Say w = if C then if C then S else S

(a) $S \xrightarrow{①}$ if C then $S \xrightarrow{②}$ if C then if C then S else S

(b) $S \xrightarrow{①}$ if C then S else $S \xrightarrow{①}$ if C then if C then S else S.

So, this grammar is ambiguous.

Now the question is which of the two is desired.



In C language the rule is: an else of this form would associate with closest if preceeding it.
So, PT_1 is desired.

In Prod. ② of grammar, the problem is caused by having an option to derive Prod ①. If we eliminate it, we can possibly eliminate the ambiguity.

$S \rightarrow M \mid U$
$M \rightarrow$ if C then M else M | other
$U \rightarrow$ if C then S | if C then M else U

This is unambiguous grammar.

* Left Recursion & Left Factoring

Apart from Ambiguity, two major problems faced by CFGs are:
(1) Left Recursion
(2) Non-left-factored production rules.

- Non-left-factored productions: A grammar may have productions of form

$$A \to \alpha\beta_1 \mid \alpha\beta_2$$

These are non-left-factored, since they have a prefix α common.

This can be left factored as:

$$A \to \alpha B$$
$$B \to \beta_1 \mid \beta_2$$

Ex. $S \to aS \mid aaA \mid c$
$A \to b$

$\Rightarrow S \to aB \mid c$
$B \to S \mid aA$
$A \to b.$

Ex. $S \to aS \mid abA \mid abS$
$A \to a \mid aA$

$\Rightarrow S \to aB$
$B \to bS \mid bC$
$C \to A \mid S$
$A \to a \mid aA$

- The problem with having non-left-factored productions in Grammar would become apparent while discussing Parsers, but basically, while trying to substitute a variable A, with production rules of above form, confuses a parser.

- Left Recursion: A variable S has left recursion iff $S \xrightarrow{*} S\alpha \ \{\alpha \in (V+T)^*\}$.

- The idea to resolve left recursion is to convert it to right recursion, since it doesn't cause any problem for parsers.

- For a variable A in CFG, with left recursion of form:

$$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_m | \beta_1 | \beta_2 \dots | \beta_n$$

it can be converted to:

$$A \rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_n A'$$

$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_m A' | \epsilon$$

↑ don't forget to add this

Ex. $S \rightarrow Sa | Sb | b | c$

$\Rightarrow S \rightarrow bS' | cS'$

$S' \rightarrow aS' | bS' | \epsilon$

Ex. $S \rightarrow Sa | SA | Ab$

$A \rightarrow aA | b$

$\Rightarrow S \rightarrow AbS'$

$S' \rightarrow aS' | AS' | \epsilon$

$A \rightarrow aA | b$

Ex. $E \rightarrow E+T | T$

$T \rightarrow T*F | F$

$F \rightarrow (E) | id$

$\Rightarrow E \rightarrow TE'$

$E' \rightarrow +TE' | \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \epsilon$

$F \rightarrow (E) | id$

This method works for direct left recursion.

- The grammar may also have indirect left recursion

Ex. $S \rightarrow Aa | b$

$A \rightarrow Sa | a$

$S \rightarrow Aa \rightarrow Saa$

$A \rightarrow Sa \rightarrow Aaa$

Ex. $S \rightarrow Aa | b$

$A \rightarrow Ac | Sd | \epsilon$

To remove left recursion in case of (direct + indirect):

{Say, G: S → Aa | b
A → Ac | Sd | ϵ}

(1) Decide an order of non-terminals.
{Say, S, A.}

(2) In order, X1, X2, ... (above order), for each Xi, if any of X1, X2 .... Xi-1 appear in the productions of Xi, substitute them. Then eliminate any direct left recursion in Xi, if there.

{So, in order:

S: S has no possibility for substitution.
S has no direct left recursion.

A: A → Ac | Sd | ϵ
(Sd: substitution)
⇓
⇒ A → Ac | Aad | bd | ϵ
(Ac, Aad: direct left recursion)
⇓
⇒ A → bdA' | A'
A' → cA' | adA' | ϵ

∴ G': S → Aa | b
A → bdA' | A'
A' → cA' | adA' | ϵ
}

Ex. $A \rightarrow Cd$

$B \rightarrow Ce$

$C \rightarrow A \mid B \mid f$

S1: Order A, B, C.

S2:

A: $A \rightarrow Cd$

No subs. No direct left rec.

B: $B \rightarrow Ce$.

No subs. No direct left rec.

C: $C \rightarrow A \mid B \mid f$

subs.

$C \rightarrow Cd \mid Ce \mid f$

direct left rec.

$C \rightarrow fC'$

$C' \rightarrow dC' \mid eC' \mid \epsilon$

$\therefore$ G': $A \rightarrow Cd$

$B \rightarrow Ce$

$C \rightarrow fC'$

$C' \rightarrow dC' \mid eC' \mid \epsilon$

So, before using a grammar G for a parser check :

(1) Is the grammar ambiguous?

(2) Does any variable in G have left recursion?

(3) Are there any non-left-factored production?

Ex. $S \rightarrow aaA \mid aB \mid S\alpha \mid \beta$

$A \rightarrow a$

$B \rightarrow b.$

① Doesn't seem to be ambiguous.

② Left Factoring:

$S \rightarrow aX \mid S\alpha \mid \beta$

$X \rightarrow aA \mid B$

$A \rightarrow a$

$B \rightarrow b$

③ Removing left recursion:

$S \rightarrow aXS' \mid \beta S'$

$S' \rightarrow \alpha S' \mid \epsilon$

$X \rightarrow aA \mid B$

$A \rightarrow a$

$B \rightarrow b.$

* • Topdown Parsing: Beginning from the start symbol, try to estimate the productions to apply to end up at the string {source program in this case}.

• Bottom Up Parsing: Beginning w/ the string {source program in this case}, try to apply productions in reverse to convert the program back to start symbol.

• Parsing: The process of constructing the parse tree for a sentence generated by a given grammar.

• For any CFG, G, any string $w \in L(G)$ can be parsed using CYK algorithm in $O(n^3)$.

• Subsets of CFG however can be used & in conjunction w/ specific algo. to make parsing more efficient.

* Top Down Parsing

• There are two types of TopDown parsers:

  • Backtracking Used: For every variable substitution if multiple options, then choose an option & see if it leads to the string, else backtrack & take different string.

  • Non Backtracking (Predictive Parsing): For a variable substitution, choose a production, if it works, great, else you can't backtrack.

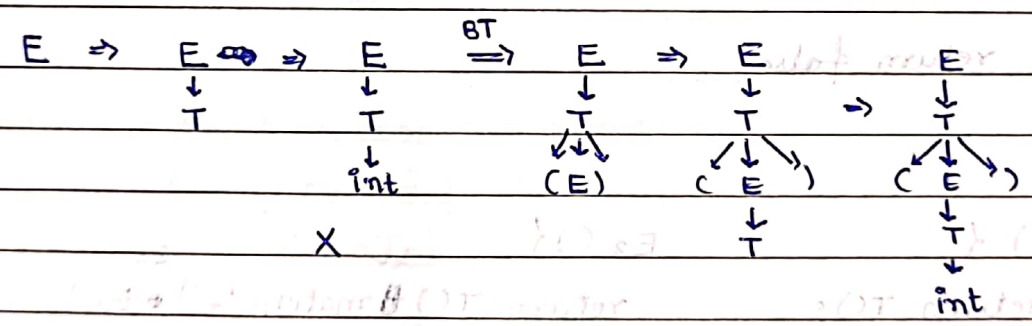
- Top Down Parsing
  - with Backtracking
    - Recursive way (Recursive Descent Parser)
    - Non recursive way
  - Without backtracking
    - Recursive way (RDP)
    - Non recursive way.

- Recursive Descent Parser : A parser which uses recursion & decents, i.e. goes top to bottom (TopDown).

- TDParsing w/ Backtracking (RDP):

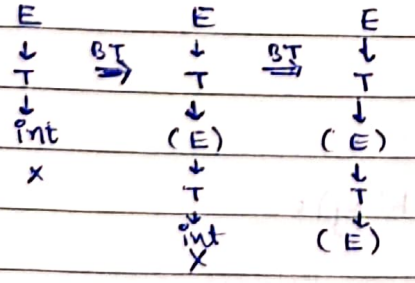
Ex. $E \to T \mid T+E$ $\quad w = (int)$

$T \to int \mid (E)$



✓ matches.

$w = (int+$



... no match.

- RDP (w/ Backtracking) :
  To implement it, for each non-terminal, write a procedure which explores all options.

Ex. E → T | T+E

T → id | (E)

```
match(t) {
    if (currSymbol == t) {
        move to next symbol;
        return true;
    }
    return false;
}
```

```
E1() {
    return T();
}
```

```
E2() {
    return T() && match('+') && E();
}
```

```
E() {
    return E1() || E2();
}
```

- Some pitfalls for RDP (w/ Backtracking):
  - Left Recursion. Causes ∞ loop.
  - Expensive due to backtracking.
  - Sometimes maynot even be able to parse a valid string w ∈ L(G). {Ex shown later}

- Predictive Parsing: The parser at every step tries to predict the production based on some criteria & never backtrack from it.
(The criteria generally is next "few" symbols, Lookahead.)
- It can also be done both in Recursive & Non-Recursive way.

- It is easy to check & match next few, say one, symbols when the productions start with terminal.

Ex. $E \rightarrow \underset{①}{+E} \mid \underset{②}{id}$ $\qquad w = ++id$ (↑ at first '+')

at cur pos., lookahead (i.e. symbol) = '+'
trying to match with prods, we get a match with prod ①. Use it.

- There can be two problems here:

① Non left factored productions.

Ex: $E \rightarrow +E \mid +T \mid id$
$T \rightarrow =$

This can be sloved by left factoring.

② Major problem is when the productions don't have non-terminals at start.

Ex. $E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$

Here E can't match terminal with lookahead directly.

- So, for Predict Parsing, we need two concepts:
(1) First Set
(2) Follow Set.

First Set:

- For a non-terminal A, the First Set, FIRST(A), is the set of terminals that can be at the start of strings produced by A.

$$\text{FIRST}(A) = \{ t \mid A \xrightarrow{*} w,\ w \in T^*,\ A = \overset{t}{\overset{w}{\blacksquare}}[0] \}$$

- Main Idea: For any production $A \to B_1 B_2 \ldots B_K$, keep processing $B_1, B_2, \ldots B_K$ until you encounter a NonTerm. that doesn't produce ϵ. Then go to next rule.
- $\alpha \in \text{FIRST}(S)$, iff $S \xrightarrow{*} \alpha$

Ex. $S \to ab \mid cd \mid de$

$\text{FIRST}(S) = \{a, c, d\}$

Ex. $S \to Ab \mid a$

$A \to aA \mid \epsilon$

$\text{FIRST}(A) = \{a, \epsilon\}$ $\quad \text{FIRST}(S) = \{a\} \cup \text{FIRST}(Ab)$

$= \{a, \epsilon\}$

but $S \not\xrightarrow{*} \epsilon$

So, $\text{FIRST}(S) = \{a, b\}$

Since $S \to \epsilon b \equiv S \to b$

Ex. $S \to a \mid bA$

$A \to AB \mid a \mid \epsilon$

$B \to c \mid d \mid e \mid f$

$\text{FIRST}(B) = \{c, d, e, f\}$ $\quad \text{FIRST}(A) = \{a, \epsilon, c, d, e, f\}$

$\text{FIRST}(S) = \{a, b\}$ $\quad A \xrightarrow{*} \epsilon$

- Formal def^n: $G(V, T, P, S)$
  - For $t \in T$:

    $FIRST(t) = \{t\}$
  - For a var. X, with prod, $X \to Y_1 Y_2 Y_3 \ldots Y_K$; $K \geq 1$:
    - $t \in FIRST(X)$, if $t \in FIRST(Y_i)$ AND $Y_1 Y_2 \ldots Y_{i-1} \xrightarrow{*} \in$
    - $\partial \in FIRST(X)$, if $Y_1 Y_2 \ldots Y_K \xrightarrow{*} \partial$.
  - If $X \to \partial$, then $\partial \in FIRST(X)$.

Ex. $S \to ab \mid ABSd \mid BC$

$A \to eS \mid Se \mid \partial$

$B \to dS \mid Sd \mid \partial$

$C \to AB \mid Sf \mid \partial$

X	FIRST(X)
S	a, e, d, ∂, f
A	∂, e, a, d, f
B	∂, d, e, a, f
C	∂, e, a, d, f

Ex. $E \to TX$

$T \to (E) \mid int\ Y$

$X \to +E \mid \partial$

$Y \to *T \mid \partial$

E	(, int
T	(, int
X	+, ∂
Y	*, ∂

- Follow Set: The follow set, for some non-terminal X, FOLLOW(X) is the set of all terminals that might come after X.

  $FOLLOW(X) = \{t \mid \$ \xrightarrow{*} \alpha X t \beta,\ t \in T;\ \alpha, \beta \in (V+T)^*\}$

  {S is start symbol of grammar}

Ex. $S \to Sa \mid Sb \mid \in$

FOLLOW(S) = {a, b, $}

- In Parsing, we always assume a string w is always followed by a special symbol on input tape called EndMarker Symbol ($). So, the start symbol S, FOLLOW(S) will always

contain $, since we assume $S \xrightarrow{*} w$, so S is followed by $ after S derives w.

- Formal def^n: Apply the following rules for all NonTerm. A, until no new symbol can be added to A their Follow sets:
  - $ ∈ FOLLOW(S), where S is start symbol.
  - If there is a production $X \to \alpha A B$, $\{X, A \in V; \alpha, B \in (V+T)^*\}$, then everything in FIRST(B) except ∂ is in FOLLOW(A).
  - If there is a prod. $(X \to \alpha A$ or $(X \to \alpha A B$ AND $\partial \in$ FIRST(B)) then, everything in FOLLOW(X) is in FOLLOW(A).

Ex. $E \to int$ | $E \to (E\ O\ E)$ | $O \to + | *$

	FIRST	FOLLOW
E	int, (	$, +, *,)
O	+, *	int, (

Ex. $S \to aS | SAb | BC$ | $A \to AS | a | \epsilon$ | $B \to BA | AB | \epsilon$ | $C \to c | \epsilon$

	FIRST	FOLLOW
S	a, b, e, ∂	$, a, b, e
A	∂, a, b, e	a, b, e, $
B	∂, a, b, e	e, $, a, b
C	∂, e	$, a, b, e

Ex. $E \to TX$ | $T \to (E) | int\ Y$ | $X \to +E | \partial$ | $Y \to *T | \partial$

	FIRST	FOLLOW
E	(, int	$,)
T	(, int	+, $,)
X	+, ∂	$,)
Y	*, ∂	+, $,)

Ex. $E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *FT' \mid \epsilon$
$F \to (E) \mid id$

	FIRST	FOLLOW
E	(, id	$,)
E'	+, ϵ	$,)
T	(, id	+, $,)
T'	*, ϵ	+, $,)
F	(, id	*, +, $,)

Ex. $S \to aBAS \mid \lambda$
$A \to ba \mid SB$
$B \to cA \mid S$

	FIRST	FOLLOW
S	a, λ	$, a, c, a,
A	b, a, c, λ	a, $, c
B	c, a, λ	a, $, c

- Predictive Parsing (cotd.)
  - TDParser which looks at next few symbols & in string & predicts which production to use.
  - It has linear parsing time, since no backtracking.
  - Predictive Parsers can parse language of LL(K) grammars:
    - L: left-to-right scan of inputs.
    - L: leftmost derivation
    - K: means predict based on next K tokens (lookahead)
  - If we are looking at only one next token, then the parser can parse LL(1) grammars.

- LL(1) Parser: Parsing Table
  - Construct a Parsing Table T for CFG G. For each production $A \to \alpha$ in G:
    - $\forall t \in FIRST(\alpha)$, do $T[A, t] = A \to \alpha$
    - If $\lambda \in FIRST(\alpha)$, then $\forall t \in FOLLOW(A)$, do $T[A, t] = A \to \alpha$.

Ex. $S \to aS$ FIRST(S) = {a, b, c}

$S \to bS$ FOLLOW(S) = {$}

$S \to c$

T	a	b	c	$
S	S→aS	S→bS	S→c	

LL(1) parsing table.

With help of this table we can perform parsing. Obtain the order of LMD. Check if string ∈ L(G) or not.

Say, w = abac.

We start with Start Symbol S. Two pointers, one on current input tape symbol & other on current derivation symbol.

Input Tape	Derivation	Action
↓abac $	↓S	Init State. Take T[S, a].
↓abac$	↓aS	Match. Move pointer.
a↓bac$	a↓S	Take T[S, b]
a↓bac $	a↓bS	Match. move pointer.
ab↓ac$	ab↓S	Take T[S, a]
ab↓ac$	ab↓aS	Match. move.
aba↓c$	aba↓S	Take T[S, c]
aba↓c$	aba↓c	Match. move.
abac↓$	abac↓	

So, the string is successfully matched.

(This is not the full algorithm, btw.)

Ex. $S \to SAa \mid \epsilon$
$A \to Sb \mid Ac \mid \epsilon$

	FIRST	FOLLOW
S	ϵ, c, b, a	$\$, a, b, c$
A	ϵ, c, b, a	$a, c,$

T	a	b	c	$
S	$S \to SAa$, $S \to \lambda$	$S \to SAa$, $S \to \lambda$	$S \to SAa$, $S \to \lambda$	$S \to \lambda$
A	$A \to Sb$, $A \to Ac$, $A \to \lambda$	$A \to Sb$, $A \to Ac$,	$A \to Sb$, $A \to Ac$, $A \to \lambda$	

- If the LL(1) Parsing Table T for a CFG G has a cell w/ more than one entry then G is not LL(1) grammar.

Ex. $E \to \overset{①}{T}X$
$T \to \overset{②}{(}E\overset{③}{)} \mid int\,Y$
$X \to \overset{④}{+}E \mid \overset{⑤}{\lambda}$
$Y \to \overset{⑥}{*}T \mid \overset{⑦}{\lambda}$

	FIRST	FOLLOW
E	(, int	$,)
T	(, int	+, $,)
X	+, λ	$,)
Y	*, λ	+, $,)

T	(	)	int	+	*	$
E	1		1			
T	2		3			
X		5		4		5
Y		7		7	6	7

- In case a CFG G has only one production per variable, it will always be LL(1). {Since for each row, any column will have atmost one entry, so conflict is imposible.}

- So, a CFG is LL(1) iff; for all prods; $A \to \alpha_1 | \alpha_2 | \dots | \alpha_k$
  - $\forall i, j$ $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi$.
  - AND If for some α_i; $\alpha_i \xrightarrow{*} \partial$.
    then, $\forall_{j \neq i}$, $FOLLOW(A) \cap FIRST(\alpha_j) = \phi$.

- So, a CFG G with a left recursive variable can never be LL(1). Since, the production will be of the form:

$$A \to A\alpha \mid \beta_1 \mid \beta_2 \dots$$

($A\alpha$: $FIRST(A\alpha)$; β_1: $FIRST(\beta)$)

So, $FIRST(\beta) \subseteq FIRST(A\alpha)$
(since $FIRST(\beta) \subseteq FIRST(A)$.

- Also, a non-left factored CFG can't be LL(1) either. Since there would be a variable A with productions of the form:

$$A \to \alpha\beta_1 \mid \alpha\beta_2$$

So, $T[A, x]$ would contain two or more entries.

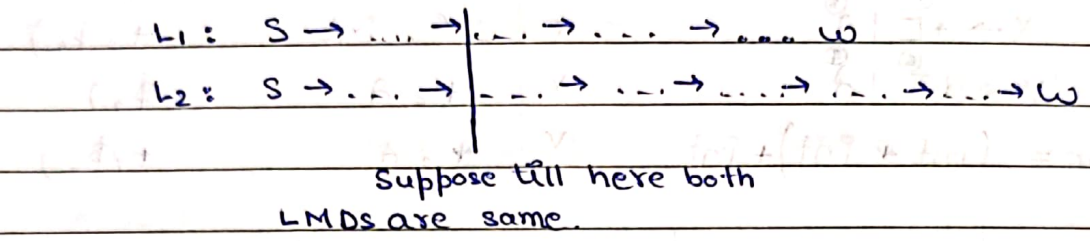
where $x \in FIRST(\alpha)$

{ in case $FIRST(\alpha) = \{\partial\}$, then conflict in $T[A,x]$ where $x \in FOLLOW(A)$ }.

Ex. $E \to T + E \mid T$ Is this LL(1) grammar?
$T \to (E) \mid id$

No, not left factored for E.
$FIRST(T+E) \cap FIRST(T) \neq \phi$.

- If a CFG is ambiguous, then its not LL(1).
  If G is ambiguous, then for some w there are two LMDs.



  At this point there is a confusion as to which path to predict.

- The Predictive parser can be implemented in both way: Recursive Descent & Non-recursive.

- LL(1) Parser : Non Recursive Algorithm
  - Input: String w. Parsing Table M.
    Initially on input buffer, we have w$.
    On stack we mark bottom w/ $ & have the start symbol S on top.
  - Output: If $w \in L(G)$, a leftmost derivation of w.
    Else, appropriate error.
  - Algorithm:

```
Let a be first symbol on input buffer;
let X be Top of Stack Value;
while (X ≠ $) {
    if (X = a) pop stack top & let a be next symbol on buffer;
    else if (X is terminal) error();
    else if (M[X,a] is error entry) error();
    else if (M[X,a] is X → Y1Y2...Yk) {
        output the prod. X → Y1Y2...Yk;
        pop stack;
        push Yk, Yk-1...Y2, Y1 on stack, Y1 on top;
    }
    let X be current ToS Value;
}
```

Ex. $E \xrightarrow{1} TX$

$T \xrightarrow{2} (E) \mid \xrightarrow{3} int\ Y$

$X \xrightarrow{4} +E \mid \xrightarrow{5} \lambda$

$Y \xrightarrow{6} *T \mid \xrightarrow{7} \lambda$

$\omega = (int * int) + int$

	FIRST	FOLLOW
E	(, int	$,)
T	(, int	+, $,)
X	+, λ	$,)
Y	*, λ	+, $,)

	int	+	*	(	)	$
E	1			1		
T	3			2		
X		4			5	5
Y		7	6		7	7

$\omega = (int * int) + int$

Buffer	Stack	Action
[(] [int] [*] [int] [)] [+] [int] [$]	E $	Init Config
(int * int) + int $	T X $	Take 1
(int * int) + int $	(E) X $	Take 2
(int * int) + int $	E) X $	Match. Pop. Incr on buffer
(int * int) + int $	T X) X $	Take 1
(int * int) + int $	int Y X) X $	Take 3
(int * int) + int $	Y X) X $	Match. Pop. Incr
(int * int) + int $	* T X) X $	Take 6.
...*...	T X) X $	Match. Pop. Incr.
...int...	int Y X) X $	Take 3.
...int...	Y X) X $	Match. Pop. Incr.
)+int $	X) X $	Take 7.
) + int $	) X $	Take 5.
) + int $	X $	Match. Pop. Incr
+ int $	+ E $	Take 4.
+ int $	E $	Match. Pop. incr
int $		

Ex. The grammar $S \to aSa|aa$ generates all even length strings of a's. A recursing-descent-parser (w/ backtracking) can be devised for the grammar.
Show that the RDP (w/ BT) recognizes a^2, a^4 & a^8 but not a^6. (Suppose that order followed by RDP is: $S \to aSa$, then $S \to aa$).

- Suppose the order followed is: ① $S \to aa$ ② $S \to aSa$. Let's try writing code for it.

```
main(){                    S(){                    S1(){
  if(S() && match('$'))      if(S1()) return true;    return(match('a')
    printf("Success");       if(S2()) return true;      && match('a'));
}                            return false;           }
                           }

S2(){
  return (match('a') && S() && match('a'));
}
```

Now, try $w = aa$.

main() → S() → S1()
↓ left match($) — true ← true ← true
Pf("success")

try $w = aaaa$.

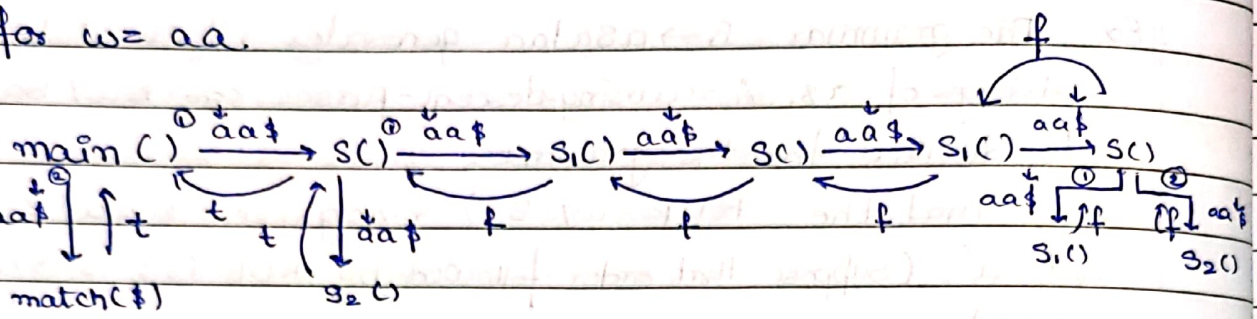
main() → S() → S1() {matches aaaa$}
false ↖ true
match('$')

So, for any a^{2^n}, $n \geq 1$, S would return true just after matching first two a's. The next 'a' won't match '$', so parsing unsuccessful even though $w \in L(G)$.

- Now suppose we have order: ① $S \to aSa$ ② $S \to aa$
$w = aa$.

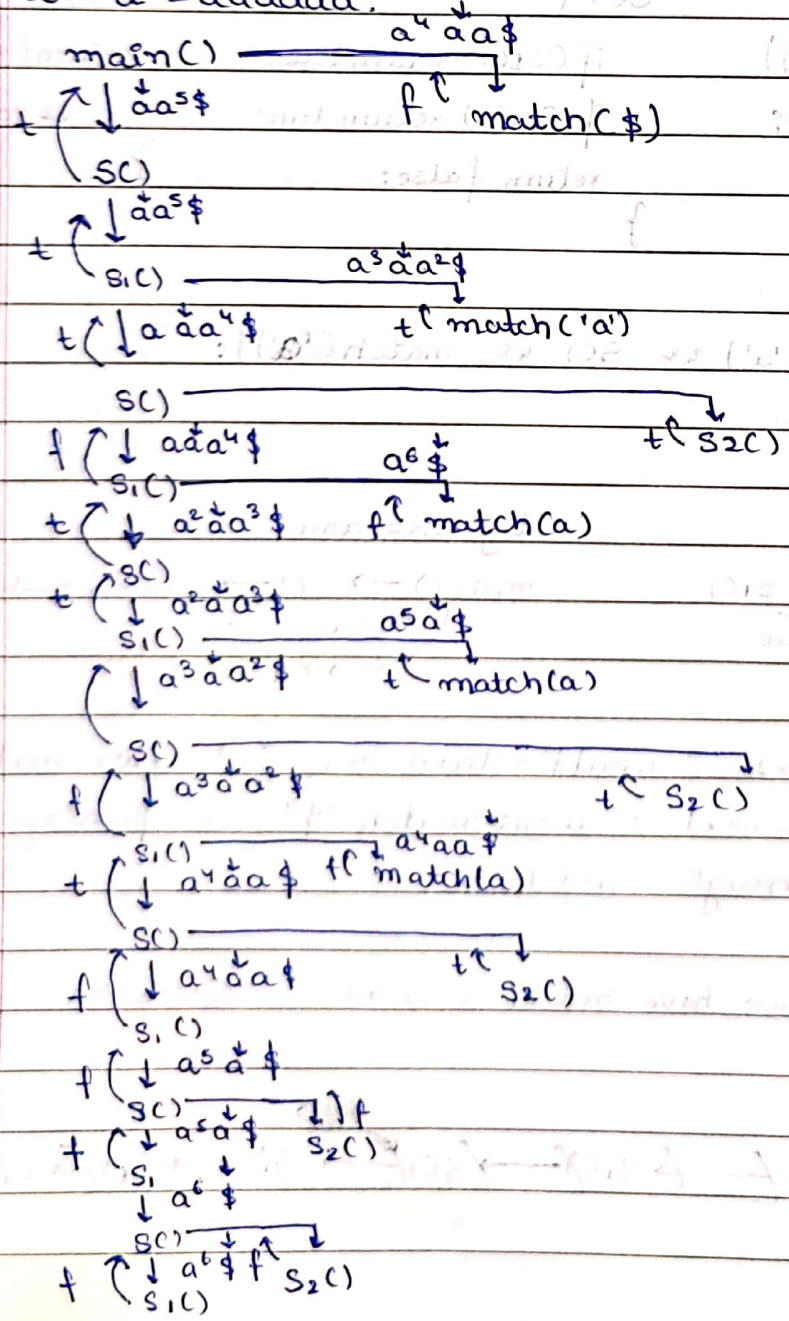
main() → S() → S1() → S() → S1() → S1() → S()

for w = aa.



So, aa is parsed successfully.

w = a^6 = aaaaaa.



∴ a^6 won't be parsed successfuly by the parser even though $a^6 \in L(G)$.

- So, RDP (w/ Backtracking) are very sensitive to the order of productions' implementation. It can happen that a string $w \in L(G)$ doesn't get successfully parsed.

- A CFG even if unambiguous & left factored & non-left-recursive can still be not LL(1).

Ex. $S \rightarrow Aa \mid B$ FIRST$(Aa) \cap$ FIRST$(B) \neq \emptyset$

$A \rightarrow aB$ $= \{a\gamma$

$B \rightarrow a \mid b.$

* Bottom Up Parsing

- Sentential Form: Any α, such that $S \xrightarrow{*} \alpha$, where S is start symbol & $\alpha \in (V+T)^*$ is called sentential Form.

- Sentence: A sentential form α, such that $\alpha \in T^*$

- Left Sentential Form: A sentential form α that occurs in the leftmost derivation of some string.

- Right Sentential Form: A sentential form α that occurs in the rightmost derivation of some string.

- Handle: For a sentential form $\alpha\beta\gamma$, such that $S \xrightarrow{*} \alpha A\gamma \rightarrow \alpha\beta\gamma$, handle is two tuple $\langle A \rightarrow \beta$, at pos. following $\alpha \rangle$, i.e. handle consists of the production & the pos. at which the production rule applies in the sentential form.

Ex. $S \rightarrow aAcBe$

$A \rightarrow Ab \mid b$

$B \rightarrow d$

List the handles in the rightmost derivation of $w = abbcde$

$S \rightarrow a\underline{A}cBe \rightarrow a\underline{Abc}Be \rightarrow abbc\underline{B}e \rightarrow abbcde$.

So, handle for:

abbcde : $\langle B \rightarrow d,$ pos 4$\rangle$ { we can also denote B→d by just the RHS: "d at pos 4" }

abbcBe : $\langle A \rightarrow b,$ pos 1$\rangle$

aAbcBe : $\langle A \rightarrow Ab,$ pos 1$\rangle$

aAcBe : $\langle S \rightarrow aAcBe,$ pos 0$\rangle$

Ex. $E \to E+T \mid T$

$T \to T*F \mid F$ $\quad\quad$ $w = id*id$. $\quad\quad$ RMD.

$F \to id$. $\quad\quad$ Find handle of: (a) id*id (b) T*id.

$\underline{E} \to \underline{T} \to T*\underline{F} \to \underline{T}*id \to \underline{F}*id \to id*id$.

Handle for:

id*id : $\langle F \to id, \text{pos } 0 \rangle$

T*id : $\langle F \to id, \text{pos } 2 \rangle$

Ex. $S \to 0S1 \mid 01$. Indicate the handle in each of these right sentential forms.

a) 000111 $\quad\quad$ (b) 00S11

Since right sentential form, RMD.

$S \to 0\underline{S}1 \to 00\underline{S}11 \to 000111$

Handle for:

000111 : $\langle S \to 01, 2 \rangle$ or 01

00S11 : $\langle S \to 0S1, 1 \rangle$ or 0S1

- "Informally, handle is substring of some sentential form such that it is on RHS of some production".

X This is misconception.

Ex. $S \to AB$; $A \to aa$; $B \to a$ $\quad$ & $w = aaa$.

$S \to A\underline{B} \to \underline{A}a \to aaa$.

So, in "aaa", 'a' at pos 1 & 2 are not handles even though they are substring of sentential form & appear on RHS of $B \to a$.

"aa" at pos 0 is handle.

Ex. Which of the following statements are true:

(A) Handle is any substring appearing on RHS of the prod.

(B) If grammar is unambiguous, then we have only one handle in any right sentential form.

(C) For any string $w \in L(G)$, if we keep on finding handles then we will reach to start symbol.

(D) If $\alpha\beta\gamma$ is right sentential form & β is handle then γ is string of terminals.

A is false. Handle must be substring of some sentential form & also represent a step in some derivation, ie.

$$S \xrightarrow{*} \alpha A \gamma \rightarrow \alpha \beta \gamma$$

(↑ step, ↑ substr. of sentential form.)

B is true.

If G is unambiguous, for any string we have only one RMD, so every step in derivation is unique for RMD.
{ we can show by contradiction }

C is true. By def^n $S \xrightarrow{*} \alpha A \gamma \rightarrow \alpha \beta \gamma$

D is true. If $\alpha\beta\gamma$ is right sentential form, then

$$S \xrightarrow{*} \alpha A \gamma \rightarrow \alpha \beta \gamma$$

If γ had some non-terminal (in $\alpha A \gamma$), then we'd have first taken that production, otherwise it violates RMD.

- **Viable Prefix :** For a sentential form $\alpha\beta\gamma$, where β is handle of the sentential form, Viable prefix is any prefix of $\alpha\beta$ substring.
So, to a viable prefix, it is always possible to add some appropriate sequence of terminals & non-terminals to obtain sentential form.

Ex. $E \to E+T \mid T$ Find viable prefix of ↑ E+(id)
$T \to (E) \mid id$

Right sentential form

$E \to E+T \to E+(E) \to E+(T) \to E+(id)$.

So, handle of E+(id) is $\langle T \to id, 3 \rangle$. So, any prefix of 'E+(id' is viable prefix.
{ E, E+, E+(, E+(id, ϵ }

Ex. $E \to E+E \mid E*E \mid id$. Find all handles in derivation of $\omega = id+id*id$. Also find viable prefixes of id+id*id. (RMD)

$\underline{E} \to E*\underline{E} \to \underline{E}*id \to E+\underline{E}*id \to \underline{E}+id*id \to id+id*id$.
Handles: {E+E} {id} {E+E} {id} {id}

So, viable prefixes of id+id*id ≡ pref. of id = { id, ϵ }.

Ex. Find all viable prefixes of the grammar: $S \to 0S1 \mid 01$.
Earlier we had a string for the RMD of which we could find viable prefixes, but to find all the viable prefixes of a grammar, we first need the concept of LR(K) grammars.

Ex. Suppose we have a string $w = ab$, which belongs to grammar:

$S \to AB$

$A \to a$

$B \to a$.

If we try to go from string w to start symbol while reading it from left to right we get the following steps:

S1: $\underline{a}b \longrightarrow Ab$ { Since a can be reduced to A }

S2: $A\underline{b} \longrightarrow AB$ { A can't, Ab can't. b can be reduced }

S3: $\underline{AB} \longrightarrow S$.

If we just reverse the steps, we essentially get RMD for w.

$\underline{S} \longrightarrow \underline{AB} \longrightarrow \underline{A}b \to ab$.

- So, this is what BottomUp parsers do. They perform reverse of rightmost derivation to go from string to start symbol. They don't do it blindly & apply some strategies. If they are able to parse the string back to start symbol, $w \in L(G)$, else not.

- If you know the handle at every step of "Reverse of RMD" we can simply replace w/ LHS of handle at pos. of handle, & get the previous sentential form in RMD of w. Eventually we'd reach S.

- Idea of Bottom Up parsing: $S \to \gamma_0 \to \gamma_1 \to \dots \to \gamma_n$ $\{w = \gamma_n\}$

  for $i = n$ down to 0:

  find the handle $\langle A_i \to \beta_i, pos \rangle$ in γ_i

  replace β_i in γ_i at 'pos' to get γ_{i-1}

So the crucial step is to find the handle at each step.

Ex. $E \to E+E \mid E*E \mid id$. $w = id+id*id$. Parse in LtoR BottomUp way.

Suppose $\alpha \mid \beta$ means, have read α, & haven't read β.

S1: | id+id*id.

S2: id | +id*id. [Shift]

We have two options:

(1) Try to reduce from what we've read.

(2) Shift | to right.

{Say we reduce whenever we can.}

S3: E | +id*id [Reduce]

We can't reduce.

S4: E+ | id*id [Shift]

S5: E+id | *id [Shift]

We can reduce 'id'.

S6: E+E | *id. [Reduce]

S7: E | *id [Reduce]

S8: E* | id [Shift]

S9: E*id | [Shift]

S10: E*E | [Reduce]

S11: E | [Reduce]

If we have a choice to reduce, then we have handle in α portion of $\alpha \mid \beta$.

Also, if I have handle in α portion of $\alpha \mid \beta$, I'll always reduce.

So, the question, "Can I reduce?" is ~~same as~~ "Is there a handle in α portion of $\alpha \mid \beta$?". dependent on.

But this doesn't mean, if we reduce some β in $\alpha\beta\gamma$, then β is handle. We may reduce wrongly.

∴ If handle in α of $\alpha \mid \beta$ then reduce.

handle in α of $\alpha \mid \beta \longrightarrow$ reduce.

- Data Structure for Bottom Up parsing : We'll be putting the α portion of $\alpha | \beta$ on stack.

- Say, we have $w = \alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5$ & we are at :

$$\alpha_1 \; \alpha_2 \; \alpha_3 \; | \; \alpha_4 \; \alpha_5$$

So, what is the question at this state?

The possibe questions are : Is

(a) α_3 OR (b) $\alpha_2 \alpha_3$ or (c) $\alpha_1 \alpha_2 \alpha_3$

handle ?

We can't ask "is α_2 a handle?" at this state, because we'd already have asked that question on state :

$$\alpha_1 \; \alpha_2 \; | \; \alpha_3 \; \alpha_4 \; \alpha_5.$$

Since we found out that the answer was no, that's why we shifted the boundary.

- So at any ~~state~~ state $\alpha | \beta$ only the substrings ending at TOS can be option for handles.

$\alpha_1 \; \alpha_2 \; \alpha_3 \; | \; \alpha_4 \; \alpha_5$

TOS

α_3 ← TOS
α_2
α_1

Ex. $E \rightarrow E+E \mid E*E \mid id$ $\quad w = id * id$

Stack	Action	Inp. Buffer	Remark / Question
$		id * id $	Init Config.
$ id	S	id * id $	Q: Is 'id' handle?
$ E	R	id * id $	'id' is handle. Q: Is E handle?
$ E*	S	id * id $	Q: Is '*' handle? OR Is 'E*' handle?
$ E*id	S	id * id $	Q: Is 'id' OR '*id' OR 'E*id' handle?
$ E*E	R	id * id $	'id' is handle.
$ E	R	id * id $	Q: Is 'E' OR '*E' OR 'E*E' handle?
			'E*E' is handle.

So, we reached start symbol. Found RMD. Verified that $w \in L(G)$.

- Note that this strategy is dependent on correct detection of handle. This blind strategy might go wrong.

Ex. $S \to ABa$

$A \to a$ $\quad w = aba.$

$B \to b.$

S1: $|aba$

S2: $a|ba \overset{R}{\Rightarrow} A|ba.$

S3: $Ab|a \overset{R}{\Rightarrow} AAB|a.$

S4: $ABa|$

at this stage if we first ask, is 'a' handle {before 'Ba' & 'ABa'} we get yes. If we blindly reduce:

$\overset{R}{\Rightarrow} ABA|$

Now we are stuck.

So, blindly reducing doesn't guarentee we are reducing handle. Being able to reduce doesn't mean we are reducing a handle.

- So, the most important question is: "How to find handle", since if we find handle on top of stack {in α part of $\alpha|\beta$}, then we can reduce."

{Since in sentential form, if we keep reducing handles, we'll eventually get to start symbol.}

- Strategies to find handles:

. Strategy 1: Whenever possible to reduce, just reduce.

Ex. $S \to aA$; $A \to a$ $\quad w = aa.$

$|aa\$ \overset{S}{\Rightarrow} a|a\$ \overset{R}{\Rightarrow} A|a\$ \overset{S}{\Rightarrow} Aa|\$ \overset{S}{\Rightarrow} Aa\$|$

So this strategy fails easily.

This assumes that if reduction is possible (by say $A \to \alpha$) then it is handle.

Strategy 1 is called LR(0). Since we are Left to Right reading string, while performing Reverse RMD & looking at 0 symbols after | in $\alpha|\beta$. $\therefore$ LR(0)

0 lookahead.

• Strategy 2: Reduce $\alpha\beta a w \$$ to $\alpha A a w \$$ only if $a \in \text{FOLLOW}(A)$

Ex. $S \to aA$; $A \to a$ $\quad w = aa$

$\text{FOLLOW}(S) = \{\$\}$

$\text{FOLLOW}(A) = \{\$\}$

$|aa\$ \xrightarrow{S} a|a\$ \xRightarrow{S} aa|\$ \xRightarrow{R} aA|\$ \Rightarrow S|\$ \Rightarrow S\$|$

don't red. with $A \to a$, ∵ $a \notin \text{FOLLOW}(A)$

∴ Successfully parsed.

This makes sense. If the next symbol after A after reduction is not in FOLLOW(A), i.e. $\alpha A a \gamma$ & $a \notin \text{FOLLOW}(A)$ that means this is not a sentential form. So why take this derivation. And since $\alpha A a \gamma$ is not sentential form, so by defn of handle β is not the handle of $\alpha\beta a\gamma$.

This Strategy is called: SLR(1) parsing.
S stands for Simple.
Simple LR(1).

• Strategy 3: Reduce $\alpha\beta a\gamma$ to $\alpha A a \gamma$ only if $a \in \text{FOLLOW}(\alpha A)$.

This strategy is called CLR(1) or just LR(1).
C stands for cannonical (i.e. standard).

Ex. S → Ba
B → aAa | aab | bAb
A → Ba

w = aaba

	FIRST	FOLLOW
S	a, b	$
B	a, b	a
A	a	a, b

• Let's apply SLR(1) strategy.

aaba$ $\xrightarrow{S}$ aaba$ $\xrightarrow{S}$ baba$ $\xrightarrow{S}$ bbaba$ $\xrightarrow{R}$ Baba$

Baba$ $\xrightarrow{S}$ aaba$ $\xrightarrow{R}$ (1) Aaba$ $\xrightarrow{S}$ Aaaba$ $\xrightarrow{R}$ (2) AAba$

AAba$ $\xleftarrow{S}$ AAbaa$ $\xleftarrow{S}$ AAbaa$ (S ↓) AAbaa$

So, parsing fails.

① since a ∈ FOLLOW(A) ② since b ∈ FOLLOW(A)

• Applying LR(1)

Suppose state is aa|ba$ $\xrightarrow{S}$ aab|a$ $\xrightarrow{R}$ B|a$ (S ↓)

S$| $\xleftarrow{S}$ S|$ $\xleftarrow{R}$ Ba|$

So, parsing successful.

• There is another strategy called LALR(1), which we'll look at later.

• LR(0) & SLR(1) use something called "LR(0) items".

• LR(1) & LALR(1) can also use LR(0) items, but they use LR(1) items to make their job easier.

- LR(0) Item : For any production $X \to \alpha B$, we have following LR(0) items :
  (1) $X \to .\alpha B$ { same as $|\alpha B$ }
  (2) $X \to \alpha . B$ { same as $\alpha | B$ }
  (3) $X \to \alpha B .$ { same as $\alpha B |$ }

- An item $X \to \alpha . \beta$ means that :
  (1) Have α on TOS { i.e. have already seen α }
  (2) Expecting to find string derived from β on the input buffer.
  (3) The parser is looking to reduce to X, if $\alpha\beta$ is found.

- CLOSURE (I) { I is LR(0) item } : For a LR(0) item I, say $X \to \alpha . A\beta$, where A is non-terminal, its closure contains itself, all prods. of A with the dot/border at start of LHS (i.e. $A \to .\gamma$) and recursively if any prod has non-terminal B after dot/border then all prods of B with dot/border at start of RHS.

Ex. $S \to AB$
$A \to a \mid Aa \mid Ba.$
$B \to a.$

Closure $(S \to .AB) = \{ S \to .AB, A \to .a, A \to .Aa, A \to .Ba, B \to .a \}$
Closure $(S \to A.B) = \{ S \to A.B, B \to .a \}$

Including $B \to .a$ in closure of $S \to A.B$, means that we've already seen A, & we expect to see string derived from B on input buffer & the options to get that include $B \to .a$.

Similarly, for LR(0) item S→.AB, next symbol is non terminal A, so we can get A through A→.a or A→.Aa or A→.Ba.

Building LR(0) automaton:

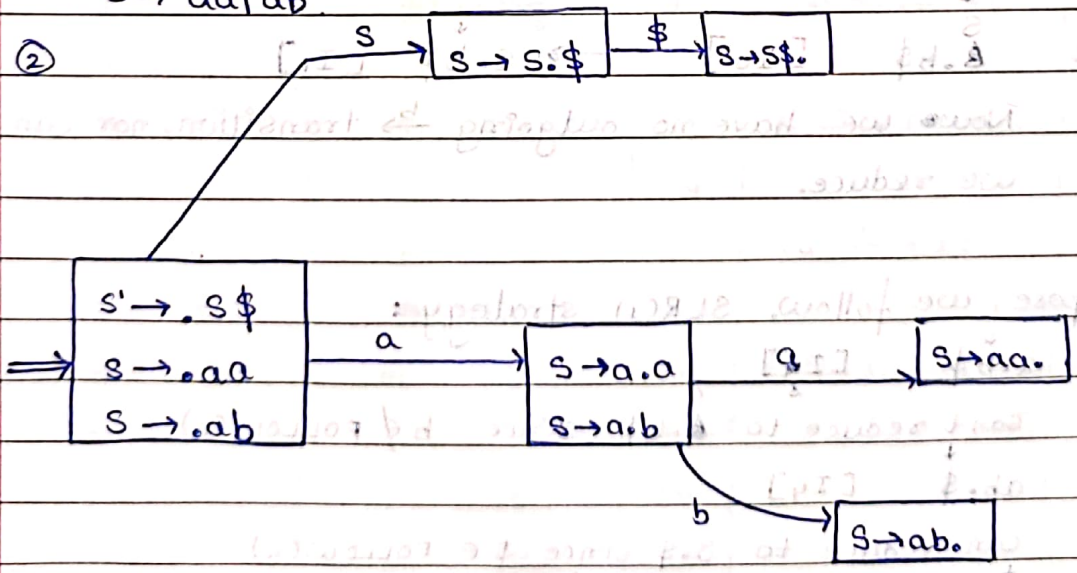
Before building LR(0) automaton for a Grammar G w/ start symbol S, we have to augment it, with the following production: S'→.S$, meaning that we've seen nothing & expect to see string derived from 'S$'. If we successfuly see a string derived from 'S$' we can reduce to S' & accept the string.

. The initial state for automaton is the closure of 'S'→.S$' & every state is closure of a LR(0) item.

Ex. S→aa|ab

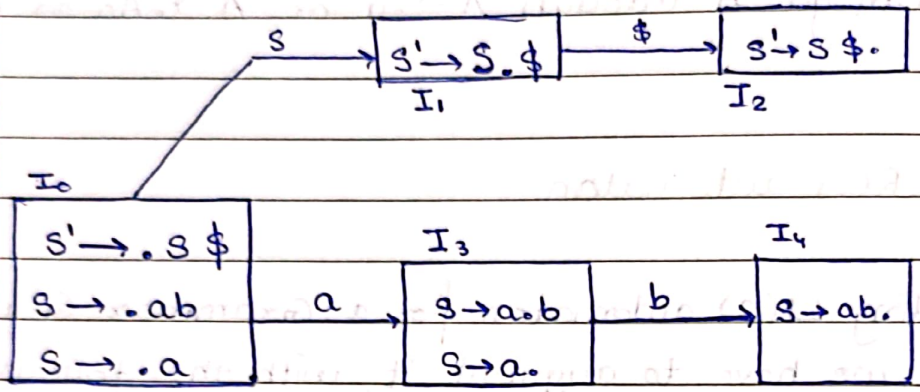
① S'→.S$
S→aa|ab

②



This automaton captures all the possibilities. Based on Parsing strategy either reduction is taken or not. Each state represents a closure: what we've seen (α), what we expect to see (β), if we see (S), what we'll reduce to, & what are the possibilities to see it. {closure (S→α.β)}

Ex. $S \rightarrow ab \mid a$



Say $w = ab$.

S_1: $\overset{\downarrow}{.}a\,b\,\$$ $[I_0]$

$\overset{S}{\downarrow}$

S_2: $a.\overset{\downarrow}{b}\$$ $[I_3]$

Now, as per DFA, we've two choices.

Shift ⇒ $ab.\$$

Reduce ⇒ $S.b\$$

Say we are following LR(0) strategy, i.e. always reduce when possible.

$\overset{R}{\downarrow}$

S_3: $\overset{S}{\downarrow}.b\$$ $[I_0]$ ⟶ $S.\overset{\downarrow}{b}\$$ $[I_1]$

Now we have no outgoing $\xrightarrow{b}$ transition, nor can we reduce.

Suppose, we follow, SLR(1) strategy:

S_2: $a.\overset{\downarrow}{b}\$$ $[I_3]$

$\overset{S}{\downarrow}$ Can't reduce to $\overset{S}{\downarrow}.b\$$ since $b \notin$ FOLLOW(S)

S_3: $ab.\overset{\downarrow}{\$}$ $[I_4]$

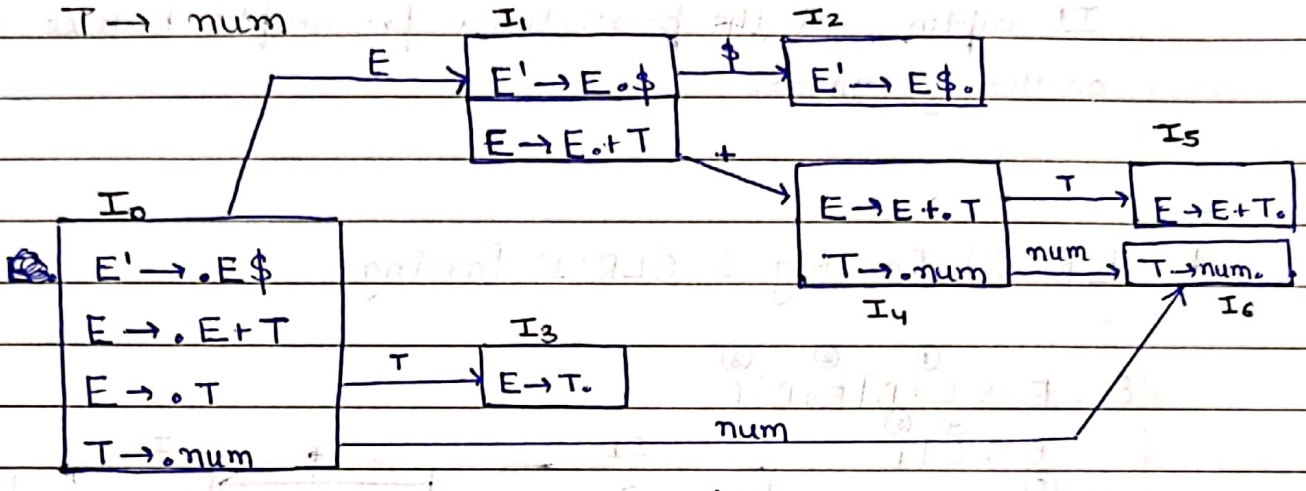
$\overset{R}{\downarrow}$ can reduce to $S.\$$ since $\$ \in$ FOLLOW(S).

S_4: $\overset{\downarrow}{S}.\$$ $[I_0]$ ⟶ $S.\overset{\downarrow}{\$}$ $[I_1]$ ⟶ $S\$.$ $[I_2]$

So, we can accept the string.

Ex. $E \to E+T \mid T$

$T \to num$



$w = num+num+num$ or $n+n+n$.

Suppose using LR(0) strategy to always reduce.

	Stack (Seen)	Input Buffer (unseen)	State	Comment	Action
1.	$	n+n+n$	I_0		Shift
2.	$n	+n+n$	I_6		Reduce.
3.	$T	+n+n$	I_0	DFA takes $I_0 \xrightarrow{T} I_3$	
	$T		I_3		Reduce.
4.	$E	+n+n$	I_0	DFA: $I_0 \xrightarrow{E} I_1$	
	$E		I_1		Shift
5.	$E+	n+n$	I_1	$I_1 \xrightarrow{+} I_4$	
	$E+		I_4		Shift
6.	$E+n		I_4	$I_4 \xrightarrow{n} I_6$	
	$E+n	+n$	I_6		Reduce.
7.	$E+T	+n$	I_0	$I_0 \xrightarrow{E} I_1 \xrightarrow{+} I_4 \xrightarrow{T} I_5$	
	$E+T		I_1		
	$E+T		I_4		
	$E+T		I_5		Reduce.
8.	$E	+n$	I_0		
	$E		I_1		Shift
9.	$E+	n$	I_1		
	$E+		I_4		Shift
10	$E+n	$	I_4		

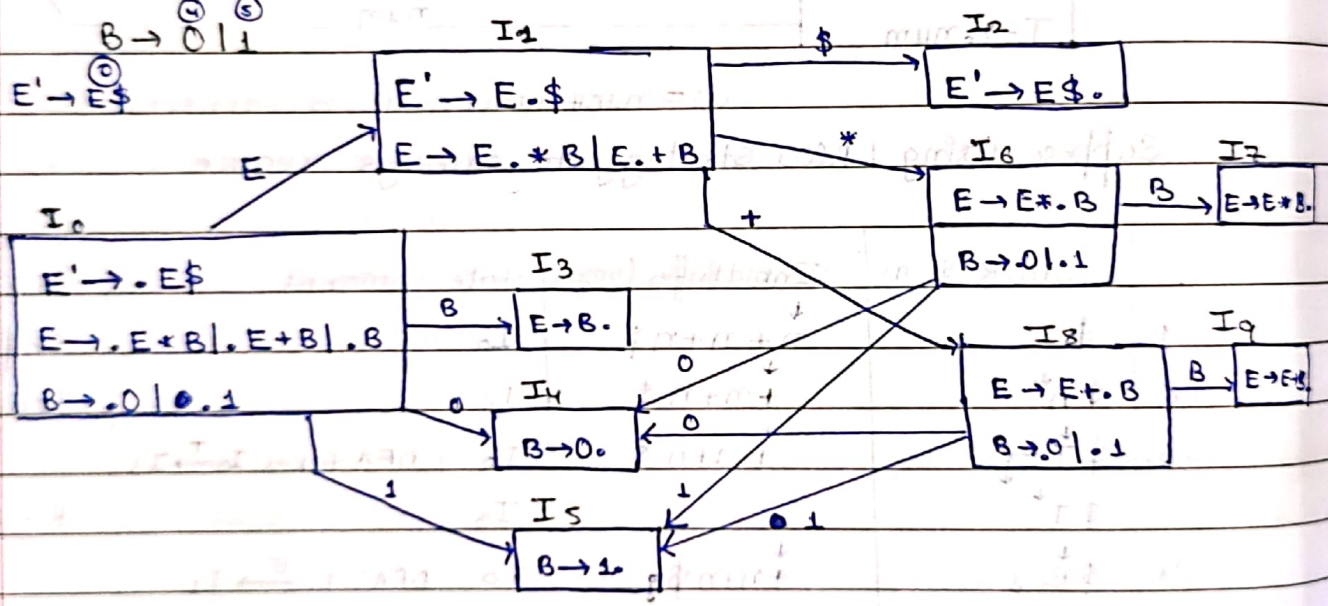
- So the Automaton is independent of the Parsing Method. It captures all the possibilities for shift & Reduce actions on the grammar.

* LR(0) Parsing & SLR(1) Parsing

Ex. $E \to \overset{①}{E*B} \mid \overset{②}{E+B} \mid \overset{③}{B}$

$B \to \overset{④}{0} \mid \overset{⑤}{1}$

$E' \to \overset{⑥}{E\$}$



We can convert this automaton to Tabular form:

States	Action +	*	0	1	$	Goto E	B
0			S_4	S_5		S_1	S_3
1	S_8	S_6			(Accept) S_2		
2	R_0	R_0	R_0	R_0	R_0		
3	R_3	R_3	R_3	R_3	R_3		
4	R_4	R_4	R_4	R_4	R_4		
5	R_5	R_5	R_5	R_5	R_5		
6			S_4	S_5			7
7	R_1	R_1	R_1	R_1	R_1		
8			S_4	S_5			9
9	R_2	R_2	R_2	R_2	R_2		

- Si entry : For action part, Si means Shift & Goto state i.
  Ri entry: For action part,
  T[Ii, $]: The entry T[Ii, $], where Ii is the current state & $ is the current symbol on top of stack, means what action to take.

- The only difference b/w LR(0) & SLR(1) parsers is that LR(0), whenever it can reduce will reduce, while SLR(1) whenever it can reduce, first checks the follow of the reduced non-terminal.

So, in the Parsing Table, the only actions that would differentiate b/w LR(0) & SLR(1) are the Ri {reduction entries}.
T[Ii, t] = Reduce $A \to \alpha\beta$ means, when you are at state Ii & the next symbol is t, then reduce $\alpha\beta$ to A in $\alpha\beta t\gamma$.
Since SLR(1) always ensures that 't' is in FOLLOW(A) before reducing, only those entries in Table T, for which
(1) It contains LR(0) item "$A \to \beta$."
(2) AND t is in FOLLOW(A)
can have $R_{A \to \beta}$ entry.

Ex. Previous Example

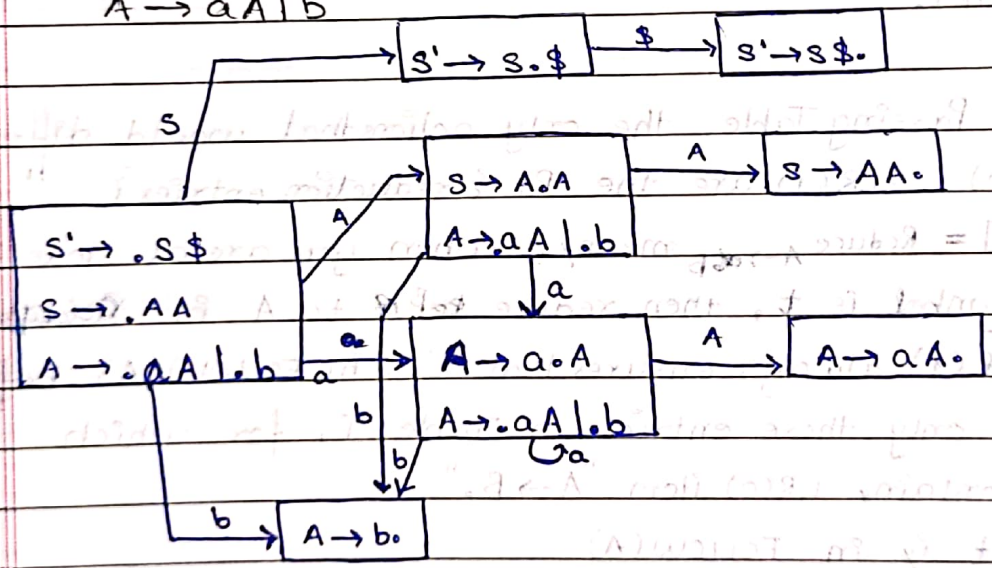
$T[I_2, \$] = R_0$ {$\because$ FOLLOW(E') = {$}}
$T[I_3, +] = T[I_3, *] = T[I_3, \$] = R_{E \to B}$
$T[I_4, +] = T[I_4, *] = T[I_4, \$] = R_{B \to 0}$
$T[I_5, +] = T[I_5, *] = T[I_5, \$] = R_{B \to 1}$
$T[I_7, +] = T[I_7, *] = T[I_7, \$] = R_{E \to E*B}$
$T[I_9, +] = T[I_9, *] = T[I_9, \$] = R_{E \to E+B}$.

So, every CFG suitable for LR(0) parsing is also suitable for SLR(1) parsing, i.e. every LR(0) grammar is SLR(1) grammar.
Since we are only reducing entries from LR(0) table to obtain SLR(1) table.

Ex. Check if the following grammar is (1) LR(0) (2) SLR(1):

$S \to AA$

$A \to aA \mid b$



We know that in no cell of LR(0) & SLR(1) parsing table we can have two shift entries since on one symbol we goto only one state.
For non-terminals however it is possible to have

① One shift entry
② One reduce entry
③ One shift, one reduce entry } conflicts
④ Two reduce entries.

The last two {③ & ④} are problem for parsing.

since the parser gets confused as to which action to take. So, for any parser its parsing table shouldn't have multiple entries. For this example, we don't have any state with any entry T[Ii, t] with multiple entries.

So, this is LR(0) grammar.

{ Also, since LR(0), hence LR(1) }

- Checking for SLR(1) conflicts when a state has:

1. Two Reduce entries:

I_i:	$A \to \alpha.$
	$B \to \beta.$

In SLR(1) table, entry T[Ii, t] can have both $R_{A\to\alpha}$ & $R_{B\to\beta}$ iff $t \in$ FOLLOW(A) & $t \in$ Follow(B).

So, iff FOLLOW(A) $\cap$ FOLLOW(B) $= \phi$ then no T[Ii, t] for any t can have multiple entries, i.e. no RR conflict.

2. One Reduce entry, One Shift entry:

I_i:	$A \to \alpha.$
	$B \to \beta.\, t\gamma$

In SLR(1) table, entry T[Ii, t] can have both $R_{A\to\alpha}$ & Sj entry iff $t \in$ FOLLOW(A) & t is terminal. So iff FOLLOW(A) $\cap \{t\} = \phi$ then no SR conflict.
(OR $t \notin$ FOLLOW(A))

- For LR(0), any state having "two reduce entries" or "one shift, one reduce" entry has conflict.

Ex. Check if LL(1), LR(0), SLR(1):

$A \to Cb \mid aCa$

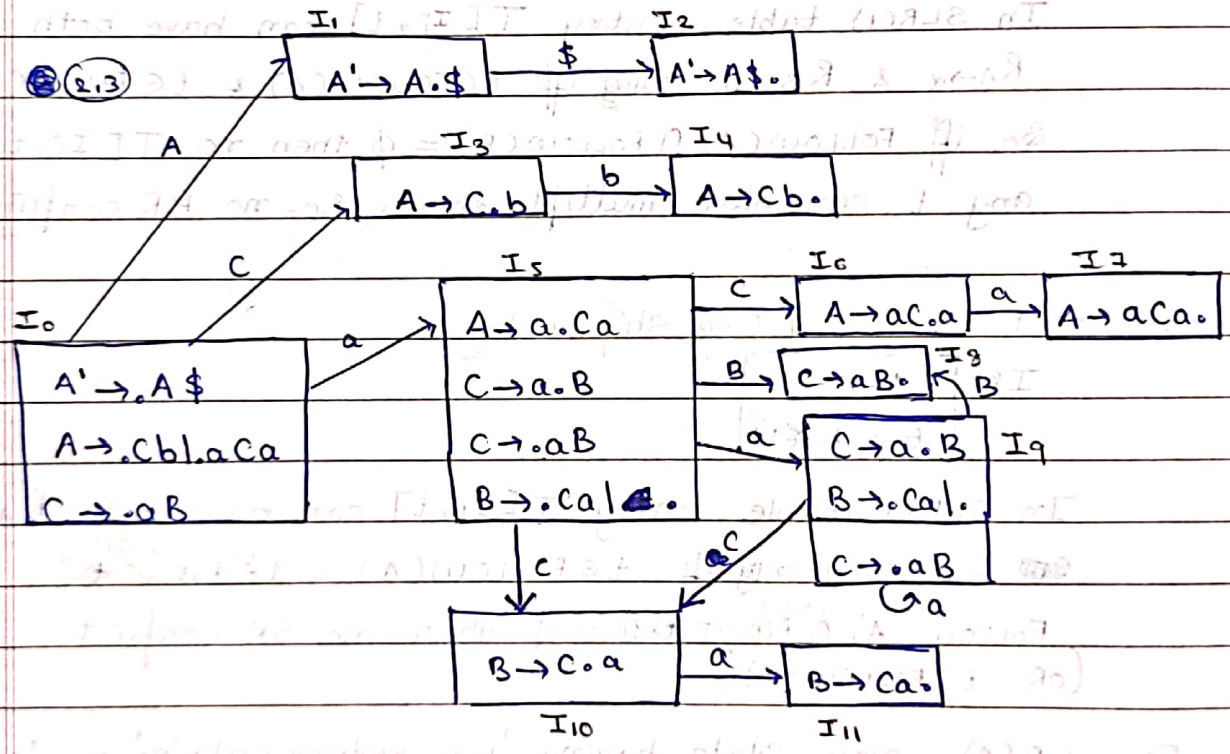
$C \to aB$

$B \to Ca \mid \epsilon$

①

	FIRST	FOLLOW
A	a	$
C	a	b, a
B	a, ∂	b, a

Not LL(1), since, FIRST(Cb) ∩ FIRST(aCa) ≠ ϕ.



Not LR(0). Since I_5 has both shift & reduce entry for 'a'.

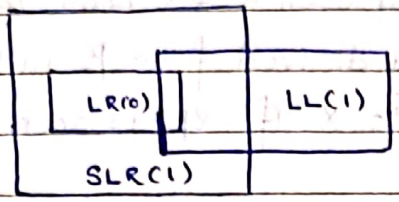
$T[I_5, a] = S_9$ & $R_{B \to \epsilon}$

similarly $T[I_9, a] = S_9$ & $R_{B \to \epsilon}$.

Only I_5 & I_9 have conflicting entries: $B \to .$ (R) & $C \to .aB$ (S). Since $a \in FOLLOW(B)$, hence conflict. Not SLR(1).

CFG, if

- Since every ↑ LR(0) is SLR(1), we can easily say:



LL(1) has no relation w/ LR(0) & SLR(1). They are independent.
i.e. CFG G is:

$$LL(1) \not\longleftrightarrow SLR(1)$$
$$LL(1) \not\longleftrightarrow LR(0)$$

but $LR(0) \longrightarrow SLR(1)$.

- Limitation of SLR(1):
  - While reducing $\alpha\beta a\gamma$ to $\alpha A a\gamma$, SLR(1) only checks if $a \in FOLLOW(A)$. It may be possible, $a \in FOLLOW(A)$ but $a \notin FOLLOW(\alpha A)$. This may lead to the parser not reaching the start state of grammar.
  - LR(1) parser takes into account the whole context, i.e., αA while checking if the next symbol a can follow A or not.
  - So basically SLR(1) parser can't remember the left context α in $\alpha.\beta$.

* LR(1) Parsing & LALR(1) Parsing

- For LR(1) parsing we'll be required to remember the entire left context. To achieve this we'll use the concept of lookahead to keep track of terminals which can follow a state. So, basically lookahead for a LR(1) state is the set of all terminals that can follow the state, i.e. FOLLOW set of the state.

- LR(1) Item: A LR(1) item $X \to \alpha . \beta, a$ means:
  - The parser has already seen α (& α is on ToS)
  - Expects to find string derived from β in the next input.
  - Looking to reduce to X if $\alpha\beta$ is seen and $\alpha\beta$ is followed by 'a'.

- LR(1) parsing also, doesn't always work.

Ex. $S \to Xab \mid aaa$ $\quad\quad$ $\omega = aaa$.

$X \to a$

$aaa \overset{S}{\Rightarrow} a|aa \overset{R}{\Rightarrow} X|aa \overset{S}{\Rightarrow} Xa|a \overset{S}{\Rightarrow} Xaa|$

Show even LR(1) passing strategy can fail.

- Since: SLR(1) allows reduction based on just FOLLOW(A) & LR(1) on FOLLOW(αA), & FOLLOW(αA) $\subseteq$ FOLLOW(A), every CFG which is SLR(1) is also LR(1) parsable.

- Closure of LR(1) Item: For a LR(1) item $S \to \alpha . AB, L$ where A is non-term & L is set of follow terminals, its closure contains:
  (1) Itself
  (2) For All prods of A, $A \to \alpha$, include $A \to . \alpha, L'$, where $L' = \bigcup_{l \in L}$ FIRST(Bl).

Ex. $A \to BC$ $\quad\quad$ Find closure of

$C \to d \mid b$ $\quad\quad$ $A \to .BC, \{a\}$

$B \to a$

closure (A → . BC, {a}) =

A → .BC, {a}

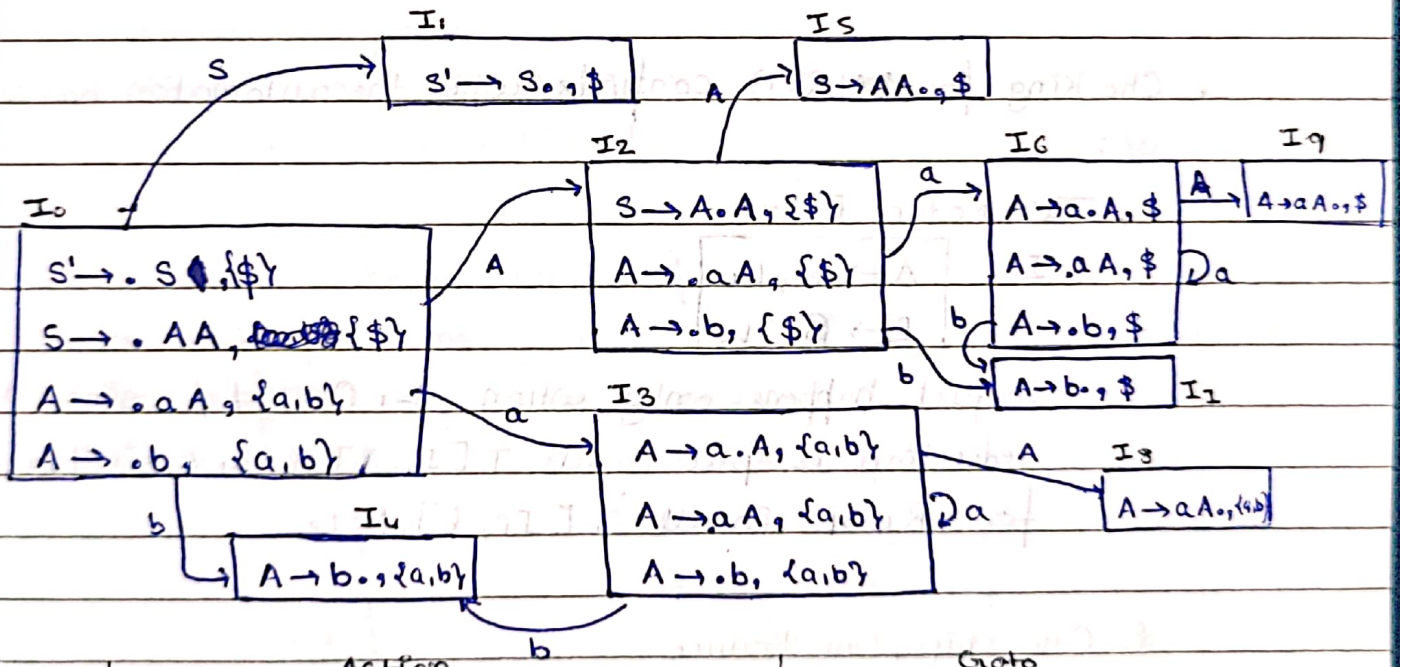
B → .a, {b, a}

- LR(1) Automaton

Ex. S → ① AA

A → ② aA | ③ b

FIRST(S) = {a, b} FIRST(A) = {a, b}



	Action			Goto	
	a	b	$	S	A
I_0	S_3	S_4		1	2
1			(R_0) 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		

- So all entries of LR(1) parsing table are same as that of LR(0) & SLR(1) parsing table, except for reduction entry. For LR(1) parsing table, for a state I_i having LR(1) item of the form: $X \to \alpha \cdot, L$ where L is the set of Lookahead $R_{X \to \alpha}$ is added only to $T[I_i, l]$ where $l \in L$.

- Again for a CFG, if the LALR(1) PT contains more than one entry in any cell, then the grammar is not LALR(1).

- Checking for LR(1) conflicts when the automaton has a state w/:

1. Two Reduce Entry

I_i: | $A \to \alpha \cdot, L_1$ |
| $B \to \beta \cdot, L_2$ |

Conflict happens only when $L_1 \cap L_2 \neq \phi$ since $R_{A \to \alpha}$ reduction is put in all $T[I_i, l]$, $l \in L_1$ & similarly for $R_{B \to \beta}$ in all $T[I_i, l]$, $l \in L_2$.

2. One Shift, One Reduce

I_i: | $A \to \alpha \cdot, L_1$ |
| $B \to \beta \cdot a\gamma, L_2$ |

Conflict happens when, $a \in L_1$.
Since $T[I_i, a] = I_j$ for some j
& if $a \in L_1$ then $T[I_i, a] = R_{A \to \alpha}$.

- LALR(1) Parsing means: Look Ahead Left to Right Reverse rightmost derivation.

LALR(1) Automaton can be created from LR(1) automaton by combining all the states which only differ by lookahead.

Ex.

I_i: $A \to \alpha . \beta, l_{11}$; $B \to \gamma . \delta, l_{12}$ $\qquad$ I_j: $A \to \alpha . B, l_{21}$; $B \to \gamma . \delta, l_{22}$

can be merged as:

I_{ij}: $A \to \alpha . \beta, l_{11} \cup l_{21}$; $B \to \gamma . \delta, l_{12} \cup l_{22}$

- Same rules to check conflict apply as LR(1)
- Notice that for some grammar G, if its ~~Auto~~ LR(1) DFA has no ~~conflict~~ shift-reduce conflict, then its LALR(1) DFA would also not have any SR conflict.

I_i: $A \to \alpha ., L_{11}$; $B \to \beta . a\gamma, L_{12}$ $\qquad$ I_j: $A \to \alpha ., L_{21}$; $B \to \beta . a\gamma, L_{22}$

if $a \notin L_{11} \wedge a \notin L_{12}$

then $a \notin L_{11} \cup L_{12}$.

- Notice that if a CFG is not LR(1) then it's not LALR(1) either, since we merge states to get LALR(1) DFA from LR(1) DFA, so this can never reduce conflict {rather it may introduce conflicts}.

$\therefore$ G is not LR(1) $\longrightarrow$ G is not LALR(1).

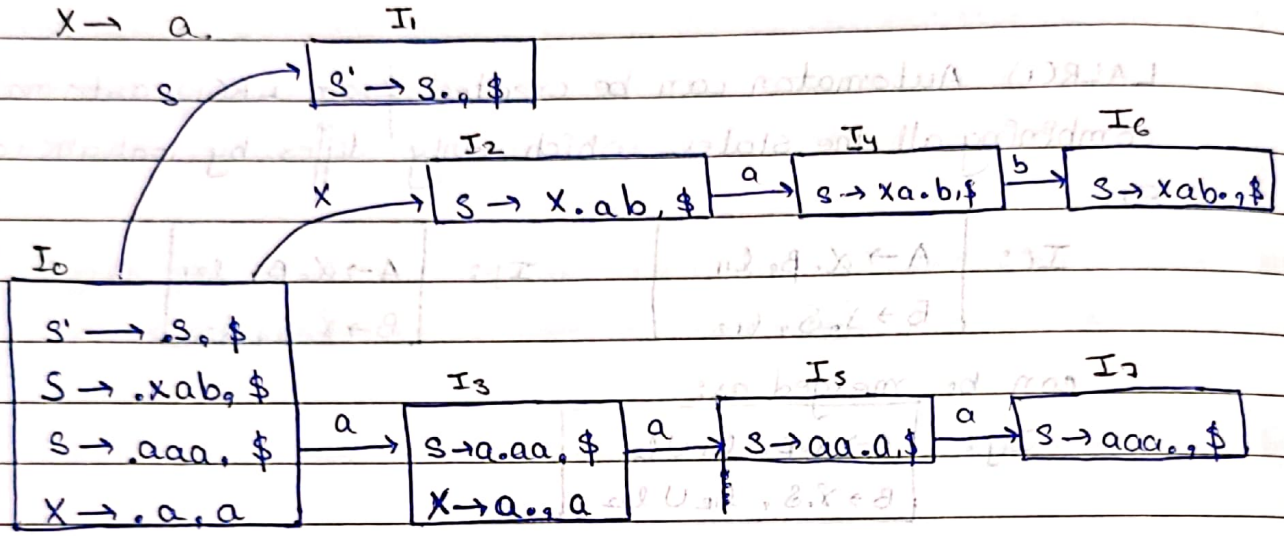
So,

If G is LALR(1) then G is LR(1).

Ex. Check if LR(1)

S → Xab | aaa

X → a.



In I_3: a ∈ LookAhead (X → a., a).

∴ Not LR(1).

So, the confusion at I_3 is: ① If a is already seen & next symbol is 'a', we can reduce. Also ② If a is already seen & a is next symbol is 'a' we can shift to I_5.

Ex. w = aab

$\overset{\downarrow}{a}ab \Longrightarrow a\overset{\downarrow}{a}b$ At this state the parser is confused.

(I_0) (I_3)

If the lookahead was of length 2, this confusion could've been avoided. We could say, if lookahead is 'aa', shift. If lookahead is 'ab' reduce.

Ex. A CFG G, if G is LR(1) but not LALR(1), then is there's a SR conflict in LALR(1) parsing table of G.

No. If G is LR(1), then there's no SR conflict in LR(1) PT. If no SR conflict in LR(1) PT, then no SR conflict in LALR(1) PT possible. So, if G is LR(1) but not LALR(1), there must be a RR conflict in LALR(1) table.

- Ambiguous grammar can't be LR grammar, since for some string there would be two (or more) rightmost derivations. So, at some step γ_i in RMD:

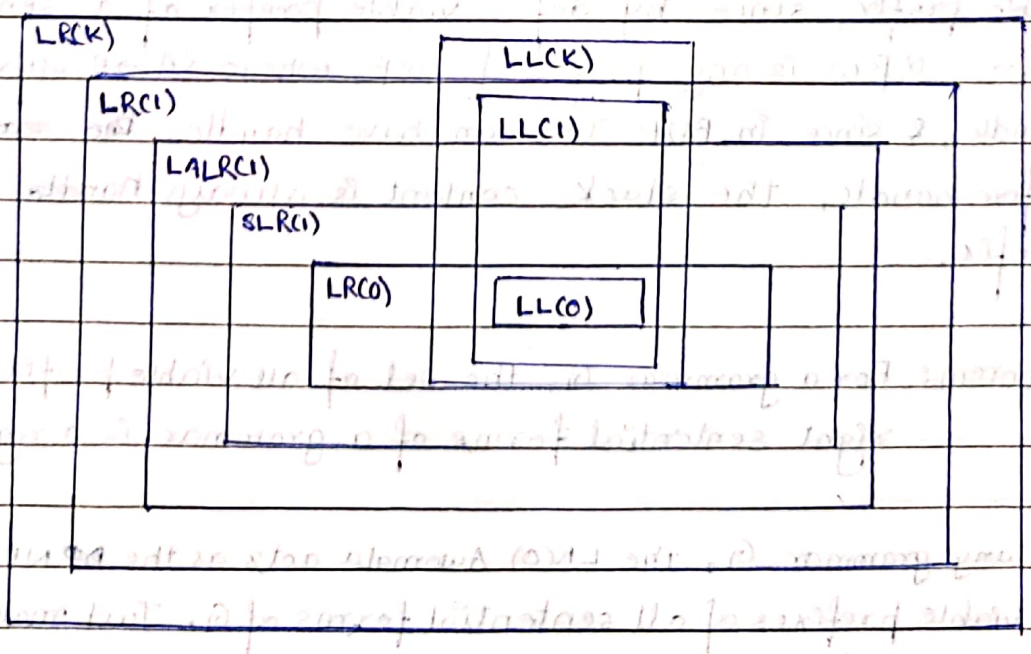
$$S \rightarrow \gamma_1 \rightarrow \gamma_2 \dots \rightarrow \gamma_i \begin{cases} \alpha \dots w \\ \beta \dots w \end{cases}$$

there would be two paths in the RMD of w.
So in reverse RMD op^n, the Parser can't know either to reduce to α or reduce to β at a certain step.

- Since every SLR(1) is also LR(1) & LALR(1) is performing op^n similar to LR(1) by considering the whole left context while deciding to reduce, every SLR(1) grammar is also LALR(1).

- LL(1) is less powerful than LR(1) parsing strategy, because LL(1) doesn't really care about context of derivation.
Also, left recursion & non-left factoring are not problems for LR grammars.

* Types of Grammars

-


Classification of CFGs (Not CFLs)

- **Note:** Power of a parser is measured in terms of no. of grammars they allow

- LL(1) can generate some subset of DCFL languages.
- LL(K) ⊂ LL(1) ⊂ LL(2) ⊂ LL(3) ... LL(K)
- LR(0) ⊂ LR(1) ⊂ LR(2) ⊂ ... ⊂ LR(K) {in terms of #grammars they allow}
- LR(1) ≡ LR(2) ≡ LR(3) ... LR(K) {in terms of lang. they produce}
- LR(0) generates DCFL w/ Prefix Property.
- LR(K) K>0 generates DCFL languages
- Although LR(1) is more powerful than SLR(1) since it can parse all the grammar that SLR(1) can & more. But both LR(1) & SLR(1) can generate DCFL languages, i.e., for a LR(1) grammar G w/ lang. L(G) there exists some SLR(1) grammar G' w/ lang L(G') such that L(G) = L(G')
- If a CFG G is LL(1) & ∄ L(G), then G is SLR(1) also.
- For a CFG G to be LL(K), it means that for any non-terminal A, deciding which of A's production to use for the next derivation step, takes atmost K symbols look-ahead on the input buffer.

- In bottom up parsing of a string w∈L(G), whatever that appears on the stack {or α part of α|β} must be a viable prefix, since by def, viable prefix of a sentential form αβw is any prefix of αβ, where ⟨A→β, after a⟩ is a handle, & since in BUP ToS can have handle, the or string before handle, the stack content is always viable prefix.

- Theorem: For a grammar G, the set of all viable prefixes of all the right sentential forms of a grammar is a regular language

- For any grammar G, the LR(0) Automata acts as the NFA to recognize all viable prefixes of all sentential forms of G. Just make every state as final.

Semantic Analysis

- Syntax Analysis checks for structural or syntactical problems with the program.
  Semantic Analysis checks if the meaning of program is clear or not & correct or not.

Ex. `a = b + c;`
In C, this is perfectly OK syntax. `id = id + id;`
But suppose that b is int, c is char array & b is double.
Does it make sense? `double = int + char[];`
It's the job of Symantic Analyzer to check the meaning of program.

* - Syntax Directed Definition: SDD (is a grammar that) specifies the value of attr. by associating semantic rules w/ grammar productions.
  Ex. $E \rightarrow E_1 + T$ $\quad E.val = E_1.val + T.val$

- Syntax Directed Translation: SDT scheme (is a grammar that) embeds program fragments called semantic actions within production body.

  Ex: $E \rightarrow E_1 + \{print\ '+'\}\ T$ $\quad$ Ex. $E \rightarrow E_1 + T\ \{E.val = E_1.val + T.val\}$

- These are two famously used Semantic Analysis schemes.
- SDD has the rule separate from production body & the order of the execution of rule is not specified.
- SDT ~~can~~ have the ~~rule~~ actions embedded within the production, could be in start, middle or end. Their position specifies the order of execution.

- Attribute Grammar: A Syntax Directed Def^n (SDD) in which the func. in the semantic rules can't have side effect {i.e. the rules only are allowed to calculate the attribute values}

* Attributes:

- Synthesized Attr: For a node, if the value of an attr. is computed from the values of only the child nodes of that node & itself then the attr. is called Synthesized.

- Inherited Attr: For a node, if the value of an attribute is computed only in terms of its parent, sibling & itself, then the attr. is called Inherited.

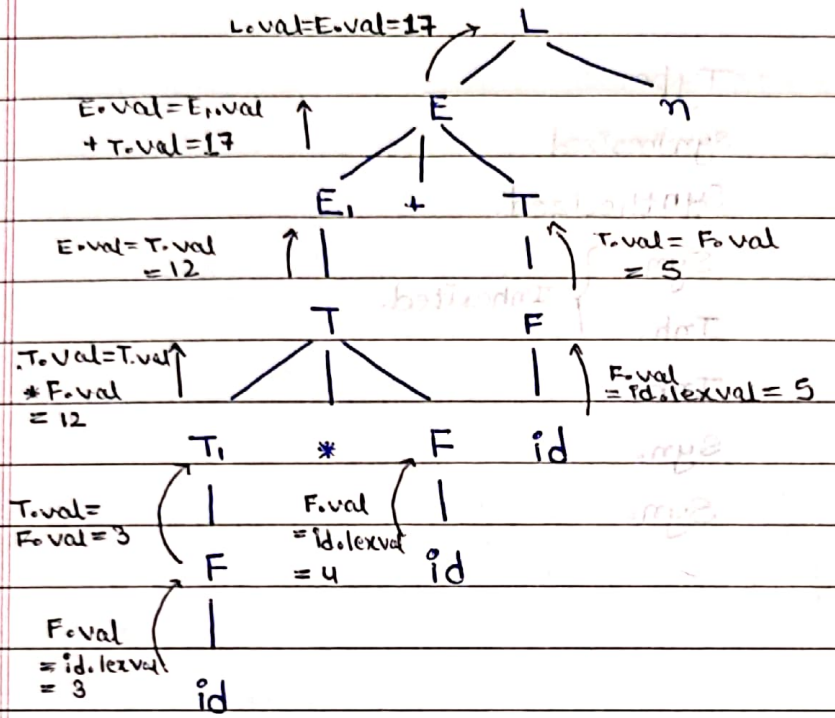
- Although Inherited attrs. of a node N can't be defined in terms of N's children, but Synthesized attr. of N can be defined in terms of Inherited attrs. of N.

- A leaf node (terminal) can only have synthesized attrs, the value of which is supplied by the lexical analyzer.

- An Attribute Grammar with only synthesized attrs, is called S-attributed grammar.

Ex.		
	$L \rightarrow En$	$L.val = E.val$
	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
	$E \rightarrow T$	$E.val = T.val$
	$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
	$T \rightarrow F$	$T.val = F.val$
	$F \rightarrow (E)$	$F.val = E.val$
	$F \rightarrow digit$	$F.val = digit.lexval$

$w = 3 * 4 + 5n$



Since the SDD rules have no sideeffect & only compute attr. values, theg Grammar is Attribute Grammar.
Since all attributes are synthesized, the AG is S-Attributed Grammar.

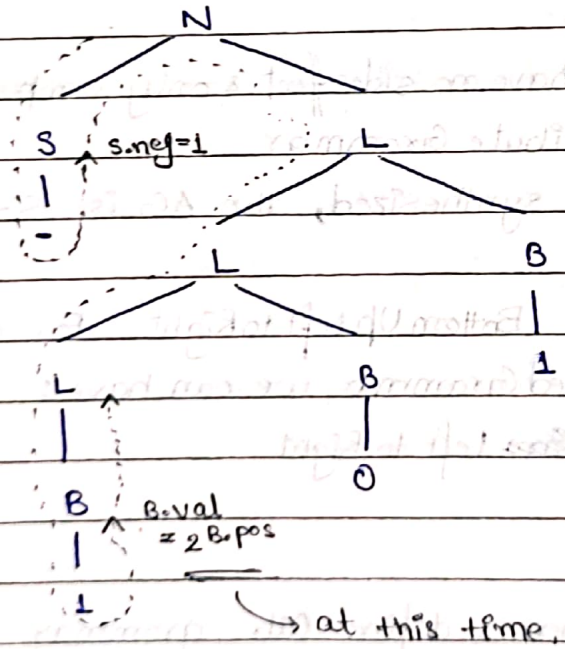
- The above approach is Bottom Up Left to Right. For evaluation of Parse Tree of S-Attributed Grammar, we can have:
(1) Bottom Up ~~Attributed Tree~~ Left to Right
(2) Topdown

Ex. Consider the grammar & define attr. grammar for for its evaluation: ~~Binary to Decimal no.~~ binary no. to decimal no.

			L.pos=0;
Num → Sign List	⇒	Num → S L	Num.val = $(-1)^{S.neg}$ × L.val
Sign → + | -		S → +	S.neg = 0
Lst → List Bit | Bit		S → -	S.neg = 1
Bit → 0 | 1		L → L_1 B	B.pos = L.pos; L_1.pos = L.pos+1; L.V = L_1.V + B.V
		L → B	B.pos = L.pos; L.val = B.val
		B → 0	B.val = 0
		B → 1	B.val = $2^{B.pos}$

Attribute	Type
Num.val	Synthesized.
S.neg	Synthesized.
List.pos	Syn. } Inherited.
L.pos	Inh. } Inherited.
B.pos	Inh.
L.val	Syn.
B.val	Syn.

Say, w = -101



→ at this time, we can't calculate B.val since we don't have B.pos.

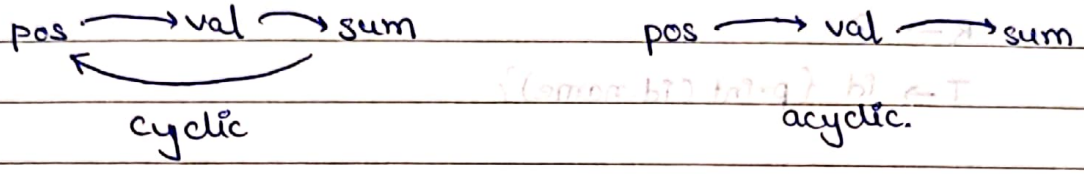
So, bottomup left to right approach doesn't work too well.

The attr. 'pos' flows downwards. Attr. 'val' flows upward.

So, we can do one topdown pass & one bottom up pass in that order since, val depends on pos.

- In general we can evaluate SDDs with attribute dependency as long as the dependency isn't cyclic.

Ex.



- Detecting cyclic dependency is exponential complexity problem. So, we restrict our discussion to the following SDDs:
  (1) S-attributed
  (2) L-attributed

  Since they always guarentee acyclic attribute dependency.

- In general the evaluation of attributes involve:
  - Construct Parse Tree.
  - Construct attr. dependency graph & perform topological sort.
  - Evaluate attrs in the topological sort order.

* S-Attributed SDD: An SDD is S-attributed iff it only has synthesized attributes.

- L-Attributed SDD: A SDD is L-attributed iff each inherited attr. of X_i, where $A \to X_1 X_2 \ldots X_n$, $1 \le i \le n$, the attr. depends only on:
  (1) The attributes of $X_1, X_2, \ldots X_{i-1}$ {i.e left siblings of X_i}
  (2) The inherited attributes of A {i.e. parent}.

- So, by def^n, every "S attributed def^n" is also L-attributed.
- Note that "L-attr. SDD" only imposes restriction on inherited attributes.

Ex. [SDT scheme]

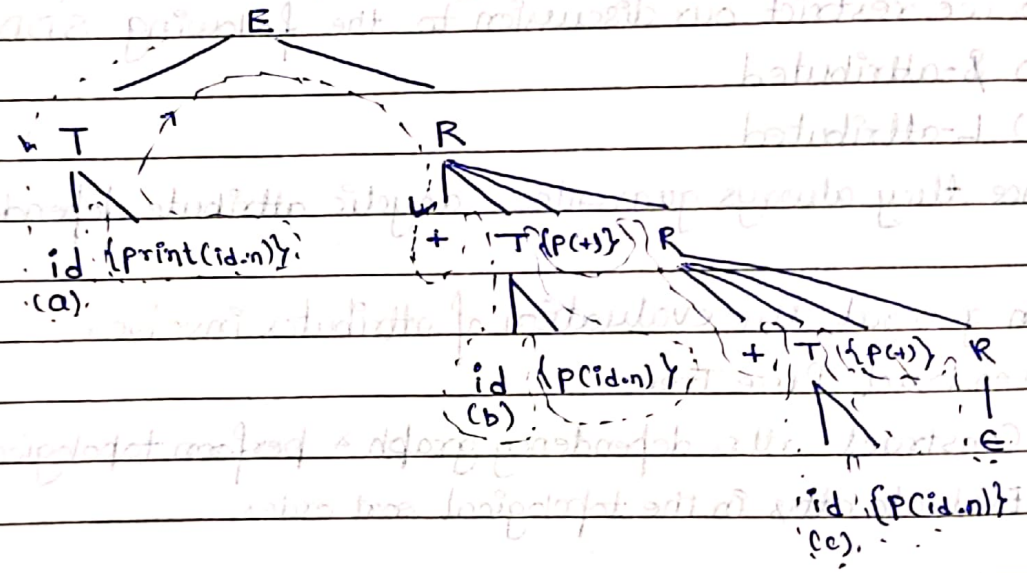
$E \rightarrow TR$

$R \rightarrow +T \ \{print('+')\} \ R_1$

$R \rightarrow \epsilon$

$T \rightarrow id \ \{print(id.name)\}$

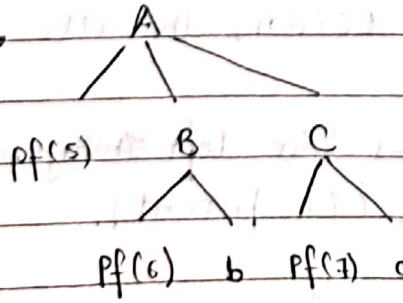
Suppose $w = a+b+c$



Since no attr dependency we can go a single pass Bottom Up or Top Down.

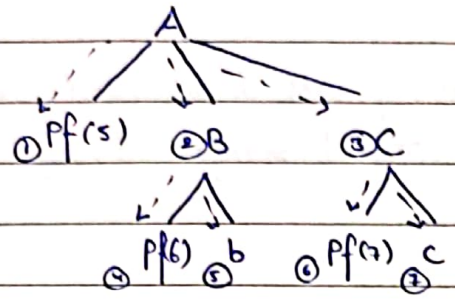
Output : ab+c+

Ex.



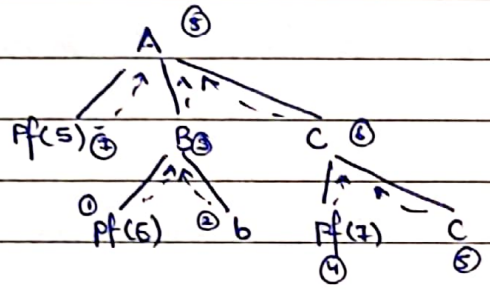
Perform both Top-down & bottomup parsing & record the output of semantic actions.

TD:



5 6 7

BUp:



6 7 5.

- In case the semantic actions are embedded before RHS of production (at start) the output of Topdown & bottom can differ. {or it says that SR parser performs actions, before reducing} just

* SDD to SDT.

- S-attributed.

Just embed the rules as action in the end of corresponding production. Works both for Topdown & bottom up evaluation.

- L-attributed

For a production $A \to X_1 X_2 X_3 \ldots X_n$, for the inherited attr. 'in' of X_i, embed the rule right before X_i. For synthesized attr., put the rule as action in the end. Works for Top Down parsing.

Ex. $A \to BCD$ $\quad B.i = A.i$; $C.i = B.i + 2$, $A.s = B.s + 2$

$\Rightarrow$ $A \to \{B.i = A.i\} B \{C.i = B.i + 2\} C D \{A.s = B.s + 2\}$

Note: L attr can be used for Bottom Up parsing as well, but it is not to be discussed.

Intermidiate Code

- Throughout the phases of Compilation, there are various intermidiate representation, such as Syntax Analyzer outputs Passe Tree.

- After the Analysis phase of compiler is over, the synthesis phase turns the representation of the Analysis phase to some "intermidiate code" so as to perform possible optimizations on it.

* Three Address Code

- In three address code atmost one operator is there on the right side of an instruction.

Ex. $x + y * z \Rightarrow t_1 = y * z$

$t_2 = x + t_1$

where t_1 & t_2 are compiler generated temp names.

- In symbol table we have entries for each variable, so in any expression, the pointer to symbol table entry is used. For convinence, we use names instead of address

- In Three Address Code representation, we treat all three as addresses : Variable name, Constants & Compilergenerated variables.

- Assignment is also called Definition.

- Standard TAC forms
  - $x = y \; op \; z$
  - $x = op \; y$
  - $x = y$
  - goto L
  - if x goto L
  - if False x goto L
  - if $x \; op \; y$ goto L
  - Function call $p(x_1, x_2, x_3 \ldots x_n)$:

    param x_1

    param x_2

    :

    param x_n

    call p, n
  - Indexed addressing mode:

    $x = y[i]$ { meaning $x \leftarrow [y + i]$ in Assembly }
  - $x = \&y$
  - $x = *y$
  - $*x = y$

Ex. Convert to TAC:

do i = i+1; while (a[i] < v);

Assume size of elements of a is 8B.

L_1: $t_1 = i + 1$

$i = t_1$

$t_2 = i * 8$

$t_3 = a[t_2]$

if $t_3 < v$ goto L_1

- There are multiple ways to represent TAC in system:
  - Quadruple: <result, oprand1, oprand2, operator>
  - Triple: <op1, op2, operator>
    $op_1 \leftarrow op_1$ operator op_2
  - Indirect Triple

* Static Single Assignment

- The 'static' part means that SSA works on "text" part of the code.

- Each variable in SSA has exactly one definition. This becomes very helpful in performing optimizations such as Constant Propagation.

Ex. Convert to TAC & SSA:

```
int dist(int x, int y){
    x = x*x;
    y = y*y;
    return isqrt(x+y);
}
```

⇒

```
dist(x,y):
x ← x*x
y ← y*y
t0 ← x+y
t1 ← isqrt(t0)
return t1
(TAC)
```

⇒

```
dist(x0, y0):
x1 ← x0 * x0
y1 ← y0 * y0
t0 ← x1 + y1
t1 ← isqrt(t0)
return t1
(SSA)
```

- SSA makes use of "def-use information".
- SSA doesn't restrict loops. Remember that it restricts a variable to have one defn in program text (static) not while running.

Ex. while (x < 5){
        y = x+1
    }

Here, in the text of the program, y appears to be assigned exactly once, although when the code executes, y may get assigned multiple times. So statically y has single def. "y = x + 1"

- SSA allows the use of a special function: ϕ-func., which allows for choosing one of the options as the value to be used.

Ex. x = 0
while (x < 10) {
    y = 2 * x
    x = y+1
}

If we try to convert this to SSA, for y = 2*x; what should we use, x_0 or x_1? similarly at x<10.

$x_0 = 0$
if (x) >= 10 goto L_1
$y_0 = 2 * (x)$
$x_1 = y_0 + 1$
L_1:

For the first time it should be x_0, then it should always be x_1. So, we use ϕ func. here:

$x_0 = 0$
if $\phi(x_0, x_1) >= 10$ goto L_1
$y_0 = 2 * \phi(x_0, x_1)$
$x_1 = y_0 + 1$
L_1:

* IC Optimization

- Code transformations to improve program, mainly in terms of the execution time, (and sometimes in program size), without affecting the desired result of the source program.

- We are mainly concerned with static optimizations, i.e optimizations on program text, i.e. analysis & transformation is done on what the inst^{n}s are, not how they function.

Ex. Suppose the following code:

① $x = y+1$
② $y = 2*z$
③ $x = y+z$
④ $z = 1$
⑤ $z = x$.

which inst^{n}s / statements can be removed without affecting program execution & result?

Inst^n ① & ④ are dead code, for sure.

Ex. Same as above:

① $x = y+1$;
② $y = 2*z$;
③ if (d) $x = y+z$;
④ $z = 1$;
⑤ $z = x$;

① Not dead code. If cond. at ③ might result to False & the value of x at ① may get used at ⑤
②, ③ we can't guarantee dead code by looking at just this block. ④ is definitely dead code since ⑤ overwrites it immediately.

Ex. Same as above:

① while (c) {
② $x = y+1$;
③ $y = 2*z$;
④ if (d) $x = y+z$;
⑤ $z = 1$;
}
⑥ $z = x$;

No dead code. $z=1$ in ⑤ can be used at ③.

- Applying certain optimizations on low-level assembly code is tedious & can differ from vendor to vendor, so, optimizations are performed on IC.

- The static optimization should ensure that all possible executions can occur & the optimization doesn't affect any dynamic execution.

*. Control Flow Graph

- It is a graphical representation of computation & control flow over the static program text.

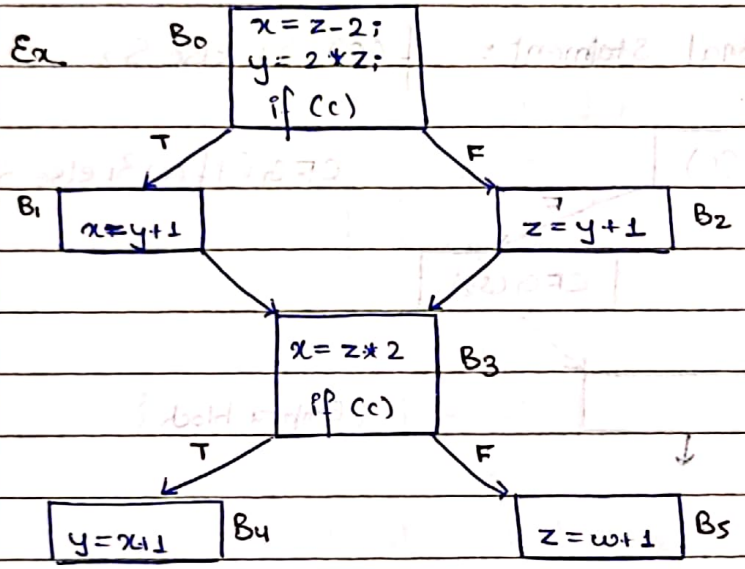
- Nodes of the graph are "basic blocks" & edges represent the possible flow of control from end of a block to beginning of other.

- Basic Block: A sequence of Instⁿs, which guarantee that if the execution happens for them, then they will always execute in the sequence from first to last & Instⁿ & every instⁿ of the block will execute. So, execution of Instⁿs in a Basic Block always starts at the first Instⁿ & all the Instⁿ in the block execute till the last Instⁿ.
So, flow of prog. enters at beginning of block and can only exit from end of the block.

- CFG makes implementation of various optimization techniques convinient.

- CFG models all possible executions of the program. All paths in CFG trace a possible execution.
  It might happen that when you run the program some of the execution path never gets taken, but we can't take any risk while performing static analysis, so CFG models all possible executions.

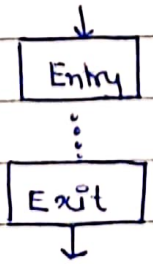
Ex.



Possible executions:
① B0 B1 B3 B4
② B0 B1 B3 B5
③ B0 B2 B3 B4
④ B0 B2 B3 B5

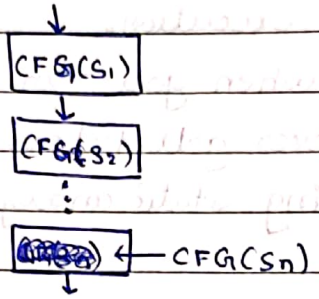
But if c is True, only B0 B1 B3 B4 is possible.
if c is False, only B0 B2 B3 B5 is possible.
So, CFG covers superset of "possible execution during runtime".

- CFG for high-level IR
  - CFG(S): Represents flow graph of high-level stmt S. It is a single entry, single exit graph, i.e. one entry node, one exit node.



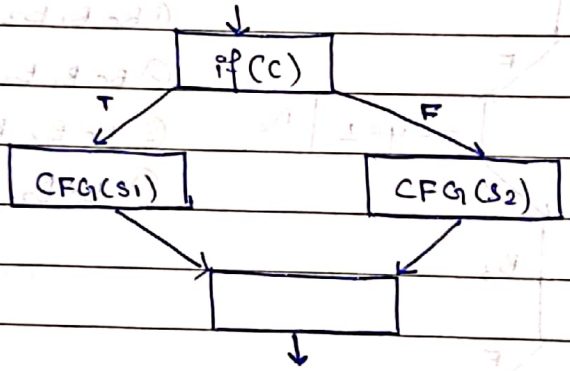
- We recursively define the func. CFG(S)

- CFG for block statements : $S_1; S_2; S_3; \ldots S_n;$



CFG($S_1; S_2; \ldots S_n$)

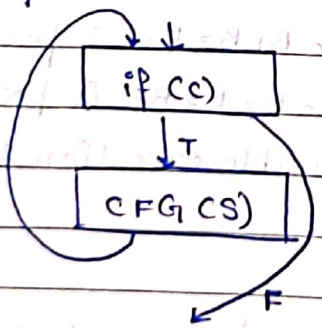
- CFG for conditional statment : if (C) S_1 else S_2



CFG(if (c) S_1 else S_2)

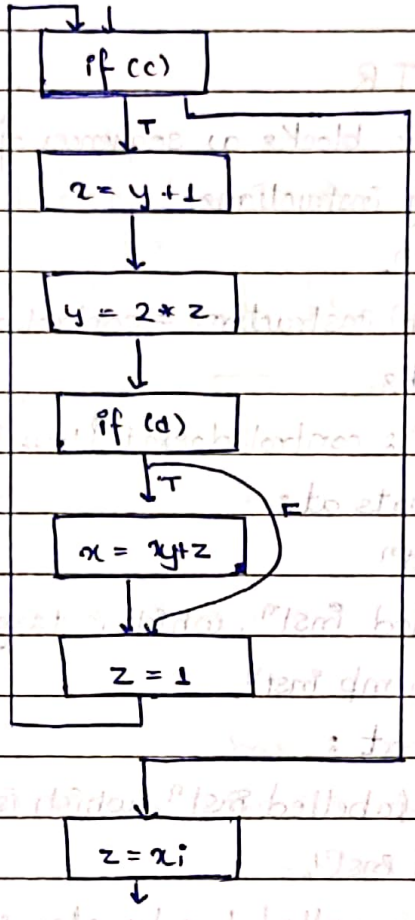
{ Empty block }

- CFG for while (c) S;



Ex. Make CFG for :

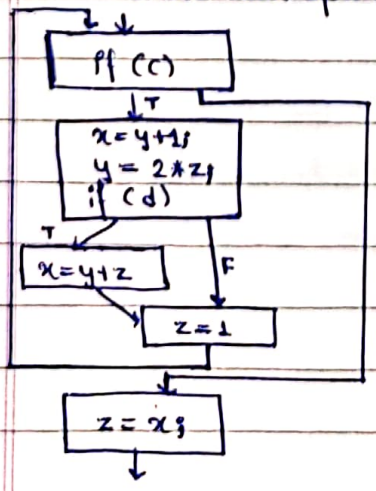
```
while (c) {
        x = y+1;
        y = 2*z
        if (d) x = y+z;
        z = 1;
}
z = x;
```



- This is simple algo., but it is inefficient. We want algo. which minimizes the number of basic blocks.
- There should be no pair of basic blocks (B_1, B_2) such that:
  - B_2 is successor of B_1.
  - B_1 has one outgoing edge.
  - B_2 has one incoming edge.

↳ There should be no empty basic blocks.

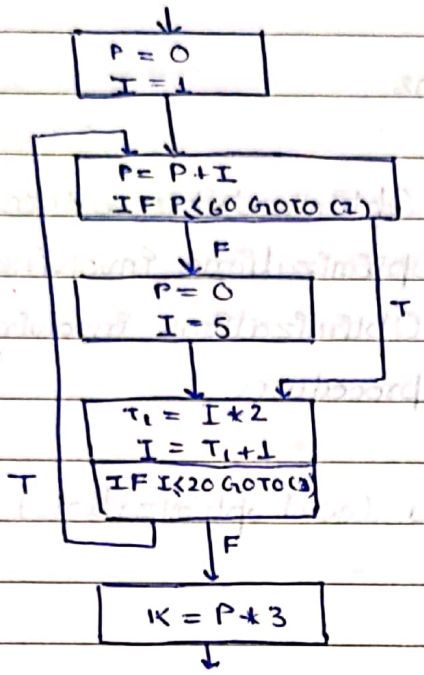
Ex. So, we can optimize above CFG:



- CFG for Lowlevel IR
  - Identify pre-basic blocks as sequences of :
    - Non-branching instructions.
    - Non-label inst^n.
  - No branches (jump) instructions = control doesn't flow out of basic blocks.
  - No labelled inst^n : control doesn't flow into blocks.
  - Basic block starts at :
    - The first inst^n.
    - Every labelled inst^n, which is target of a jump.
    - Right after jump inst^n.
  - Basic block ends at :
    - Right before labelled inst^n, which is target of a jump.
    - Every jump inst^n.
  - The above algo. is called Leader algo. & the first inst^n of every basic block is called leader.
  - In case of Unconditional Jump there would be single successor, and for conditional jump two successor based on the condition.

Note: Some author also put special Entry & Exit nodes. So, check the options of questions accordingly.

Ex. (1) P = 0
(2) I = 1
(3) P = P + I
(4) IF P <= 60 GOTO (7)
(5) P = 0
(6) I = 5
(7) T_1 = I * 2
(8) I = T_1 + 1
(9) IF I <= 20 GOTO (3)
(10) K = P * 3



So, we have 5 leaders. One basic block per leader.

Total no. of basic blocks = 5 (OR 7 If one ENTRY & one EXIT block is also counted.)

- So, the leaders are:
  - The first inst^n
  - Target of branch (jump) inst^n
  - Inst^n that follows a branch (jump) inst^n.

- CFG is created for every procedure in the program.
- For a block B_i, any block B_j, such that $B_j \rightarrow B_i$, is called predecessor of the block B_i.

For a block B_i, any block B_j, such that $B_i \rightarrow B_j$, is called successor of the block B_i.

Note: Most authors include Entry & Exit block along with others. So by default count two extra blocks.

- HALT inst^n equals exit block.

* Types of Optimizations

- Local optimization: Optimizations within a single basic block.
- Global Optimization: Optimizations involving a single CFG.
- Interprocedural Opt: Optimizations involving CFGs of multiple procedures.

(GATE syllabus only has local optimization.)

* Local Optimizations

- Optimization performed within a single basic block.
- Common Subexpression Elimination: If an expression E is calculated & stored in a variable T_1 & then later in T_2, & the value of T_1 & E hasn't changed while T_2 is assigned value of E we can replace E with T_1 in "$T_2 = E$", i.e.,

$$T_1 = a_1 \text{ op } a_2 \quad \Rightarrow \quad T_1 = a_1 \text{ op } a_2$$
$$\vdots \qquad\qquad\qquad\qquad \vdots$$
$$T_2 = a_1 \text{ op } a_2 \qquad\qquad T_2 = T_1$$

- Copy Propagation: If we have a variable assignment, such as "$v_1 = v_2$", then as long as v_1 & v_2 are not reassigned, any statement of the form "$a = \ldots v_1 \ldots$" can be rewritten as "$a = \ldots v_2 \ldots$" provided that such a rewrite is legal.

$$v_1 = v_2 \quad \Rightarrow \quad v_1 = v_2$$
$$\vdots \qquad\qquad\qquad \vdots$$
$$a = v_1 \qquad\qquad a = v_2$$

This helps greatly w/ dead code elimination.

- A variable can be:
  (1) Used = read from
  (2) Defined = Assigned = write to.

Ex. a = b + c; b, c are used. a is defined.

- Dead Code Elimination: An assignment to variable v is called dead if the value of that assignment is never read anywhere. Such dead assignments can be removed.

Note: There is no fixed order in which optimizations can be applied. You may have to apply each optimization multiple times.

Ex.

b = a * a	b = a * a	b = a * a	b = a * a
c = a * a $\xrightarrow{\text{CSElim.}}$	c = b. $\xrightarrow{\text{CProp.}}$	c = b $\xrightarrow{\text{CSElim}}$	c = b
d = b * c	d = b * c	d = b * b	d = b * b
c = b * b.	e = b * b.	e = b * b	e = d.

- Arithmetic Simplification: Replace complex op^{n}s w/ easier ones.
  Ex. $x = 4 * a \Rightarrow x = a \ll 2$.

- Constant Folding: Evaluate expressions at compile time if they have constant value.
  Ex. $x = 4 * 5 \Rightarrow x = 20$

* Optimization & Analyses: Most optimizations are only possible by analyzing the program's behaviour. In order to implement optimizations, we must also talk about the corresponding program analysis.

- Dead Code Elimination requires Liveness Analysis, which helps us know which variables are alive at which point in program.
- Available Expression Analysis helps w/ Common Subexpression Elimination, & Copy Propagation.

* Available Expression Analysis:

- An expression is called available if some variable in the program holds the value of that expression.
- In Common Subexpr. Elimination, we replace available expression by the variable holding its value.
- In Copy Propagation, we replace the use of a variable by the available expression it holds.
- Finding available expressions:
  - Initially no expressions are available.
  - Whenever we execute a statement, $a = b+c$:
    - Any expression having 'a' becomes invalidated. ('a' on LHS or RHS)
    - The expression $a = b+c$ becomes available.
  - If $a = a+c$, then $a = a+c$ doesn't become available.
- Iterate across the basic block, beginning w/ the empty set of expressions & updating available expressions at each

Ex. ① $a = b$; {}

② $c = b$; {a=b}

③ $d = a+b$; {a=b, c=b}

④ $e = a+b$; {a=b, c=b, d=a+b}

⑤ $d = b$; {a=b, c=b, d=a+b, e=a+b}

⑥ $f = a+b$; {a=b, c=b, d=b, e=a+b}

{a=b, c=b, d=b, e=a+b, f=a+b}

Now, we can apply Common Subexpression Elimination using the set of Available Expressions.

```
a = b;
c = a;
d = a + b;
e = d;
d = 8r;
f = e;
```

* Liveness Analysis

- It gives us the set of live & dead variables at a point in program.
- A variable is called Live at a point in program if later in the program, its value is read before being rewritten.
- DCE limination works by computing liveliness for each variable & then eliminating assignment to dead variables.

Ex. ① $a = b$ {b}

② $c = b$ {a, b}

③ $d = a + b$ {a, b} {}

{ we go in reverse order }

- Computing Live Variables
  - To know if a variable will be used at some point, we iterate accross the statements, in a basic block in reverse order.
  - Initially some set of values are known to be alive. (Given)
  - When we see a stmt, $a = b + c$; just before the stmt:
    - The value of a is dead, since its value is about to be overwritten.
    - The value of b & c are alive, since they are about to be read

- If the stmt. is of the form: a = a+b; then both a & b are alive right before the stmt, since both will be read before getting rewritten.

Ex.

{b} - S6
a = b; — a is live after.
{a,b} - S5
c = a; — c is dead after assignment. eliminate c.
{a,b} - S4
d = a+b; — d is live after.
{b,d,a} - S3
e = d; — Don't elim. e is alive after
{b,e,a} - S2
d = a; — d is live after assigment. Don't eliminate
{b,d,e} - S1
f = e; — Since f is not live after assignment, it can be removed.
{b,d} — Initial

∴
{b}
a = b
{b,a}
d = a+b
{b,a,d}
e = d
{b,a}
d = a.
{b,d}

We can again apply liveness analysis. So now e is dead after assignment.

∴
{b}
a = b
{a,b}
d = a+b
{b,a}
d = a.
{b,d}

So, d is dead in stmt 2.

∴
{b}
a = b
{b,a}
d = a
{b,d}

So all vars are live right after assignment.

- We can combine (interleave) LA & DCE together. (It will require just single pass)

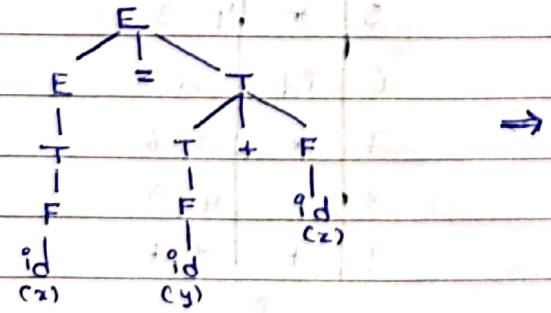
* Other Intermidiate Code

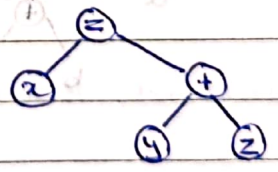
- There are various other Intermidiate Representation along with Three Address Code & Static Single Assignment, such as, Parse Tree, Abstract Syntax Tree, Directed Acyclic Graph, Control Flow Graph, Two Address Code, etc.

- Interestingly, not only the above data structures, but other programming languages may also be used as IRepr., such as C language.

- Abstract Syntax Tree (or Syntax Tree):
  - It is an abstract representation of Parse Tree.
  - The operations are root & intermidiate nodes & operands are on leaf.

Ex. $x = y + z$

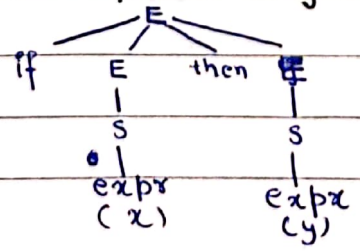


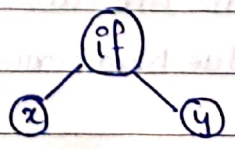


Parse Tree

AST

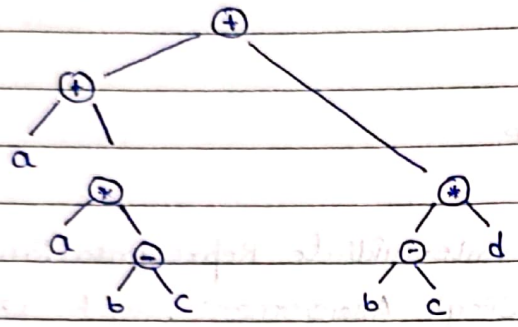
Ex. If x then y





Ex. a + a * (b-c) + (b-c) * d

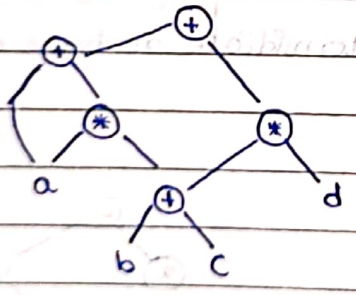
[By default AST is constructed with Precedence & Assoc. in mind]



- The 'Abstract' in AST means that the syntax is abstracted, it only bothers about semantics & it does so by the structure of tree.

- Directed Acyclic Graph: A variant of AST which doesn't recompute expressions more than once.

Ex. a + a * (b+c) + (b+c) * d.



1	id	b
2	id	c
3	+	1 ; 2
4	id	a
5	*	4 ; 3
6	id	d
7	*	3 ; 6
8	+	4 ; 5
9	+	8 ; 7

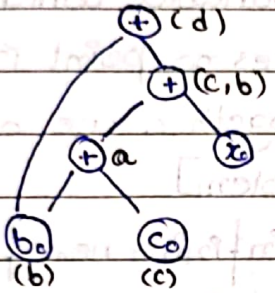
- The above implementation is called Value Numbering.
- DAC can also be used to express basic blocks, also using Value Numbering.

Ex. $a = b + c$ $\quad\quad a_1 = b_0 + c_0$

$c = a + x$ $\quad \Rightarrow \quad c_1 = a_1 + x_0$

$d = b + c$ $\quad\quad d_1 = b_0 + c_1$

$b = a + x$ $\quad\quad b_1 = a_1 + x_0$



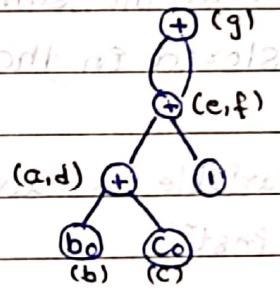
Ex. $a = b + c$ $\quad\quad a_1 = b_0 + c_0$

$e = a + 1$ $\quad\quad e_1 = a_1 + 1$

$d = b + c$ $\quad \Rightarrow \quad d_1 = b_0 + c_0$

$f = d + 1$ $\quad\quad f_1 = d_1 + 1$

$g = e + f$ $\quad\quad g_1 = e_1 + f_1$



- DAG can easily be used for Common Subexpression Elimination.

* There are two types of analysis:
  - Control Flow Analysis: Discovering control structure (such as calls, basic blocks, loops)
  - Data Flow Analysis: Discovering data flow structure (variable use, expression evaluation).

  Ex. Dead code elimination eliminates stmts, the result (data) of which isn't used. So, part of Dataflow Analysis.
  Unreachable code eliminates the stmts/procedures which are never executed under any circumstance. So, its part of Control Flow analysis.

* Liveness Analysis (cotd.)

- Liveness analysis can also be used for register allocation. At every point in prog. we know which variables are live & which are not. So, at a point all the variables which are live must be given different registers. If there's no point in program where variable x & y are live together, we can allocate them same registers. [Graph Coloring Problem]
During IR generation we asume infinite no. of registers, but the system maynot have enough registers to hold all the variables at the same time, so some variables may have to be stored in the m/m. This is called Register Spilling.

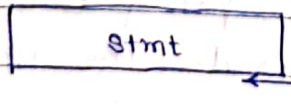
- A variable is live "at an inst^n" if it is live "right before the inst^n".

Ex. S_1: $a = a + c$ — {d, a, b, c}
S_2: $f = a + b$ — {d, a, b}
S_3: $a = d + f$ — {a, d}
S_4: ret a, d — {}

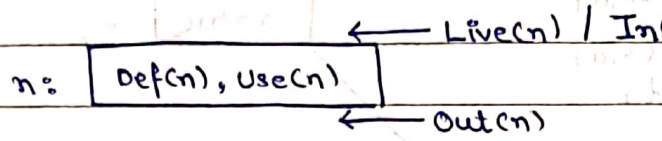
Find live vars at S_1, S_2, S_3 & S_4. Assume no var is live after S_4.

∴ At S_1: {d, a, b, c}
S_2: {d, a, b}
S_3: {d, f}
S_4: {a, d}.

- For a stmt, the variables live right after it are called OUT or OutLive.



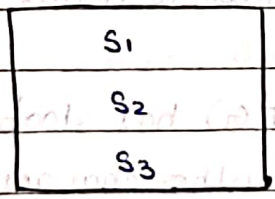
- For a stmt, give its Out-Live, its defined variable & its used variables as:



then, $Live(n) = [Out(n) - Def(n)] \cup Use(n)$.
($Live(n) \equiv In(n)$)

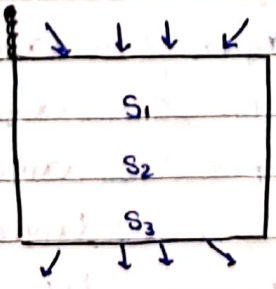
- Live Variable Analysis is backward analysis (for efficiency), but we can also do it in fwd way. For each defn, go to all the instns after it. If the defined var is used before reassignment it is alive at that point in program.

- Use ≡ Read ≡ Reference.
- Define ≡ Assign ≡ Write ≡ Kill
- In a basic block:



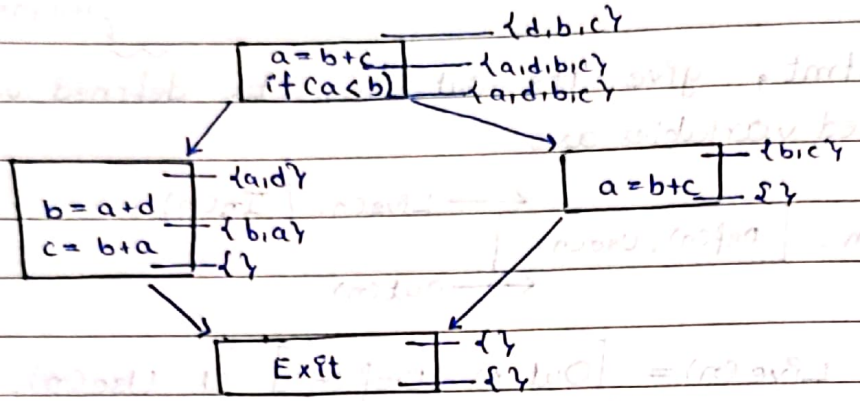
the no. of successor for: $S_1 \Rightarrow 1$, $S_2 \Rightarrow 1$, $S_3 = ?$
For S_3 we can't say there can be 0 or many successors.

- $In(S_3) = Out(S_2)$
- $In(S_2) = Out(S_1)$
- But what would be $Out(S_3)$?



The $Out(S_3)$ = Union of "In" set of all successors of S_3.

Eg.

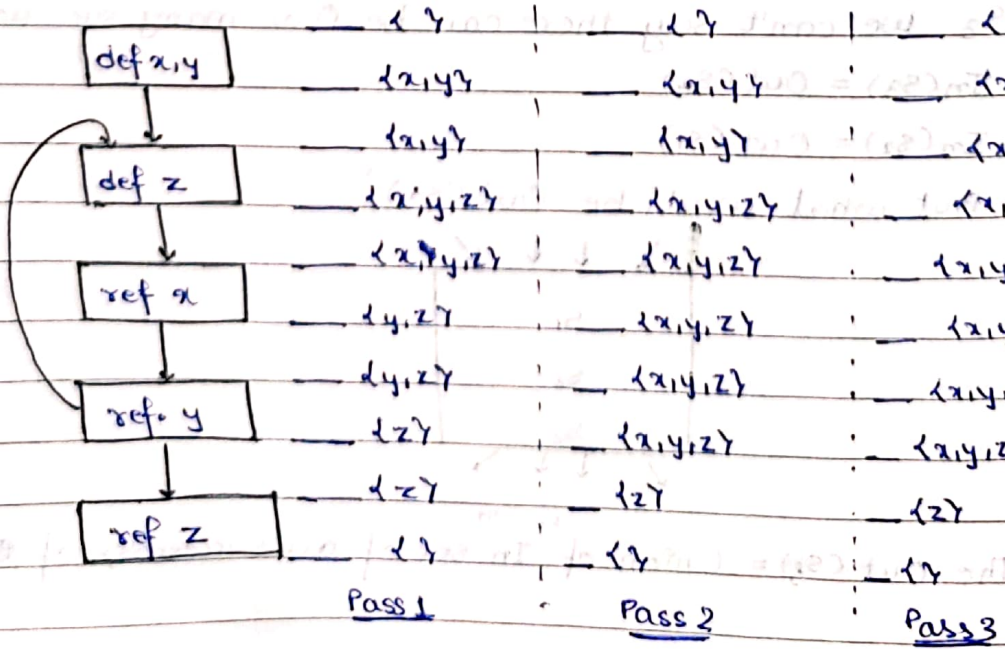


- So we can make a general stmt, that for any stmt S_n, its "Out" set is equal to Union of "In" set of all its successors.

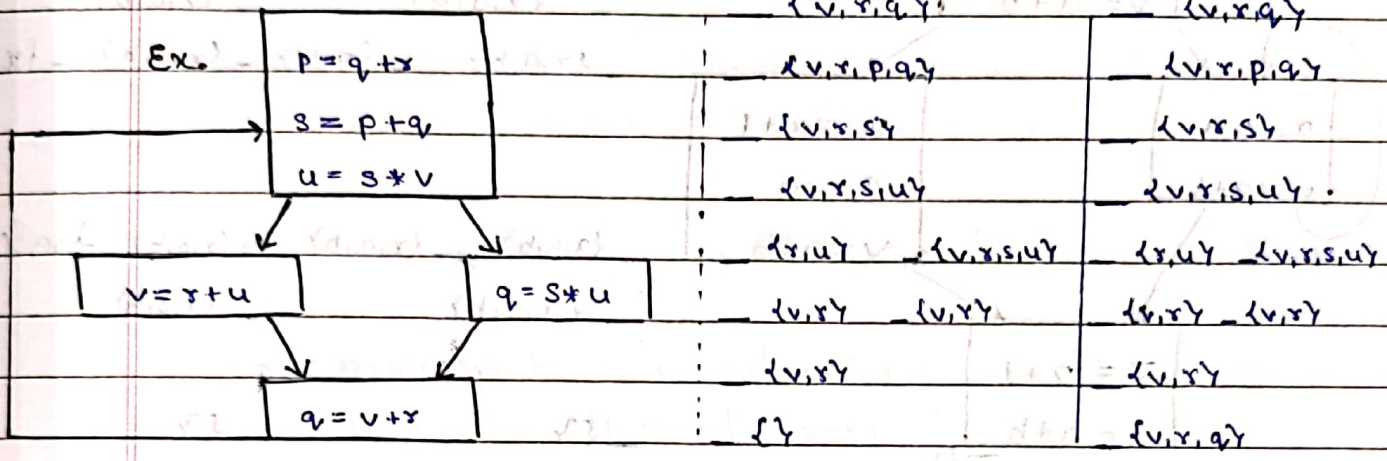
- For a stmt S_n, its "out" set Out(S_n), if:
  - Def(S_n) ⊆ Out(S_n), then the defined variable is alive. Don't remove S_n.
  - Else defined variable is dead. remove.

- If the program (also CFG) has loops we'd require multiple passes to ensure no further changes to In & Out set of stmts are possible.

Ex.



Since Pass 2 & 3 are same, no further changes.



Pass 2 & 3 would be same, so no further changes.

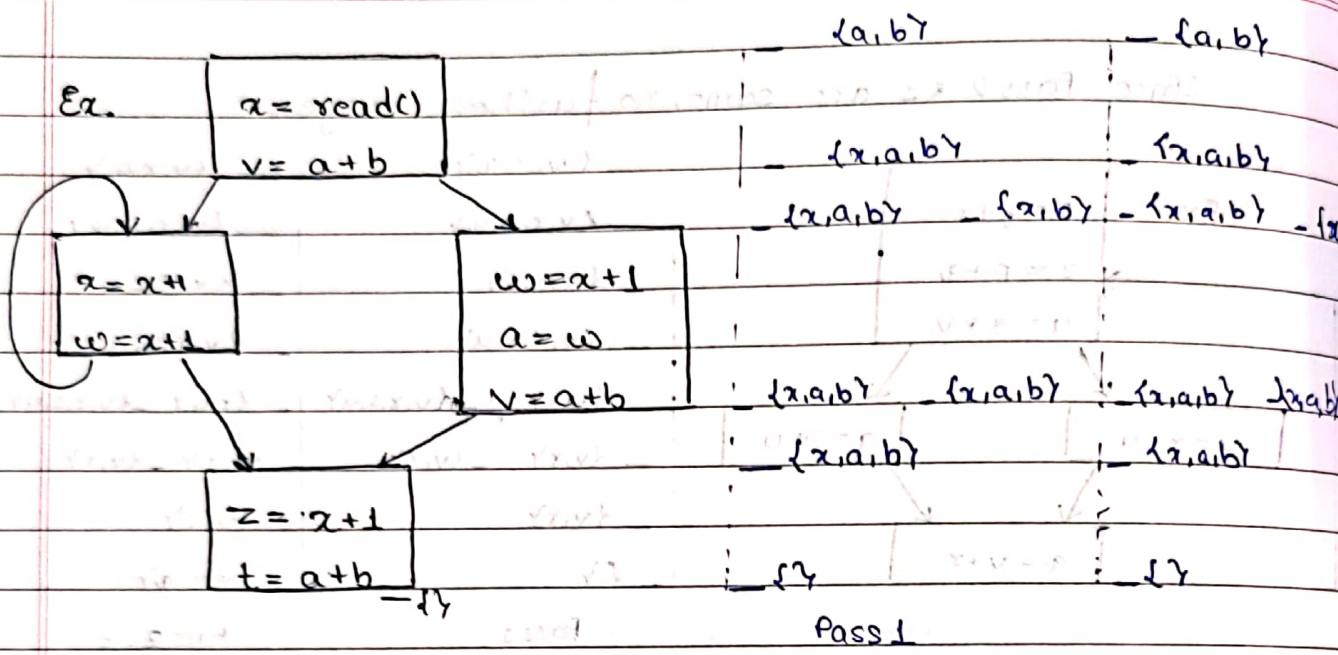
Ex. For the basic block, find defined & used.

a = b + c ;
a = a + b;
b = b + c;
x = b + d;

a is not in Used, since it is defined before being used.
b, c, d are used.
x is defined. b is defined

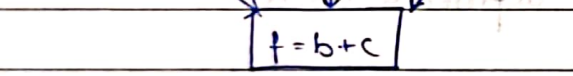
$\therefore$ Def = {a, x, b}
Use = {b, c, d}

- For a basic block B, the Use(B) is the set of all the variables 'b' such that b is used in B, before it is defined.
Def(B) contains all the variables defined within the block.

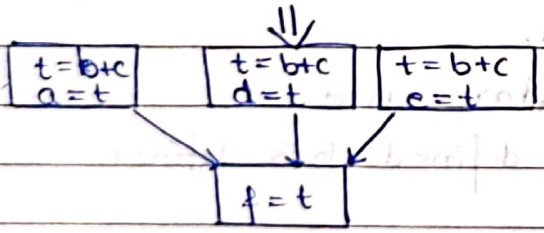


* Compiler puts computed expressions in temporary variables.

Ex. a = b+c, d = b+c, e = b+c

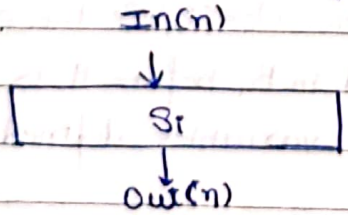


So with which variable should f = b+c be replaced? a? d? e?



So whichever t reaches to f on execution path.

- For available expression analysis; for a stmt Si:



In(n) is the set of available expr. at Si, Out(n) is the set of available expr. after Si.

$$Out(n) = [In(n) \cup Gen(n)] - Kill(n)$$

where $Gen(n)$ for $x = a+b$ is $\{x=a+b\}$ & $Kill(n)$ contains any expr. in $[In(n) \cup Gen(n)]$ with 'x'.

& $In(n)$ = Intersect of $Out(p)$ of all predecesors p of n.

& For a basic block B, an expression is available if it is in $Out(p)$ of all predecesors p of B.

• Since Common Subexpr. elimination can lead to increased no. of temp variables, it leads to program size getting increased.

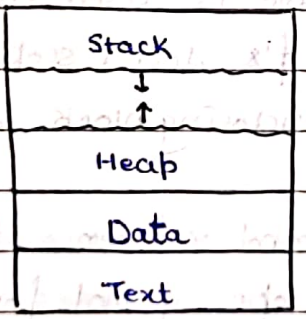
Runtime Environment

* Static: Some issue regarding the program, that can be decided by the compiler (during compilation) is called as "Static".

- Dynamic: Some issue regarding the program that can only be decided by running the program (during runtime) is called as "dynamic".

- Compiler only deals w/ Virtual m/m, i.e. it assumes program starts at loc. 0 & grows up. It can compile the program as such. (→ contiguously.)

- Virtual m/m of a program:



- Static Keyword in C:
  - The static variable is allocated m/m once in the 'Data' area of the program & retains its value b/w function calls. The m/m in the Data section is allocated at Compile Time. The lifetime of such a variable is throughout the program.

- Global Variable vs Static Variable: Global variable is visible to every entity in the program, static variable is only visible to the function it is defined in. If static var. is global, then its scope is limited to the file (compilation unit).

- So the "Data" section & "Text" sections are allocated during compile time.

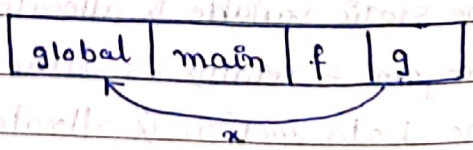
- "Stack" & "Heap" sections are allocated during runtime.

* Scoping

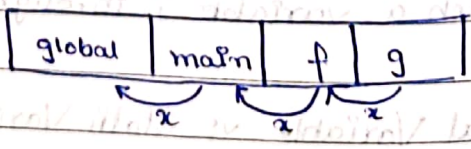
- Static Scoping: For a symbol reference, look for the definition of the symbol at the physical location of the reference, i.e. look within scope, if not found look in the enclosing block, if not found, repeat.
(Lexical Scoping)
Ex. Suppose that a variable 'x' is used in func. f, then 'x' will be first searched in f's local scope, if not found, will be searched in f's enclosing block, i.e. file scope & so on.

- Dynamic Scope: For a symbol reference to x in func. f, look for def^n in f's local scope, if not found, look in f's caller's scope, if not found, repeat.

Ex.
```
int x = 5
void g(){
    pf(x);
}
void f(){
    g();
}
void main(){
    f();
}
```



Static Scoping



Dynamic scoping.

- In the activation record of the func., the access link points to the activation record to which the func. must refer for symbol reference resolution.

* Memory Allocation

- Static Allocation: If the decision of allocation can be made by the compiler looking only at the text of the program.
- Dynamic Allocation: If the decision of allocation can be made only when the program executes.

* Activation Record

When a func. 'f' calls another func. 'g', an activation record is created on the stack for 'g'. Some of the possible contents of Act. Record (from bottom to TOS):

- Actual Parameters passed by f to g.
- Returned Value: Space for g to place the return value.
- Control Link: Pointing to AR of func. to which g would return control, 'f'.
- Access Link: Points to location for access of non-local data.
- Saved M/C Status: Includes return addr (old PC value) & contents of registers
- Local Data: Belonging locally to g.
- Temporary Vars: Used to store temp vals., for expressions, etc.