2/1/18

Compiler Design

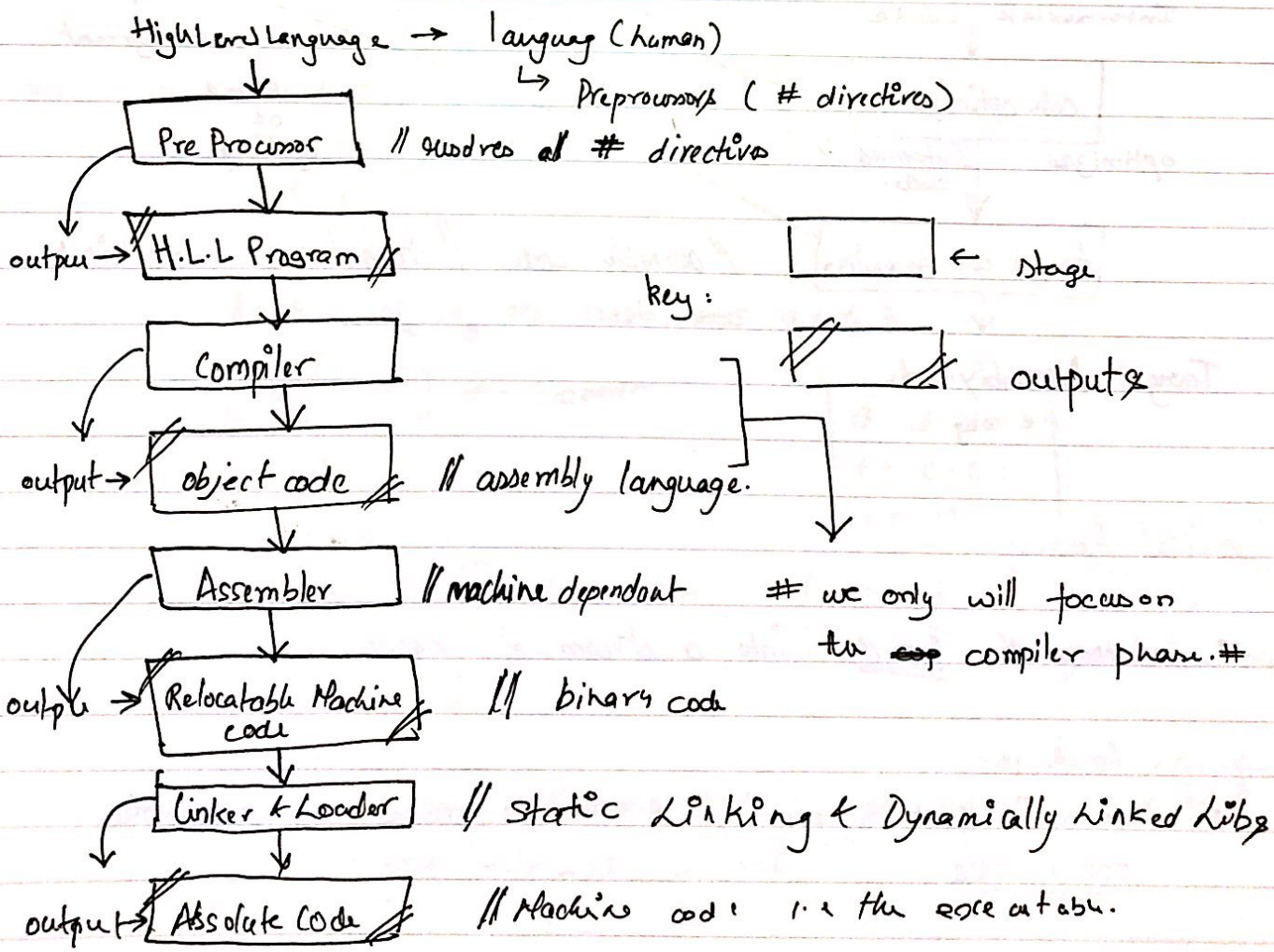
books : 1) [Compiler Design : Aho, Ullman & Sethi
[Dragon series]
↳ 4 books all are same
↳ use Aho, Ullman & Sethi
2) Compiler Design → Rajni Jindal

Syllabus

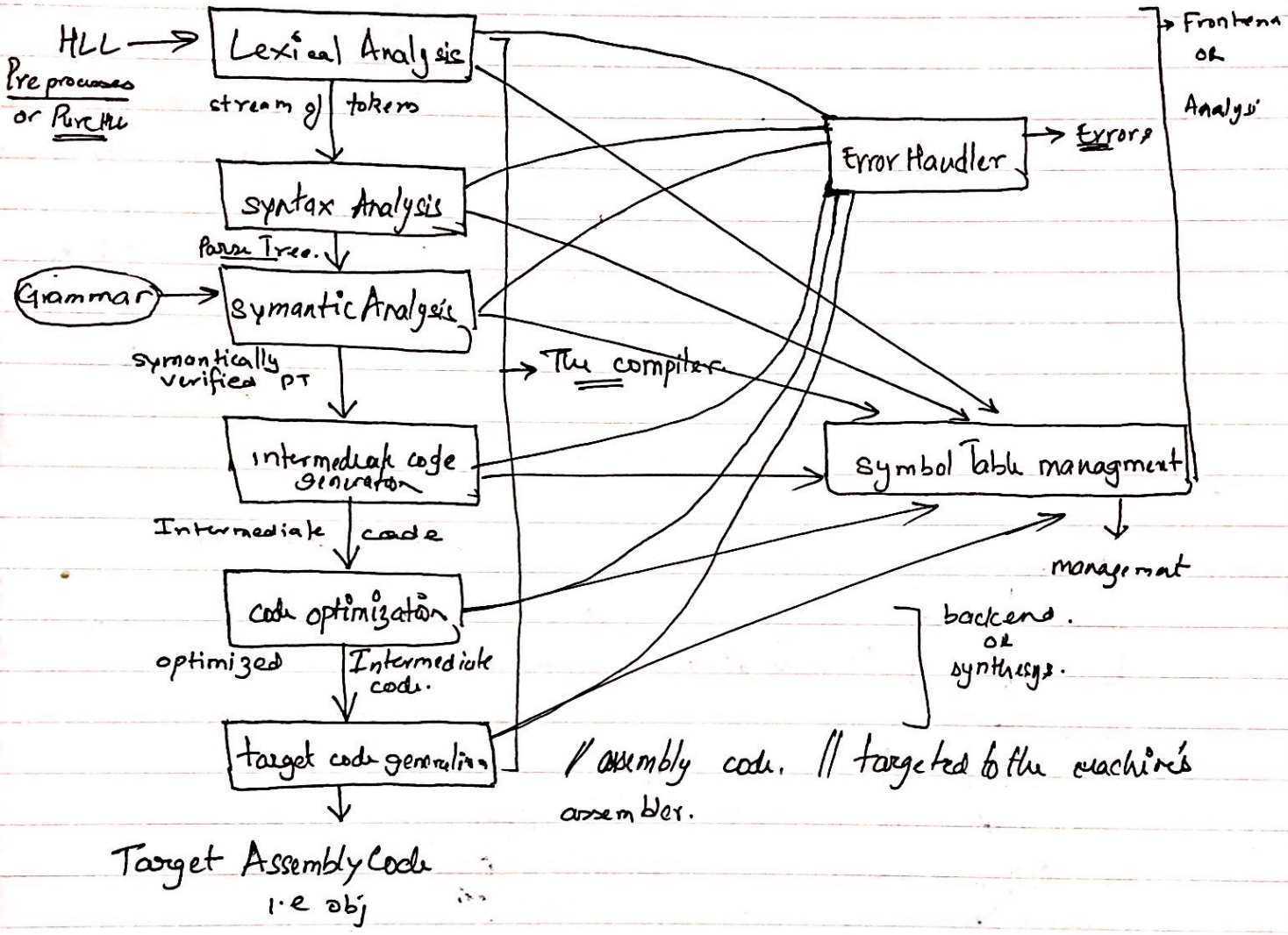
Unit-1 : DFA, NFA, NFA → DFA
RE's

3/1/18

" What is a language translator, why do we need it? "



Phases of the compiler



Lexical Analyzer

→ converts the Pure HLL into a stream of tokens.

Syntax Analyzer

→ the main part of compiler, A.K.A the parse, it builds the parse tree (like a derivation tree)

Symantic Analyzer

It builds the symantically verified parse tree.

Intrmediate Code Generator

It builds the intermediate eg 3 address codes / quadruple rep code
This is a syntax tree (a way to rep IC)

Code optimizer

This optimizes the IC. It make the optimized IC (optimized Syntax tree)

Error handler: " ; is what error "

→ Lexical error : misspelt words. ~~Eg~~ eg if() → fi()

example of Lexical analysis: line

C := a + b ; ⇐ match with RE.

↓

Lex Ana

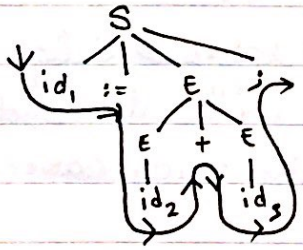
id_1 OP_1 id_2 OP_2 id_3 OP_3 OR $id_1 := id_2 + id_3$;

↓

Sylt Ana ← Grammar

S : id := E ;
E = E + E ;
E = id ;

↓



// parse tree. if tree is similar to the input then it's syntatically correct.

↓

sematic. Ana // type checking eg Float = ch + double ✗

↓

ICG → Optimizer

"How the 1st compiler was compiled".

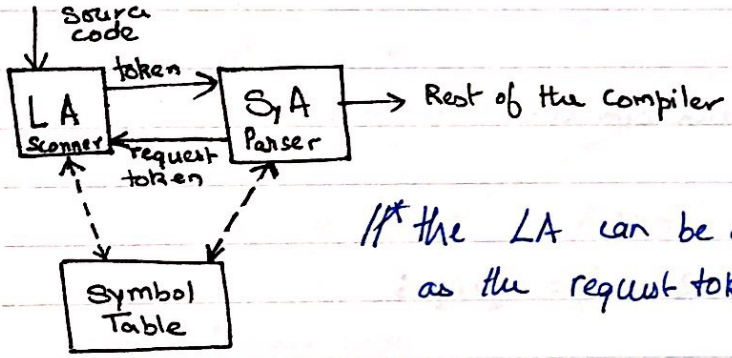
8/1/18

Lexical Analyzer (Scanner)

The reads the source code char by char & translates it into sequence of primitive units called tokens.
eg keywords, identifiers, consts.

Tasks of L.A

1) Read Input, convert to token
2) Removes comments, white spaces (' ', '↦', '\n')
3) Makes a copy of the program with error message marked in it, ie errors are associated with line numbers
[keeps track of '\n' to get line no]



// * the LA can be a subroutine of the parser. as the request token handler. *//

// Advantages of LA as an Independent Phase //

- To simplify the overall design : we are dealing with structure of tokens. The Sy deals with the systactical structure - doing these seperately is much easier and faster.
- Reading the source is the most time consuming part. Seperating this from Sy makes it faster, This allow to have input buffering ~~schemes~~ schemes.
- Changes to seperate modules are much easier to encorporate

- Enhances the portability of the compiler.
- All device related matters consolidated to the L.A.

Design and Implementation of L.A

Hand Implement → using handmade DFAs / NFAs etc

Automatic Generation (short notes) → Using automatic tools like language tools eg LEX, Yak

Design Issues

1> a way to describe tokens (i.e Regex)
2> Mechanism to recognize a token (i.e Recognizers / DFAs)
3> Actions, what to do with a token like buffer, Pass to SA, add to symbol table error/diagonistic messages.

Token, lexeme & Pattern

Lexeme : (An example of token). Is the smallest logical unit of a program or sequence of chars for which a token is produced. (a, b, c, int)

Token : Class of similar lexemes given a name (identifiers)

Pattern : Rule describing a token (Regex, Grammars) $(L[l|d]^*)$

$l(lord)^* \rightarrow$ (T) $\rightarrow$ int a,b,c;

The pattern — identifier — lexemes.

↳ attributes (values, types, scope, other)

<token name, pointer to symbol Table>

eg <keyword, [if (value of keyword)]>
<id, [value of id]>
int → kw | if → kw

[// different keyword will have different entries //]

Issues in implementation.

1> Handling of blanks (if lexeme match it, else don't produce a token)

delimiter → blank, tab, \n
whiteSpace → delimiter.

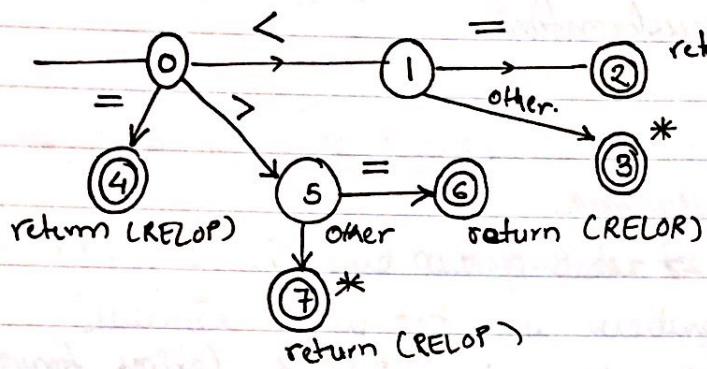
2> different languages treat blanks differently.

DO 5 I = 1.25

↳ some language may call this an identifier.

15/1/18

Lexical Analyzer [DFA as Acceptors]



RELOP: Relational OP.

*: Backtrack when something other is read.

Transition Diagram for Relational Operator.

Lexical Analyzer is nothing but a big transition diagram

- If we fail on one TD we move on to the next TD.
- If we fail on all TD then error recovery must be invoked.
- The more frequently used symbols will be placed near the start state.

What is a Lexical Error?

- Any lexime which does not match any pattern TD
  i.e No token is produced

Error Recovery

i) replace with correct
ii) delete extra char
iii) insert correct
iv) Transpose the two chars.

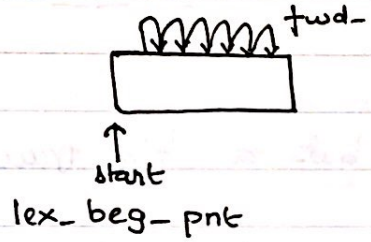
- See whether the prefix of remaining input can be transformed into a valid lexime by single transformation.
- It may take multiple transformations.

Buffering Techniques (Device dependent)

1> Single Buffer 2> two Buffer

Single Buffer

- We have 2 pointers, lexime_beginning_pointer & lexime_forward_pointer.
  (mark start of lexime)

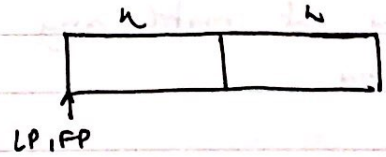


→ Initially both are @ start, then only fwd moves till it finds a match

Drawback

- Buffer size < Lexime size then wrap around happens.
  [common to both]

- ### two Buffer



In one read, read N chars simultaniously into the 1st half of the buffer.

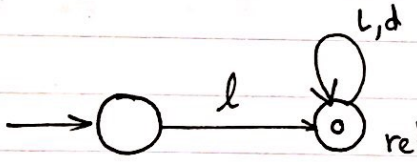
- When FP is about to reach halfway mark, then load the next buffer.

In some languages keywords are not reserved.

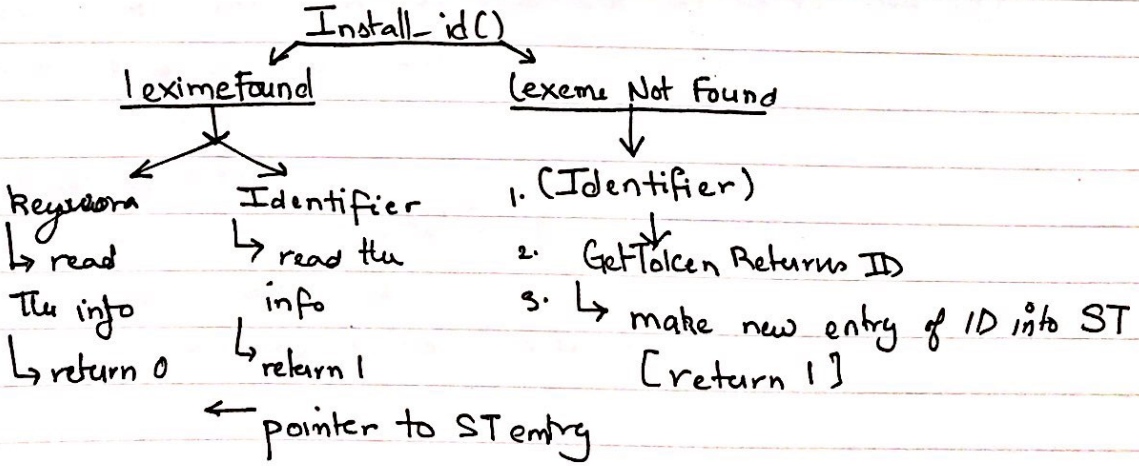
Keywords Vs Identifiers, How to tell them apart.

1) Make a TD and own a code to identify whether it is a identifier / keyword. on reading the end state own code to check

2) Make use of symbol table intelligently. Pre enter all keywords into the symbol table.



GetToken() returning the type (
Install_id() tells which token is to be produced.



Logic Behind Install_id()

benifits :

> Allows easy changes for new keys as we only need to change the symbol table.

If some other method is used then multiple updates may be needed.

17/1/18

Syntax Analyzer (Parser)

The grammar (production rules) provide syntactic structure to the language.

V T P S — start Σ {a, b, c .. z, A, B, C, ... Z}

(production → P; Nonterm → V; Term → T)

stmt → if (exp) else ____ .

This takes a sequence of tokens and converts it into a parse tree.

Q1: Give some example of constructs that cannot be expressed by RE.

↳ RE are incapable of representing program structure eg, loops, conditional
↳ To express this we use CFG.

Q2: When everything can be expressed as CFG, then why do we have regular expressions.

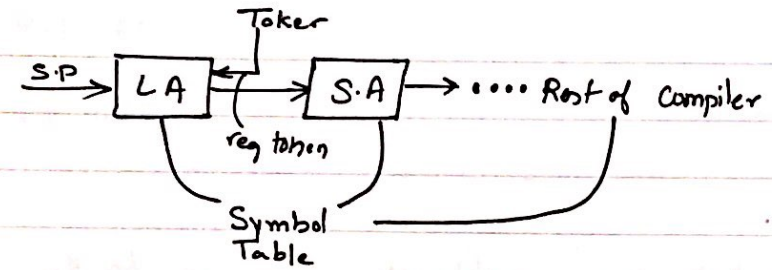
Advantages of CFG:

- Gives precise specification of a language.
- Easy to understand.
- Easy to expand rules. (adding new rules)

SA — Automatic / Manual

Yet Another Compile Compiler

YACC



Role of Parser

1) Generation of Parse tree.
2) Reporting of syntax errors.
3) Recovery from frequently occuring errors & continue parsing.
4) Parser is the most important phase of the compiler.

Types of Parsers

1) CYK Parsers 2) Top Down Parsers 3) Bottom Up Parsers

- CYK Parser uses CYK algos, also called universal parser.
- It can parse any kind of language.
- They are inefficient for producing compilers / compiler generation

Top Down Parsers

- They (Top down) generate P tree starting from the start state
- Bottom up starts with the string and works it way up to the start state
- The TD parser works for subclass of grammar called LL grammer. [L to Right Scan, L definition]
- The BU parser works for subclass of grammar called LR grammar [L to Right Scan, R derivation]
- We read the tree from top to down, left to right.

Error Handling in Parsing Phase

There are 4 classes of errors.

1) Lexical Errors. : misspelt words,
2) Syntactical Error : missing ;, unbalanced })]
3) Symantic Error. : Type mismatch
4) Logical Error: infinite loops.

Role of error Handler

1) Report errors clearly & accurately.
2) Recovery from errors quickly
3) Even if program is correct, it also needs to check that program is not show.

Error Handling & Recovery

1) Report the error type/ diagonostic messag and the location.
(if not exact then at least in the vicinity of the error)

Q3: On detecting error, how should a parser recover from it.

There are 4 strategies

(1) Panic Mode Recovery
(2) phase Recovery
(3) Error Production
(4) Global Correction.

In order to recover, parser jumps from error to a state from where it can continue parsing.

Panic Mode Recovery

- In panic mode we skip the token and move to the next one till it reaches the delimiter. i.e try find the next delimeter & then continue.
  - Look for a special token called the sync token eg delimeters.
  - The skipped part is not structurally checked by the parser.

→ The simplest approach.

Phrase Recovery

- The parser adds some phrase that is expected for error free parsing
- This may lead to spurious errors. (due to addition & deletion by compiler) possibly ∞ errors.

Error Productions

- Add Rule to handle the error into the grammar. These rules are called error productions.
  eg to erase some chars. using ϵ moves.

→ The most common

Global Correction

$$X \; G \longrightarrow Y$$

stat' grammar nearest correct

- As minimum as possible no of transformations from an errous stat X to Y using the Grammar G.

22/1/18

Notational Conventions

Grammar ~~(NTPS)~~

- Nonterminals : Capital letters.
- terminals : initial small letters of the alpha (last letters used for string)
- Start : The LHS of 1st production, unless it's stated.
- Production

Gramatical Symbols (α, β, γ): These can be both NT, & T

$S \xrightarrow{*} \alpha$ [reach in k steps] (sentential form $NT+T = \alpha\beta\gamma$)

sentence → only terminals

$S \rightarrow \alpha$ [reach in one step]

Derivations & trees

\# At every step of derivation, we have 2 choices
1. which NT to expand.
2. Among all alternatives for NT in 1. which one to chose.

Ambiguity : A string for which multiple parse tree exist or more than one LMD or RMD exist.

eg) $\alpha \xrightarrow[LMD]{*} \beta$ [α derives β using LMD]

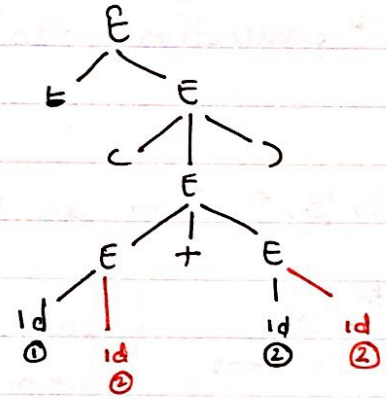
- A parse tree ignores the variations in the order in which symbols are replaced.

eg) $E \rightarrow E+E \mid (E) \mid -E \mid id$

$E \rightarrow -E \rightarrow -(E) \rightarrow -(E+E) \xrightarrow{RMD} -(E+id) \rightarrow -(id+id)$

LMD $\hookrightarrow -(id+E) \rightarrow -(id+id)$

Parse tree



To generate
The Expression
$-(id+id)$

— RMD — LMD

- Every Parse tree has a unique leftmost derivation & unique RMD but every sentence does not have
  1. only ~~only~~ one parse tree
  2. only one LMD
  3. only one RMD

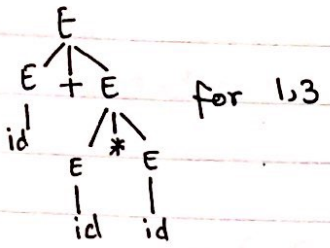
Eg: $E \rightarrow E+E \mid E*E \mid id$

To generate id+id*id

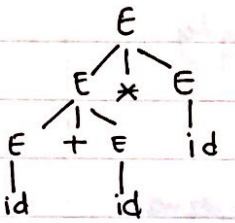
LMD

1. $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E \Rightarrow id+id*E \rightarrow id+id*id$
2. $E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id.$
3. $E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow E+E*id \Rightarrow E+id*id \Rightarrow id+id*id$
4. $E \rightarrow E*E \Rightarrow E*id \Rightarrow E+E*id \Rightarrow E+id*id \Rightarrow id+id*id.$

RMD



for 1,3



for 2,4

// Ambigious Situation //

parser cannot decide which one to pick.

Each trees has only one LMD / RMD, but not v.v.

Ambigious if

1. more than one Parse Tree
2. more than one LMD / RMD

No parser can handle an ambigious grammar except the operator precedence parser.

An ambigious grammar can be converted to an unambigious representation.

$$AG \xrightarrow[\text{Rules.}]{\text{disambiguity}} U_{\wedge} AG$$

Reasons for Ambiguity

eg $id_1 + id_2 * id_2$

when we have operand b/w two operators parser cannot decide to which operator we can assign this operand to.

⇒ 1. The problem of associativity.

2 [redacted]

2. The problem of precedance

when multiple operands exists which to evaluate first.

Solution:

1. define correct associativity, "recursion" helps in defining associativity
2. Introduce levels in the grammar.

↱ Problem for TD parser ↱

recursion → Left Recursion | Right Recursion

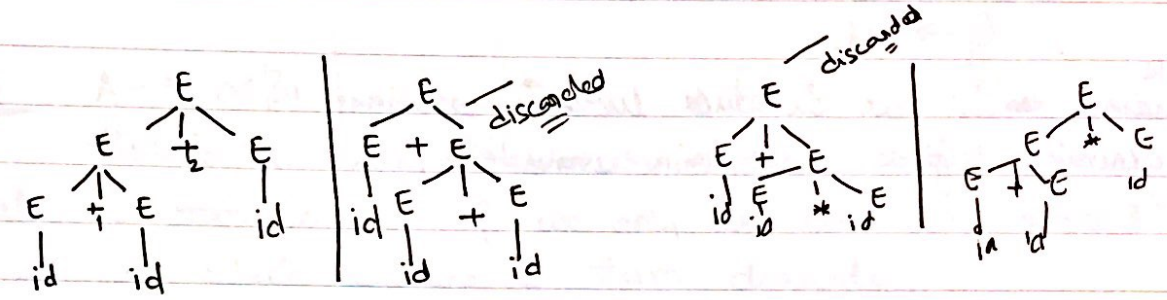
in order to remove ambiguity we associativity we put recursion but that can have problem of its own

due to Left Reaursion, Top DownParser goes into ∞ loop this is solved by changing to Right Recursive.

24/1/23

eg $id +_1 id +_2 id$ | $id + id * id$

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



There are some operators which are both associative to left/right
÷ is not both associative.
Left Associativity in a seq of ops LAs means the leftmost operand is evaluated first. then next...
Generation vs Evaluation.
Every operator has a predefined associativity.

"How to enforce associativity?" (in ambigious grammar)
"How to enfore precedence?" (" " ")

"if we want an op to be left asso, we'll use left recursion & the Right recursion of Right asso."

1. $A \rightarrow A\alpha$ || Left recursion
2. $A \rightarrow \alpha A$ || Right "

To mala + LAsso.

$E \rightarrow E+E$

$= \quad E \rightarrow E+T \mid T$ // now no more LR

$= \quad T \rightarrow T*F \mid F$

$F \rightarrow$ ~~id~~ $G \wedge F \mid G$

$G \rightarrow id.$

To handle precedence ~~too~~ : we introduce levels in grammar.
generate highest last to evaluate it first.

$E \rightarrow E+E \mid E*E \mid id$

$E \rightarrow E+T \mid T$

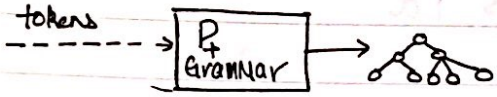
$T \rightarrow T*F \mid F$

$F \rightarrow F \wedge G \mid G$

$F \rightarrow id.$

29/1/18

Non-determinism



eg $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$ (A)

this is an example of nondeterminism.

at a state (say A) if we only see some α ahead then we can't select any one of them directly.

$$N.D \xrightarrow{\text{Left factoring}} \text{Determinism.}$$

all A productions have a common prefix.

∴ this is also called the common prefix problem

then we come to decision by delaying the selection of Right factor as

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$$

Non determinism vs Ambiguity

- Only the path is checked & we don't have the ans
- Two or more ways to reach the solution
- we have the answer.

$$D \not\Rightarrow UA$$

Ques s = " iEtiEtSeS ".

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

$S \rightarrow iEtS \rightarrow iEtiEtSeS$ ①

$S \rightarrow iEtSeS \rightarrow iEtiEtSeS$ ②

∴ G is ambigious.

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

① $S \rightarrow iEtSS' \rightarrow iEtiEtSS' \rightarrow iEtiEtSeS$

② $S \rightarrow iEtSS' \rightarrow iEtSeS \rightarrow iEtiEtSeS$.

↳ Ambigious.

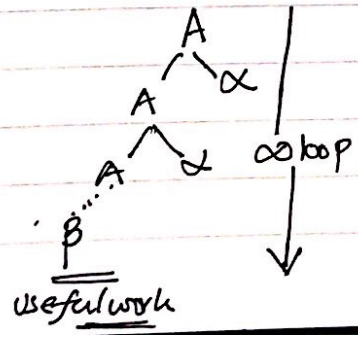
" Determinism doesn't imply unambiguity "

" Non determinism is a problem for parser that cannot backtrack. "

- How does LR become a problem ?

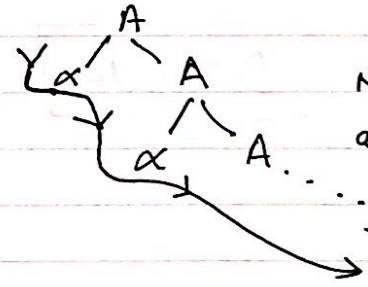
LR

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



RL

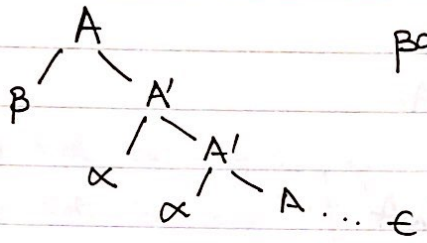
$A \rightarrow \alpha A \mid \beta$



Not an ∞ loop as such due to work being done.

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$



$\beta\alpha^*$

- Left recursion is converted Right Recursion.

eg) $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$

[T

$E \rightarrow E\alpha \mid \epsilon T$

$T \rightarrow T\beta \mid F$

$F \rightarrow d$

Solution

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id.$

$E \rightarrow TE'$

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

$T \rightarrow T\beta \mid$

$T \rightarrow FT'$

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

$F \rightarrow id.$

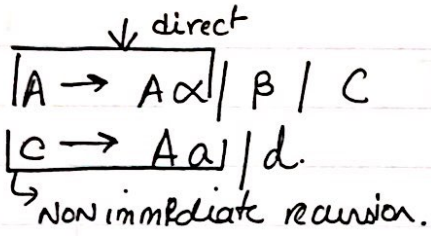
Eg $A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_n | \beta_1 | \beta_2 | \ldots | \beta_m$

$\Rightarrow$

$A \rightarrow \beta_1 A' | \beta_2 A' | \ldots \beta_m A |$

$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_n A' | \epsilon$

This was an example of immediate left recursion.
there can also be ~~indirect~~ NON immediate.



" We have an algo for converting LRG to RRG. ".
→ It handles both immediate & non immediate recursions.
→ when there are cycles in the grammar then this Algo fails & we cannot convert LRG to RRG.

30/1/18

Algorithm to Convert Left Recursion / Remove Left Recursion

$[A \overset{+}{\Rightarrow} A \mid A \to \epsilon]$ Not OK

↳ Cycle ↳ Null Prod

1. There should Not be any cycles.
2. There should Not be any ϵ production.

ip G —Algorithm→ G

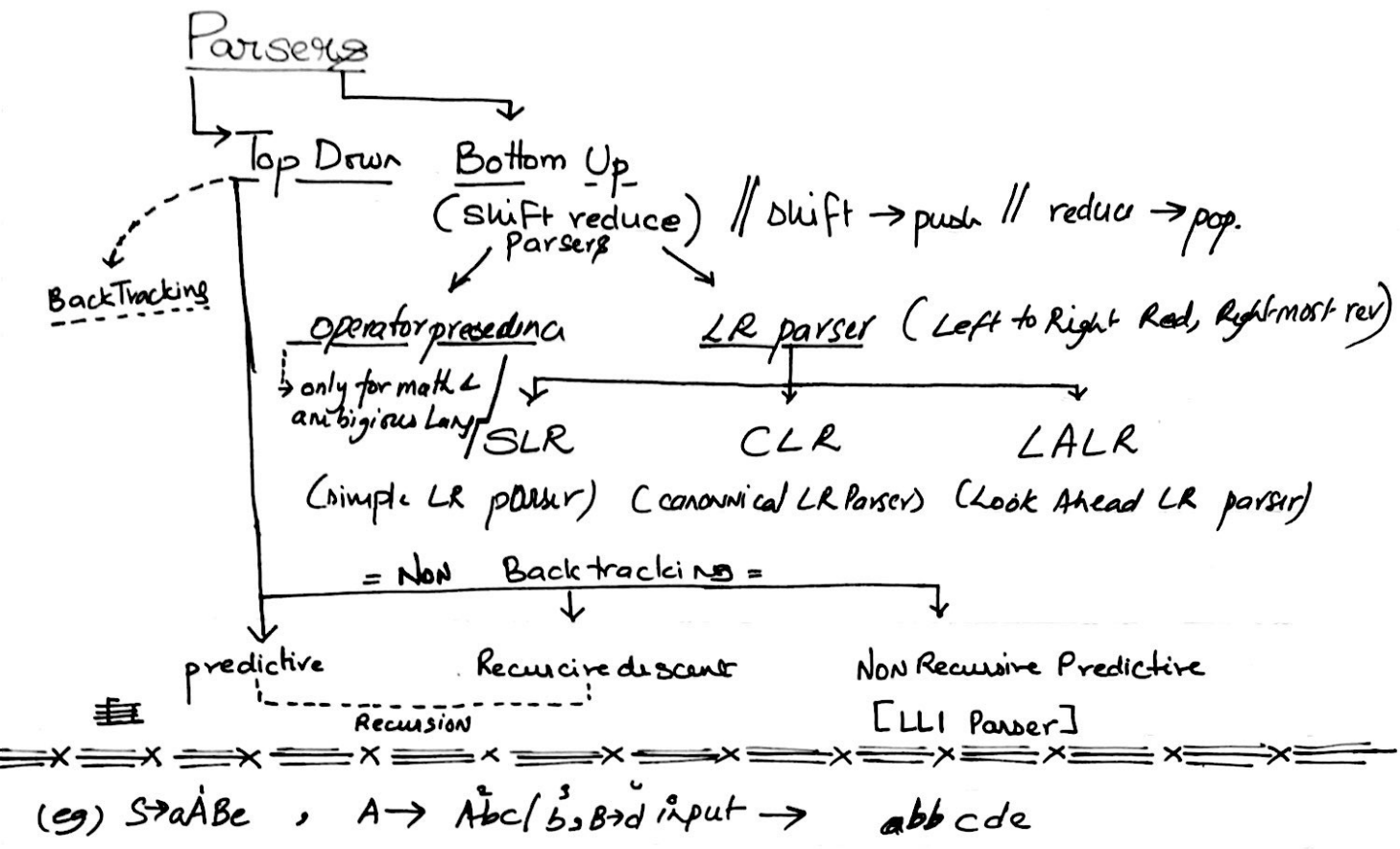
LR/NC/NE RR/~~NC/ε~~

1. Arrange NT as $A_1 \dots A_n$
2. for i = to n do begin
   - 2.1 for j = 1 to i-1 do begin
     - 2.1.x [ ]
   - 2.2 end
   - 2.3 eliminate the immediate LR among the A_i productions.

2.1.1 Replace each $A_i \to A_j\gamma$ by the Prod $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$ where $A_j \to \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are all current A_j prods.

eg) $S \to Aa \mid b$

$A \to Ac \mid Sd \mid \epsilon$ [ϵ doesn't introduce recursion]



(eg) $S \to aABe$, $A \to Abc / b$, $B \to d$ input → abbcde

Top Down (LL)

$S \overset{1}{\Rightarrow} aABe \overset{2}{\Rightarrow} aAbcBe$
$\overset{3}{\Rightarrow} abbcBe \overset{4}{\Rightarrow} abbcde$

Bottom Up (LR)

$S \overset{1}{\Rightarrow} aABe \overset{4}{\Rightarrow} aAbcde$
$\overset{2}{\Rightarrow} aAbcde \overset{3}{\Rightarrow} abbcde$

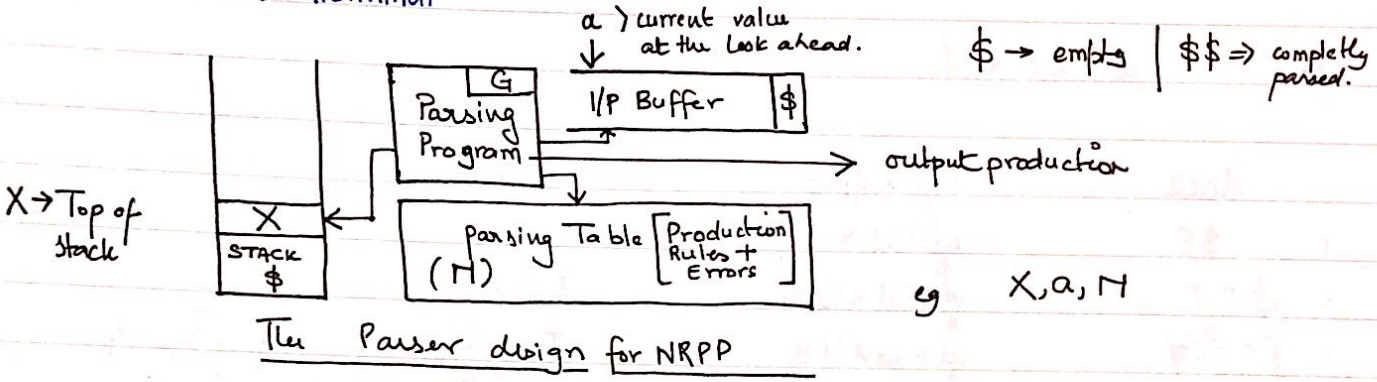
$abbcde \Rightarrow aAbcde \Rightarrow aAde \Rightarrow aABe \Rightarrow S$

31/1/18

Non Recursive Predictive Parser (LL(1) Parser)

→ Unambiguous, Deterministic, Right Recursive.

– For LL(1) Grammar –



The Parser design for NRPP

G:

$E \rightarrow TE'$

$E' \rightarrow +TE' \,/\, \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \,/\, \epsilon$

$F \rightarrow id \,/\, (E)$

M:	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ϕ	E'→ϵ
T	T→FT'			T→FT'		
T'		T'→ϵ	T'→*FT'		T'→ϵ	T'→ϵ
F	F→id			F→(E)		

T'→ϵ T'→*FT'

Rules for making NRPP

1. if $X = a = \$$ [X: Top of stack a: input]
   parser halts & announce success
2. if $X = a \neq \$$
   pop(X) and advance the i/p to next symbol // only time ip is moved
3. if $X = NT$
   consult $M[X, a]$

The entry will be X production of grammar / blank → error entry.

if the entry is blank, error recovery is invoked

if $M[X,a] = \{ X \rightarrow UVW \}$

Parser will replace X by W, V, U (U @ Top)

since ~~the~~ ~~and~~ parser has used some rule from the table.

← output the rule.

eg) $id + id * id$

	Stack	i/p buffer	o/p
1	$E	id + id * id $	—
2	$ E'T	id + id * id $	E → TE'
3	$ E'TF	id + id * id $	T → FT'
4	$ E'T'id	id + id * id $	F → id
5	$ E'	+ id * id $	T' → ϵ
6	$ E'T+	+ id * id $	E' → +TE'
7	$ E'T	id * id $	—
8	$ E'T'F	id * id $	T → FT'
9	$ E'T'id	id * id $	F → id
10	$ E'TF*	* id $	T' → *FT'
11	$ E'T'id	id $	F → id
12	$ E'T'	$	T' → ϵ
13	$ E'	$	E' → ϵ
14	$	$	—
→	Parsed		

The o/p:

~~E/TF/TT/E'/~~

Computation first and follow
Construction of table using
first and follow

$E \Rightarrow TE'$
$\Rightarrow FT'E'$
$\Rightarrow id\ T'E'$
$\Rightarrow idE'$
$\Rightarrow id + TE'$
$\Rightarrow id + FT'E'$
$\Rightarrow id + id\ T'E'$

id+id → FT'E'
id+id* idT'E'
id+id * id E'
id+id* id

$A \rightarrow Bxy \mid \alpha$

$B \rightarrow CD$

$C \rightarrow Ac$ $\quad (C \rightarrow A|c)$

$D \rightarrow d$

20/2/18

Apply the rules to calculate first of (X) until no more terminals or ϵ can be added to First (X) (First(α))

↳ can be anything

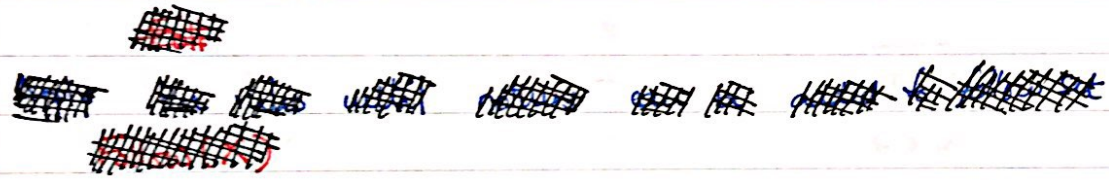
Better Method

Rule 1: if X is a terminal, then First (X) = {X}

Rule 2: if $X \overset{*}{\Rightarrow} \epsilon$ is a production, then add ϵ to first (X)

Rule 3: if X is a non-terminal and $X \to Y_1, Y_2 \ldots Y_k$ is a production then place 'a' in First(X) if for some i, a is in First(Y_i) and ϵ is in all of first (Y_1) to first(Y_{i-1}) i.e $Y_1, Y_2 \ldots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$

if ϵ is in First (Y_j) for all $j = 1, k$ then add ϵ to First(X)



- if α is any string then First(α) is the set of terminals that begins the strings derived from α
- if $\alpha \overset{*}{\Rightarrow} \epsilon$ then ϵ will also be in First(α).

↳ N.T only.

Follow (X)

① place $ in follow(S) where 'S' is in the start symbol and $ is the end marker

② If there is a production $A \to \alpha B \beta$ then everything in First(B) except ϵ is placed in Follow (B)

③ If there is a production $A \to \alpha B \beta$ or $A \to \alpha B$ where First(β) contains ϵ then everything in follow(A) is in follow(B).

Follow (X) for NON term X is the set of terminals that appear immediately to the right of X in some sentential form

Example

Ex 1

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id \mid (E)$

#

tx 2

$S \rightarrow ABCDE$

$A \rightarrow a \mid \epsilon$

$B \rightarrow b \mid \epsilon$

$C \rightarrow c$

$D \rightarrow d \mid \epsilon$

$E \rightarrow e \mid \epsilon$

=

① First (+) = {+}
First (*) = {*}
First (id) = {id}
First (() = {(}
First ()) = {)}

} first of all terminals.

→ =
First (S) = {a, b, c}
First (A) = {A, ε}
First (B) = {b, ε}
First (C) = {c}
First (D) = {d, ε}
First (E) = {e, ε}

② First (E') = ~~{ε}~~ {id, (}
First (E') = {+, ε}
First (T) = {id, (}
First (T') = {*, ε}
First (F) = {id, (}

Follow (E) = { $,) }

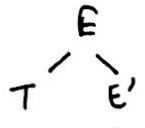
Follow (E') = { $,) }

Follow (T) = { +, $,) }

Follow (T') = { +, $,) }

Follow (F) = { *, +, $,) }

if nothing is following a symbol in RHS that doesn't mean $ will follow.



Follow (S) = { $ }

Follow (A) = { b, c }

Follow (B) = { c }

Follow (C) = { d, e, $ }

Follow (D) = { e, $ }

Follow (E) = { $ }

21/2/18

Construction of predictive parsing table

- Input grammar 'G', the output is 'M'

1. For each $A \to \alpha$ of G do 2., 3.
2. For each terminal 'a' in First(α) add $A \to \alpha$ to M[A, a]
3. if ϵ is in First(α), then add $A \to \alpha$ to M[A, b] for each terminal 'b' in follow(A)
   if ϵ is in First(α) and \$ is in follow(A) add $A \to \alpha$ to M[A, \$]
4. Make each undefined entry of M to be an error (blank)

$E \to TE'$
$E' \to +TE' / \epsilon$
$T \to FT'$
$F \to id / (E)$

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

	a	b	c	d	e	$
S	$S \to ABCDE$	$S \to ABCDE$	$S \to ABCDE$			
A	$A \to a$	$A \to \epsilon$	$A \to \epsilon$			
B		$B \to b$	$B \to \epsilon$			
C			$C \to c$			
D				$D \to d$	$D \to \epsilon$	$D \to \epsilon$
E					$E \to e$	$E \to \epsilon$

Example

[Mid: ~~Non~~ Predictive Parser (NRPP)
Top down complete C-recursive complete]

Ambiguity | PDA
UNIT - 1 ⊕ Test for LL1 ...
UNIT-2 (General theory)

≠ [[DFA ⊕ (minimise / solve)]
NFA → DFA (possible) !
[CFG ⊕ (PDA)]]

$S \rightarrow a\ BD\ h$
$B \rightarrow cC$
$C \rightarrow bC / \epsilon$
$D \rightarrow EF$
$E \rightarrow g / \epsilon$
$F \rightarrow f / \epsilon$

first(S) = { a }
First(B) = { c }
First(C) = { b, ε }
First(D) = { g, f, ε }
First(E) = { g, ε }
First(F) = { f, ε }

Follow(S) = { $, }
Follow(B) = { g, f, h }
follow(C) = { g, f, h }
follow(D) = { h }
follow(E) = { f, h }
follow(F) = { h }

	a	b	c	d	e	f	g
S							
B							
C							
D							
E							
F							

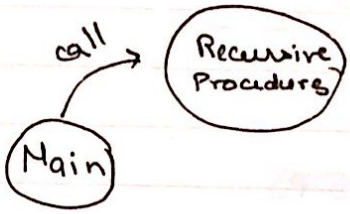
12-3-18

Recursive Descent Parser

- No backtracking.
- Top down.
- Use of recursive procedures for each N.T.
- Predictive.

$E \rightarrow idE'$

$E' \rightarrow *idE' \mid \epsilon$




```
main ()
{
        E();        //①
        if (l == $) //②
            print "success" //③
}
```

// l → look ahead pointer

```
E()
{  if (l == id)
   {
        Advance ('id');
        E'();
   }
}
```

```
E'()
{
    if (l == *)
    {
        Advance ('*');  ← Add if
        Advance ('id');
        E'();
    }
else
    return
```

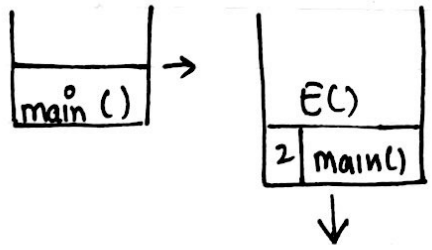
```
Advance (t)
{
    if (l==t)
        l = getchar();
    else
        print(error);
}
```

// calls the LA for the next token.

// t = expected input

Example

input : id * id $



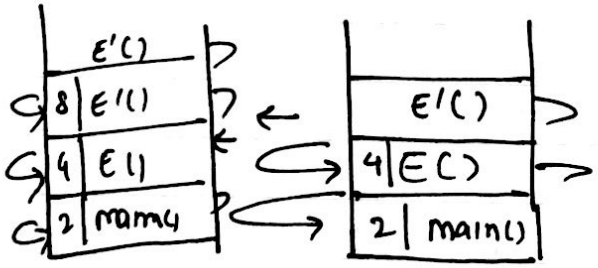
Advance

id * id $ ↑

→ id * id $ ↑

l = id

i = *



Advance

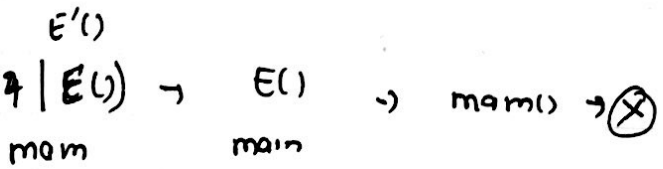
id * id $ ↑

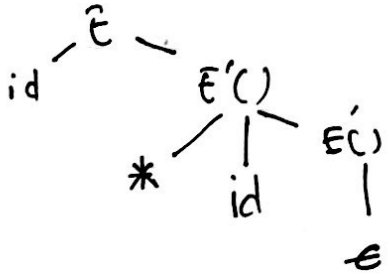
→ id * id $ ↑

Advance

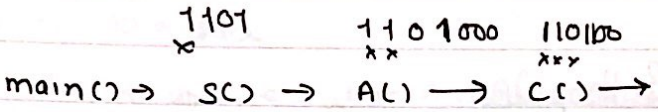
id * id $ ↑

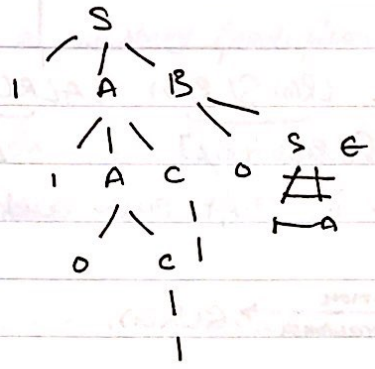
→ id * id $ ↑





1101





17 Error Recovery in LL(1) Parser. (Top Down Parser)

$E \rightarrow TE'$

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

$T \rightarrow FT'$

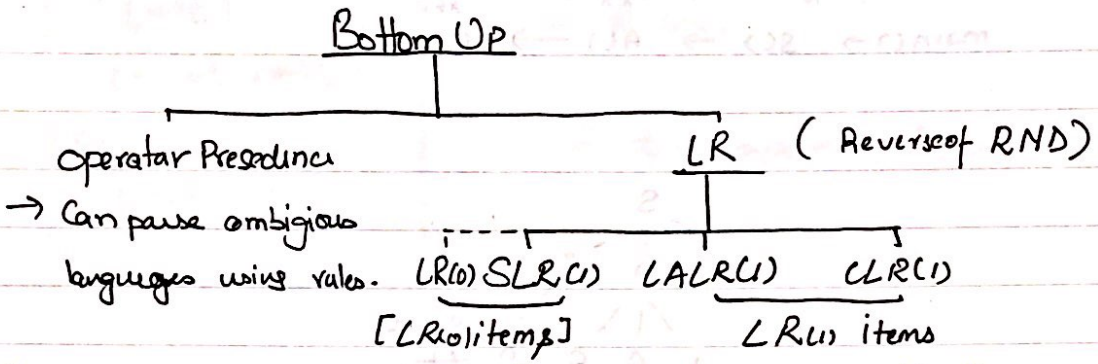
$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow id \mid (E)$

Eg) $S \rightarrow TA/PQ/\epsilon$

19/3/18

Bottom-Up Parsers



They're also called SR parser i.e Shift (Push) and reduce (Pop).

LR(0) —Remove Drawback→ SLR(1).
[No Need to study]

S → aABe
A → Abc
A → b
B → d

given | abbcde
aAbcde
aAde
aABe
S

// Handle
The substring which is chosen for reduction.

// RHS is reduced to LHS

Eg a b b c d e
A → b
⇒ a A b c d e
A → Abc
⇒ a A d e
B → d
⇒ a A B e
S → aABe
S

handle

// find handles @ every sentential form.

// The grammar must be unambigious.

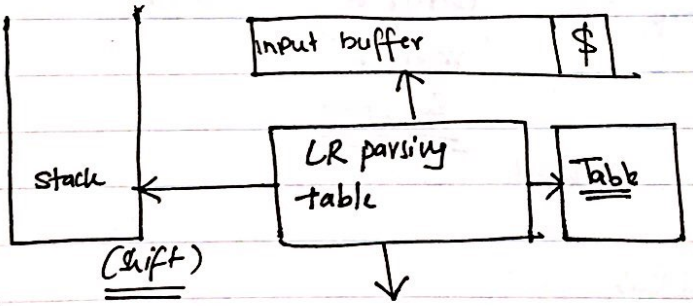
// ques of LR parser

1. When to reduce
2. What to use i.e what Reduction to do.

// handle a name given to a substring / production which will ultimately give S //

Construction is such that the handle will automatically be on top of the stack.

Stack implementation for LR parsers



Stack	action						Goto		
	id	+	×	(	)	$	E	T	F
0	s_5			s_4			1	2	3
1		s_6				Acc			
2		r_2	s_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	s_5			s_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	s_5		s_4					9	3
7	s_5		s_4						10
8		s_6			s_{11}				
9		r_1	s_7		r_1	r_1			
11		r_3	r_3		r_3	r_3			
12		r_5	r_5		r_5	r_5			

r_2 → reduce using 2.

s_5 → shift with 5 state on top.

move input buffer curr to top

$E \rightarrow E+T$ — (1)

$E \rightarrow T$ — (2)

$T \rightarrow T*F$ — (3)

$T \rightarrow F$ — (4)

$F \rightarrow (E)$ — (5)

$F \rightarrow id$ — (6)

Stack	Input	Output
0	id+id*id $	Shift
0id5	id+id*id $	reduce F→id
0F3	+id*id $	reduce using ④ T→F
0T2	+id*id $	reduce ② E→T
0E1	+id*id $	Shift
0E1+6	id*id $	Shift
0E1+6id5	*id $	reduce ⑥
0E1+6F3	*id $	reduce ④
0E1+6T9	*id $	Shift 7
0E1+6T9*7	id $	Shift
0E1+6T9*7id5	$	reduce 6
0E1+6T9*7F10	$	reduce ③ T→T*F
0E1+6T9	$	reduce ①
0E1	$	Accept

- Construction of SLR(1)
- Cannonical collection diagram.

2/4/18

Construction of SLR(1) parse table

Items & Closures

"Cannonical Collection of LR(0) items"

Items : A production having a '.' in the RHS of the production.
The dot tells what we have seen and what is left.

$S \longrightarrow \cdot AA$ [AN ITEM]

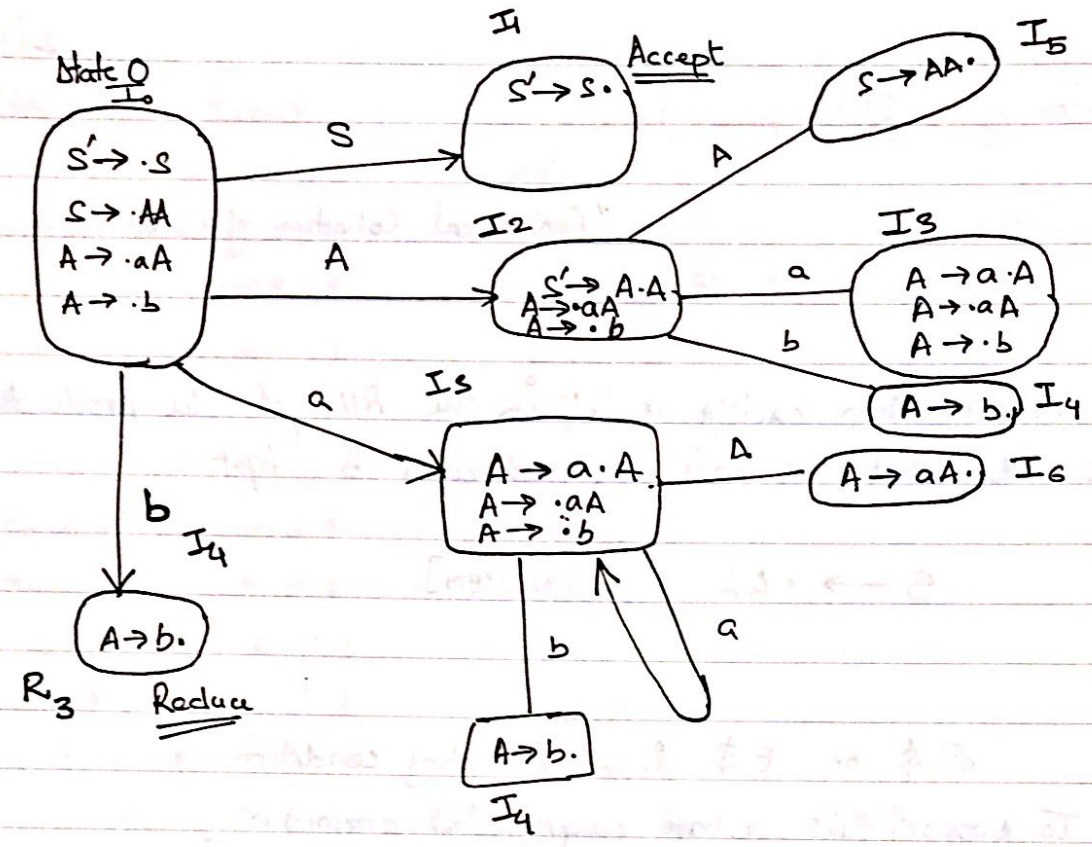
S$ or E$ is our accepting condition
To support this we have augumented grammar.
Gram mar + Start' { G + S' = Aug 1 }

$S \rightarrow AA$
$A \rightarrow aA \mid b$
] Make SLR(1) table

Closure If there is "." immediate left to N.T then add all prod of that N.T by putting a "." in the extreme of RHS of production

1. Make Aug grammar

$S' \rightarrow S$
① $S \rightarrow AA$ ③
② $A \rightarrow aA \mid b$
] Aug



Stat	Action			Goto	
	a	b	$	A	S
0	S3			2	1
1			Accept		
2	S3	S4		5	
3	S3	S4		6	
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

Filling Reduction Rules in SLR(1) table

Fill the reduce moves in follow of non terminal on LHS of the production used in reduction

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

productions with . in extreme Right are called final items.

The . in SLR(0) item means lookahead does not matter.

aabb$ check LR1 parser.

Stk		out
0	aabb$	
0 a 3		
0 a 3 a 3 b 4		

There are certain problems called conflicts i.e to shift / reduce (shift + reduce in same state) and reduce/reduce conflicts.

These are done using conflict resolution regimes.

" Construct SLR(1) for the grammar in Assignment 1 "

Unit : Memory Allocation (4)
Ch : Symbol Table Management
from Rajni Jindal Book

4/4/18

Construction of CLR(1) passe table

- Here we use a LR(1) item = LR(0) item + bot ahead.

$A \rightarrow \cdot Bb$, a/b → LR(1) item.

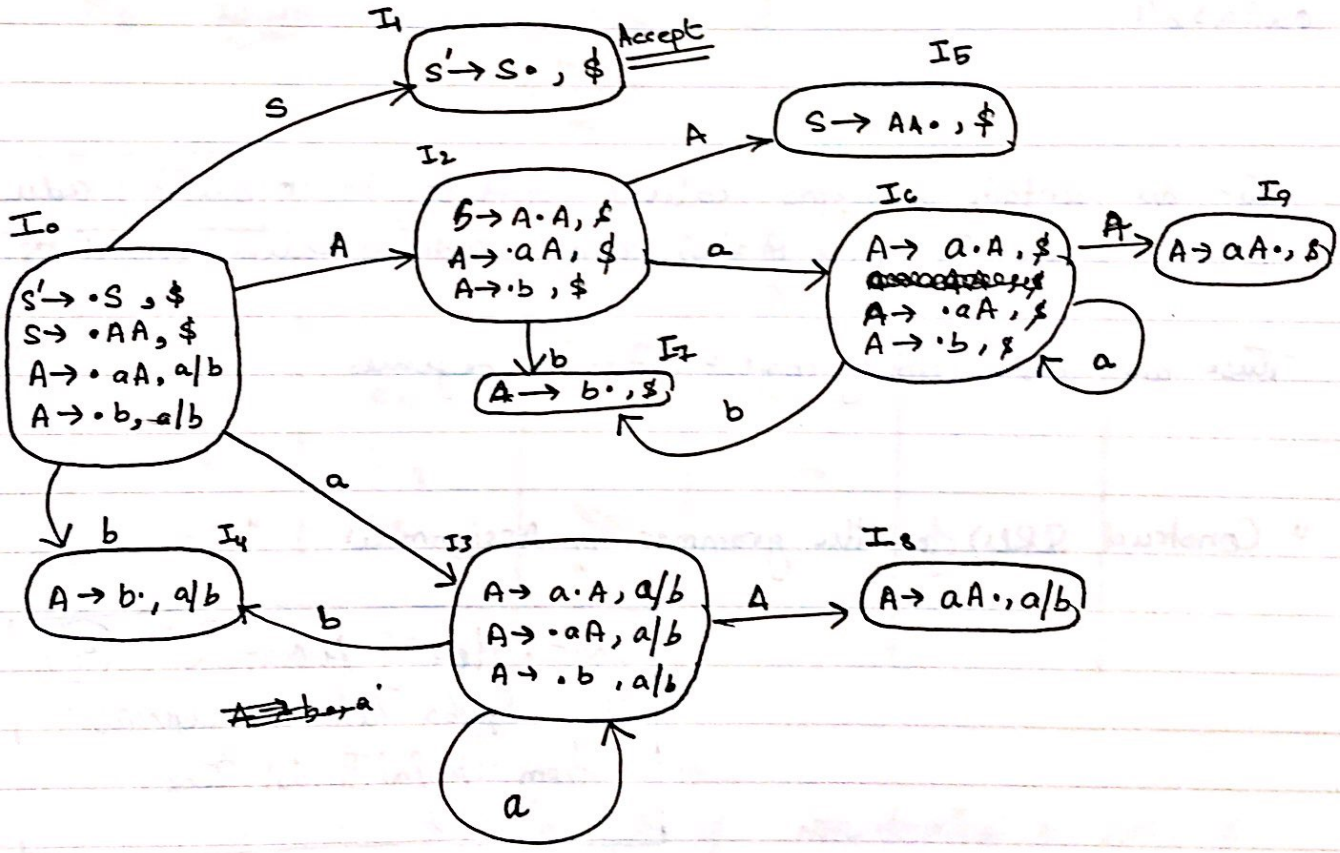
Reduction is based on the look-aheads.

$S' \rightarrow S$
$S \rightarrow AA$
$A \rightarrow aA$
$A \rightarrow b$

1. Look ahead does not change on transition between states.
2. If a look aheads, it is while computing closure.

if there is $A \rightarrow \alpha \cdot B\beta, a/b$ ~~B → ·γ, A~~ =

To find closure B, LC is first of RHS of B



~~A → b, a~~

State	Action a	Action b	$	A Goto	S Goto
0	S_3	S_4		2	1
1			Acc		
2	S_6	S_7		5	
3	S_3	S_4		8	
4	r_3	r_3			
5			r_4		
6	S_6	S_7		9	
7			r_3		
8	r_2	r_2			
9			r_2		

"There is a conflict in CLR(1) parser."

$A \to \epsilon$

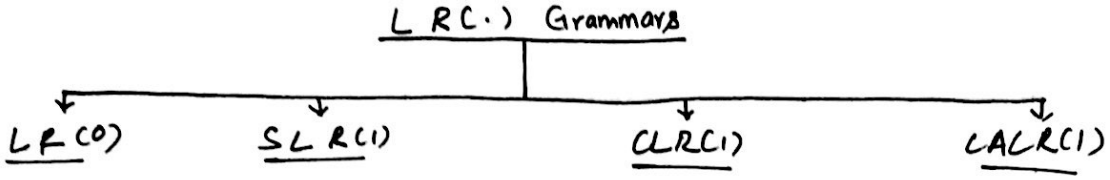
ikm = $A \to \cdot$

Construction of LALR(1) Parse Table.

1. combine state 6 & 3 & replace all instances with combined.
2. combine 4,7.
3. Combining criteria: if the LR(0) are same then combine.

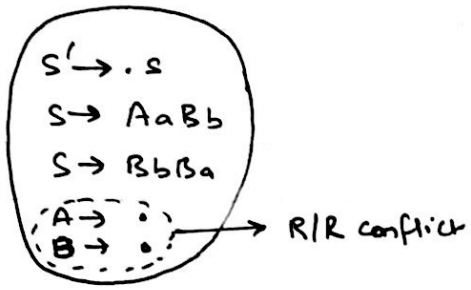
	a	b	$	A	S
0	S_{36}	S_{47}		2	1
1			Acc		
2	S_{36}	S_{47}		5	
3,6	S_{36}	S_{47}		8,9	
4,7	r_3	r_3	r_3		
5			r_1		
8,9	r_2	r_2	r_2		

// if there is a conflict, then the grammar is not that type.



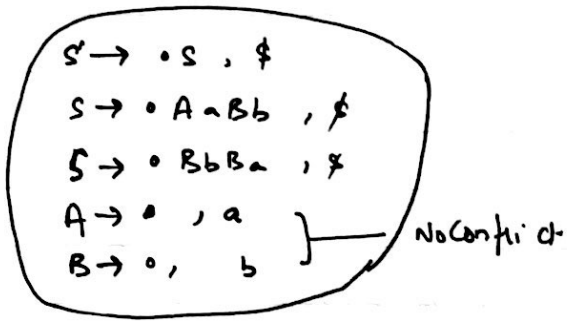
Eg> $S \rightarrow AaBb$
$S \rightarrow BbBa$
$A \rightarrow \in$
$B \rightarrow \in$

$I_0 =$



SLR(1)

I_0



CLR(1)

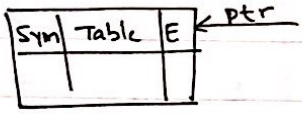
9/4/18

Semantic Analysis

- CFG
- Semantic Rules

For info which cannot be covered by CFG &. eg types, scopes.

"Attributes" each element has one.

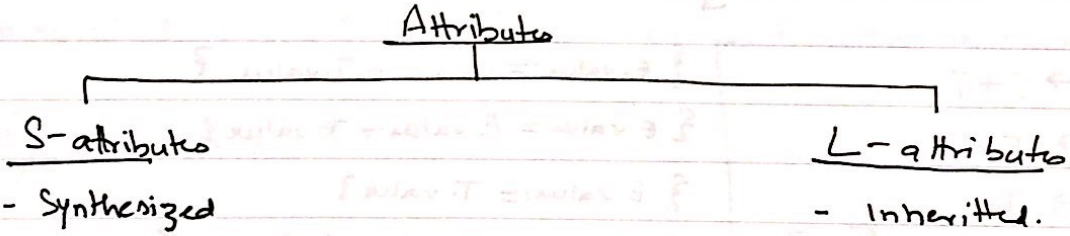


$id_1 = id_2 * id_3$

↳ <name, attr> eg <name, type, value>

– Attaching meaning to the productions.

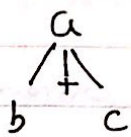
For attaching extra info with prog construct we associate semantic attr/ rules with grammar symbols/ production



With every symbol there is at least one attribute. But exception do exit depending on implementation

S-Attributes

The value of S attributes depends on the attributes of children



a depends (b, c)

* S-attributed defⁿ are evaluated bottom up.

L-attribute

The value depends on value of parent or left siblings.

→ can be done by both TD and BO parser

There are two notations for associating semantic rules.

Notation

SDD → Syntax Directed Defⁿ

SDT → Syntax directed Translation

Sem Anaⁿ is not an independent phase but happens with the parser.

S.D.D

Action

$E \rightarrow E+T$	{ E.value := E.value + T.value }
$E \rightarrow E-T$	{ E.value := E.value − T.value }
$E \rightarrow T$	{ E.value := T.value }
$T \rightarrow T/F$ [÷]	{ T.value := T.value ÷ F.value }
$T \rightarrow T*F$	{ T.value := T.value * F.value }
$T \rightarrow F$	{ T.value := F.value }
$F \rightarrow (E)$	{ F.value := E.value }
$F \rightarrow num$	{ F.value := num.value /or/ lexical value }

The syntax ⟶ Defines Action

we always write the action extreme right

S.D.D does not specify the time of execution of Action. It is only an abstract way to show action

S.D.T specifies the time of execution of instruction

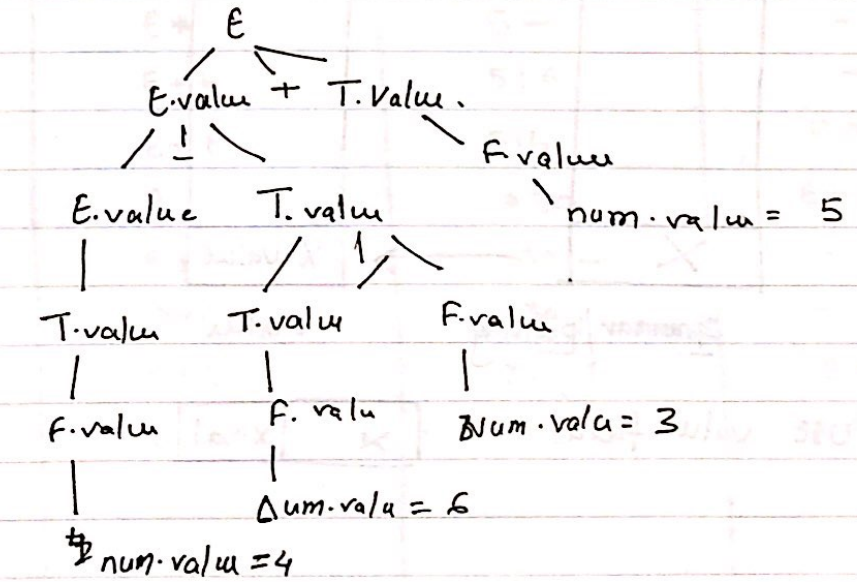
Grammar for Declarative statements

$D \rightarrow T$ list	{ list.type := T.type }	(inherited)	[Sibling]
list $\rightarrow$ list$_1$, id	{ list$_1$.type := list.type }	{ id.type = list.type }	
list $\rightarrow$ id.	{ id.type := list.type }	[parent]	
$T \rightarrow$ int	{ T.type = int }		
$T \rightarrow$ float	{ T.type := float }		

Every S-att difinition is always an L-attributed dif^n.

Annotated Parse Tree: Parse Tree but with actions.

Eg 4 − 6 / 3 + 5



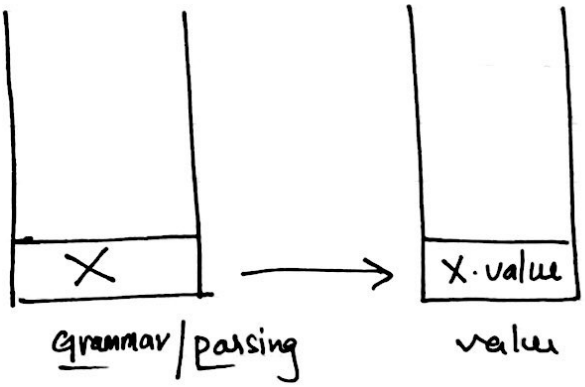
10/4/18

Semantic Analysis

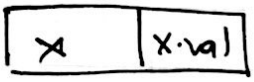
1. Attributes → Synth, Inherited
2. Definition → S-attributed, L-attributed
   - Implemented by:
     - S-attributed → Bottom-up parser
     - L-attributed → Bottom-up, Top-down
3. Notation → S.D.D, S.D.T

- S-attributed B-U- Parser

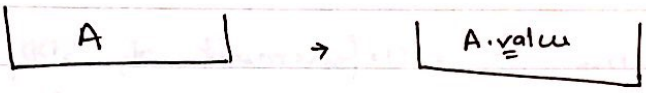
» USE TWO STACKS



» USE value field



After Reduction



Blank entries in value stack (due operators) are not errors

If we have K symbol on RHS of production Then p Top is (Top - K + 1)

$S \longrightarrow id := E$	{ value [n_top] := value [top] }
$E \rightarrow E_1 + E_2$	{ value [top'] = value [Top-2] + value [Top] }
$E \rightarrow E_1 * E_2$	{ value [top'] = value [top-2] + value [Top] }
$E \rightarrow - E_1$	{ value [top'] = -value [top] }
$E \rightarrow id$	{ ~~value [top] = value [id]~~ }

a=5 | b=6 | c=2

i/p	Parse	value	Reduc.
~~a*b+c~~	-	-	-
a*b+c	a	5	-
*b+c	E	5	E→id.
b+c	E*	5-	-
+c	E+b	5|6	-
+c	E+E	5|6	E→id
+c	E	30	E→E*E1
c	E+	30 ~	-
-	E+C	30 | 2	-
	E+E	30 | 2	E→id
	E	32	E→E+E
-	-	-	-

Translation Schemes (S.D. Translation)

A translation scheme is a refinement of SDD in which semantic actions are enclosed b/w braces & inserted within RHS of production to specify the exact time (point) when actions are to be executed.

Eg-

$E \to E + T$	{ Print(+) }
$E \to T$	{ }
$T \to id$	{ print(id.value) }

} S.D.D for Post fix

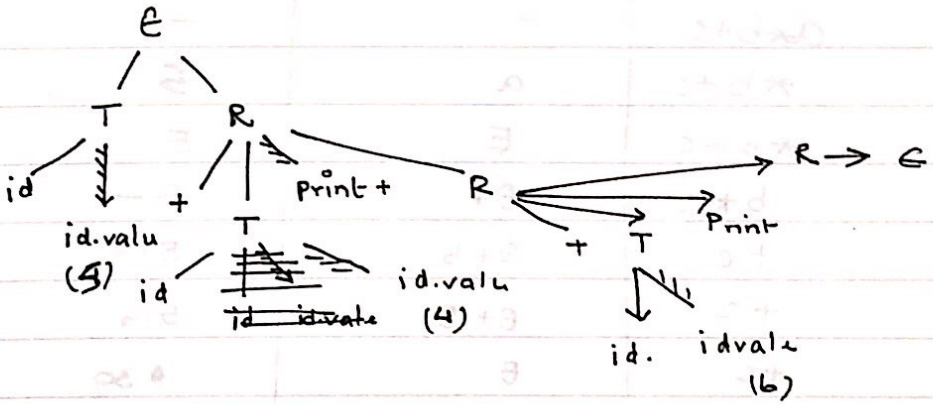
Algo generate IC

$E \to TR$

$R \to +T \{ print(+) \} R / \epsilon$

$T \to id \quad \{ id.value \}$

Eg: 5+4+6



5 4 + 6 +

Guidelines for writing translation Schemes Pg 136 Rajni

// Top-Down - L no longer with us · RIP //

11/4/18

Implementation of L-attributed with Bottom-Up

'Marker Non-Terminal' whenever there is an action [Inherit] b/w X_{k-1} and X_k we will put a Marker NT in the place of the action & add $M \rightarrow \epsilon$ { } M is marker.

$E \rightarrow X\ Y$

$Y \rightarrow +\ X$ { } $Y \mid \epsilon$

$X \rightarrow$ ~~id~~ { ~~print~~. print(value(num)) }

Changing to Marker augmented

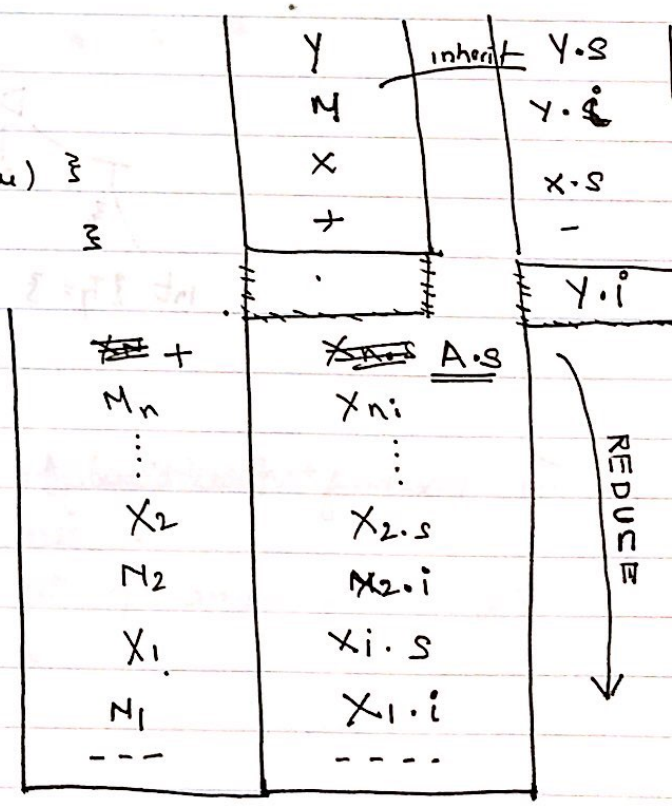
$E \rightarrow X\ Y$

$Y \rightarrow +\ X\ M_1\ Y \mid \epsilon$

$X \rightarrow id$ { print (id.value) }

$M_1 \rightarrow \epsilon$ { }

eg) $A \rightarrow M_1 X_1 \ldots M_n X_n$



Eg

$D \rightarrow T \{ L.in = T.type \} L$

$L \rightarrow \{ id.type := L.type \} id \{ L_1.in = L.in \} L_1$

$L \rightarrow \epsilon$

$T \rightarrow int \{ T.type = int \}$

Marking

$D \rightarrow T M_1 L$

$L \rightarrow M_2 id M_3 L_1$

$L \rightarrow \epsilon$

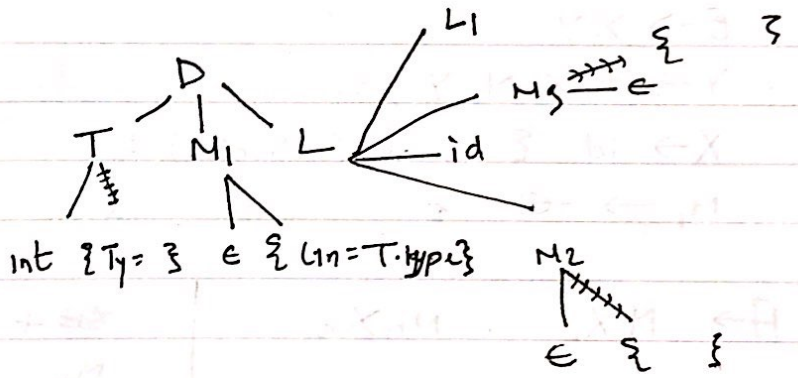
$T \rightarrow int$ ~~M4~~

$M_1 \rightarrow \epsilon \{ L.in = T.type \}$

$M_2 \rightarrow \epsilon \{ id.Type := L.Type \}$

$M_3 \rightarrow \epsilon \{ L_1.in = L.in \}$

$M_4 \rightarrow \epsilon \{ T.type = int \}$ [Not Needed]



= Type checking [self Study]

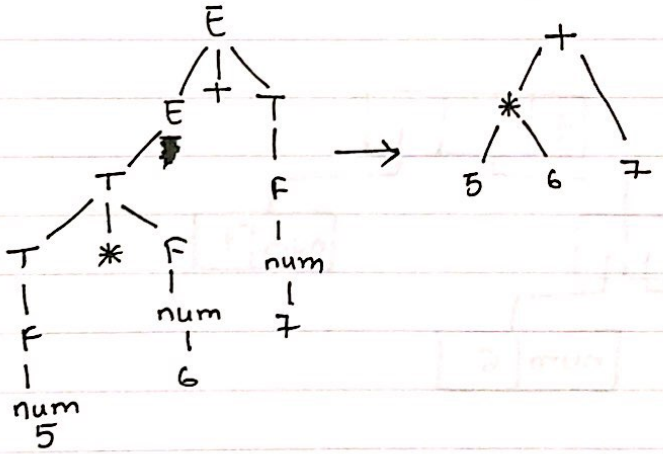
16/9/18

Intermediate code generation (SDD)

- Representation of IC
  - Postfix
  - Syntax tree
  - 3 address code
    - Triple
      - direct triple.
      - indirect triple.
    - Quadruple.

Syntax tree is a condensed form of parse tree, operators and keyword @ interior and do not appear as leaf.

5*6+7

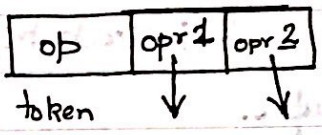


Construction of syntax tree

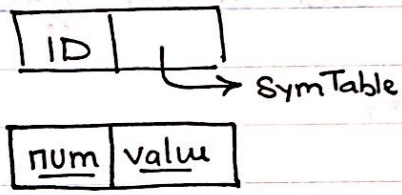
1. Create a leaf for all ids and consts.
2. Creat subtrees for subexpression
3. Op one @ root of subtrees with operands and children
   children = id/ Num/ subtree.

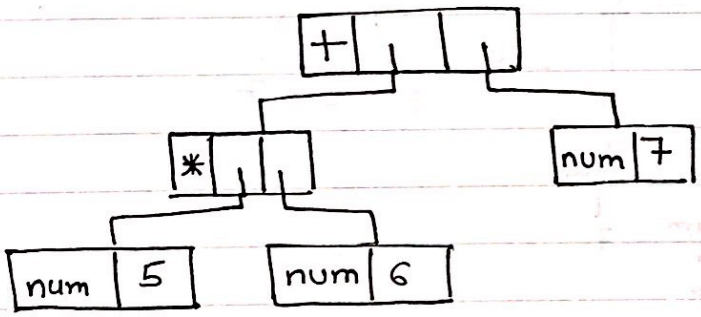
We make use of structures for nodes $\in \{$ id, Num, op $\}$

Node corresponding to operator



Node corresponding to ID





A function creating this node is added as an action to the grammar

1. C idLeaf (id, spt)
2. C numleaf (num, value)
3. C op Node (op, opr_1, opr_2)

S attributed definition for making syntax tree

$E \to E+T$	{ E.np := CopNode (+, E.np, T.np) }
$E \to T$	{ E.np := T.np }
$T \to T*F$	{ T.np := CopNode (*, T.np, F.np) }
$T \to F$	{ T.np := F.np }
$F \to num$	{ F.np := CnumLeaf (num, value) }
$S \to id := E$	{ id.np = CidLeaf (id, Sstr) S := CopNode (:=, id.np, E.np) }

np = new pointer

3 Address Code

Every statement is written comprising of maximum 3 addresses.

$a+b*9$

$t_1 = b*9$

$t_2 = a+t_1$

Pg 158 [list of operators]

relop (a,b) goto L_1 (if relop(a,b) is true goto L or goto Next-stmt

Triple it has 3 parts

op	opr1	opr2

for each expr we have a triple, and all these are in a triple table, this is the Intermediate Code.

$x := a+b*c + c*d*e$

Table

Trno	op	opr1	opr2
(0)	*	b	c
(1)	*	c	d
(2)	*	(1)	e
(3)	(+)	a	(0)
(4)	(+)	(3)	(2)
5	:=	x	(4)

- helps to find common sequences
- This structure is inflexible esp with loops

To add code movement capabilities we add another columns.

~~Entry~~	op	opr1	opr2	res
	*	b	c	t_1
	*	c	d	t_2
	*	t_2	e	t_3
	+	t_1	a	t_4
	+	t_4	t_3	t_5
	:=	t_5	-	-

Generally quadruple allows for better optimization, this is done by making a list of temporaries.

- Indirect and Direct

* SDD for 3add code.

SDD for generating 3address code

17/4/18

$S \rightarrow id := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow -E_1$

$E \rightarrow id.$

E.place = E.name = E.value (all the same.)

functions : gen() ; gentemp() ; lookup() ; E.name

gen(t_1 := 'E.val' + 'E.val') → 3add code.

gentemp() → generates a temporary

lookup() → check if id is in symbol table or not. for error handling

E.name → synth attr of E.

lookup (id.entry). { value → found entry. / Nil → Error

$S \rightarrow id := E$ { gen (id.name ':=' E.name) }

$E \rightarrow E_1 + E_2$ { E = gentemp() gen (E.name ':=' E1name '+' E2.name) }

$E \rightarrow E_1 * E_2$ { E = gentemp() gen = (E.name ':=' E1name '*' E2.name }

$E \Rightarrow - E_1$ { E.name = genTemp() gen (E.name ':=' '-') E1.name }

$E \Rightarrow id$ { lookup(id.entry) E.name = id.name }.

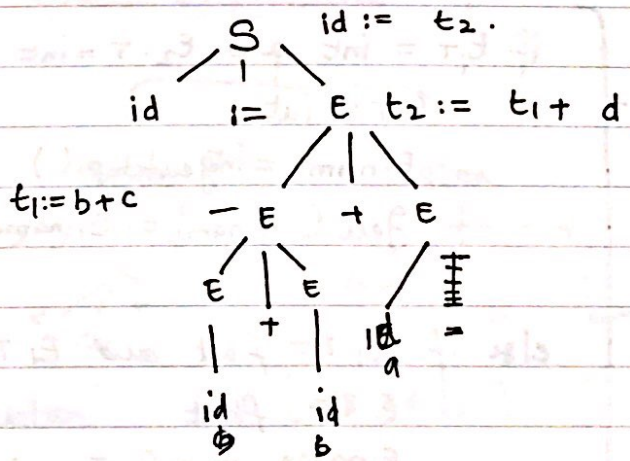
Example

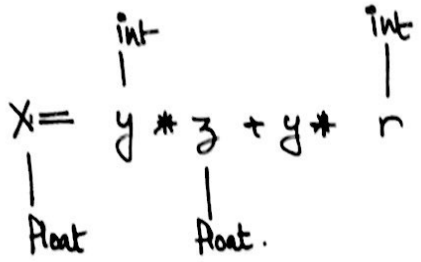
$a := b+c+d$

$t_1 = b+c$

$t_2 = t_1 + d$

$a = t_2$





$t_1 := \text{int2float}\ y$

$t_2 := t_1 *_{float} 3$

~~$t_3 := \text{int2float}\ r$~~

$t_4 := y *_{int} r$

$t_4 := \text{int2float}\ (t_3)$

$t_5 := t_4.\text{float} +_{float} t_2$

$x := t_5$

Adding Type

$E.T \rightarrow$ Type.

$E \rightarrow E_1 + E_2$

if $E_1.T$ = int and $E_2.T$ = int :

$E.T$ = int

$E.name$ = intgentemp ( )

gen ($E.name$ ':=' $E_1.name$ '+int' $E_2.name$)

else if $E_1.T$ = float and $E_2.T$ = float

$E.T$ = float

$E.name$ = floatGenTemp()

gen($E.name$ ':=' $E_1.name$ 'float+' $E_2.name$)

else if $E_1.T$ = float and $E_2.T$ int

~~$t.name$~~ $E.name$ = float Gentemp()

gen ($t.name$ ':=' 'int-to-float' $E.name$)

[S] → ⌜ ——— ⌟

else if $E_1.T = int$ and $E_2.T = float$.

[else : t.error]

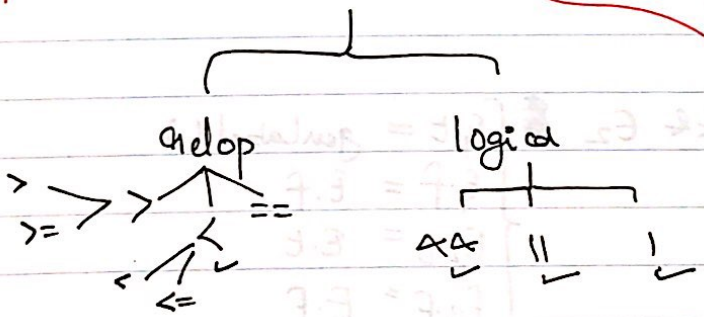
for Ha := gen (id.name ':=' t.name) // only when id and temp match.

Note

'Back Patching' = 'SDD for boolean (numerical) Exp'

20/4/18

SDD for Boolean expressions (using control flow)



'Partial Evaluation' [a && b] if a is false, jump out

⌜ 'if E else E' ⌟ eg if a ① relop ② b goto L_1
else goto L_x

It has two exits, true / False.

Each sym node has a true/ false attribute for T/F.

here we use `genLabel()` generates a new label.

&& & a<b && c<d

```
if a < b goto 4
else goto E.false

4: if c < d goto E.true.
   else goto E.false.
```

E.code = 3 addr code

$E \rightarrow E_1 \&\& E_2$

$E \rightarrow E_1 \text{ relop } E_2$.

1. $E \rightarrow E_1 \&\& E_2$

$E_1.t = \text{genLabel}()$
$E_1.f = E.f$
$E_2.t = E.t$
$E_2.f = E.f$
$E.code = E_1.code \,||\, \text{gen}(E_1.t\ ':') \,||\, E_2.code$

$E \rightarrow E_1.\text{relop}\, E_2$

$E.code := \text{gen}(\text{'if'}\ E.name\ \text{"relop"}\ E_2.name\ \text{'goto'}\ E.t) \,||\, \text{gen}(\text{'goto'}\ E.f)$

" Support to program for execution : runtimeEnvt "

$E \rightarrow E_1 \,||\, E_2$ = { $E_1.t = E.t$

$E_1.F = gen(label())$

$E_2.true = E.t$

$E_2.F = E.f$

$E.code = E_1.code \,||\, gen(E_1.t':') \,||\, E_2.code$

" Code optimization : defn and Examples [~14-15]

— Redundant LD and ST operations — ✓ ✓

— Dead code elimination " — ✓ ✓

— Const propagation. — ✓ ✓