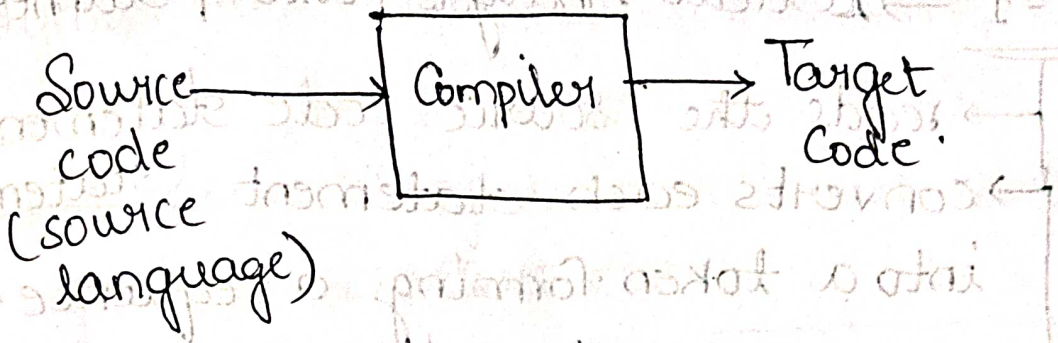
05/02/2024

→ Compiler converts high-level language to low language.

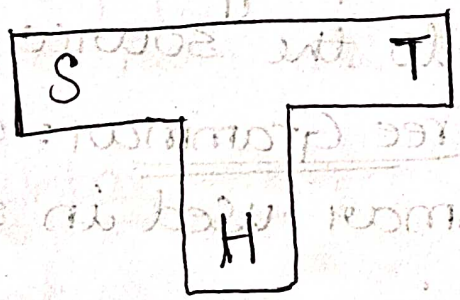


→ High-level language: human readable.

→ low-level language: machine understandable.

→ opcode: combination of 0's & 1's.

→ Represented as:



S → Source Code

T → target code

H → host language.

→ Compilers (C): TurboC, gcc

→ Host language for C compiler is C language.

→ Using the same host language as the programm language for a compiler is called Bootstrapp

→ Most of compilers are implemented using C.

→ Pass: group of phases.

→ Phase: consists of a specified functionality which takes a I/P and gives a o/P.

→ O/P of final phase is the target code.

* Basic phases of a compiler:

Phase-1 → Lexical Analysis (Lexer / Scanner)

- → reads the source code statement by statement
- → converts each statement into a token forming a sequence.
- → removes all the extra spaces and new line characters.
- → removes the comments.

token: <name, loc> (name: mandatory, loc: optional)

Phase-2: → Syntax Analysis (Parser)

- → checks if the program is correct according to the source language.
- → Context Free Grammar: Specifies the grammar used in source language.
- → O/P of this phase is a syntax or parse tree.

06/02/2024

Phase-1: Lexical Analysis

- → If it is a reserved word then token consists only <name>.
- → If it is not a reserved word then token consists <name, loc>

Eg: variables / identifiers.

Eg: int a;

<int> <ID, 1> <;>

→ Symbol table is the data structure where we store the pre-defined symbols.

* We store all symbols, reserved words, special characters etc.

→ If the symbol is not present in the symbol table, we create a token and update the symbol table. No repetition is allowed.

Name	Type	descri
int	Keyword	integer

Phase-2: Syntax Analysis (Parser)

* all terminal symbols are given by operators

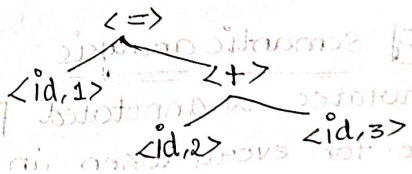
Ex: a = b + c

Phase-1: <id, 1><=><id, 2><+><id, 3><;>

⇒ We draw the parse tree to these tokens.

⇒ Types: bottom-up; top-down

⇒ Leaf nodes are variables and operators would be the internal nodes.



Phase-3: Semantic Analysis:

- Meaning of the statement.
- Checks if the given tokens generate meaningful statements.
- for this it checks the type of identifier and verifies if further operations are meaningful (or) not.
- performs type conversion for the required types to be modified.
- Strongly typed languages: The type of the variable is explicitly declared.

  Eg: C, C++, Java.

  Interview questions

  Weakly typed languages: Type of the variable is known while usage in the code.

  Eg: Python.

Explicit type conversion: 'C'

↳ O/P of semantic analysis generates unannotated parse tree.
i.e. for every token in parse tree we add a "type" tag.

Phase-4: Intermediate Code Generation:

↳ Eg: source code ⇒ high-level language ; target code ⇒ low level language.

→ intermediate code ⇒ 3-address code

$c = a + b * d;$

⇒ 3-address code:

$t_1 = b * d$
$t_2 = a + t_1$
$c = t_2$

```
LD R0, b
LD R1, d
MUL R1, R0
LD R2, a
ADD R1, R2
ST c, R2
```

3-address code ⇓ to the right side of the assignment operator; we have atmost 1 operator and atmost 2 operands.

3-address code helps in generating the assembly language code easily.

Phase-5: Code Generation: Target Code.

Phase-6: Code Optimization (optional)

↳ Checks to reduce the size or time of execution of the code.
↳ Machine dependent code optimization.

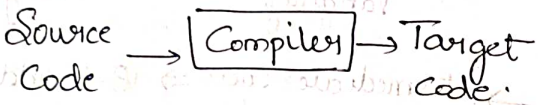
Phase - 1, 2, 3 : Analysis Phases.
Phase - 4, 5 : Synthetic Phases.

10/02/2024

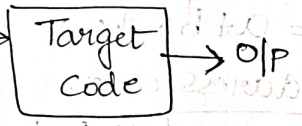
UNIT-1

* Language Processors:

→ Compiler:



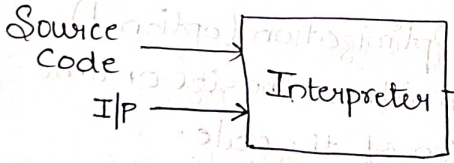
* Target Code is executable.



→ Compiler is machine dependent; we must execute the executable code on the same machine.

→ Interpreter:

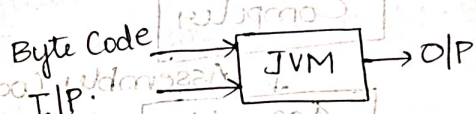
* Converts each statement of the code one at a time to target code.



* Compiler is faster than interpreter due to complete source code being converted to target code.

** Portability: The target code/byte code which is executable on any other machine; is called language portability.

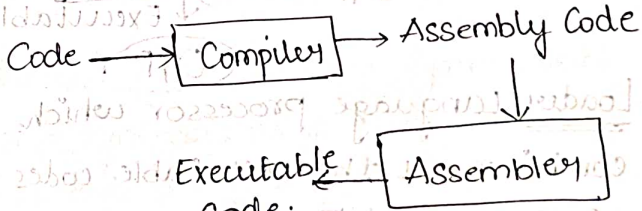
Ex: Java (runs on any system having JVM).



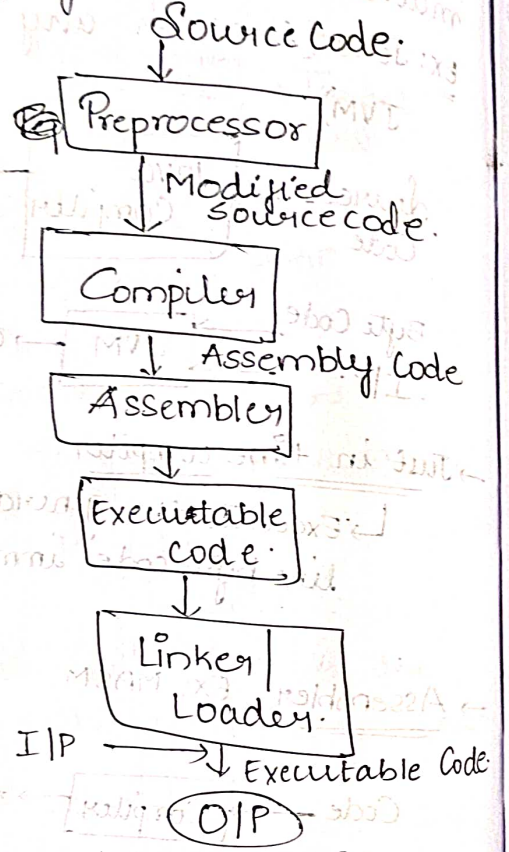
→ Just in time compiler:

↳ Executes the generated single line byte code immediately.

→ Assembler: Ex: MASM



→ Macros are replaced into the body of the code by the process of unwrapping



→ Loader: Language processor which combines all the executable codes and gives a complete code.

→ Linker: used to link executable codes which

* Structure of the compiler:

→ 2 parts → Analysis: Analyse each part of source
→ Intermediate represent code.
→ Synthesis: Code and O/P

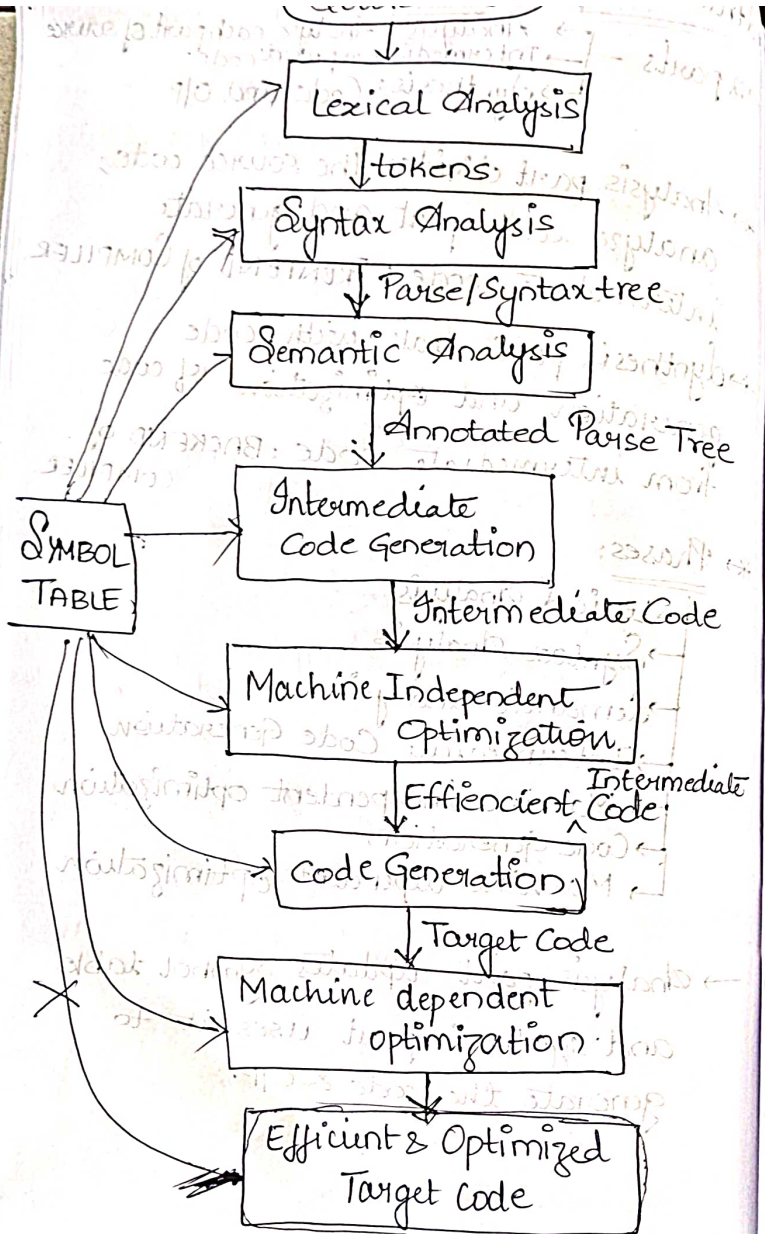
*→ Analysis part divides the source code; analyze each part and generate intermediate code : FRONTEND of COMPILER

*→ Synthesis part deals with code generation and optimization of code from intermediate code : BACKEND of COMPILER

*→ Phases:

→ Lexical Analysis
→ Syntax Analysis
→ Semantic analysis
→ Intermediate Code Generation
→ Machine independent optimization
→ Code Generation
→ Machine dependent optimization

→ Analysis part updates symbol table and synthesis part uses it to generate the code & O/P.



* Ex:

position = initial + rate * 60

① Lexical Analysis:

→ Converts into legal set of lexemes.

→ lexeme: set of symbols that can match with one of pre-defined pattern

→ Lexemes: position, =, initial, +, rate, *, 60

→ format of token: <token name, attribute value>

token name → abstract name

attribute value → pointer to symbol table where the symbol is given.

Ex: identifier: name of token = ID

number: name of token = num

→ Symbol table:

attribute value (pointer) →

Name	type	scope	default

Ex:

→ <ID,1> <=> <ID,2> <+> <ID,3> <*> <60>

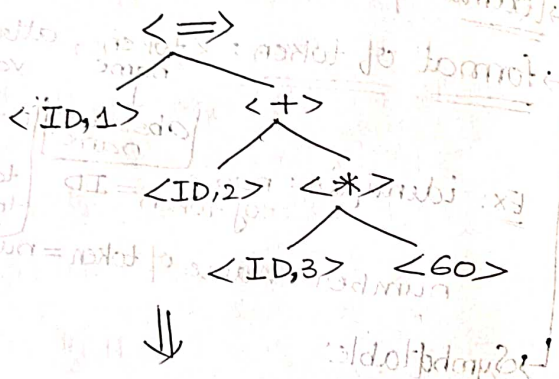
② ~~Syntax Analysis~~:

→ If any ~~one~~ of the lexeme is not matched to the given patterns is called lexical error.

② Syntax Analysis:

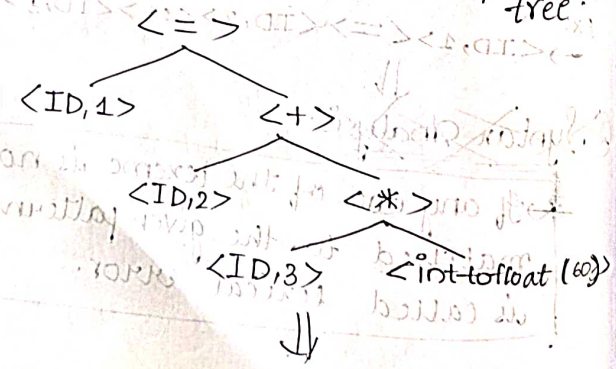
Errors detected here are called syntax errors.

→ If no error; syntax tree is generated.



⇓

③ Semantic Analysis: Annotated parse tree:



⇓

④ Intermediate Code Generation:

3-address code. $(a = b + c)$

$t_1 = \text{int-tofloat}(60)$

$t_2 = \text{rate} * t_1$ | $t_2 = \langle id,3 \rangle * t_1$

$t_3 = \text{initial} + t_2$ | $t_3 = \langle id,2 \rangle + t_2$

$\langle id,1 \rangle = t_3$

⇓

⑤ MIO:

$t_1 = \langle id,3 \rangle * 60.0$

$\langle id,1 \rangle = \langle id,2 \rangle + t_1$

⑥ Code Generation:

```
MOVF R0, #60.0
MULF R0, R0, ID3
ADDF R0, R0, ID2
STF  ID1, R0
```

12/02/21

→ Target code is machine dependent as we convert intermediatary code (machine independent as they are just repetition of source code) to target code which is a set of micro instructions. As micro instructions contains register and unique instructions set that depends on your machine (risk/cisk). Hence target code is machine dependent.

* SYMBOL TABLE:
We store info regarding constants & identifiers.

* Compiler design tools:
→ Compiler is a software.
→ Lex: It is a tool that automatically generates/implements lexical analyser.
→ YACC/parser generator: automatic parse tree generation with taking input as grammar.

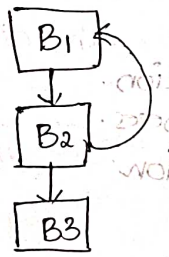
YACC: Yet Another Compiler Compiler.

→ Syntax directed translation engine: Annotated parse tree generation.

→ Data flow analysis engine: Tracks flow of data in 3-address code.

flow → data flow
flow → contrd flow

→ 3-address code is divided into blocks where each block has statements that executes only sequentially.



→ code generator: helps to efficiently use register.

*Applications of Compiler Technology. 13/02/24

*→ register: variable is allocated to the register.

→ Implementation of High-level programming languages:

* Improve the efficiency of generated code:
* using registers.

→ optimization of computer architectures.
  └→ Parallelism
    ├→ Instructor level
    └→ Processor level.

→ Program Translation:
  ├→ Compiled simulation.
  ├→ Hardware syntheses.
  └→ Binary Translation.

Programming language basics

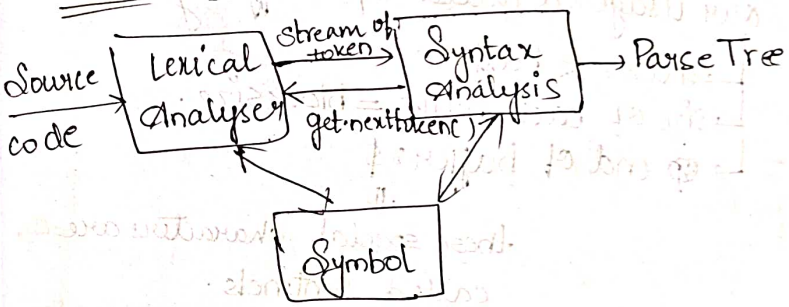
→ Static | dynamic distinction.

→ Aliasing
→ Parameter passing
→ Dynamic Scope
→ Explicit Access Control
→ Static scope and block structure.

→ Environment & states.

13/02/2024

→ Lexical Analysis:



*Lexical error:

→ If the lexeme is not matched with any one of the pattern, such error is called lexical error.

→ Error recovery takes place in a compiler whenever an error is occurred, then it overcomes the error and compiles the other statements.

- Panic Mode Error Recovery:

1) Skip the symbols unless the lexeme is matched with any one of the pattern.
2) Transpose the symbols.

* The process of moving the data from main memory to the buffer and scanning for our usage is called input buffering.
  - uses 2 buffers.
  - size of each buffer = blocksize.
  - end of buffer: $

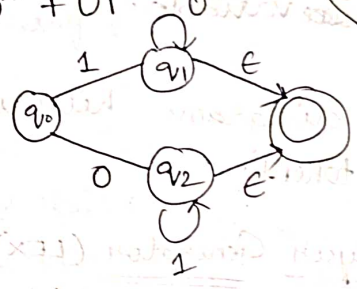
  ⇓ these special characters are called sentinels.

1, 0, 10, 01

→ Specification of patterns/tokens
  - Regular Expressions.
  - Representation of regular languages.
  - Operations:
    - Union $L_1 \cup L_2$
    - Intersection $L_1 \cap L_2$
    - Concatenation $L_1 L_2$
    - closure: L_1^* ; L_1^+
    - complement $\overline{L_1}$
    - Reverse L_1^R

Ex: 1000110
0110 → substring
0010 → subsequence.

Ex: $10^* + 01^*$



letter → [A-Z a-z]
digit → [0-9]

Regular definitions

[letter] [letter|digit]

⇒ [l][l|d]*

17/02/24

→ Regular expressions are used to recognise tokens.

* The final state of the automata drawn to recognise the given regular expression; gives the token to be sent to the parser.

→ Predefined variables:

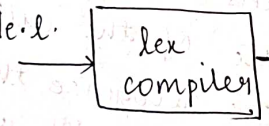
yytext: current lexeme scanned by the lexical analyser.
↳ character variable

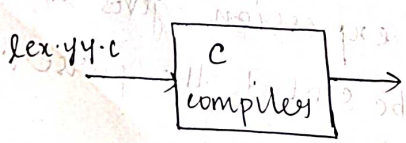
yylval: holds the attribute value (of token) which is passed to parser
↳ integer variable

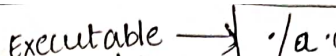
→ Transition diagrams helps us to recognise tokens.

* Lexical Analyser Generator (LEX):

→ programming language; which uses C and lexical specifications to write the code.
→ .l is the extension.







* LEX specifications:

Declaration section → Regular definitions & variables are declared.
%%
Transition Rules → Set of patterns & corresponding actions
%%
Auxiliary functions.

EX:

```
%{
#include<stdio.h>
#define LT, GT, EQ, IF, THEN, ID, NUM
%}
```

These are copied into C program as it is.

In Lex
. → any character
\. → helps in generating .

```
delim   [\n\t]
ws      {delim}+
digit   [0-9]
digits  {digit}+[.]?[E|e]?[+|-]?
```

digits {digit}{digit}* [.]? {digit}* [E|e]? [+|-]? {digit}

```
letter  [a-zA-Z]
ID      {letter}[{letter}{digit}]*
```

{ws} { no action to be performed }

IF { Return (IF); A2; A3 - - - }

n no. of actions to the parser.

THEN { Return (THEN); }

< { Return (LT); }

> { Return (GT); }

== { Return (EQ); }

yylval=

{ID} { Return (ID); InstallID(); }

yylval=

{digits} { Return (NUM); InstallNUM(); }

} any no. of patterns can be written.

%%

int main()

{ yylex(); → func where complete lexical analysis is stored

}

int InstallID() { } → Return (pointer to Symbol table where the value is stored)

int InstallNUM() { } → Return (value of the number)

17/02/202

* Syntax Analysis: (Parser)

→ Stream of tokens & grammar are the I/P.

→ CFG is used $G = (V, T, P, S)$ (V+T)

P: E → E+T | T

T → T* F | F

F → (E) | id

head of the production (only one variable)

body of the production

V = {E, T, F}

T = {id, *, +, (,)}

total no. of productions = 6

order of production.

$S = E$

→ Issues in CFG:

- Left recursion: $A \to A\alpha \mid \beta \mid a$ — Correction: $A \to \beta A' \mid \epsilon$, $A' \to \alpha A' \mid \epsilon$
- Left factoring: $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \dots \mid \alpha\beta_n$
- Ambiguous: $(A \to \alpha A(1) \mid \beta)$, $A' \to \beta_1 \mid \beta_2 \mid \dots \beta_n)$

* Few algorithms do not produce efficient parsers.

∵ Given productions have left recursions:

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

Ex: $S \to iEtS \mid iEtSeS \mid a$

$S \to ietSS' \mid a$

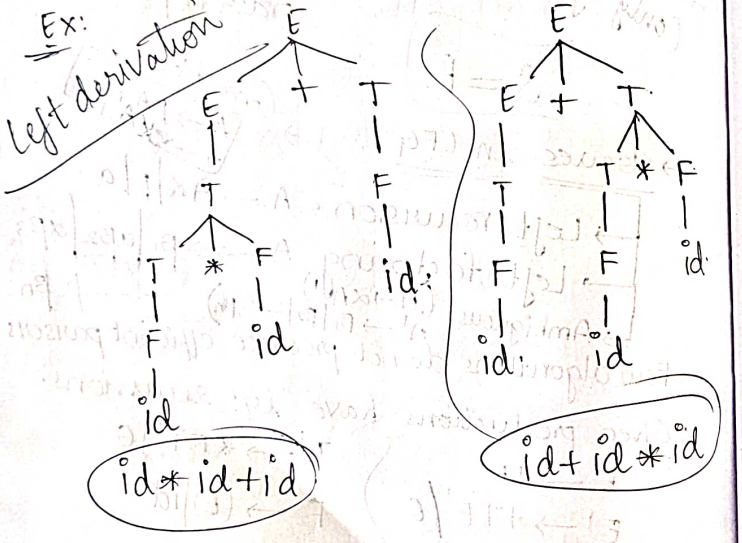
$S' \to eS \mid \epsilon$

* Derivation
  - Left most derivation
  - Right most derivation
  - ↳ deriving strings from the given productions

→ In any step, the replacing variable is called handle.

Ex:

Left derivation

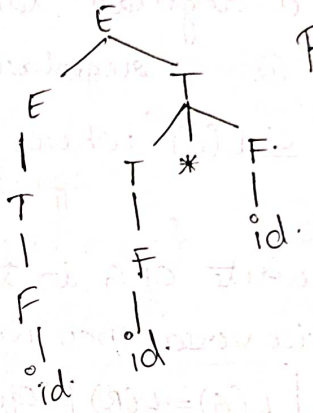


Parse Tree

$E \Rightarrow E+T \Rightarrow E = E+T*F \Rightarrow E+F*id$

$\Rightarrow E = E+id*id \Rightarrow E = T+id*id$

$\Rightarrow E = F+id*id \Rightarrow E = id+id*id$

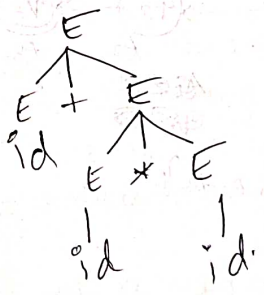


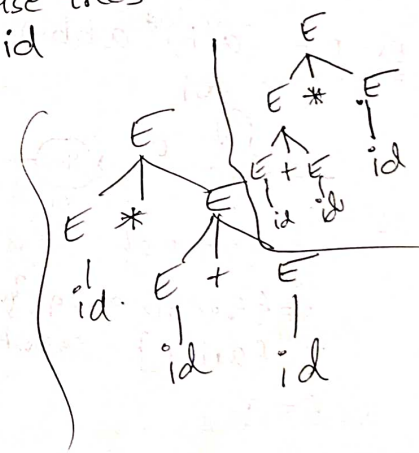
Right derivation

* Ambiguity: For any string 2 or more parse trees

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

$id + id * id$





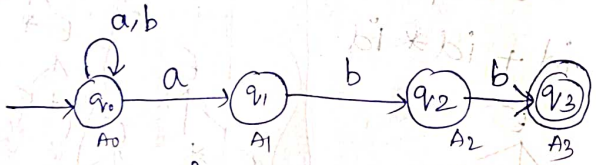
19/02/2024

* Context Free Grammar:

→ L(R) be a regular language where R is a regular expression. then $L(R) \subseteq L(G)$ where L(G) is CFG.

* If we can write CFG in terms of a RE (or) vice versa then we can say that $\boxed{L(G) = L(R)}$ (Equivalent)

Ex: $R = (a/b)^* abb$



$\because G = \{V, T, P, S\}$

$V = \{q_0, q_1, q_2, q_3\}$

$T = \{a, b, \epsilon\}$ $= \{A_0, A_1, A_2, A_3\}$.

$S = q_0$.

Productions:

$A_0 \rightarrow aA_0 | bA_0 | aA_1$

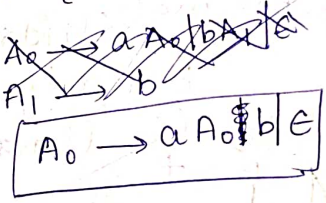
$A_1 \rightarrow bA_2$.

$A_2 \rightarrow bA_3$.

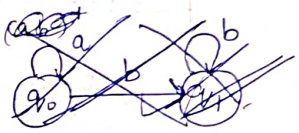
$A_3 \rightarrow \epsilon$

→ L(G) and L(R) are said to be equivalent if and only if ; they consist of similar strings.

Ex: $L = \{a^n b^n / n \geq 0\}$



$\boxed{A_0 \rightarrow aA_0 | b | \epsilon}$



* We use CFG in syntax analysis to specify the syntax of the programming language.

* Parsing is the process where we can derive the string from the start symbol (or) to check if the given set of strings are generated from the given grammar.

→ To get a proper string; we must keep on back track the parse tree to get a correct O/P.

→ This can be done using different parsing algorithms which helps us to select the efficient production.

→ done symbol by symbol

* Types of Parsing:

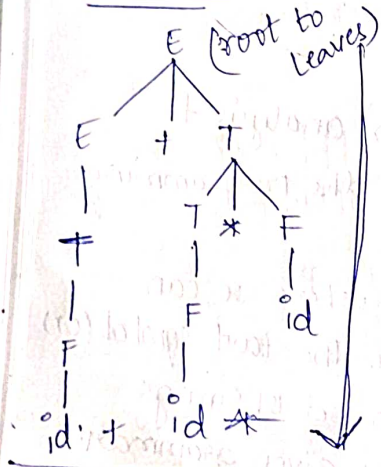
- Top down parsing
- Bottom Up parsing

Ex:

E → E+T/T
T → T*F/F
F → (E)/id

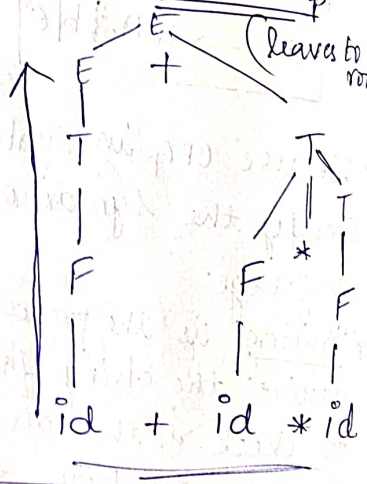
id+id*id

Top down



(root to leaves)

Bottom Up



(leaves to root)

⇒ Differentiate b/w top-down & bottom-up parsing techniques (2M).

* Top-Down Parsing Technique:

→ Recursive-Descent Parser (RDP): (Predictive Parser)

↳ For every non-terminal, we define a procedure. * Parser starts with calling the procedure of start symbol.

Ex: A → Y

```
A()
{ if (Y is non-terminal)
      Y();
  if (Y is terminal)
      terminal(Y) = current I/P symbol.
      next symbol.
  else
      error
}
```

⇒ If above procedure works properly and reach End of string i.e. $; the given grammar is proper.

→ If the given grammar has left recursion; then it cannot be passed to recursive descent parser.

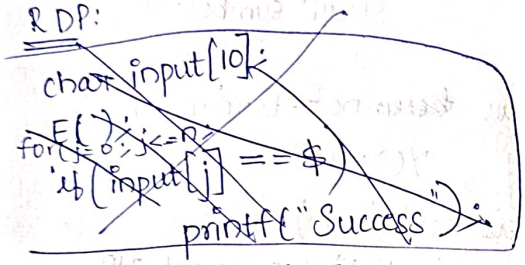
Ex:

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

$\Rightarrow$

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

RDP:



$\Rightarrow$

```
char input[10];
for(j=0;j<=n;j++)
{ E();
  if (input[j] == $)
      printf("Success"); }
```

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

```
E1()
{ if(input[j]=='+')
      j++; T(); E1(); }
```

```
T()
{ F();
  T1(); }
```

```
T1()
{ if (input[j] == '*')
      j++; F(); T1(); }
```

```
F()
{ if(input[j]=='(') { j++;
    E();
    if (input[j] == ')')
        j++; }
  else if(input[j]=='i')
        j++;
}
```

20/02/2024

→ LL(1) parser: (Predictive Non-backtracking alg.)

LL(1) → 1 Symbol is parsed at a time.
↑ → Left most derivation is used.
Scanning of the string is from left to right

* for any G (CFG) if we can construct LL(1) parser; such G is called LL(1) grammar.
→ The grammar G must ~~be~~ not have left recursion and ambiguity.

* Ex: $A \to \alpha \mid \beta \Rightarrow$ LL(1) grammar. (Checking)

$A \to \alpha \mid \epsilon$
(i) $First(\alpha) \cap First(\beta) = \emptyset$
(ii) $\alpha \overset{*}{\Rightarrow} \epsilon \Rightarrow Follow(A) \cap First(\beta) = \emptyset$
$\beta \overset{*}{\to} \epsilon \Rightarrow Follow(A) \cap First(\alpha) = \emptyset$

** FIRST SET and FOLLOW SET:

$\alpha \Rightarrow$ terminal : First Set only
$\alpha \Rightarrow$ Non-terminal : Both first set & follow set.

$\alpha \rightarrow$ grammar Symbol.
First(α) = {set of terminal symbols which are derived from α}

First(α) = a : $\alpha \overset{*}{\Rightarrow} a$

Follow(A) = Symbol that is following A in the body of production.
$X \rightarrow A\alpha\beta$

if $\alpha \rightarrow$ non-terminal ; find a terminal.

First Set: ⇒ ~~First set ∝ current I/P symbol~~
↳ if current I/P symbol is in First set (α) ; Such α is substituted

Follow set: used when non-terminal produces ϵ.

* First Set:

Let x be grammar Symbol.

1. First(x) = {x} ; if x is terminal.
2. If x is non-terminal
$X \rightarrow Y_1 Y_2 Y_3 \ldots\ldots Y_k$

if Y_1 is terminal symbol.
First(x) = {Y_1}.

If Y_1 is non-terminal First(Y_1) is having ϵ.
First(x) = First(Y_1) ∪ First(Y_2) ∪ ..... First(Y_k)

if $X \rightarrow \epsilon$ First(x) = {ϵ}

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

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

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

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

** $S \rightarrow iESS'$
$S' \rightarrow eS \mid \epsilon$
$E \rightarrow b$

First(S) = i
First(S') = {e, ϵ}
First(E) = {b}

* $S \to (L) \mid a$
  $L \to L, S \mid S$ } Left Recursion

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

$S \to (L) \mid a$
$L \to S\, L'$
$L' \to , S L' \mid \epsilon$

First(S) = { (, a }
First(L) = First(S) = { (, a }
First(L') = { , , ϵ }

* First(+EE') = {+}
* First(ET') = First(E)
  ↓ if E is having ϵ
  First(E) ∪ First(T') ∪ {ϵ}
  ⇓ if T' is having ϵ

* Follow Set:
  ↳ for non-terminals.

① If 'S' is start symbol; then Follow(S) = {$}

② Let A be any non-terminal except start symbol; $X \to \alpha A \beta B$
  if β is terminal
  then Follow(A) = {β}

  if β is non-terminal
  then Follow(A) = First(β) − {ϵ}

③ $X \to \alpha A B$ (or) $X \to \alpha A$
  if $B \to \epsilon$
  then
  Follow(A) = Follow(X)

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

Follow(E) = {$,)}
Follow(E') = {$,)}
Follow(T) = (First(E') − {ϵ}) ∪ Follow(E) ∪ Follow(E')
= {+, $,)}

∵ E' has ϵ production

Follow(T') = Follow(T) = {+, $,)}
Follow(F) = (First(T') − {ϵ}) ∪ Follow(T') ∪ Follow(T)
= {*, +, $,)}

* $A \to \alpha \mid \beta$
  First(α) ∩ First(β) = ∅
  $A \to \alpha \mid \epsilon$
  First(α) ∩ Follow(A) = ∅

Check if these conditions are satisfied for LL(1) grammar

* 28/02/2024 * Parsing Table:

1) Find first set and follow set
2) LL(1) parsing table.

rows: non-terminal.

columns: terminals including $

	a	b	c	d	$
A					
B					
C					
D					
E					

M

* LL(1) table filling algorithm:

$A \to \alpha$

$\alpha \in (V \cup T)^+$

$M[A, a] \leftarrow A \to \alpha$

$\forall\ a \in First(\alpha)$

$A \to \epsilon$

$M[A, b] \leftarrow A \to \epsilon$

$\forall\ b \in Follow(A)$

* $E \to TE'$

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

$T \to FT'$

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

$F \to (E)$

$F \to id$

$First(E) = First(T) = First(F) = \{(, id\}$

$First(E') = \{+, \epsilon\}$

$First(T') = \{*, \epsilon\}$

$Follow(E) = \{\$,)\}$

~~$Follow(E') = \{\$\}$~~ $Follow(E') = \{\$,)\}$

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

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

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

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

LL(1) parsing table.

① $E \to TE'$

$A \to \alpha$

$First(\alpha) = First(T) = \{(, id\}$

$M[E, (]$, $M[E, id]$

② $E' \to +TE' \mid \epsilon$

$A \to \alpha \mid \epsilon$

$\alpha = +TE'$

$First(\alpha) = \{+\}$

$E' \to \epsilon$

$M[E', b]$

$b \in Follow(E') = \{\$,)\}$

$M[E', \$]$, $M[E',)]$

③ $T \to FT'$

$A \to \alpha$

$a\ M[A, a]$

$a \in First(F) = \{(, id\}$

$M[T, (]$, $M[T, id]$

④ $T' \to *FT'$

$M[T', *]$

$T' \to \epsilon$

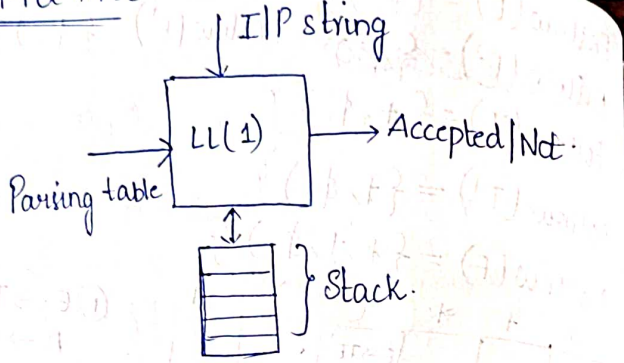
$M[T', Follow(T')]$

$M[T', +]$, $M[T', \$]$, $M[T',)]$

⑤ $F \to (E)$

$M[F, (]$

* id + id * id



Matched symbol	Stack	Input	Action
	$E	id+id*id$	M[E, id] ← E → TE'
	$E'T	id+id*id$	M[T, id] ← T → FT'
	$E'T'F	id+id*id$	M[F, id] ← F → id
id	$E'T'id	id+id*id$	pop
	$E'T'	+id*id$	M[T', +] ← T' → ε
	$E'	+id*id$	M[E', +] ← E' → +TE'
	$E'T+	+id*id$	pop
id+	$E'T	id*id$	M[T, id] ← T → FT'
	$E'T'F	id*id$	M[F, id] ← F → id
	$E'T'id	id*id$	pop
id+id	$E'T'	*id$	M[T', *] ← T' → *FT'
id+id*	$E'T'F*	*id$	pop

id+id*id	$E'T'F	id$	M[F, id] ← F → id
	$E'T'id	id$	pop
	$E'T'	$	M[T', $] ← T' → ε
	$E'	$	M[E', $] ← E' → ε
	$	$	Accepted

* S → (L) | a
L → L, S | S

Follow(S) = {$, ,,)}
Follow(L) = {)}
Follow(L') = {)}

First(S) = {(, a}
First(L) = First(S) = {(, a}
First(L') = {,, ε}

11/03/2024

* Bottom-Up Parsing:

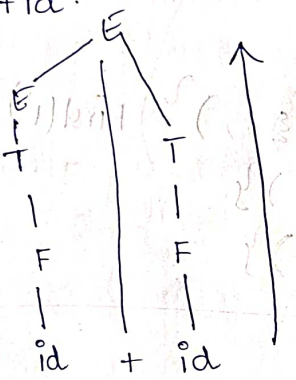
→ Leaves to the root node.

E → E+T | T

T → T* F | F

F → (E) | id

Ex: id + id



Right MD

E ⇒ E+T ⇒ E+F

⇒ E+id ⇒ T+id

⇒ F+id

⇒ id+id

→ Bottom-up parsing uses right most derivation in reverse direction.

*→ Types:

- Shift-Reduce Parser.
- LR parsers
  - Simple LR (SLR)
  - Canonical LR (CLR)
  - LALR (Look Ahead)

→ YACC: uses LALR parser
  - covers many grammars.

* Shift-Reduce Parser:

*→ operations:
- Shift
- Reduce
- Accept
- Error.

→ Moving I/P symbol on to stack: SHIFT.

→ Using any of the given productions we reduce the top of the stack symbol: REDUCE

→ When we reach the start symbol as top of stack: ACCEPT.

*→ Backtracking is possible.

Stack	Input	Action
$	id * id $	shift
$id	* id $	F → id reduce
$F	* id $	shift
$F*	* id $	shift
$T	* id $	T → F reduce
$T*	id $	shift
$T*id	$	shift
$T*F	$	F → id reduce

$T	$	I → T*F &
$E	$	E → T Reduce
$E	$	Accept.

*→ Conflicts in Shift-Reduce Parser:

→ Reduce-Reduce problem where ; we have 2 chances to reduce particular Symbol.

→ Shift-Reduce problem where we must choose b/w shifting (or) reducing.

*→ These problems are raised due to Brute-force method.

→ Parsing a lengthy string using shift-reduce parser is costly ; as backtracking is difficult and takes long time.

* LR(K) Parsers: → K = 0/1 → how many symbols parser can predict after a step.

→ Parsing is from left to right.

→ Right most derivation in reverse.

SLR → LR(0) parser.

CLR → LR(1) parser.

→ '.' placed in any productions. Specifies to use reduce (or) shift operations

A → • XYZ
A → X • YZ
A → XY • Z
} Shift operation

A → XYZ • → reduce operation

→ LR parsers are powerful and we LR grammar to implement this parser.

*→ LR is the superset of LL grammar

*→ SIMPLE LR (SLR) → placing of a dot in the production

→ LR(K) item sets
→ LR parsing table → Action part, goto part
→ Parse string

→ LR(0) item sets:

Let G → grammar.

S → start symbol.

* Convert grammar G to Augmented grammar G'

→ We add S' → S to G

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

→ Original Grammar (G)

→ Aggumented Grammar (G')

12/03/2024

* → Given grammar:

G: 1. $E \to E+T$
2. $E \to T$
3. $T \to T*F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

G': $E' \to E$
→ G.

* Operations required for LR(0) itemsets:

→ Closure
→ goto. ⇒ used to move 'dot' one symbol from #.

** Closure: checks if the '.' is placed before non-terminal or not.
and if it is before a non-terminal

Ex: Closure (I)

$A \to \alpha \cdot B \beta$
$B \to \cdot \gamma$

⇒ for the grammar G **

→ Item Sets

I_0

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

Adding the productions if the '.' is before non-terminal and this process continues till no new item is added.

If any item set exists; it need not be included:

I_1 : goto (I_0, E)

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

Items from I_0 where '.' is before E

I_2 : goto (I_0, T)

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

I_3 : goto (I_0, F)

$T \to F.$

production into the item sets.

I_4 : goto $(I_0, ()$

$F \to (\cdot E)$

$E \to \cdot E+T$

$E \to \cdot T$

$T \to \cdot T*F$

$T \to \cdot F$

$F \to \cdot(E)$

$F \to \cdot id$

I_5 : goto (I_0, id)

$F \to id\cdot$

I_6 : goto $(I_1, +)$

$E \to E+\cdot T$

$T \to \cdot T*F$

$T \to \cdot F$

$F \to \cdot(E)$

$F \to \cdot id$

I_7 : goto $(I_2, *)$

$T \to T*\cdot F$

$F \to \cdot(E)$

$F \to \cdot id$

I_8 : goto (I_4, E)

$F \to (E\cdot)$

$E \to E\cdot+T$

goto $(I_4, T) = I_2$

goto $(I_4, F) = I_3$

goto $(I_4, () = I_4$

goto $(I_4, id) = I_5$

* 2 items are equal iff : all the items are equal

I_9 = goto (I_6, T)

$E \to E+T\cdot$

$T \to T\cdot*F$

goto $(I_6, F) = I_3$

goto $(I_6, () = I_4$

goto $(I_6, id) = I_5$

Not to be written

I_{10} : goto (I_7, F)

$T \to T*F\cdot$

goto $(I_7, () = I_4$

goto $(I_7, id) = I_5$

I_{11} : goto $(I_8,))$

$F \to (E)\cdot$

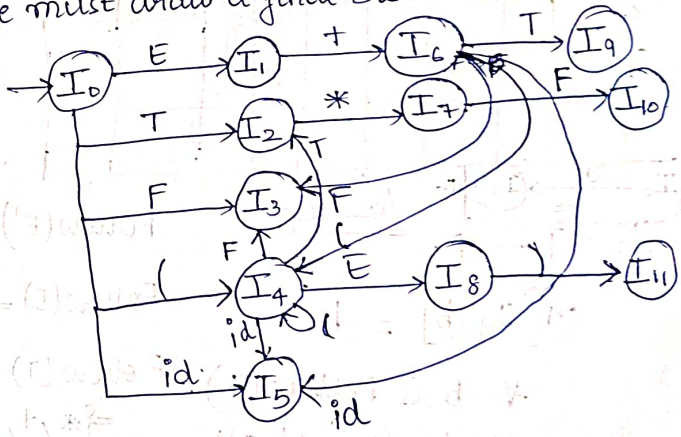
goto $(I_8, 4) = I_6$

goto $(I_9, *) = I_7$

→ To check if the given items are correct; we must check if all the items contain each & every production with '·' at its right end.

Ex: $E \to E+T\cdot$

→ We must draw a finite state machine



	Action						Goto		
Items sets	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			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

$I_0 \xrightarrow{id\ s5.} I_5$ *

S → shift

$I_3 : T \rightarrow F.$

$M[I_3, b] = b$

$\forall$ b is Follow(T).

We must place r_4 at all the places of Follow(T)

r → reduce

4 → no. in production set.

Follow(E') = {$}

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

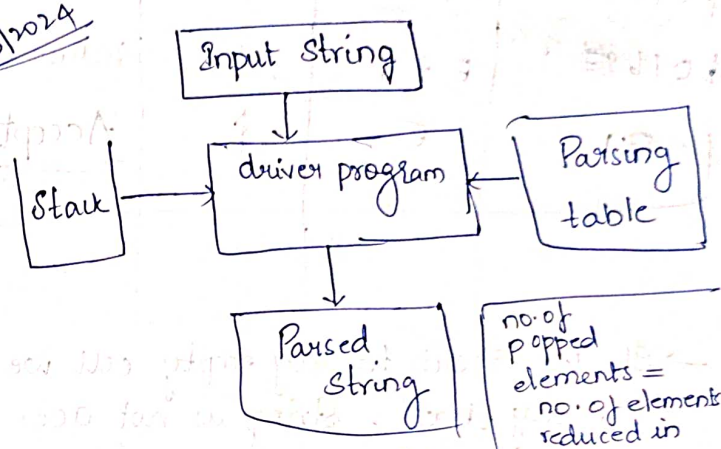
Follow(T) = {*, +,), $}

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

$I_5 : F \rightarrow id.$ r_6

$I_9 : E \rightarrow E+T.$

* 14/03/2024



→ Given string : id+id*id.

Stack	Symbol	Input	Action
$0		id+id*id$	shift (s5)
$05	id	+id*id$	reduce F→id
$03	F	+id*id$	reduce T→F
$02	T	+id*id$	reduce E→T
$01	E	+id*id$	shift (s6)
$016	E+	id*id$	shift (s5).
$0165	E+id.	*id$	reduce F→id.
$0163	E+F	*id$	reduce T→F
$0169	E+T	*id$	shift (s7

$0169 7	E+T*	id $	shift (5)
$0169 7 5	E+T*id	$	reduce F
$0169 7 10	E+T*F	$	reduce T
$0169 9	E+T	$	reduce E
$0 1	E	$	Accept

→ If we reach to any empty cell we can say that ; string is not acc. to grammar.

* S → CC
C → aC
C → d

Aggumented Grammar
G' ⇒ S' → S
S → CC
C → aC
C → d

I_0:
S' → .S
S → .CC
C → .aC
C → .d

I_1 = goto(I_0, S)
S' → S.

I_2 = goto(I_0, C)
S → C.C
C → .aC
C → .d

First(s) = First(C) = {a, d}
First(C) = {a, d}
Follow(s) = {$}
Follow(C) = First(C) - {ε} = {a, d, $}

I_3 = goto(I_0, a)
C → a.C
C → .aC
C → .d

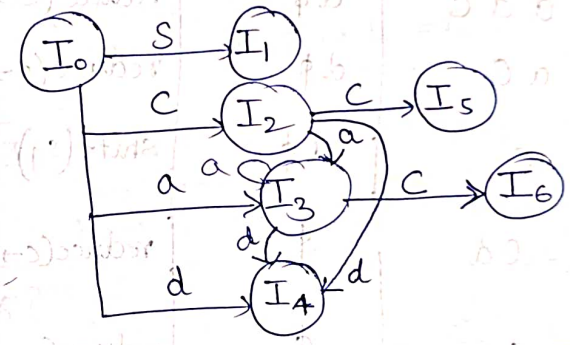
I_4 = goto(I_0, d)
C → d.

goto(I_2, d) = I_4
goto(I_2, a) = I_3

I_5 = goto(I_2, C)
S → CC.

I_6 = goto(I_3, C)
C → aC.

goto(I_3, a) = I_3
goto(I_3, d) = I_4



	a	d	$	S	C
0	S_3	S_4		1	2
1			Acc		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5			r_1		

String aadd.

Stack	Symbol	I/P	Action
$ 0		a a d d $	shift(s_3)
$ 0 3	a	a d d $	shift(s_3)
$ 0 3 3	a a	d d $	shift(s_4)
$ 0 3 3 4	a a d	d $	reduce($c \to d$)
$ 0 3 3 6	a a C	d $	reduce($c \to$
$ 0 3 6	a C	d $	reduce($c \to$
$ 0 2	C	d $	shift(s_4)
$ 0 2 4	C d	$	reduce($c \to d$)
$ 0 2 5	C C	$	reduce($s \to cc$)
$ 0 1	S	$	Accept ~~Reduce~~

1 $S \to L=R$
2 $S \to R$
3 $L \to *R$
4 $L \to id$
5 $R \to L$

Aggumented Grammar
$S' \to S$
$S \to L=R$
$L \to *R$
$L \to id$
$R \to L$
$S \to R$

First(S) = First(L) = {*, id}
First(R) = First(L) = {*, id}
Follow(S) = {$}
Follow(L) = {=}
Follow(R) = ~~{=}~~ {=, $}

I_0 =
$S' \to \cdot S$
$S \to \cdot R$
$S \to \cdot L=R$
$L \to \cdot *R$
$L \to \cdot id$
$R \to \cdot L$

$I_1 = goto(I_0, S)$
$S' \to S \cdot$

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

$I_4 = goto(I_0, *)$
$L \to * \cdot R$
$R \to \cdot L$

$I_5 = goto(I_0, id)$
$L \to id \cdot$

$I_6 = goto(I_2, =)$
$S \to L= \cdot R$
$R \to \cdot L$

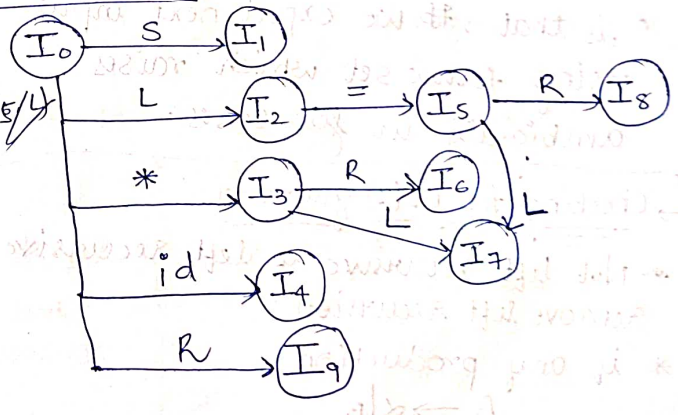
$I_7 = goto(I_3, R)$
$L \to *R \cdot$

$I_8 = goto(I_3, L)$
$R \to L \cdot$

$I_7 = goto(I_5, L) = I_7$

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

$I_3 = goto(I_0, R) \Rightarrow S \to R \cdot$



	=	*	id	$	S	RL	R
0		S4	S5		1	2	3
1				ACC			
2	S6/r5			r5			
3						8	7
4							
5						8	9
6							
7							
8							

∵ there is an ambiguity to choose between shift/reduce operation in cell (2, =)

* The demerits with SLR grammar is that; we expect next input using follow set which raises ambiguity in few cases.

* → Checking for LL(1) grammar:

* Not left recursive if left recursive, remove left recursion.
* if any production
  $A \to \alpha | \beta$

  $First(\alpha) \cap First(\beta) = \emptyset$
* if $A \to \alpha | \epsilon$

  $First(\alpha) \cap Follow(A) = \emptyset$

→ The demerits in SLR is due to LR(0) item sets; which can be avoided using LR(1) item sets.

* → LR(1) item sets:

→ Closure operation:

* Lookahead | the next input ⇒ shd be expected a terminal
* 2 item sets are equivalent only if both items and lookaheads are equal.

* $A \to \cdot \alpha, \$$

  $A \to \alpha \cdot B\beta, \alpha$

  $B \to \cdot \gamma$, $First(\beta, a)$.

→ The parser constructed from LR(1) item sets is called Canonical LR parser.

$S' \to S$

1. $S \to L = R$
2. $S \to R$
3. $L \to *R$
4. $L \to id$
5. $R \to L$

First($) = {$}

I_0:

$S' \to \cdot S, \$$

$S \to \cdot L = R, \$$

$S \to \cdot R, \$$

$L \to \cdot *R, =$

$L \to \cdot id, =$

$R \to \cdot L, \$$

$I_1 = goto(I_0, S)$

$S' \to S \cdot, \$$

$I_2 = goto(I_0, L)$

$S \to L \cdot = R, \$$

$R \to L \cdot, \$$

$I_3 = goto(I_0, R)$

$S \to R \cdot, \$$

$I_4 = goto(I_0, *)$

$L \to * \cdot R, =$

$R \to \cdot L, =$

$I_5 = goto(I_0, id)$

$L \to id \cdot, \$$

$I_6 = goto(I_2, =)$

$S \to L = \cdot R, \$$

$R \to \cdot L, \$$

$I_7 = goto(I_4, R)$

$L \to *R \cdot, =$

$I_8 = goto(I_4, L)$

$R \to L \cdot, =$

$I_9 = goto(I_6, R)$

$S \to L = R \cdot, \$$

$I_{10} = goto(I_6, L)$

$R \to L \cdot, \$$

* Parsing Table:

	id	*	=	$	S	L	R
0	S_5	S_4			1	2	3
1				Acc			
2			S_6	r_5			
3				r_2			
4						8	7
5				r_4			
6						10	9
7			r_3				
8			r_5				
9				r_1			
10				r_5			

$C \to \cdot d, \$$

I_3: goto (I_0, a)
$C \to a \cdot C, a/d$
$C \to \cdot d, a/d$
$C \to \cdot aC, a/d$

$I_4 = goto(I_0, d)$
$C \to d \cdot, a/d$

$I_5 = goto(I_1, C)$
$S \to CC \cdot \$$

$I_7 = goto(I_2, d)$
$C \to d \cdot, \$$

$I_8 = goto(I_3, C)$
$C \to aC \cdot, a/d$

$I_6 = goto(I_2, a)$
$C \to a \cdot C, \$$
$C \to \cdot aC \$$
$C \to \cdot d \$$

$I_9 = goto(I_6, C)$
$C \to aC \cdot, \$$

→ For LALR; if the item sets have same items; but having different look aheads; can be merged.

I_0: $S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot ac, a/d$
$C \to \cdot d, a/d$

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

I_2: goto (I_0, C)
$S \to C \cdot C, \$$
$C \to \cdot aC, \$$
$C \to \cdot d, \$$

I_{36}:
$C \to a \cdot C, a/d/\$$
$C \to \cdot aC, a/d/\$$
$C \to \cdot d, a/d/\$$

I_{47}:
$C \to d \cdot, a/d/\$$

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

I_{89}: $C \to aC \cdot, a/d/\$$

* $S \to CC$
$S \to aC$
$S \to d$

Aiggumented Grammar:

$S' \to S$
$S \to CC$
$S \to aC$
$S \to d$

I_0:
$S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot aC, a/d$
$C \to \cdot d, a/d$

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

I_2: goto (I_0, C):
$S \to C \cdot C, \$$

	a	d	$	S	C	
0	S_{36}	S_{47}		1	2	
1			Acc.			
2	S_{36}	S_{47}			5	
36	S_{36}	S_{47}			89	
47	r_3	r_3	r_3			
5			r_1			
89	r_2	r_2	r_2			