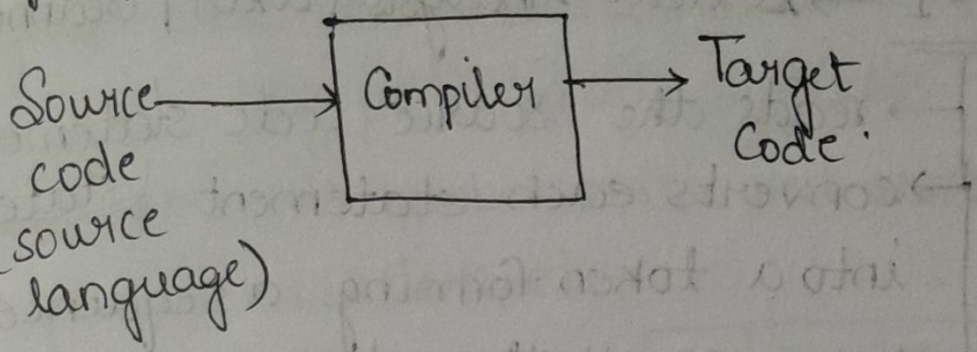
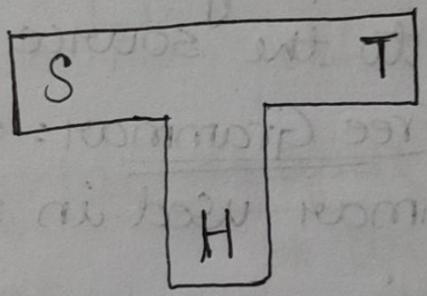
05/02/2024

→ 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 programming language for a compiler is called Bootstrapping
→ 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.

| Name | Type | descri |
|---|---|---|
| int | keyword | integer |
| | | |
| | | |

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

Phase-2: Syntax Analysis (Parser)

* all terminal symbols are given by operators.

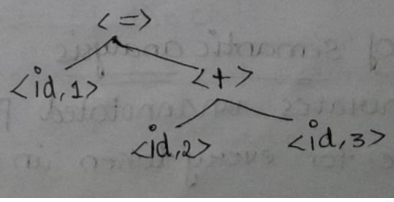
Ex: a = b + c

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.
- Interview questions — **Strongly typed languages**: The type of the variable is explicitly declared.

  Eg: C, C++, Java.

  **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, R1
```

↓ to the rightside 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.

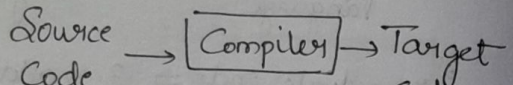
Phase - 4, 5: Synthetic Phases.

10/02/2024

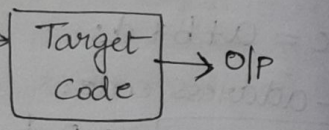
# UNIT-1

## * Language Processors:

→ Compiler:



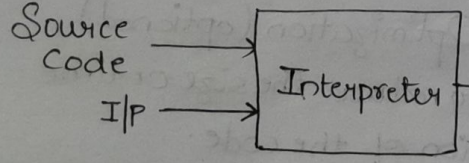

* Target Code is executable.




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

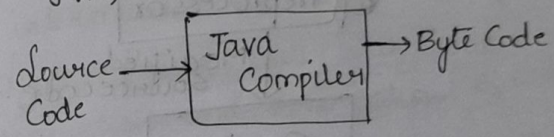
→ Interpreter:

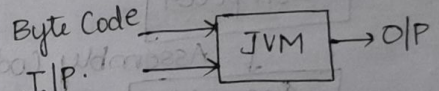
* Converts each statement of the code one at a time to target code.




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

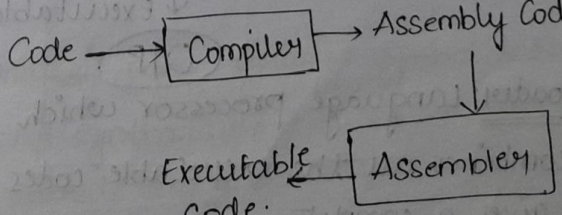
* Portability: The target code/byte code which is executable on any other machine; is called language portability
Ex: Java (runs on any system having JVM).



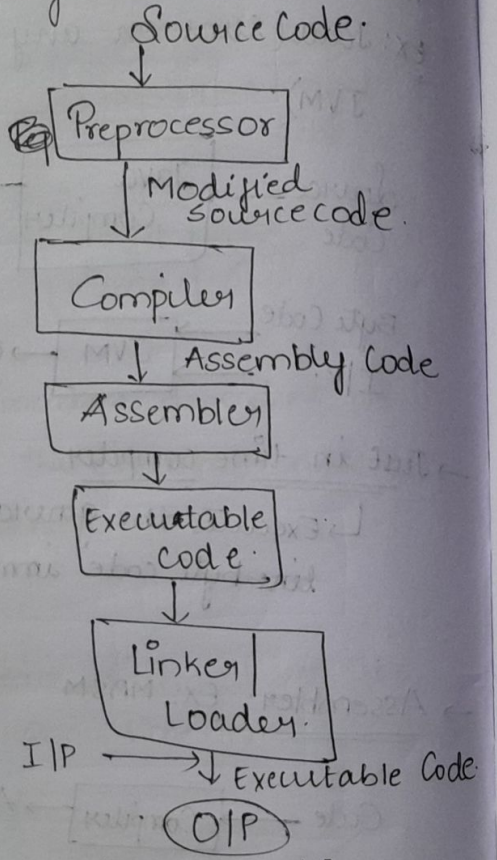




→ Just in time compiler:
↳ Executes the generated single line byte code immediately.

→ 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 are used to get the O/P.

* Structure of the compiler:

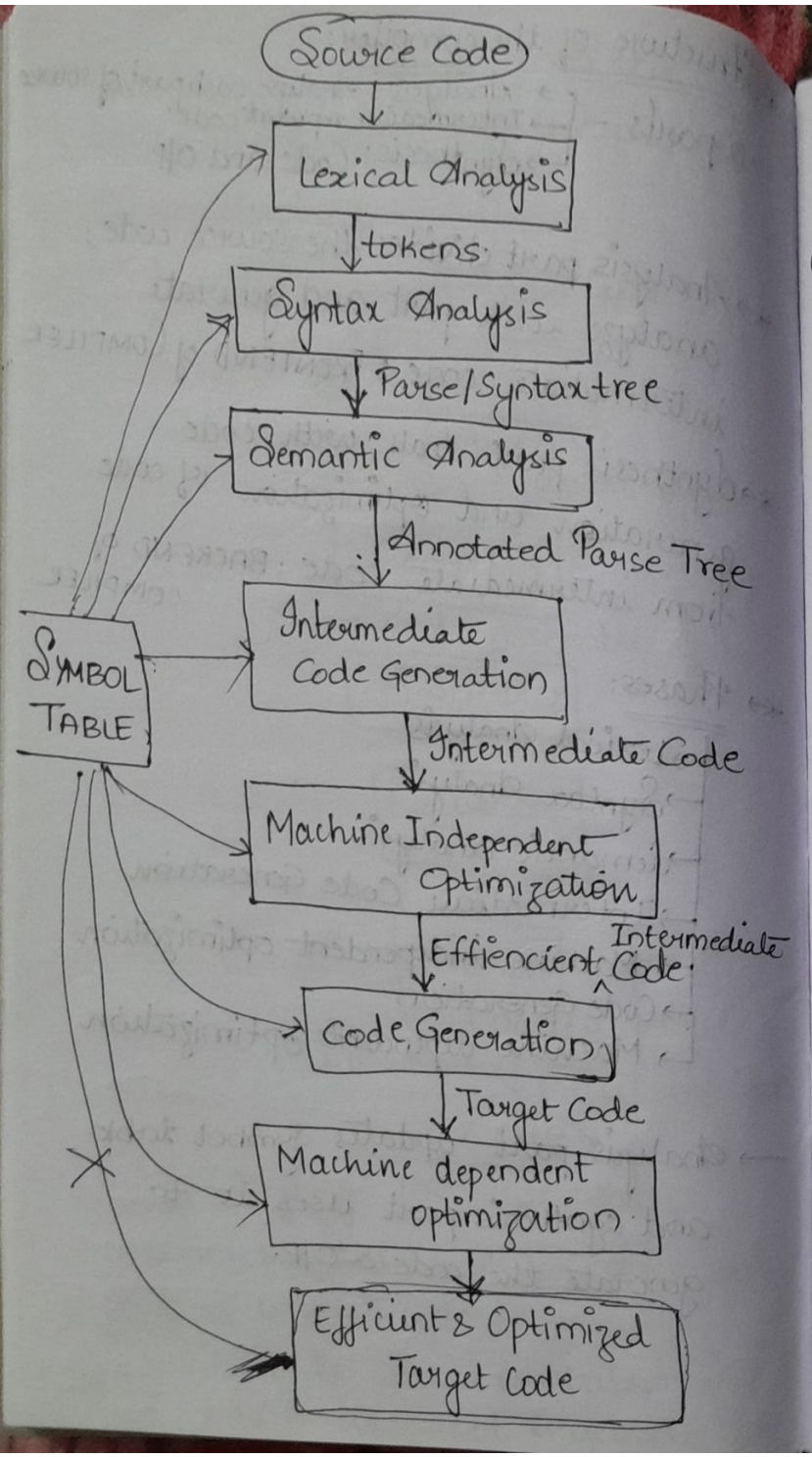
→ 2 parts
- Analysis: Analyse each part of source code. Intermediate represent code.
- Synthesis: Code and O/P

** 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>

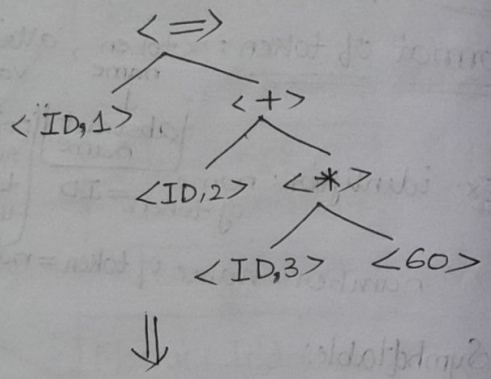
(2) Syntax Analysis:

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

## ② Syntax Analysis:

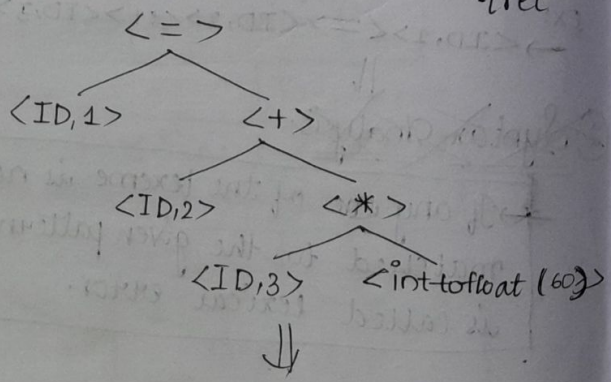
Errors detected here are called syntax errors.

→ If no error; syntax tree is generated.




⇓

## ③ Semantic Analysis: Annotated parse tree.




⇓

## ④ Intermediate Code Generation:

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

$t_1$ = inttofloat(60)

$t_2$ = rate * $t_1$ | $t_2$ = <id,3> * $t_1$

$t_3$ = initial + $t_2$ | $t_3$ = <id,2> + $t_2$

<id,1> = $t_3$

⇓

## ⑤ MIO:

$t_1$ = <id,3> * 60.0

<id,1> = <id,2> + $t_1$

## ⑥ Code Generation:

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

12/02/2024

→ 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.

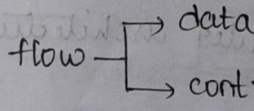
→ Lex: It is a tool that automatically generates/implements lexical analyser.

→ YACC/parser generator: automatic parse tree generation with taking input as grammar.

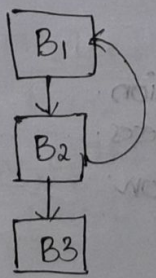
YACC: Yet Another Compiler Compiler.

→ syntax directed translation engine: Annotated parse-tree generation.

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




→ 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 synthesis.
  └→ Binary Translation.

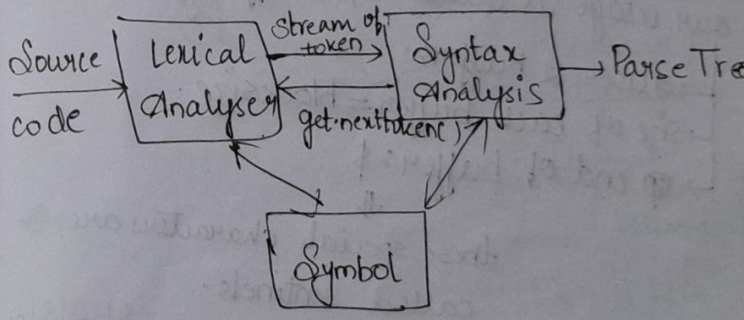
Programming language basics.

→ Static | dynamic distinction.

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

12/02/2024

→ 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 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

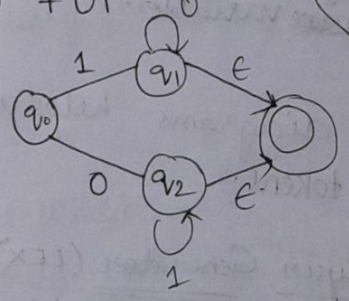
→ Specification of patterns/tokens
- → Regular Expressions.
- → Representation of regular languages.
- → Operations:
  - → Union $L_1 \cup L_2$
  - → Intersection $L_1 \cap L_2$
  - → Concatenation $L_1 L_2$
  - → closure : $L_1^*$ ; $L_1^+$
  - → Complement $\overline{L_1}$
  - → Reverse $L_1^R$

Ex: 1000110

0110 → substring

0010 → subsequence.

Ex: $10^* + 01^*$




letter → [A-Z a-z]

digit → [0-9]

Regular definitions

*[letter]

[letter|digi

⇒[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 and 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.
↳ file.l → lex compiler → lex.yy.c

lex.yy.c → C compiler → Executable code.

Executable code → ./a.out → O/P

* 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)

17/02/2024

* 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

E → head of the production

E+T | T → body of the production

V = { E, T, F }

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

} total no. of productions = 6

↓↓

order of production.

(only one variable)

S = E

→ Issues in CFG:

↳ Left recursion: A → Aα | β | a

Correction: A → βA' | ε ; A' → αA' | ε

↳ Left factoring: A → αβ1 | αβ2 | αβ3

↳ Ambiguous: (A → αA' ; A' → β1 | β2 | ... βn) ... | αβn

* → Few algorithms do not produce efficient parsers.

∵ Given productions have left recursions.

∴ E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

→ Left Factoring:

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

$S \to ietss' \mid a$

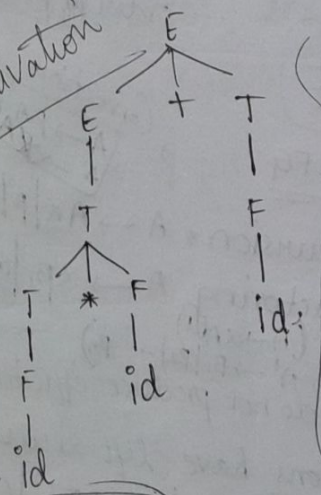
$s' \to eS \mid \epsilon$

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

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

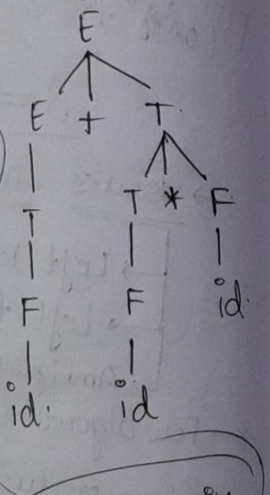
Ex:

Left derivation




id * id + id

Parse Tree

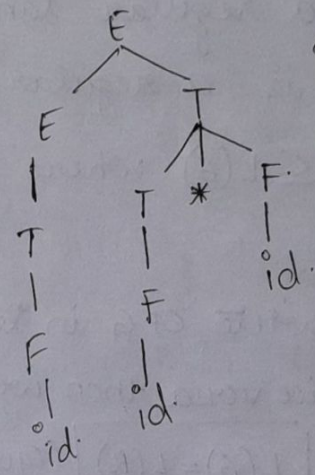



id + id * id

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

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

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

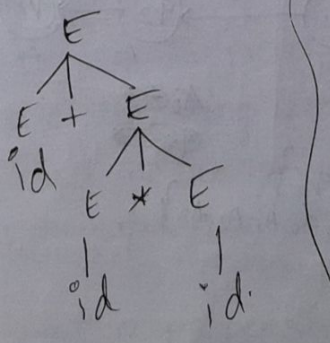


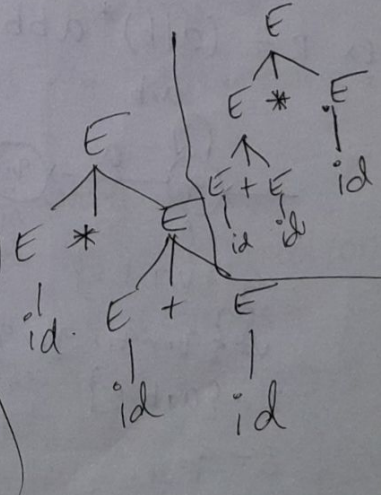

Right derivation

* Ambiguity: for any string ≥ or more parse trees

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

id + id * id



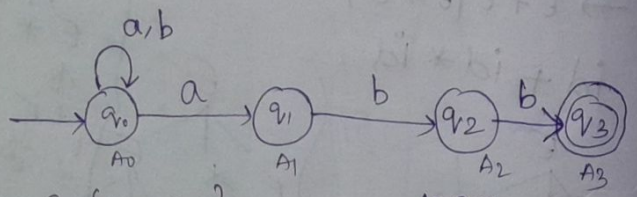
19/02/2024

## * 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:

$A_0 \rightarrow aA_0 | bA_0 | aA_1$

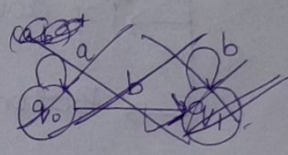
$A_1 \rightarrow bA_2$.

$A_2 \rightarrow bA_3$.

$A_3 \rightarrow \epsilon$

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

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




$A_0 \rightarrow aA_0 | bA_1 | \epsilon$

$A_1 \rightarrow b$

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

* 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

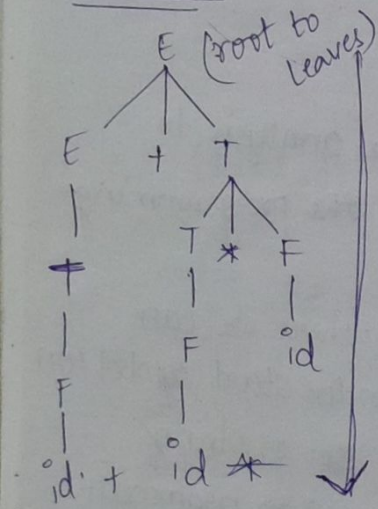
* → Types of Parsing:

- → Top down parsing
- → Bottom Up parsing

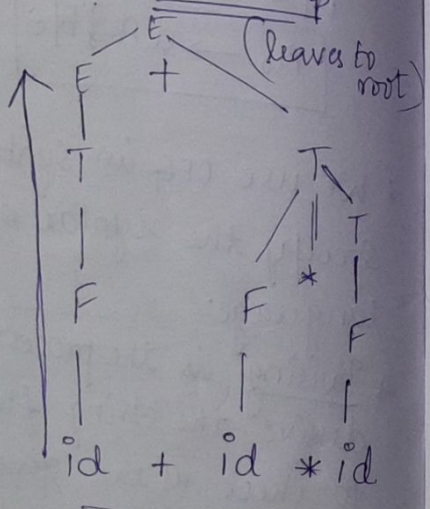
Ex:

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:

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

$\Rightarrow$

```
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:

L → Left to right scanning of the string
L(1) → 1 symbol is parsed at a time
→ Left most derivation is used.

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

## ** 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($\alpha$) = {set of terminal symbols which are derived from $\alpha$}

First($\alpha$) = 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 ($\alpha$); Such $\alpha$ is substituted

Follow set: used when non-terminal produces $\epsilon$.

## * 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 $\epsilon$.
First(x) = First($Y_1$) ∪ First($Y_2$) ∪ ..... First($Y_k$)

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

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

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

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

~~First(X)~~

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

First(S) = i
First(S') = {e, $\epsilon$}
First(E) = {b}

* $S \to (L) | a$
  $L \to L, S | S$
} Left Recursion
$A \to \beta A'$
$A' \to \alpha A' | \epsilon$

$S \to (L) | a$
$L \to \cancel{S} L' | S$
$L' \to , S L' | \epsilon$

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 $X \to \cancel{\alpha} AB$ (or) $X \to \alpha A$
  if $B \to \epsilon$
  then
  Follow(A) = Follow(X)
  Follow(A) = Follow(X)

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

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

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

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