Context free Grammar (CFG)

Lang → Collection of strings

String is accepted by finite Automata → Regular Language

string can be combined by means of rules called Grammar.

↳ Denoted by G

G(V, T, P, S)

V → Variables

T → Terminals

P → Productions

S → Start Symbol

Input alphabets

T → Terminals.

↳ Σ → Input alphabet

for C is literals, operators, identifier, keywords, constants

for english A-Z

In order to combine tokens require rules called rewriting rules

Re-writing rules

Terminals

Stmt → { Stmt1; Stmt2; } (Compound Stmt)

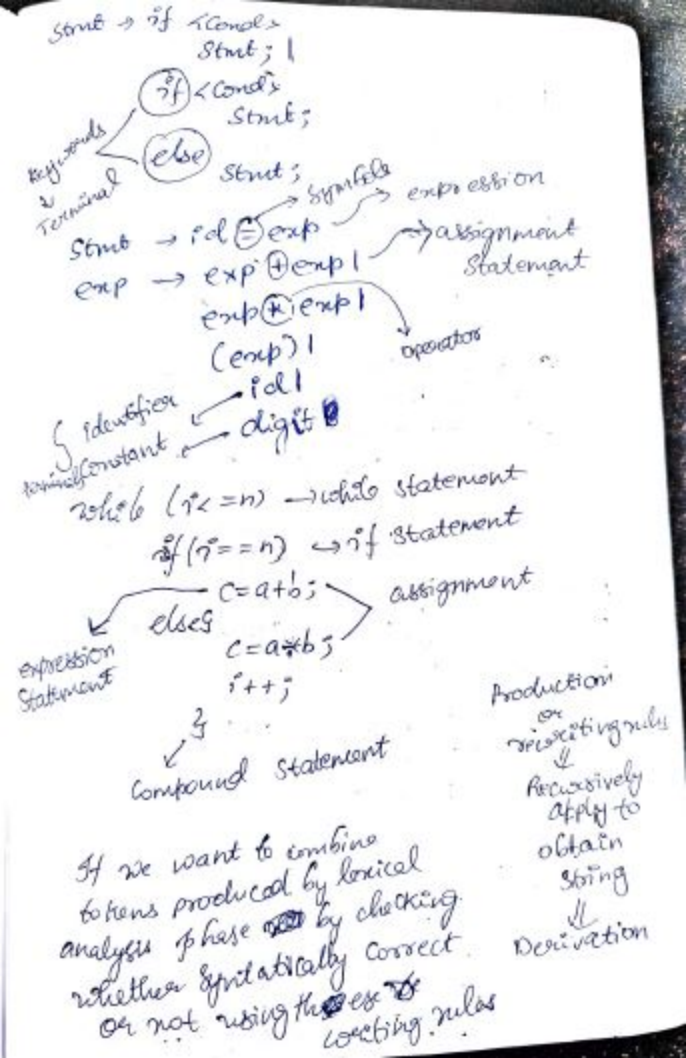
Stmt → ; | Stmt (single Stmt)

Stmt → while (cond) Stmt (single Stmt)

Stmt; can be single or compound statement

Single Stmt → Need not be in curly braces

Compound Stmt → Can be enclosed within curly braces


Stmt → if <Cond>
Stmt ; |
if <Cond>
Stmt ;
else
Stmt ;
Keywords & Terminal
Stmt → id = exp
exp → exp + exp |
symbols
expression
assignment Statement
exp * exp |
(exp) |
operator
identifier
terminal
id |
digit
while (i <= n) → while statement
if (i == n) → if statement
c = a + b ;
else
c = a * b ;
assignment
expression Statement
i ++ ;
}
Compound Statement
Production or rewriting rules
Recursively apply to obtain String
Derivation
If we want to combine tokens produced by lexical analysis phase by checking whether syntatically correct or not using these rewriting rules

Left hand side of arrow → Variables

Ex: Start, exp

Terminal → which can be participated in formation of string

Tokens are input alphabet.

S → From where to start in grammar. So that we can obtain valid string

To combine individual alphabets to obtain valid strings of programming language, so G(V, T, P, S) used

& valid statement of programming language

Derivation: The process of obtaining valid string by applying production rules recursively starting with start symbol.

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

id + id * id

$E \Rightarrow (E) + E$

$\Rightarrow id + (E)$

$\Rightarrow id + (E) * E$

$\Rightarrow id + id * (E)$

$\Rightarrow (id + id * id)$ → sentence

Left most derivation
Right most derivation

String which contain only terminals is called as sentence

String contains both terminals & non terminals is called Sentential form
↓
N.T & & T's

$\Rightarrow$ → Notation for single step derivation

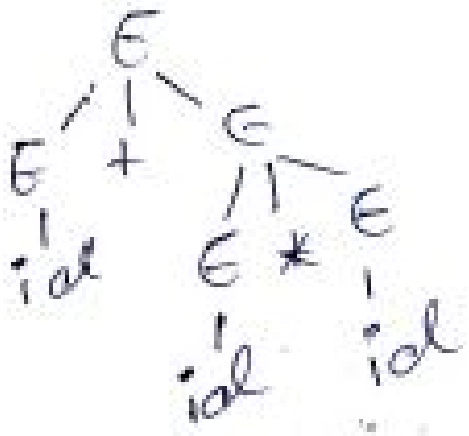
$\overset{*}{\Rightarrow}$ → Notation for 0 or more steps

$E \Rightarrow E+E \Rightarrow id+E \overset{*}{\Rightarrow} id+id*id$

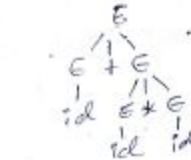
$\underset{lm}{\Rightarrow}$ → Notation for left most derivation & single step

$\underset{rm}{\overset{*}{\Rightarrow}}$ right most derivation used & multiple steps

Parse Tree → Graphical representation of derivation of string



Ambiguity → for given grammar we can obtain two or more left most derivation / two or more right most derivation / two diff parse tree for given string.



Yield → combination of all terminals from right to left

If string valid, yield always give sentence

Two diff parse tree ⇓ So ambigious

① Non terminals → All uppercase letters
② All terminals → lowercase letters
③ Grammar Symbol → Can be terminal or non terminal

T NT

X, Y, Z → Capital

④ α, β, γ → Sentential form $(VT)^*$

⑤ u, v, w → Sentences $- T^*$

6) $A \to \alpha_1 \quad A \to \alpha_2 \ldots A \to \alpha_n$

$A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n$

$S \overset{*}{\Rightarrow} \alpha A \gamma \Rightarrow \alpha \beta \gamma = \omega$

↓ multiple derivation step

↓ single derivation

$S \overset{*}{\Rightarrow} \omega$ (since ω derived from start symbol so called sentence of Grammar)

Grammar → set of rules to obtain valid string

Nonterminals / Variables → Doesn't take place in string but it helps us to obtain string

Rewriting rules → Left hand side → Variable
→ Right hand side → Variables & terminals

$S \rightarrow iss'|a$

$\rightarrow i(S)s'$

$\rightarrow ia(S')$

$\rightarrow iae(S)$

$\rightarrow iaea$

$S \rightarrow i(S)s'$

$\rightarrow i(S)s's'$

$\rightarrow i(a(S')s's'$

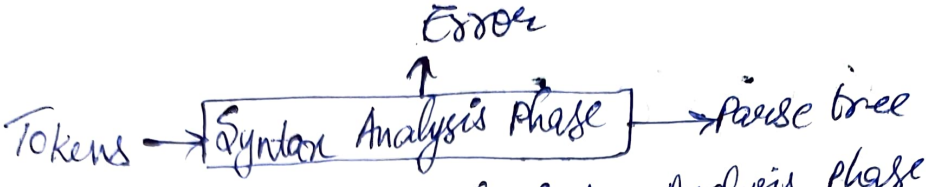
$\rightarrow i(aes's'$

$\rightarrow i(aea(S')$

$\rightarrow i(aea$

Unit-2

Syntax Analysis phase



Another name for Syntax Analysis phase is Parser

Techniques are broadly classified into two

Parser
- Top-down technique
  - Brute force Parser
  - Recursive-Descent Parser
  - Predictive Parser
- Bottom-up technique
  - simple
  - efficient

(LL parser) Top down technique (uses left most derivation)
↳ starting with root (start symbol) expand it repeatedly until come down leaves (all terminals we get)

(LR parser) Bottom-up techniques (uses right most derivation in reverse)
↳ start with leaves & go till root

Ex: CYK algorithm

Predictive Parser (Non Recursive Recursive Descent Parser)

Recursive-Descent → simple
Predictive Parse - efficient

simple - shift-reduce
- operator-precedence

efficient → simple LR
↳ canonical LR
↳ lookahead LR

Ambiguous grammar
YACC

Brute-force parser
↳ Its like derivation but in systematic manner

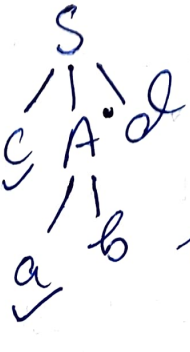
$S \rightarrow cAd$
$A \rightarrow ab \mid a$

w → string of terminals
w = cabd → whether accepted by grammar or not

Start with start symbol S
Always try with first alternative of closest
go back (undo)
work go for another alternative

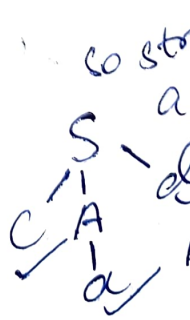
String accepted by grammar

w = cad



b ≠ d didnt match
undo expansion

⇒

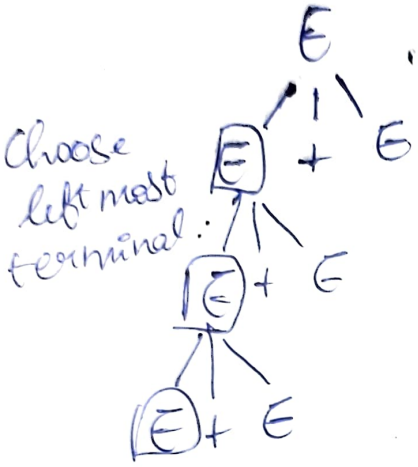


c matched
a matched
d matched
so string accepted

$E \rightarrow E + E / id$

id + id

(*) If grammar Contain left recursion then algo doesnt work as it goes into infinite loop



Choose left most terminal

Enter into infinite loop

As grammar Contains left recursion

Algorithm:

① Start with Start symbol

② In expanded string choose left most non terminal & expand it with its first alternative

③ If obtained string not matched with current input symbol then undo expansion & try with next alternative.

④ Repeat Step ② & ③ until no more non terminals to expand.

⑤ If obtained string Contains all terminals & matched with input string announce successful

parsing otherwise call error().
↳ error recovery routine.

Drawbacks of this algorithm

① If Grammar contains left recursion the parser goes into infinite loop.

② Undoing expansion (Backtracking)
↓
The process of undoing previous expansion is called backtracking

During expansion we need to maintain symbol table

Undoing of changes made in Symbol table becomes difficult.

③ The way the alternatives are choosing is another problem.
(Ex: we always choose 1st alternative)

④ When error encounters it is very difficult to trace where exactly error happens.

How to implement brute force parser

① For every non terminal or production write a code.
For diff production diff code.

function or procedure

```
void S()
{
    if (i/p-Sym == 'c')
    {   Advance();
        A();
        if(i/p-Sym == 'd')
        {
            Advance();
            return true;
        }
        else{
            error();
        }
    }
    else{
        error();
    }
    return false;
}

bool A()
{
    this-ptr = curr-i/p-ptr;
    if(i/p-sym == 'a'){
        Advance();
        if(i/p-sym == 'b'){
            Advance();
            return true;
        }
    }
```

```
char-I/P-ptr = this->ptr;
if (I/P->sym == 'a')
{
    Advance();
    return true();
}
error();
}
```

Left-Recursion ↗ G_1

If prod $A \to A\alpha \mid \beta$ then replace it with

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$ $\to G_2$

$G_1 \cong G_2$ (They are equivalent)

$\Downarrow$

$L(G_1) = L(G_2)$ Both recognize same language

Ex:

$E \to E + T \mid T$

$A \to A \alpha \mid \beta$

New non terminal A' acts as bridge between two productions

$E \to TE'$

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

Ex: $T \to T * F \mid F$

$A \to A \alpha \mid \beta$

$\Rightarrow$ $T \to FT'$

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

③ $R \to R\,\underbrace{\&\&}_{\alpha}\,T \mid \underbrace{T}_{\beta}$

$A \to A\alpha \mid \beta$

$R \to TR'$

$R' \to \&\&TR' \mid \varepsilon$

④ $T \to T \,||\, F \mid F$

$T \to FT'$

$T' \to ||FT' \mid \varepsilon$

⑤ $A \to \underbrace{Aa}_{\alpha} \mid \underbrace{bcd}_{\beta}$

$A \to bcdA'$

$A' \to aA' \mid \varepsilon$

Left recursion when multiple

$A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

$A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$

⑥ $E \to \underbrace{E+E}_{\alpha_1} \mid \underbrace{E*E}_{\alpha_2} \mid \underbrace{E-E}_{\alpha_3} \mid \underbrace{E}_{\alpha_4} \mid \underbrace{(E)}_{\beta_1} \mid \underbrace{a}_{\beta_2} \mid \underbrace{\beta}_{\beta_3}$

$E \to (E)E' \mid aE' \mid \beta E'$

$E' \to +EE' \mid *EE' \mid -EE' \mid \varepsilon EE' \mid \varepsilon$

⑦ Left recursion found after one or more steps

$A \to \underbrace{Aa}_{\alpha_1} \mid \underbrace{Abc}_{\alpha_2} \mid \underbrace{Adef}_{\alpha_3} \mid \underbrace{g}_{\beta_1} \mid \underbrace{h}_{\beta_2} \mid \underbrace{f}_{\beta_3} \mid \underbrace{klm}_{\beta_4}$

$A \to gA' \mid hA' \mid fA' \mid klmA'$

$A' \to aA' \mid bcA' \mid defA' \mid \varepsilon$

~~Substitute~~

$S \to Sa \mid Ab \mid d$ → before this no production of A to substitute

$A \to Ab \mid Sd \mid e$ → Here substitution possible

$S \rightarrow Ab s' | ds'$

$s' \rightarrow as' | \varepsilon$

$A \rightarrow Ab | Abs'd | ds'd | e$

(3) $A \rightarrow ds'dA' | eA'$

(4) $A' \rightarrow bA' | bs'dA' | \varepsilon$

Backtracking came into existence bez we always choose the first alternative, so to eliminate backtra-cking whichever is matched with current input symbol pick it up. (we pick up correct alternative that matches with input string)

$S \rightarrow iS | iSeS | a$

If input here is i we have two alternatives which one to pick dont know

If input string iaea if we choose first one is S then we need to backtrack

So there is ambiguity in alternative. So to overcome we use left factoring

$A \rightarrow aA | bA | cA$

Before we first looked a A if error went to bA otherwise if error go to cA, but now if a is input choose first one if input is b then choose second one.

Left-factoring : The process of factoring out the alternatives in manner no two alternatives begin with same symbol.

It overcomes ambiguity in alternatives.

We factor out common prefixes such that they dont begin with same symbol

i.e If productions are form $A \to \alpha\beta \mid \alpha\gamma$.

Both begin with α, so α known as Common prefix

So we factor out common prefix such that they dont start with same symbol

$A \to \alpha A'$

$A' \to \beta \mid \gamma$

$S \to iS \mid iSeS \mid a$

$\alpha = iS$

$S \to iSS' \mid a$

$S' \to eS \mid \varepsilon$

Stmt → if Stmt | if Stmt else Stmt | a

i → stands for if

e → Stands for else

$E \rightarrow a+b \mid a+c \mid d$

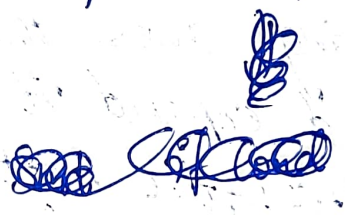
$\alpha \rightarrow a+$

$E \rightarrow \alpha E' \mid d$

$E' \rightarrow b \mid c$

$S \rightarrow iCtS \mid iCtSeS \mid a$

$C \rightarrow b$



Stmt → if Condition then Statement |
if Condition then Statement else Statement | a

Condition → b

$S \rightarrow ictSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$C \rightarrow b$

E → expression

Note: Always epsilon comes at last

$A \rightarrow ab \mid abcdabcd$

$A \rightarrow abA'$

$A' \rightarrow \underline{c \mid cd} \mid \varepsilon$

$A' \rightarrow cA'' \mid \varepsilon$

$A'' \rightarrow d \mid \varepsilon$

Recursive Descent Parsing technique

1. Eliminate left recursion if any. Eg. remove ambiguity in alternatives if any by applying left factoring. (Preprocessing alone)
2. Start with start symbol
3. In the obtained string choose the left-most non terminal and expand it by picking the correct alternative matched with current input symbol (Curr-I/p-Symbol)
4. Repeat step 3 until no more non terminal to expand.
5. If the resultant string contains all T's (terminals) and matched with I/p string then announce 'Successful parsing otherwise error (?).

In general parser we pick up first alternative but here pick correct alternative matched with current I/p symbol

Ex:

$E \to E + T \mid T$ ...①

$T \to T * F \mid F$

$F \to (E) \mid id$

↓

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

$w = id + id$ $\quad E \to TE'$

$E \Rightarrow TE'$ (only 1 option choose that only.)

$\Rightarrow FT'E'$ (Always expanding left non terminal)

$F \to id$

~~Randy Orton~~ (and alternate (moved pointer) (-ve choose)

$\Rightarrow id\,T'E'$ (Left most non terminal)

$T' \to \varepsilon$

$\Rightarrow id\,E'$ ($T' \to \varepsilon$ only one option)

$\Rightarrow id + TE'$ $\quad E' \to +TE'$

$\Rightarrow id + FT'E'$ $\quad T \to FT'$

$\Rightarrow id + id\,T'E'$ $\quad F \to id$

$\Rightarrow id + id\,E'$ $\quad T' \to \varepsilon$

$\Rightarrow id + id$ $\quad E' \to \varepsilon$

$w = id * id$

$E \to \boxed{T} E'$

$\to \boxed{F} T' E'$

$\to id \boxed{T'} E'$

$\to id * \boxed{F} T' E'$

$\to id * id \boxed{T'} E'$

$\to id * id E'$

$\to id * id$

$w = id + id * id$

$E \to \boxed{T} E'$

$\to \boxed{F} T' E'$

$\to id \boxed{T'} E'$

$\to id \boxed{E'}$

$\to id + \boxed{T} E'$

$\to id + \boxed{F} T' E'$

$\to id + id \boxed{T'} E'$

$\to id + id * \boxed{F} T' E'$

$\to id + id * id \boxed{T'} E'$

$\to id + id * id E'$

$\to id + id * id$

$w = (id + id) * id$

$E \to \boxed{T} E'$

$\to \boxed{F} T' E'$

$\to (\boxed{E}) T' E'$

$\to (\boxed{T} E') T' E'$

$\to (\boxed{F} T' E') T' E'$

$\to (id \boxed{T'} E') T' E'$

$\to (id \boxed{E'}) T' E'$

$\to (id + \boxed{T} E') T' E'$

$\to (id + \boxed{F} T' E') T' E'$

$\to (id + id \boxed{T'} E') T' E'$

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

$\to (id + id) \boxed{T'} E'$

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

$\to (id + id) * id \boxed{T'} E'$

$\to (id + id) * id \boxed{E'}$

$\to (id + id) * id$

HW

1) $S \to is|iSeS|a$ $\longrightarrow$ $S \to isS'|a$, $S' \to eS|\varepsilon$

$w1 = iaea$

$w2 = iiaea$

$S \to isS'|a$

$\to iSS'$

$\to iaS'$

$\to iaeS$

$\to iaea$

$S \to iSS'$

$\to iiSS'S'$

$\to iiaS'S'$

$\to iiaeSS'$

$\to iiaeaS'$

$\to iiaea$

Lex Tool

It is used to generate lexical analyzer.

for lex compiler give input lex specification & give output lexical analyzer

lex specification → [lex compiler] → lexical analyzer

tokens

spec.l → [lex / FLEX] → lex.yy.c

.l extension

lex.yy.c → [cc] → a.out

ex Specification — Declaration section — Auxiliary declaration

Auxiliary function

Transition rules

Regular expression definition or declaration

Auxiliary function → Diff function used by Transition rules

Auxiliary declaration → Just like c declaration which going to use in c

Reg. exp declaration

l [a..zA..Z]

↓

Stands for only one symbol

d [0...9]

↓

Only one symbol

id {l}({l}|{d})* → identifier

%{ int a = 10; %}

↓

Auxiliary declaration

Transition rules are enclosed in double %. symbols. Inside we can give pattern followed by action.

%%

pattern Action

%%

whatever defined put in curly braces
what not difined we can give in
[ ]

%%

{id} { printf "%s, is an id",
yytext); }
↙
Global variable
declared by
lex Compiler

%%

Auxiliary functions
↳ Simple c functions
↳ whatever functions written in
this copied to lex.yy.c

Transition rules

%% %%

{id} {printf("welcome");
printfun(yytext); }

%% %%

Auxiliary function
void printfun(char *str)
{ printf("%s is identifier", str);
}

Calling lex compiler

```
int main()
{
    yylex();
}
```

Declared
Transition
Auxiliary
↓
Specification.l

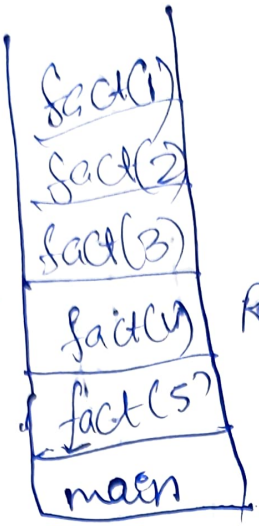
↳ If pattern matches only printed & function called

All programming languages may not support recursion.

Recursion uses internal stack.

Memory → some portion → Program
→ Dynamic memory allocation (Run time stack)
→ RAM

fact(5)
5 × fact(4)
4 × fact(3)
3 × fact(2)
2 × fact(1)



Run-time Stack

what is actually created when function called

Activation Record

↙ Return address of calling function

↘ local parameter

like this we can maintain our own stack.

Predictive Parsing technique (Non-Recursive Recursive descent parsing technique) (LL(1) parser) (Tabular implementation of Recursive descent parser)

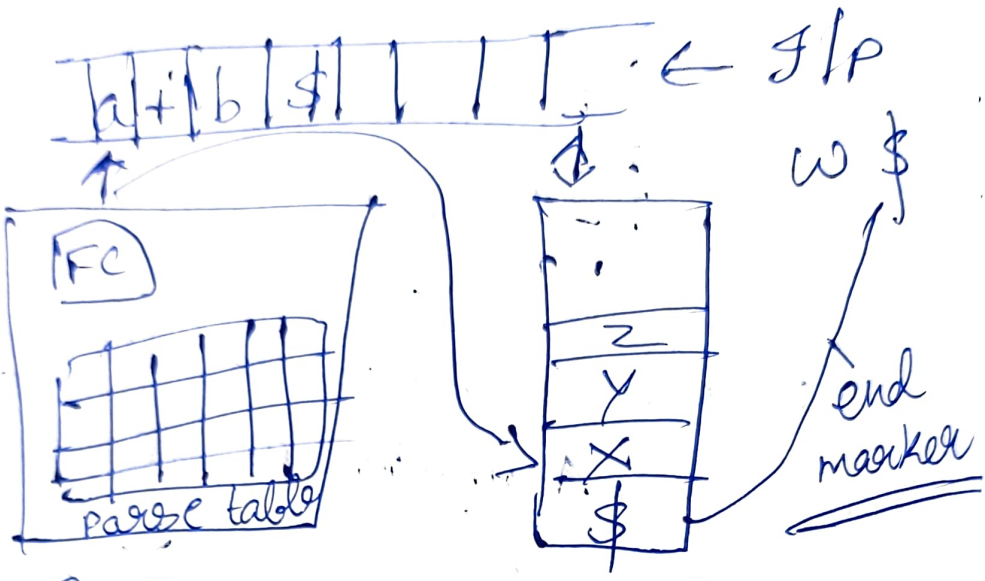
1) a - function
   first(x)
   follow(A)

2) Build parse tree

3) Parsing Algorithm

Model

Input tape divided into cells



finite state control

Pushdown State Stack

Two pointers → one to top of stack
↳ tape start

PDA → machine that can represent all strings that follows CFG grammar

X, Y, Z → grammar symbol

Input tape contains string (w). (i.e all terminals)

Input tape ends with $

w$ is contained in input tape

Parse table denoted by M

- Rows are non terminal
- Columns are terminals
- M[A, a] (Every cell)
- It contains production of form

$$A \rightarrow \alpha$$

a → current input symbol to be matched with one of alternative α

α begins with a then only expand $A \rightarrow \alpha$

Sometimes have production $A \rightarrow \varepsilon$

a is follow(A) then only A can be expanded.

First(α): The set of all terminals which are that begin string derived from α.

Rules to find First(X) → Grammar symbol (Terminal or non terminal)

① If X is a 'T' then add X to first(X)

(non terminal)

② If X is a nt and $X \to a\alpha$ then add a to first(X)

or if $X \to \varepsilon$ then add 'ε' to first(X)

③ If X is non terminal and

$X \to Y_1 Y_2 \ldots Y_k$ then add

non ε-symbols first(y_i) to first(X) iff first(y_{i-1}) contains ε and if all y_i contains 'ε' then add 'ε' to first(X)

Ex: $F \to (E) / id$

① First(F) = {(, id}

set

② $E' \to +TE' / \varepsilon$

First(E') = {+, ε}

③ $T' \to *FT' / \varepsilon$

First(T') = {*, ε}

$T \rightarrow FT'$

$first(T) = first(F)$ → As it doesn't contain ε so stop

$= \{(, id\}$

$E \rightarrow TE'$

$first(E) = first(T)$

$= \{(, id\}$

---

for eg

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

$first(F) = \{(, id, \varepsilon\}$

$T' \rightarrow *FT' / \varepsilon$

$first(T') = \{*, \varepsilon\}$

$T \rightarrow FT'$

$First(T) = first(F)$ → As it Contains ε so for first(T') also

$= \{(, id, *, \varepsilon\}$

If $T' \rightarrow *FT'$

$first(T') = \{*\}$

$T \rightarrow FT'$

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

$S \to iSS' \mid a$

$S' \to eS \mid \varepsilon$

$First(S) = \{i, a\}$

$First(S') = \{e, \varepsilon\}$

$A \to AB \mid AC \mid c \mid d$

$B \to b$

$C \to c$

$A \to cA' \mid dA'$

$A' \to BA' \mid cA' \mid \varepsilon$

$B \to b$

$C \to c$

$First(C) = \{c\}$

$First(B) = \{b\}$

$First(A') = \{b, c, \varepsilon\}$

$First(A) = \{c, d\}$

HW

1) $R \to R \&\& R \mid R \,||\, R \mid !R \mid t \mid f$

2) $S \to S \,||\, T \mid T$

$T \to T \&\& R \mid R$

$R \to !S \mid (S) \mid t \mid f$

1) $R \to !RR' \mid tR' \mid fR'$

$R' \to ||RR' \mid \varepsilon$

$first(R) = \{!, t, f\}$

$first(R') = \{||, \varepsilon\}$

2) $S \to TS'$

$S' \to ||TS' \mid \varepsilon$

$T \to RT'$

$T' \to \&\&RT' \mid \varepsilon$

$R \to !S \mid (S) \mid t \mid f$

$first(S) = first(T) = \{!, (, t, f\}$

$first(S') = \{||, \varepsilon\}$

$first(T) = first(R) = \{!, (, t, f\}$

$first(T') = \{\&\&, \varepsilon\}$

$first(R) = \{!, (, t, f\}$

When R contains ε, go for T'. If T' also contains ε then only include ε otherwise not.

1) $E \to E+E \mid E*E \mid (E) \mid id$

$A \to A\alpha \mid \beta$

```
Struct prodS
    char left
    char right[10][10];
     int noa;
};P S
```

```
p.left = A
p.right[0] = Aα
      [1] = β
p.noa = 2
```

```
for(i...)
p.left = = p.right[i][0]
```

2) $E \to E+T \mid T$

```
p.left = E
p.right[0] = E+T | Aα
right[1] = T | β
p.noa = 2
```

```
q[0].left = p.left
q[0].right[0] = TU
q[0].noa = 1
q[1].left = U
q[1].right[0] = +TU
      .right[1] = e
      .noa = 2
```

3) $G \rightarrow G+G \mid G * (G) \mid id$

q[0].left = E

q[0] = q.right[0] = (G)

[1] = id

q[1].left = U

q[1].right[0] = + E U

right[1] = * E U

[2] = e

.noa = 3

$S \rightarrow TS'$

$S' \rightarrow || TS'$

$T \rightarrow RT'$

$T' \rightarrow \&\& RT'$

$R \rightarrow ! S \mid (S) \mid t \mid f \mid \varepsilon$

First

	{!, (, t, f, &&}
S	{!, (, t, f, &&}
S'	{||, ε}
T	{!, (, t, f, &&}
T'	{&&, ε}
R	{!, (, t, f, ε}

As here T' didn't contain ε so didn't include ε even though R included ε.

Rules to find follow(B)

1) Put '$' in follow(B) if B is start symbol
2) If $A \to \alpha B \beta$ ($\beta \overset{*}{\not\Rightarrow} \varepsilon$)
   then add first(β) to follow(B)
3) If production of form $A \to \alpha B$ ($\beta \overset{*}{\Rightarrow} \varepsilon$)
   then add follow(A) to follow(B)

Ex: Aa
$\alpha B a$ → follow(B) is a
follow(A) is a

Follow(B): The set of terminals that appear immediately to right of B in grammar G.

Ex: BA
Follow(B) = a.

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

follow(E) = { $,) }

follow(E') = { $,) }

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

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

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

$S \rightarrow iSS'|a$

$S' \rightarrow eS|\varepsilon$

follow(S) : { $, e }

follow(S') : { $, e }

1) $R \rightarrow R\&\&R|R||R|!R|t|f$

$R \rightarrow !RR'|tR'|fR'$

$R' \rightarrow \&\&RR'||RR'|\varepsilon$

follow(R) : { $, &&, || }

follow(R') : { $, &&, || }

2) $S \rightarrow S||T|T$

$T \rightarrow T\&\&R|R$

$R \rightarrow !S|(S)|t|f$

$\Rightarrow S \rightarrow TS'$

$S' \rightarrow ||TS'|\varepsilon$

$T \rightarrow RT'$

$T' \rightarrow \&\&RT'$

$R \rightarrow !S|(S)|t|f$

follow(S) : { $,), &&, || }

follow(S') : { $,), dd, ;; }

follow(T) : { [illegible], $,), dd, ;; }

follow(T') : { $,), dd, ;; }

follow(R) : { dd, $,), ;; }

	follow(B)
S	{ $,), f(R) } = { $,), ;;, dd }
S'	{ $,), f(R) } = { $,), ;;, dd }
T	{ ;;, $,), f(R) } = { ;;, $,), dd }
T'	{ ;;, $,), f(R) } = { dd, ;;, $,) }
R	{ dd, ;;, $,), f(R) } = { dd, ;;, $,) }

First → Alternative first symbol which matches

Follow → If first symbol → ε use follow

Filling up parse Table

If every production A → α in G do the following:

1) If $\alpha \not\Rightarrow^* \varepsilon$ then set $M[A, a] = A \to \alpha$ where 'a' is first(α)

2) If $\alpha \Rightarrow^* \varepsilon$ then set $M[A, b] = A \to \varepsilon$ where b is follow(A)

3) Make all undefined entries as error entries

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

(4) If all entries in parse table are unique entries then the grammar is called as LL1 grammar

L → left to right scanning

L → left most derivation

	first()	follow()
E	{(, id}	{$,)}
E'	{+, ε}	{$,)}
T	{(, id}	{+, $,)}
T'	{*, ε}	{+, $,)}
F	{(, id}	{*, +, $,)}

M[A, a]

↓ non terminal — Top of stack

↳ current i/p symbol at i/p buffer

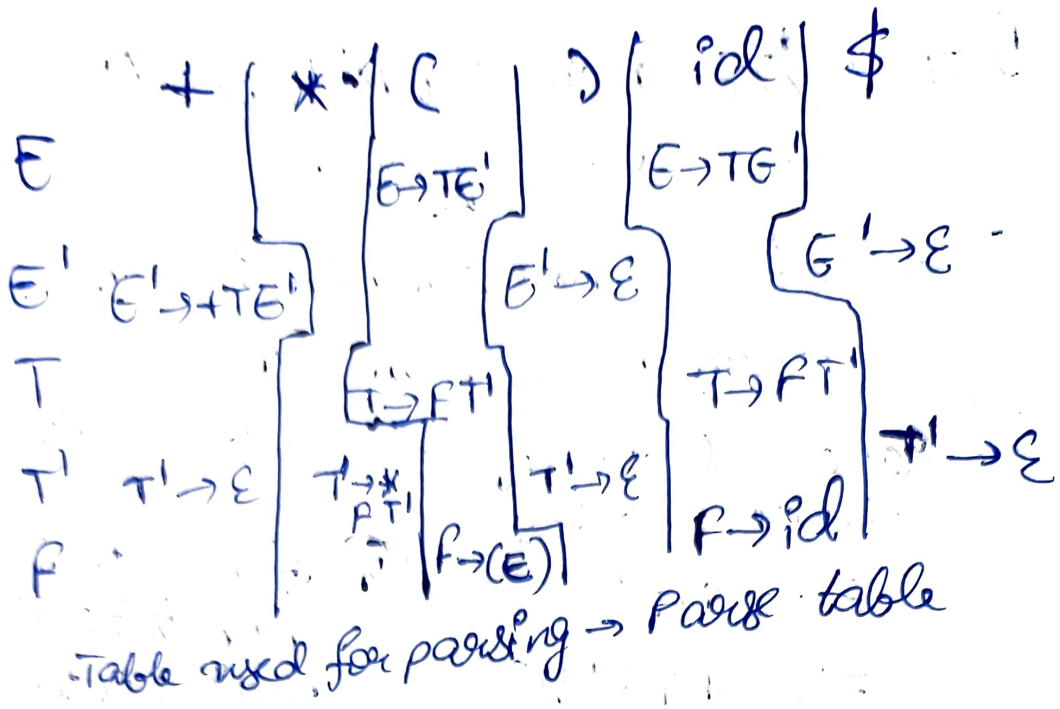


Table used for parsing → Parse table

Construct for

$S \to iS \mid iSeS \mid a$

$S \to iSS' \mid a$

$S' \to eS \mid \varepsilon$

	first()	follow()
S	{i, a}	{$, e}
S'	{e, ε}	{$, e}

	i	e	a	$
S	S→iSS'	⊗	S→a	⊗
S'	⊗	S'→eS S'→ε	⊗	S'→ε

since there are 2 entries for S' on e so not unique so not LL1

$S \to S\,||\,R \mid R$

$R \to R\,\&\&\,T \mid T$

$T \to !S \mid (S) \mid t \mid f$

$S \to RS'$

$S' \to ||\,RS' \mid \varepsilon$

$R \to TR'$

$R' \to \&\&\,TR' \mid \varepsilon$

$T \to !S \mid (S) \mid t \mid f$

	First()	Follow()
S		
S'		
R		
R'		
T		