28 Apr 2025 - Compilers - I - Week 05

Context - Free Grammars, Pushdown Automata and Parsing

Outline

→ Syntax analysis

→ Context-free grammars ← basis of specification of programming languages ??

→ Parsing CFG ⇝ pushdown automata

→ Parsing → Top-down → LL(1)
→ Recursive-descent

Parsing → Bottom-up → LR

tool YACC → parses construction

Grammars

→ used for precise description of rules

e.g.: Rules stating how functions are made out of parameter lists, declarations and statements.

↙ expressions, etc.

→ Certain types of grammars

↓

parsers can be automatically constructed from the grammar

→ Parsers / Syntax Analysers are generated for a particular grammar

input grammar → YACC → output parser

→ CFG → <u>subclass</u> of languages

- regular
- CFG
- context-sensitive
- type-0 etc.

CFGs specify context-free languages

Char stream → Lexical analyser →

→ Parser

→ verifies that the string of tokens for a program in that language can be generated from that grammar

→ reports syntax errors

→ construct parse tree representation ← not always necessary

→ usually calls lexical analyzer to supply tokens when necessary.
??

→ handwritten or automatically generated.

→ CFG

$G = (N, T, P, S)$

- N → finite set of non-terminals
- T → finite set of terminals → tokens of lexical analyses
- P → finite set of production/rules
- S → start symbol $\in N$

$A \rightarrow \alpha$ $A \in N$, $\alpha \in (N \cup T)^*$

Context free languages ≡ Regular Languages

Context free languages ↕ Parser — Pushdown automata

Regular Languages ↕ Lexical Analyzer — Finite State Automata e.g

start (implicit)

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

$S \rightarrow aSb$

$S \rightarrow \varepsilon$

loosely, $\varepsilon \in T$

start

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA$

$B \rightarrow b \mid bS \mid aBB$

equivalent

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

Derivations

productions

$$\underbrace{E}_{\text{start symbol}} \xRightarrow{E \rightarrow E + E} E + E \xRightarrow{E \rightarrow id} id + E$$

$$\Downarrow E \rightarrow id$$

$$\underbrace{id + id}_{\text{terminal string}}$$

if $id, + \} \in T$

Which production to choose depends on what you want to derive

Short : $E \underbrace{\Longrightarrow^{*}}_{\text{0 or more steps}} id + id$

E $\overline{\text{derives}}$ id + id

Context - free Languages

$$\text{CFGs} \underset{\text{specified by}}{\overset{\text{generate}}{\rightleftarrows}} \text{CFLs}$$

$$L(G) = \{ w \mid w \in T^{*} \text{ and } S \Rightarrow^{*} w \}$$

all strings derivable from start symbol and consist only of terminals

$$\left.\begin{array}{l} S \to 0S0 \\ S \to 1S1 \\ S \to 1 \\ S \to \varepsilon \end{array}\right\} \qquad L(G) = \text{palindromes over } 0 \text{ and } 1$$

Sentential form

A string $\alpha \in (N \cup T)^*$

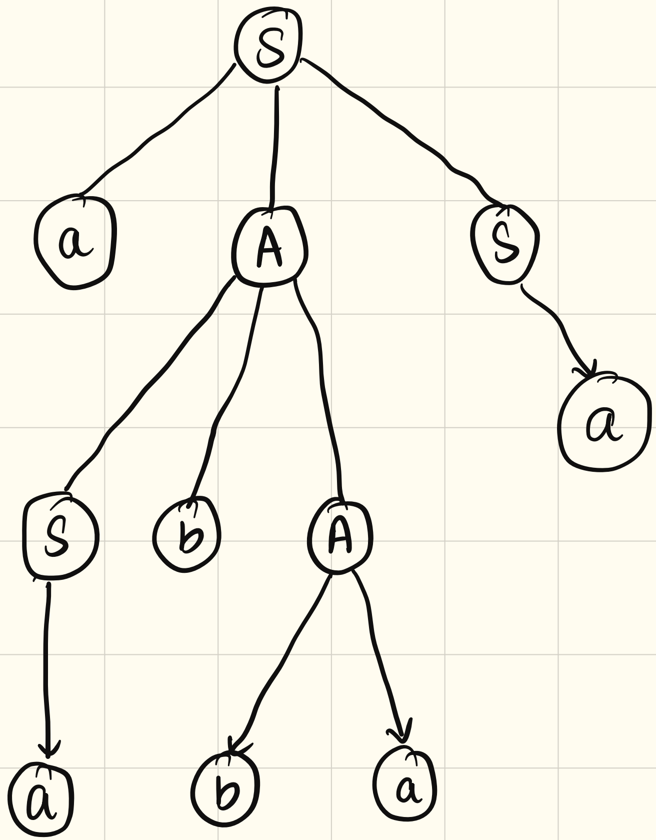
is a sentential form if $S \overset{*}{\Longrightarrow} \alpha$

sentence $\longrightarrow$ only terminals

$$G_1 \equiv G_2 \iff L(G_1) = L(G_2)$$

Derivation Tree example

$$S \to aAS \mid a$$
$$A \to SbA \mid SS \mid ba$$



leaves = terminals

non-terminals = internal nodes

$$S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aabAS \Rightarrow aabbaS$$
$$\Rightarrow \underbrace{aabbaa}_{\text{yield of derivation tree}}$$

Leftmost and Rightmost derivation

- Leftmost → expand left-most non-terminal first in each step
- Rightmost → expand right-most non-terminal first in each step

→ if some $w \in L(G)$ has more than one parse tree → G is **ambiguous**

if not ambiguous → unique leftmost and rightmost derivations

→ A CFL for which every G is ambiguous → inherently ambigous language

Ambigous Grammars

If the grammar is ambiguous → tools for generating parsers and parsing techniques will be in trouble.

↓ 2 parse trees for same grammar, same word

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

→ + and * have same precedence

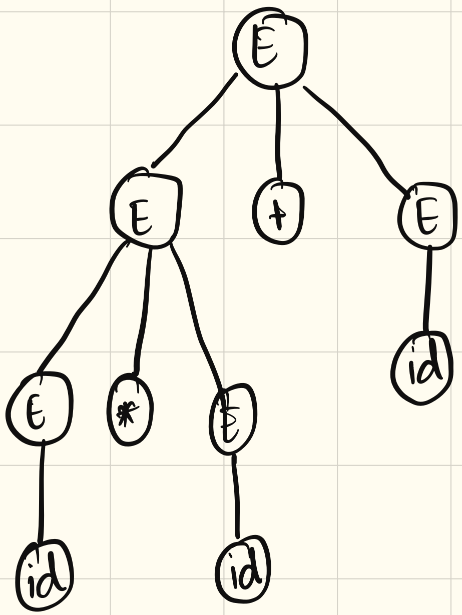
↕ same language

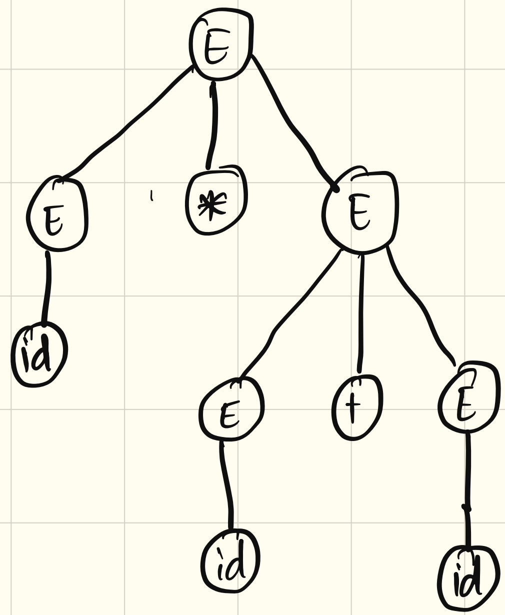
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

* has more precedence than +

Example 1: id * id + id

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

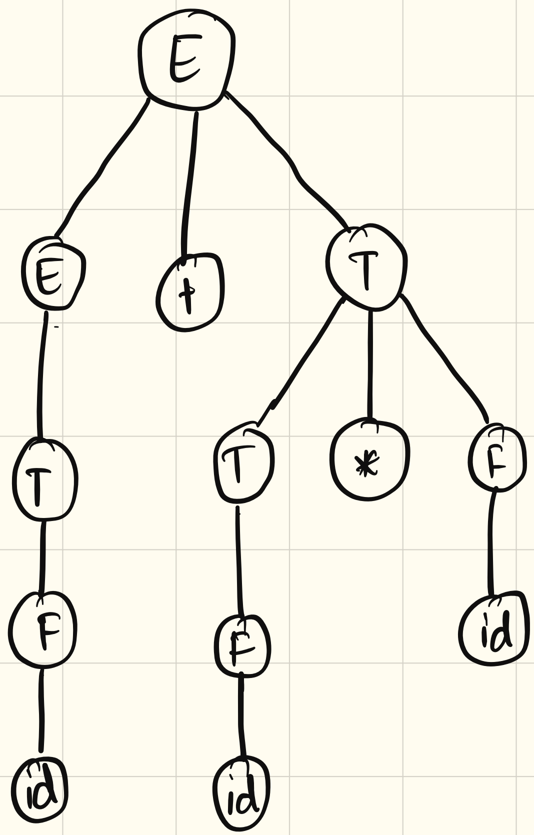




Equivalent Unambiguous Grammar

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

$id + id * id$



leftmost derivation (unique)

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

Trying * first   Leftmost derivation

$$E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow (E) * F \Rightarrow (E + T) * F$$

cannot go to E
no such rule

$$\Downarrow$$

$$(T + T) * F$$

$$\Downarrow$$

$$(id + F) * F \Leftarrow (id + T) * F \Leftarrow (F + T) * F$$

$$\Downarrow$$

$$(id + id) * F \Rightarrow \underline{(id + id) * id}$$

Ambiguity Example 2

stmt → IF expr stmt | IF expr stmt ELSE stmt | other_stmt

ambiguous

stmt → IF expr stmt | IF expr matched_stmt ELSE stmt

matched_stmt → IF expr matched_stmt ELSE matched_stmt | other_stmt

equivalent unambiguous (example later)

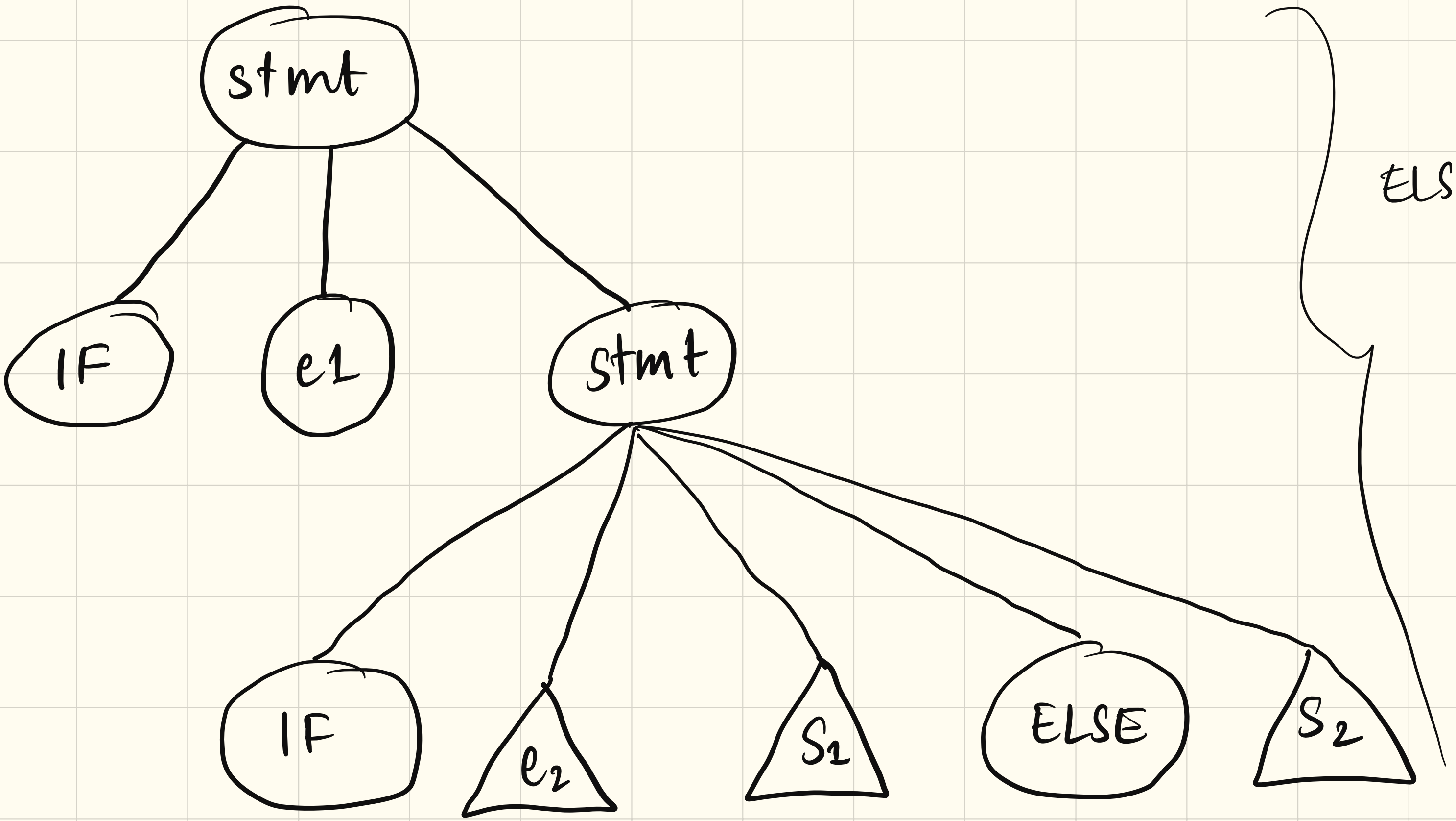
* No algorithm exists → to convert ambiguous grammars to unambiguous

$$L = \{ a^n b^n c^m d^m \mid n, m \geq 1 \} \cup \{ a^n b^m c^m d^n \mid n, m \geq 1 \}$$

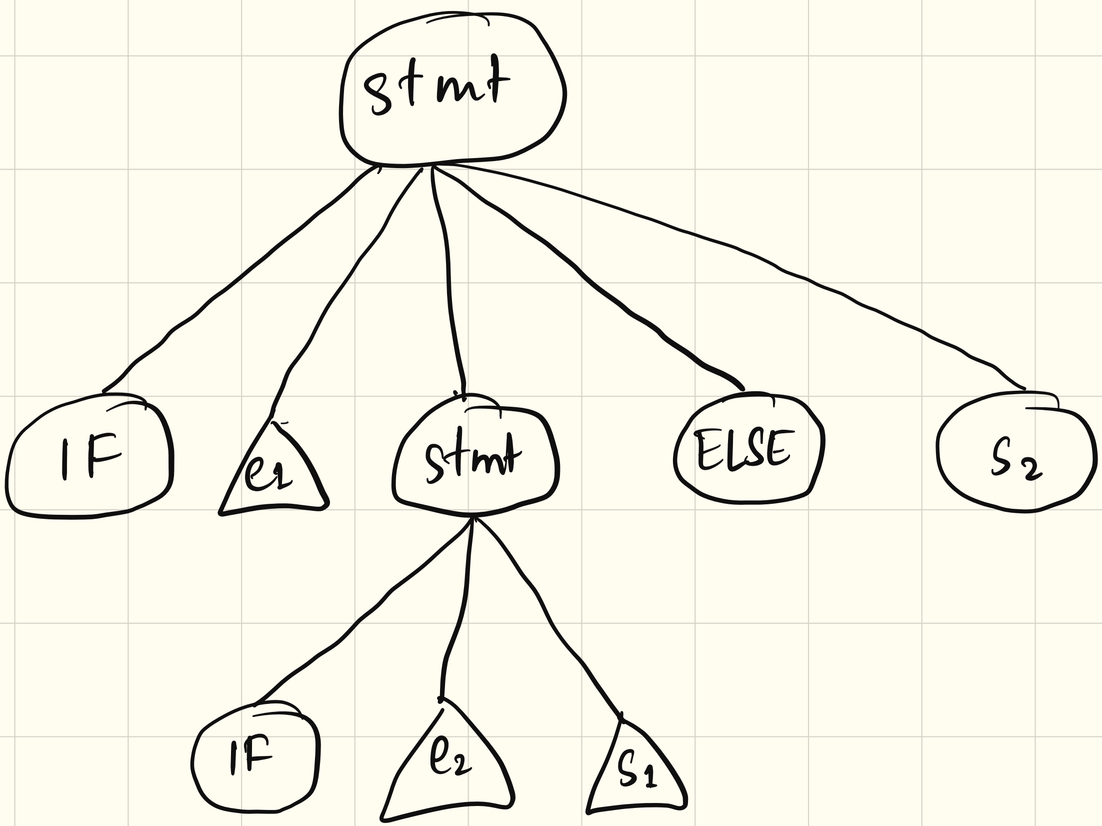
inherently ambiguous
no grammar that
is not ambigous

Example 2 : Two parse trees for the sentence

IF e1 IF e2 s1 ELSE s2



ELSE corresponds to inner if



ELSE corresponds to outer if

$S \rightarrow \text{IF} \;\; e \;\; S \;\; | \;\; \text{IF} \;\; e \;\; ms \;\; \text{ELSE} \;\; S$

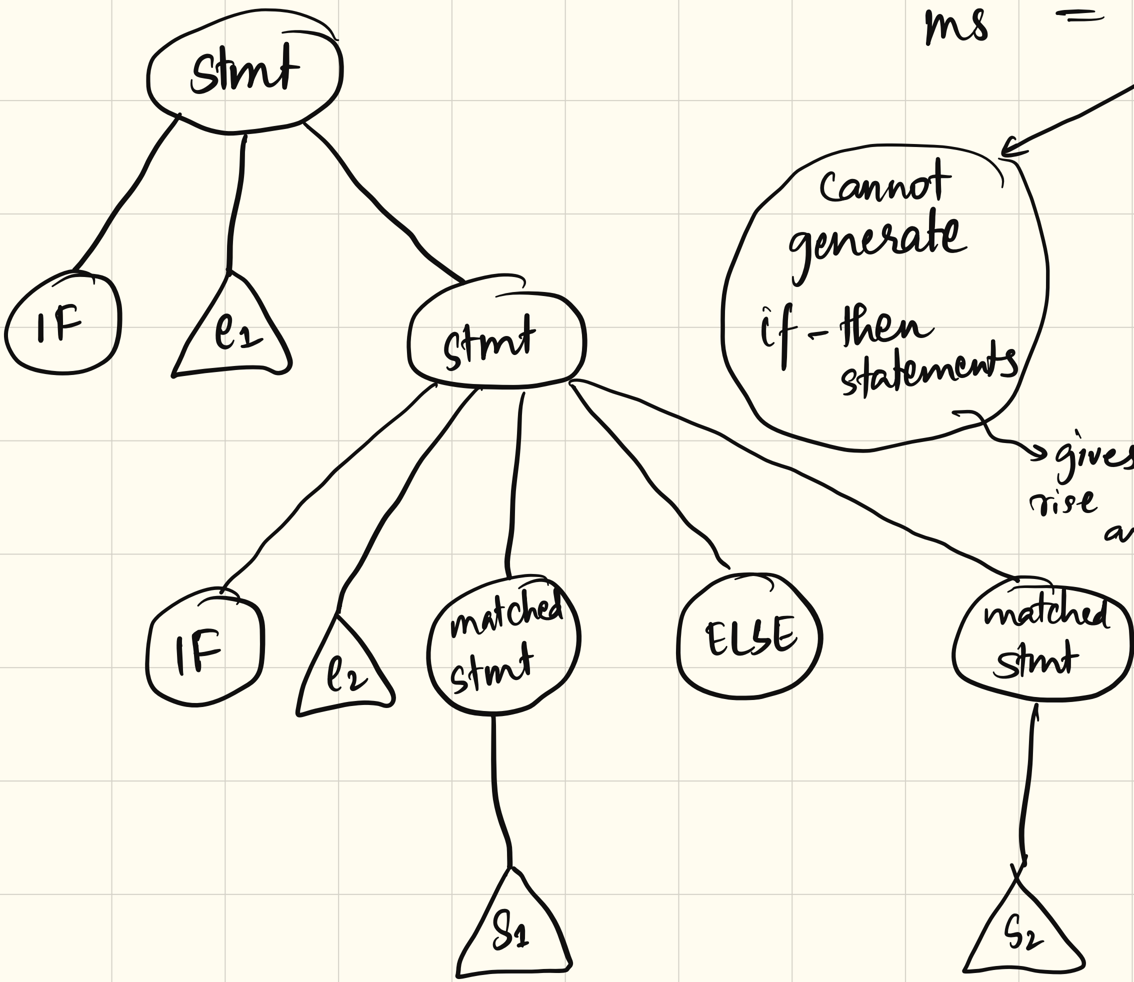
$\underline{ms} \rightarrow \text{IF} \;\; e \;\; ms \;\; \text{ELSE} \;\; ms \;\; | \;\; other\_s$

ms = matched statement

cannot generate if-then statements

gives rise to ambiguity

always generates an if-then-else matched statement



Fragment of C grammar (Expressions)

??

logical_or_expr → logical_and_exp
| logical_or_expr OR_OP logical_and_expr

precedence { OR is at lower level

AND is at higher level

logical_and_expr → equality_expr
| logical_and_expr AND_OP equality expression

== > &&

lower priority ⟶ higher priority

| lower priority (operation) higher priority

Pushdown Automata

Machine → can be automatically derived with a given grammar

→ stack based system

PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$

- Q → finite set of states
- Σ → input alphabet
- Γ → stack alphabet
- δ → transition fn
- q_0 → start state
- z_0 → start symbol on stack
- F → set of final states $F \subseteq Q$

$$\delta : Q \times \underbrace{\Sigma \cup \{\varepsilon\}}_{\text{can be } \varepsilon} \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma^*)$$

can be ε

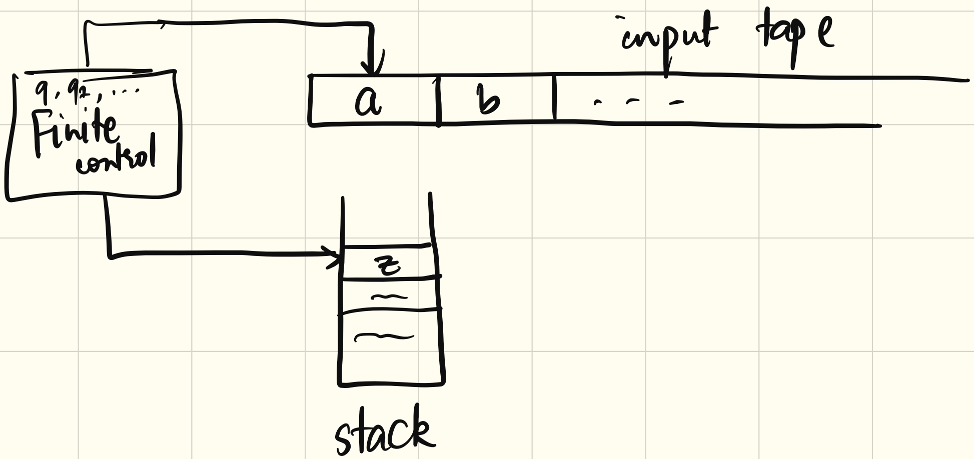
$$\delta(q, a, z) = \underbrace{\{ (q_1, \gamma_1), (q_2, \gamma_2), \dots, (q_m, \gamma_m) \}}$$

a: input symbol

z: top of stack symbol

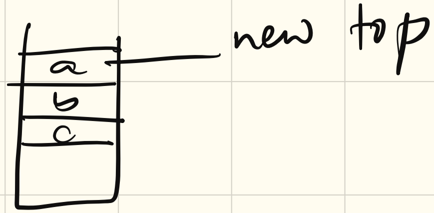
PDA can enter any of the states q_i and replace the symbol z by string γ_i

Non-deterministic control



→ leftmost symbol of γ_i will be new top of stack

$\gamma_i = abc$



$L(M)$ ⇝ language accepted by M <u>by final state</u>

series of moves

$$L(M) := \{ w \mid (\overset{\text{start}}{q_0}, \overset{\text{input string}}{w}, \overset{\text{stack start}}{Z_0}) \vdash^* (p, \varepsilon, \gamma)$$

$$\text{for some } p \in F \text{ and } \gamma \in \Gamma^* \}$$

$N(M) \rightsquigarrow$ language accepted by M by empty stack

F becomes irrelevant
we usually set $F = \phi$
here

$N(M) = \{ w \mid (q_0, w, Z_0) \vdash^* (p, \varepsilon, \varepsilon)$ for some $p \in Q \}$

Part 2

---

reverse

$L = \{ ww^R \mid w \in \{a, b\}^+ \}$

non-deterministically guess middle of input

Nondeterministic and deterministic PDA

$\underbrace{NFA \equiv DFA}_{\text{accept the same set of languages}}$ finite automata

But

NPDA is more powerful than DPDA

$\{ww^R \ldots\}$ ×

$\{wcw^R\}$ → can be recognized by DPDA

In practice we need DPDA, since they have exactly one possible move at any instant.

→ Our parsers all are DPDA.

Parsing

Parsing is the process of constructing a parse tree for a sentence generated by a given grammar.

using a DPDA
+ some actions

no restrictions on language and the form of grammar $\Longrightarrow$ parsers for CFLs require $O(n^3)$ time (n = length of the string parsed)

* CYK algorithm $\longrightarrow$ DP
* Earley's algo

$\underbrace{\text{Subsets of CFLs}}_{\text{we are interested in these}}$ $\longrightarrow$ $O(n)$ time

① Predictive parsing $\longrightarrow$ on class of grammars called LL(1) grammars

uses top-down parsing

② Shift - Reduce parsing

using LR(1) grammers (bottom - up)

Top-down parsing using LL grammars

→ Basic idea: trace the leftmost derivation of the string while constructing the parse tree

$S \rightarrow aAS \mid c$

$A \rightarrow ba \mid SB$

$B \rightarrow bA \mid S$

consider acbbac

Leftmost derivation

$$\underset{1}{S} \Rightarrow \underset{2}{aAS} \Rightarrow \underset{3}{aSBS} \Rightarrow \underset{4}{acBS} \Rightarrow \underset{5}{acbAS} \Rightarrow \underset{6}{acbbaS} \Rightarrow \underset{7}{acbbac}$$

→ start from the start symbol, predicts the next production used in the derivation. → through parse tables (stored)

→ Next production to be used in the derivation is determined using the next input symbol to lookup the parsing table (look-ahead symbol)

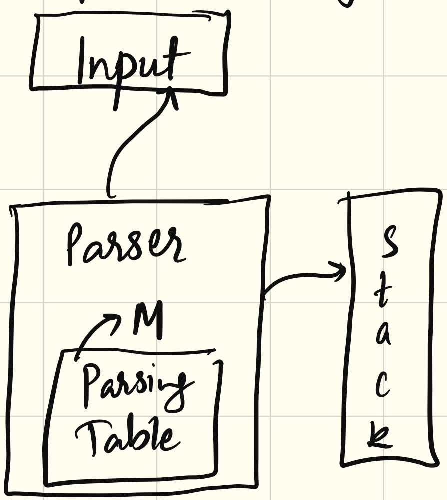
→ Restriction on the grammar (that helps us get O(n) time) $\xrightarrow{\text{ensures}}$ no slot in the parsing table contains more than one production.

→ if more than one, we cannot decide which to use next

Parse table construction

→ if two productions become eligible to be placed in the same slot → grammar is declared unfit for predictive parsing.

LL(1) parsing algorithm



Initial configuration :

stack = S

Input = $w\,\$$

$\$ \equiv$ end of file marker

S = start symbol

```
repeat {
    let X be the top stack symbol;
    let a be the next input symbol          /* may be $ */
    if X is a terminal symbol or $ then
        if X == a then {                    } top of stack
                                            } matches input
            pop X from the stack;
            remove a from input;
        } else ERROR();   } stack symbol and input symbol
                            do not match
    else /* X is a non-terminal symbol */
        if M[X, a] == X → Y1 Y2 ... Yk then {
                                      (single unique production)
            pop X from stack;
            push Yk, Yk-1, ... Y1 onto stack }
                                 (Y1 on top)
```

} until stack has emptied.

Example:

Grammar

$S' \to S\$$

$S \to aAS \mid c$

$A \to ba \mid SB$

$B \to bA \mid S$

construction later

LL(1) Parsing Table

(No slot can have more than 1 entry)

	a	b	c	\$
S'	$S' \to S\$$		$S' \to S\$$	
S	$S \to aAS$		$S \to c$	
A	$A \to SB$	$A \to ba$	$A \to SB$	
B	$B \to S$	$B \to bA$	$B \to S$	

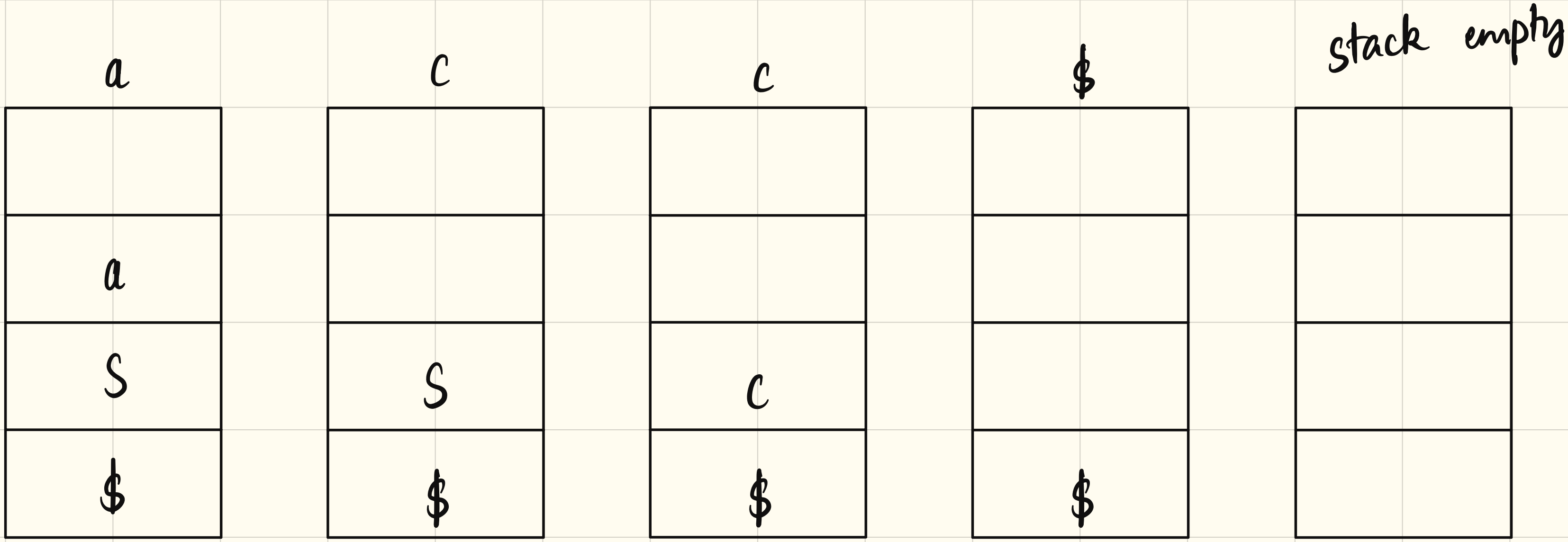
Rows indexed by non-terminals

Columns indexed by terminals

String : a c b b a c

a	a	a	c	c
		a		S
		A	A	B
	S	S	S	S
S'	$	$	$	$

c	b	b	b	b
c		b		b
B	B	A	A	a
S	S	S	S	S
$	$	$	$	$



How is the parsing table constructed ?

LL(1) grammers ⟶ sub class of CFGs
some restrictions

Strong LL(k) grammars

Let the given grammar be G.

→ Input is extended with k EOF symbols $\$^k$

k → lookahead of the grammar

$k = 1$ → allowed to see 1 symbol at a time

$k = 2$ → allowed to see 2 symbols at a time

New non-terminal S' and production : $S' \rightarrow S\$^k$

→ Consider leftmost derivations only

assume grammar has no useless symbols

(terminals / non-terminals that are never used)

1. not part of any production

OR

2. LHS of a production can never be reached

such productions and corresponding symbols are useless

→ A production $A \to \alpha$ in G is called a strong LL(k) production, if in G

Can we look at string z at some point, and determine whether $A \to \alpha$ was applied, or $A \to \beta$??

2 productions in G

$$S' \Rightarrow^* \underbrace{w}_{\text{can be } \varepsilon} A\gamma \Rightarrow w\alpha\gamma \Rightarrow^* w\overbrace{z}^{\text{terminals}}y$$

not necessary

$\cancel{\alpha \Rightarrow^* z}$

$\cancel{\gamma \Rightarrow^* y}$

k symbols in z

$\boxed{OR}$

$A \to \beta$

$$S' \Rightarrow^* w'A\delta \Rightarrow w'\beta\delta \Rightarrow^* w'zx$$

$|z| = k$, $z \in \Sigma^*$. w and $w' \in \Sigma^*$, then $\alpha = \beta$

$\gamma \neq \delta$

z = lookahead

Strong LL(k) condition: If the lookahead (z) is same at some point, then we know exactly which production was applied at a point

* A grammar (non-terminal) is strong LL(k) if all its productions are strong LL(k).

$$\alpha\gamma \overset{*}{\Rightarrow} \underline{z}y$$
$$\beta\delta \overset{*}{\Rightarrow} \underline{z}x$$

forces α and β to be same } strong LL(k) grammar e.g:

$$S \rightarrow Abc \mid aAcb$$

$$A \rightarrow \varepsilon \mid b \mid c$$

S is a strong $LL(1)$ non-terminal
$k = 1$

$*$ $S' \Rightarrow S\$ \Rightarrow \underbrace{Abc}_{\alpha}\$ \Rightarrow$ $bc\$$ or $bbc\$$ or $cbc\$$

$A \rightarrow \varepsilon$, $A \rightarrow b$, $A \rightarrow c$

$z = b$, $z = b$, $z = c$

$w = \varepsilon$ here

1^{st} symbol = lookahead = z
$(k=1)$

$*$ $S' \Rightarrow S\$ \Rightarrow \underbrace{aAcb}_{\beta} \Rightarrow$ $acb\$$ or $abcb\$$ or $accb\$$

$A \rightarrow \varepsilon$, $A \rightarrow b$, $A \rightarrow c$

$z = a$ in all cases

In this case, $w = w' = \varepsilon$,

$\alpha = Abc$

$\beta = aAcb$

but z is different in the two derivations,
in all the derived strings

$\therefore$ vacously true $\longrightarrow$ LL(1) ✓

$$\left.\begin{array}{l} S \rightarrow Abc \mid aAcb \\ A \rightarrow \varepsilon \mid b \mid c \end{array}\right\}$$ A is **NOT** strong LL(1)

$S' \Rightarrow^* Abc\$ \Rightarrow bc\$ \quad A \rightarrow \underbrace{\varepsilon}_{}^{\alpha} \quad z = b$

$bbc\$ \quad A \rightarrow \overset{\beta}{b} \quad z = b$

$\left.\begin{matrix} z = b \\ z = b \end{matrix}\right\} \rightarrow \therefore$ Not LL(1)

$cbc\$$

$w = \varepsilon$
$w' = \varepsilon$

Even though the lookaheads are same $(z = b)$, $\alpha \neq \beta$, $\therefore$ grammar is not strong LL(1).

A is not strong LL(2)

$w = \varepsilon, \; w' = a$

$S' \Rightarrow^* Abc\$ \Rightarrow \underline{bc}\$ \quad A \rightarrow \varepsilon$

$S' \Rightarrow^* aAcb\$ \Rightarrow a\underline{bc}b\$ \quad A \rightarrow b$

$\left.\begin{matrix} A \rightarrow \varepsilon \\ A \rightarrow b \end{matrix}\right\} z = bc$

$\therefore$ A is not LL(2)

A is strong $LL(3)$

$$S' \Rightarrow^* Abc\$ \Rightarrow \overline{bc\$}$$

$$\xRightarrow{A \to b} \overline{bbc}\,\$$$

$$\xRightarrow{A \to c} \overline{cbc}\,\$$$

$$S' \Rightarrow^* aAcb\$ \Rightarrow a\,|\,\overline{cb\$}$$

$$\Rightarrow a\,|\,\overline{bcb}$$

$$\Rightarrow a\,|\,\overline{ccb}$$

all z are different

vacuously true

$A \to \alpha$

$A \to \beta$

$$S' \Rightarrow^* wA\gamma \Rightarrow^* wzy$$

$$S' \Rightarrow^* w'A\delta \Rightarrow^* w'zx$$

if A is $LL(|z|)$ nonterminal $\begin{cases} \text{same} \\ \Downarrow \\ \alpha = \beta \end{cases}$

Testable conditions for LL(1)

→ Lookaheads longer than 1 will not be considered from now on

→ ordinary weak LL(1)
strong LL(1)
→ for lookahead = 1
strong = weak
for others (LL(2), strong ≠ weak)

The classical condition for LL(1) property uses FIRST and FOLLOW sets

def^n → cond^n
next

FIRST

If α is any string of grammar symbols

$\alpha \in (N \cup T)^*$

$$\text{FIRST}(\alpha) = \{ a \mid a \in T \text{ and } \alpha \Rightarrow^* ax , x \in T^* \}$$

① collect all (terminal) strings (sentence) derivable from α

② add first symbol of each string to FIRST(α)

By defn : $\text{FIRST}(\epsilon) = \{\epsilon\}$

FOLLOW

defined only for a non-terminal

FIRST

$(N \cup T)^*$

If A is a nonterminal, then

$$\text{FOLLOW}(A) = \{ a \mid S \Rightarrow^* \alpha A a \beta,\ \alpha, \beta \in (N \cup T)^*$$
$$a \in T \cup \{\$\} \}$$

① Start with S, derive all sentential in which A appears

② take the first terminal that appears after A. add it to FOLLOW(A)

requires a context

If $\alpha A a \beta$ cannot be reached from S

a not included in FOLLOW(A)

Example:

$$S' \to S\$$$

$$S \to aAS \mid c$$

$$A \to ba \mid SB$$

$$B \to bA \mid S$$

* $\text{FIRST}(S') = \text{FIRST}(S) = \{a, c\}$ because

$\text{FIRST}(\alpha) = \{ a \mid a \in T \text{ and } \alpha \Rightarrow^* ax, \; x \in T^* \}$

$$S' \Rightarrow S\$ \Rightarrow \underline{c}\$$$

$$S' \Rightarrow S\$ \Rightarrow \underbrace{\underline{a}AS\$ \Rightarrow \underline{a}baS\$ \Rightarrow \underline{a}bac\$}$$

NOTE: Do the complete derivation

$\alpha \Rightarrow^* ax, \; x \in T^*$ } check if terminal string possible

* $\text{FIRST}(A) = \{a, b, c\}$ because

$$A \Rightarrow \underline{b}a$$

$$A \Rightarrow SB$$

$$\therefore \underset{\{a, c\}}{\underset{\parallel}{\text{FIRST}(S)}} \subseteq \text{FIRST}(A)$$

Incomplete method ?

??

* $\text{FOLLOW}(S) = \{a, b, c, \$\}$ because

$$S' \Rightarrow S\underline{\$}$$

$$S' \Rightarrow aAS\$ \Rightarrow aSBS\$ \xRightarrow{B \to bA} aS\underline{b}AS\$$$

$$\xRightarrow{b \to S} a\,SSS\$$$

$$\underline{a}AS \qquad \underline{c}$$

* $\text{FOLLOW}(A) = \{a, c\}$ because

$$S' \Rightarrow^* aAS\$ \Rightarrow aA\underline{a}AS\$$$

$$S' \Rightarrow^* aAS\$ \Rightarrow aA\underline{c}\$$$

Algorithms to compute FIRST

1. terminals and non-terminals
2. general string $\in (N \cup T)^*$

(1) Terminals and nonterminals

{

for each $(a \in T)$ FIRST$(a) = \{a\}$; FIRST$(\varepsilon) = \{\varepsilon\}$; // terminals

for each $(A \in N)$ FIRST$(A) = \phi$; // initialize for nonterminals

// fixed point computation : see also → Ford-Fulkerson Algorithm

while (FIRST sets are still changing) {

for each production p {

Let p be the production $A \rightarrow X_1 X_2 \ldots X_n$

$$\text{FIRST}(A) = \underbrace{\text{FIRST}(A)}_{\text{could have elements from other production}} \cup \left(\underbrace{\text{FIRST}(X_1) - \{\varepsilon\}}\right)$$

// if $X_1 \Rightarrow^* \varepsilon$

```
i = 1 ;
while (ε ∈ FIRST(Xi) && i ≤ n-1) {
    FIRST(A) = FIRST(A) ∪ (FIRST(Xi+1) - {ε}) ;
    i++ ;
}                    // X1 → ε  ⟶  consider X2 ... Xn
                     //  X1 X2 → ε  ⟶  consider next
                     //  and so on

if (i == n && (ε ∈ FIRST(Xn))     // reached end
                                  // no symbol
    FIRST(A) = FIRST(A) ∪ {ε}     // A ⇒* ε ✓

   }
}
```

② FIRST(β) : β, a string of grammar symbols

```
{ /* Assume FIRST(terminal), FIRST(nonterminal) computed */
    FIRST(β) = φ
    while (FIRST sets are still changing) {
        Let β be the string X1 X2 .. Xn
        FIRST(β) = FIRST(β) ∪ (FIRST(X1) - {ε});
        i = 1
        while (ε ∈ FIRST(Xi) && i ≤ n-1) {
            FIRST(β) = FIRST(β) ∪ (FIRST(Xi) - {ε});
            i++;
        }
```

$$\text{if } (i == n \ \ \&\& \ \ (\varepsilon \in \text{FIRST}(X_n))$$

$$\text{FIRST}(\beta) = \text{FIRST}(\beta) \cup \{\varepsilon\}$$

}

<u>Example:</u>

$$S' \rightarrow S\$$$

$$S \rightarrow aAS \mid \varepsilon$$

$$A \rightarrow ba \mid SB$$

$$B \rightarrow cA \mid S$$

Initially: $\text{FIRST}(S) = \text{FIRST}(A) = \text{FIRST}(B) = \phi$

Iteration 1:

* $\text{FIRST}(S) = \{a, \varepsilon\}$ from $S \to aAS \mid \varepsilon$

* $$\text{FIRST}(A) = \{b\} \cup (\underset{a,\,\varepsilon}{\text{FIRST}(S)} - \{\varepsilon\}) \cup \underbrace{(\underset{\phi \text{ atm}}{\text{FIRST}(B)} - \{\varepsilon\})}_{\text{Because } S \to \varepsilon}$$
  $$= \{b, a\}$$

* $$\text{FIRST}(B) = \{c\} \cup (\text{FIRST}(S) - \{\varepsilon\}) \cup \underbrace{\{\varepsilon\}}_{\text{included because } \varepsilon \in \text{FIRST}(S)}$$
  $$= \{a, c, \varepsilon\}$$

Iteration 2 (values stabilize and do not change in iteration 3)

* $FIRST(S) = \{a, \varepsilon\}$ (no change from iteration 1)

* $FIRST(A) = \{b\} \cup (FIRST(S) - \{\varepsilon\}) \cup (FIRST(B) - \{\varepsilon\}) \cup \{\varepsilon\}$

contains ε contains ε

$= \{b, a, c, \varepsilon\}$

* $FIRST(B) = \{c, a, \varepsilon\}$ no change from iteration 1.

Computation of FOLLOW

{

initialize follow of every nonterminal

for each $(X \in N \cup T)$ FOLLOW $(X) = \phi$;

FOLLOW $(S) = \{\$\}$; /* S is the start symbol of the grammar */

repeat {

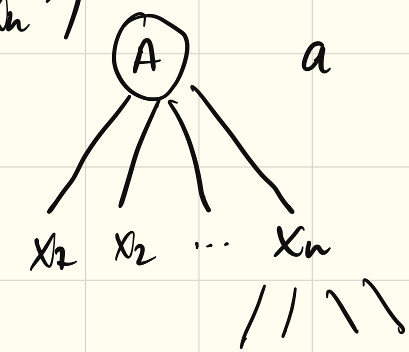
for each production $A \rightarrow \underbrace{X_1 X_2 \ldots X_n}_{}$ { /* $X_i \neq \varepsilon$ */ $A \rightarrow \varepsilon$ not allowed

/* To compute the follow of A, we need to look at the production in which the RHS contains an A */

/* $\therefore$ follow(A) requires a context $S' \Rightarrow^* \alpha A a \beta$ */

/* symbols which follow A will also follow X_n */

$$FOLLOW(X_n) = \underbrace{FOLLOW(X_n)}_{\text{from prev iterations}} \cup FOLLOW(A);$$



REST = FOLLOW(A); shouldn't this be X_n ??

for $i = n \rightarrow 2$ {

  if ($\varepsilon \in FIRST(X_i)$) {

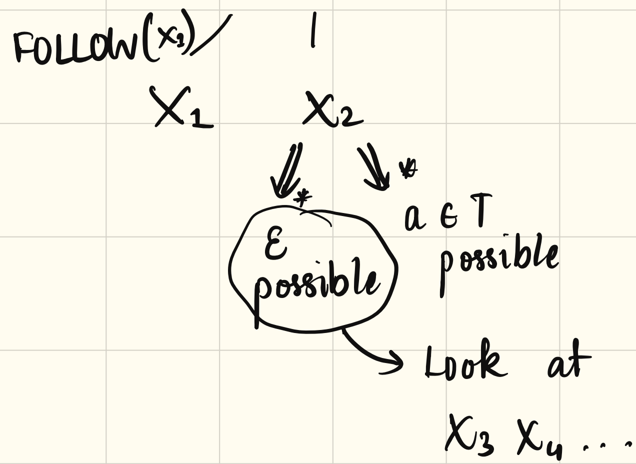
    $FOLLOW(X_{i-1}) = FOLLOW(X_{i-1}) \cup$

      $FIRST(X_i) - \{\varepsilon\} \cup REST;$

    $REST = FOLLOW(X_{i-1})$

  }

  else {



$FOLLOW(X_{i-1}) = FOLLOW(X_{i-1}) \cup FIRST(X_i)$;

$REST = FOLLOW(X_{i-1})$;

}

}

} until no FOLLOW set has changed.

Example:

$S' \rightarrow S\$$

$S \rightarrow aAS \mid \varepsilon$

$A \rightarrow ba \mid SB$

$B \rightarrow cA \mid S$

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

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

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

$\text{FIRST}(A) = \{a, b, c, \varepsilon\}$

$\text{FIRST}(B) = \{a, c, \varepsilon\}$

Iteration 1

* $S \rightarrow aAS$

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

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

$\text{FOLLOW}(A) \cup= (\text{FIRST}(S) - \{\varepsilon\}) \cup \text{REST} = \{a, \$\}$

* $A \rightarrow SB$

$FOLLOW(B) \cup= FOLLOW(A) = \{a, \$\}$

$REST = FOLLOW(A) = \{a, \$\}$

$FOLLOW(S) \cup= FIRST(B - \{\varepsilon\}) \cup REST = \{a, c, \$\}$

* $B \rightarrow cA$

$FOLLOW(A) \cup= FOLLOW(B) = \{a, \$\}$

* $B \rightarrow S$

$FOLLOW(S) \cup= FOLLOW(B) = \{a, c, \$\}$

---

iteration 1 : $FOLLOW(S) = \{a, c, \$\}$; $FOLLOW(A) = FOLLOW(B) = \{a, \$\}$

Iteration 2

* $S \rightarrow aAS$

$FOLLOW(S) \cup= FOLLOW(S) = \{a, c, \$\};$

$REST = FOLLOW(S) = \{a, c, \$\};$

$FOLLOW(A) \cup= (FIRST(S) - \{\varepsilon\}) \cup REST = \{a, c, \$\}$ (changed!)

* $A \rightarrow SB$

$FOLLOW(B) \cup= FOLLOW(A) = \{a, c, \$\}$ (changed!)

$REST = FOLLOW(A) = \{a, c, \$\}$

$FOLLOW(S) \cup= (first(B) - \{\varepsilon\}) \cup REST = \{a, c, \$\}$

(no change)

* At the end of iteration 2

$$\text{FOLLOW}(S) = \text{FOLLOW}(A) = \text{FOLLOW}(B) = \{a, c, \$\}$$

* FOLLOW sets do not change any further

LL(1) conditions

→ based on FIRST and FOLLOW

why? → we want an algorithm to compute the parsing table for LL(1) grammars.

↳ based on FIRST and FOLLOW

* Let G be a context-free grammar.

* G is LL(1) iff for every pair of productions

$$A \to \alpha$$

$$A \to \beta,$$

the following condition holds:

$$\boxed{\text{dirsymb}(\alpha) \cap \text{dirsymb}(\beta) = \phi}$$

$$\text{dirsymb}(\gamma) = \begin{cases} \text{if } (\varepsilon \in \text{first}(\gamma)) \text{ then} \\ \quad ((\text{first}(\gamma) - \{\varepsilon\}) \cup \text{follow}(A)) \\ \text{else first}(\gamma) \end{cases}$$

γ: α, β

direction symbol set

first symbol in the derivation $\neq \varepsilon$

* Equivalent formulation ALSU $\rightarrow$ Dragon book

Ali Sethi when Ravi sethi walks in ...

$$\text{first}\,(\alpha \cdot \underbrace{\text{follow}(A)}_{\substack{\text{set of}\\ \text{terminals}\\ \{a,b,c,d\}}}) \cap \text{first}\,(\beta \cdot \text{follow}(A)) = \phi$$

$\{\beta a, \beta b, \dots\}$

$\underbrace{\text{first of } \{\alpha a, \ \alpha\beta, \dots\}}_{\text{same as dirsymb}}$

Given this definition for LL(1) grammar,
we now give an algorithm for constructing the parsing table.

for each production $A \rightarrow \alpha$
    for each symbol $s \in dirsymb(\alpha)$
    /* s may be either a terminal symbol or \$ */
        add $A \rightarrow \alpha$ to LLPT [A, s]
Make each undefined entry of LLPT as error.

OR

for each production $A \rightarrow \alpha$
    for each terminal symbol $a \in first(\alpha)$
        add $A \rightarrow \alpha$ to LLPT [A, a]
    if ($\varepsilon \in first(\alpha)$)
        for each terminal symbol $b \in follow(A)$
            add $A \rightarrow \alpha$ to LLPT [A, b]

if $\$ \in follow(A)$

add $A \rightarrow \alpha$ to $LLPT[A, \$]$

Make each undefined entry as error.

---

* After the construction of the LL(1) table is complete → using any method

if any slot in the table has two or more productions → then the grammar is **not** LL(1)

Example:

P1 : $S \rightarrow$ if (a) else S | while (a) S | begin SL end

P2 : $SL \rightarrow SS'$

P3 : $S' \rightarrow$; SL | ε

{ if, while, begin, end, a, (,), ; } → all terinimal symbols

→ clearly, all alternatives of P_1 start with distinct symbols (if, while, begin)

→ ∴ no problem LL(1) check ✓

→ P2 has no choices LL(1) check ✓

→ P3 → dirsymb(; SL) = { ; }

$= \{\varepsilon\}$

dirsymb (ε) = if ($\epsilon \in$ first(ε))

(first$(\varepsilon) - \{\varepsilon\}$) $\cup$ FOLLOW(S')

else first(ε)

= FOLLOW(S')

= {end}

S → begin SL end

SS'

; SL ε

S' → last symbol

∴ FOLLOW(S') = FOLLOW(SL)

∴ dirsymb(; SL) $\cap$ dirsymb(ϵ) = ϕ

No common symbols

LL(1) check ✓

Hence the grammar is LL(1)