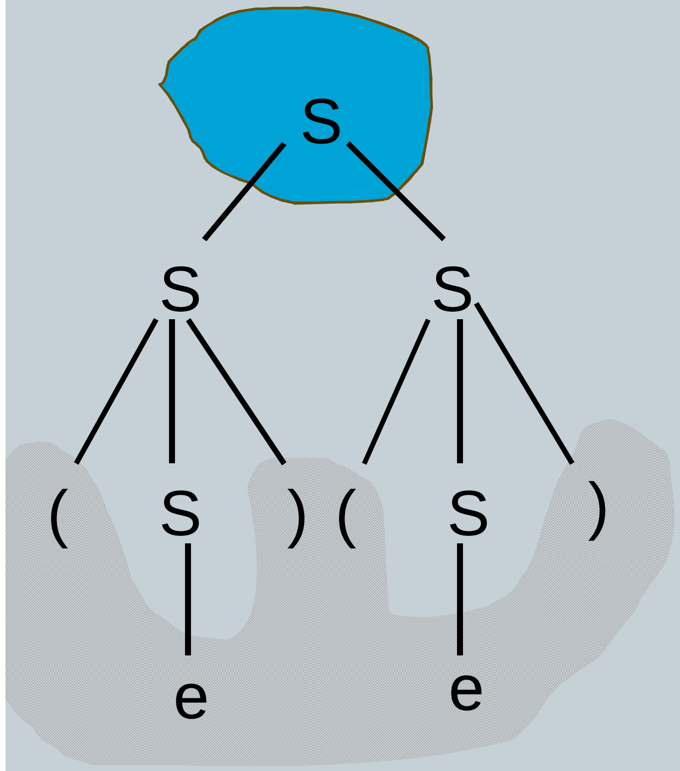
PARSE TREES AND PARSING

Derivations and Parse trees

- G: S --> e | SS | (S)   // L(G) = PAREN
- Now consider the derivations of the string: “()()”.
  - D_1: S--> SS -->(S) S --> ()S --> ()(S) --> ()()
  - D_2: S-->SS -->S(S) -->(S)(S)-->(S)()-->()()
  - D_3: S-->SS -->S(S) --> S() -->(S)() --> ()()
- Notes:
  - 1. D_1 is a leftmost derivation, D_3 is a rightmost derivation while D_2 is neither leftmost nor rightmost derivation.
  - 2. $D_1 \sim D_3$ are the same in the sense that:
    - The rules used (and the number of times of their applications) are the same.
    - All applications of each rule in all 3 derivations are applied to the same place in the string.
    - More intuitively, they are equivalent in the sense that by reordering the applications of applied rules, they can be transformed to the same derivation.

Parse Trees

- $D_1 \sim D_3$ represent different ways of generating the following parse tree for the string “()()”.



Features of the parse tree:

1. The root node is [labeled] the start symbol: S

2. The left to right traversal of all leaves corresponds to the input string : ( ) ( ).

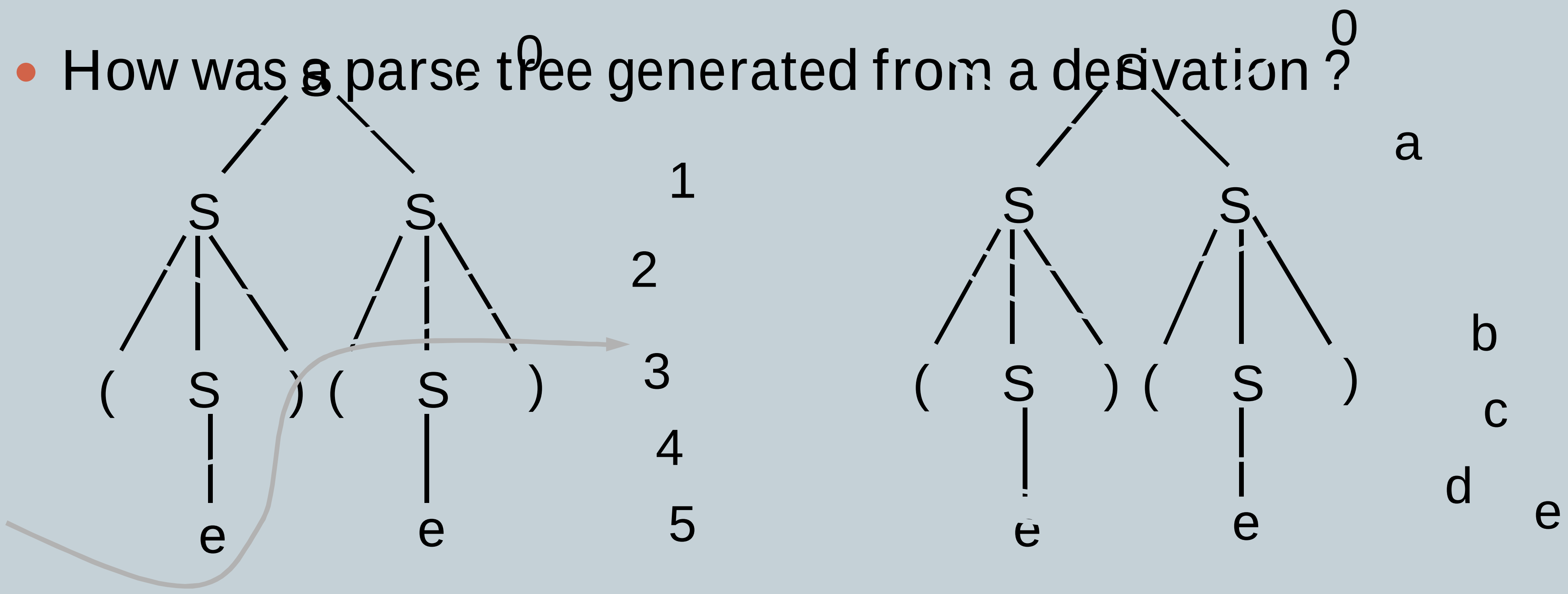
3. If X is an internal node and Y_1 Y_2 … Y_K are an left-to-right listing of all its children in the tree, then X --> Y_1Y_2… Y_k is a rule of G.

4. Every step of derivation corresponds to one-level growth of an internal node

A parse tree for the string “( ) ( )”.

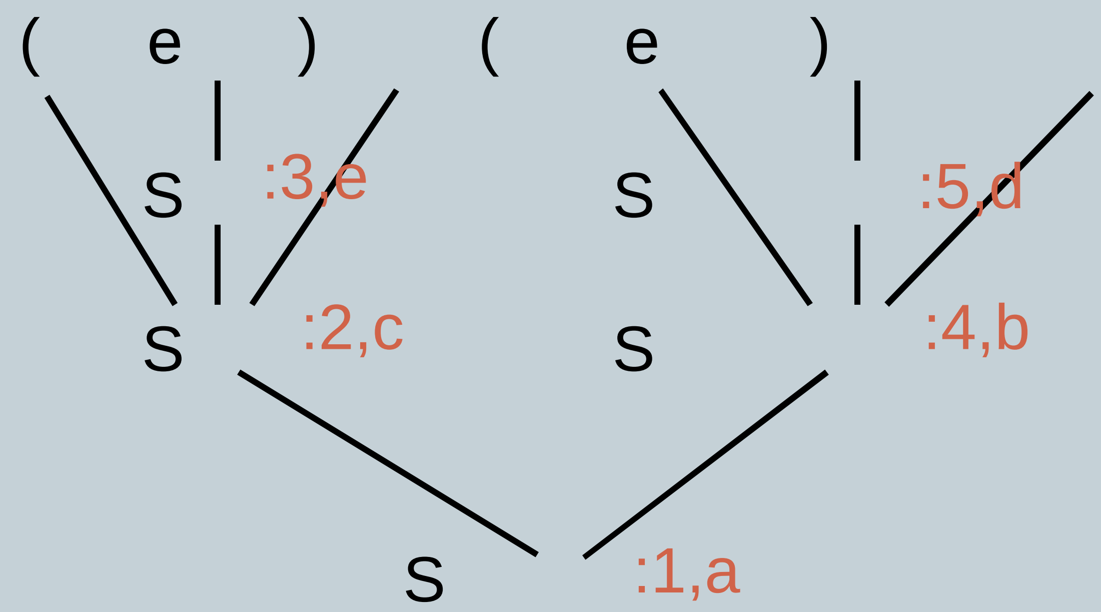
Mapping derivations to parse tree

- How was a parse tree generated from a derivation ?



Top-down view of D_1: S -->* ()() and D_2: S -->* ()().

Bottom-up view of the generation of the parse tree



Remarks:

1. Every derivation describes completely how a parse tree grows up.
2. In practical applications (e.g., compiler), we need to know not only if a input string $w \in L(G)$, but also the parse tree (corresponding to $S \rightarrow^* w$)
3. A grammar is said to be *ambiguous* if there exists some string which has more than one parse tree.
4. In the above example, '()()' has at least three derivations which correspond to the same parse tree and hence does not show that G is ambiguous.
5. Non-uniqueness of derivations is a necessary *but not sufficient* condition for the ambiguity of a grammar.
6. A CFL is said to be ambiguous if every CFG generating it is ambiguous.

An ambiguous context free language

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

- It can be proved that the above language is inherently ambiguous. Namely, all context free grammars for it are ambiguous.

Parse trees and partial parse trees for a CFG

- G= (N,S,P,S) : a CFG

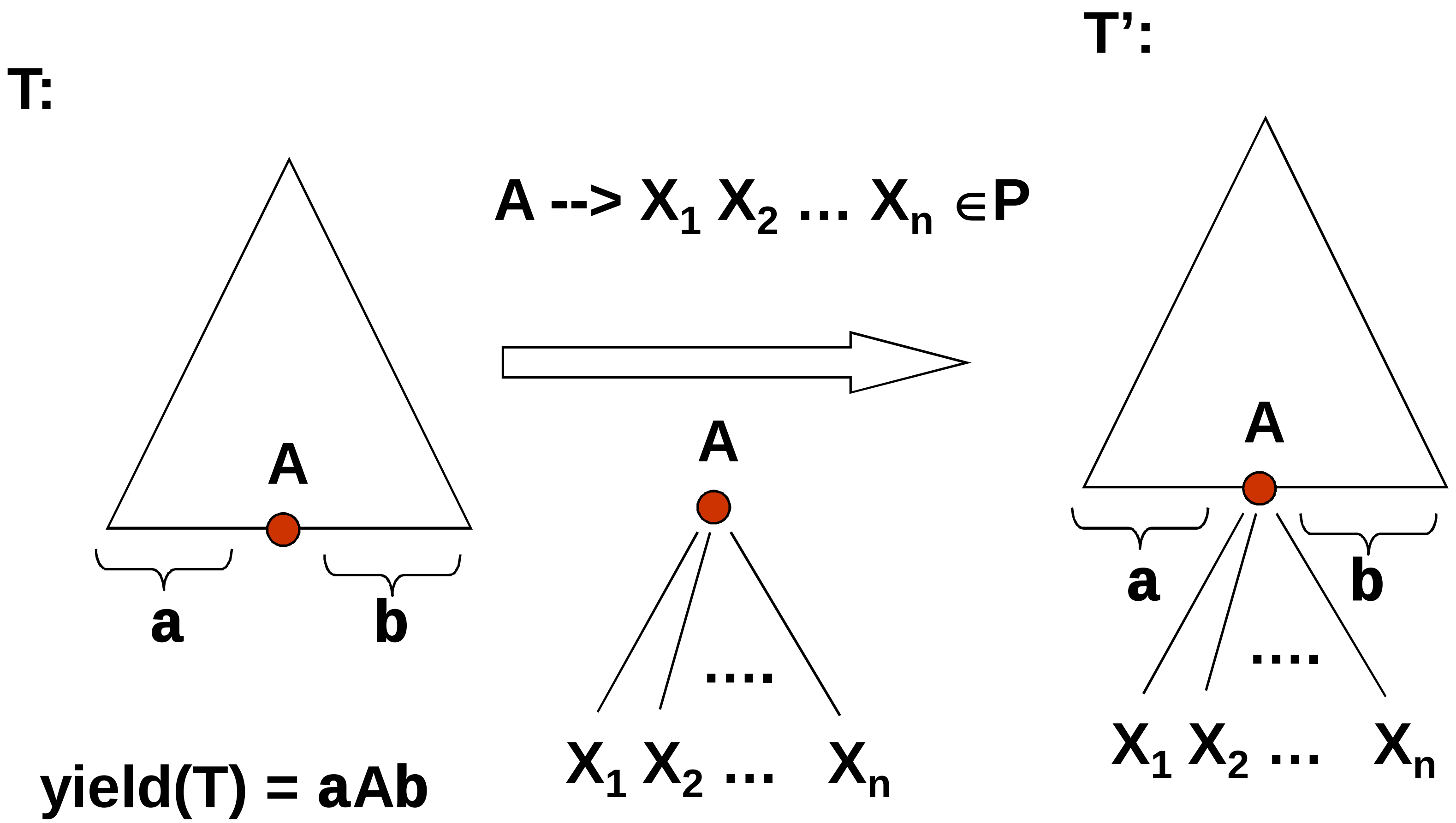
PT(G) $=_{def}$ the set of all parse trees of G, is the set of all trees corresponding to complete derivations (I.e., A -->* w where w $\in$ S*) .

PPT(G) $=_{def}$ the set of all partial parse tree of G is the set of all trees corresponding to all possible derivations (i.e.,

A -->* a , where A $\in$ N and a $\in$ (NUS)*).

- The set PPT(G) and PT(G) are defined inductively as follows:

1. Every nonterminal A is a PPT (with root A and yield A)
2. If T = (... A ...) is a PPT where A a nonterminal leaf and T has yield aAb. and A --> $X_1X_2...X_n$ (n$\geq$ 0) is a production, then the tree T' = (.... (A X_1 X_2 ...X_n) ...) obtained from T by appending $X_1...X_n$ to the leaf A as children of A is a PPT with yield a $X_1...X_n$ b.
3. A PPT is called a partial X-tree if its root is labeled X.
4. A PPT is a parse tree (PT) if its yield is a terminal string.



$\text{yield}(T) = aAb$

$\text{yield}(T') = aX_1X_2\ldots X_nb.$

Relations between parse trees and derivations

Lemma 4.1: If T is a partial X-tree with yield a, then X $-->^*_G$ a.

Pf: proved by ind. on the structure(or number of nodes) of T.

Basis: T = X is a single-node PPT. Then a = X. Hence X $-->^0_G$ a .

Ind: T = (... (A b) ...) can be generated from T' = (.... A ...) with yield mAn by appending b to A. Then

X $-->^*_G$ mAn // by ind. hyp. on T'

$-->_G$ mbn // by def. A --> b in P QED.

- Let D : X --> a_1 --> a_2 --> ... --> a_n be a derivation.

The partial X-tree generated from D, denoted T_D, which has yield(T_D) = a_n, can be defined inductively on n:

1. n = 0 : (i.e., D = X). Then T_D = X is a single-node PPT.
2. n = k+1 > 0: let D = [X --> a_1 --> ... --> a_k = aAb -->a X_1...X_m b]

   = [D' --> a X_1...X_m b]

   then T_D = $T_{D'}$ with leaf A replaced by (A X_1...X_m)

Relations between parse trees and derivations (cont'd)

Lemma 4.2: D = X --> a_1 --> a_2 --> ... --> a_n a derivation. Then T_D is a partial X-tree with yield a_n.

Pf: Simple induction on n. left as an exercise.

- Leftmost and rightmost derivations:
- G: a CFG. Two relations
  - ^{L}-->$_G$ (leftmost derivation),
  - ^{R}-->$_G$ (rightmost derivation) ⊆ $(N \cup S)^+ \times (N \cup S)^*$ are defined as follows: For a , b∈ (NUS)*

1. a ^{L}-->$_G$ b iff $ x ∈ S*, A ∈ N, g∈ (NUS)* and A --> d ∈ P s.t.

   a = xA g and b = x d g

2. a ^{R}-->$_G$ b iff $ x ∈ S*, A ∈ N, g∈ (NUS)* and A --> d ∈ P s.t.

   a = gA x and b = gd x .

3. define ^{L}-->*$_G$ (resp., ^{R}-->*$_G$) as the ref. & trans. closure of ^{L}-->$_G$ (^{R}-->$_G$) .

parse tree and leftmost/rightmost derivations

- Ex: S --> SS | (S) | e. Then

  (SSS) $\longrightarrow_G$ ((S) SS) leftmost

  $\longrightarrow_G$ (SS(S)) rightmost

  $\longrightarrow_G$ (S (S) S) neither leftmost nor rightmost

Theorem 3 : G; a CFG, A ∈ N, w ∈ S*. Then the following statements are equivalent:

(a) A $\longrightarrow^*_G$ w.

(b) $ a parse tree with root A and yield w.

(c) $ a leftmost derivation A $\overset{L}{\longrightarrow}{}^*_G$ w

(d) $ a rightmost derivation A $\overset{R}{\longrightarrow}{}^*_G$ w

pf:

$$\begin{tikzcd} (a) \arrow[r, Leftrightarrow] \arrow[d, dash] \arrow[dr, dash] & (b) \arrow[dl] \arrow[d] \\ (c) & (d) \end{tikzcd}$$

// (a) <==> (b) direct from Lemma 1 & 2.

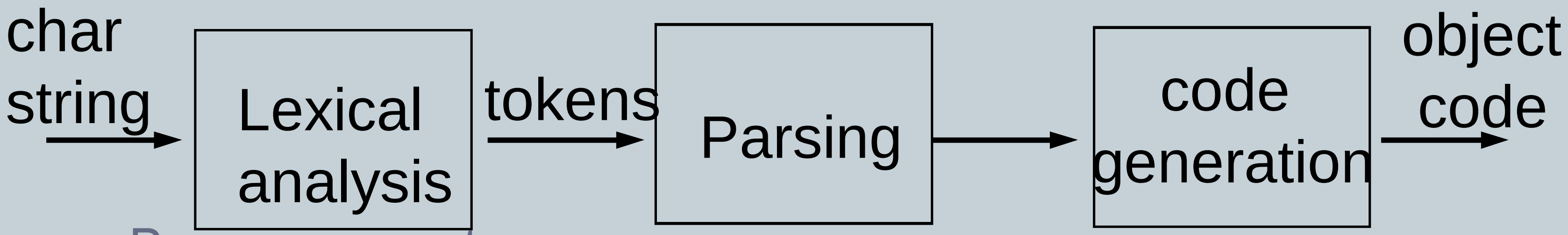
// (c),(d) ==> (a) : by definition

// need to prove (b) ==>(c),(d) only.

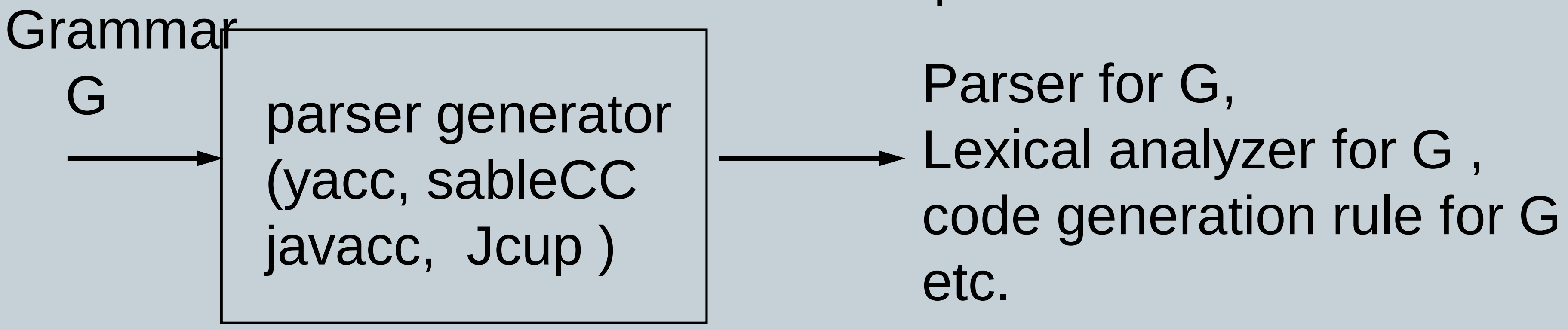
// left as exercise.

Parsing

- Major application of CFG & PDAs:
  - Natural Language Processing(NLP)
  - Programming language, Compiler:
  - Software engineering : text 2 structure



  - Parser generator :

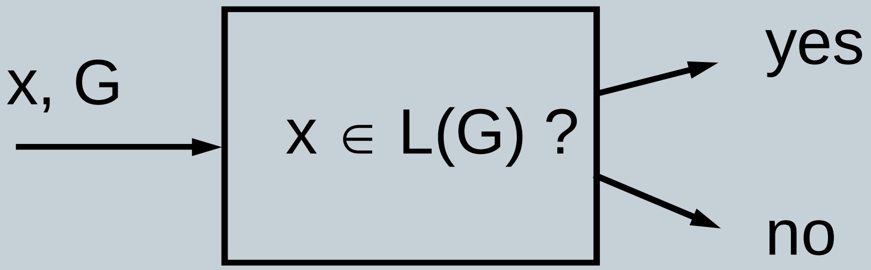

Grammar G
parser generator (yacc, sableCC javacc, Jcup)
Parser for G,
Lexical analyzer for G ,
code generation rule for G
etc.

Parsing (cont'd)

- Parsing is the process of the generation of a parse tree (or its equivalents) corresponding to a given input string w and grammar G.

Note: In formal language we are only concerned with if w $\in$ L(G), but in compiler , we also need to know how w is derived from S (i.e., we need to know the parse tree if it exists).

- A general CFG parser:
  - a program that can solve the problem:
  - x: any input string; G: a CFG



The CYK algorithm

- A general CFG parsing algorithm
  - run in time O(|x|^3).
  - using dynamic programming (DP) technique.
  - applicable to general CFG
  - but our demo version requires the grammar in Chomsky normal form.
- Example : G =

  S --> AB | BA | SS | AC | BD

  A --> a B --> b C --> SB D --> SA

Let x = aabbab, n = |x| = 6.

Steps: 1. Draw n+1 vertical bars separating the symbols of x and number them 0 to n:

```
 |  a  |  a  |  b  |  b  |  a  |  b  |
 0     1     3     3     4     5     6
```

The CYK algorithm (cont'd)

2. /* For each $0 \le i < j \le n$. Let x_{ij} = the substring of x between bar i and bar j.

For each $0 \le i < j \le n$. Let $T(i,j) = \{ X \in N \mid X \text{-->}_G x_{ij} \}$. I.e., T(i,j) is the set of nonterminal symbols that can derive the substring x_{ij} .

- note: $x \in L(G)$ iff $S \in T(0,n)$.

/* The spirit of the algorithm is that the value T(0,n) can be

computed by applying DP technique. */

Build a table with C(n,2) entries as shown in next slide:

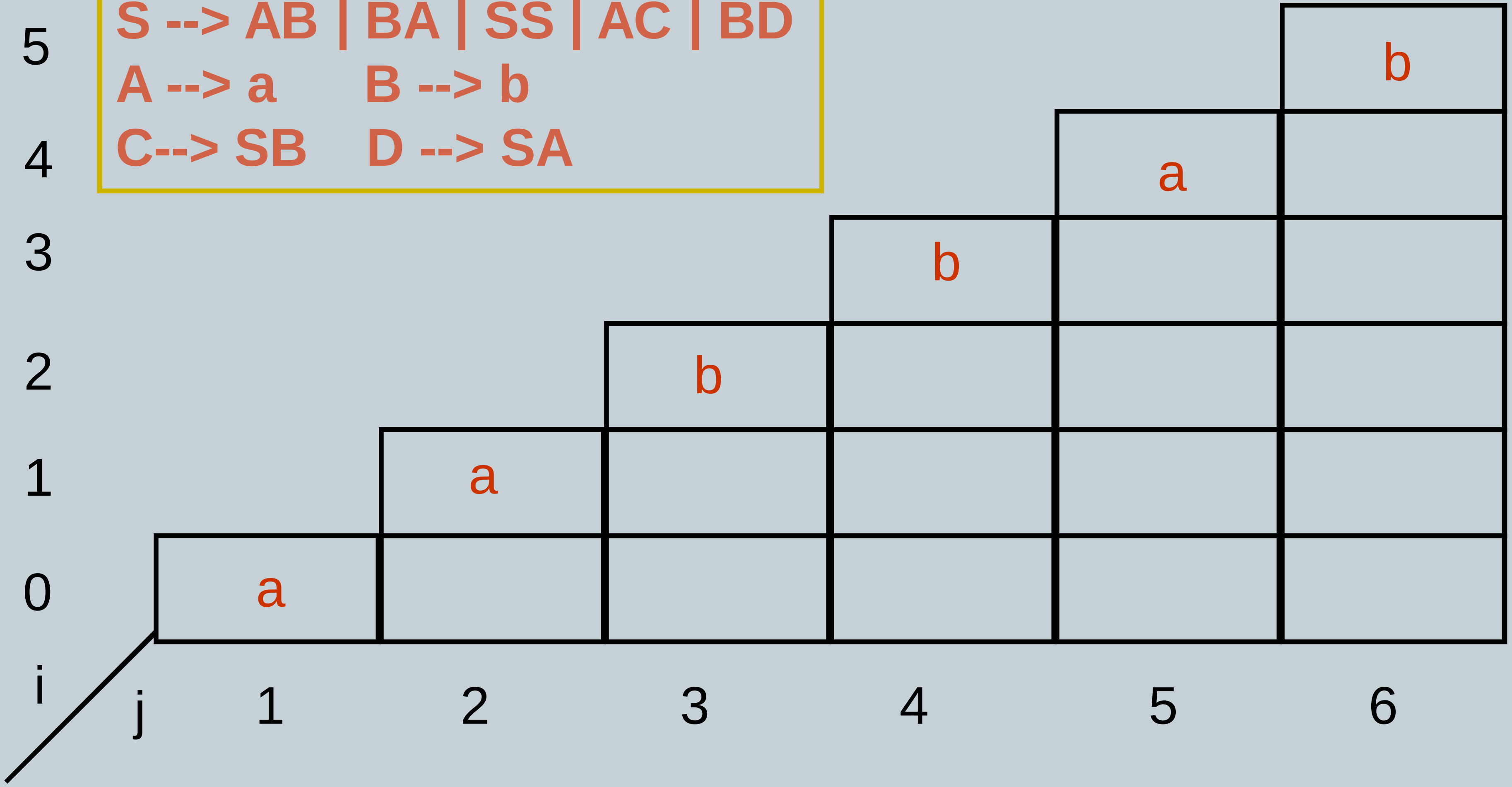
The CYK chart

- The goal is to fill in the table with cell(i,j) = T(i,j).

  Problem: how to proceed ?

  ==> diagonal entries can be filled in immediately !! (why ?)

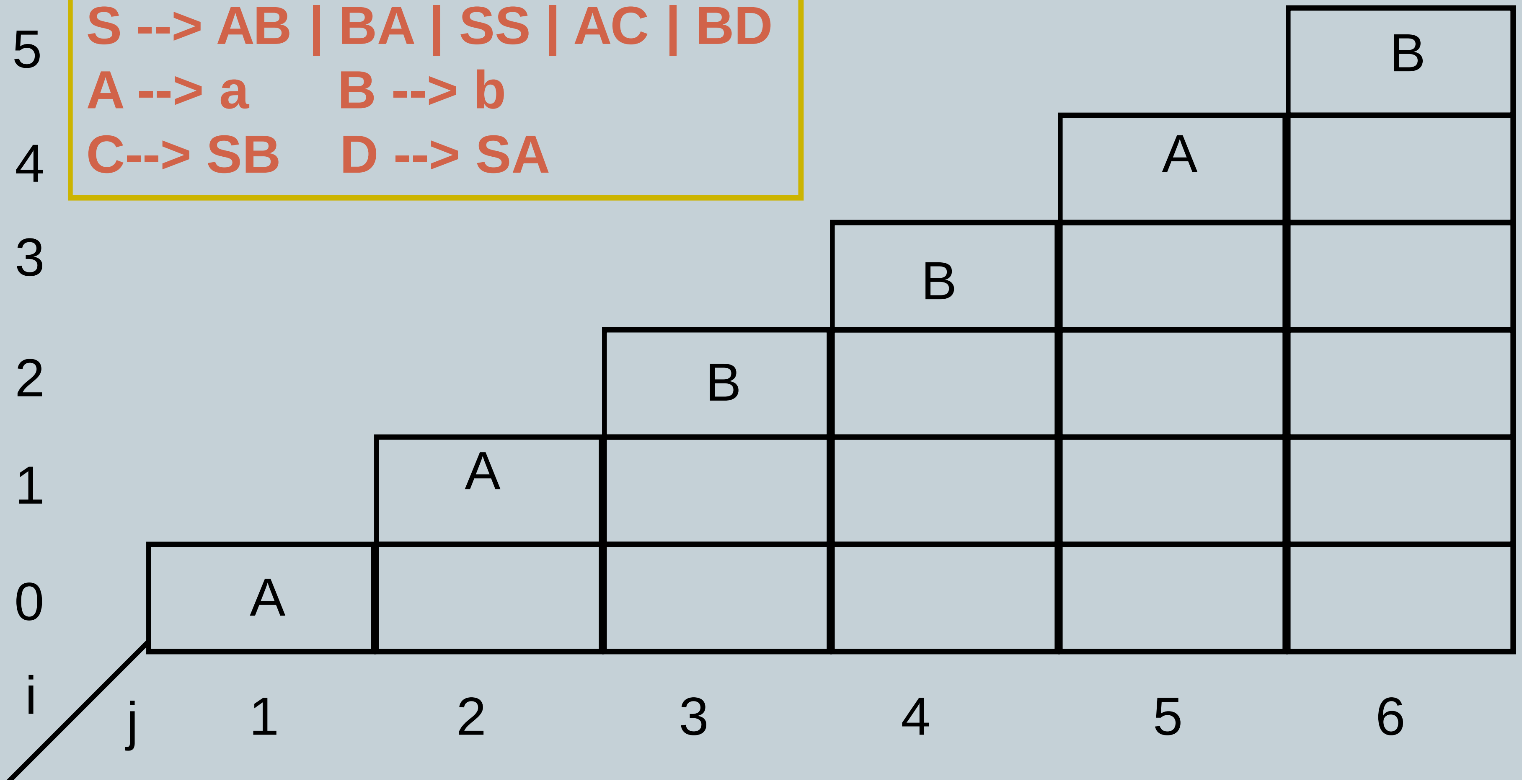
S --> AB | BA | SS | AC | BD
A --> a B --> b
C--> SB D --> SA



Fill in the CYK chart:

- Why C(4,5) = { A } ? since A --> a = x_{45}.
- Once the main diagonal entries were filled in, the next upper diagonal entries can be filled in. (why ?)

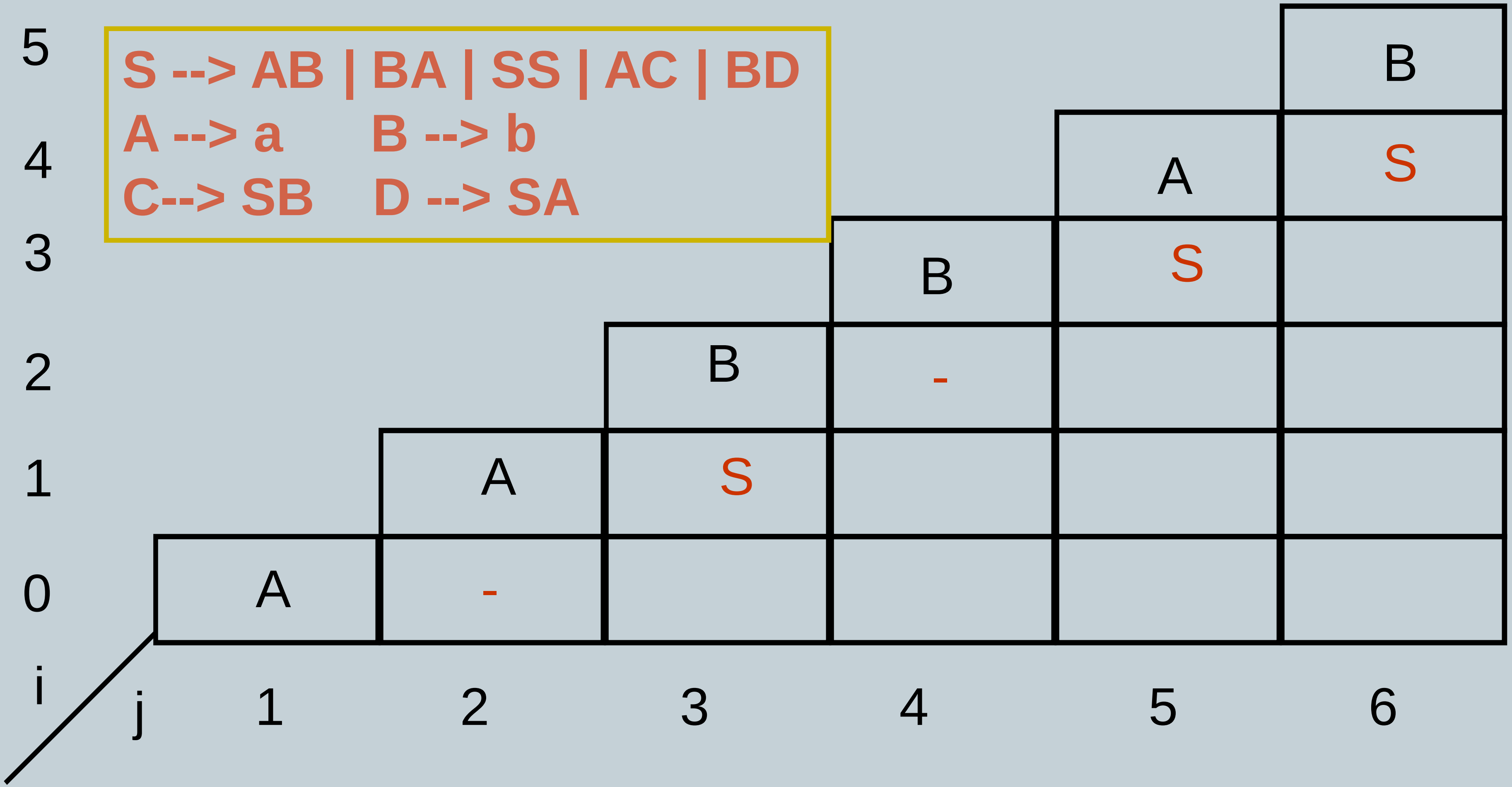
S --> AB | BA | SS | AC | BD
A --> a   B --> b
C--> SB   D --> SA



how to fill in the CYK chart

- T(3,5) = S since $x_{35} = x_{34}\, x_{45}$ <-- T(3,4) T(4,5) = B A <-- S
- In general $T(i,j) = \bigcup_{i < k < j} \{ X \mid X \rightarrow Y Z \mid Y \in T(i,k), Z \in T(k,j)\}$

S --> AB | BA | SS | AC | BD
A --> a B --> b
C--> SB D --> SA



the demo CYK version generalized

- Let P_k = { X --> a | X --> a $\in$ P and | a | = k }.
- Then $T(i,j) = \bigcup_{k > 0} \bigcup_{i = t0 < t1 < t2 < \ldots < tk < j = t(k+1)} \{ X \mid X \to X_1 X_2 \ldots X_k \in P_k$ and for all $m < k+1$ $X_m \in T(t_m, t_{m+1}) \}$

S --> AB | BA | SS | AC | BD
A --> a   B --> b
C--> SB   D --> SA

i \ j	1	2	3	4	5	6
5						B
4					A	S
3				B	S	C
2			B	-	-	-
1		A	S	C	S	C
0	A	-	-	S	D	S^2

The CYK algorithm

```
// input grammar is in Chomsky normal form
1. for i = 0 to n-1  do {        // first do substring of length 1
     T(i,i+1) = {};
     for each rule of the form A-> a do
        if a = $x_{i,i+1}$ then T(i,i+1) = T(i,i+1) U {A};
2. for m = 2 to n do                // for each length m > 1
      for i = 0 to n - m do{        // for each substring of length m
           T(i, i + m ) = {};
           for j = i + 1 to i + m -1 do{   // for each break of the string
              for each rule of the form A --> BC do
               If B $\in$ T(i,j) and C $\in$ T(j,i+m) then
                     T(i,i+m) = T(i,i+m) U {A}
  }}
```