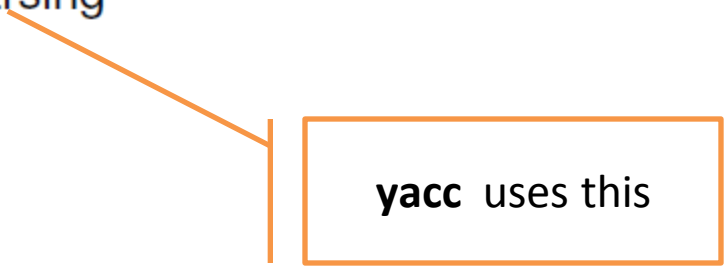
Syntax Anaysis (Parsing)

Outline of the Lecture

- What is syntax analysis?
- Specification of programming languages: context-free grammars
- Parsing context-free languages: push-down automata
- Top-down parsing: LL(1) and recursive-descent parsing
- Bottom-up parsing: LR-parsing



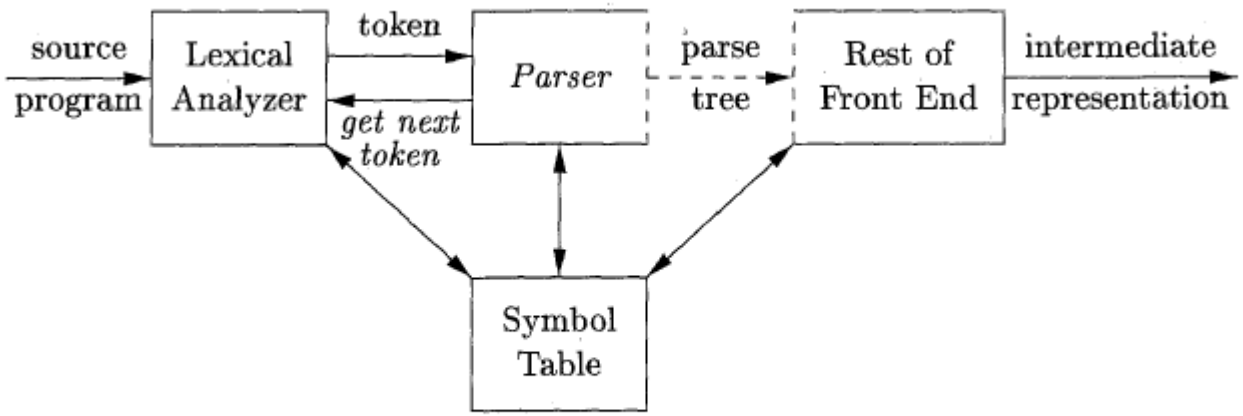


Figure 4.1: Position of parser in compiler model

Parsers

- Universal parsers (such as Cocke-Younger-Kasami algorithm) can parse any grammar (not limited to CFGs). But are too inefficient.
- Top-down -- eg: LL
- Bottom-up -- eg: LR

Grammars

- Every programming language has precise grammar rules that describe the syntactic structure of well-formed programs
  - In C, the rules state how functions are made out of parameter lists, declarations, and statements; how statements are made of expressions, etc.
- Grammars are easy to understand, and parsers for programming languages can be constructed automatically from certain classes of grammars
- Parsers or syntax analyzers are generated *for* a particular grammar
- Context-free grammars are usually used for syntax specification of programming languages

What is Parsing or Syntax Analysis?

- A parser for a grammar of a programming language
  - verifies that the string of tokens for a program in that language can indeed be generated from that grammar
  - reports any syntax errors in the program
  - constructs a parse tree representation of the program (not necessarily explicit)
  - usually calls the lexical analyzer to supply a token to it when necessary
  - could be hand-written or automatically generated
  - is based on *context-free* grammars
- Grammars are generative mechanisms like regular expressions
- Pushdown automata are machines recognizing context-free languages (like FSA for RL)

Context-free Grammars

- A CFG is denoted as $G = (N, T, P, S)$
  - N: Finite set of non-terminals
  - T: Finite set of terminals
  - $S \in N$: The start symbol
  - P: Finite set of productions, each of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$
- Usually, only P is specified and the first production corresponds to that of the start symbol
- Examples

Tokens are terminals

(1)
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow id$

(2)
$S \rightarrow 0S0$
$S \rightarrow 1S1$
$S \rightarrow 0$
$S \rightarrow 1$
$S \rightarrow \epsilon$

(3)
$S \rightarrow aSb$
$S \rightarrow \epsilon$

(4)
$S \rightarrow aB \mid bA$
$A \rightarrow a \mid aS \mid bAA$
$B \rightarrow b \mid bS \mid aBB$

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T / F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

- From the notational convention and context it is clear that,
- Non-terminals = {E, T, F},
- Terminals = {+,-,*,/,(,), id },
- Start symbol = E.

Grammar 1

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

Derivations

- $E \Rightarrow^{E \to E+E} E + E \Rightarrow^{E \to id} id + E \Rightarrow^{E \to id} id + id$ is a derivation of the terminal string $id + id$ from E
- In a derivation, a production is applied at each step, to replace a nonterminal by the right-hand side of the corresponding production
- In the above example, the productions $E \to E + E$, $E \to id$, and $E \to id$, are applied at steps 1,2, and, 3 respectively
- The above derivation is represented in short as, $E \Rightarrow^* id + id$, and is read as S **derives** $id + id$

1. $\alpha \stackrel{*}{\Rightarrow} \alpha$, for any string α, and
2. If $\alpha \stackrel{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \stackrel{*}{\Rightarrow} \gamma$.

Likewise, $\stackrel{+}{\Rightarrow}$ means, "derives in one or more steps."

1. In *leftmost* derivations, the leftmost nonterminal in each sentential is always chosen. If $\alpha \Rightarrow \beta$ is a step in which the leftmost nonterminal in α is replaced, we write $\alpha \underset{lm}{\Rightarrow} \beta$.
2. In *rightmost* derivations, the rightmost nonterminal is always chosen; we write $\alpha \underset{rm}{\Rightarrow} \beta$ in this case.

Context-free Languages

- Context-free grammars generate context-free languages (grammar and language resp.)
- The *language generated by G*, denoted $L(G)$, is $L(G) = \{w \mid w \in T^*, \text{ and } S \Rightarrow^* w\}$
  i.e., a string is in $L(G)$, if
  1. the string consists solely of terminals
  2. the string can be derived from S
- Examples
  1. $L(G_1)$ = Set of all expressions with +, *, names, and balanced '(' and ')'
  2. $L(G_2)$ = Set of palindromes over 0 and 1
  3. $L(G_3) = \{a^n b^n \mid n \geq 0\}$
  4. $L(G_4) = \{x \mid x \text{ has equal no. of } a\text{'s and } b\text{'s}\}$
- A string $\alpha \in (N \cup T)^*$ is a **sentential form** if $S \Rightarrow^* \alpha$
- Two grammars G_1 and G_2 are equivalent, if $L(G_1) = L(G_2)$

(1)
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow id$

(2)
$S \rightarrow 0S0$
$S \rightarrow 1S1$
$S \rightarrow 0$
$S \rightarrow 1$
$S \rightarrow \epsilon$

(3)
$S \rightarrow aSb$
$S \rightarrow \epsilon$

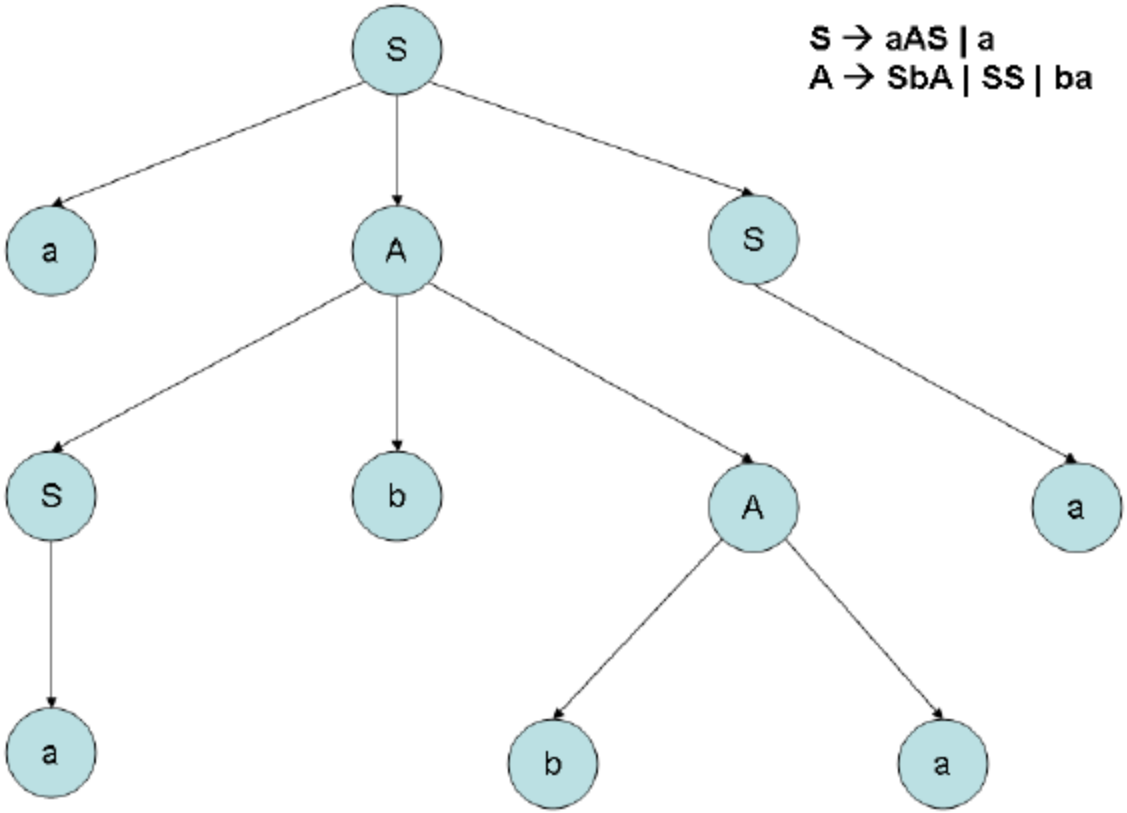
(4)
$S \rightarrow aB \mid bA$
$A \rightarrow a \mid aS \mid bAA$
$B \rightarrow b \mid bS \mid aBB$

•Sentence consists of only terminals.
•Sentential form can consist of both variables and terminals
•Proof by induction can be used to show that the grammar generates a particular language.

Derivation Trees

- Derivations can be displayed as trees
- The internal nodes of the tree are all nonterminals and the leaves are all terminals
- Corresponding to each internal node A, there exists a production $\in$ P, with the RHS of the production being the list of children of A, read from left to right
- The **yield** of a derivation tree is the list of the labels of all the leaves read from left to right
- If α is the yield of some derivation tree for a grammar G, then $S \Rightarrow^* \alpha$ and conversely

Derivation Tree Example



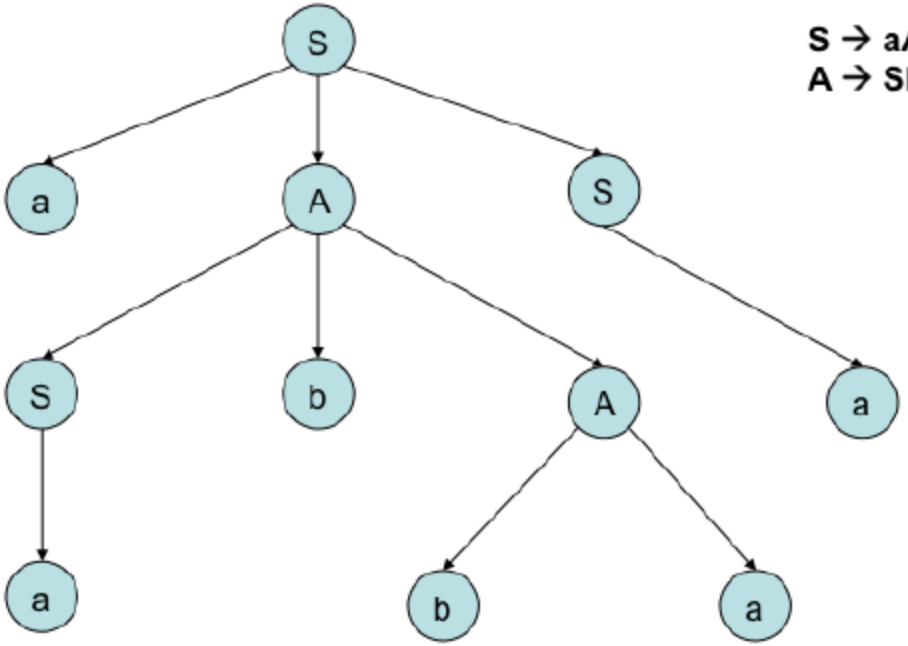
S → aAS | a
A → SbA | SS | ba

S => aAS => aSbAS => aabAS => aabbaS => aabbaa

Leftmost and Rightmost Derivations

- If at each step in a derivation, a production is applied to the leftmost nonterminal, then the derivation is said to be **leftmost**. Similarly **rightmost derivation**.
- If $w \in L(G)$ for some G, then w has at least one parse tree and corresponding to a parse tree, w has unique leftmost and rightmost derivations
- If some word w in $L(G)$ has two or more parse trees, then G is said to be **ambiguous**
- A CFL for which every G is ambiguous, is said to be an **inherently ambiguous** CFL

Leftmost and Rightmost Derivations: An Example



S → aAS | a
A → SbA | SS | ba

Leftmost derivation: **S $\underset{lm}{\Rightarrow}$ aAS $\underset{lm}{\Rightarrow}$ aSbAS $\underset{lm}{\Rightarrow}$ aabAS $\underset{lm}{\Rightarrow}$ aabbaS $\underset{lm}{\Rightarrow}$ aabbaa**

Rightmost derivation: **S $\underset{rm}{\Rightarrow}$ aAS $\underset{rm}{\Rightarrow}$ aAa $\underset{rm}{\Rightarrow}$ aSbAa $\underset{rm}{\Rightarrow}$ aSbbaa $\underset{rm}{\Rightarrow}$ aabbaa**

Note, you need to give production on top of =>

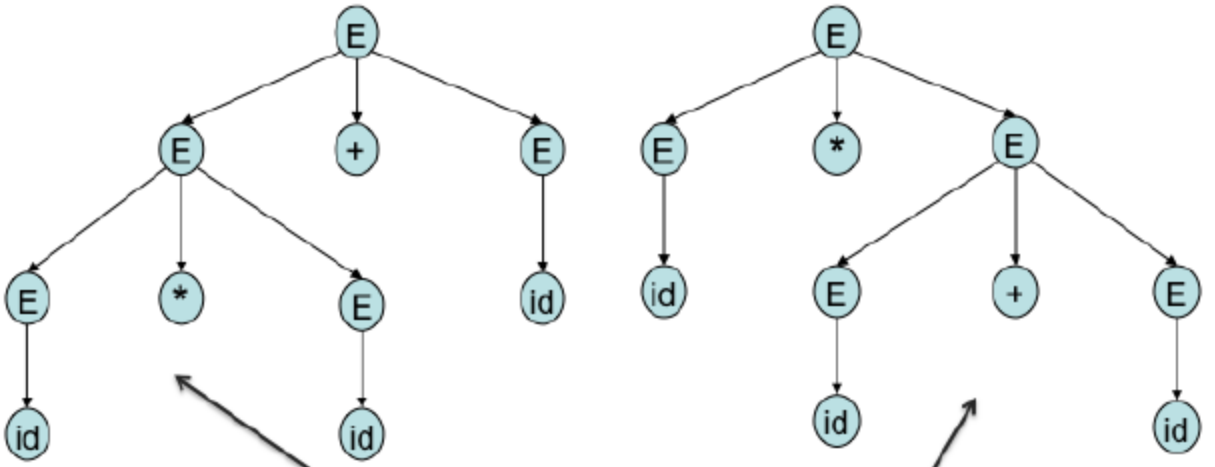
Ambiguous Grammar Examples

- The grammar, $E \rightarrow E + E | E * E | (E) | id$
  is ambiguous, but the following grammar for the same language is unambiguous
  $E \rightarrow E + T | T$, $T \rightarrow T * F | F$, $F \rightarrow (E) | id$
- The grammar,
  $stmt \rightarrow IF\ expr\ stmt | IF\ expr\ stmt\ ELSE\ stmt | other\_stmt$

  is ambiguous, but the following equivalent grammar is not

  $stmt \rightarrow IF\ expr\ stmt | IF\ expr\ matched\_stmt\ ELSE\ stmt$
  $matched\_stmt \rightarrow$
  $IF\ expr\ matched\_stmt\ ELSE\ matched\_stmt | other\_stmt$
- The language,
  $L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$,
  is inherently ambiguous

Ambiguity Example 1

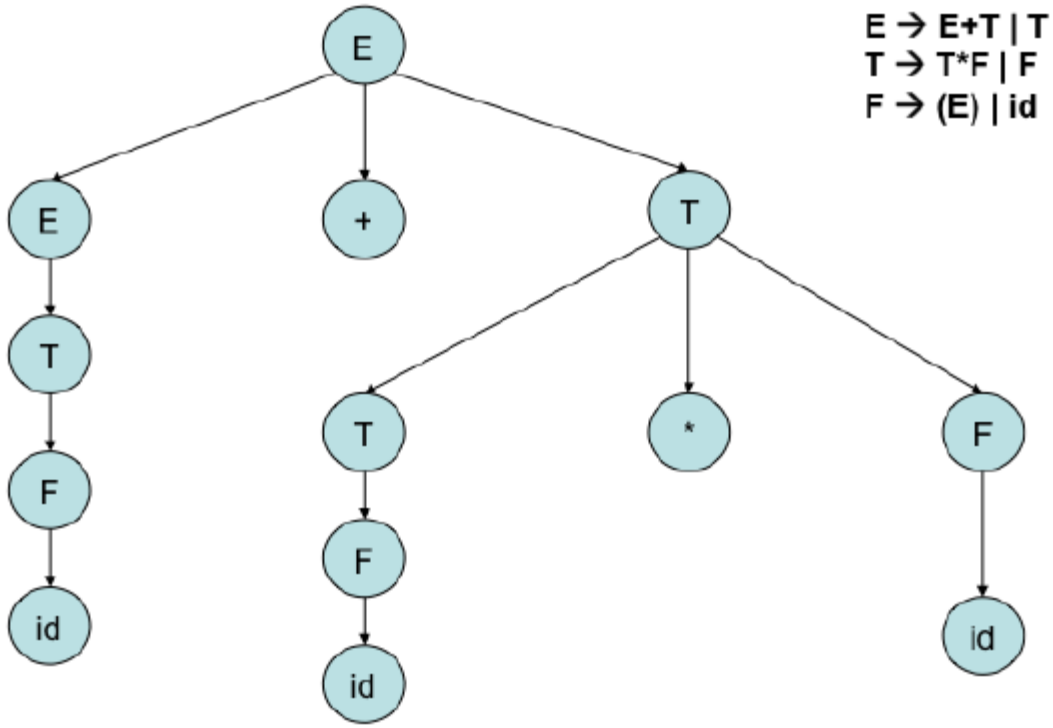


E => E+E => E*E+E => id*E+E => id*id+E => id*id+id

E => E*E => id*E => id*E+E => id*id+E => id*id+id

E → E+E | E*E | (E) | id

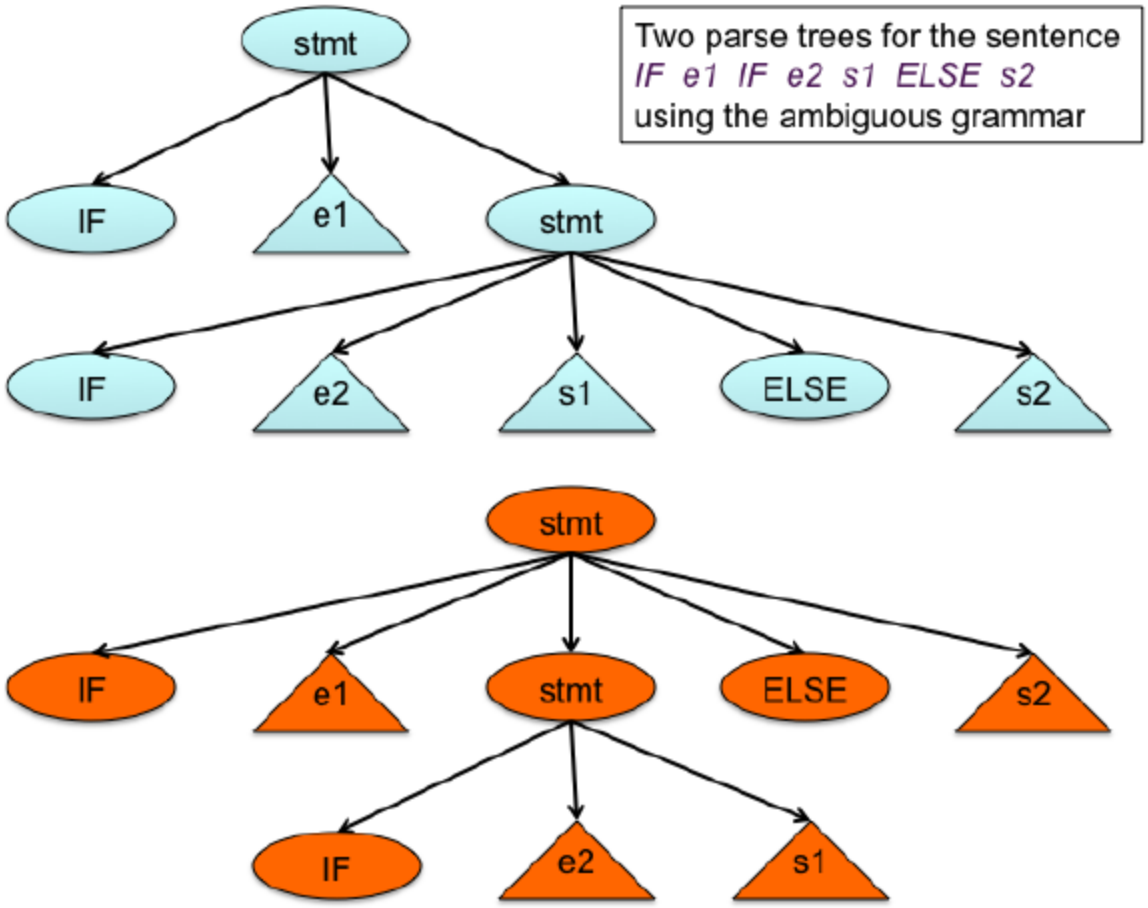
Equivalent Unambiguous Grammar



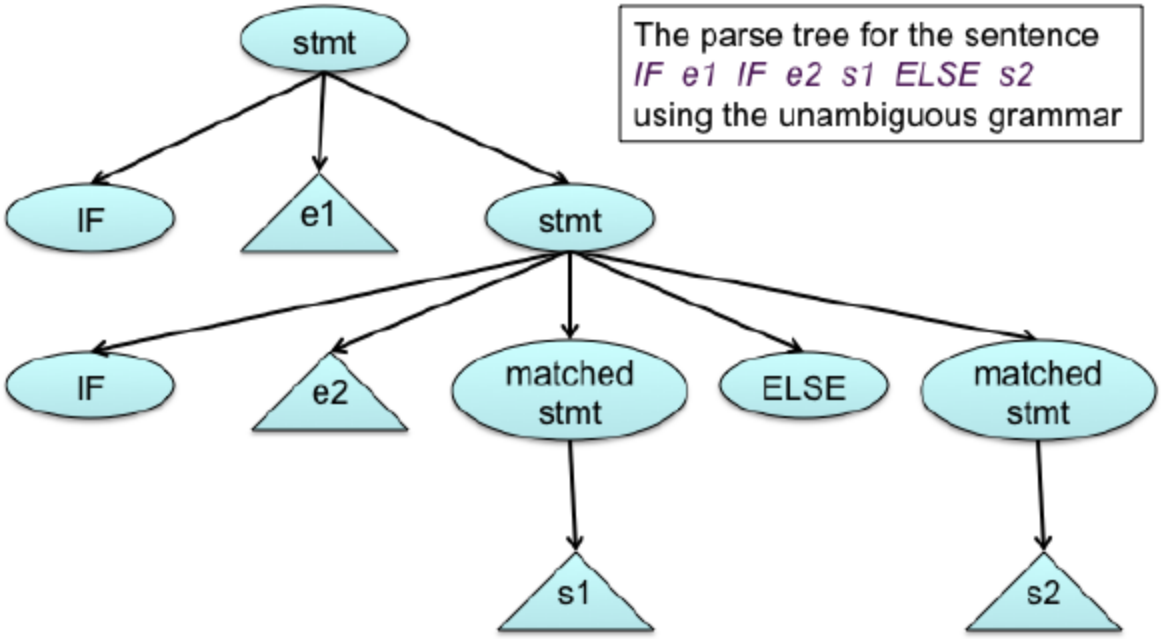
E => E+T => T+T => F+T => id+T => id+T*F => id+F*F => id+id*F => id+id*id

E => T*F => F*F => (E)*F => (E+T)*F => (T+T)*F => (F+T)*F => (id+T)*F
=> (id+F)*id => (id+id)*F => (id+id)*id

Ambiguity Example 2



Ambiguity Example 2 (contd.)



The parse tree for the sentence
IF e1 IF e2 s1 ELSE s2
using the unambiguous grammar

s→ IF e s | IF e ms ELSE s
ms → IF e ms ELSE ms | other_s

Fragment of C-Grammar (Statements)

```
program --> VOID MAIN '(' ')' compound_stmt
compound_stmt --> '{' '}' | '{' stmt_list '}'
          | '{' declaration_list stmt_list '}'
stmt_list --> stmt | stmt_list stmt
stmt --> compound_stmt| expression_stmt
            | if_stmt | while_stmt
expression_stmt --> ';'| expression ';'
if_stmt --> IF '(' expression ')' stmt
   | IF '(' expression ')' stmt ELSE stmt
while_stmt --> WHILE '(' expression ')' stmt
expression --> assignment_expr
             | expression ',' assignment_expr
```

Fragment of C-Grammar (Expressions)

```
assignment_expr --> logical_or_expr
        | unary_expr  assign_op  assignment_expr
assign_op --> '='| MUL_ASSIGN| DIV_ASSIGN
                | ADD_ASSIGN| SUB_ASSIGN
                | AND_ASSIGN| OR_ASSIGN
unary_expr --> primary_expr
        | unary_operator   unary_expr
unary_operator --> '+'| '-'| '!'
primary_expr --> ID| NUM| '(' expression ')'
logical_or_expr --> logical_and_expr
        | logical_or_expr   OR_OP logical_and_expr
logical_and_expr --> equality_expr
        | logical_and_expr   AND_OP  equality_expr
equality_expr --> relational_expr
            | equality_expr EQ_OP relational_expr
            | equality_expr NE_OP relational_expr
```

Fragment of C-Grammar (Expressions and Declarations)

```
relational_expr --> add_expr
                  | relational_expr '<' add_expr
                  | relational_expr '>' add_expr
                  | relational_expr LE_OP add_expr
                  | relational_expr GE_OP add_expr
add_expr --> mult_expr| add_expr '+' mult_expr
                      | add_expr '-' mult_expr
mult_expr --> unary_expr| mult_expr '*' unary_expr
                        | mult_expr '/' unary_expr
declarationlist --> declaration
                  | declarationlist  declaration
declaration --> type  idlist ';'
idlist --> idlist ',' ID | ID
type --> INT_TYPE | FLOAT_TYPE | CHAR_TYPE
```

CFG Vs. Regular Expressions

- CFGs are more powerful than REs
  - Every RE can be stated as a CFG
  - But every CFG cannot be written as a RE

For example, the regular expression $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$

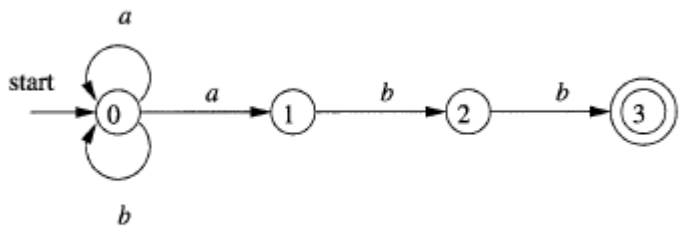


Figure 3.24: A nondeterministic finite automaton

$$
\begin{aligned}
A_0 &\rightarrow aA_0 \mid bA_0 \mid aA_1 \\
A_1 &\rightarrow bA_2 \\
A_2 &\rightarrow bA_3 \\
A_3 &\rightarrow \epsilon
\end{aligned}
$$

Right linear grammar

Right Linear Grammars and Regular Languages

Q: Can every CFG be converted into a right-linear grammar?

Right Linear Grammars and Regular Languages

A: NO! This would mean that all context free languages are regular.

EG:

$$S \rightarrow \varepsilon \mid aSb$$

cannot be converted because $\{a^n b^n\}$ is not regular.

Chomsky Normal Form

Noam Chomsky came up with an especially simple type of context free grammars which is able to capture all context free languages.

Chomsky's grammatical form is particularly useful when one wants to prove certain facts about context free languages. This is because assuming a much more restrictive kind of grammar can often make it easier to prove that the generated language has whatever property you are interested in.

Chomsky Normal Form DEFINITION

DEF: A CFG is said to be in ***Chomsky Normal Form*** if every rule in the grammar has one of the following forms:

- $S \rightarrow \varepsilon$ (ε for epsilon's sake only)
- $A \rightarrow BC$ (dyadic variable productions)
- $A \rightarrow a$ (unit terminal productions)

Where S is the start variable, A,B,C are variables and a is a terminal. Thus epsilons may only appear on the right hand side of the start symbol and other RHS are either 2 variables or a single terminal.

CFG → CNF

Converting a general grammar into Chomsky Normal Form works in four steps:

1. Ensure that the start variable doesn't appear on the right hand side of any rule.
2. Remove all epsilon productions, except from start variable.
3. Remove unit variable productions of the form $A \rightarrow B$ where A and B are variables.
4. Add variables and dyadic variable rules to replace any longer non-dyadic or non-variable productions

CFG → CNF Example

Let's see how this works on the following example grammar for pal:

$$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$$

CFG → CNF
1. Start Variable

Ensure that start variable doesn't appear on the right hand side of any rule.

$$S' \rightarrow S$$

$$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$$

CFG → CNF
2. Remove Epsilons

Remove all epsilon productions, except from start variable.

$$S' \rightarrow S \mid \varepsilon$$

$$S \rightarrow \cancel{\varepsilon} \mid a \mid b \mid aSa \mid bSb \mid aa \mid bb$$

CFG → CNF
3. Remove Variable Units

Remove unit variable productions of the form $A \rightarrow B$.

$$S' \rightarrow \cancel{S} \mid \varepsilon \mid a \mid b \mid aSa \mid bSb \mid aa \mid bb$$

$$S \rightarrow a \mid b \mid aSa \mid bSb \mid aa \mid bb$$

CFG → CNF
4. Longer Productions

Add variables and dyadic variable rules to replace any longer productions.

$$S' \to \varepsilon \mid a \mid b \mid \cancel{aSa} \mid \cancel{bSb} \mid \cancel{aa} \mid \cancel{bb} \mid AB \mid CD \mid AA \mid CC$$

$$S \to a \mid b \mid \cancel{aSa} \mid \cancel{bSb} \mid \cancel{aa} \mid \cancel{bb} \mid AB \mid CD \mid AA \mid CC$$

$$A \to a$$

$$B \to SA$$

$$C \to b$$

$$D \to SC$$

CFG → CNF Result

$$S' \to \varepsilon \mid a \mid b \mid AB \mid CD \mid AA \mid CC$$

$$S \to a \mid b \mid AB \mid CD \mid AA \mid CC$$

$$A \to a$$

$$B \to SA$$

$$C \to b$$

$$D \to SC$$

CYK Algorithm (Uses CNF)

Example 7.34: The following are the productions of a CNF grammar G:

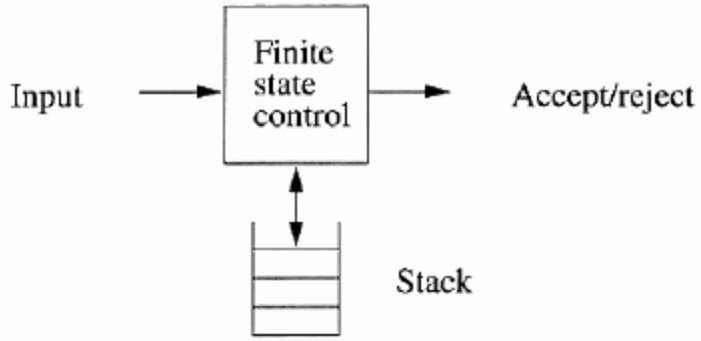
$$\begin{array}{lcl} S & \rightarrow & AB \mid BC \\ A & \rightarrow & BA \mid a \\ B & \rightarrow & CC \mid b \\ C & \rightarrow & AB \mid a \end{array}$$

We shall test for membership in $L(G)$ the string *baaba*. Figure 7.14 shows the table filled in for this string.

$\{S,A,C\}$				
-	$\{S,A,C\}$			
-	$\{B\}$	$\{B\}$		
$\{S,A\}$	$\{B\}$	$\{S,C\}$	$\{S,A\}$	
$\{B\}$	$\{A,C\}$	$\{A,C\}$	$\{B\}$	$\{A,C\}$
b	a	a	b	a

Figure 7.14: The table for string *baaba* constructed by the CYK algorithm

Pushdown automaton



A pushdown automaton is essentially a finite automaton with a stack data structure

Pushdown Automata

A PDA M is a system $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, where

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $q_0 \in Q$ is the start state
- $z_0 \in \Gamma$ is the start symbol on stack (initialization)
- $F \subseteq Q$ is the set of final states
- δ is the transition function, $Q \times \Sigma \cup \{\epsilon\} \times \Gamma$ to finite subsets of $Q \times \Gamma^*$

A typical entry of δ is given by
$\delta(q, a, z) = \{(p_1, \gamma_1), ((p_2, \gamma_2), ..., (p_m, \gamma_m)\}$
The PDA in state q, with input symbol a and top-of-stack symbol z, can enter any of the states p_i, replace the symbol z by the string γ_i, and advance the input head by one symbol.

Instantaneous Descriptions of a PDA

Instantaneous description (ID) or the configuration of a PDA by a triple (q, w, γ), where

1. q is the state,
2. w is the remaining input, and
3. γ is the stack contents.

One move of the PDA could be shown as : $(q, aw, X\beta) \vdash (p, w, \alpha\beta)$

We write $I \vdash^* J$ if there is a sequence of ID's $K_1, K_2, \ldots, K_n$ such that $I = K_1$, $J = K_n$, and for all $i = 1, 2, \ldots, n-1$, we have $K_i \vdash K_{i+1}$.

Pushdown Automata (contd.)

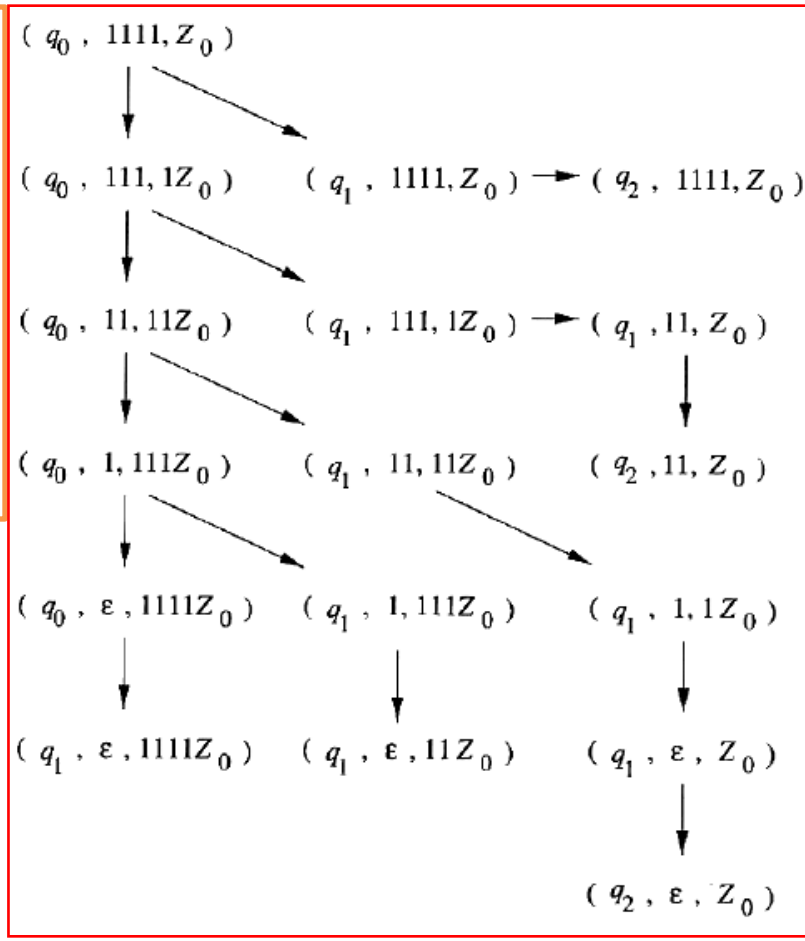
- The leftmost symbol of γ_i will be the new top of stack
- a in the above function δ could be ϵ, in which case, the input symbol is not used and the input head is not advanced
- For a PDA M, we define $L(M)$, the language accepted by M **by final state**, to be
  $L(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \gamma)$, for some $p \in F$ and $\gamma \in \Gamma^*\}$
- We define $N(M)$, the language accepted by M **by empty stack**, to be
  $N(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$, for some $p \in Q$
- When acceptance is by empty stack, the set of final states is irrelevant, and usually, we set $F = \phi$

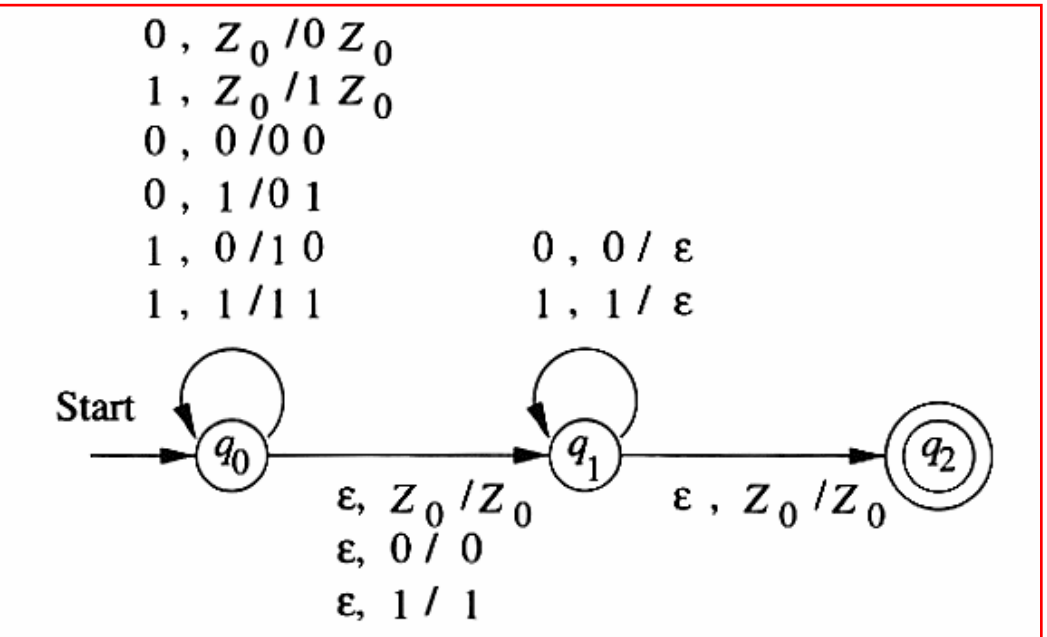
PDA for wwr

$$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

$$\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$$
$$\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$$
$$\delta(q_0, 0, 0) = \{(q_0, 00)\}$$
$$\delta(q_0, 0, 1) = \{(q_0, 01)\}$$
$$\delta(q_0, 1, 0) = \{(q_0, 10)\}$$
$$\delta(q_0, 1, 1) = \{(q_0, 11)\}$$

$$\delta(q_0, \epsilon, Z_0) = \{(q_1, Z_0)\}$$
$$\delta(q_0, \epsilon, 0) = \{(q_1, 0)\}$$
$$\delta(q_0, \epsilon, 1) = \{(q_1, 1)\}$$
$$\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$$
$$\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$$
$$\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$$





PDA - Examples

- $L = \{0^n 1^n \mid n \geq 0\}$
  $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \{Z, 0\}, \delta, q_0, Z, \{q_0\})$, where δ is defined as follows
  $\delta(q_0, 0, Z) = \{(q_1, 0Z)\}$, $\delta(q_1, 0, 0) = \{(q_1, 00)\}$,
  $\delta(q_1, 1, 0) = \{(q_2, \epsilon)\}$, $\delta(q_2, 1, 0) = \{(q_2, \epsilon)\}$,
  $\delta(q_2, \epsilon, Z) = \{(q_0, \epsilon)\}$
- $(q_0, 0011, Z) \vdash (q_1, 011, 0Z) \vdash (q_1, 11, 00Z) \vdash (q_2, 1, 0Z) \vdash (q_2, \epsilon, Z) \vdash (q_0, \epsilon, \epsilon)$
- $(q_0, 001, Z) \vdash (q_1, 01, 0Z) \vdash (q_1, 1, 00Z) \vdash (q_2, \epsilon, 0Z) \vdash$ *error*
- $(q_0, 010, Z) \vdash (q_1, 10, 0Z) \vdash (q_2, 0, Z) \vdash$ *error*

Nondeterministic and Deterministic PDA

- Just as in the case of NFA and DFA, PDA also have two versions: NPDA and DPDA
- However, NPDA are strictly more powerful than the DPDA
- For example, the language, $L = \{ww^R \mid w \in \{a, b\}^+\}$ can be recognized only by an NPDA and not by any DPDA
- In the same breath, the language, $L = \{wcw^R \mid w \in \{a, b\}^+\}$, can be recognized by a DPDA
- In practice we need DPDA, since they have exactly one possible move at any instant
- Our parsers are all DPDA

Parsing

- Parsing is the process of constructing a parse tree for a sentence generated by a given grammar
- If there are no restrictions on the language and the form of grammar used, parsers for context-free languages require $O(n^3)$ time (n being the length of the string parsed)
  - Cocke-Younger-Kasami's algorithm
  - Earley's algorithm
- Subsets of context-free languages typically require $O(n)$ time
  - Predictive parsing using $LL(1)$ grammars (top-down parsing method)
  - Shift-Reduce parsing using $LR(1)$ grammars (bottom-up parsing method)

Top-Down Parsing using LL Grammars

- Top-down parsing using predictive parsing, traces the left-most derivation of the string while constructing the parse tree
- Starts from the start symbol of the grammar, and “predicts” the next production used in the derivation
- Such “prediction” is aided by parsing tables (constructed off-line)
- The next production to be used in the derivation is determined using the next input symbol to lookup the parsing table (look-ahead symbol)
- Placing restrictions on the grammar ensures that no slot in the parsing table contains more than one production
- At the time of parsing table constrcution, if two productions become eligible to be placed in the same slot of the parsing table, the grammar is declared unfit for predictive parsing

An illustration

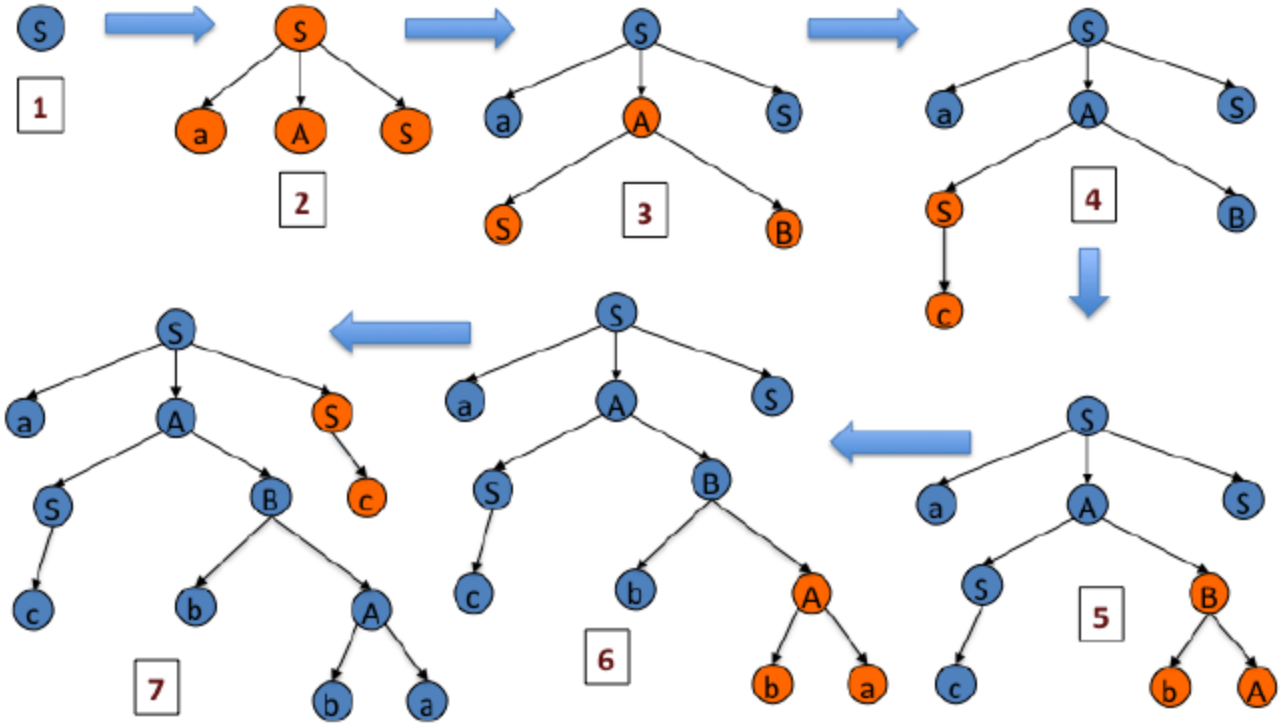
Top-Down LL-Parsing Example

S → aAS | c
A → ba | SB
B → bA | S

Leftmost derivation of the string *acbbac*

S => aAS => aSBS => acBS => acbAS => acbbaS => acbbac

1 2 3 4 5 6 7



Next …

PARSING → TOP-DOWN