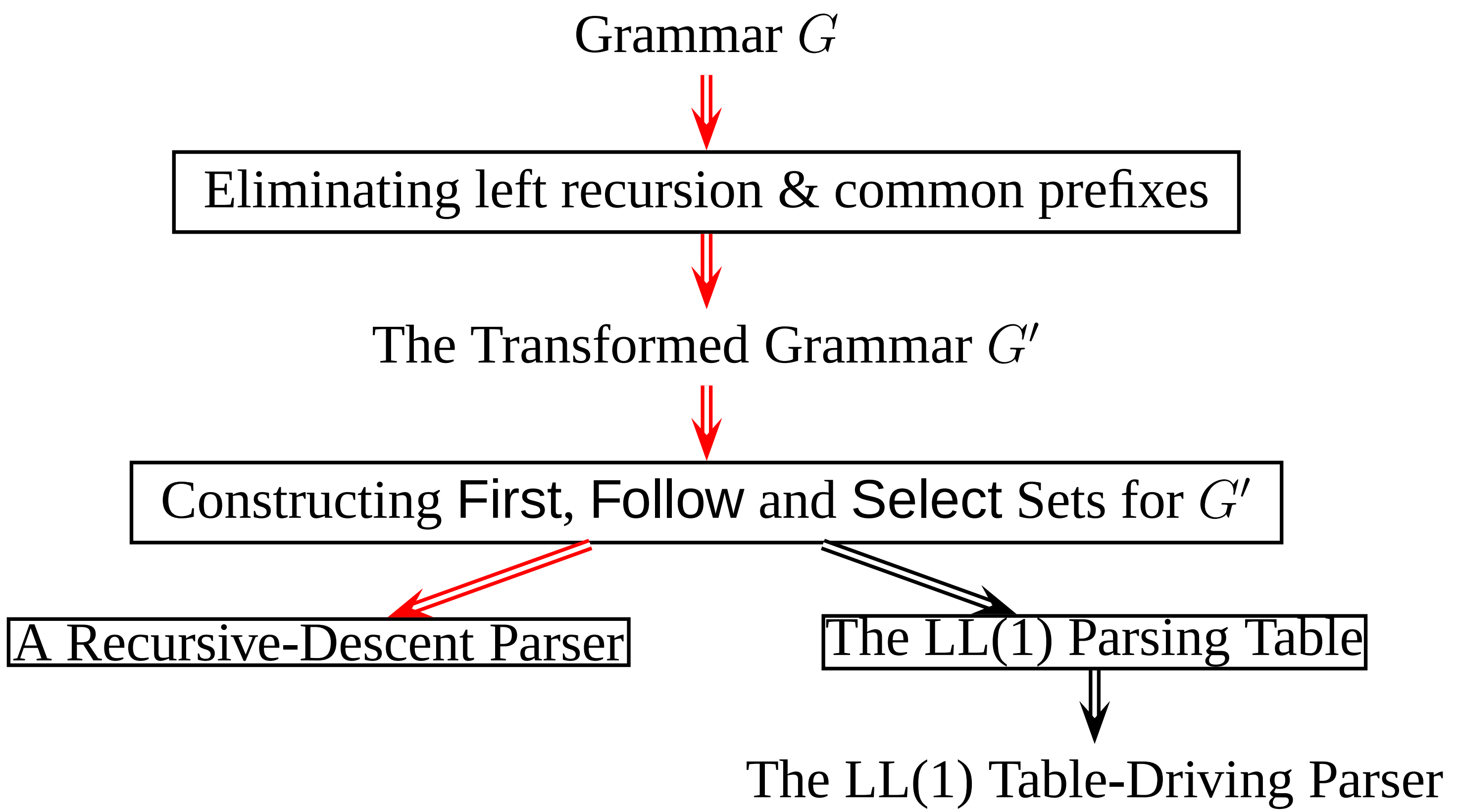
Lecture 4: Top-Down Parsing: Recursive-Descent

1. Compare and contrast top-down and bottom-up parsing
2. Write a predictive (or non-backtracking) top-down parser



The micro-English Grammar Revisited

$$
\begin{array}{lllll}
1 & \langle \text{sentence} \rangle & \to & \langle \text{subject} \rangle \ \langle \text{predicate} \rangle \\
2 & \langle \text{subject} \rangle & \to & \textbf{NOUN} \\
3 & & \mid & \textbf{ARTICLE NOUN} \\
4 & \langle \text{predicate} \rangle & \to & \textbf{VERB} \ \langle \text{object} \rangle \\
5 & \langle \text{object} \rangle & \to & \textbf{NOUN} \\
6 & & \mid & \textbf{ARTICLE NOUN}
\end{array}
$$

The English Sentence

PETER PASSED THE TEST

The micro-English Grammar Revisited (Cont'd)

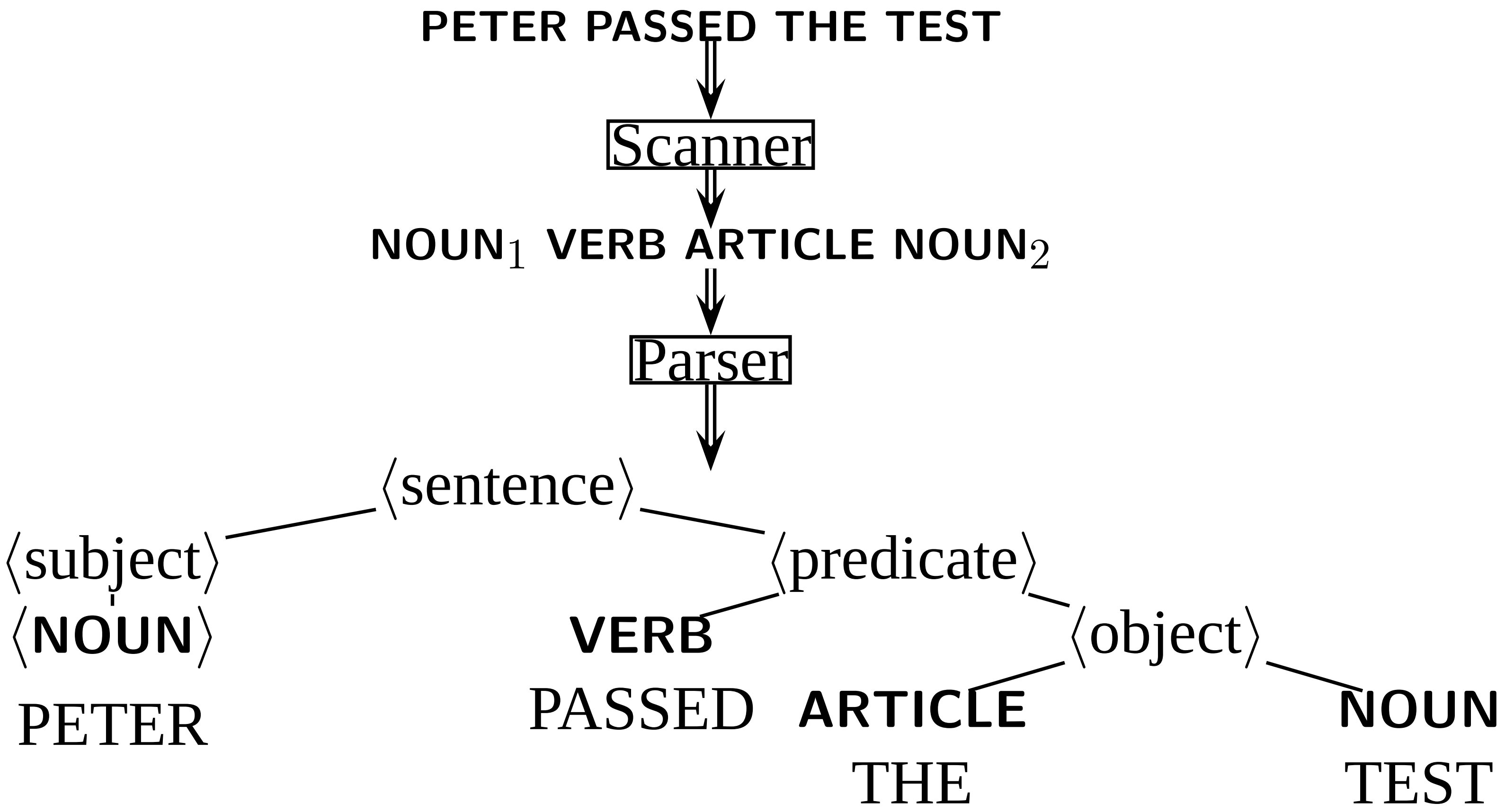
- **The Leftmost Derivation:**

$$
\begin{array}{llll}
\langle \text{sentence} \rangle & \Longrightarrow_{\text{lm}} & \langle \text{subject} \rangle\ \langle \text{predicate} \rangle & \text{by P1} \\
 & \Longrightarrow_{\text{lm}} & \textbf{NOUN}\ \langle \text{predicate} \rangle & \text{by P2} \\
 & \Longrightarrow_{\text{lm}} & \textbf{NOUN VERB}\ \langle \text{object} \rangle & \text{by P4} \\
 & \Longrightarrow_{\text{lm}} & \textbf{NOUN VERB ARTICLE NOUN} & \text{by P6}
\end{array}
$$

- **The Rightmost Derivation:**

$$
\begin{array}{llll}
\langle \text{sentence} \rangle & \Longrightarrow_{\text{rm}} & \langle \text{subject} \rangle\ \langle \text{predicate} \rangle & \text{by P1} \\
 & \Longrightarrow_{\text{rm}} & \langle \text{subject} \rangle\ \textbf{VERB}\ \langle \text{object} \rangle & \text{by P4} \\
 & \Longrightarrow_{\text{rm}} & \langle \text{subject} \rangle\ \textbf{VERB ARTICLE NOUN} & \text{by P6} \\
 & \Longrightarrow_{\text{rm}} & \textbf{NOUN VERB ARTICLE NOUN} & \text{by P2}
\end{array}
$$

The Role of the Parser



Two General Parsing Methods

1. Top-down parsing – Build the parse tree top-down:
   - Productions used represent the leftmost derivation.
   - The best known and widely used methods:
     - Recursive descent
     - Table-driven
     - LL(k) (Left-to-right scan of input, Leftmost derivation, k tokens of lookahead).
     - Almost all programming languages can be specified by LL(1) grammars, but such grammars may not reflect the structure of a language
     - In practice, LL(k) for small k is used
   - Implemented more easily by hand.
   - Used in parser generators such as JavaCC
2. Bottom-up parsing – Build the parse tree bottom-up:
   - Productions used represent the rightmost derivation in reverse.
   - The best known and widely used method: LR(1) (Left-to- right scan of input, Rightmost derivation in reverse, 1 token of lookahead)
   - More powerful – every LL(1) is LR(1) but the converse is false
   - Used by parser generators (e.g., Yacc and JavaCUP).

Lookahead Token(s)

- **Lookahead Token(s)**: The currently scanned token(s) in the input.
- In Recogniser.java, currentToken represents the lookahead token
- For most programming languages, one token lookahead only.
- Initially, the lookahead token is the leftmost token in the input.

Top-Down Parse Of NOUN VERB ARTICLE NOUN

TREE

⟨sentence⟩

↑

INPUT **NOUN**

↑

TREE

⟨sentence⟩

⟨subject⟩ ⟨predicate⟩

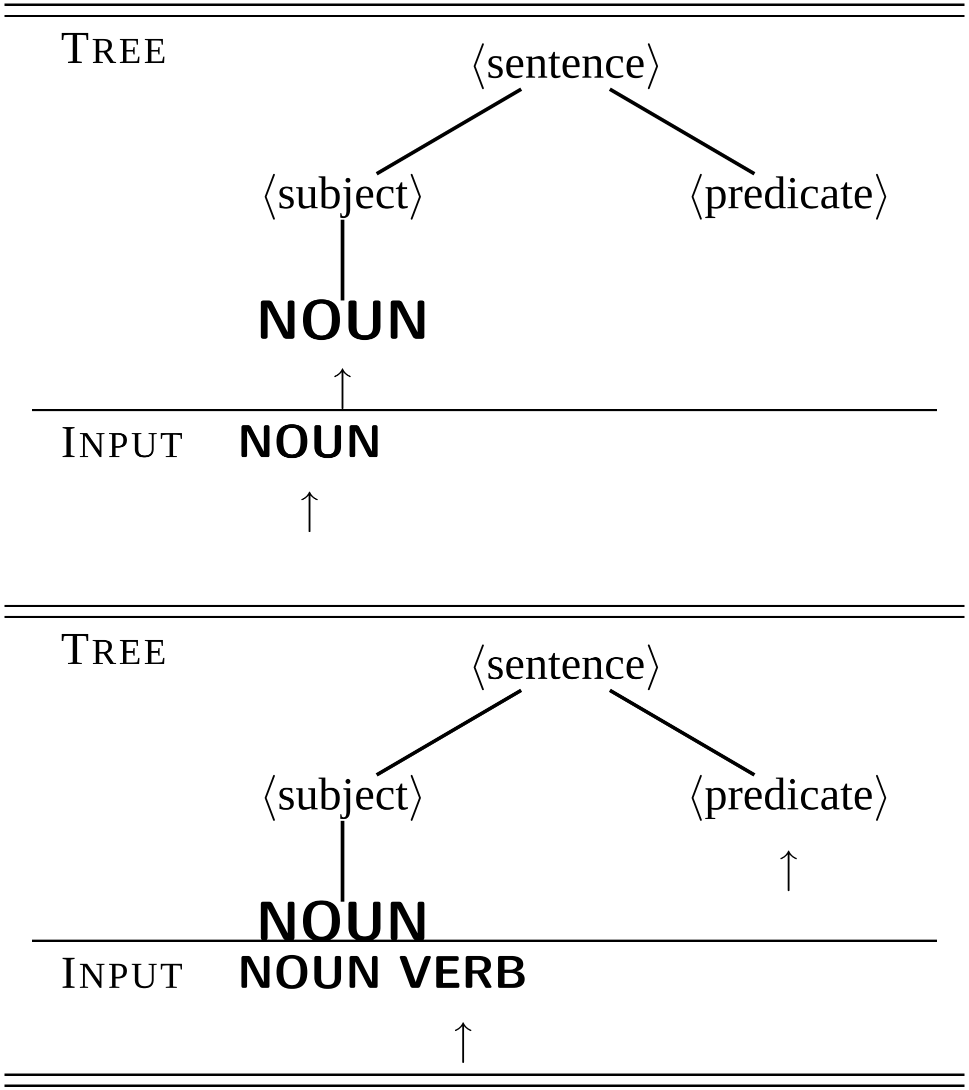
↑

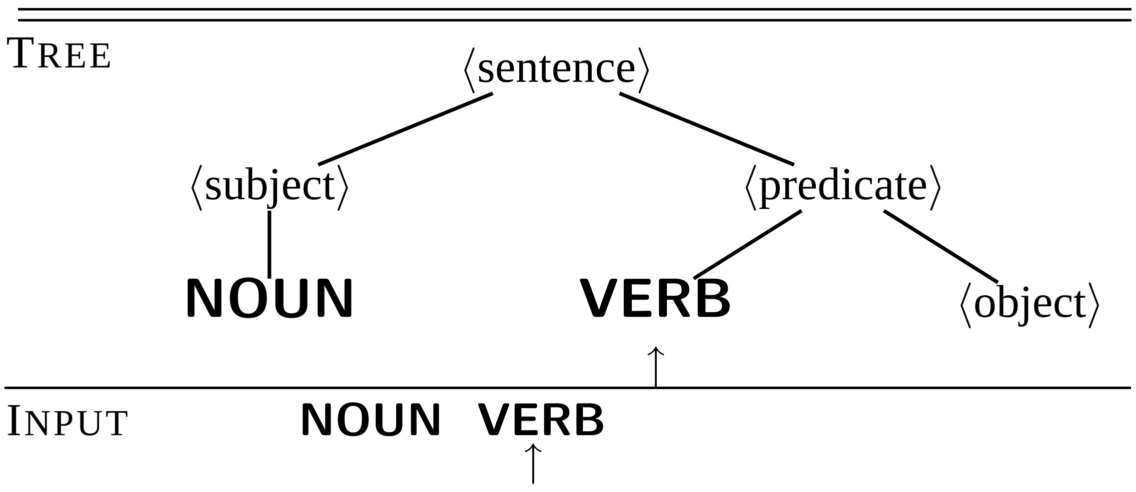
INPUT **NOUN**

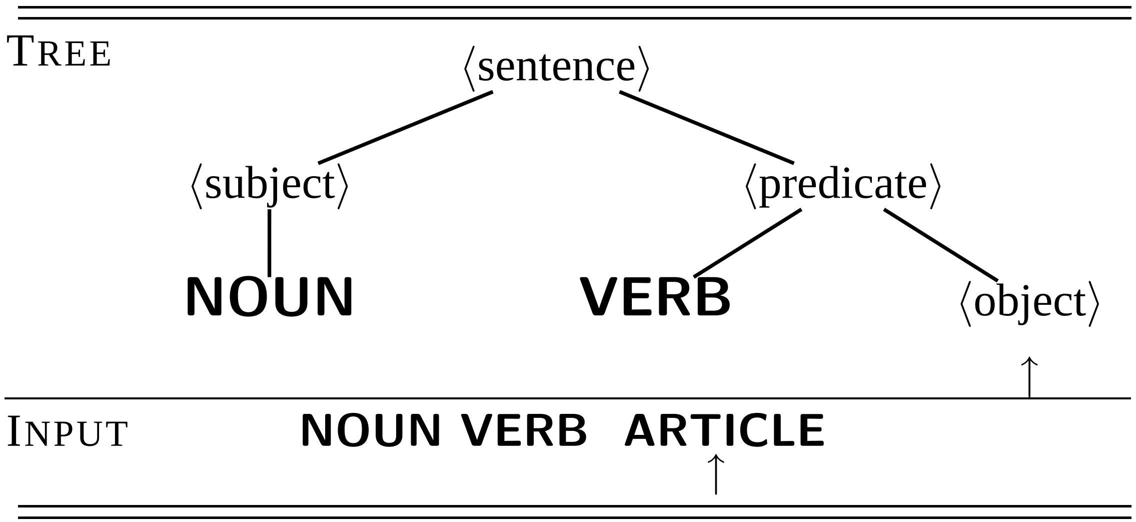
↑

Notations:

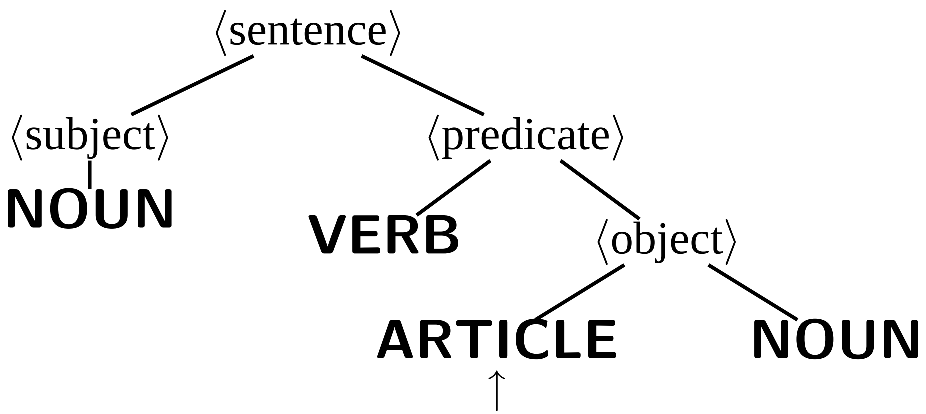
- ↑ on the tree indicates the nonterminal being expanded or recognised
- ↑ on the sentence points to the lookahead token
  - All tokens to the **left** of ↑ have been read
  - All tokens to the **right** of ↑ have **NOT** been processed


TREE
⟨sentence⟩
⟨subject⟩
⟨predicate⟩
NOUN
INPUT
NOUN
TREE
⟨sentence⟩
⟨subject⟩
⟨predicate⟩
NOUN
INPUT
NOUN VERB


TREE
⟨sentence⟩
⟨subject⟩
⟨predicate⟩
NOUN
VERB
⟨object⟩
INPUT
NOUN VERB

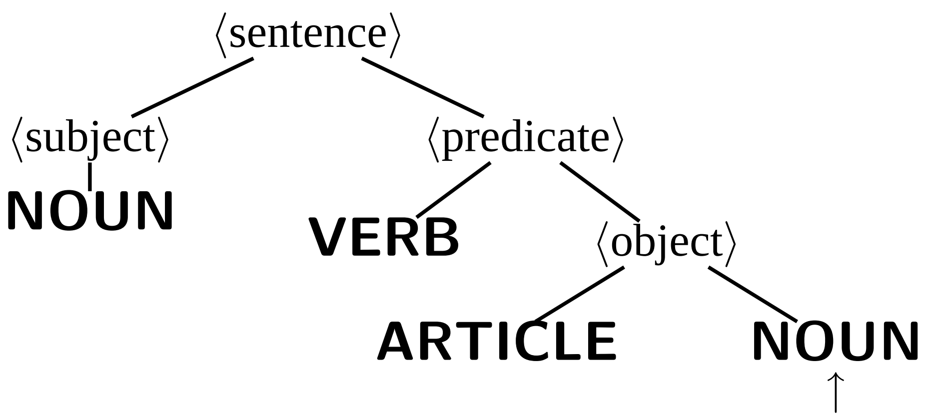

TREE
⟨sentence⟩
⟨subject⟩
⟨predicate⟩
NOUN
VERB
⟨object⟩
INPUT
NOUN VERB ARTICLE

TREE



INPUT **NOUN VERB ARTICLE**

TREE



INPUT **NOUN VERB ARTICLE NOUN**

Top-Down Parsing

- Build the parse tree starting with the start symbol (i.e., the root) towards the sentence being analysed (i.e., leaves).
- Use one token of lookahead, in general
- Discover the leftmost derivation

  I.e, the productions used in expanding the parse tree represent a leftmost derivation

Predictive (Non-Backtracking) Top-Down Parsing

- To expand a nonterminal, the parser always predict (choose) the right alternative for the nonterminal by looking at the lookahead symbol only.
- Flow-of-control constructs, with their distinguishing keywords, are detectable this way, e.g., in the VC grammar:

$$
\begin{array}{lcl}
\langle\text{stmt}\rangle & \rightarrow & \langle\text{compound-stmt}\rangle \\
 & | & \textbf{if}\ ”(”\ \langle\text{expr}\rangle\ ”)”\ (\textbf{ELSE}\ \langle\text{stmt}\rangle)? \\
 & | & \textbf{break}\ ”;” \\
 & | & \textbf{continue}\ ”;” \\
 & \ldots &
\end{array}
$$

- Prediction happens before the actual match begins.

Bottom-Up Parse Of NOUN VERB ARTICLE NOUN

TREE	
INPUT	**NOUN** ↑

TREE	⟨subject⟩ **NOUN**
INPUT	**NOUN** ↑

TREE	⟨subject⟩ **NOUN**
INPUT	**NOUN VERB** ↑

TREE ⟨subject⟩

NOUN

INPUT **NOUN VERB ARTICLE**

↑

TREE ⟨subject⟩

NOUN

INPUT **NOUN VERB ARTICLE NOUN**

↑

TREE ⟨subject⟩ | ⟨object⟩

NOUN | ARTICLE NOUN

INPUT **NOUN VERB ARTICLE NOUN**

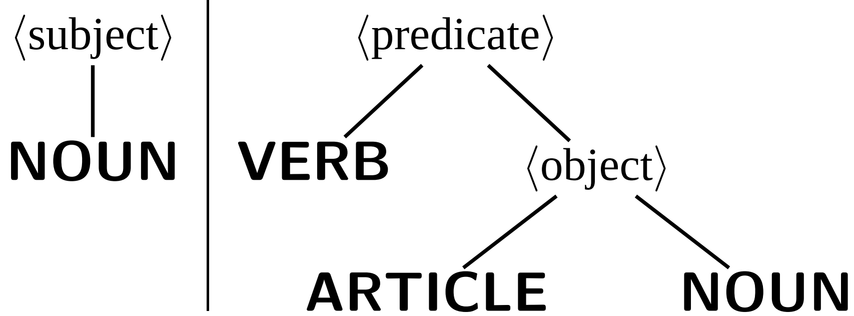
↑

Note: What if the parser had chosen ⟨subject⟩ → **ARTICLE NOUN** instead of ⟨object⟩ → **ARTICLE NOUN**?

In this case, the parser would not make any further process.

Having read a ⟨subject⟩ and **VERB**, the parser has reached a state in which it should not parse another ⟨subject⟩.

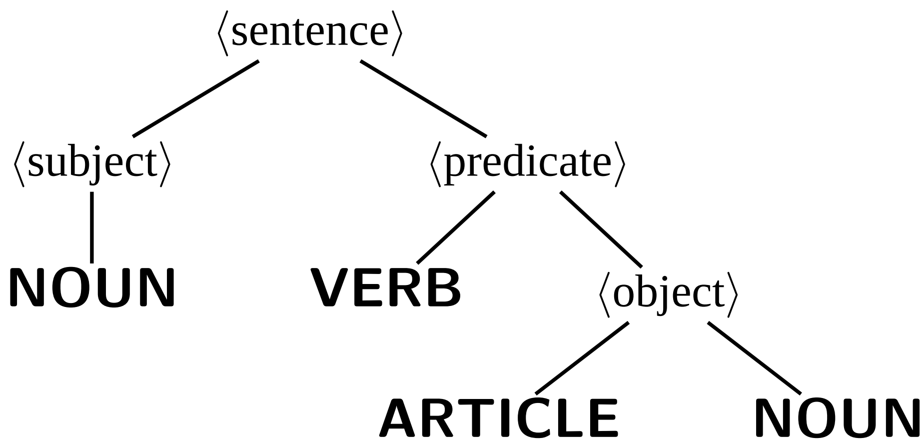
TREE



INPUT **NOUN VERB ARTICLE NOUN**

↑

TREE



INPUT **NOUN VERB ARTICLE NOUN**

↑

Bottom-Up Parsing

- Build the parse tree starting with the the sentence being analysed (i.e., leaves) towards the start symbol (i.e., the root).
- Use one token of lookahead, in general.
- The basic (smallest) language constructs recognised first, then they are used to discover more complex constructs.
- Discover the rightmost derivation in reverse — the productions used in expanding the parse tree represent a rightmost derivation in reverse order

Lecture 4: Top-Down Parsing: Recursive-Descent

1. Compare and contrast top-down and bottom-up parsing $\surd$
2. Write a predictive (or non-backtracking) top-down parser

Grammar G Assignment 2!

$\Downarrow$

Eliminating left recursion & common prefixes

$\Downarrow$

The Transformed Grammar G'

$\Downarrow$

Constructing First, Follow and Select Sets for G'

a Recursive-Descent Parser

The LL(1) Parsing Table

$\Downarrow$

The LL(1) Table-Driving Parser

Which of the Two Alternatives on S to Choose?

- Grammar:

$$
\begin{aligned}
S &\rightarrow aA \mid bB \\
A &\rightarrow \cdots \\
B &\rightarrow \cdots
\end{aligned}
$$

- Sentence: $a \cdots$
- The leftmost derivation:

$$
\begin{aligned}
S &\Longrightarrow_{\text{lm}} aA \\
&\Longrightarrow_{\text{lm}} \cdots
\end{aligned}
$$

Select the first alternative aA

Which of the Two Alternatives on S to Choose?

- Grammar:

$$
\begin{aligned}
S &\rightarrow Ab \mid Bc \\
A &\rightarrow Df \mid CA \\
B &\rightarrow gA \mid e \\
C &\rightarrow dC \mid c \\
D &\rightarrow h \mid i
\end{aligned}
$$

- Sentence: $gchfc$

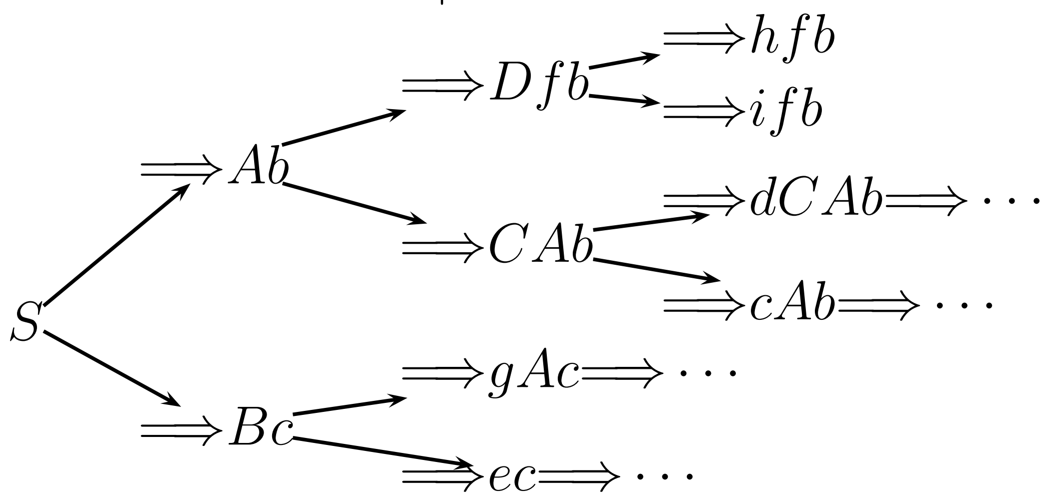
- The leftmost derivation:

$$
\begin{aligned}
S &\Longrightarrow_{\text{lm}} Bc \Longrightarrow_{\text{lm}} gAc \Longrightarrow_{\text{lm}} gCAc \\
&\Longrightarrow_{\text{lm}} gcAc \Longrightarrow_{\text{lm}} gcDfc \Longrightarrow_{\text{lm}} gchfc
\end{aligned}
$$

Intuition behind First Sets

- Grammar:

$$\begin{aligned} S &\rightarrow Ab \mid Bc \\ A &\rightarrow Df \mid CA \\ B &\rightarrow gA \mid e \\ C &\rightarrow dC \mid c \\ D &\rightarrow h \mid i \end{aligned}$$



- All possible leftmost derivations:

  First$(Ab) = \{c, d, h, i\}$ First$(Bc) = \{e, g\}$

Definition of First Sets

First(α):

- The set of all terminals that can begin any strings derived from α.
- if $\alpha \Longrightarrow^* \epsilon$, then ϵ is also in First(α)

Nullable Nonterminals

A nonterminal A is nullable if $A \Longrightarrow^* \epsilon$.

A Procedure to Compute $\mathsf{First}(\alpha)$

1. Case 1: α is a single symbol or ϵ:

   If α is a terminal a, then $\mathsf{First}(\alpha) = \mathsf{First}(a) = \{a\}$

   else if α is ϵ, then $\mathsf{First}(\alpha) = \mathsf{First}(\epsilon) = \{\epsilon\}$

   else if α is a nonterminal and $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \cdots$ then

   $$\mathsf{First}(\alpha) = \cup_k \mathsf{First}(\beta_k)$$

2. Case 2: $\alpha = X_1 X_2 \cdots X_n$:

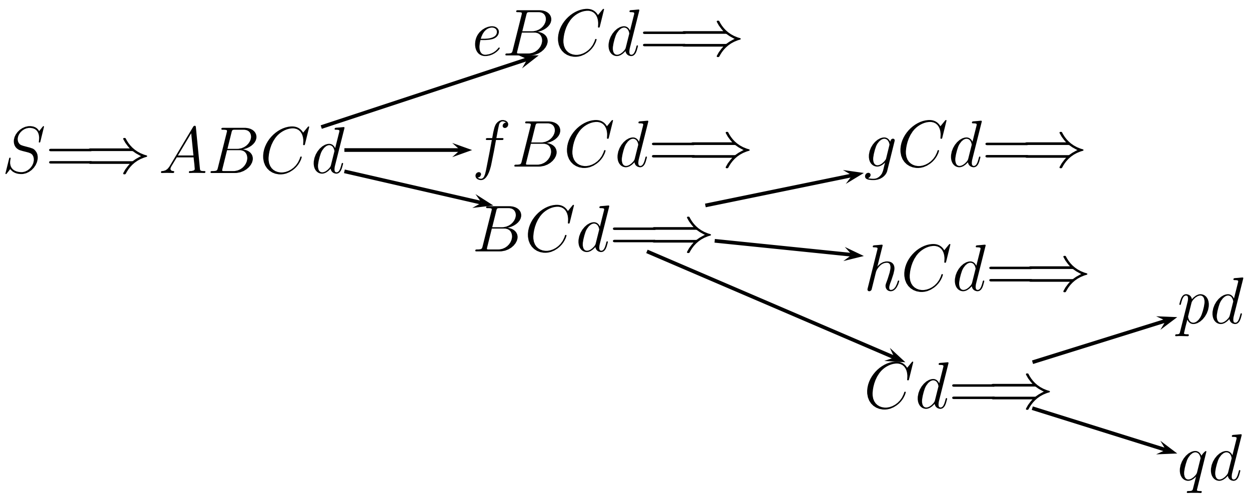
   If $X_1 X_2 \ldots X_i$ is nullable but X_{i+1} is not, then

   $$\mathsf{First}(\alpha) = \mathsf{First}(X_1) \cup \mathsf{First}(X_2) \cup \cdots \cup \mathsf{First}(X_{i+1})$$

   If $i = n$ is nullable, then add ϵ to $\mathsf{First}(\alpha)$

Case 2 of the Procedure for Computing First

$$
\begin{aligned}
S &\to ABCd \\
A &\to e \mid f \mid \epsilon \\
B &\to g \mid h \mid \epsilon \\
C &\to p \mid q
\end{aligned}
$$

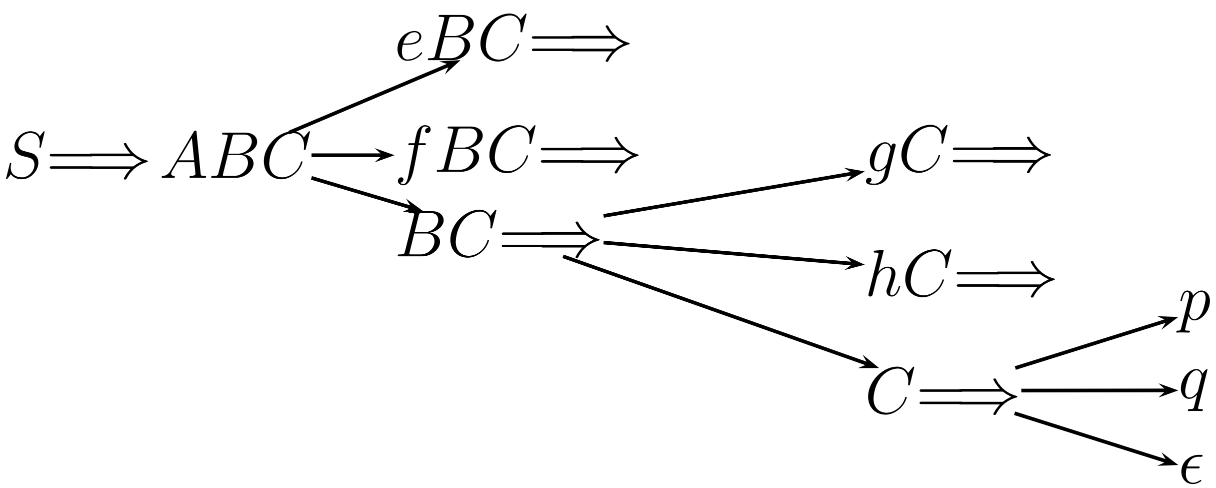


$\text{First}(ABCd) = \{e, f, g, h, p, q\}$

Case 2 of the Procedure for Computing First Again

$$S \rightarrow ABC$$
$$A \rightarrow e \mid f \mid \epsilon$$
$$B \rightarrow g \mid h \mid \epsilon$$
$$C \rightarrow p \mid q \mid \epsilon$$

d was deleted from the grammar in slide 218



$$\text{First}(ABC) = \{e, f, g, h, p, q, \epsilon\}$$

The Expression Grammar

- The grammar with left recursion:

$$\text{Grammar 1:}\quad \begin{aligned} E &\to E + T \mid E - T \mid T \\ T &\to T * F \mid T/F \mid F \\ F &\to \textsf{INT} \mid (E) \end{aligned}$$

- The transformed grammar without left recursion:

$$\text{Grammar 2:}\quad \begin{aligned} E &\to TQ \\ Q &\to +TQ \mid -TQ \mid \epsilon \\ T &\to FR \\ R &\to *FR \mid /FR \mid \epsilon \\ F &\to \textsf{INT} \mid (E) \end{aligned}$$

First Sets for Grammar 1 (with Recursion)

$$
\begin{array}{lclclcl}
\mathsf{First}(E) & = & \mathsf{First}(E+T) & = & \mathsf{First}(E-T) & = \\
\mathsf{First}(T) & = & \mathsf{First}(T*F) & = & \mathsf{First}(T/F) & = \\
\mathsf{First}(F) & = & \{(, \textbf{INT}\} \\
\mathsf{First}((E)) & = & \{(\} \\
\mathsf{First}(\textbf{INT}) & = & \{\textbf{INT}\}
\end{array}
$$

The explicit construction is left as an exercise.

First Sets for Grammar 2 (without Recursion)

$$
\begin{array}{lcl}
\text{First}(E) = \text{First}(TQ) & = & \{(, \textbf{INT}\} \\
\text{First}(T) = \text{First}(FR) & = & \{(, \textbf{INT}\} \\
\text{First}(Q) & = & \{+, -, \epsilon\} \\
\text{First}(R) & = & \{*, /, \epsilon\} \\
\text{First}(F) & = & \{\textbf{INT}, (\} \\
\text{First}(+TQ) & = & \{+\} \\
\text{First}(-TQ) & = & \{-\} \\
\text{First}(*FR) & = & \{*\} \\
\text{First}(/FR) & = & \{/\} \\
\text{First}((E)) & = & \{(\} \\
\text{First}(\textbf{INT}) & = & \{\textbf{INT}\}
\end{array}
$$

Why Follow Sets?

- First sets do not tell us when to apply $A \rightarrow \alpha$ such that $\alpha \Longrightarrow^* \epsilon$ (the important special case is $A \rightarrow \epsilon$)
- Follow sets do
- Follow sets constructed only for nonterminals
- By convention, assume every input is terminated by a special end marker (i.e., the EOF marker), denoted $
- Follow sets do not contain ϵ

Definition of Follow Sets

Let A be a nonterminal. Define $\textsf{Follow}(A)$ to be the set of terminals that can appear immediately to the right of A in some sentential form. That is,

$$\textsf{Follow}(A) \;=\; \{a \mid S \Longrightarrow^{*} \cdots Aa \cdots\}$$

where S is the start symbol of the grammar.

A Procedure to Compute Follow Sets

1. If A is the start symbol, add \$ to Follow(A).
2. Look through the grammar for all occurrences of A on the right of productions. Let a typical production be:

$$B \rightarrow \alpha A \beta$$

   There are two cases – both may be applicable:

   (a) Follow(A) includes First$(\beta) - \{\epsilon\}$.

   (b) If $\beta \Longrightarrow^* \epsilon$, then include Follow$(B)$ in Follow(A).

Follow Sets for Grammar 1 (with Recursion)

$$
\begin{aligned}
\mathsf{Follow}(E) &= \{+, -,), \$\} \\
\mathsf{Follow}(T) = \mathsf{Follow}(F) &= \{+, -, *, /,), \$\}
\end{aligned}
$$

The explicit construction is left as an exercise.

Follow Sets for Grammar 2 (without Recursion)

$$
\begin{aligned}
\text{Follow}(E) &= \{), \$\} \\
\text{Follow}(Q) &= \{), \$\} \\
\text{Follow}(T) &= \{+, -,), \$\} \\
\text{Follow}(R) &= \{+, -,), \$\} \\
\text{Follow}(F) &= \{+, -, *, /,), \$\}
\end{aligned}
$$

Select Sets for Productions

- One Select set for every production in the grammar:
- The Select set for a production of the form $A \to \alpha$ is:
  - If $\epsilon \in \text{First}(\alpha)$, then

$$\text{Select}(A \to \alpha) \;=\; (\text{First}(\alpha) - \{\epsilon\}) \cup \text{Follow}(A)$$

  - Otherwise:

$$\text{Select}(A \to \alpha) \;=\; \text{First}(\alpha)$$

- The Select set predicts $A \to \alpha$ to be used in a derivation.
- Thus, the Select not needed if A has has one alternative

Select Sets for Grammar 1

Follow sets not used since the grammar has no ϵ-productions

$$
\begin{array}{lll}
\text{Select}(E \rightarrow E + T) & = \text{First}(E + T) & = \{(, \textbf{INT}\} \\
\text{Select}(E \rightarrow E - T) & = \text{First}(E - T) & = \{(, \textbf{INT}\} \\
\text{Select}(E \rightarrow T) & = \text{First}(T) & = \{(, \textbf{INT}\} \\
\text{Select}(T \rightarrow T * F) & = \text{First}(T * F) & = \{(, \textbf{INT}\} \\
\text{Select}(T \rightarrow T / F) & = \text{First}(T / F) & = \{(, \textbf{INT}\} \\
\text{Select}(T \rightarrow F) & = \text{First}(F) & = \{(, \textbf{INT}\} \\
\text{Select}(F \rightarrow \textbf{INT}) & = \text{First}(\textbf{INT}) & = \{\textbf{INT}\} \\
\text{Select}(F \rightarrow (E)) & = \text{First}((E)) & = \{(\}
\end{array}
$$

Select Sets for Grammar 2

$$
\begin{array}{lll}
\text{Select}(E\rightarrow TQ) & = \text{First}(TQ) & = \{(, \textbf{INT}\} \\
\text{Select}(Q\rightarrow +TQ) & = \text{First}(+TQ) & = \{+\} \\
\text{Select}(Q\rightarrow -TQ) & = \text{First}(-TQ) & = \{-\} \\
\text{Select}(Q\rightarrow \epsilon) & = (\text{First}(\epsilon) - \{\epsilon\}) \cup \text{Follow}(Q) & = \{), \$\} \\
\text{Select}(T\rightarrow FR) & = \text{First}(FR) & = \{(, \textbf{INT}\} \\
\text{Select}(R\rightarrow *FR) & = \text{First}(+FR) & = \{*\} \\
\text{Select}(R\rightarrow /FR) & = \text{First}(/FR) & = \{/\} \\
\text{Select}(R\rightarrow \epsilon) & = (\text{First}(\epsilon) - \{\epsilon\}) \cup \text{Follow}(T) & = \{+, -,), \$\} \\
\text{Select}(F\rightarrow \textbf{INT}) & = \text{First}(\textbf{INT}) & = \{\textbf{INT}\} \\
\text{Select}(F\rightarrow (E)) & = \text{First}((E)) & = \{(\}
\end{array}
$$

Outline for the Rest of the Lecture

1. Definition of LL(1) grammar
2. One simplification in the presence of a nullable alternative
3. Eliminate left recursion and common prefixes
4. Write parsing methods in the presence of regular operators
5. $LL(k)$ for small k often necessary for some constructs

Definition of LL(1) Grammar

- A grammar is LL(1) if for every nonterminal of the form:

$$A \;\to\; \alpha_1 \mid \cdots \mid \alpha_n$$

  the select sets are pairwise disjoint, i.e.:

$$\textsf{Select}(A\to\alpha_i) \cap \textsf{Select}(A\to\alpha_j) \;=\; \emptyset$$

  for all i and j such that $i \neq j$.

- This implies there can be at most one nullable alternative

Left Recursion

- Direct left-recursion:

$$A \rightarrow A\alpha$$

- Non-direct left-recursion:

$$\begin{aligned} A &\rightarrow B\alpha \\ B &\rightarrow A\beta \end{aligned}$$

  - Algorithm 4.1 of text eliminates both kinds of left recursion
  - In real programming languages, non-direct left-recursion is rare
  - Not required

- A grammar with left recursion is not LL(1)

Eliminating Direct Left Recursion

- The grammar G_1:

$$\begin{array}{lcl} A & \rightarrow & \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n \quad \text{// } \alpha_i \text{ does not beging with } A \\ A & \rightarrow & A\beta_1 \mid A\beta_2 \mid \cdots \mid A\beta_m \end{array}$$

- The transformed grammar G_2:

$$\begin{array}{lcl} A & \rightarrow & \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A' \\ A' & \rightarrow & \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_m A' \mid \epsilon \end{array}$$

- G_1 and G_2 define the same language: $L(G_1) = L(G_2)$
- Example: in Slide 220, Grammar 2 is the transformed version of Grammar 1

Eliminating Direct Left Recursion: Special Case

- The grammar G_1:

$$\begin{array}{lcll} A & \rightarrow & \alpha & \text{// } \alpha \text{ does not beging with } A \\ A & \rightarrow & A\beta & \end{array}$$

- The transformed grammar G_2:

$$\begin{array}{lcl} A & \rightarrow & \alpha A' \\ A' & \rightarrow & \beta A' \mid \epsilon \end{array}$$

Eliminating Common Prefixes: Left-Factoring

- The grammar G_1:

$$\begin{array}{lcl} A & \to & \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_m \\ A & \to & \gamma \end{array}$$

- The transformed grammar G_2:

$$\begin{array}{lcl} A & \to & \alpha A' \\ A & \to & \gamma \\ A' & \to & \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m \end{array}$$

- $L(G_1) = L(G_2)$
- A grammar with common prefixes is not LL(1)

Example: Eliminating Common Prefixes

$$
\begin{array}{lcl}
\langle \text{stmt} \rangle & \longrightarrow & \textbf{IF} \text{ "(" } \langle \text{expr} \rangle \text{ ")" } \langle \text{stmt} \rangle \ \textbf{ELSE} \ \langle \text{stmt} \rangle \\
\langle \text{stmt} \rangle & \longrightarrow & \textbf{IF} \text{ "(" } \langle \text{expr} \rangle \text{ ")" } \langle \text{stmt} \rangle \\
\langle \text{stmt} \rangle & \longrightarrow & \textbf{other}
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{lcl}
\langle \text{stmt} \rangle & \longrightarrow & \textbf{IF} \text{ "(" } \langle \text{expr} \rangle \text{ ")" } \langle \text{stmt} \rangle \ \langle \text{else-cluase} \rangle \\
\langle \text{else-clause} \rangle & \longrightarrow & \textbf{ELSE} \ \langle \text{stmt} \rangle \ \mid \epsilon \\
\langle \text{stmt} \rangle & \longrightarrow & \textbf{other}
\end{array}
$$

Eliminating Direct Left Recursion Using Regular Operators

- The grammar G_1:

$$\begin{array}{rcl} A & \rightarrow & \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n \\ A & \rightarrow & A\beta_1 \mid A\beta_2 \mid \cdots \mid A\beta_m \end{array}$$

- The transformed grammar G_2:

$$A \rightarrow (\alpha_1 \mid \cdots \mid \alpha_n)(\beta_1 \mid \cdots \mid \beta_m)^*$$

- G_1 and G_2 define the same language: $L(G_1) = L(G_2)$
- Recommended to use in Assignment 2, where $n = m = 1$ for most left-recursive cases:

$$A \rightarrow \alpha\beta^*$$

The Expression Grammar

- The grammar with left recursion:

$$
\begin{aligned}
\text{Grammar 1:}\quad & E \rightarrow E + T \mid E - T \mid T \\
& T \rightarrow T * F \mid T/F \mid F \\
& F \rightarrow \textbf{INT} \mid (E)
\end{aligned}
$$

- Eliminating left recursion using the Kleene Closure

$$
\begin{aligned}
\text{Grammar 3:}\quad & E \rightarrow T\ (\text{"+"}\ T \mid \text{"-"}\ T)^* \\
& T \rightarrow F\ (\text{"*"}\ F \mid \text{"/"}\ F)^* \\
& F \rightarrow \textbf{INT} \mid \text{"("}\ E\ \text{")"}
\end{aligned}
$$

  All tokens are enclosed in double quotes to distinguish them for the regular operators: (,) and *

- Compare with Slide 220

Eliminating Common Prefixes using Choice Operator

- The grammar G_1:

$$\begin{array}{lcl} A & \to & \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_m \\ A & \to & \gamma \end{array}$$

- The transformed grammar G_2:

$$\begin{array}{lcl} A & \to & \alpha(\beta_1 \mid \beta_2 \mid \cdots \mid \beta_m) \\ A & \to & \gamma \end{array}$$

- Recommended to use in Assignment 2

Example: Eliminating Common Prefixes

$$\begin{array}{lcl} \langle \text{stmt} \rangle & \longrightarrow & \textbf{IF} \text{ "(" } \langle \text{expr} \rangle \text{ ")" } \langle \text{stmt} \rangle \ \textbf{ELSE} \ \langle \text{stmt} \rangle \\ \langle \text{stmt} \rangle & \longrightarrow & \textbf{IF} \text{ "(" } \langle \text{expr} \rangle \text{ ")" } \langle \text{stmt} \rangle \\ \langle \text{stmt} \rangle & \longrightarrow & \textbf{other} \end{array}$$

$$\Downarrow$$

$$\begin{array}{lcl} \langle \text{stmt} \rangle & \longrightarrow & \textbf{IF} \text{ "(" } \langle \text{expr} \rangle \text{ ")" } \langle \text{stmt} \rangle \ (\ \textbf{ELSE} \ \langle \text{stmt} \rangle)? \\ \langle \text{stmt} \rangle & \longrightarrow & \textbf{other} \end{array}$$

Compare with Slide 251

LL(k) Grammar and Parsing

- A grammar is LL(k) if it can be parsed deterministically using k tokens of lookahead
- A formal definition for LL(k) grammars can be found in Grune and Jacobs' book

es

- Grammar 1 in Slide 220 is not LL(k) for any k!
- However, Grammar 2 in Slide 220 is LL(1)
- Only a understanding of LL(1) is required this year ear