# Parsing

a.y. 2022-2023





## Parsing

- Given a grammar $\mathcal{G} = (V, T, S, \mathcal{P})$ and a word $w$
- Say whether $w \in \mathcal{L}(\mathcal{G})$ and, if so, provide its derivation tree

- Two relevant kinds of parsing
  - Top-down: construct leftmost derivation from the root to the yield
  - Bottom-up: construct rightmost derivation (in reverse order) from the yield to the root

## Top-down parsing

- Let $w = bd$ and

$$\mathcal{G}: \begin{array}{lcl} S & \to & Ad \mid Bd \\ A & \to & a \\ B & \to & b \end{array}$$

- Let us try to derive $w$
- If we choose $S \Rightarrow Ad$ we fail to get $bd$
- If we choose $S \Rightarrow Bd$ we get it through the derivation $S \Rightarrow Bd \Rightarrow bd$





## Top-down parsing

- Let $w = id + id * id$ and

$$\mathcal{G}: \begin{array}{lcl} E & \to & TE' \\ E' & \to & +TE' \mid \epsilon \\ T & \to & FT' \\ T' & \to & *FT' \mid \epsilon \\ F & \to & (E) \mid id \end{array}$$

- What is, if any, a leftmost derivation of $w$ in $\mathcal{G}$?

## Top-down parsing

- Let $w = cad$ and

$$\mathcal{G}: \begin{array}{lcl} S & \rightarrow & cAd \\ A & \rightarrow & ab \mid a \end{array}$$

- What is a leftmost derivation of $w$ in $\mathcal{G}$?
- At the first step we get $S \Rightarrow cAd$
- By choosing $A \rightarrow ab$ we get $S \Rightarrow cabd$
- Bad luck, backtrack





## Predictive top-down parsing

- No backtrack
- LL(1) grammars

# Predictive top-down parsing

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid \epsilon \\ F & \rightarrow & (E) \mid id \end{array}$$

For LL(1) grammars we can set up a parsing table that can drive leftmost derivations

| | $id$ | $+$ | $*$ | ( | ) | \$ |
|---|---|---|---|---|---|---|
| $E$ | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | | |
| $E'$ | | $E' \rightarrow +TE'$ | | | $E' \rightarrow \epsilon$ | $E' \rightarrow \epsilon$ |
| $T$ | $T \rightarrow FT'$ | | | $T \rightarrow FT'$ | | |
| $T'$ | | $T' \rightarrow \epsilon$ | $T' \rightarrow *FT'$ | | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ |
| $F$ | $F \rightarrow id$ | | | $F \rightarrow (E)$ | | |





# Predictive top-down parsing

- We have a word, say $id + id * id$, and we must derive it from $E$
- Check the table entry $[E, id]$
- The entry contains $E \rightarrow TE'$, take the derivation step $E \Rightarrow TE'$
- The current leftmost non-terminal is $T$, the current symbol is $id$, then check the entry $[E, id]$
- The entry contains $T \rightarrow FT'$, take the derivation step $TE' \Rightarrow FT'E'$
- The leftmost non-terminal is $F$, the current symbol is $id$, then check the entry $[F, id]$
- The entry contains $F \rightarrow id$, take the derivation step $FT'E' \Rightarrow idT'E'$
- The first occurrence of $id$ in $id + id * id$ was generated, move the input pointer forward to point to '$+$' and go on checking the entry $[T', +]$

# Implementation: $w = id + id * id$

| stack | input | output | leftmost derivation |
|---|---|---|---|
| $\$E$ | $id + id * id\$$ | $E \to TE'$ | $E \Rightarrow TE'$ |
| $\$E'T$ | $id + id * id\$$ | $T \to FT'$ | $\Rightarrow FT'E'$ |
| $\$E'T'F$ | $id + id * id\$$ | $F \to id$ | $\Rightarrow id\,T'E'$ |
| $\$E'T'id$ | $id + id * id\$$ | | |
| $\$E'T'$ | $+id * id\$$ | $T' \to \epsilon$ | $\Rightarrow id\,E'$ |
| $\$E'$ | $+id * id\$$ | $E' \to +TE'$ | $\Rightarrow id+TE'$ |
| $\$E'T+$ | $+id * id\$$ | | |
| $\$E'T$ | $id * id\$$ | $T \to FT'$ | $\Rightarrow id+FT'E'$ |
| $\$E'T'F$ | $id * id\$$ | $F \to id$ | $\Rightarrow id+id\,T'E'$ |
| $\$E'T'id$ | $id * id\$$ | | |
| $\$E'T'$ | $*id\$$ | $T' \to *FT'$ | $\Rightarrow id+id*FT'E'$ |
| $\$E'T'F*$ | $*id\$$ | | |
| $\$E'T'F$ | $id\$$ | $F \to id$ | $\Rightarrow id+id*id\,T'E'$ |
| $\$E'T'id$ | $id\$$ | | |
| $\$E'T'$ | $\$$ | $T' \to \epsilon$ | $\Rightarrow id+id*id\,E'$ |
| $\$E'$ | $\$$ | $E' \to \epsilon$ | $\Rightarrow id+id*id$ |
| $\$$ | $\$$ | | |





# Algorithm for predictive top-down parsing

- **Input:**
  string $w$; top-down parsing table $M$ for $\mathcal{G} = (V, T, S, \mathcal{P})$
- **Output:**
  leftmost derivation of $w$ if $w \in \mathcal{L}(\mathcal{G})$, error() otherwise
- **Initialization:**
  $w\$$ in the input buffer; $\$S$ onto the stack, with $S$ on top

## Algorithm for predictive top-down parsing

**let** *b be the first symbol of* $w\$$ *;*
**let** $X$ *be the top of the stack ;*
**while** $X \neq \$$ **do**
  **if** $X = b$ **then**
    pop $X$ ;
    **let** $b$ *be the next symbol of* $w\$$ ;
  **else if** $X$ *is a terminal* **then** error();
  **else if** $M[X, b]$ *is error* **then** error();
  **else if** $M[X, b] = X \to Y_1 \dots Y_k$ **then**
    output($X \to Y_1 \dots Y_k$) ;
    pop $X$ ;
    push $Y_k$ ;... ; push $Y_1$ ;
  **let** $X$ *be the top of the stack ;*





## Parsing tables

How do we fill the entries of parsing tables?

The entry $M[A, b]$ is consulted to expand $A$ when the next input character is $b$

Then we set $M[A, b] = A \to \alpha$ if

- Either $\alpha \Rightarrow^* b\beta$
- Or $\alpha \Rightarrow^* \epsilon$ and we can have $S \Rightarrow^* wA\gamma$ with $\gamma \Rightarrow^* b\beta$

# Parsing tables

Take the grammar

$$
\begin{array}{lcl}
S & \rightarrow & aA \mid bB \\
A & \rightarrow & c \\
B & \rightarrow & c
\end{array}
$$

How would you fill the table for it?

| | $a$ | $b$ | $c$ | $\$$ |
|---|---|---|---|---|
| $S$ | | | | |
| $A$ | | | | |
| $B$ | | | | |





# Parsing tables

Take the grammar

$$
\begin{array}{lcl}
S & \rightarrow & aAb \\
A & \rightarrow & \epsilon
\end{array}
$$

How would you fill the table for it?

| | $a$ | $b$ | $\$$ |
|---|---|---|---|
| $S$ | | | |
| $A$ | | | |

## *first*($\alpha$)

Set of terminals that begin strings derived from $\alpha$

Also, if $\alpha \Rightarrow^* \epsilon$ then $\epsilon \in \mathrm{first}(\alpha)$





## *first*($\alpha$)

$\mathrm{first}(\epsilon) = \{\epsilon\}$

$\mathrm{first}(a) = \{a\}$

$\mathrm{first}(A) = \bigcup_{A \to \alpha} \mathrm{first}(\alpha)$

Computation of $\mathrm{first}(Y_1 \dots Y_n)$:

$\mathrm{first}(Y_1 \dots Y_n) = \emptyset$;
$j = 1$ ;
**while** $j \leq n$ **do**
  add $\mathrm{first}(Y_j) \setminus \{\epsilon\}$ to $\mathrm{first}(Y_1 \dots Y_n)$ ;
  **if** $\epsilon \in \mathrm{first}(Y_j)$ **then** $j = j + 1$ ;
  **else** break ;
**if** $j = n + 1$ **then** add $\epsilon$ to $\mathrm{first}(Y_1 \dots Y_n)$ ;

# $first(\alpha)$
**TRAINING**

$$
\begin{array}{lcl}
E & \rightarrow & TE' \\
E' & \rightarrow & +TE' \mid \epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT' \mid \epsilon \\
F & \rightarrow & (E) \mid id
\end{array}
$$



Formal Languages and Compilers Paola Quaglia, 2022

# $first(\alpha)$
**TRAINING**

$$
\begin{array}{lcl}
E & \rightarrow & TE' \\
E' & \rightarrow & +TE' \mid \epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT' \mid \epsilon \\
F & \rightarrow & (E) \mid id
\end{array}
$$

| | first | |
|---|---|---|
| E | {id, (} | |
| E' | {$\epsilon$,+} | |
| T | {id, (} | |
| T' | {$\epsilon$,*} | |
| F | {id, (} | |

# follow($A$)

- follow($A$) is the set of terminals that can follow $A$ in some derivation





# *follow*($A$)

```
follow(S) = {$};
foreach A ≠ S do
    follow(A) = ∅;
repeat
    foreach B → αAβ do
        if β ≠ ε then
            add first(β) \ {ε} to follow(A)
        if β = ε or ε ∈ first(β) then
            add follow(B) to follow(A)
until saturation;
```

# *follow*$(A)$
## TRAINING

$$
\begin{array}{lcl}
E & \to & TE' \\
E' & \to & +TE' \mid \epsilon \\
T & \to & FT' \\
T' & \to & *FT' \mid \epsilon \\
F & \to & (E) \mid id
\end{array}
$$





# *follow*$(A)$
## TRAINING

$$
\begin{array}{lcl}
E & \to & TE' \\
E' & \to & +TE' \mid \epsilon \\
T & \to & FT' \\
T' & \to & *FT' \mid \epsilon \\
F & \to & (E) \mid id
\end{array}
$$

**foreach** $B \to \alpha A \beta$ **do**
  **if** $\beta \neq \epsilon$ **then**
    add $\text{first}(\beta) \setminus \{\epsilon\}$ to $\text{follow}(A)$
  **end**
  **if** $\beta = \epsilon$ *or* $\epsilon \in \text{first}(\beta)$ **then**
    add $\text{follow}(B)$ to $\text{follow}(A)$
  **end**
**end**

| | first | computation of follow | follow |
|---|---|---|---|
| E | {id, (} | $ ; ) | {$,)} |
| E' | {ϵ,+} | add follow(E) | {$,)} |
| T | {id, (} | +; add follow(E ); +; add follow(E') | {$,),+} |
| T' | {ϵ,*} | add follow(T) | {$,),+} |
| F | {id, (} | *; add follow(T); *; add follow(T') | {$,),+,*} |

# $follow(A)$
## TRAINING

$$
\begin{array}{lcl}
S & \to & aABb \\
A & \to & Ac \mid d \\
B & \to & CD \\
C & \to & e \mid \epsilon \\
D & \to & f \mid \epsilon
\end{array}
$$





# $follow(A)$
## TRAINING

$$
\begin{array}{lcl}
S & \to & aABb \\
A & \to & Ac \mid d \\
B & \to & CD \\
C & \to & e \mid \epsilon \\
D & \to & f \mid \epsilon
\end{array}
$$

**foreach** $B \to \alpha A \beta$ **do**
  **if** $\beta \neq \epsilon$ **then**
    add $\mathrm{first}(\beta) \setminus \{\epsilon\}$ to $\mathrm{follow}(A)$
  **end**
  **if** $\beta = \epsilon$ *or* $\epsilon \in \mathrm{first}(\beta)$ **then**
    add $\mathrm{follow}(B)$ to $\mathrm{follow}(A)$
  **end**
**end**

| | first | computation of follow | follow |
|---|---|---|---|
| S | a | $ | $ |
| A | d | (production 1) **beta=Bb**, add e f b<br>(production 2) **beta=c**, add c | b c e f |
| B | e,f,epsilon | (production 1) **beta=b**, add b | b |
| C | e, epsilon | (production 4) **beta=D**, add f | b f |
| D | f, epsilon | (production 4) **beta=epsilon** | b |

## *follow*$(A)$

**TRAINING**

$$
\begin{aligned}
S &\rightarrow aA \mid bBc \\
A &\rightarrow Bd \mid Cc \\
B &\rightarrow e \mid \epsilon \\
C &\rightarrow f \mid \epsilon
\end{aligned}
$$







## *follow*$(A)$

**TRAINING**



$$
\begin{aligned}
S &\rightarrow aA \mid bBc \\
A &\rightarrow Bd \mid Cc \\
B &\rightarrow e \mid \epsilon \\
C &\rightarrow f \mid \epsilon
\end{aligned}
$$

**foreach** $B \rightarrow \alpha A \beta$ **do**
  **if** $\beta \neq \epsilon$ **then**
    add $\mathrm{first}(\beta) \setminus \{\epsilon\}$ to $\mathrm{follow}(A)$
  **end**
  **if** $\beta = \epsilon$ *or* $\epsilon \in \mathrm{first}(\beta)$ **then**
    add $\mathrm{follow}(B)$ to $\mathrm{follow}(A)$
  **end**
**end**

| | first | computation of follow | follow |
|---|---|---|---|
| S | a, b | $ | $ |
| A | e,d, f,c | (production 1) **beta=epsilon** | $ |
| B | e, epsilon | (production 2) **beta=c,** add **c**<br>(production 3) **beta=d**, add **d** | c d |
| C | f, epsilon | (production 4) **beta=c**, add **c** | c |

# Construction of predictive parsing tables

**ALGORITHM**

**input** : Grammar $\mathcal{G} = (V, T, S, \mathcal{P})$
**output** : Predictive parsing table $M$
**foreach** $A \rightarrow \alpha \in \mathcal{P}$ **do**
  add $A \rightarrow \alpha$ to $M[A, b]$ for each $b \in \mathrm{first}(\alpha)$
  **if** $\epsilon \in \mathrm{first}(\alpha)$ **then**
    add $A \rightarrow \alpha$ to $M[A, x]$ for each $x \in \mathrm{follow}(A)$

set to *error*() all the empty entries;

Recall: by notational convention $b$ is a terminal symbol

Observe: $\mathrm{follow}(A)$ can contain \$, hence we use $x$ (rather than $b$) to range over it





# LL(1) grammars

If no entry of the predictive parsing table for $\mathcal{G}$ is multiply-defined then $\mathcal{G}$ is an **LL(1) grammar**

## LL(1) grammars

$$\begin{array}{lcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid id \end{array}$$

Is it LL(1)?

$\mathrm{first}(E) = \mathrm{first}(T) = \mathrm{first}(F) = \{(, id\}$

Hence, e.g.,

- $M[E, id]$ contains both $E \to E + T$ and $E \to T$

What if we always expand $E$ into $E + T$ when we see the input $id$?

$E \Rightarrow E + T \Rightarrow E + T + T \Rightarrow E + T + T + T \ldots$





## Left recursion

A grammar is **left recursive** if, for some $A$ and some $\alpha$, $A \Rightarrow^* A\alpha$

Example:

$$\begin{array}{lcl} S & \to & B \mid a \\ B & \to & Sa \mid b \end{array}$$

Is left recursive by $S \Rightarrow B \Rightarrow Sa$

## Left recursion

A grammar is **immediately left recursive** if it has a production of the form $A \rightarrow A\alpha$

Example:

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

Is immediately left recursive by either $E \rightarrow E + T$ or $T \rightarrow T * F$





## Left recursion

**LEMMA**
If $\mathcal{G}$ is left recursive then $\mathcal{G}$ is not LL(1)

# Elimination of immediate left recursion
## GOAL

Transform

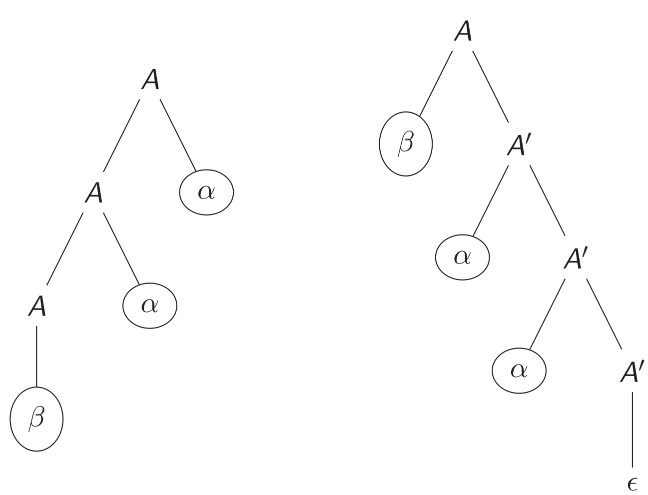
$$A \to A\alpha \mid \beta \text{ where } \alpha \neq \epsilon \text{ and } \beta \neq A\gamma$$

To get a non left recursive grammar which generates the same language





# Elimination of immediate left recursion
## INTUITION



# Elimination of immediate left recursion
## STRATEGY

Substitute

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

Where $\alpha \neq \epsilon$ and $\beta \neq A\gamma$

With

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

Where $A'$ is a fresh non-terminal





# Elimination of immediate left recursion
## STRATEGY FOR THE MOST GENERAL CASE

Substitute

$$A \to A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_k$$

Where $\alpha_j \neq \epsilon$ for every $j = 1 \ldots n$ and $\beta_i \neq A\gamma_i$ for every $i = 1 \ldots k$

With

$$\begin{array}{lcl} A & \to & \beta_1 A' \mid \ldots \mid \beta_k A' \\ A' & \to & \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \epsilon \end{array}$$

Where $A'$ is a fresh non-terminal

# Elimination of left recursion
## INTUITION

- Transform the grammar so to decrease the number of steps of the derivation $A \Rightarrow^* A\alpha$
- Eliminate immediate left recursion





# Elimination of left recursion
## EXAMPLE

Take

$$\begin{array}{lcl} A & \rightarrow & Ba \mid b \\ B & \rightarrow & Bc \mid Ad \mid b \end{array}$$

Left recursion shows in two derivation steps:

$$A \Rightarrow Ba \Rightarrow Ada$$

To decrease the number of steps of the derivation, replace the production $B \rightarrow Ad$ by

$$B \quad \rightarrow \quad Bad \mid bd$$

# Elimination of left recursion
## EXAMPLE

Get

$$\begin{array}{lcl} A & \rightarrow & Ba \mid b \\ B & \rightarrow & Bc \mid Bad \mid bd \mid b \end{array}$$

Eliminate immediate left recursion from $B$ and get the grammar

$$\begin{array}{lcl} A & \rightarrow & Ba \mid b \\ B & \rightarrow & bdB' \mid bB' \\ B' & \rightarrow & cB' \mid adB' \mid \epsilon \end{array}$$





# Elimination of left recursion
## EFFECTIVENESS

Eliminate left recursion from the grammar

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

Get the LL(1) grammar

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid \epsilon \\ F & \rightarrow & (E) \mid id \end{array}$$

# Elimination of immediate left recursion
## EFFECTIVENESS

Eliminate left recursion from the **ambiguous** grammar

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

Get the grammar

$$\begin{array}{lcl} E & \rightarrow & (E)E' \mid idE' \\ E' & \rightarrow & +EE' \mid *EE' \mid \epsilon \end{array}$$





# Elimination of immediate left recursion
## EFFECTIVENESS

$$\begin{array}{lcl} E & \rightarrow & (E)E' \mid idE' \\ E' & \rightarrow & +EE' \mid *EE' \mid \epsilon \end{array}$$

Is it LL(1)?

Not really: the entry $[E', +]$ of the parsing table contains two productions:

- $E' \rightarrow +EE'$
- $E' \rightarrow \epsilon$

In fact:

| | elements in first | elements in follow |
|---|---|---|
| $E$ | ( $id$ | \$ + $*$ ) |
| $E'$ | + $*$ $\epsilon$ | \$ + $*$ ) |

**Lesson learnt:** Elimination of left recursion does not guarantee to get an LL(1) grammar

# Elimination of immediate left recursion

**EFFECTIVENESS**

$$
\begin{array}{lcl}
E & \to & (E)E' \mid idE' \\
E' & \to & +EE' \mid *EE' \mid \epsilon
\end{array}
$$

Is it ambiguous or not?



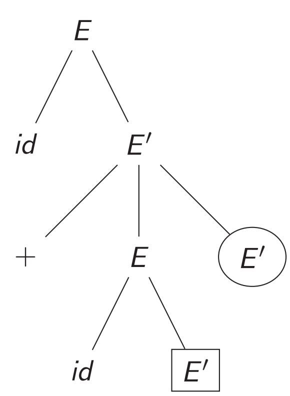


# Elimination of immediate left recursion

**EFFECTIVENESS**

$$
\begin{array}{lcl}
E & \to & (E)E' \mid idE' \\
E' & \to & +EE' \mid *EE' \mid \epsilon
\end{array}
$$

Take the partial derivation tree

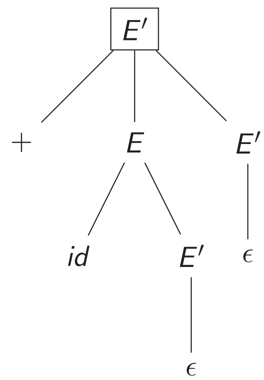


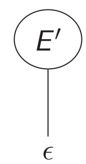
# Elimination of immediate left recursion

## EFFECTIVENESS

$$
\begin{array}{lcl}
E & \rightarrow & (E)E' \mid idE' \\
E' & \rightarrow & +EE' \mid *EE' \mid \epsilon
\end{array}
$$

Get a derivation of $id + id + id$ by completing the tree with this instantiation of the missing sub-trees





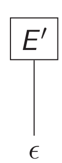


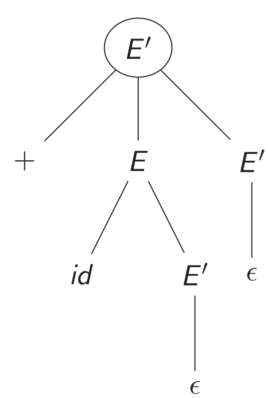


# Elimination of immediate left recursion

## EFFECTIVENESS

Get a different derivation of $id + id + id$ by completing the tree with this instantiation





**Lesson learnt:** Elimination of left recursion does not eliminate ambiguity

## Left factoring

Take the grammar

$$S \rightarrow aSb \mid ab$$

Is it LL(1)?

Not really: the entry $[S, a]$ of the parsing table contains two productions:

- $S \rightarrow aSb$
- $S \rightarrow ab$





## Left factoring

A grammar can be **left factorized** (can undergo left factorization) if multiple productions for the same non-terminal have the same prefix

**LEMMA**
If $\mathcal{G}$ can be left factorized then $\mathcal{G}$ is not LL(1)

# Left factoring
## STRATEGY

Delay as much as possible the choice between productions with the same prefix

Substitute

$$A \to \alpha\beta_1 \mid \alpha\beta_2$$

With

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

Where $A'$ is a fresh non-terminal





# Left factoring
## ALGORITHM

**input** : Grammar $\mathcal{G}$ that can be left factorized
**output** : Left factorized version of $\mathcal{G}$
**repeat**
  **foreach** $A$ **do**
    find the longest prefix $\alpha$ common to two or more productions for $A$ ;
    **if** $\alpha \neq \epsilon$ **then**
      choose a fresh non-terminal $A'$ and replace
      $A \to \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_k$ by
      $A \to \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_k$
      $A' \to \beta_1 \mid \ldots \mid \beta_n$
**until** *no pair of productions for any $A$ has common prefix*;

# Left factoring
## EFFECTIVENESS

Apply left factorization to

$$S \rightarrow aSb \mid ab$$

Get the grammar

$$\begin{array}{lcl} S & \rightarrow & aS' \\ S' & \rightarrow & Sb \mid b \end{array}$$

Is it LL(1)?



Formal Languages and Compilers Paola Quaglia, 2022


# Left factoring
## EFFECTIVENESS


$$\begin{array}{lcl} S & \rightarrow & aS' \\ S' & \rightarrow & Sb \mid b \end{array}$$

| | elements in first | elements in follow |
|---|---|---|
| $S$ | $a$ | \$ $b$ |
| $S'$ | $a$ $b$ | \$ $b$ |

Top-down parsing table

| | $a$ | $b$ | \$ |
|---|---|---|---|
| $S$ | $S \rightarrow aS'$ | | |
| $S'$ | $S' \rightarrow Sb$ | $S' \rightarrow b$ | |

# Left factoring

## EFFECTIVENESS: DANGLING ELSE

Apply left factorization to the **ambiguous** grammar

$$S \rightarrow \texttt{if } b \texttt{ then } S \mid \texttt{if } b \texttt{ then } S \texttt{ else } S \mid c$$

Get the grammar

$$\begin{aligned} S &\rightarrow \texttt{if } b \texttt{ then } S\,S' \mid c \\ S' &\rightarrow \texttt{else } S \mid \epsilon \end{aligned}$$





# Left factoring

## EFFECTIVENESS

$$\begin{aligned} S &\rightarrow \texttt{if } b \texttt{ then } S\,S' \mid c \\ S' &\rightarrow \texttt{else } S \mid \epsilon \end{aligned}$$

Is it LL(1)?

Not really: the entry $[S', \texttt{else}]$ of the parsing table contains two productions:

- $S' \rightarrow \texttt{else } S$
- $S' \rightarrow \epsilon$

In fact:

| | elements in first | elements in follow |
|---|---|---|
| $S$ | if $c$ | \$ else |
| $S'$ | else $\epsilon$ | \$ else |

**Lesson learnt:** Left factorization does not guarantee to get an LL(1) grammar

# Left factoring
## EFFECTIVENESS

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{if}\ b\ \texttt{then}\ S\,S' \mid c \\
S' & \rightarrow & \texttt{else}\ S \mid \epsilon
\end{array}
$$

Is it ambiguous or not?

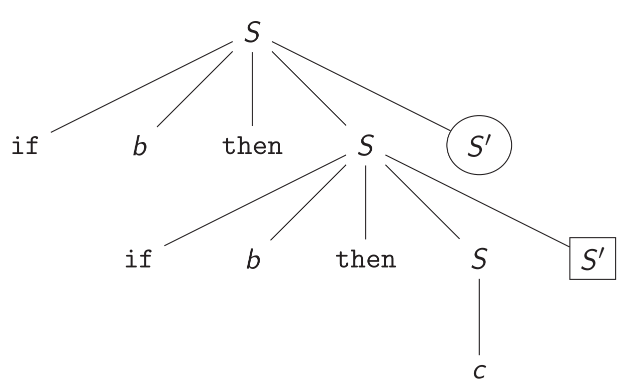




# Left factoring
## EFFECTIVENESS

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{if}\ b\ \texttt{then}\ S\,S' \mid c \\
S' & \rightarrow & \texttt{else}\ S \mid \epsilon
\end{array}
$$

Take the partial derivation tree



# Left factoring
## EFFECTIVENESS

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{if}\ b\ \texttt{then}\ S\,S' \mid c \\
S' & \rightarrow & \texttt{else}\ S \mid \epsilon
\end{array}
$$

Get a derivation of "`if` $b$ `then` `if` $b$ `then` $c$ `else` $c$" by completing the tree with this instantiation of the missing sub-trees






# Left factoring
## EFFECTIVENESS

Get a different derivation of "`if` $b$ `then` `if` $b$ `then` $c$ `else` $c$" by completing the tree with this instantiation


**Lesson learnt:** Left factorization does not eliminate ambiguity

# Dangling else

## INNERMOST BINDING

A common strategy to avoid the ambiguity due to the dangling else is to impose the so-called **innermost binding**: every `else` must match the closest unmatched `then`

Innermost binding can be enforced by defining a grammar that allows only matched `then-else` pairs between occurrences of `then` and `else`

$$\begin{array}{lcl} S & \rightarrow & M \mid U \\ M & \rightarrow & \texttt{if}\ b\ \texttt{then}\ M\ \texttt{else}\ M \mid c \\ U & \rightarrow & \texttt{if}\ b\ \texttt{then}\ S \mid \texttt{if}\ b\ \texttt{then}\ M\ \texttt{else}\ U \end{array}$$





# Summary

- No left-recursive grammar is LL(1)
- No grammar that can be left-factorized is LL(1)
- No ambiguous grammar is LL(1)

# LL(1) grammars

**LEMMA**
$\mathcal{G}$ is LL(1) iff if $\mathcal{G}$ has productions $A \to \alpha \mid \beta$ then

- $\mathrm{first}(\alpha) \cap \mathrm{first}(\beta) = \emptyset$
- If $\epsilon \in \mathrm{first}(\alpha)$ then $\mathrm{first}(\beta) \cap \mathrm{follow}(A) = \emptyset$ and, v.v., if $\epsilon \in \mathrm{first}(\beta)$ then $\mathrm{first}(\alpha) \cap \mathrm{follow}(A) = \emptyset$