ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

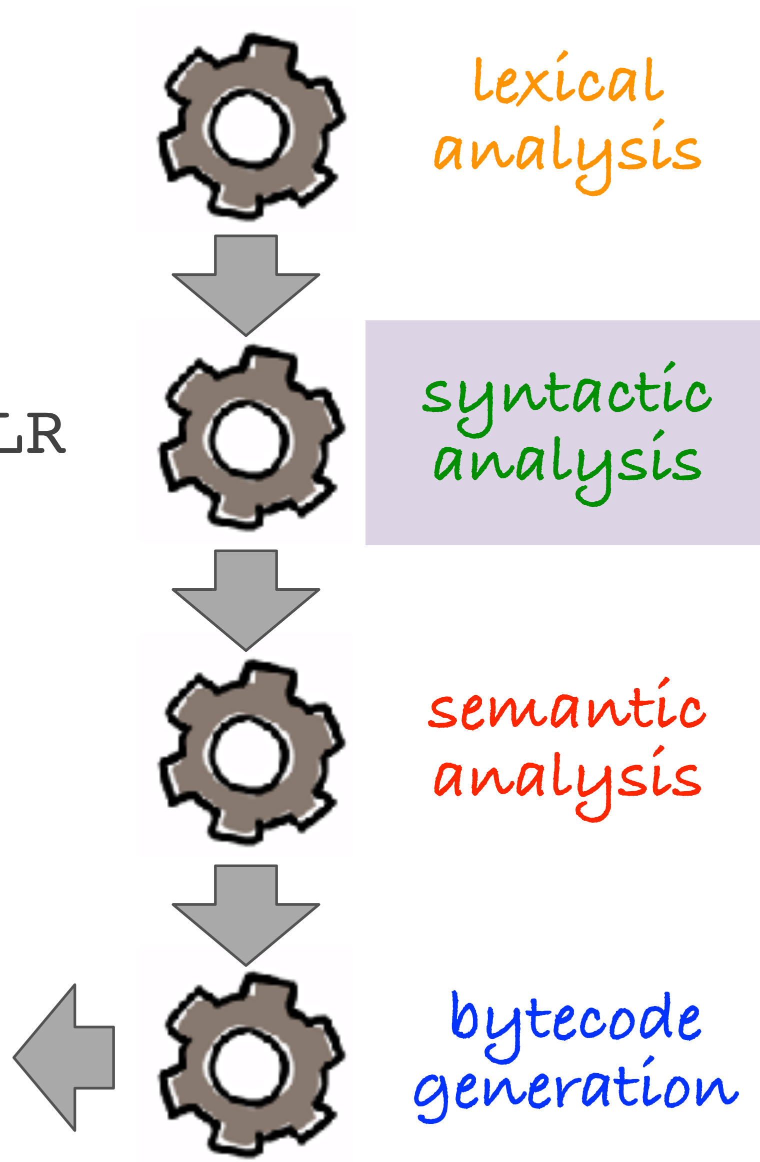
RECURSIVELY DESCENT PARSING AND LL PARSING

COSIMO LANEVE
cosimo.laneve@unibo.it

CORSO 72671 - COMPLEMENTI DI LINGUAGGI DI PROGRAMMAZIONE

THIS LECTURE

LL-parsing and **ANTLR**



the SimpLan interpreter

OUTLINE

* recursive descent parsing
  * problems
  * implementations
* predictive parsers
* parsers `LL(1)`
  * **`FIRST` and `FOLLOW` sets**
  * `LL(1)` tables
  * ambiguities
* `ANTLR`

reference: Torben Morgensen: Basics of Compiler Design, Chapter 3, sections 6—13

RECURSIVE DESCENT PARSING

analyze the sequence of tokens trying to reconstruct the steps of a **leftmost derivation**

these parsers are called **top-down** because they mimic an anticipated visit of the syntax tree — anticipated = from the root to the leaves

idea: the rules for a non-terminal `A` define a method that recognises `A`

- the right-hand sides of the rules define the structure of the method code
- the sequence of terminals and non-terminals in the rules corresponds to a check that terminals match and to invocations of the methods corresponding to the non-terminal symbols
- the presence of different rules for `A` is implemented by a `case` or a `if`

RECURSIVE DESCENT PARSING — EXAMPLE

take the grammar

$$E \rightarrow T + E \mid T$$

$$T \rightarrow (E) \mid (E) * T \mid int \mid int * T$$

the tokens returned by the lexer are

`lpar` `rpar` `plus` `times` `int(k)` [k ∈ Nat]

assume to analyze the token stream:

`int(5) times int(2)`

the parsing starts with the expansion of the initial symbol `E` and **every rule** of `E` is checked, one at a time . . .

RECURSIVE DESCENT PARSING — EXAMPLE

try `E → T + E`

`E → T + E | T`
`T → (E) | (E) * T | int | int * T`

`int(5) times int(2)`

- then you check the **first rule** for `T`: `T → (E)`
  - but there is no match with the input token `int(5)`
- then you check the **second rule** for `T`: `T → (E) * T`
  - but there is no match with the input token `int(5)`
- then you check the **third rule** for `T`: `T → int`
  - there is match with the input token `int(5)`
  - but there is no match with the token `plus` after `T` and the token `times` of the input stream
- then you check the **forth rule** for `T`: `T → int * T`
  - there is match with `int(5)` and then `times` and `int(2)`
  - but there is no match with the token `plus` because the input stream ends
- **we have saturated the choices for `T` without succeeding**
  - backtrack to the other choices for `E`

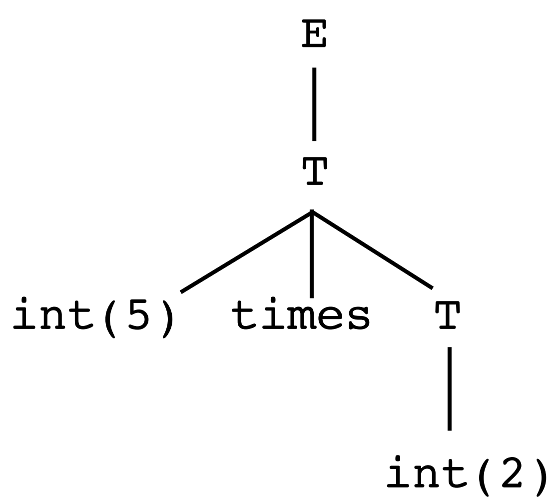
RECURSIVE DESCENT PARSING — EXAMPLE

```
E → T + E | T
T → (E) | (E) * T | int | int * T
```

`int(5) times int(2)`

then try **E → T** and perform the same steps done for **E → T + E**

- the parsing succeeds with the rule **T → int*T** and **T → int**
- the returned parse tree is the following one



RECURSIVE DESCENT PARSING — IMPLEMENTATION

define a method **boolean** that verifies the matches of the token stream

- verify the match with a given **terminal**

```
public boolean term(TOKEN tok){
        TOKEN x = in[next] ;
        next = next + 1 ;
        return x == tok;
}
```

- verify the match with a rule of S (say the n-th)

```
public boolean S_n(){ ... }
```

- verify the match with a whatever rule of S:

```
public boolean S(){ ... }
```

note: every foregoing method increments **next**

RECURSIVE DESCENT PARSING — IMPLEMENTATION

```
E → T + E | T
T → (E) | (E) * T | int | int * T
```

for the rule `E → T + E`

```
public boolean E_1( ){
    return (T() && term(plus) && E()) ;
}
```

this corresponds to

```
boolean B1 = T();
boolean B2 = B1 && term(plus);
return(B2 && E()) ;
```

for the rule `E → T`

```
public boolean E_2( ){
    return T();
}
```

for (all) the rules of `E` (with backtracking)

```
public boolean E() {
    int saved = next ;
    if (E_1()) return true ;
        else { next = saved ; return (E_2()) ; }
}
```

backtrack!

RECURSIVE DESCENT PARSING — IMPLEMENTATION

```
E → T + E | T
T → (E) | (E) * T | int | int * T
```

methods for the non-terminal `T`

```
public boolean T_1(){
        return ( term(lpar) && E() && term(rpar) );
}

public boolean T_2(){
    return ( term(lpar) && E() && term(rpar) && term(times) && T() );
}

public boolean T_3(){ return ( term(int) ); }

public boolean T_4(){
        return ( term(int) && term(times) && T() ); }
public boolean T(){
        int saved = next;
        if (T_1()) return true ;
        else { next = saved ;
               if (T_2()) return true ;
               else { next = saved ;
                      if (T_3()) return true ;
                      else { next = saved ; return T_4() ; }
               }
        }
}
```

RECURSIVE DESCENT PARSING — REMARKS

to trigger the parsing

* initialize `next` in such a way it points to the first token
* invoke `E( )`
* assume that a special character `$` represents the end of the input stream in the array `in[ ]`
* the parsing **ends with success** if, at the end of the execution, `in[next] == $`
* **remark**: the execution of the recursive descent parsing coincides with the abstract execution computed at the beginning
* **other remark**: this is simple to implement (also by hand) **but it does not work, sometimes!**

RECURSIVE DESCENT PARSING — LEFT-RECURSIVE GRAMMARS

take the rule $\texttt{S} \rightarrow \texttt{S a}$

and try to analyze this rule in the recursive descent parsing

- **why the process does not work?**

Definition: left recursive grammar

A grammar (**N**, **T**, $\rightarrow$, $\texttt{S}$) is **left-recursive** if there is $\texttt{A} \in \mathbf{N}$ such that

$$\texttt{A} \rightarrow^{+} \texttt{A}\,\gamma \text{, for some } \gamma$$

the recursive descent parsing **does not work** for lr-grammars

- because it performs **an infinite cycle**

note: in these cases you need to **change the grammar** **by removing the left-recursion** (see following slides)

RECURSIVE DESCENT PARSING — SUMMING UP

the parsing strategy is **extremely simple**

- in case you need to **remove the left-recursion** … but this task can be performed **automatically**

it is **not common** because it uses the backtracking

- it is very inefficient
- in practice, the backtracking may be reduced or eliminated by changing the grammar (**left-factorization**)

it is **good for small grammars**

- you need to be careful: the order of productions is important even after left-recursion is eliminated
- try to reverse the order of `T → int*T` and `T → int`
- what goes wrong? (consider input `int*int`)

PREDICTIVE PARSERS — MOTIVATIONS

to avoid the backtracking, it would be useful

- if the recursive-descent parser **knows the next production to expand**
- **idea**: replace the code

```
saved = next ;
if (E_1()) return true;
else {  next = saved; return E_2(); }
```

with

```
switch ( something )  {
           case L1: return E_1();
           case L2: return E_2();
           default: System.out.print("syntax error") ;
}
```

- what is the meaning of "`something`", `L1`, `L2` ?

  they are defined by a **lookahead** (analysis of the next tokens)

PREDICTIVE PARSING AND LEFT FACTORING

the grammar

```
E → T + E | T
T → (E) | (E)*T | int | int * T
```

is impossible to predict because

- the non-terminal `T` has **two productions** that begin with "`(`" and **two productions** that begin with "`int`"
- the non-terminal `E` has the two productions that begin with `T` and **it is not evident how to predict**

this grammar **must be left-factorized** before using predictive parsers (see following slides)

```
E → T X                X → + E | ε
T → (E) Y | int Y      Y → * T | ε
```

PREDICTIVE PARSERS

they are similar to recursive-descent parsers except that they can **predict** which production to use

- by looking at the next tokens
- without backtracking

predictive parsers accept `LL(k)` grammars

- `L` means "left-to-right" input scan
- `L` means "leftmost derivation"
- `k` means "predict using k tokens of lookahead"

we study `LL(1)` analysis

- **ANTLR** uses `LL(*)`, a sophisticated technique that consider as many token as needed (this technique is not covered in this course)

`LL(1)` LANGUAGES

in recursive-descent parsers, for each non-terminal and input token there may be several possible productions

> `LL(1)` means: for each non-terminal and input token there may be **at most one production** that can be used

`LL(1)` parsers can be defined by a **2 dimension table**

* one dimension for the non-terminal to expand
* one dimension for the next token
* the table entry contains the production to use

PARSER `LL(1)`

in practice, instead of using the code

```
switch ( something )  {
            case L1: return E_1();
            case L2: return E_2();
            default: System.out.print("syntax error") ;
}
```

use a table `LL(1)` and a **parsing stack**

- the `LL(1)` table will replace the switch instruction
- the parsing stack will replace the call stack

PARSER `LL(1)` — PARSING TABLE/EXAMPLE

the `LL(1)` parsing table of

```
E → T X                  X → + E | ε
T → (E) Y | int Y        Y → * T | ε
```

	int	*	+	(	)	$
T	T → int Y			T → (E)Y		
E	E → T X			E → T X		
X			X → + E		X → ε	X → ε
Y		Y → * T	Y → ε		Y → ε	Y → ε

- **for the `[E,int]` entry**: when the non-terminal on the stack is `E` and the next token in input is `int`, use the production `E → T X`
- **for the `[Y,+]` entry**: when the non-terminal on the stack is `Y` and the next token in input is + then remove `Y` (we'll see why)
- the empty entries indicate an **error**: example `[E,*]`

PARSER `LL(1)` — THE PARSING TABLE

the technique is **similar to recursive descent**, but instead of nondeterminism (and the backtrack)

* for every non-terminal `S`, look at the next token, say **`a`**, and the entry `[S,a]` in the table

we use a **stack** in order to record the terminals and non-terminals in the rhs of the production in `[S,a]`

* the input is **rejected** when an erroneous state is found (empty entry in the parsing table)
* the input is **accepted** when the entry contains **end-of-input** token

PSEUDO-ALGORITHM OF `LL(1)` PARSING

```
add $ at the end of the array TOKENS ;
next = 0 ;
stack =⟨S $⟩ ;
repeat
   switch (stack){
     case ⟨X rest⟩: if (LL1_TABLE[X,TOKENS[next]] = α1 ... αn)
                            stack = ⟨α1 ... αn rest⟩;
                    else  System.out.println("error") ;
                    break ;
          ⟨t rest⟩: if (t == TOKENS[next]) {
                            stack = ⟨rest⟩ ;
                            next = next+1 ;
                     } else  System.out.println("error") ;
                     break ;
    }
until (stack == ⟨ ⟩)
```

LL(1) PARSING: EXAMPLE

	int	*	+	(	)	$
T	T → int Y			T → (E)Y		
E	E → T X			E → T X		
X			X → + E		X → ε	X → ε
Y		Y → * T	Y → ε		Y → ε	Y → ε

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε
$	$	terminal/ACCEPT

THE DEFINITION OF THE `LL(1)` PARSING TABLE

let $\mathcal{G} = (\mathbf{N}, \mathtt{T}, \rightarrow, S)$, its `LL(1)` table is defined as follows:

1. it has non-terminal symbols in the rows and terminal symbols in the columns
2. for every rule $\mathtt{X} \rightarrow \gamma$ in $\mathcal{G}$ and for every $\mathtt{t}$ such that $\gamma \rightarrow^* \mathtt{t}\,\delta$, add the rule $\mathtt{X} \rightarrow \gamma$ in the entry $(\mathtt{X}, \mathtt{t})$
3. for every rule $\mathtt{X} \rightarrow \gamma$ in $\mathcal{G}$ such that $\gamma \rightarrow^* \varepsilon$ add the rule $\mathtt{X} \rightarrow \gamma$ in the entry $(\mathtt{X}, \mathtt{t})$, for every $\mathtt{t}$ such that $S \rightarrow^* \delta\, \mathtt{X}\, \mathtt{t}\, \delta'$

they seem difficult to compute!

the `LL(1)` grammars are those with `LL(1)` parsing tables that **do not have multiple entries**

THE DEFINITION OF THE LL(1) PARSING TABLE — NULLABLE

Definition: the function NULLABLE

Let $\mathcal{G} = (\mathbf{N}, \mathbf{T}, \rightarrow, S)$ be a context-free grammar. NULLABLE is a function on $\mathcal{G}$ defined as follows

$$\texttt{NULLABLE}(\mathcal{G}) = \{ \texttt{A} \mid \texttt{A} \rightarrow^{*} \varepsilon \}$$

remark: by definition $\texttt{NULLABLE}(\mathcal{G}) \subseteq \mathbf{N}$

example:

```
E → T X                 X → + E | ε
T → (E) Y | Y           Y → * T | ε
```

then $\texttt{NULLABLE}(\mathcal{G}) = \{ \texttt{X}, \texttt{Y}, \texttt{T}, \texttt{E} \}$. Are you sure about E ?

remark: this definition is not algorithmic

THE DEFINITION OF THE LL(1) PARSING TABLE — NULLABLE

Algorithmic definition: the function NULLABLE

Let $\mathcal{G} = (\mathbf{N}, \mathbf{T}, \rightarrow, \mathbf{S})$ be a context-free grammar. $\mathtt{NULLABLE}_i$ are functions on $\mathcal{G}$ defined as follows

1. $\mathtt{NULLABLE}_0(\mathcal{G}) = \{ A \mid A \rightarrow \varepsilon \text{ in } \mathcal{G} \}$
2. $\mathtt{NULLABLE}_{i+1}(\mathcal{G}) = \mathtt{NULLABLE}_i(\mathcal{G})$

$$\cup \ \{ A \mid A \rightarrow A_1 \dots A_n \text{ in } \mathcal{G} \ \wedge \ A_1, \dots, A_n \in \mathtt{NULLABLE}_i(\mathcal{G}) \}$$

---

* it is easy to show that $\mathtt{NULLABLE}_i(\mathcal{G}) \subseteq \mathtt{NULLABLE}_{i+1}(\mathcal{G}) \subseteq \mathbf{N}$
* therefore there is $\mathtt{k}$ such that $\mathtt{NULLABLE}_k(\mathcal{G}) = \mathtt{NULLABLE}_{k+1}(\mathcal{G})$

  then $\mathtt{NULLABLE}(\mathcal{G}) = \mathtt{NULLABLE}_k(\mathcal{G})$

FUNCTION `NULLABLE`: EXAMPLES

grammar

$E \to T\ X$ $\qquad X \to +\ E \mid \varepsilon$

$T \to (E)Y \mid int\ Y$ $\qquad Y \to *\ T \mid \varepsilon$

predicate **NULLABLE**

$\text{NULLABLE}_0(\mathcal{G}) = \{X, Y\}$

$\text{NULLABLE}_1(\mathcal{G}) = \{X, Y\}$

$$\text{NULLABLE}(\mathcal{G}) = \text{NULLABLE}_0(\mathcal{G}) = \{X, Y\}$$

FUNCTION NULLABLE: EXAMPLES

grammar

$$
\begin{aligned}
\mathtt{Z} &\rightarrow \mathtt{b} \mid \mathtt{X\ Y\ Z} \\
\mathtt{X} &\rightarrow \mathtt{Y} \mid \mathtt{a} \\
\mathtt{Y} &\rightarrow \varepsilon \mid \mathtt{c}
\end{aligned}
$$

predicate NULLABLE

$$
\begin{aligned}
\mathtt{NULLABLE}_0(\mathcal{G}) &= \{\, \mathtt{Y} \,\} \\
\mathtt{NULLABLE}_1(\mathcal{G}) &= \{\, \mathtt{X},\ \mathtt{Y} \,\} \\
\mathtt{NULLABLE}_2(\mathcal{G}) &= \{\, \mathtt{X},\ \mathtt{Y} \,\}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{NULLABLE}(\mathcal{G}) &= \mathtt{NULLABLE}_1(\mathcal{G}) \\
&= \{\, \mathtt{X},\ \mathtt{Y} \,\}
\end{aligned}
$$

grammar

$$
\begin{aligned}
\mathtt{S} &\rightarrow \mathtt{a} \mid \mathtt{X} \\
\mathtt{X} &\rightarrow \mathtt{Y} \\
\mathtt{Y} &\rightarrow \mathtt{X}
\end{aligned}
$$

predicate NULLABLE

$$
\begin{aligned}
\mathtt{NULLABLE}_0(\mathcal{G}) &= \varnothing \\
\mathtt{NULLABLE}_1(\mathcal{G}) &= \varnothing
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{NULLABLE}(\mathcal{G}) &= \mathtt{NULLABLE}_0(\mathcal{G}) \\
&= \varnothing
\end{aligned}
$$

DEFINITION OF `LL(1)` PARSING TABLES: `FIRST`

Algorithmic definition: the function `FIRST`

Let $\mathcal{G} = (\mathbf{N}, \mathbf{T}, \rightarrow, S)$ be a context-free grammar. $\mathtt{FIRST}_i$ are functions on $\mathbf{N} \cup \mathbf{T}$ that are defined as follows

1. $\mathtt{FIRST}_i(\mathtt{t}) = \{ \mathtt{t} \}$, with $\mathtt{t} \in \mathbf{T}$ `// for every i`

2. $$\mathtt{FIRST}_0(A) = \begin{cases} \{ \varepsilon \} & \text{if } A \in \mathtt{NULLABLE}(\mathcal{G}) \\ \emptyset & \text{if } A \notin \mathtt{NULLABLE}(\mathcal{G}) \ \wedge \ A \in \mathbf{N} \end{cases}$$

3. $$\mathtt{FIRST}_{i+1}(A) = \mathtt{FIRST}_i(A) \cup \bigcup_{\substack{A \rightarrow \alpha_1 \cdots \alpha_n \text{ in } \mathcal{G} \\ \forall\, \mathtt{i} \in \mathtt{1..k-1} \,:\, \alpha_i \in \mathtt{NULLABLE}(\mathcal{G})}} \mathtt{FIRST}(\alpha_k) \setminus \{ \varepsilon \}$$

- it is easy to show that, for every `i`: $\mathtt{FIRST}_i(A) \subseteq \mathtt{FIRST}_{i+1}(A) \subseteq \mathbf{T} \cup \{ \varepsilon \}$
- therefore there is `k` such that, for every A, $\mathtt{FIRST}_k(A) = \mathtt{FIRST}_{k+1}(A)$

then $\mathtt{FIRST}(A) = \mathtt{FIRST}_k(A)$

SETS FIRST: EXAMPLES

grammar

$E \rightarrow T\ X \qquad X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow (E)\ Y \mid \texttt{int}\ Y \qquad Y \rightarrow *\ T \mid \varepsilon$

sets $FIRST_i$ // we only compute FIRST for nonterminals

$FIRST_0(X) = \{ \varepsilon \}$ $FIRST_0(Y) = \{ \varepsilon \}$ $FIRST_0(E) = \varnothing$ $FIRST_0(T) = \varnothing$

$FIRST_1(X) = \{ +, \varepsilon \}$ $FIRST_1(Y) = \{ *, \varepsilon \}$ $FIRST_1(E) = \varnothing$ $FIRST_1(T) = \{ (, \texttt{int} \}$

$FIRST_2(X) = \{ +, \varepsilon \}$ $FIRST_2(Y) = \{ *, \varepsilon \}$ $FIRST_2(E) = \{ (, \texttt{int} \}$ $FIRST_2(T) = \{ (, \texttt{int} \}$

$FIRST(X) = \{ +, \varepsilon \}$ $FIRST(Y) = \{ *, \varepsilon \}$

$FIRST(E) = \{ (, \texttt{int} \}$ $FIRST(T) = \{ (, \texttt{int} \}$

SETS FIRST: EXAMPLES

grammar

$$S \rightarrow (S)\ S \ \mid\ \varepsilon$$

sets $FIRST_i$

$$FIRST_0(S) = \{ \varepsilon \}$$

$$FIRST_1(S) = \{ (, \varepsilon \}$$

$$FIRST(S) = FIRST_1(S) = \{ (, \varepsilon \}$$

SETS FIRST: EXAMPLES

grammar

$$Z \rightarrow b \mid X\ Y\ Z$$
$$X \rightarrow Y \mid a$$
$$Y \rightarrow \varepsilon \mid c$$

sets $FIRST_i$:

$FIRST_0(Z) = \varnothing$ $\quad FIRST_0(X) = \{ \varepsilon \}$ $\quad FIRST_0(Y) = \{ \varepsilon \}$

$FIRST_1(Z) = \{ b \}$ $\quad FIRST_1(X) = \{ a, \varepsilon \}$ $\quad FIRST_1(Y) = \{ c, \varepsilon \}$

$FIRST_2(Z) = \{ a, b, c \}$ $\quad FIRST_2(X) = \{ a, c, \varepsilon \}$ $\quad FIRST_2(Y) = \{ c, \varepsilon \}$

$FIRST(X) = \{ a, c, \varepsilon \}$ $\quad FIRST(Y) = \{ c, \varepsilon \}$

$FIRST(Z) = \{ a, b, c \}$

SETS FIRST: EXAMPLES

grammar

$$S \to X \mid XS$$
$$X \to x \mid \varepsilon$$

sets $FIRST_i$:

$FIRST_0(S) = \{ \varepsilon \}$ $FIRST_0(X) = \{ \varepsilon \}$

$FIRST_0(S) = \{ \varepsilon \}$ $FIRST_0(X) = \{ \varepsilon \}$

$FIRST(S) = \{ \varepsilon \}$ $FIRST(X) = \{ \varepsilon \}$

ESTENSIONE DI $\mathtt{FIRST}$ A SEQUENZE IN $\mathbf{N} \cup \mathbf{T}$

it is easy to compute $\mathtt{FIRST}(\gamma)$ where $\gamma \in (\mathbf{N} \cup \mathbf{T})^*$

$\mathtt{FIRST}(\varepsilon) = \{ \varepsilon \}$

$\mathtt{FIRST}(\mathtt{t}\gamma) = \{ \mathtt{t} \}$, with $\mathtt{t} \in \mathbf{T}$

$\mathtt{FIRST}(\mathsf{A}\gamma) = \mathtt{FIRST}(\mathsf{A})$, with $\mathsf{A} \notin \mathtt{NULLABLE}(\mathcal{G})$

$\mathtt{FIRST}(\mathsf{A}\gamma) = \mathtt{FIRST}(\mathsf{A}) \setminus \{ \varepsilon \} \cup \mathtt{FIRST}(\gamma)$, with $\mathsf{A} \in \mathtt{NULLABLE}(\mathcal{G})$

the algorithm for computing **FOLLOW** uses this extension

DEFINITION OF `LL(1)` PARSING TABLES: `FOLLOW`

Algorithmic definition: the function `FOLLOW`

Let $\mathcal{G} = (\mathbf{N}, \mathbf{T}, \rightarrow, \mathtt{S})$ be a context-free grammar. $\mathtt{FOLLOW}_\mathtt{i}$ are functions on $\mathbf{N}$ and defined as follows

1. $\mathtt{FOLLOW}_0(\mathtt{S}) = \{ \$ \}$ and $\mathtt{FOLLOW}_0(\mathtt{A}) = \varnothing$

2. $\mathtt{FOLLOW}_{\mathtt{i+1}}(\mathtt{X}) = \mathtt{FOLLOW}_\mathtt{i}(\mathtt{X}) \bigcup_{\mathtt{Z} \rightarrow \delta\, \mathtt{X}\, \gamma \text{ in } \mathcal{G}} \mathtt{FIRST}(\gamma) \setminus \{\varepsilon\}$
$\qquad \bigcup_{\mathtt{Z} \rightarrow \delta\, \mathtt{X}\, \gamma \text{ in } \mathcal{G} \text{ and } \mathtt{NULLABLE}(\gamma)} \mathtt{FOLLOW}_\mathtt{i}(\mathtt{Z})$

- it is easy to show that, for every `i`: $\mathtt{FOLLOW}_\mathtt{i}(\mathsf{A}) \subseteq \mathtt{FOLLOW}_{\mathtt{i+1}}(\mathsf{A}) \subseteq \mathbf{T} \cup \{ \$ \}$
- therefore there is `k` such that, for every A, $\mathtt{FOLLOW}_\mathtt{k}(\mathsf{A}) = \mathtt{FOLLOW}_{\mathtt{k+1}}(\mathsf{A})$

then $\mathtt{FOLLOW}(\mathsf{A}) = \mathtt{FOLLOW}_\mathtt{k}(\mathsf{A})$

remarks: `(1)` when the initial symbol does not appear on the rhs of productions, "`$`" is the unique symbol in its `FOLLOW`
`(2)` `FOLLOW` never contains "ε"

FOLLOW SETS — EXAMPLES

grammar

$$E \to T\ X \qquad\qquad X \to +\ E \mid \varepsilon$$
$$T \to (E)\ Y \mid \text{int}\ Y \qquad\qquad Y \to *\ T \mid \varepsilon$$

sets FOLLOW_i

$\text{FOLLOW}_0(E) = \{ \$ \}$ $\text{FOLLOW}_0(T) = \varnothing$ $\text{FOLLOW}_0(X) = \varnothing$ $\text{FOLLOW}_0(Y) = \varnothing$

$\text{FOLLOW}_1(E) = \{ \$,) \}$ $\text{FOLLOW}_1(T) = \{+, \$ \}$ $\text{FOLLOW}_1(X) = \{ \$ \}$ $\text{FOLLOW}_1(Y) = \varnothing$

$\text{FOLLOW}_2(E) = \{ \$,) \}$ $\text{FOLLOW}_2(T) = \{+, \$,) \}$ $\text{FOLLOW}_2(X) = \{\$,) \}$ $\text{FOLLOW}_2(Y) = \{+, \$\}$

$\text{FOLLOW}_3(E) = \{ \$,) \}$ $\text{FOLLOW}_3(T) = \{+, \$,) \}$ $\text{FOLLOW}_3(X) = \{\$,) \}$ $\text{FOLLOW}_3(Y) = \{+, \$,) \}$

$\text{FOLLOW}(E) = \{ \$,) \}$ $\text{FOLLOW}(T) = \{+, \$,) \}$

$\text{FOLLOW}(X) = \{\$,) \}$ $\text{FOLLOW}(Y) = \{+, \$,) \}$

FOLLOW SETS — EXAMPLES

grammar

$$S \rightarrow (S) S \mid \varepsilon$$

$FOLLOW_i$ sets

$$FOLLOW_0(S) = \{ \$ \}$$

$$FOLLOW_1(S) = \{ \$,) \}$$

$$FOLLOW(S) = FOLLOW_1(S) = \{ \$,) \}$$

FOLLOW SETS — EXAMPLES

grammar

$$
\begin{aligned}
Z &\rightarrow b \mid X\ Y\ Z \\
X &\rightarrow Y \mid a \\
Y &\rightarrow \varepsilon \mid c
\end{aligned}
$$

FOLLOW_i sets

$\mathrm{FOLLOW}_0(\ Z\ ) = \{\ \$\ \}$ $\mathrm{FOLLOW}_0(\ X\ ) = \varnothing$ $\mathrm{FOLLOW}_0(\ Y\ ) = \varnothing$

$\mathrm{FOLLOW}_1(\ Z\ ) = \{\ \$\ \}$

$\mathrm{FOLLOW}_1(\ X\ ) = \mathrm{FOLLOW}_0(\ X\ ) \cup \mathrm{FIRST}(\ Y\ )\backslash\{\varepsilon\} \cup \mathrm{FIRST}(Z\ ) = \{\ c\ \} \cup \{a,\ b,\ c\}$

$\mathrm{FOLLOW}_1(\ Y\ ) = \mathrm{FOLLOW}_0(\ Y\ ) \cup \mathrm{FIRST}(\ Z\ ) \cup \mathrm{FOLLOW}_0(X\ ) = \{a,\ b,\ c\}$

$\mathrm{FOLLOW}(\ Z\ ) = \{\ \$\ \}$ $\mathrm{FOLLOW}(\ X\ ) = \{a,\ b,\ c\}$

$\mathrm{FOLLOW}(\ Y\ ) = \{a,\ b,\ c\}$

DEFINITION OF `LL(1)` PARSING TABLES

the parsing table $\mathtt{LL}^1_{\mathcal{G}}$ for a grammar $\mathcal{G}$:

for every $\mathtt{A} \to \alpha$ in $\mathcal{G}$ do:

1. for every terminal $\mathtt{t} \in \mathtt{FIRST}(\alpha)$ do
   * $\mathtt{LL}^1_{\mathcal{G}}[\mathtt{A}, \mathtt{t}] = \mathtt{A} \to \alpha$

2. if $\varepsilon \in \mathtt{FIRST}(\alpha)$, for each $\mathtt{t} \in \mathtt{FOLLOW}(\mathtt{A})$ do
   * $\mathtt{LL}^1_{\mathcal{G}}[\mathtt{A}, \mathtt{t}] = \mathtt{A} \to \alpha$

[this rule applies also to $\mathtt{\$}$, i.e. when $\mathtt{\$} \in \mathtt{FOLLOW}(\mathtt{A})$:

if $\varepsilon \in \mathtt{FIRST}(\alpha)$ and $\mathtt{\$} \in \mathtt{FOLLOW}(\mathtt{A})$ do $\mathtt{LL}^1_{\mathcal{G}}[\mathtt{A}, \mathtt{\$}] = \mathtt{A} \to \alpha$

]

DEFINITION OF `LL(1)` PARSING TABLES: EXAMPLE

take the grammar

$$
\begin{array}{ll}
\texttt{E} \rightarrow \texttt{T X} & \texttt{X} \rightarrow \texttt{+ E} \mid \varepsilon \\
\texttt{T} \rightarrow \texttt{(E) Y} \mid \texttt{int Y} & \texttt{Y} \rightarrow \texttt{* T} \mid \varepsilon
\end{array}
$$

where in the line of `Y` we put `Y` → `*T` ?

- in the columns of `FIRST(*T)` = { `*` }

where in the line of `Y` we put `Y` → ε ?

- in the columns of `FOLLOW(Y)` = { `$, +, )` }

	int	*	+	(	)	$
T	T → int Y			T → (E)Y		
E	E → T X			E → T X		
X			X → +E		X → ε	X → ε
Y		Y → *T	Y → ε		Y → ε	Y → ε

REMARKS ABOUT `LL(1)` TABLES

if any entry is **multiply defined** then $\mathcal{G}$ is not LL(1)

in particular when

* $\mathcal{G}$ is **left recursive**
* $\mathcal{G}$ is **not left-factored**
* $\mathcal{G}$ is **ambiguous**
* and in other cases as well

most programming language grammars are not `LL(1)`

* there are tools that build `LL(1)` tables
* the parser generator **`ANTLR`** uses the **`LL`** approach

REMOVING LEFT RECURSION

[see def. slide 12] a grammar si called **left-recursive** if it has a non-terminal $\mathtt{A}$ such that $\mathtt{A} \Longrightarrow^{+} \mathtt{A}\,\gamma$, for some γ

case of DIRECT LEFT-RECURSION, i.e. there is $\mathtt{A}$ such that

$$
\left.
\begin{array}{lll}
\mathtt{A} & \rightarrow & \mathtt{A}\,\gamma_1 \\
 & \vdots & \\
\mathtt{A} & \rightarrow & \mathtt{A}\,\gamma_m \\
\mathtt{A} & \rightarrow & \delta_1 \\
 & \vdots & \\
\mathtt{A} & \rightarrow & \delta_n
\end{array}
\right.
$$

($\mathtt{A} \rightarrow \delta_1$ … $\mathtt{A} \rightarrow \delta_n$): $\delta_1 \ldots \delta_n$ do not start with $\mathtt{A}$

remark: the grammar is equivalent to the regular expression

$$(\,\delta_1 |\ldots|\,\delta_n\,)(\,\gamma_1\,|\ldots|\,\gamma_m\,)^*$$

REMOVING DIRECT LEFT RECURSION

$$\left.\begin{array}{lll} A \to A\,\gamma_1 & \quad & A \to \delta_1 \\ \vdots & & \vdots \\ A \to A\,\gamma_m & & A \to \delta_n \end{array}\right\} \; \delta_1 \ldots \delta_n \text{ do not start with } A$$

is rewritten into — we use a new non terminal A'

$$\begin{array}{lllll} A \to \delta_1\, A' & \quad & A' \to \gamma_1\, A' & \quad & A' \to \varepsilon \\ \vdots & & \vdots & & \\ A \to \delta_n\, A' & & A' \to \gamma_m\, A' & & \end{array}$$

remarks

* since the δ_i do not start with A there is no direct left-recursion anymore
* since the A' is a new non-terminal, the γ_i *cannot start with it*
* there may be **indirect left-recursions** if, for some i, NULLABLE(γ_i)

REMOVING DIRECT LEFT RECURSION/EXAMPLE

$$
\begin{aligned}
E &\to E + F \\
E &\to E - F \\
E &\to F \\
F &\to F * T \\
F &\to F / T \\
F &\to T \\
T &\to \texttt{num} \\
T &\to (E)
\end{aligned}
$$

is rewritten into

$$
\begin{aligned}
E &\to F\ E' \\
E' &\to +\ F\ E' \\
E' &\to -\ F\ E' \\
E' &\to \varepsilon \\
F &\to T\ F' \\
F' &\to *\ T\ F' \\
F' &\to /\ T\ F' \\
F' &\to \varepsilon \\
T &\to \texttt{num} \\
T &\to (E)
\end{aligned}
$$

exercise: build the `LL(1)` table

REMOVING INDIRECT LEFT RECURSION

there are several possibilities

1. case of MUTUAL LEFT-RECURSION:

$$\left.\begin{array}{lcl} A_1 & \to & A_2\,\gamma_1 \\ A_2 & \to & A_3\,\gamma_2 \\ & \vdots & \\ A_{k-1} & \to & A_k\,\gamma_{k-1} \\ A_k & \to & A_1\,\gamma_k \end{array}\right\}$$

break the mutual recursion, e.g. replace

$$A_1 \to A_2\,\gamma_1$$

with

$$A_1 \to A_1\,\gamma_k \ldots \gamma_1$$

and solve the direct left recursion

2. there is a production

$$A \to \gamma\, A\, \delta \quad \text{where } \mathtt{NULLABLE}(\gamma)$$

3. any combination of 1 and 2

it is always possible to rewrite indirect left recursion into direct one

- the **process is a bit complex**

LEFT FACTORING

the grammar

```
E  →   T + E | T
T  →   (E) | (E)*T  |  int  |  int * T
```

is impossible to predict because

- the non-terminal `T` has **two productions** that begin with "`(`" and **two productions** that begin with "`int`"
- the non-terminal `E` has the two productions that begin with `T` and **it is not evident how to predict**

the above grammar **must be left-factored** before using predictive parsers

LEFT-FACTORING — AN EXAMPLE

the grammar

```
E  →  T + E | T
T  →  (E) | (E)*T | int  |  int * T
```

is left-factored as follows

```
E   →  T E'
E'  →  + E | ε
T   →  (E) T' | int T'
T'  →  *T | ε
```

PROBLEM: left-factoring the standard `if-then-else` statement

```
Stat → if Exp then Stat else Stat | if Exp then Stat
```

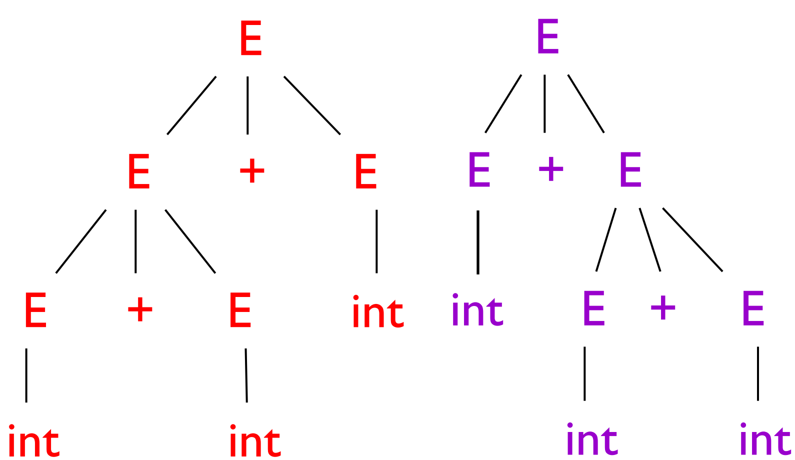
AMBIGUITY

a grammar is **ambiguous** if it has **more than one parse tree** for some string

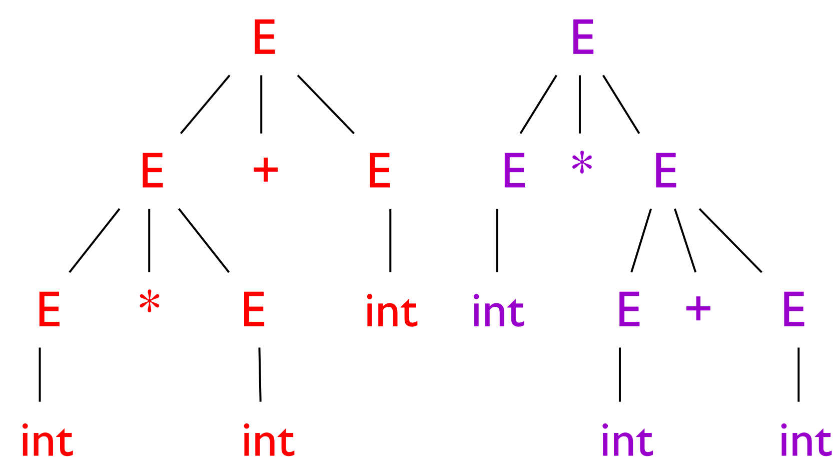
- equivalently, there is **more than one rightmost** or **leftmost derivation** for some string

example: `E → E + E | E * E | (E) | int` is ambiguous

because `int+int+int` `int*int+int` have two parse trees



+ is left-associative



* has higher precedence than +

AMBIGUITY

ambiguity is **bad**

- leaves meaning of some programs ill-defined

ambiguity is **common** in programming languages

- arithmetic expressions
- `if-then-else`

in `LL` parsing it is possible to deal with ambiguity by

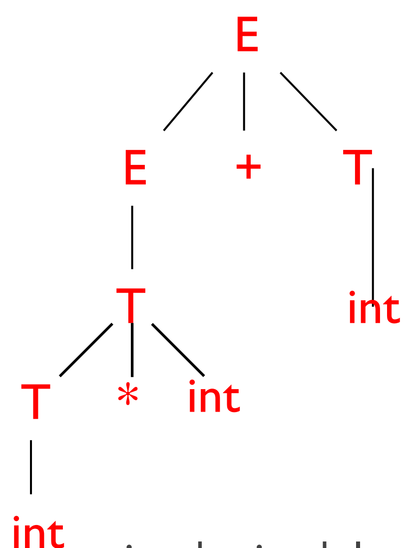
- **rewriting grammars**

DEALING WITH AMBIGUITY

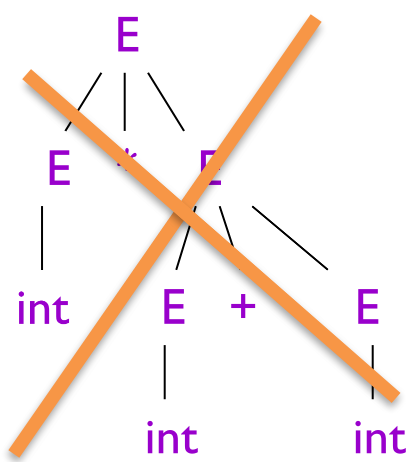
rewrite the grammar for expressions in an unambiguous way:

```
E → E + T | T
T → T * int | T * ( E ) | int | ( E )
```

- enforces precedence of * over +
- enforces left-associativity of + and *



is derivable



is not derivable

the new grammar is neither `LL(1)` nor adeguate for rec.descent parsing (left-recursion)

DEALING WITH AMBIGUITY: THE DANGLING ELSE

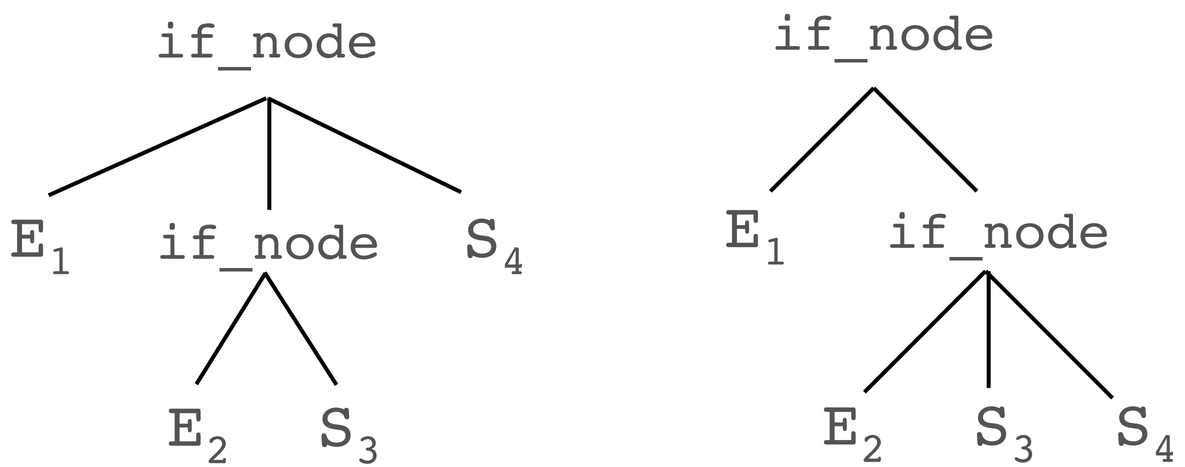
the grammar

```
S → ID '=' E | 'if' E 'then' S | 'if' E 'then' S 'else' S
```

is also **ambiguous** because the statement

$$\texttt{if } E_1 \texttt{ then if } E_2 \texttt{ then } S_3 \texttt{ else } S_4$$

has two (abstract) parse trees



in programming languages we want the tree in the right

THE DANGLING ELSE: A FIX

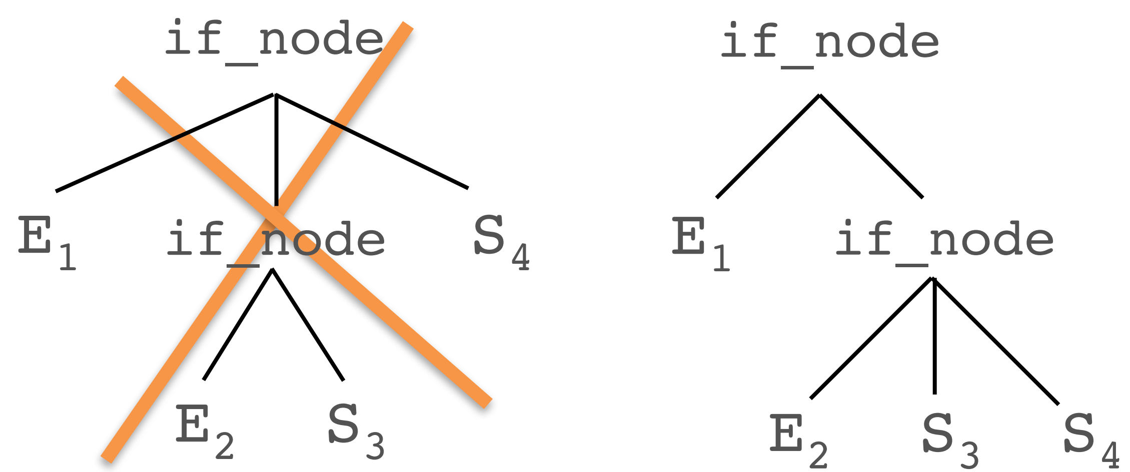
else matches the closest unmatched then

we can **factorize** the **`if-then`** part and **rewrite** the grammar as follows:

```
S     →    ID = E
           | if E then S ELSE
ELSE  →    else S | ε
```

(this new grammar describes the same set of strings and allows the same derivations)

this is a standard hack to ban this derivation: give priority to the "**`else`**" token



THE DANGLING ELSE: A THEORETICAL FIX

see Gabbrielli-Martini

```
statement             : statementNoIf
                      | ifThenStatement
                      | ifThenElseStatement
                      ;

statementNoIf         : // the statements without if
                      ;
ifThenStatement       : 'if' '(' exp ')' 'then' statement ;

ifThenElseStatement : 'if' '(' exp ')' 'then' statementNoShortIf 'else'
                      statement ;

statementNoShortIf  : statementNoIf
                      |'if' '(' exp ')' 'then'
                          statementNoShortIf 'else' statementNoShortIf ;
```

the grammar is NOT LL(1): it must be left-factorized!
even if it is accepted in `ANTLR`!

AMBIGUITY

there is no general techniques for handling ambiguity by transforming grammar

- it is always preferable **not to change a grammar**

instead of rewriting the grammar

- use the more natural (ambiguous) grammar
- along with **disambiguating declarations**

ANTLR

* start rule
* ambiguity
* left recursion
* non-`LL(*)` decision errors

ANTLR — START RULE

it is not so anymore in ANTLR v4

any grammar needs a so-called start rule

- start rule is a rule that is not referenced by another rule
- if your grammar does not have such rule, ANTLR generator will issue a warning:

```
no start rule (no rule can obviously be followed by EOF)
```

to avoid it, add a dummy start rule to your grammar:

```
start_rule: someOtherRule ;
```

ANTLR — AMBIGUITY

example:

```
exp : exp '+' exp | exp '*' exp | NUM | '(' exp ')';
NUM: ('0'..'9')+;
```

this grammar should recognize inputs for a simple calculator

- `ANTLR v4` behaves badly
- to understand the problem, recall that `ANTLR` goes from **left to right** whenever parsing an input
  - it first decides which alternative it will use **following the order of the rules**
  - then it sticks with the decision
- **remark**: left-factorization is solved automatically by `ANTLR`

ANTLR — AMBIGUITY

example:

```
exp : exp '+' exp | exp '*' exp | NUM | '(' exp ')';
NUM: ('0'..'9')+;
```

- try to simulate it on the input

  `1 + 3 * 4`

- does it match an **`exp '+' exp`** alternative, or an **`exp '*' exp`** alternative?

- the error suggests to reorder the rules as follows:

```
exp : exp '*' exp | exp '+' exp | NUMBER | '(' exp ')';
NUMBER: ('0'..'9')+;
```

ANTLR — PROBLEMS WITH ASSOCIATIVITY

problem: extend the grammar with "/" and force it to be right-associative

```
exp : exp '/' exp | exp '*' exp | exp '+' exp | '(' exp ')' | NUM   ;
NUM: ('0'..'9')+;
```

solution: use a new nonterminal!

```
exp : term | exp '+' term ;
term : factor | factor '/' term | term '*' factor  ;
factor : '(' exp ')' | NUM  ;
```

better solution: use the "right-associativity" annotation!

```
exp : <assoc=right> exp '/' exp      // the annotation must be written to
      | exp '+' exp                  // the left
      | exp '*' exp
      | '(' exp ')'
      | NUM              ;
```

NEXT LECTURE