Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-15/

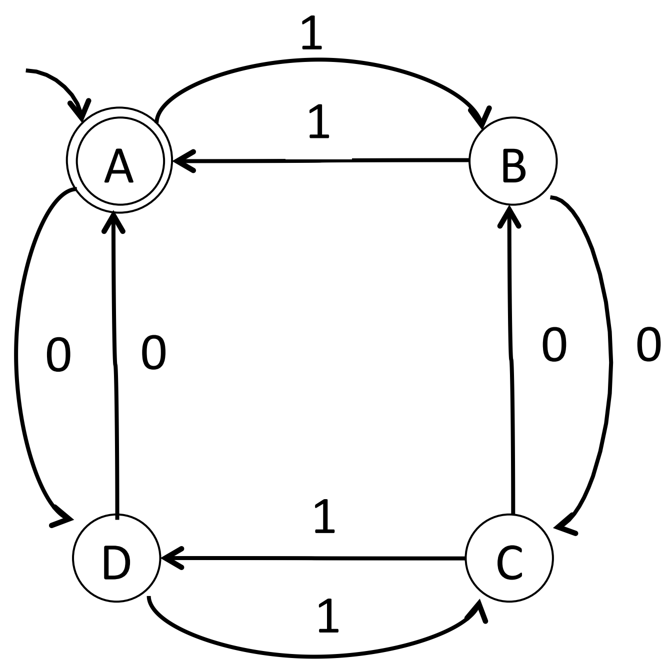
Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 7

- From DSA to Regular Expression
- Top-down parsing

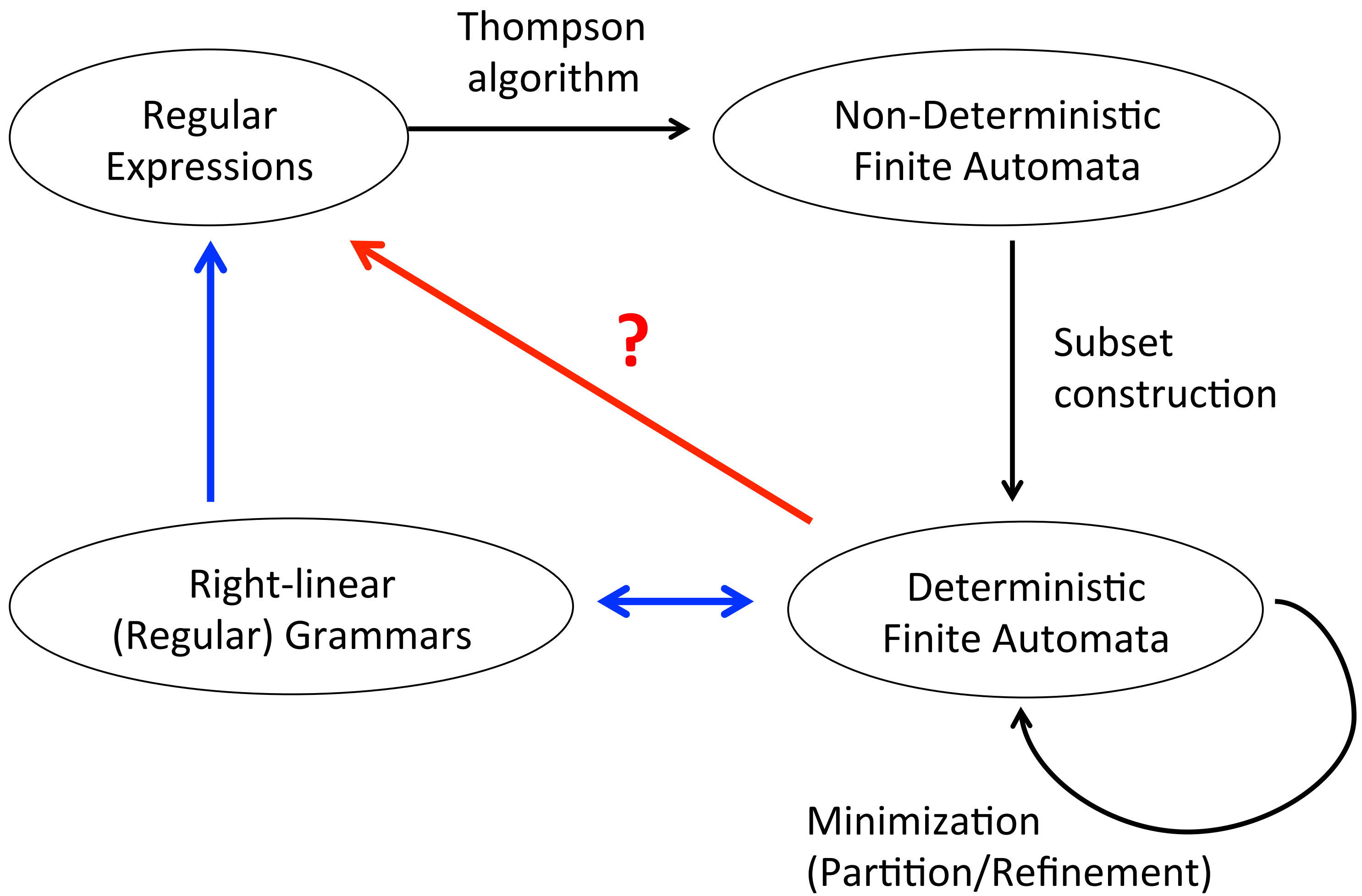
Motivations: exercise 7(b)

- Write a regular expression over the set of symbols {0,1} that describes the language of all strings having an even number of 0’s and of 1’s
  - Not easy….
  - A solution: (00|11)*((01|10)(00|11)*(01|10)(00|11)*)*
  - How can we get it?



- Towards the solution: a deterministic automaton accepting the language
- But how do we get the regular expression defining the language accepted by the automaton?

Regular expressions, Automata, and all that…

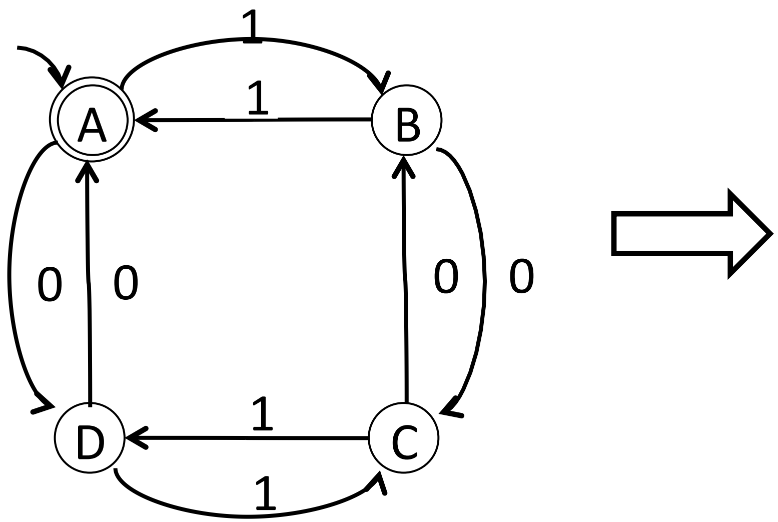


From automata to Regular Expressions

- Three approaches:
  - Dynamic Programming [Scott, Section 2.4 on CD] [Hopcroft, Motwani, Ullman, *Introduction to Automata Theory, Languages and Computation*, Section 3.2.1]
  - Incremental state elimination [HMU, Section 3.2.2]
  - Regular Expression as fixed-point of a continuous function on languages

DFAs and Right-linear Grammars

- In a *right-linear* (*regular*) grammar each production is of the form $A \rightarrow w\,B$ or $A \rightarrow w$ $(w \in T^*)$
- From a DFA to a right-linear grammar



A → ε | 1B | 0D
B → 1A | 0C
C → 0B | 1D
D → 0A | 1C

- The construction also works for NFA
- A similar construction can transform any right-linear grammar into an NFA (productions might need to be transformed introducing new non-terminals)

Kleene fixed-point theorem

- A *complete partial order (CPO)* is a partial order with a least element $\bot$ and such that every increasing chain has a supremum
- Theorem: *Every continuous function F over a complete partial order (CPO) has a least fixed-point, which is the supremum of chain*

$$F(\bot) \leq F(F(\bot)) \leq \ldots \leq F^n(\bot) \leq ..$$

Context Free grammars as functions on the CPO of languages

- Languages over Σ form a *complete partial order* under set inclusion
- A context free grammar defines a continuous function over (tuples of) languages
  - **A -> *a* | *b*A** $F(L) = \{a\} \cup \{bw \mid w \in L\}$
- The language generated by the grammar is the least-fixed point of the associated function
  - $\varnothing \subset \{a\} \subset \{a, ba\} \subset \{a, ba, bba\} \subset \ldots \subset \{b^n a \mid n \geq 0\}$
- In the case of right-linear grammars we can describe the least fixed-point as a regular expression
  - *Lang(***A***) = b*a*

Example: from right-linear grammar to regular expression

A → ε | 1B | 0D
B → 1A | 0C
C → 0B | 1D
D → 0A | 1C

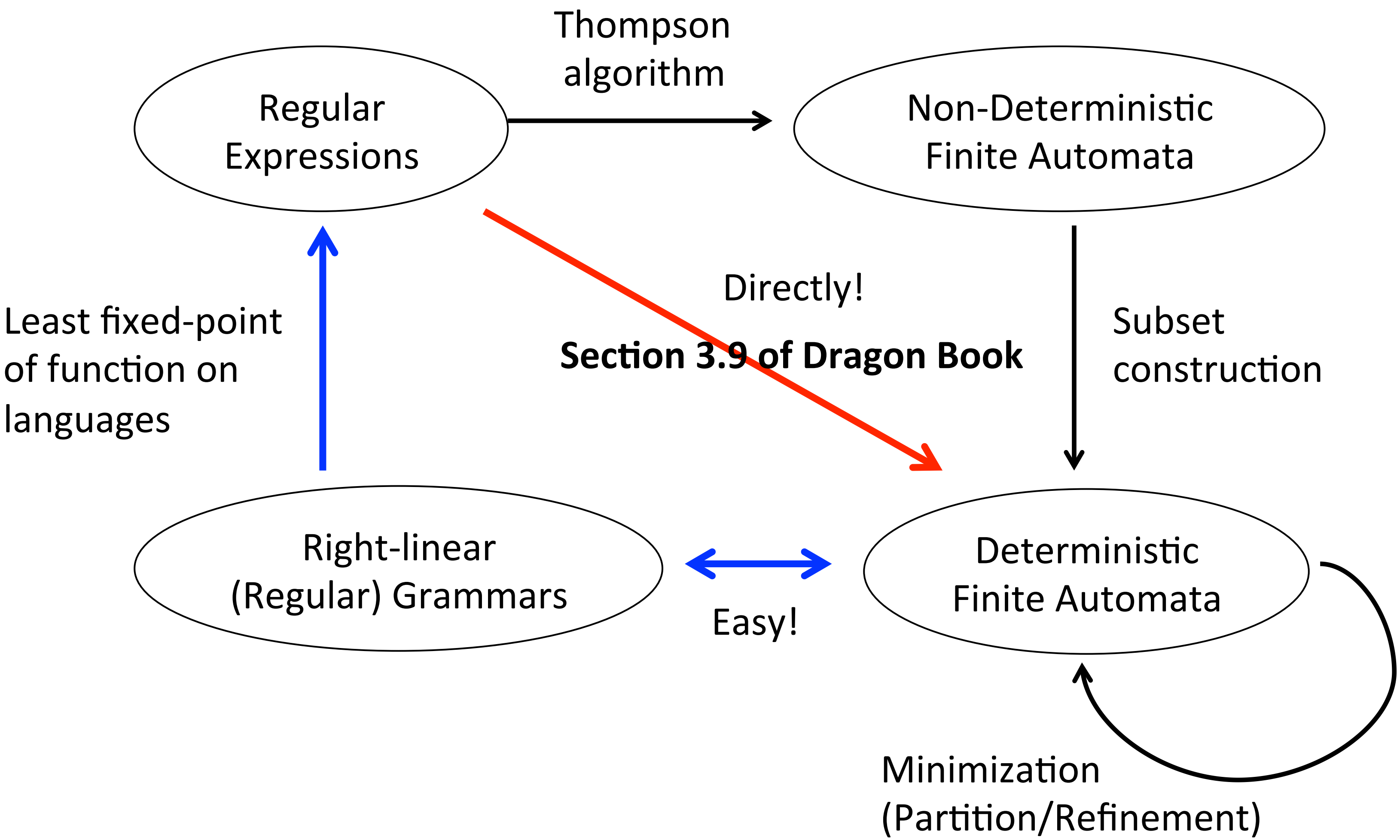
1) Substitute D in A and C
A → ε | 1B | 0(0A | 1C)
B → 1A | 0C
C → 0B | 1(0A | 1C)

2) Substitute B in A and C
A → ε | 1(1A | 0C) | 0(0A | 1C)
C → 0(1A | 0C) | 1(0A | 1C)

3) Put C in form C = α | βC
A → ε | 1(1A | 0C) | 0(0A | 1C)
C → 01A | 10A | (00 | 11)C
4) Solve C: C = (00 | 11)*(01A | 10A)

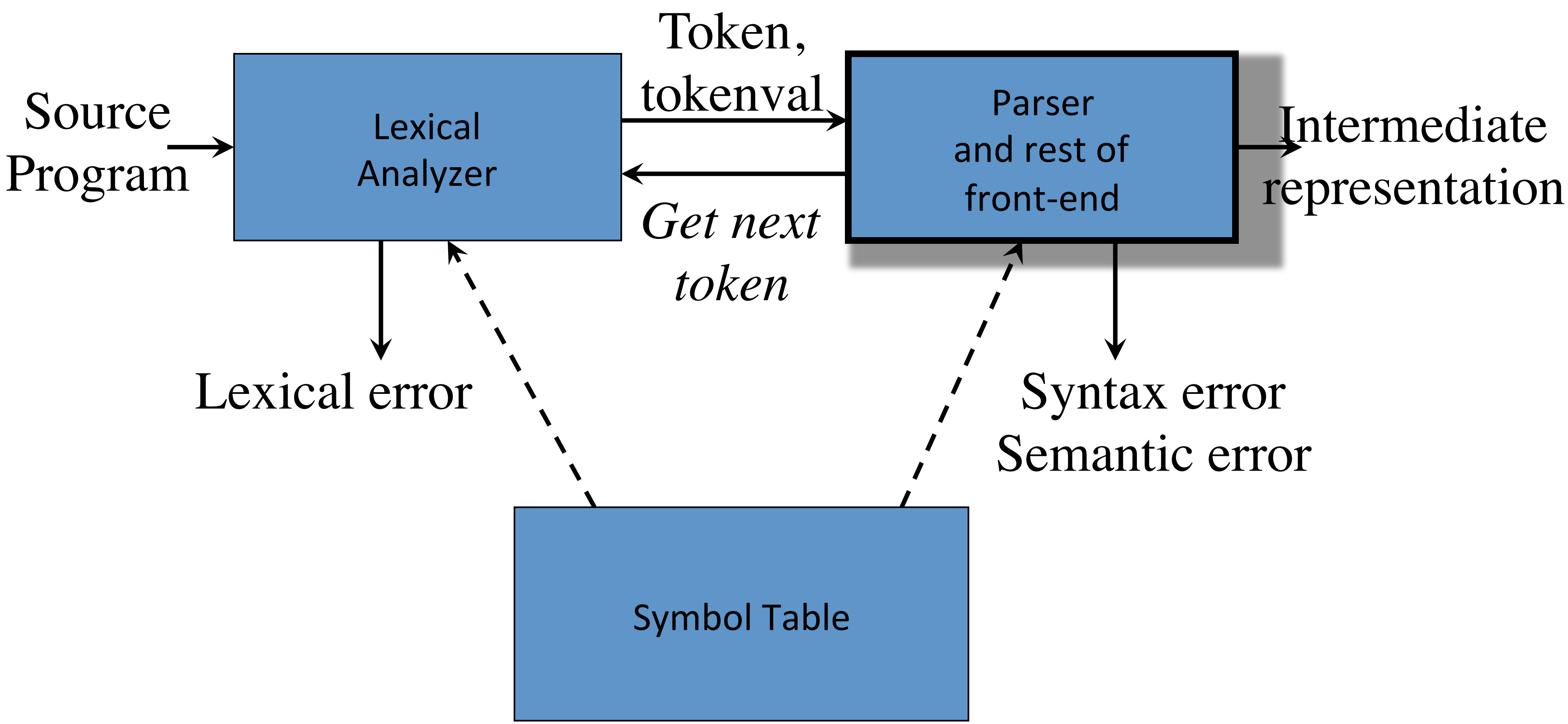
5) Factorize C in A
A → ε | 11A | 00A | (10 | 01)C
6) Substitute C in A
A → ε | 11A | 00A | (10 | 01) (00 | 11)*(01A | 10A)
7) Put A in form A = α | βA
A → ε | (11 | 00 | (10 | 01) (00 | 11)*(01 | 10))A
8) Solve A: A = **(11 | 00 | (10 | 01) (00 | 11)*(01 | 10))***
The other solution: (00|11)*((01|10)(00|11)*(01|10)(00|11)*)*

Regular expressions, Automata, and all that…



Top-down Parsing

Position of a Parser in the Compiler Model



The syntax of programming languages

- The syntax of a programming language is typically defined by two grammars
  - Lexical grammar
    - Regular, often presented as regular expressions
    - Terminal symbols are characters
    - Defines tokens
  - Syntax grammar
    - Context-free, often presented in Backus-Naur form
    - Terminal symbols are tokens
    - Defines constructs of the language, not expressible with REs
  - Note: there are non-context free syntact constructs
    - Variables are declared before use → $\{wcw \mid w \in (a \mid b)^*\}$
    - Number of actual/formal parameters → $\{a^n b^m c^n d^m \mid n > 0, m > 0\}$

Towards parsing

- A parser implements a Context-Free grammar as a recognizer of strings
  - It checks that the input string (of tokens) is generated by the syntax grammar
  - Possibly generates the parse tree
  - Reports syntax errors accurately
  - *Invokes semantic actions*
    - *For static semantics checking, e.g. type checking of expressions, functions, etc.*
    - *For syntax-directed translation of the source code to an intermediate representation*

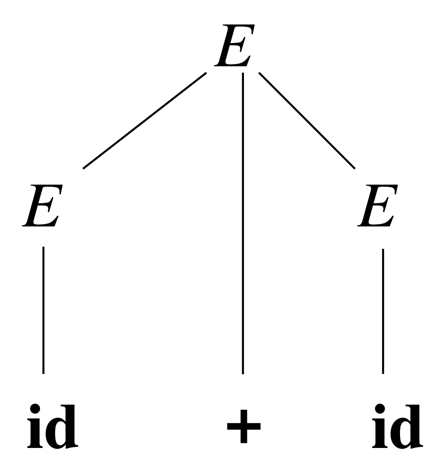
Parse trees and derivations

- A parse tree may correspond to several derivations
- A parse tree has a unique *rightmost* (*leftmost*) derivation

$P = E \rightarrow E \mathbf{+} E \mid \mathbf{id}$

$E \Rightarrow_{rm} E \mathbf{+} E \Rightarrow_{rm} E \mathbf{+\ id} \Rightarrow_{rm} \mathbf{id + id}$

$E \Rightarrow_{lm} E \mathbf{+} E \Rightarrow_{lm} \mathbf{id} \mathbf{+} E \Rightarrow_{lm} \mathbf{id + id}$



Parsing algorithms

- *Universal* (any C-F grammar)
  - Cocke-Younger-Kasimi, Earley
  - Based on dynamic programming, $O(n^3)$
- *Top-down* (C-F grammar with restrictions)
  - Recursive descent (predictive parsing)
  - LL (Left-to-right, Leftmost derivation) methods
  - Linear on certain grammars; easier to do manually
- *Bottom-up* (C-F grammar with restrictions)
  - Operator precedence parsing
  - LR (Left-to-right, Rightmost derivation) methods
    - SLR, canonical LR, LALR
  - Linear on certain grammars; typically generated by tools

Top-Down Parsing

- LL methods (Left-to-right, Leftmost derivation) and recursive-descent parsing

Grammar:

$E \rightarrow T$ **+** T

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

$T \rightarrow$ **-** E

$T \rightarrow$ **id**

String

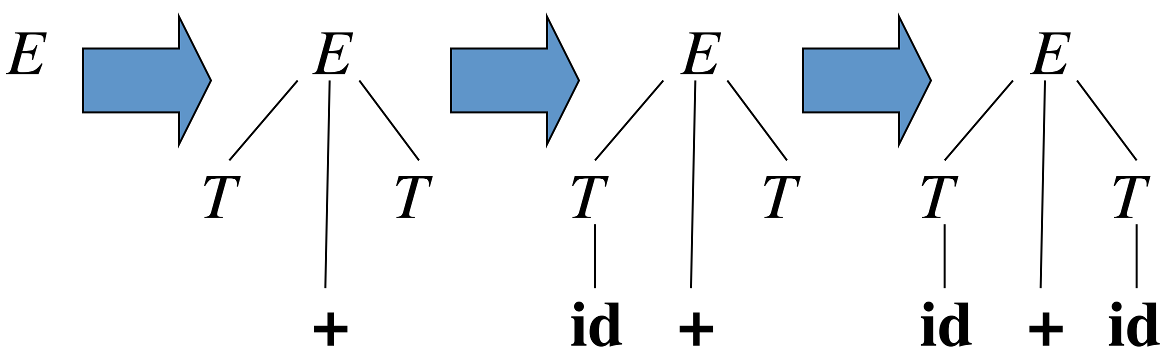
id + id

Leftmost derivation:

$E \Rightarrow_{lm} T$ **+** T

$\Rightarrow_{lm}$ **id +** T

$\Rightarrow_{lm}$ **id + id**



LL(*k*) parsing

- Top-down parsing is efficient if the grammar satisfies certain conditions
- Whenever we have to expand a non-terminal, the next *k* token should determine the production to use (*lookahead*)
- In this case the grammar is LL(*k*)
- Most constructs are LL(1), and we will focus on this class of grammars

Left Recursion

- A grammar is *left-recursive* if there is a non-terminal *A* such that $A \Rightarrow^{+} A\eta$ for some string η
  - Example of immediate left-recursion:
    $A \rightarrow A\alpha \mid A\beta \mid \gamma \mid \delta$
  - Left recursion can be indirect
- If the grammar is left-recursive, it cannot be LL(*k*): a top-down parser loops forever on certain inputs
- Immediate left recursion elimination:

$A \rightarrow \gamma\, A_R \mid \delta\, A_R \qquad A_R \rightarrow \alpha\, A_R \mid \beta\, A_R \mid \varepsilon$

A General Left Recursion Elimination Method

- *Input: Grammar G with no cycles or ε-productions*
- Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$

for $i = 1, \ldots, n$ **do**
  for $j = 1, \ldots, i\text{-}1$ **do**
    replace each
      $A_i \rightarrow A_j\, \gamma$
    with
      $A_i \rightarrow \delta_1\, \gamma \mid \delta_2\, \gamma \mid \ldots \mid \delta_k\, \gamma$
    where
      $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
  enddo
  eliminate the *immediate left recursion* in A_i
enddo

Example of left-recursion elimination

$$\left.\begin{aligned} A &\rightarrow B\ C \mid \mathbf{a} \\ B &\rightarrow C\ A \mid A\ \mathbf{b} \\ C &\rightarrow A\ B \mid C\ C \mid \mathbf{a} \end{aligned}\right\} \text{Choose arrangement: } A, B, C$$

$i = 1$: nothing to do

$i = 2, j = 1$: $B \rightarrow C\ A \mid \underline{A}\ \mathbf{b}$

$\Rightarrow\ B \rightarrow C\ A \mid \underline{B\ C}\ \mathbf{b} \mid \underline{\mathbf{a}}\ \mathbf{b}$

$\Rightarrow_{\text{(imm)}}\ B \rightarrow C\ A\ B_R \mid \mathbf{a}\ \mathbf{b}\ B_R$

$B_R \rightarrow C\ \mathbf{b}\ B_R \mid \varepsilon$

$i = 3, j = 1$: $C \rightarrow \underline{A}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow\ C \rightarrow \underline{B\ C}\ B \mid \underline{\mathbf{a}}\ B \mid C\ C \mid \mathbf{a}$

$i = 3, j = 2$: $C \rightarrow \underline{B}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow\ C \rightarrow \underline{C\ A\ B_R}\ C\ B \mid \underline{\mathbf{a}\ \mathbf{b}\ B_R}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow_{\text{(imm)}}\ C \rightarrow \mathbf{a}\ \mathbf{b}\ B_R\ C\ B\ C_R \mid \mathbf{a}\ B\ C_R \mid \mathbf{a}\ C_R$

$C_R \rightarrow A\ B_R\ C\ B\ C_R \mid C\ C_R \mid \varepsilon$

Example of left-recursion elimination: Grammars for expressions

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

Grammar after left recursion elimination

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

Left Factoring

- If a nonterminal has two or more productions whose right-hand sides start with the same symbol, the grammar is not LL(1)
- Example:
  - *stmt* **::=** **if** *expr* **then** *stmt* **else** *stmt*
    | **if** *expr* **then** *stmt*
- Solution: replace productions

$$A \to \alpha\, \beta_1 \mid \alpha\, \beta_2 \mid \ldots \mid \alpha\, \beta_n \mid \gamma$$

with

$$A \to \alpha\, A_R \mid \gamma$$

$$A_R \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

- Example:
  - *stmt* **::=** **if** *expr* **then** *stmt* *stmt'*
  - *stmt'* **::=** **else** *stmt* | ε

Predictive Parsing

- Eliminate left recursion from grammar
- Left factor the grammar
- Compute FIRST and FOLLOW, and check if the grammar is LL(1)
- FIRST and FOLLOW are used in the parsing algorithm
- Two variants:
  - Recursive (recursive-descent parsing)
  - Non-recursive (table-driven parsing)

FIRST (Revisited)

- FIRST(α) = { *the set of terminals that begin all strings derived from* α }
- FIRST(a) = {a} if $a \in T$
- FIRST(ε) = {ε}
- FIRST(A) = $\cup_{A\to\alpha}$ FIRST(α) for $A\to\alpha \in P$
- FIRST($X_1X_2 \ldots X_k$) =

  if for all j = 1, …, i-1 : $\varepsilon \in$ FIRST(X_j) **then**

  add non-ε in FIRST(X_i) to FIRST($X_1X_2 \ldots X_k$)

  if for all j = 1, …, k : $\varepsilon \in$ FIRST(X_j) **then**

  add ε to FIRST($X_1X_2 \ldots X_k$)

FOLLOW

- FOLLOW(A) = { *the set of terminals that can immediately follow nonterminal A* }
- FOLLOW(A) =

  for all ($B \rightarrow \alpha\ A\ \beta$) $\in P$ **do**
  add FIRST(β)\{ε} to FOLLOW(A)

  for all ($B \rightarrow \alpha\ A\ \beta$) $\in P$ and $\varepsilon \in$ FIRST(β) **do**
  add FOLLOW(B) to FOLLOW(A)

  for all ($B \rightarrow \alpha\ A$) $\in P$ **do**
  add FOLLOW(B) to FOLLOW(A)

  if A is the start symbol S **then**
  add **$** to FOLLOW($A$)

LL(1) Grammar

- A grammar G is LL(1) if it is not left recursive and for each collection of productions
  $$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$
  for nonterminal A the following holds:

  1. $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$ for all $i \neq j$
  2. if $\alpha_i \Rightarrow^* \varepsilon$ then

     2.a. $\alpha_j \not\Rightarrow^* \varepsilon$ for all $i \neq j$

     2.b. $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$ for all $i \neq j$

Non-LL(1) Examples

Grammar	*Not LL(1) because:*
$S \rightarrow S\,\mathbf{a} \mid \mathbf{a}$	Left recursive
$S \rightarrow \mathbf{a}\,S \mid \mathbf{a}$	$\text{FIRST}(\mathbf{a}\,S) \cap \text{FIRST}(\mathbf{a}) \neq \emptyset$
$S \rightarrow \mathbf{a}\,R \mid \varepsilon$ $R \rightarrow S \mid \varepsilon$	For R: $S \Rightarrow^* \varepsilon$ and $\varepsilon \Rightarrow^* \varepsilon$
$S \rightarrow \mathbf{a}\,R\,\mathbf{a}$ $R \rightarrow S \mid \varepsilon$	For R: $\text{FIRST}(S) \cap \text{FOLLOW}(R) \neq \emptyset$

Recursive-Descent Parsing

- Grammar must be LL(1)
- Every nonterminal has one (recursive) procedure responsible for parsing the nonterminal's syntactic category of input tokens
- When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input look-ahead information

Using FIRST and FOLLOW in a Recursive-Descent Parser

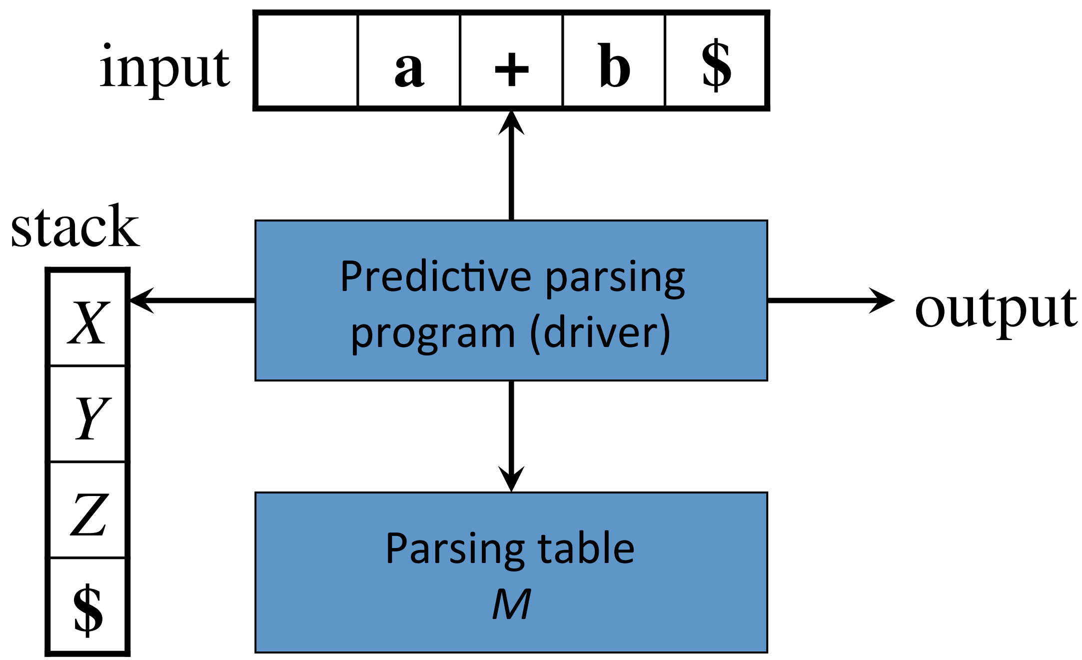
expr → *term rest*
rest → **+** *term rest*
| **-** *term rest*
| ε
term → **id**

```
procedure rest();
begin
    if lookahead in FIRST(+ term rest) then
        match( '+' ); term(); rest()
    else if lookahead in FIRST(- term rest) then
        match( '-' ); term(); rest()
    else if lookahead in FOLLOW(rest) then
        return
    else error()
end;
```

where FIRST(**+** *term rest*) = { **+** }
FIRST(**-** *term rest*) = { **-** }
FOLLOW(*rest*) = { **$** }

Non-Recursive Predictive Parsing: Table-Driven Parsing

- Given an LL(1) grammar $G = (N, T, P, S)$ construct a table M and use a *driver program* with a *stack*
- The stack replaces the runtime stack of the recursive algorithm. It will contain symbols of the grammar.



Constructing an LL(1) Predictive Parsing Table

- Table M has one entry $M[A, a]$ for each $A \in N$ and $a \in T$
- Entry $M[A, a]$ contains the production to apply when A has to be reduced and a is the lookahead

```
for each production A → α do
    for each a ∈ FIRST(α) do
        add production  A → α  to M[A,a]
    enddo
    if ε ∈ FIRST(α) then
        for each b ∈ FOLLOW(A) do
            add A → α to M[A,b]
        enddo
    endif
enddo
```

- Mark each undefined entry in M error
- **Note:** The grammar is LL(1) iff $M[A, a]$ contains at most one production for each $A \in N$ and $a \in T$

Example Table

$E \rightarrow T\ E_R$
$E_R \rightarrow \mathbf{+}\ T\ E_R \mid \varepsilon$
$T \rightarrow F\ T_R$
$T_R \rightarrow *\ F\ T_R \mid \varepsilon$
$F \rightarrow \mathbf{(}\ E\ \mathbf{)} \mid \mathbf{id}$

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$E \rightarrow T\ E_R$	**(id**	**$)**
$E_R \rightarrow \mathbf{+}\ T\ E_R$	**+**	**$)**
$E_R \rightarrow \varepsilon$	ε	
$T \rightarrow F\ T_R$	**(id**	**+ $)**
$T_R \rightarrow *\ F\ T_R$	*	**+ $)**
$T_R \rightarrow \varepsilon$	ε	
$F \rightarrow (\ E\ )$	**(**	* **+ $)**
$F \rightarrow \mathbf{id}$	**id**	* **+ $)**

	id	**+**	*	**(**	**)**	**$**
E	$E \rightarrow T\ E_R$			$E \rightarrow T\ E_R$		
E_R		$E_R \rightarrow \mathbf{+}\ T\ E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F\ T_R$			$T \rightarrow F\ T_R$		
T_R		$T_R \rightarrow \varepsilon$	$T_R \rightarrow *\ F\ T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (\ E\ )$		

Predictive Parsing Program (Driver)

```
push($)
push(S)
a := lookahead
repeat
    X := pop()
    if X is a terminal or X = $ then
        match(X)      // moves to next token and a := lookahead
    else if M[X,a] = X → Y1Y2…Yk then
        push(Yk, Yk-1, …, Y2, Y1) // such that Y1 is on top
        … invoke actions and/or produce IR output …
    else    error()
    endif
until X = $
```

Example Table-Driven Parsing

Stack	Input	Production applied
$\$E$	$\mathbf{id{+}id{*}id\$}$	$E \rightarrow T\ E_R$
$\$E_R T$	$\mathbf{id{+}id{*}id\$}$	$T \rightarrow F\ T_R$
$\$E_R T_R F$	$\mathbf{id{+}id{*}id\$}$	$F \rightarrow \mathbf{id}$
$\$E_R T_R \mathbf{id}$	$\mathbf{id{+}id{*}id\$}$	
$\$E_R T_R$	$\mathbf{{+}id{*}id\$}$	$T_R \rightarrow \varepsilon$
$\$E_R$	$\mathbf{{+}id{*}id\$}$	$E_R \rightarrow \mathbf{+}\ T\ E_R$
$\$E_R T \mathbf{+}$	$\mathbf{{+}id{*}id\$}$	
$\$E_R T$	$\mathbf{id{*}id\$}$	$T \rightarrow F\ T_R$
$\$E_R T_R F$	$\mathbf{id{*}id\$}$	$F \rightarrow \mathbf{id}$

Stack	Input	Prod. applied
$\$E_R T_R \mathbf{id}$	$\mathbf{id{*}id\$}$	
$\$E_R T_R$	$\mathbf{{*}id\$}$	$T_R \rightarrow *\ F\ T_R$
$\$E_R T_R F *$	$\mathbf{{*}id\$}$	
$\$E_R T_R F$	$\mathbf{id\$}$	$F \rightarrow \mathbf{id}$
$\$E_R T_R \mathbf{id}$	$\mathbf{id\$}$	
$\$E_R T_R$	$\mathbf{\$}$	$T_R \rightarrow \varepsilon$
$\$E_R$	$\mathbf{\$}$	$E_R \rightarrow \varepsilon$
$\$$	$\mathbf{\$}$	

	id	**+**	*	(	)	**$**
E	$E \rightarrow T\ E_R$			$E \rightarrow T\ E_R$		
E_R		$E_R \rightarrow \mathbf{+}\ T\ E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F\ T_R$			$T \rightarrow F\ T_R$		
T_R		$T_R \rightarrow \varepsilon$	$T_R \rightarrow *\ F\ T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (\ E\ )$		

LL(1) Grammars are Unambiguous

Ambiguous grammar

$S \rightarrow \textbf{i}\ E\ \textbf{t}\ S\ S_R \mid \textbf{a}$

$S_R \rightarrow \textbf{e}\ S \mid \varepsilon$

$E \rightarrow \textbf{b}$

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$S \rightarrow \textbf{i}\ E\ \textbf{t}\ S\ S_R$	**i**	**e $**
$S \rightarrow \textbf{a}$	**a**	
$S_R \rightarrow \textbf{e}\ S$	**e**	**e $**
$S_R \rightarrow \varepsilon$	ε	
$E \rightarrow \textbf{b}$	**b**	**t**

Error: duplicate table entry

	a	**b**	**e**	**i**	**t**	**$**
S	$S \rightarrow \textbf{a}$			$S \rightarrow \textbf{i}\ E\ \textbf{t}\ S\ S_R$		
S_R			$S_R \rightarrow \varepsilon$ $S_R \rightarrow \textbf{e}\ S$			$S_R \rightarrow \varepsilon$
E		$E \rightarrow \textbf{b}$				

Error Handling

- A good compiler should assist in identifying and locating errors
  - *Lexical errors*: compiler can easily recover and continue (e.g. misspelled identifiers)
  - *Syntax errors*: can almost always recover (e.g. missing ';' or '{', misplaced **case**)
  - *Static semantic errors*: can sometimes recover (e.g. type mismatches, variable used before declaration)
  - *Dynamic semantic errors*: hard or impossible to detect at compile time, runtime checks are required (e.g. null pointer, division by zero, invalid array access)
  - *Logical errors*: hard or impossible to detect (e.g. `if (b = true) …`)

Viable-Prefix Property

- The *viable-prefix property* of parsers allows early detection of syntax errors
  - Enjoyed by LL(1), LR(1) parsers
  - Goal: detection of an error *as soon as possible* without further consuming unnecessary input
  - How: detect an error as soon as the prefix of the input does not match a prefix of any string in the language

Error is detected here ↓

Prefix {

```
...
for (;)
...
```

Error Recovery Strategies

- *Panic mode*
  - Discard input until a token in a set of designated "synchronizing tokens" is found (e.g. "}", ";")
- *Phrase-level recovery*
  - Perform local correction on the input to repair the error
- *Error productions*
  - Augment grammar with productions for erroneous constructs
- *Global correction*
  - Choose a minimal sequence of changes to obtain a global least-cost correction

Panic Mode Recovery

Add synchronizing actions to undefined entries based on FOLLOW

Pro:	Can be automated
Cons:	Error messages are needed

FOLLOW(E) = {) $ }
FOLLOW(E_R) = {) $ }
FOLLOW(T) = { **+**) $ }
FOLLOW(T_R) = { **+**) $ }
FOLLOW(F) = { **+** *) $ }

	id	**+**	*	(	)	$
E	$E \rightarrow T\ E_R$			$E \rightarrow T\ E_R$	***synch***	***synch***
E_R		$E_R \rightarrow \mathbf{+}\ T\ E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F\ T_R$	***synch***		$T \rightarrow F\ T_R$	***synch***	***synch***
T_R		$T_R \rightarrow \varepsilon$	$T_R \rightarrow *\ F\ T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow$ **id**	***synch***	***synch***	$F \rightarrow (\ E\ )$	***synch***	***synch***

synch: the driver pops current nonterminal A and skips input till synch token or skips input until one of FIRST(A) is found

Phrase-Level Recovery

Change input stream by inserting missing tokens
For example: **id id** is changed into **id * id**

Pro:	Can be fully automated
Cons:	Recovery not always intuitive

Can then continue here

	id	**+**	*	(	)	**$**
E	$E \rightarrow T\, E_R$			$E \rightarrow T\, E_R$	***synch***	***synch***
E_R		$E_R \rightarrow \mathbf{+}\, T\, E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F\, T_R$	***synch***		$T \rightarrow F\, T_R$	***synch***	***synch***
T_R	***insert*** *	$T_R \rightarrow \varepsilon$	$T_R \rightarrow *\, F\, T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow$ **id**	***synch***	***synch***	$F \rightarrow (\, E\,)$	***synch***	***synch***

insert *: driver inserts missing * and retries the production

Error Productions

$E \rightarrow T\ E_R$
$E_R \rightarrow \mathbf{+}\ T\ E_R \mid \varepsilon$
$T \rightarrow F\ T_R$
$T_R \rightarrow *\ F\ T_R \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \mathbf{id}$

Add "*error production*":

$T_R \rightarrow F\ T_R$

to ignore missing *, e.g.: **id id**

Pro:	Powerful recovery method
Cons:	Manual addition of productions

	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow T\ E_R$			$E \rightarrow T\ E_R$	***synch***	***synch***
E_R		$E_R \rightarrow \mathbf{+}\ T\ E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F\ T_R$	***synch***		$T \rightarrow F\ T_R$	***synch***	***synch***
T_R	$T_R \rightarrow F\ T_R$	$T_R \rightarrow \varepsilon$	$T_R \rightarrow *\ F\ T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow \mathbf{id}$	***synch***	***synch***	$F \rightarrow (\ E\ )$	***synch***	***synch***

Shift-Reduce Parsing

Grammar:

$S \rightarrow$ **a** A B **e**

$A \rightarrow A$ **b c** | **b**

$B \rightarrow$ **d**

Reducing a sentence:

a <u>b</u> b c d e

a <u>*A* b c</u> d e

a *A* **<u>d</u> e**

<u>**a** *A B* **e**</u>

S

Shift-reduce corresponds to a rightmost derivation:

$S \Rightarrow_{rm}$ **a** A B **e**

$\Rightarrow_{rm}$ **a** A **d e**

$\Rightarrow_{rm}$ **a** A **b c d e**

$\Rightarrow_{rm}$ **a b b c d e**