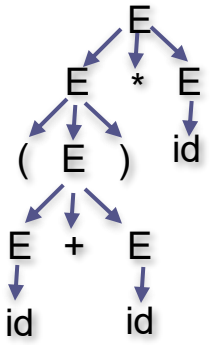
Context free languages

Syntatic parsers and parse trees



Context Free Grammars

- CF grammar production rules have the following structure

$$X \rightarrow \alpha \qquad X \in N \text{ and } \alpha \in (T \cup N)^*$$

  - They are "context free" because the replacement of X is independent on the context where it appears
    - it is always possibile to rewrite X with α

$$\beta X \gamma \overset{G}{\Rightarrow} \beta \alpha \gamma \qquad \forall\, \beta,\gamma \in (T \cup N)^*$$

  - They allow the definition of quite expressive languages as programming languages, arithmetic expressions and… regular expressions

CF Grammar– arithmetic expressions

$E \rightarrow$ "number"

$E \rightarrow (E)$

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow E * E$

$E \rightarrow E / E$

$T = \{$"number",(,),+,-,*,/$\}$

$N = \{E\}$

- The grammar defines recursively the structure of arithmetic expressions
- The terminal symbol "number" corresponds to a set of strings that can be defined by a RE
- Starting from the symbol E, the grammar can generate any correct arithmetic expression

CF Grammars- example

$S \rightarrow aB$ $A \rightarrow bAA$

$S \rightarrow bA$ $B \rightarrow b$

$A \rightarrow a$ $B \rightarrow bS$

$A \rightarrow aS$ $B \rightarrow aBB$

$T = \{a,b\}$

$N = \{S,A,B\}$

- it may be complex to describe the language defined by a grammar in a compact way
  - Given the choice of the start symbol, the language L(G) in the example defines the following sets of strings
    - $S \stackrel{*}{\Rightarrow} w$ such that w has an equal number of a and b
    - $A \stackrel{*}{\Rightarrow} w$ such that w has a number of a 1 unity greater that that of b
    - $B \stackrel{*}{\Rightarrow} w$ such that w has a number of b 1 unity greater that that of a
  - The proof is by induction

CF Grammars- example: proof

- For |w| = 1 the only derivations are
  - $A \stackrel{*}{\Rightarrow} a$ and $B \stackrel{*}{\Rightarrow} b$
- If we suppose that the hypothesis is true for |w|=k-1
  - $S \stackrel{*}{\Rightarrow} w$ can be obtained from
    - $S \rightarrow aB$ where $aB = a\, w_1$ being $|w_1| = k\text{-}1$ and $B \Rightarrow w_1$.
      By induction w_1 has 1 b more and hence w has the same number of a and b
    - $S \rightarrow bA$ where $bA = b\, w_1$ with $|w_1| = k\text{-}1$ and $A \Rightarrow w_1$.
      By induction w_1 has 1 a more and hence w has the same number of a and b
  - $A \Rightarrow w$ can be obtained from
    - $A \rightarrow aS$ where $aS = a\, w_1$ with $|w_1| = k\text{-}1$ and $S \Rightarrow w_1$.
      By induction w_1 has the same number of a and b and hence w has 1 a more
    - $A \rightarrow bAA$ where $bAA = b\, w_1 w_2$ with $|w_1| < k\text{-}1$ $|w_2| < k\text{-}1$ and $A \Rightarrow w_1$, $A \Rightarrow w_2$.
      By induction w_1, w_2 have 1 a more than b and hence there is 1 a more in w
  - $B \stackrel{*}{\Rightarrow} w$ the sketch of the proof is similar

Parse trees

- The derivation of any string in the language generated by a CF grammar can be represented by a tree structure

$E \rightarrow id$

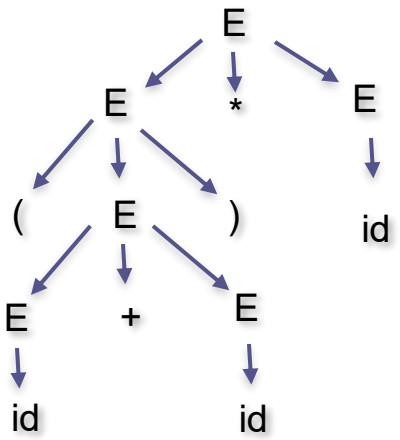
$E \rightarrow (E)$

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow E * E$

$E \rightarrow E / E$



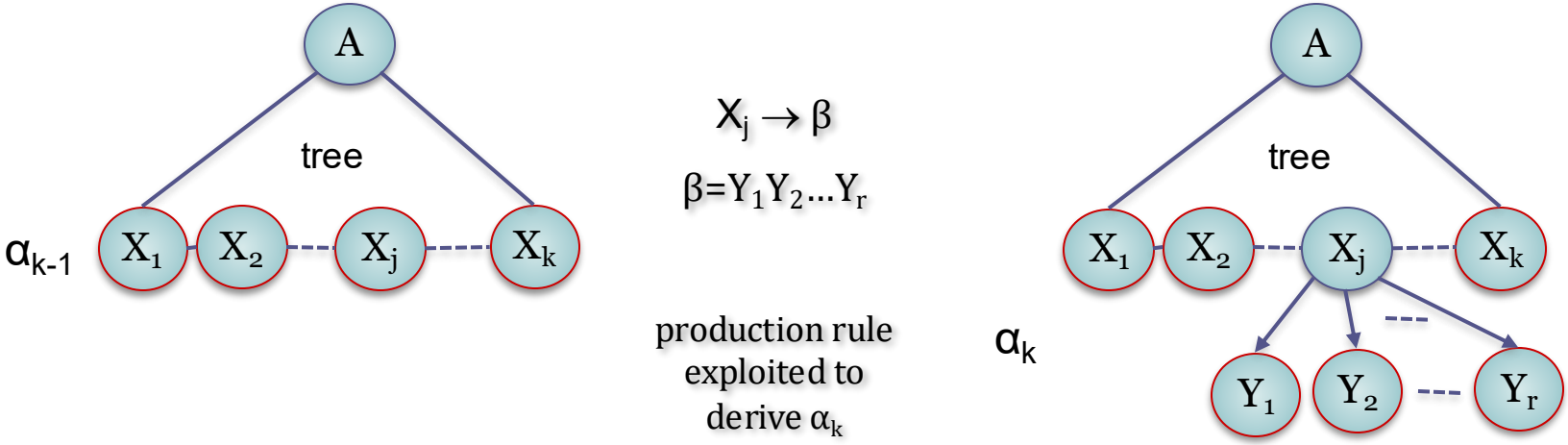
(id+id)*id

Parse tree - definition

- Nodes in a parse tree are labeled with terminal or non terminal symbols
  - Terminal symbols are in the leaves
  - Non terminal symbols are in the internal nodes
  - The root node corresponds to the non terminal start symbol
- Each internal node corresponds exactly to one production rule
  - The parent node is the non terminal symbol that is expanded
  - The child nodes correspond to the terminal/non terminal symbols in the right side of the production rule
    - children are ordered as the corresponding symbols in the production
- The parse structure is a tree since the production rules have the structure $N \rightarrow \alpha$ that characterizes CF grammars
  - The parsed string can be read by a pre-ordered traversal of the tree that outputs only the terminal symbols

Parse tree and derivations

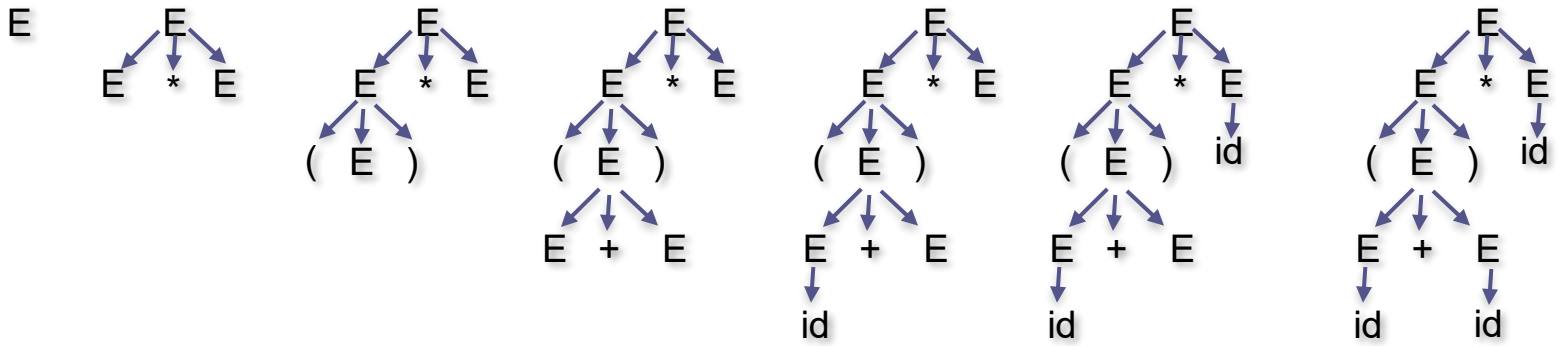
- A parse tree can be interpreted as a representation of a sequence of derivations $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots. \Rightarrow \alpha_n$ where $\alpha_1 = A \in N$
  - For each string α_i the derivation that produces α_{i+1} corresponds to the activation of a production rule and, hence, to the expansion of a sub-tree for any non terminal symbol in α_i



Parse tree - example

- The resulting parse tree shows which production rules were exploited to obtain the derivation $\alpha_1 \overset{*}{\Rightarrow} \alpha_n$ but not the order in which they are activated

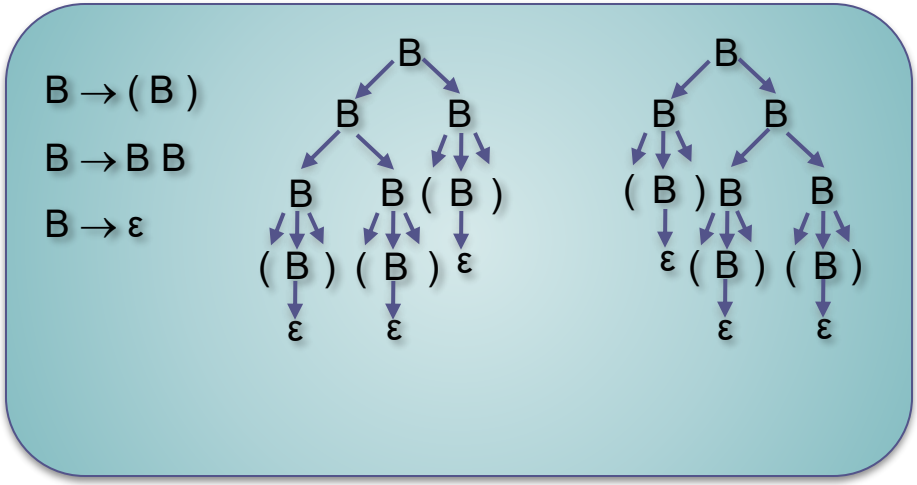
$$E \Rightarrow E * E \Rightarrow (E) * E \Rightarrow (E + E) * E \Rightarrow (id + E) * E \Rightarrow (id + E)* id \Rightarrow (id + id)*id$$



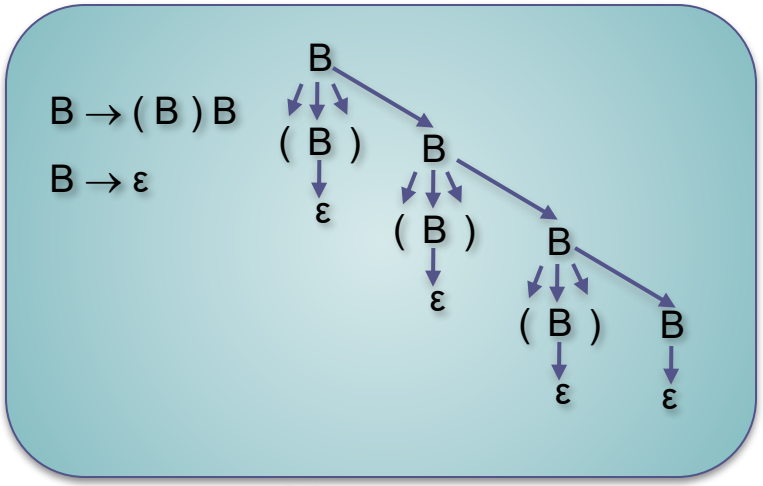
Ambiguous grammars

- A grammar is ambiguous if it is possible to build more than one parse tree to describe the derivation for a same string

generation of ()()()



ambiguous grammar



non ambiguous grammar

Ambiguous Grammars – problems

- In general, it is complex to prove if a given grammar is ambiguous
- The ambiguity can cause problems when the parse tree is used to give a semantic interpretation to the input string
  - more interpretations are possible

$E \rightarrow id$

$E \rightarrow (E)$

$E \rightarrow E + E$

$E \rightarrow E * E$

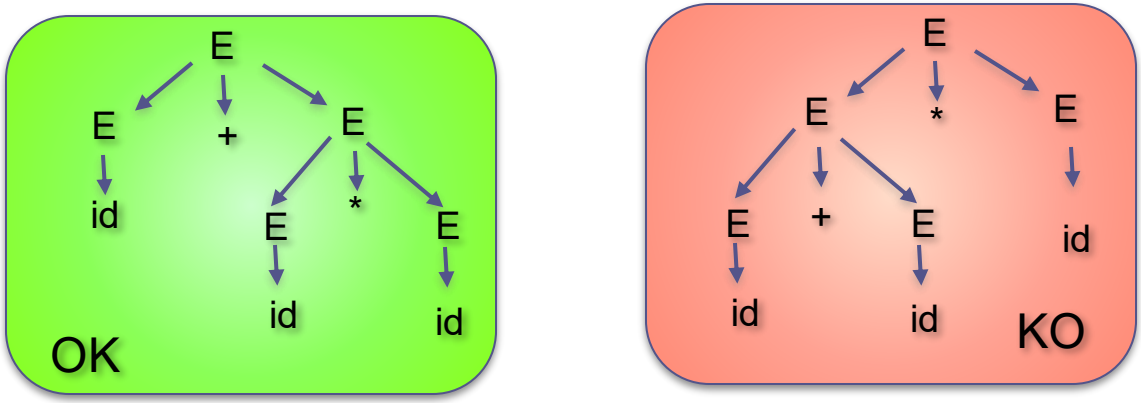
$E \rightarrow - E$

$E \Rightarrow \boxed{E + E} \Rightarrow E + E * E \Rightarrow id + E * E \Rightarrow id + id * E \Rightarrow id + id * id$

id+id*id

$E \Rightarrow \boxed{E * E} \Rightarrow E + E * E \Rightarrow id + E * E \Rightarrow id + id * E \Rightarrow id + id * id$

Ambiguous Grammars – expressions 1



- The grammar does not model the operator precedence
  - When using the parse tree on the right the evaluation of the expression would generate an incorrect result (the sum is evaluated first)
  - The problem can be solved with a more precise model
    - The grammar should use a different model for the two operators * and +
    - More syntactic categories (non terminal symbols) are exploited to yield the correct grouping of the expression parts (terms and factors)

Ambiguous Grammars – expressions 2

- Three syntactic categories are used
  - F – factor: a single operand or an expression between ()
  - T – term: a product/quotient of factors
  - E – expression: a sum/subtraction of terms

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

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow (E) \mid id$$

- Production rules such as E→ E + T cause a grouping of the terms from left to right (f.i. 1+2+3 → (1+2)+3)

Ambiguous Grammars - expressions

id+id*id

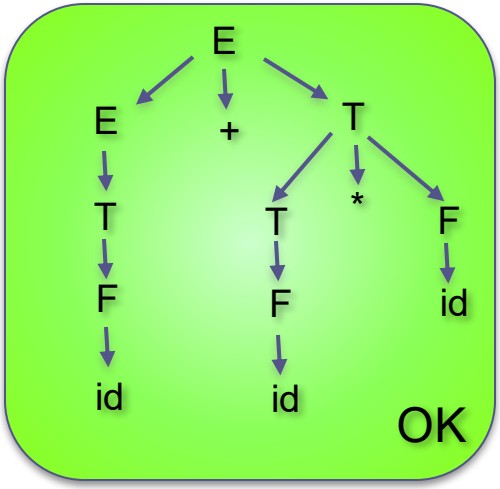
derivation with left rewriting

$$E \Rightarrow \boxed{E} + T \Rightarrow \boxed{T} + T \Rightarrow \boxed{F} + T \Rightarrow id + \boxed{T} \Rightarrow id + \boxed{T} * F \Rightarrow id + \boxed{F} * F \Rightarrow id + id * \boxed{F} \Rightarrow id + id * id$$

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

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid id$



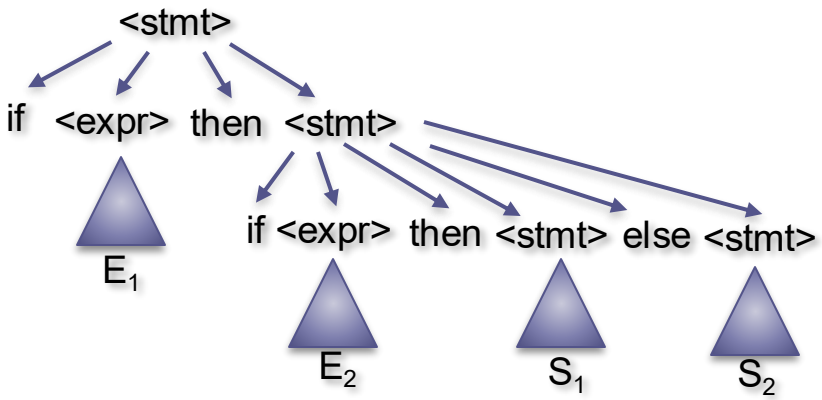
Ambiguity – if… then… else

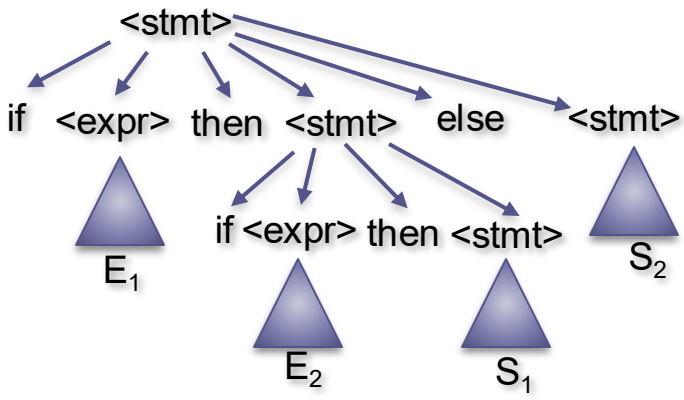
<stmt> → if <expr> then <stmt>

<stmt> → if <expr> then <stmt> else <stmt>

<stmt> → "instruction" ….

- The grammar is ambiguous because the string "if E_1 then if E_2 then S_1 else S_2" has two valid parse trees





Ambiguity – non ambiguous "if.. then.. else"

- The ambiguity is due to the fact that the grammar does not allow a clear association between the "else" and the "if"
  - The most used rule is that the "else" is attached to the closest "if"

<stmt> → <stmt_c> | <stmt_u>

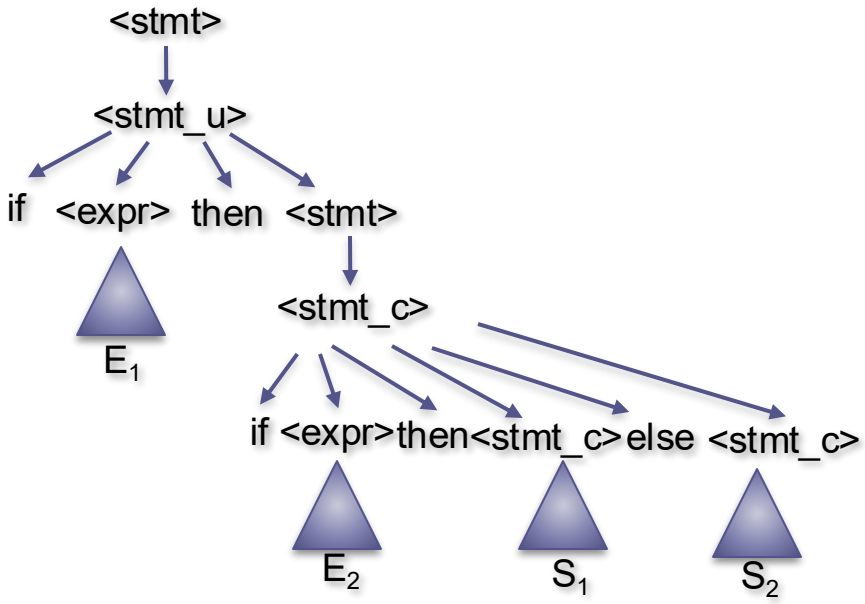
<stmt_c> → if <expr> then <stmt_c> else <stmt_c>

<stmt_c> → "instruction" ….

<stmt_u>→ if <expr> then <stmt>

<stmt_u>→ if <expr> then <stmt_c> else <stmt_u>

between "then-else" we can find only complete "if-then-else" expressions



Equivalent productions

- Some productions can be rewritten to obtain an equivalent grammar (generating the same language) whose production rules follow specific patterns
  - Removal of left recursion
    A grammar is left recursive if there exists a non terminal symbol A for which there exists a derivation $A \stackrel{+}{\Rightarrow} A\alpha$ being $\alpha \in (T \cup N)^*$

Simple production rule

$A \rightarrow A\,\alpha$

$A \rightarrow \beta$

$A \rightarrow \beta\,A'$

$A' \rightarrow \alpha\,A' \mid \varepsilon$

It generates the strings

$A \rightarrow \beta\,\alpha^n$

Left recursion

- Left recursion in production rules can be easily removed even in the most general case

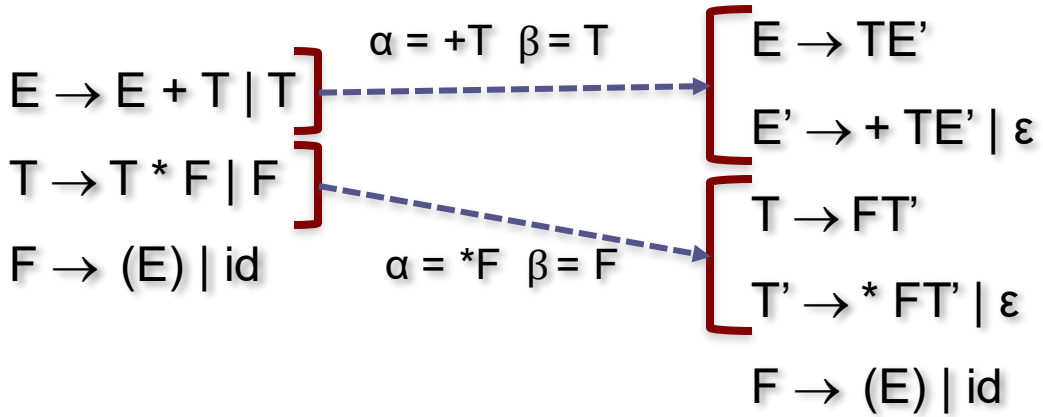
$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m$$
$$A \rightarrow \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_n$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots. \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots. \mid \alpha_m A' \mid \varepsilon$$

- There exists an algorithm to remove the left recursion for derivations in one ore more steps
  - removal of left recursion simplifies the implementation of left-to-right parsers (parser that read the input string from left to right)
  - right recursion implements an expansion of strings from left to right

Left recursion- example

- Removal of the left recursion in the grammar for arithmetic expressions



Left factorization

- Left factorization can be used to rewrite the production rules obtaining an equivalent grammar

$$A \rightarrow \alpha\,\beta_1 \mid \alpha\,\beta_2 \mid \ldots \mid \alpha\,\beta_n$$

$$A \rightarrow \alpha\,A'$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

- It is assumed that $\beta_1, \beta_2, \ldots, \beta_n$ do not share a same prefix
- When considering a left-to-right parsing, left factorization allows us to decide the expansion of α postponing the choice of the expansion of one out of $\beta_1, \beta_2, \ldots, \beta_n$ to the next step

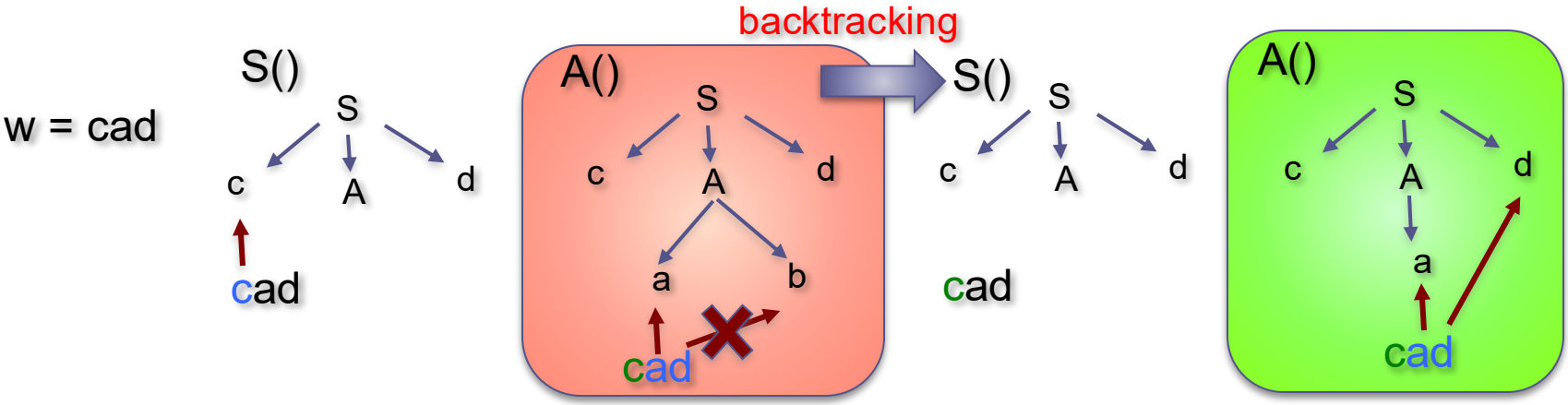
Top-down parsing

- Top-down parsing is a processing scheme that aims at building the parse tree starting form the root, adding nodes in pre-order
  - In general, there is the need of backtracking
    - when there are multiple choices for the production rule to be expanded, the first one is tried, and the others are tried if this choice leads to a failure
  - The grammar defines a set of mutually recursive functions, each corresponding to one of the syntactic categories (non terminal symbols)
  - The call of one function corresponds to the expansion of a given production rule (expansion of the corresponding non terminal symbol)
  - The function for the start symbol S reads an input strings and returns the pointer to the root node of the generated parse tree
    - a null pointer if the input string does not belong to the language (a parse error is generated)

Top-down parsing - example

$S \rightarrow cAd$ S()

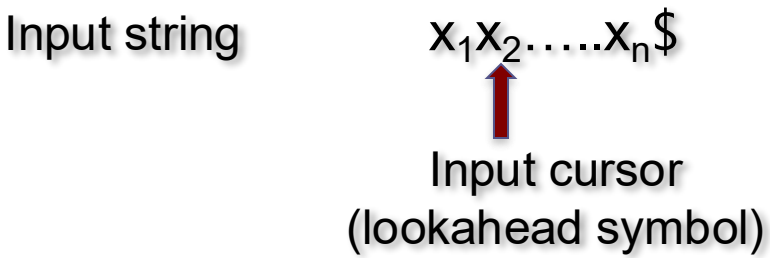
$A \rightarrow ab \mid a$ A()



- A left recursive grammar can produce an infinite number of expansions in a recursive top-down parser (the same symbol is expanded at each step)

Top-down parsing – parsing process 1

- A cursor is used to track the next terminal symbol in the input string that is to be generated in the parse tree
  - This terminal symbol allow us to restrict the set of production rules that can be selected to expand a non terminal symbol in the partial tree
- Tree nodes are expanded from left to right
  - a terminal symbol satisfies the goal if it matches the next symbol in the input string
  - a non terminal symbol satisfies the goal if it is satisfied by the call of the corresponding recursive function



Top-down parsing – parsing process 2

- When a production rule is expanded (the corresponding recursive function is called) and a terminal symbol is generated
  - the operation is successful if the generated symbol matches the terminal symbol pointed by the input cursor
    - Yes – the cursor moves to the next position and the parsing continues
    - No – the current solution fails and the next hypothesis for the previous goal is generated (backtracking)

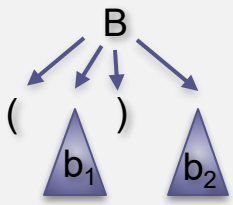
- If the expansion adds a non terminal symbol T, the corresponding recursive function is called
  - the function generates the parse sub-tree rooted at the node T

Top-down parsing- example

input cursor input string

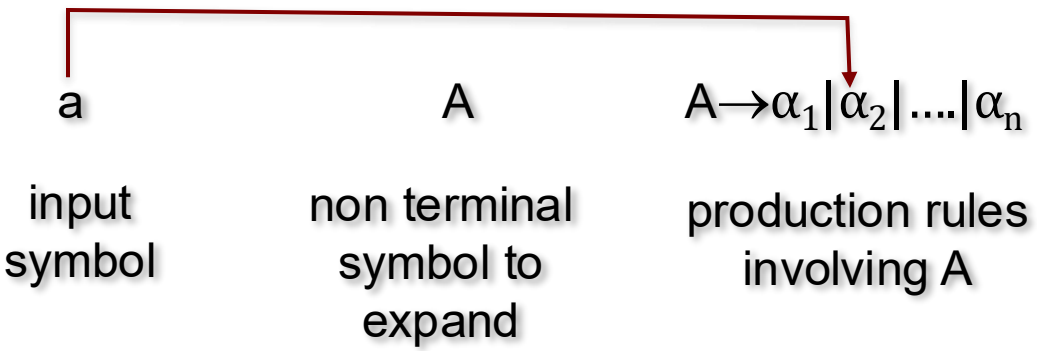
```
def B(cursor,input):
B -> (B)B | e"""Function to expand non terminal B

    """
    if cursor<len(input) and input[cursor]=='(':
        cursor=cursor+1
        cursor, B1 = B(cursor,input)
        if input[cursor]==')' and B1 is not None:
            cursor = cursor+1
            cursor, B2 = B(cursor,input)
            if B2 is None:
                return cursor, None
            else:
                return cursor, tree.Node('B',
                                [tree.Node('(',[]),B1,tree.Node(')',[]),B2])
        else:
            return cursor, None
    else:
        return cursor, tree.Node('B',[tree.Node('e',[])])
```

$B \rightarrow (B) B \mid \varepsilon$



Lookahead parsers

- Left recursion is removed
- Production rules are left factorized
  - a lookahead parser can be defined such that the parsing procedure does not require backtracking for a subset of CF grammars (not all)
    - The lookahead terminal symbol always allows the selection of only one production rule to be expanded



Lookahead parsing

- State diagrams represent the sequences in the right side of production rules
  - they can be exploited to determine the production rule to be applied
  - state transitions are triggered by a terminal symbol (a symbol is read from the input and the cursor move ahead) or by a non terminal symbol (the corresponding expansion is activated)

$E \rightarrow TE'$

$E' \rightarrow + TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow * FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

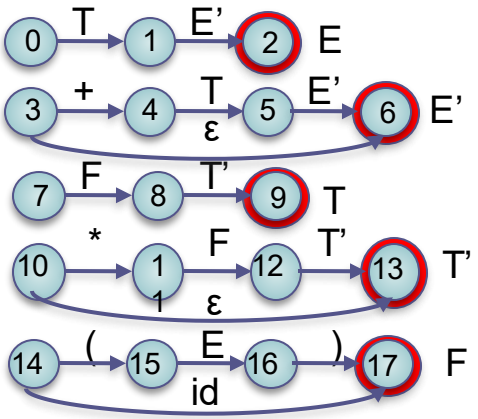


Table based parsing

- A stack is directly used to implement the recursive calls

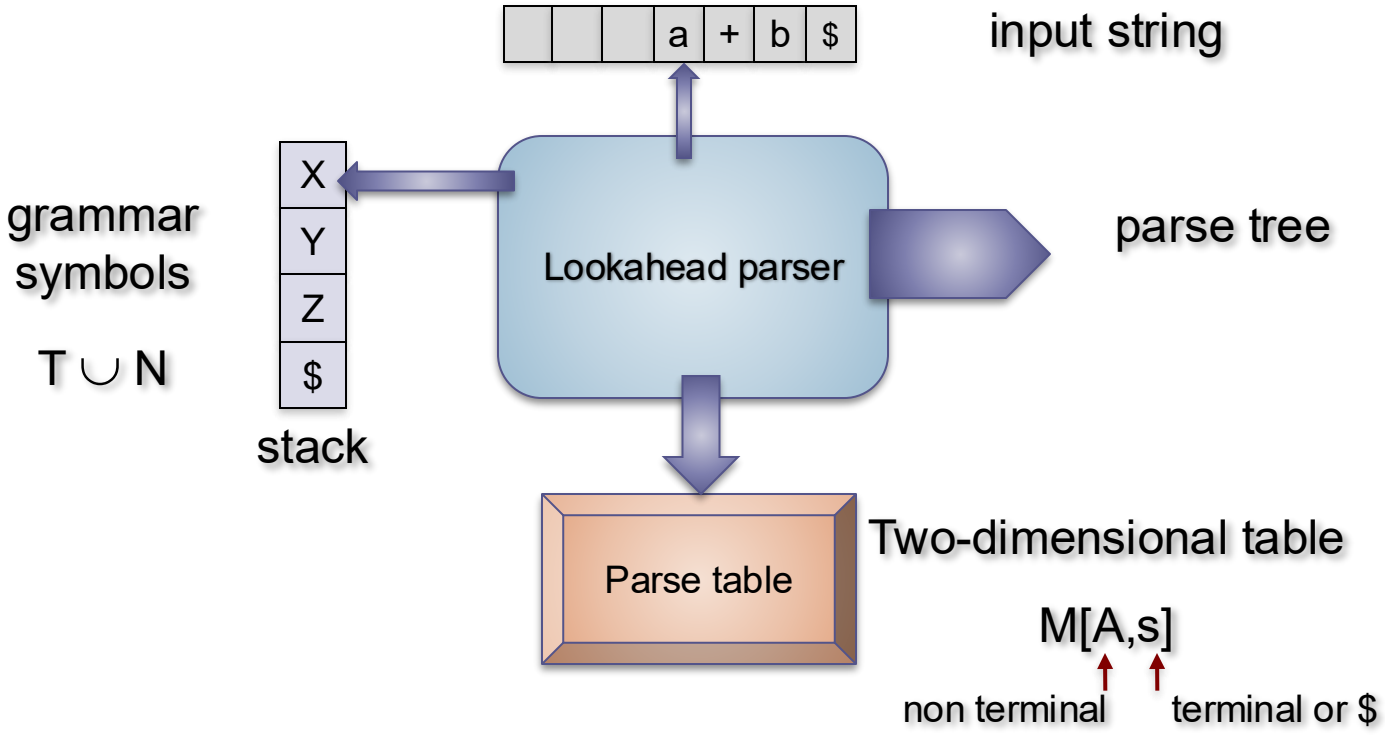


Table based parsing - procedure

- Initially the stack contains the start symbol S
- The control selects the action to be executed using the symbol at the top of the stack (X) and the current input terminal symbol (a)
  - If X=a=$ the parser halts with success
  - If X=a≠$ the parser pops X from the stack and moves ahead the input cursor by 1 position (lookahead symbol match)
  - X is a non terminal symbol – the entry M[X,a] is checked
    - If it corresponds to a production rule, the elements in its right side are pushed into the stack

      $X \rightarrow UVW$ push(W); push(V); push(U)
    - Otherwise, a parse error is issued and the parser halts
  - If X≠a and X is a terminal symbol, then a parse error is generated

Table based parsing- example

$E \rightarrow TE'$

$E' \rightarrow + TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow * FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Grammar

	id	+	*	(	)	$
E	TE'			TE'		
E'		+TE'			ε	ε
T	FT'			FT'		
T'		ε	*FT'		ε	ε
F	id			(E)		

Parse table

stack	input	output
$E	id+id*id$	
$E'T	id+id*id$	$E \rightarrow TE'$
$E'T'F	id+id*id$	$T \rightarrow FT'$
$E'T'id	id+id*id$	$F \rightarrow id$
$E'T'	+id*id$	
$E'	+id*id$	$T' \rightarrow \varepsilon$
$E'T+	+id*id$	$E' \rightarrow +TE'$
$E'T	id*id$	
$E'T'F	id*id$	$T \rightarrow FT'$
$E'T'id	id*id$	$F \rightarrow id$
$E'T'	*id$	
$E'T'F*	*id$	$T' \rightarrow *FT'$
$E'T'F	id$	
$E'T'id	id$	$F \rightarrow id$
$E'T'	$	
$E'	$	$T' \rightarrow \varepsilon$
$	$	$E' \rightarrow \varepsilon$

Parse trace

Table based parsing– table generation 1

- We consider the following two functions
  - FIRST(α) is the set of terminal symbols that can start a string generated from $\alpha \in (T \cup N)^*$
  - FOLLOW(A) is the set of thterminal symbols that can appear at the right just after the symbol $A \in N$ in a string derived from S
    - there exists a derivation $S \Rightarrow \alpha A a \beta$ being $a \in T$
- Computation of FIRST(x) $x \in T \cup N$
  - if $x \in T$ then $FIRST(x) = \{x\}$
  - if $x \in N$ and there is the production rule $x \rightarrow \varepsilon$ then $\varepsilon \in FIRST(x)$
  - if $x \in N$ and there is the production rule $x \rightarrow Y_1 Y_2 \ldots Y_k$ then
    - $a \in FIRST(x)$ if $a \in FIRST(Y_i)$ and $\varepsilon \in FIRST(Y_j)$ $j=1,..,i-1$ ($Y_1 \ldots Y_{i-1} \Rightarrow \varepsilon$)
    - $\varepsilon \in FIRST(x)$ if $\varepsilon \in FIRST(Y_j)$ $j=1,..,k$

    Basically the elements of $FIRST(Y_i)$ are added to FIRST(x) until a symbol Y_i is found such that $\varepsilon \notin FIRST(Y_i)$

Table based parsing- table generation 2

- Computation of FIRST(α) $\alpha \in (T \cup N)^*$ with $\alpha = Y_1 Y_2 \ldots Y_k$
  - $F = FIRST(Y_1)$
  - for(i=1; i<k && $\varepsilon \notin FIRST(Y_i)$;i++)
    - $F = F \cup FIRST(Y_{i+1})$
- Computation of FOLLOW(A)
  - $\$ \in FOLLOW(S)$
  - If there is the production rule $B \rightarrow \alpha A \beta$ all the symbols in FIRST(β) except ε are in FOLLOW(A)
  - If there is the production rule $B \rightarrow \alpha A$ or $B \rightarrow \alpha A \beta$ and FIRST(β) contains ε then all the symbols in FOLLOW(A) are also in FOLLOW(B)

Table based parsing– example FIRST/FOLLOW

$E \rightarrow TE'$

$E' \rightarrow + TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow * FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

FIRST(E) = {(,id}

FIRST(T) = {(,id}

FIRST(F) = {(,id}

FIRST(E') = {+,ε}

FIRST(T') = {*,ε}

FOLLOW(E) = {),$}

FOLLOW(T) = {),+,$}

FOLLOW(F) = {),*,+,$}

FOLLOW(E') = {),$}

FOLLOW(T') = {),+,$}

Table based parsing- table generation

- Definition of the entries in the parse table
  - If there is the production rule $A \rightarrow \alpha$ then for any symbol a in FIRST(α) M[A,a] = {$A \rightarrow \alpha$}
    - In fact, if the parser is in the state A and a is read from the input, the production rule $A \rightarrow \alpha$ is to be expanded since it guarantees the generation of the terminal symbol a
  - If $\varepsilon \in$ FIRST(α) then M[A,b] = {$A \rightarrow \alpha$} for any symbol b in FOLLOW(A)
    - It implements the fact that if $\alpha \Rightarrow \varepsilon$ then the symbol a must be generated by some production rule where A appears followed by another expression that can generate the terminal symbol a

Table based parsing– ambiguous grammars

- The procedure for the generation of the parse table can produce entries of the matrix M that contain more than one alternative
  - the grammar cannot be parsed without backtracking
    - f.i. ambiguous grammars

$S \rightarrow i\ E\ t\ S\ S' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

grammar for

if-then-else

FIRST(S) = {i,a}

FIRST(S') = {e, ε}

FIRST(E) = {b}

FOLLOW(S) = {e,$}

FOLLOW(S') = {e,$}

FOLLOW(E) = {t}

	a	b	e	i	t	$
S	a			iEtSS'		
S'			ε eS			ε
E		b				

requires to check both options (backtrcking)

LL(1) grammars

- A grammar that does not have multiple entries in the parse table is an LL(1) grammar
  - Left-to-right in the input scanning
  - Leftmost – the leftmost symbol is always expanded
  - 1 lookahead symbol is exploited to select the "correct" action

- The LL(1) grammars are a subset of the CF grammars
  - left recursion is to be removed
  - grammars must be left factorized
  - Not all CF grammars are LL(1)
    - for instance, ambiguous grammars are not LL(1)

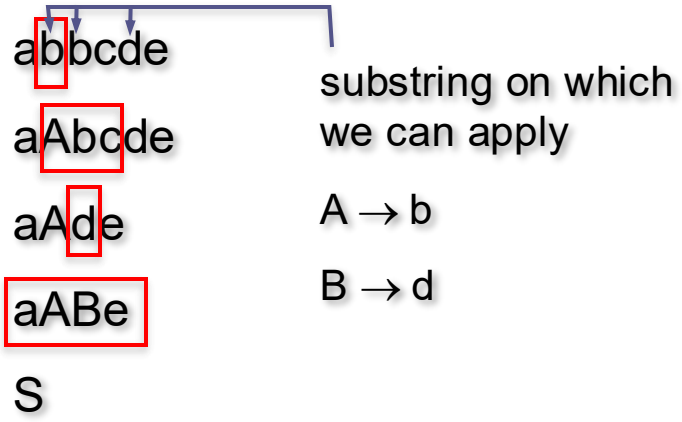
Bottom-up parsing

- The parse tree is generated starting from the leaves up to the root
  - The input string is reduced to the start symbol S
  - At each step a substring that matches the right side of a production rule is replaced by the non terminal symbol in the left side
  - The corresponding node in the parse tree is generated by connecting the child nodes to their parent node

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$



Bottom-up parsing– selecting reductions

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

$S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$

- In the example, the reductions are selected considering a derivation of the string in which the rightmost non terminal symbol is rewritten
  - The bottom-up approach reduces the string from left to right
- How can we select the string to be reduced?
  - We can select the leftmost substring that matches the right side of a production rule
  - It is not guaranteed that the whole string is reduced to the start symbol S for a given selection (*backtracking may be needed in general*)

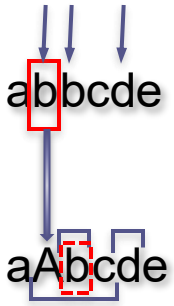
Bottom-up parsing– selecting reductions

- In the example, the choice of reducing the leftmost string at the first step leads to a successful parsing
  - In general, this may not happen….
  - If we make the wrong option, we find an intermediate result in which there are no substrings that match the right side of one production rule
    - in this case backtracking is required

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$



if at the second step we select $A \rightarrow b$

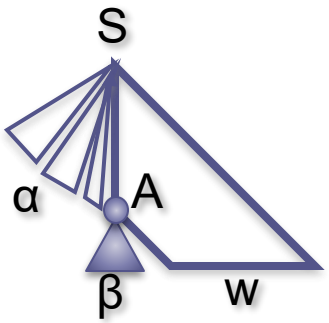
aAAcde

aAAcBe ✗

is no more reducible….

Handle substrings

- A handle substring
  - right side β of a production rule $A \rightarrow \beta$
  - A can be replaced in the current string γ obtaining a step in the right derivation of γ from S $S \overset{*}{\Rightarrow} \alpha A w \overset{rm}{\Rightarrow} \alpha \beta w$
    - w contains only terminal symbols since the derivation is rightmost
  - If the grammar is ambiguous, the same substring may belong to more than one handle



handle

the reduction of β to A corresponds to the removal of all the children of node A from the tree

Reduction process

- The reduction process is aimed at the progressive substitution of the handle substrings with the corresponding non terminal symbol
  - Starting from the input string containing only terminal symbols

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots\cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

  - The handle β_n is substituted in γ_n exploiting the production rule $A_n \rightarrow \beta_n$ such that $\gamma_{n-1} = \alpha_{n-1} A w_{n-1}$
  - the process is repeated until the start symbol S is obtained
- This process has two correlated tasks to be solved
  - How to detect the substring to be reduced
  - How to select the correct production rule for the reduction

Bottom-up parsers with stack

- intermediate results are stored in a stack (the tree frontier)
  - the parser pushes symbols into the stack from the input string w until a handle β is found at the top of the stack
  - the parser reduces the handle β to the non terminal A ($A \rightarrow \beta$)
    - it pops the handle β from the stack
    - it pushes A into the stack
  - the parser halts with success when the stack contains only the symbol S and the input string is empty
- Actions executed by the parser
  - SHIFT – the next symbol in w is pushed into the stack
  - REDUCE – a handle substring is matched at the top of the stack and it is replaced by the corresponding non terminal symbol
  - ACCEPT – successful halt of the parser
  - ERROR – the parser outputs a syntax error

Bottom-up parsing- example

$E \rightarrow id$

$E \rightarrow (E)$

$E \rightarrow E + E$

$E \rightarrow E * E$

id+id*id

stack	input	output
$	id+id*i$	SHIFT
$id	+id*id$	REDUCE E → id
$E	+id*id$	SHIFT
$E+	id*id$	SHIFT
$E+id	*id$	REDUCE E → id
$E+E	*id$	SHIFT (*)
$E+E*	id$	SHIFT
$E+E*id	$	REDUCE E → id
$E+E*E	$	REDUCE E → E*E
$E+E	$	REDUCE E → E+E
$E	$	**ACCEPT**

- The grammar is ambiguous and there is another valid reduction
  - There is a SHIFT/REDUCE conflict in (*) that was resolved with SHIFT
    - REDUCE E → E+E could have been selected, as well

Bottom-up parsers - conflicts

- Stack-based bottom-up parsing not requiring backtracking cannot be realized for any CF grammar
  - Given the stack contents and the next input symbol, it fails when
    - There is a SHIFT/REDUCE conflict
    - It is not possible to select the correct reduction in a set of valid reductions
      - REDUCE/REDUCE conflict
- The actions in a stack-based bottom-up parser can be univocally determined only if the grammar has specific properties
  - Parser for operator-precedence grammars
    - peculiar subset of grammars where the production rules such as $A \rightarrow \varepsilon$ are not allowed and for any production rule $A \rightarrow \beta$ the string β does not contain two adjacent non terminal symbols (they are always separated by an "operator"). F.i. the arithmetic expressions
  - LR parsers (Left-to-right Right-most-derivation)

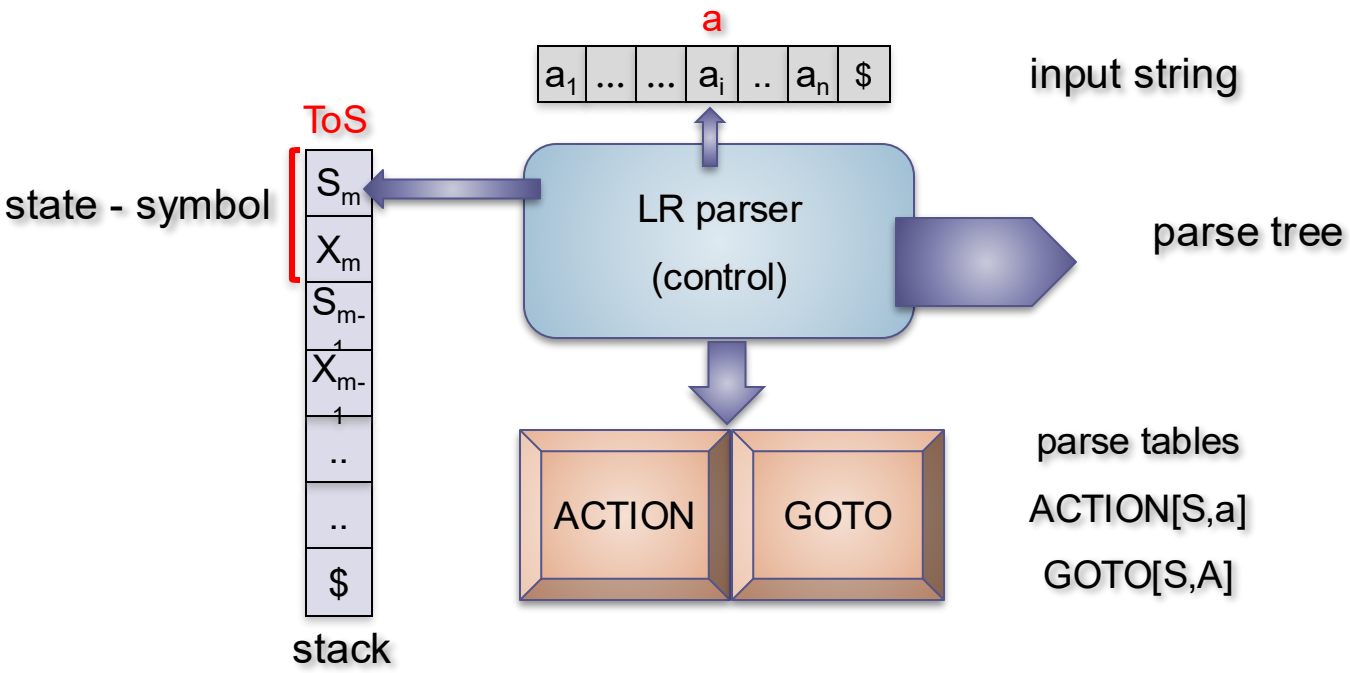
LR(k) parsers

- Left-to-right parsing and selection of the derivation that always expands the rightmost non terminal symbol (Right-most-derivation)
- k lookahead symbols are exploited to select the action
  - LR is for LR(1)

- ADVANTAGES
  - LR parsers can recognize the statements of programming languages
  - Most general SHIFT/REDUCE method not requiring backtracking
  - The class of LR grammars properly contains all the grammars that can be parsed by a lookahead parser
  - A LR parser can report and error as soon as it appears in a left-to-right scanning of the input string

LR(k) parsers- structure

- DISADVANTAGES
  - "handcrafting" a LR parser is difficult but it can be generated procedurally



LR parsers – processing scheme 1

- The stack stores a string of symbol-state pairs

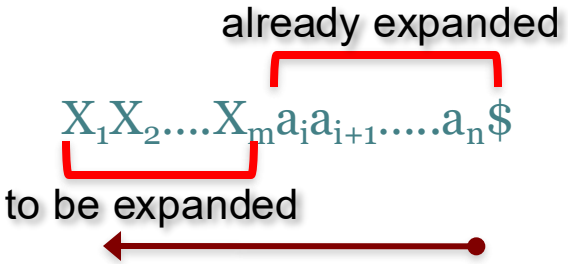
$$S_0X_1S_1X_2S_2....X_mS_m$$

  - each state summarizes the information contained in the stack required to recognize a handle substring
  - the parser action is determined by the state at the top of the stack and the current input symbol (ToS,a)
  - in the implementation the language symbols $X_i \in (T \cup N)$ are not strictly needed
    - the state already stores the partial processing of their sequence
  - The parser configuration is given by the stack contents and the remaining part of the input string

$$S_0X_1S_1X_2S_2....X_mS_ma_ia_{i+1}.....a_n\$$$

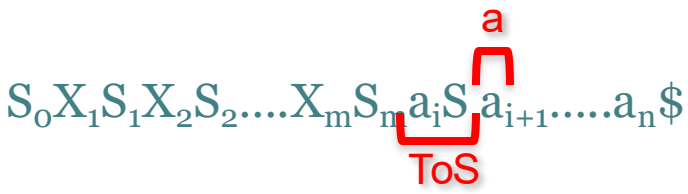
LR parsers – processing scheme 2

- The configuration represents a right-derived substring



  - The control of the LR parser selects the action to be performed given the state S_m at the top of the stack and the current input symbol a_i

1. ACTION[S_m,a_i]=SHIFT S (PUSH a_i, PUSH S)
   The new configuration is



LR parsers – processing scheme 3

2. ACTION[S_m,a_i]=REDUCE $A \rightarrow \beta$
   A reduction is applied causing the new configuration

$$S_0X_1S_1X_2S_2....X_{m-r}S_{m-r}AS\,a_i.....a_n\$$$

a

ToS

   where $S = GOTO[S_{m-r},A]$ and $r=|\beta|$ $\beta=X_{m-r+1}....X_m$

3. ACTION[S_m,a_i]=ACCEPT
   The parsing is halted with success

4. ACTION[S_m,a_i]=ERROR
   An error is detected and the error handling procedure is executed

LR parsing – example 1

1 E → E+T

2 E → T

3 T → T*F

4 T → F

5 F → (E)

6 F → id

s# = shift #new state

r# = reduce #production

	ACTION						GOTO		
state	**id**	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	s5			s4			1	2	3
1		s6				Ac			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

LR parsing- example 2

stack	input	ACTION	reduction	GOTO
0|	id*id+id$	SHIFT 5		
0|id|5|	*id+id$	REDUCE 6	F→id	G[0,F]=3
0|F|3|	*id+id$	REDUCE 4	T→F	G[0,T]=2
0|T|2|	*id+id$	SHIFT 7		
0|T|2|*|7|	id+id$	SHIFT 5		
0|T|2|*|7|id|5|	+id$	REDUCE 6	F→id	G[7,F]=10
0|T|2|*|7|F|10|	+id$	REDUCE 3	T→T*F	G[0,T]=2
0|T|2|	+id$	REDUCE 2	E→T	G[0,E]=1
0|E|1|	+id$	SHIFT 6		
0|E|1|+|6|	id$	SHIFT 5		
0|E|1|+|6|id|5|	$	REDUCE 6	F→id	G[6,F]=3
0|E|1|+|6|F|3|	$	REDUCE 4	T→F	G[6,T]=9
0|E|1|+|6|T|9|	$	REDUCE 1	E→E+T	G[0,E]=1
0|E|1|	$	**ACCEPT**		

LR grammars- definition

- LR grammar (definition)
  - grammar for which it is possible to univocally fill the ACTION and GOTO tables for an LR parser
  - There are CF grammars that are not LR
  - A grammar is LR if the SHIFT/REDUCE parser is able to recognize a handle substring when it appears at the top of the stack
    - only the state is needed to perform this check
    - the handle recognizer can be implemented by a finite state automaton that scans the symbols in the stack and outputs the correct right side of the production rule as soon as it is detected
    - this mechanism is realized by the GOTO table
    - the state at the top of the stack is the current state of this automaton after the processing of the symbols from the bottom to the top of the stack

LR grammars - properties

- LR(k) grammars are more general than LL(k) grammars
  - LR(k) grammars require to recognize the right side of a production rule $A \rightarrow \beta$ (the handle) given k lookahead symbols after having seen all the symbols that derive from β
  - LL(k) grammars require to recognize a production rule given the first k symbols of what can derive from its right side

- What about filling the parsing tables?
  - We consider the case of Simple LR (SLR) grammars that is a proper subset of LR grammars

SLR grammars

- An LR(0) element of a grammar G is a production rule tagged with a dot (•) in a given position in the right side

#i $A \rightarrow xyz$

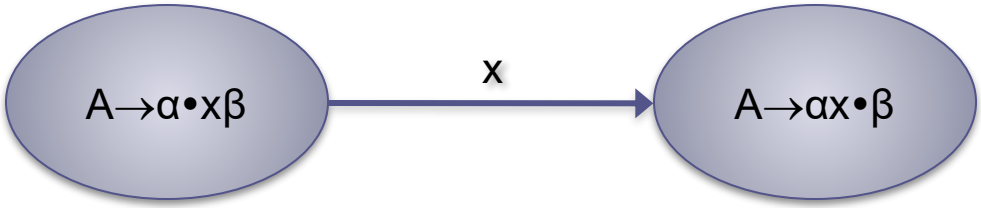
- $A \rightarrow \bullet xyz$ (i,0)
- $A \rightarrow x \bullet yz$ (i,1)
- $A \rightarrow xy \bullet z$ (i,2)
- $A \rightarrow xyz \bullet$ (i,3)

  - An element is defined by a pair of indexes (#production, dot position)
  - An element keeps track of how many symbols in the right side have been already found up to a given step of the parsing procedure

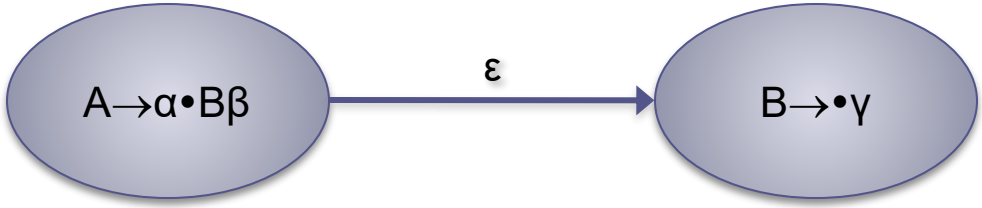
- The filling of the parse table begins with the construction of a finite state automaton that recognizes the prefixes associated to the production rules in a right derivation process

SLR grammars– elements

- the elements defined by each production rule can be seen as the states of a finite state automaton



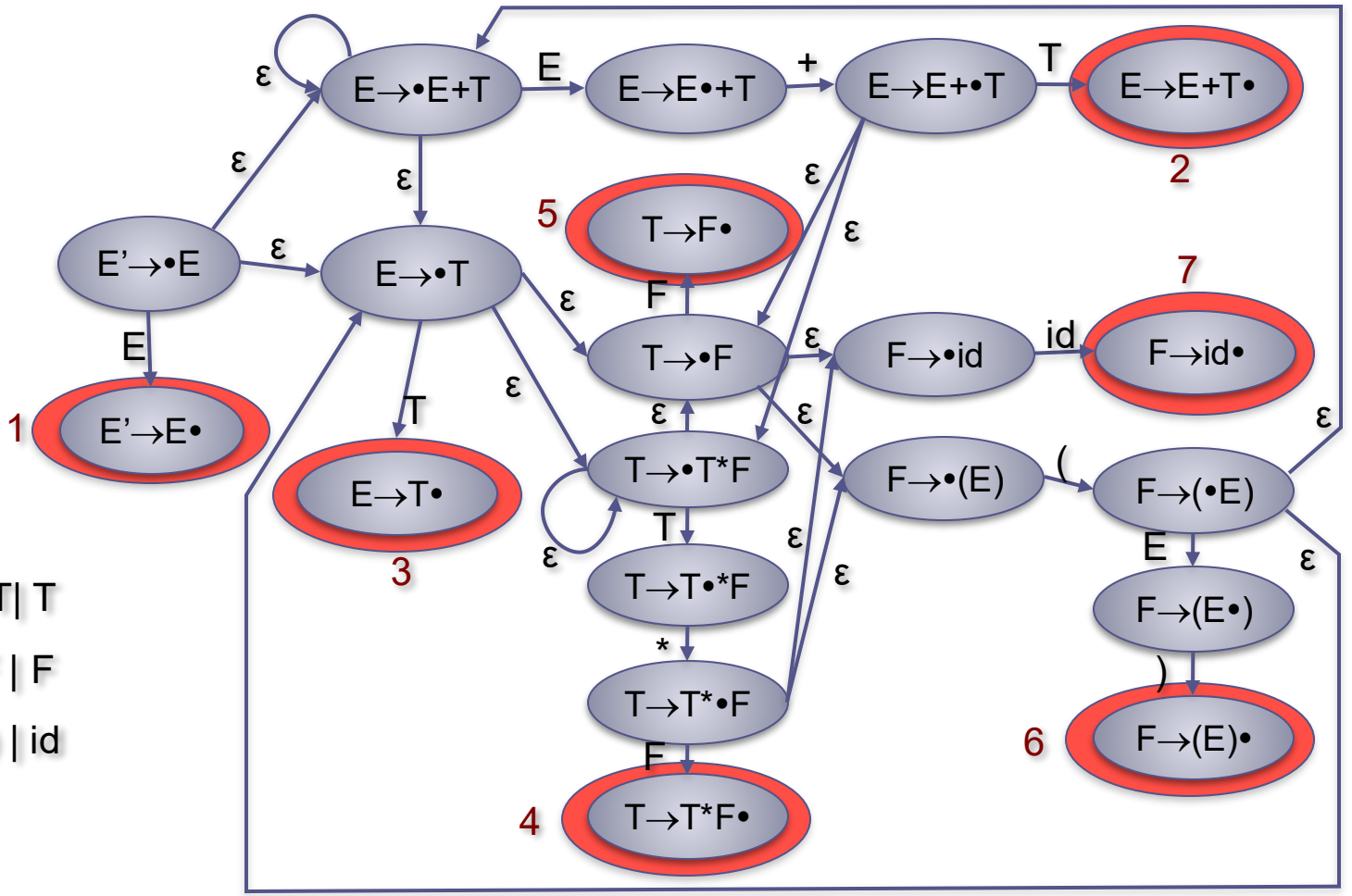
the input x is accepted to move one step forward in the recognition of αxβ



the detection of B requires to apply any possible expansion of this non terminal symbol

  - A new start symbol S' is added with the production rule S' →S
    - it is the reduction that causes the acceptance of the input string
  - the FSA recognizing the right sides of the production rules is built

SLR grammars- example 1



1 E' → E

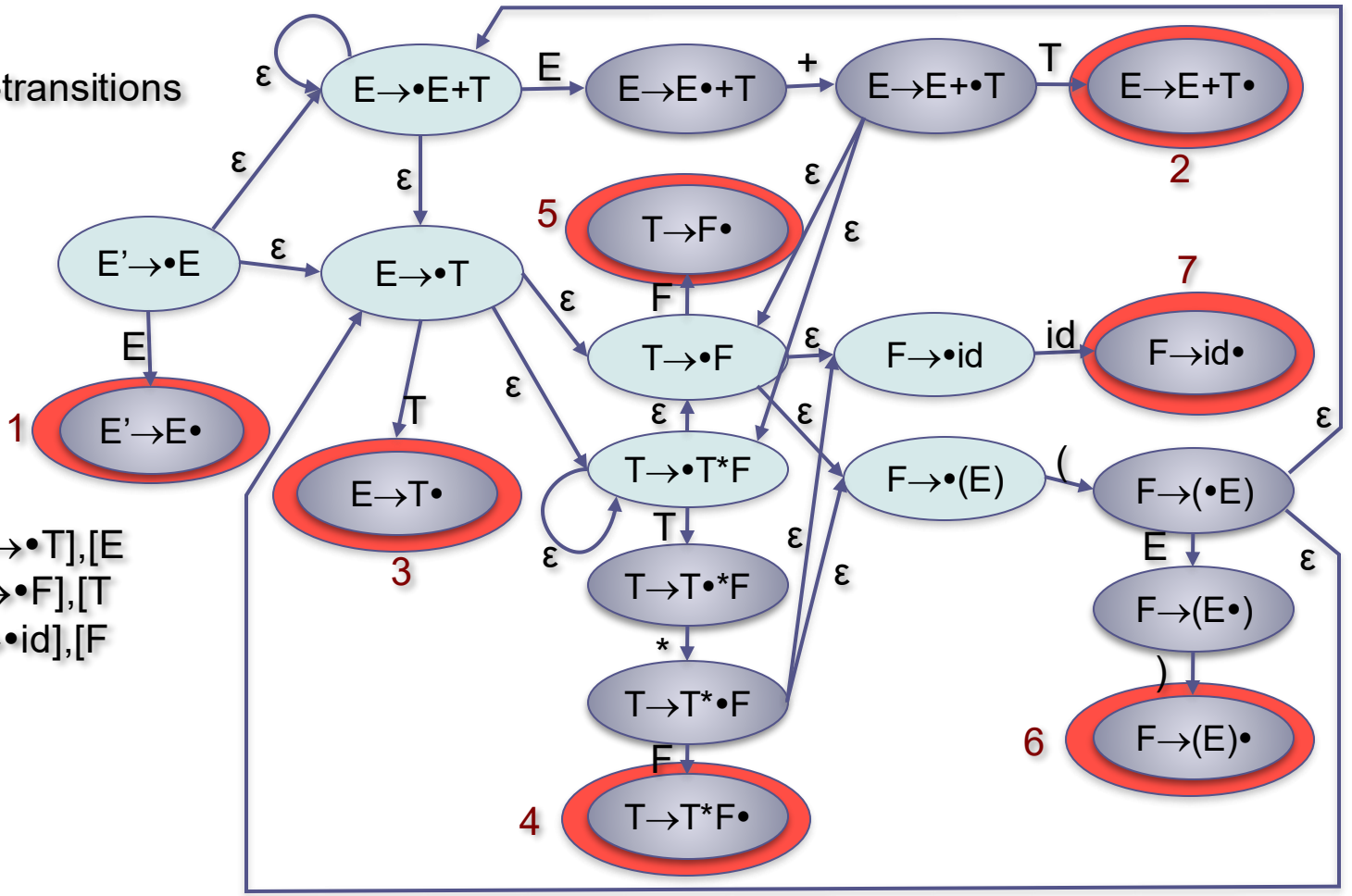
2-3 E → E+T| T

4-5 T → T*F | F

6-7 F → (E) | id

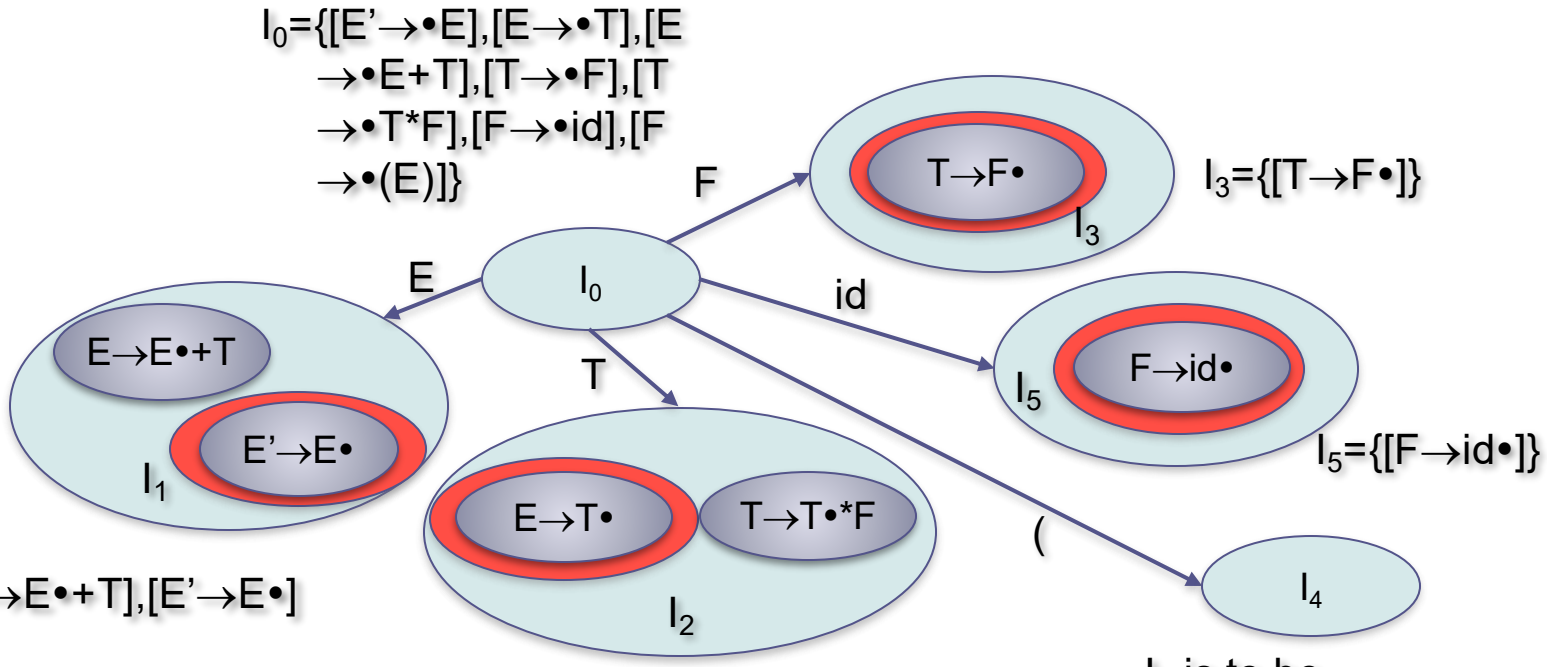
SLR grammars- example 2

removal of ε-transitions



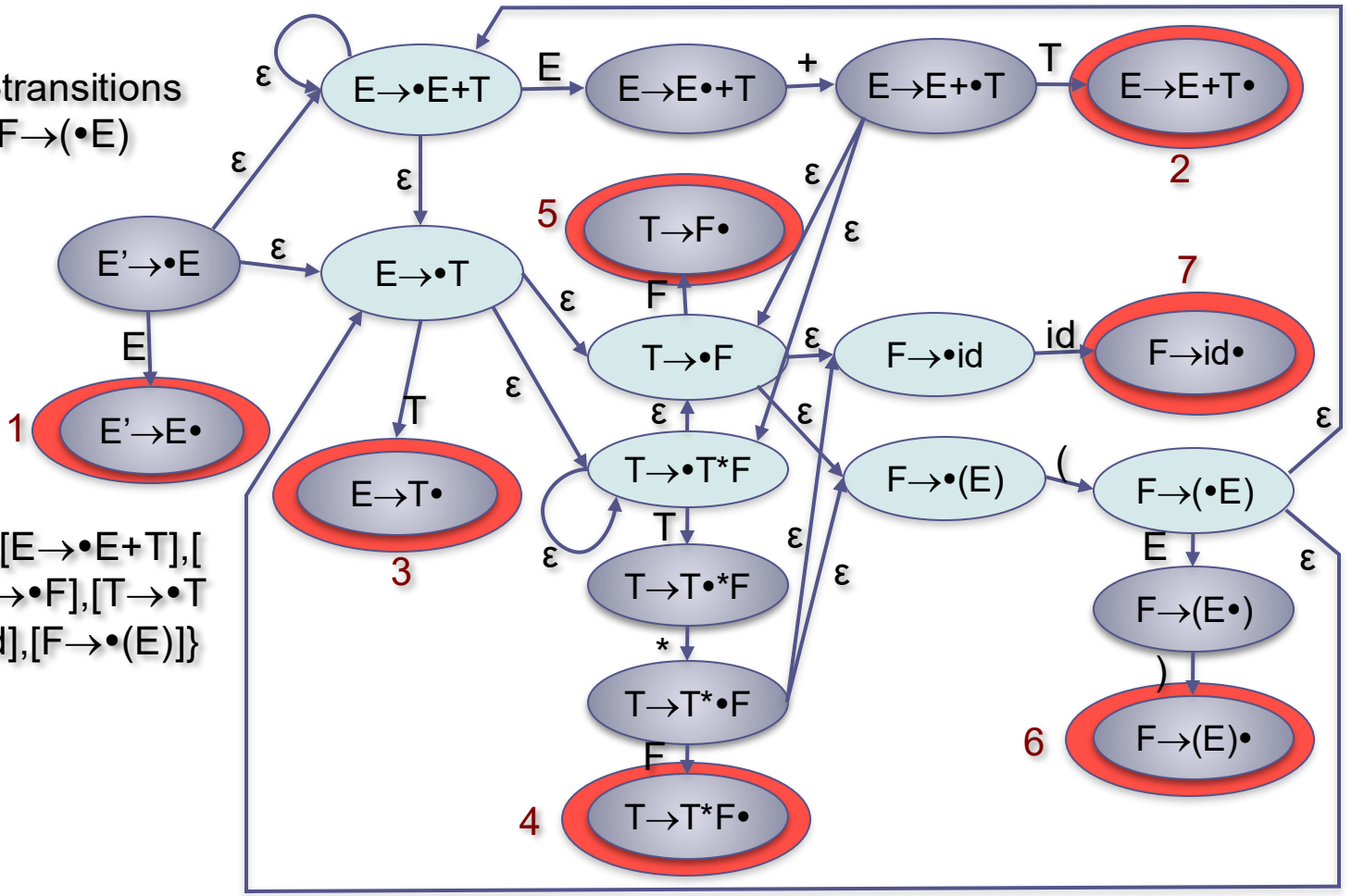
I_0={[E'→•E],[E→•T],[E →•E+T],[T→•F],[T →•T*F],[F→•id],[F →•(E)]}

SLR grammars – example 3



SLR grammars- example 4

removal of ε-transitions for the state F→(•E)

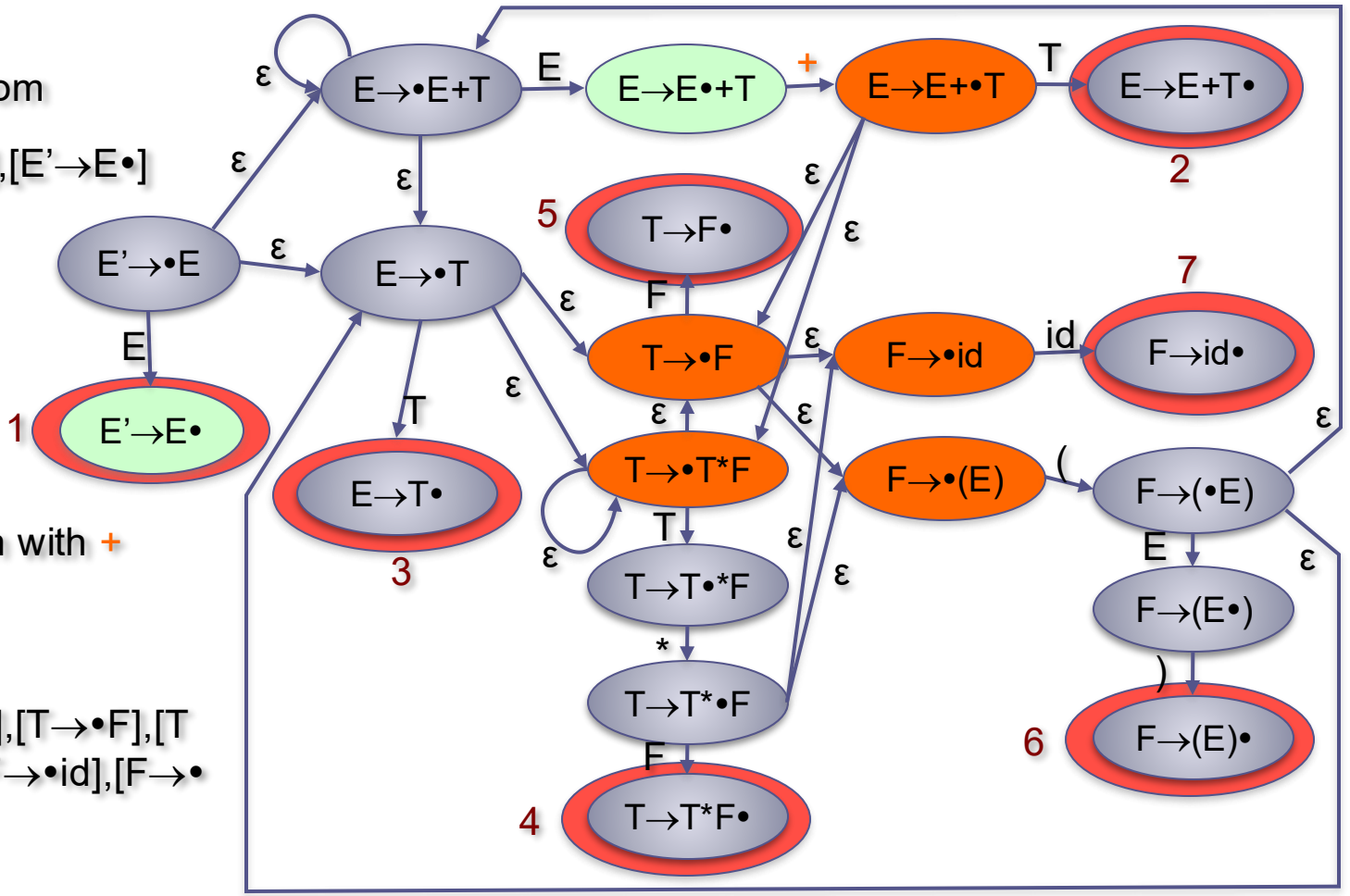


I_4={[F→(•E)],[E→•E+T],[E→•T],[T→•F],[T→•T*F],[F→•id],[F→•(E)]}

SLR grammars- example 5

transitions from

I_1={[E→E•+T],[E'→E•]
}



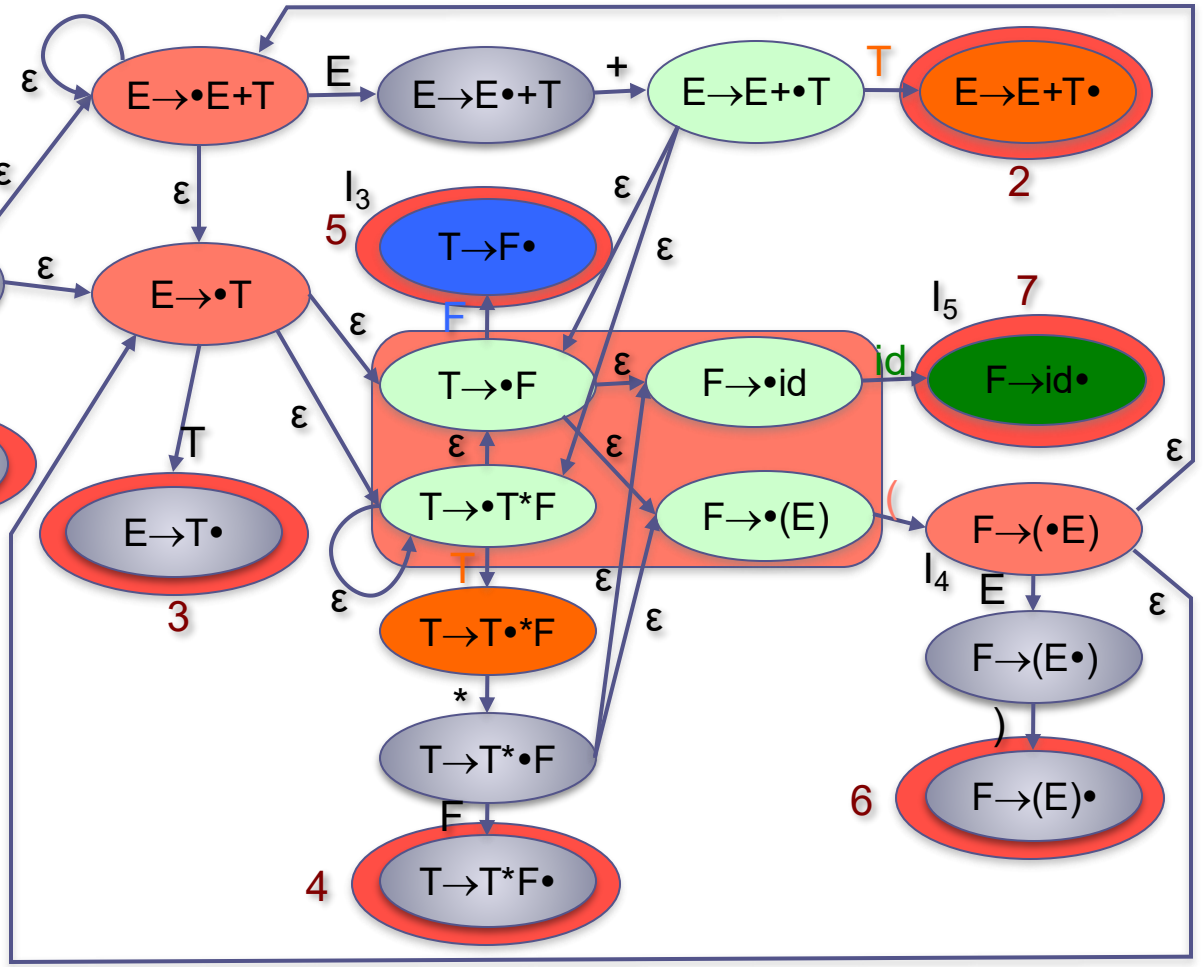
The transition with + defines

I_6={
[E→E+•T],[T→•F],[T→•T*F],[F→•id],[F→•(E)]}

SLR grammars- example 6

transitions from

I_6={
[E→E+•T],[T→•F],[T→•T*F],[F→•id],[F→•(E)]}



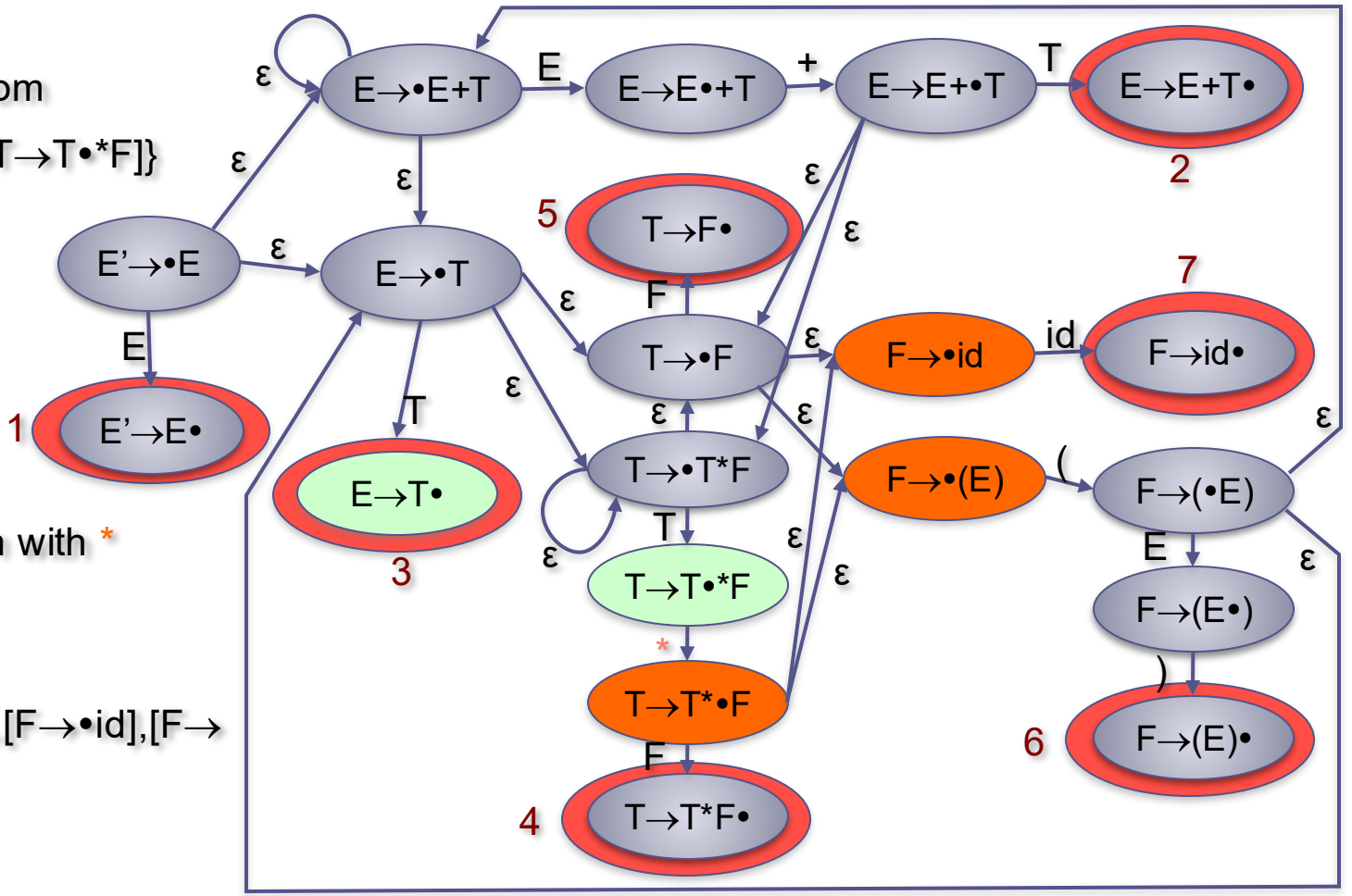
The transition with T defines

I_9={ [E→E+T•],[T→T•*F]}

SLR grammars- example 7

transitions from

I_2={[E→T•],[T→T•*F]}



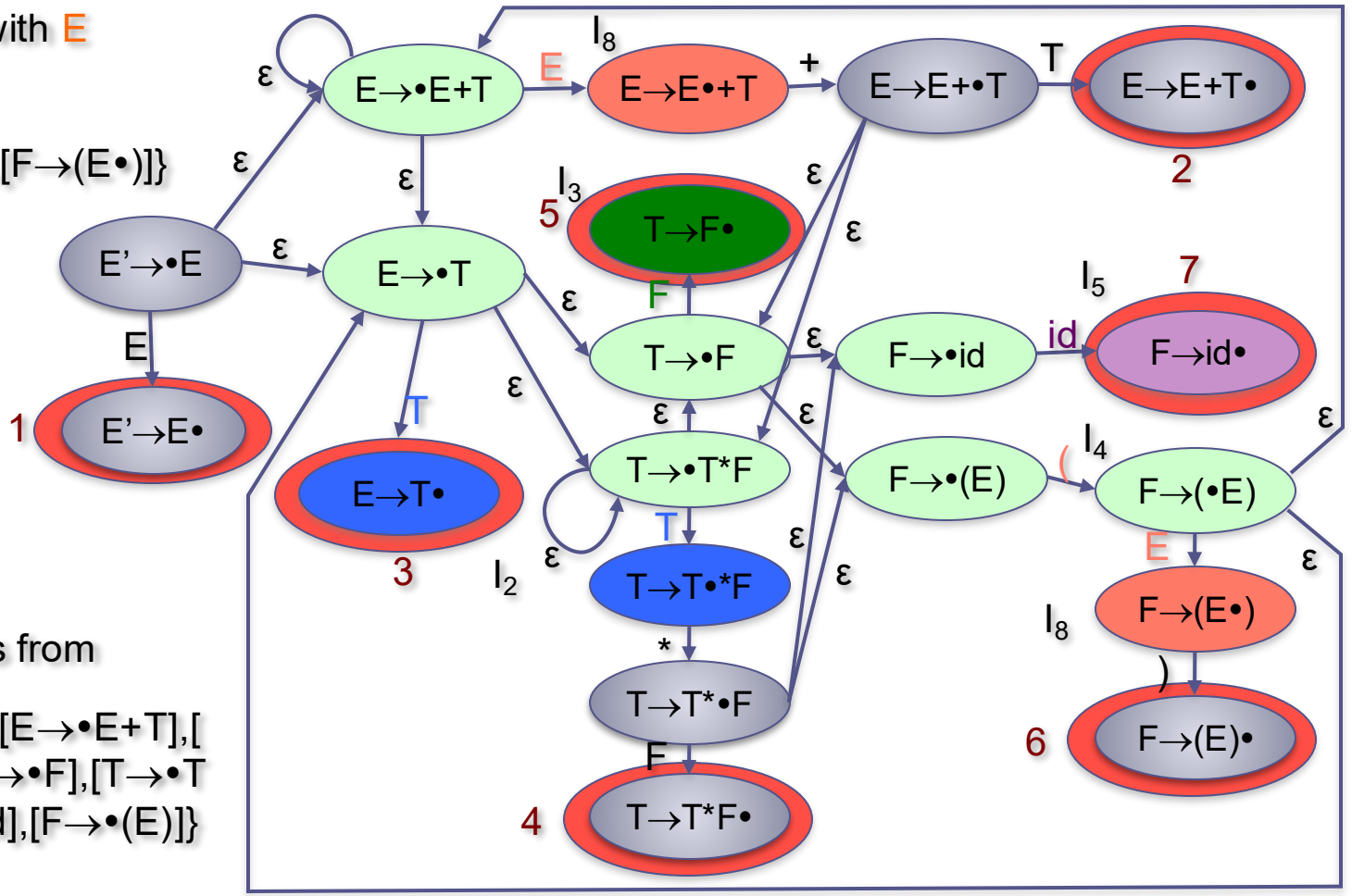
The transition with * defines

I_7={
[T→T*•F],[F→•id],[F→•(E)]}

SLR grammars- example 8

The transition with E defines

I_8={ [E→E•+T],[F→(E•)]}



transitions from

I_4={[F→(•E)],[E→•E+T],[E→•T],[T→•F],[T→•T*F],[F→•id],[F→•(E)]}

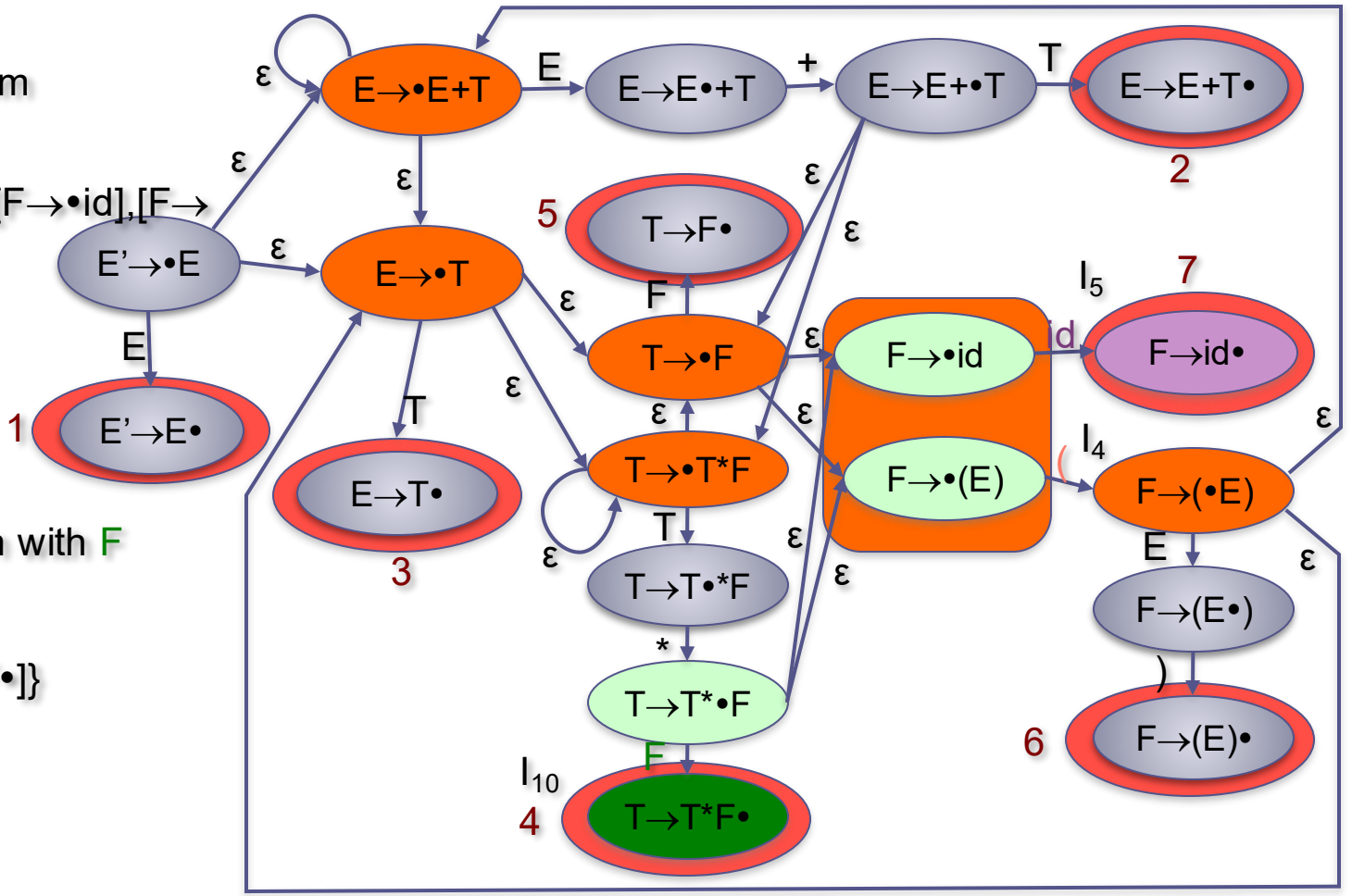
SLR grammars- example 9

transitions from

I_7={
[T→T*•F],[F→•id],[F→•(E)]}

The transition with F defines

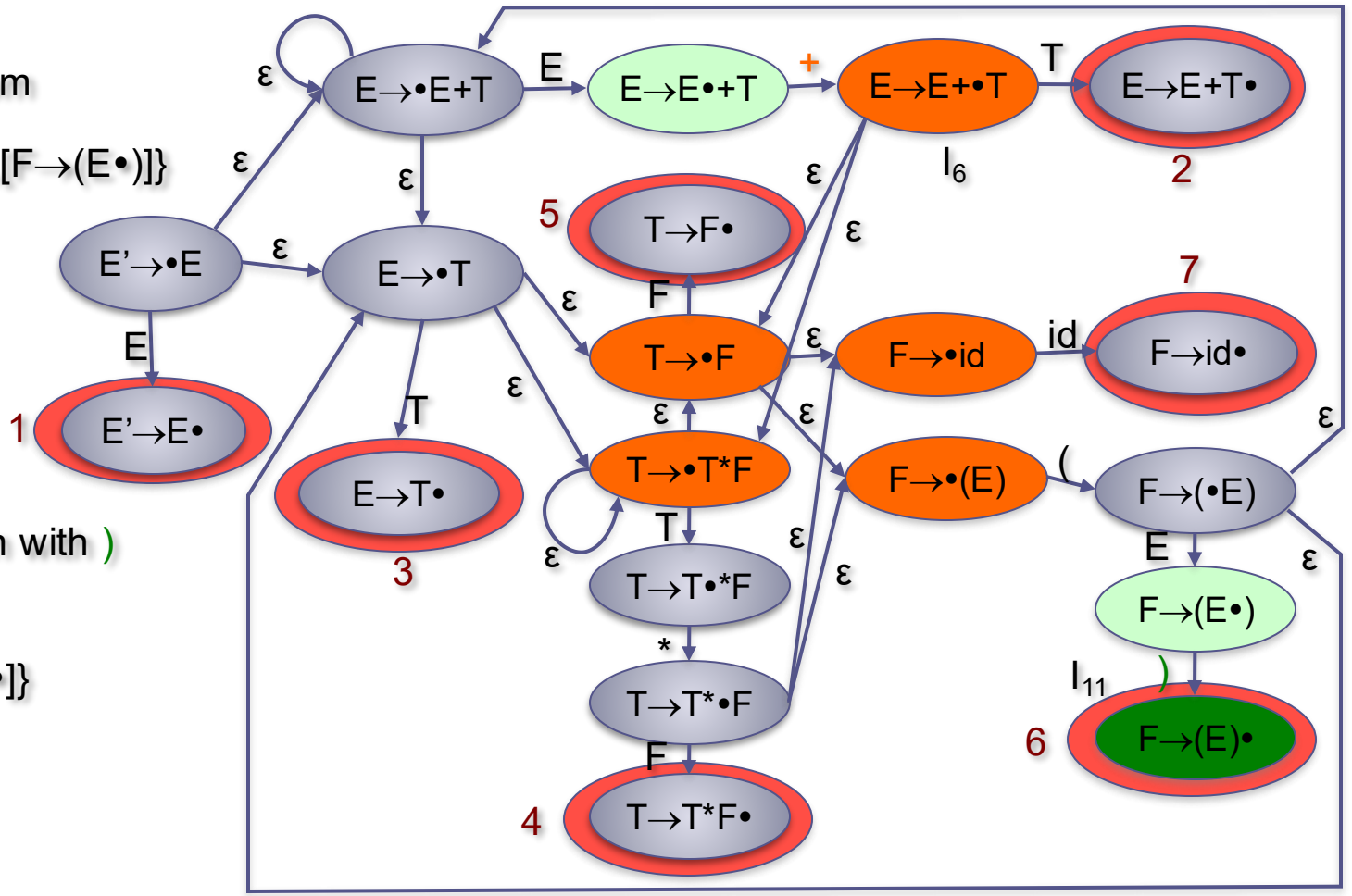
I_{10}={ [T→T*F•]}



SLR grammars- example 10

transitions from

I_8={ [E→E•+T],[F→(E•)]}



The transition with) defines

I_{11}={ [F→(E)•]}

SRL grammars– example 11

I_0={[E'→•E],[E→•T],[E→•E+T],[T→•F],[T→•T*F],[F→•id],[F→•(E)]}

I_1={[E→E•+T],[E'→E•]}

I_2={[E→T•],[T→T•*F]}

I_3={[T→F•]}

I_4={[F→(•E)],[E→•E+T],[E→•T],[T→•F],[T→•T*F],[F→•id],[F→•(E)]}

I_5={[F→id•]}

I_6={ [E→E+•T],[T→•F],[T→•T*F],[F→•id],[F→•(E)]}

I_7={ [T→T*•F],[F→•id],[F→•(E)]}

I_8={ [E→E•+T],[F→(E•)]}

I_9={ [E→E+T•],[T→T•*F]}

I_{10}={ [T→T*F•]}

I_{11}={ [F→(E)•]}

SRL grammars– example 12

state	id	+	*	(	)	E	T	F
0	5			4		1	2	3
1		6						
2			7					
3								
4	5			4		8	2	3
5								
6	5			4			9	3
7	5			4				10
8		6			11			
9			7					
10								
11								

State transition table

Parsing tables- filling ACTION

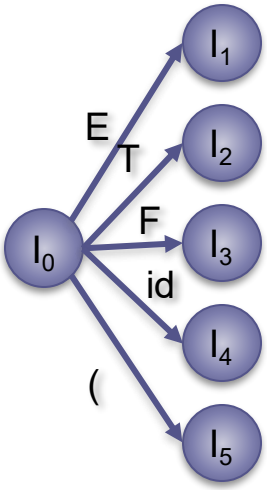
- Given the deterministic automaton that recognizes the prefixes of the right sides of the production rules, it is possible to fill the ACTION and GOTO parse tables
  - The automaton states $C=\{I_0,I_1,\ldots,I_n\}$ correspond to the states 0,1,..n in the scanner
- The actions for state i are defined as follows
  - If $[A\rightarrow\alpha\textcircled{10}a\gamma] \in I_i$ and there is the transition $I_i\rightarrow I_j$ for the input $a\in T$ then ACTION[i,a]=SHIFT j
    - the automaton enters a new state and the match of the right side of a production rule is not yet completed
  - If $[A\rightarrow\alpha\textcircled{10}] \in I_i$ then ACTION[i,a]=REDUCE $A\rightarrow\alpha$ for any terminal symbol a in FOLLOW(A)
  - If $[S'\rightarrow S\textcircled{10}] \in I_i$ then ACTION[i,$]=ACCEPT

Parse tables- filling GOTO

- The GOTO table is filled considering the transitions produced by non terminal symbols for each state
  - If the transition $I_i \rightarrow I_j$ exists for input $A \in N$ then GOTO[i,A]=j
- The start state includes [S'→❿S]
- The missing entries correspond to parse errors
- The parse tables filled with this algorithm are said SLR(1)
  - An SLR(1) grammar is a grammar that admits an SLR(1) parser

Parse tables- example

I_0={[E'→•E],[E→•T],[E→•E+T],[T→•F],[T→•T*F],[F→•id],[F→•(E)]}



ACTION[0,id] = SHIFT 4

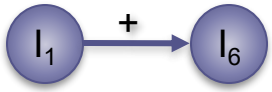
ACTION[0,(] = SHIFT 5

GOTO[0,E] = 1

GOTO[0,T] = 2

GOTO[0,F] = 3

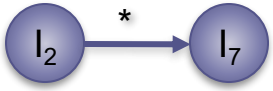
I_1={[E→E•+T],[E'→E•]}



ACTION[1,$] = ACCEPT

ACTION[1,+] = SHIFT 6

I_2={[E→T•],[T→T•*F]}



FOLLOW(E)={$,+,)}

ACTION[2,*] = SHIFT 7

ACTION [2,$] = REDUCE E→T

ACTION [2,)] = REDUCE E→T

ACTION [2,+] = REDUCE E→T

Non SLR(1) grammars

- The filling of the parse tables for a SLR(1) parser fails when there is a conflict in the definition of one of its entries
- For LR more general languages than SLR(1) languages we can build
  - Canonical LR tables
  - LALR tables (LookAhead LR)
- Problems arise when there is more than one valid reduction and we need to avoid to apply incorrect reductions that would lead to a dead end requiring a backtracking step
  - A solution is to use a more informative state that explicitly memorizes the symbols that can follow a handle α for which the reduction $A \rightarrow \alpha$ can be applied

LR grammars

- The elements of a LR(1) grammar are defined by the pairs

$$[A \rightarrow \alpha \bullet \downarrow, a]$$

  - The lookahead element a is used only for the elements with the structure $[A \rightarrow \alpha \bullet, a]$ where we need to consider the reduction only if the next symbol is a
  - In fact, it is not guaranteed that the reduction is valid for all the elements in FOLLOW(A) as it is assumed in the construction of the SLR tables
- A canonical LR parser has more states than SLR and LALR parsers
- The automatic generators of CF parsers yield LALR parsers

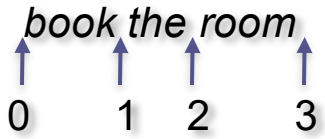
The Earley algorithm (1970)

- General CF may not belong to the language subclasses for which an efficient parser can be obtained (e.g. LL(k) o LR(k))
  - For instance, grammars for NLP are usually ambiguous and may require to build all the possible parse trees for an input sentence
  - The **Earley algorithm** exploits dynamic programming to make the parse step efficient for any CF grammar by storing all the partial parse subtrees corresponding to the sequence components in memory
    - It is a **parallel top-down parser** that avoids the repetition of the solution of the same sub-problems generated by the search with backtracking to reduce the complexity (an expansion is done only once)
    - In the worst case the algorithm has a $O(N^3)$ complexity where N is the number of words in the sentence
    - The algorithm executes a single scan from left to right filling an array (**chart**) of N+1 elements

The chart

- The **chart** stores efficiently the states visited while parsing
  - For each word in the sentence the chart contains the list of all the partial parse trees that have been generated up to a certain step
  - The chart stores a compact encoding of all the computed parse trees in the position corresponding to the sentence end
  - The parse subtrees are stored only once in the chart (the first time they are generated) and they are referred by pointers in the trees using them
  - Each state contains three elements
    - a subtree corresponding to a grammar rule
    - the degree of completion of the subtree (a dotted rule is used – LR(0) element)
    - The position of the subtree with respect to the input sequence encoded as a pair of indexes corresponding to the subtree start and to the dot position

States in the Chart

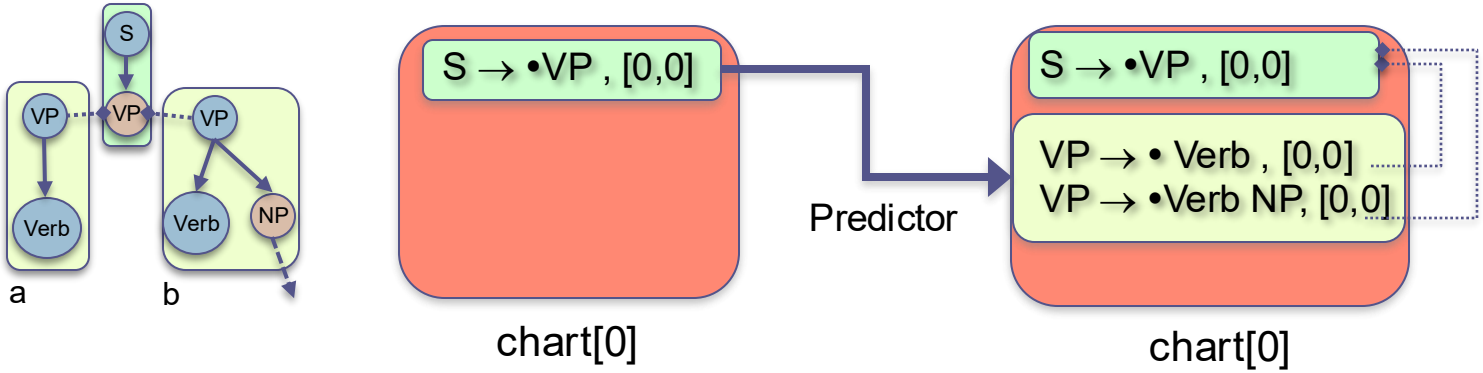


S → •VP , [0,0]
NP → Det •Nominal , [1,2]
VP → Verb NP •, [0,3]

- Parsing processes the states in the chart left to right
  - At each step a state is selected, and one out of three possible operations is applied
    - The operation may generate a new state in the current or next chart position
  - The algorithm always moves forward without removing the generated states but adding new states
  - The presence of the state S → α •, [0,N] in the last chart position indicates a successful parse

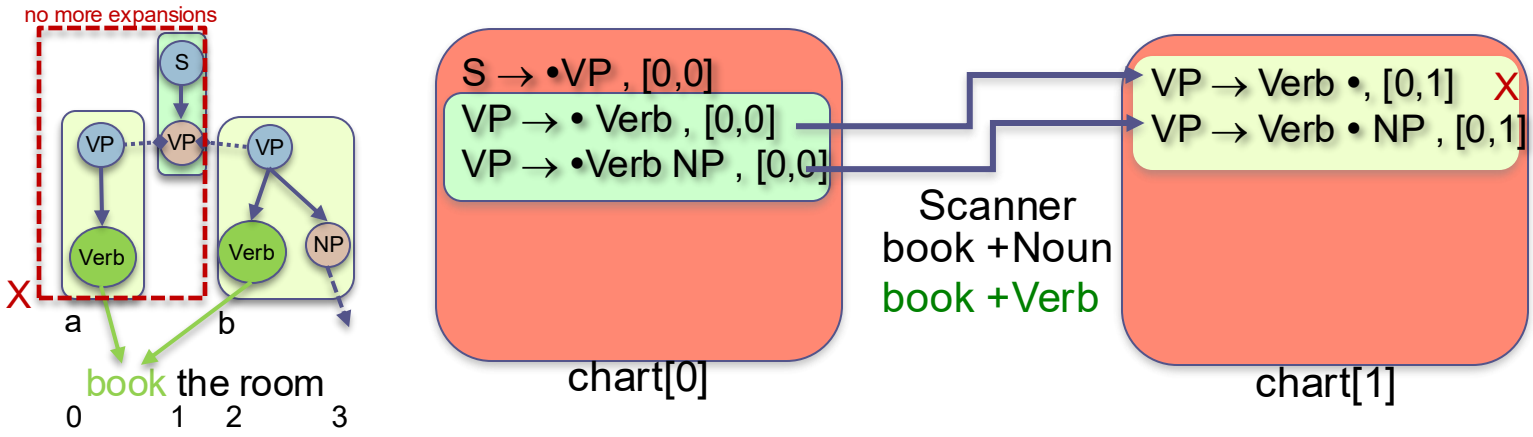
Parser operators - predictor

- **Predictor**
  - It creates new states by the top-down parsing procedure
  - It is applied to each state that has a non terminal symbol on the right of the dot
  - A state is generated for any possible expansion of the non terminal symbol and it is inserted in the same chart position
  - The start and end positions for the inserted states are the same as those of the state that generated them



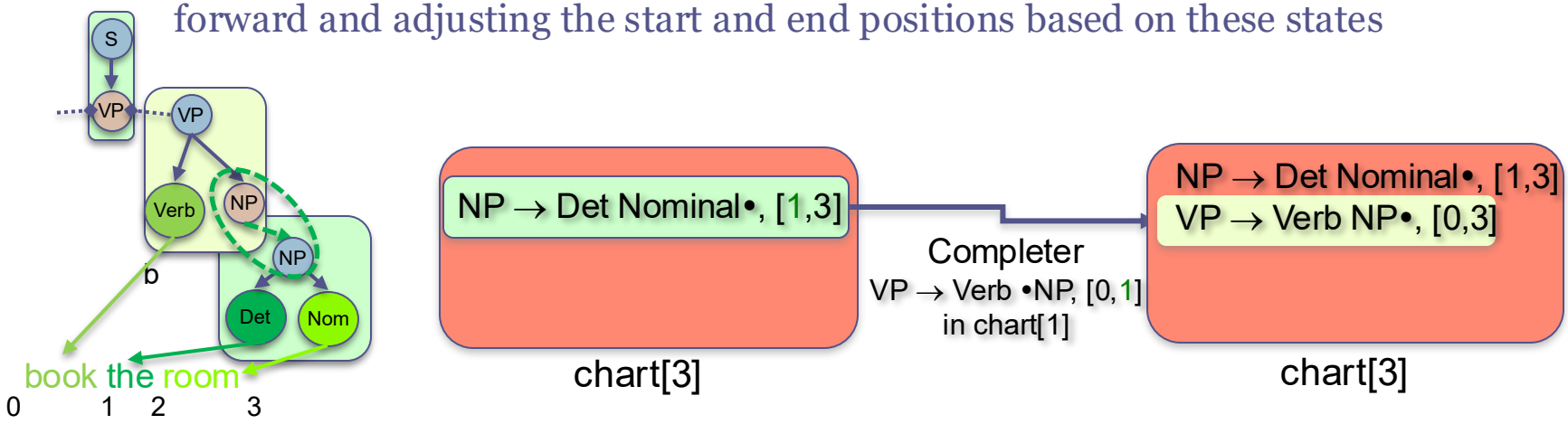
Parser operators - scanner

- **Scanner**
  - When a state has a terminal symbol on the right of the dot, the scanner checks the current symbol in the input sequence
  - A new state is created moving the dot on the right of the predicted terminal symbol
  - The new state in inserted into the following chart position and the end position is increased by 1



Parser operators- completer

- **Completer**
  - It is applied to a state when the dot reaches the right end of the rule
    - It corresponds to the application of the production rule that expands a portion of the input with a non-terminal symbol
  - The completer looks for all the states created in the previous steps that depend on the expansion of this non-terminal symbol
    - It adds all the found states to the current position moving the dot one position forward and adjusting the start and end positions based on these states



Building the parse tree

- The complete parse trees correspond to the states S → α •, [0,N] in the last chart position
  - The completions by the Completer module are used to build the tree
  - For each state we need to keep track of the set of completed states that generated its components
    - This information can be added by the completer once an identifier is associated to each state (for instance, a number)
  - The procedure starts from the complete rule at the chart position N tracing back recursively all the rewrite operations that have been used