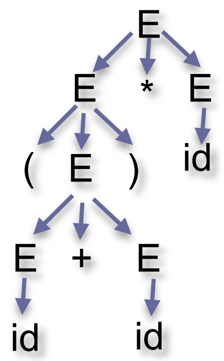
Context free languages

Syntatic parsers and parse trees



Context Free Grammars

- The CF grammar production rules have the following structure

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

  - 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

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

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

CF Grammar– arithmetic expressions

E → "number"

E → (E)

E → E + E

E → E – E

E → E * E

E → 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 all the legal arithmetic expressions

CF Grammars- example

$S \to aB$	$A \to bAA$
$S \to bA$	$B \to b$
$A \to a$	$B \to bS$
$A \to aS$	$B \to aBB$

$T = \{a,b\}$

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

- In general it is 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 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 $|w_1| = k-1$ 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 $|w_1| = k-1$ 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 $|w_1| = k-1$ 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 $|w_1| < k-1$ $|w_2| < k-1$ 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
  - $B \underset{*}{\Rightarrow} w$ 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

$E \rightarrow id$

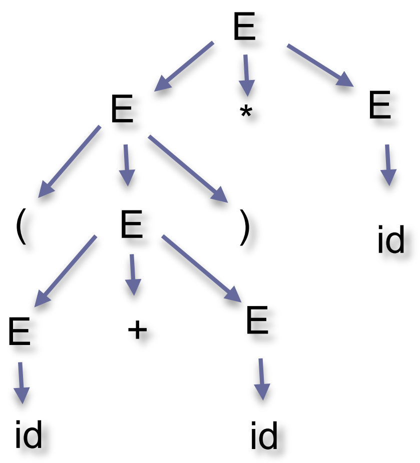
$E \rightarrow (E)$

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow E * E$

$E \rightarrow E / E$



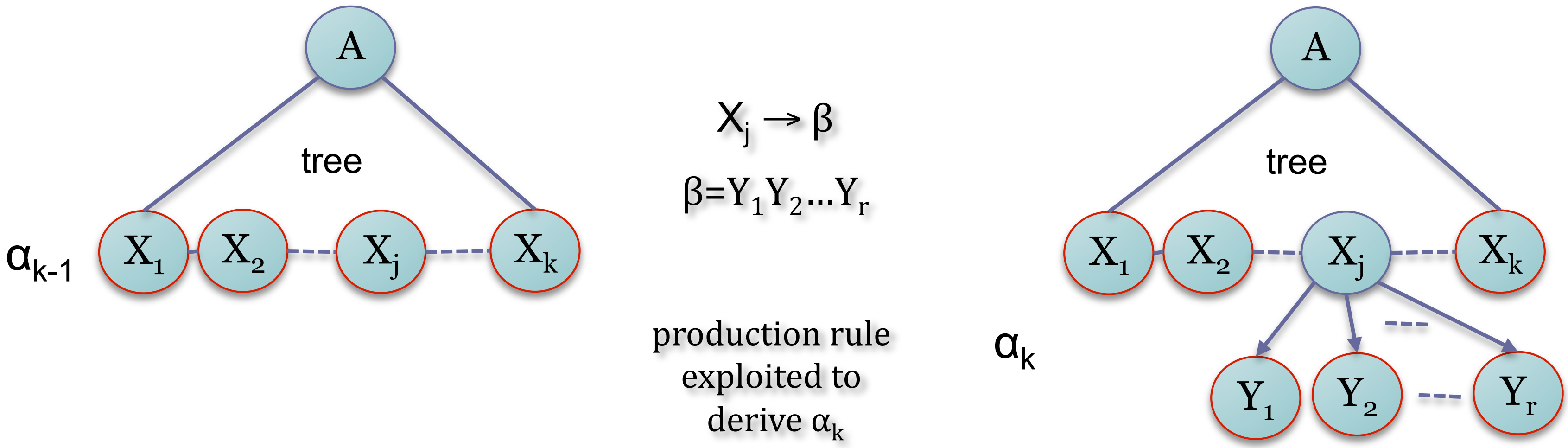
(id+id)*id

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 and non terminal symbols in the right side of the production rule. The children are ordered as the corresponding symbols in the production string
- The parse structure is a tree since the production rules have the structure $N \rightarrow \alpha$ that characterizes the 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

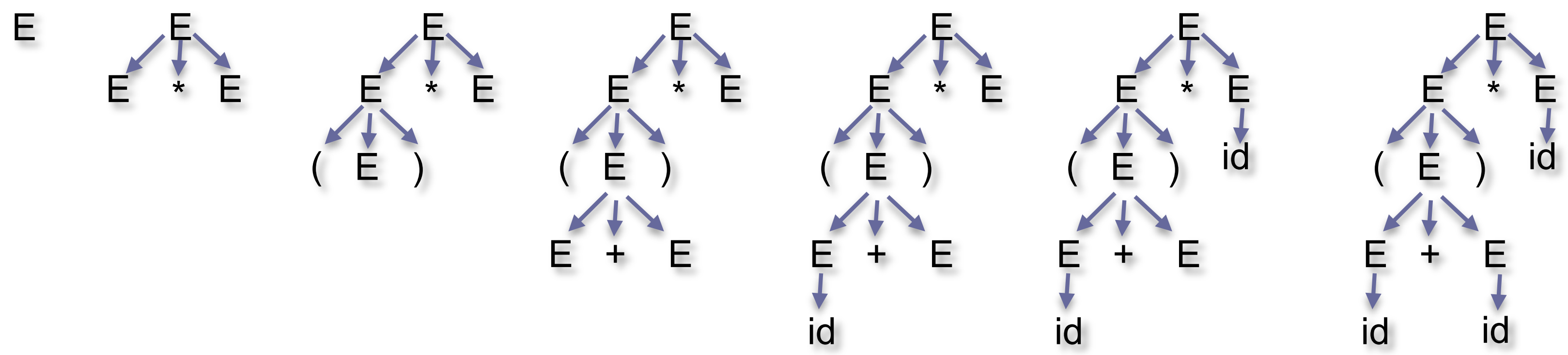
- A parse tree can be interpreted as a representation of a sequence of derivations $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots. \Rightarrow \alpha_n$ where $\alpha_1 = A \in N$
  - For each string α_i the derivation that produces α_{i+1} corresponds to the activation of a production rule and, hence,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

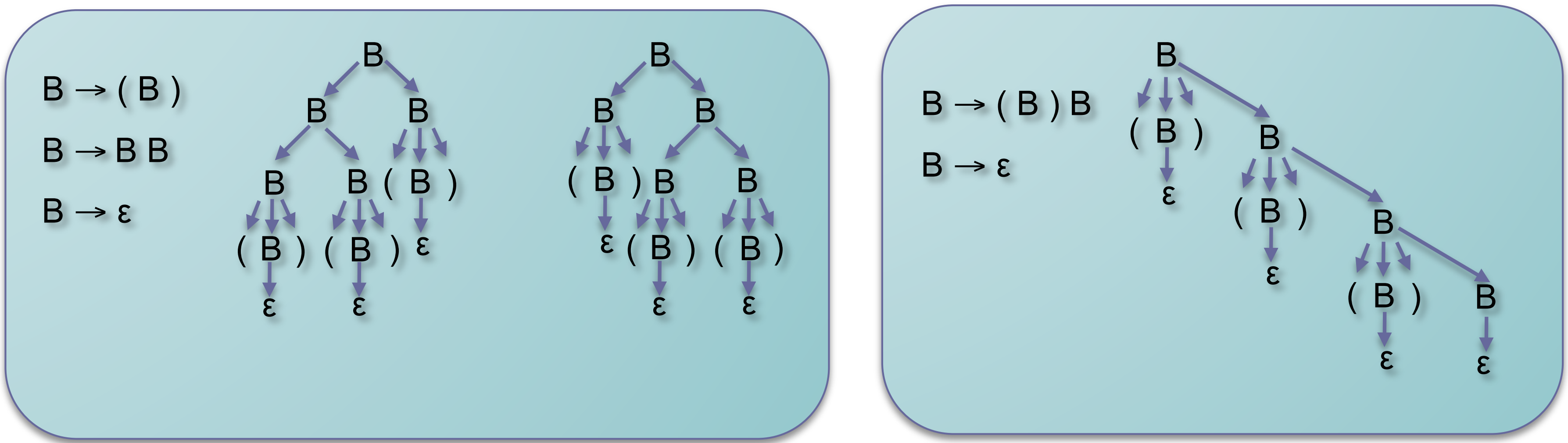
$$E \Rightarrow E * E \Rightarrow (E) * E \Rightarrow (E + E) * E \Rightarrow (id + E) * E \Rightarrow (id + E) * id \Rightarrow (id + id) * id$$



Ambiguous grammars

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

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 in some cases when the parse tree is used to give a semantic interpretation to the input string

$E \rightarrow id$

$E \rightarrow (E)$

$E \rightarrow E + E$

$E \rightarrow E * E$

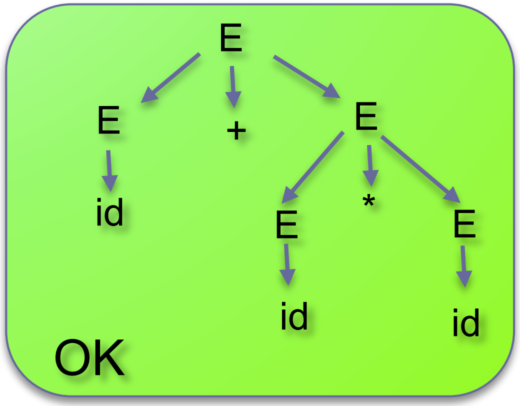
$E \rightarrow -E$

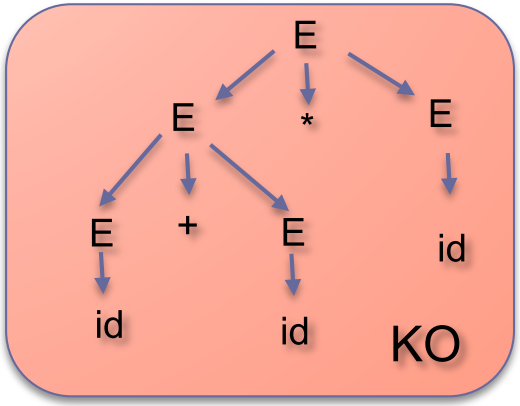
$$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow id + E * E \Rightarrow id + id * E \Rightarrow id + id * id$$

id+id*id

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow id + E * E \Rightarrow id + id * E \Rightarrow id + id * id$$

Ambiguous Grammars – expressions 1





- The grammar does not model the operator
  - 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: it is a single operand or an expression between ()
  - T – terms: it is a product/quotient of factors
  - E – expression: it is the sum/subtraction of terms

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

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

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

- The 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)

Marco Maggini

Ambiguous Grammars - expressions

id+id*id

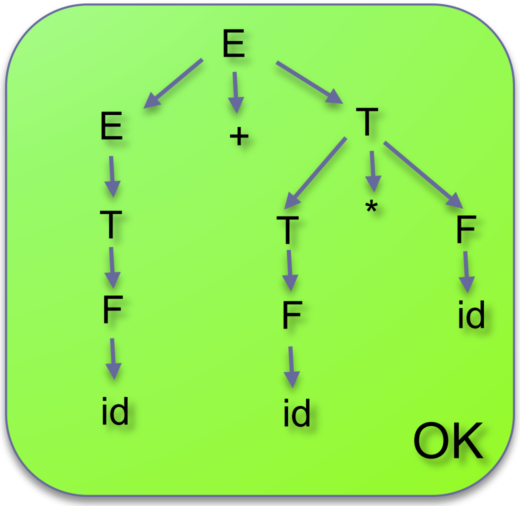
derivation with left rewriting

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow id + T \Rightarrow id + T * F \Rightarrow id + F * F \Rightarrow id + id * F \Rightarrow id + id * id$

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

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

$F \rightarrow (E) \mid id$



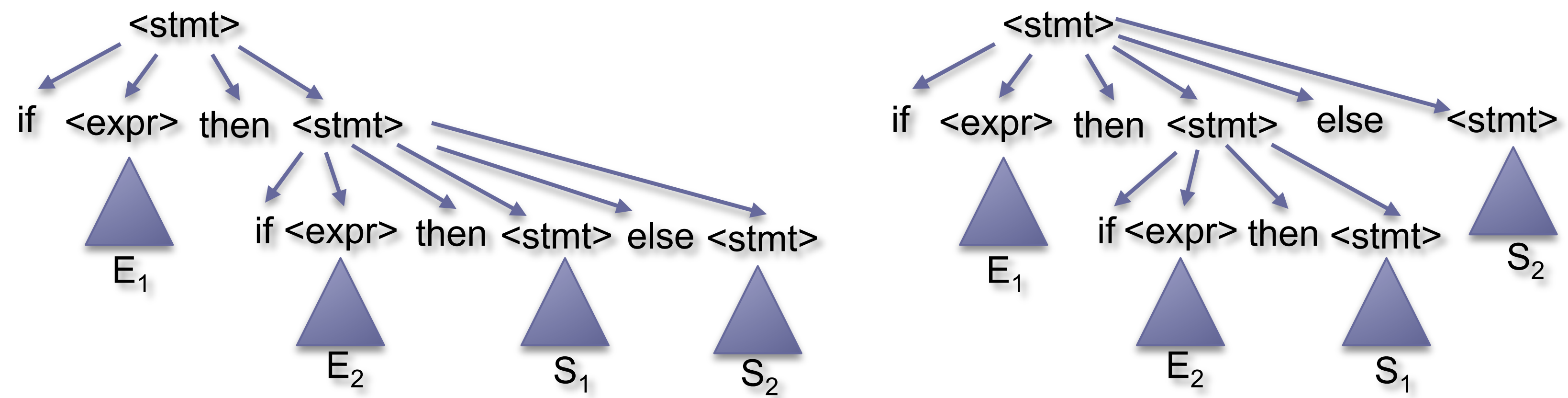
Ambiguity – if… then… else

<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” in the string
  - The common used rule is that the “else” is attached to the closest “if”

<stmt> → <stmt_c> | <stmt_u>

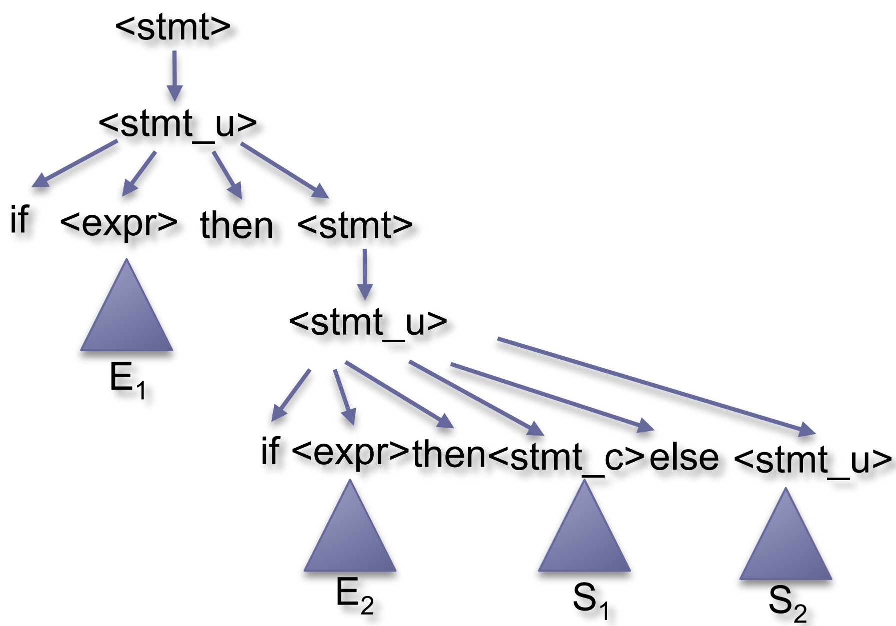
<stmt_c> → if <expr> then <stmt_c> else <stmt_c>

<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 in order 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 α $(T \cup N)^*$

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

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

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

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

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

Left recursion- example

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

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$\alpha = +T \quad \beta = T$

$\alpha = *F \quad \beta = F$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Left factorization

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

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \quad \Longrightarrow \quad \begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \end{array}$$

  - It is assumed that $\beta_1, \beta_2, \ldots, \beta_n$ do not share a same prefix
  - When considering a left-to-right parsing, the left factorization allows the decision of 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

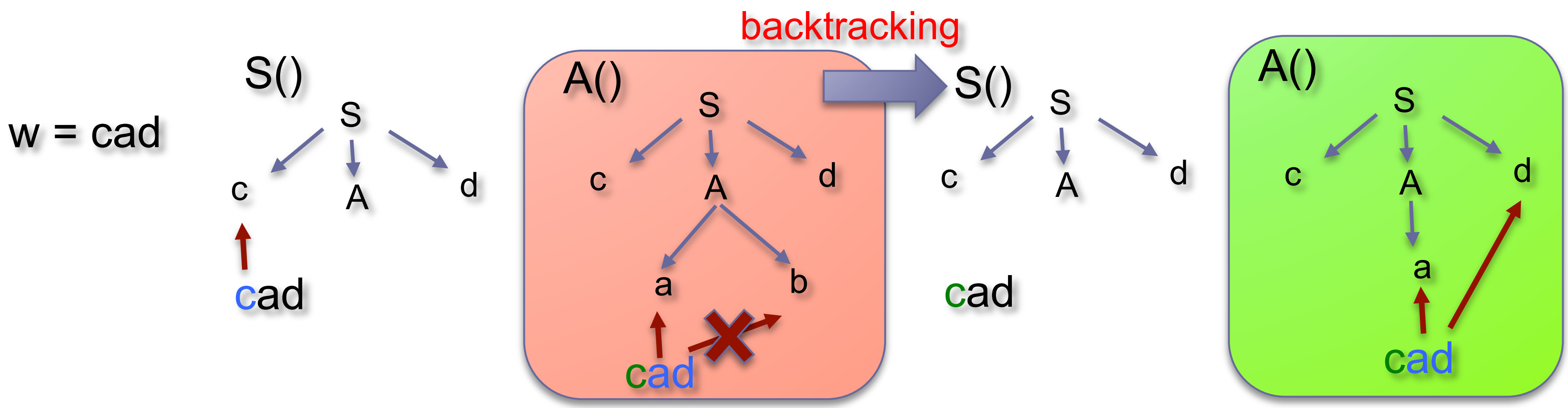
Top-down parsing

- The top-down parsing is an algorithm that tries to build the parse tree starting form the root, adding nodes in pre-order
- The top-down parsing
  - In general there is the need of backtracking –when there are more choices for the production rule to expand, the first one is tried and the others are tried only when 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 and a parse error is generated)

Top-down parsing - example

S → cAd S()

A → ab | 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 han can be activated to expand a non terminal symbol in the partial tree
- The 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

Input string $x_1x_2.....x_n\$$

Input cursor (lookahead symbol)

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 symbols T, the corresponding recursive function is called
  - the function generates the parse sub-tree rooted at the node T

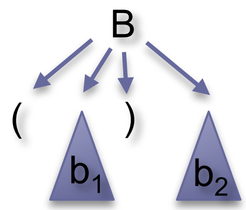
Top-down parsing- example

B → (B) B | ε

```
Node *B(int *lh, char *s) {
  Node *b1,*b2;
  if(s[*lh]==‘(‘) {
    (*lh)++; b1 = B(lh,s);
    if((s[*lh]==‘)‘)&&(b1!=NULL)){
      (*lh)++; b2 = B(lh,s);
      if(b2==NULL) {
        freeTree(b1); return(NULL);
      } else
        return tree4(‘B‘,tree0(‘(‘),b1,tree0(‘)’),b2);
    } else {
      freeTree(b1); return NULL;
    }
  } else
    return tree1(‘B’,tree0(‘e’));
}
```

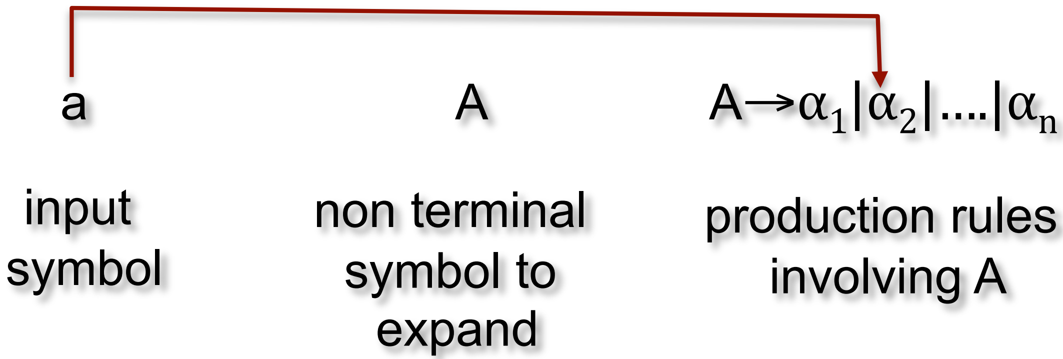
input cursor

input string



Lookahead parsers

- The left recursion is removed
- The production rules are left factorized
  - For a subset of grammar a lookahead parser can be constructed such that the parsing procedure does not require backtracking
  - The lookahead terminal symbol always allows the selection of only one production rule to be expanded



Marco Maggini

Lookahead parsing

- State diagrams that represent the sequences in the right side of the production rules can be exploited to determine the production rule to be applied
  - The 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)

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

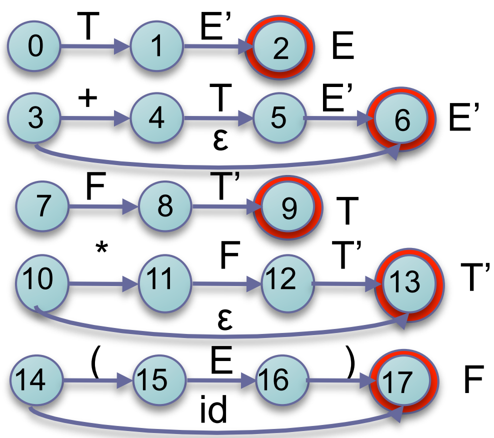


Table based parsing

- A stack is directly used to implement the recursive calls

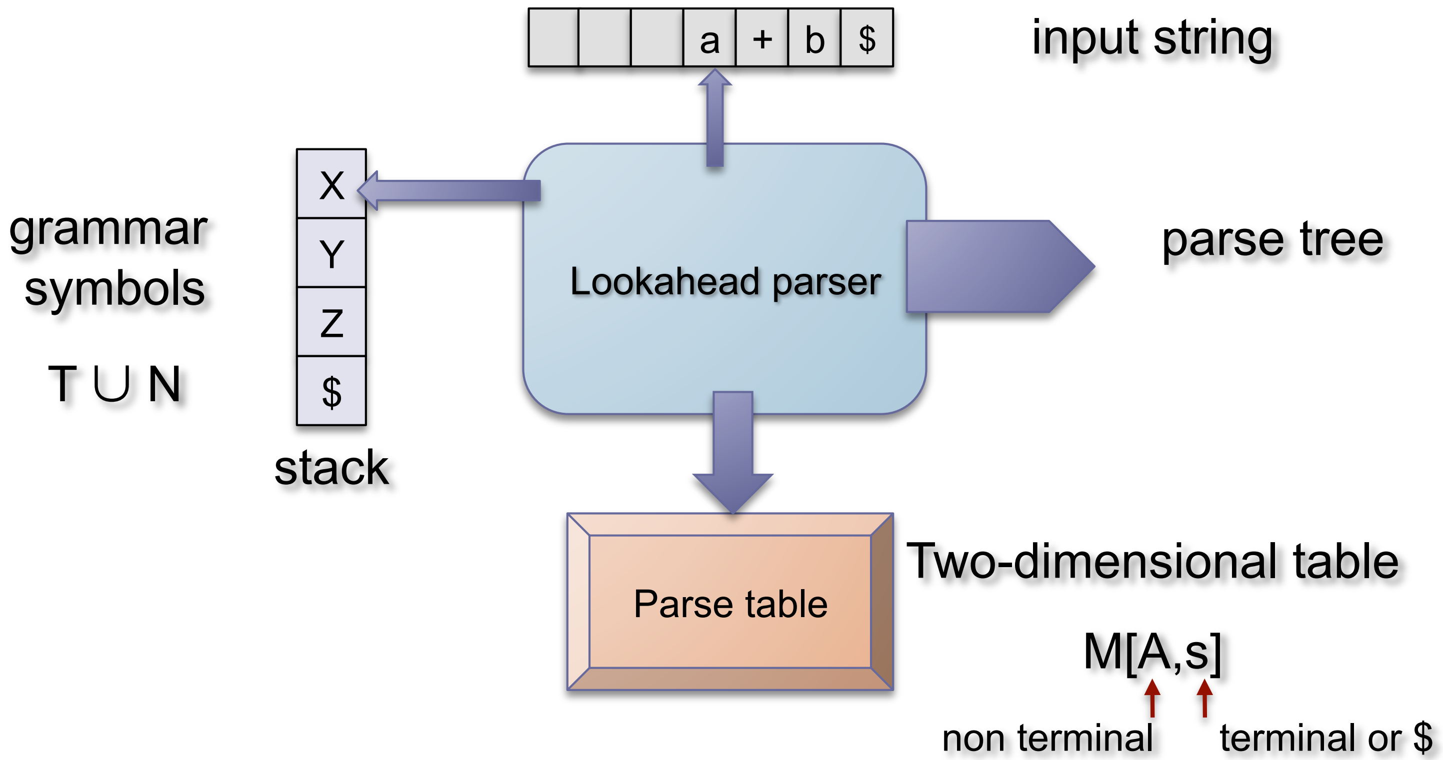


Table based parsing - procedure

- 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 → TE'

E' → + TE' | ε

T → FT'

T' → * FT' | ε

F → (E) | id

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

stack	input	output
$E	id+id*id$	
$E'T	id+id*id$	E → TE'
$E'T'F	id+id*id$	T → FT'
$E'T'id	id+id*id$	F → id
$E'T'	+id*id$	
$E'	+id*id$	T' → ε
$E'T+	+id*id$	E' → +TE'
$E'T	id*id$	
$E'T'F	id*id$	T → FT'
$E'T'id	id*id$	F → id
$E'T'	*id$	
$E'T'F*	*id$	T' → *FT'
$E'T'F	id$	
$E'T'id	id$	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

Table based parsing– table generation 1

- We consider the following two functions
  - FIRST(α) is the set of the terminal symbols that can start a string generated from $\alpha \in (T \cup N)^*$
  - FOLLOW(A) is the set of the terminal symbols that can appear at the right just afterthe 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

- The entries in the parse table are defined by
  - 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 parse 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] = {$A \rightarrow \alpha$} forn 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

S → i E t S S' | a

S' → eS | ε

E → 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				

LL(1) grammars

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

- The LL(1) grammars are a subset of the CF grammars
  - The left recursion is to be removed
  - The grammars are to be left factorized
  - The resulting grammar may not be a LL(1) grammar

Bottom-up parsing

- The parse tree is generated starting from the leaves up to the root
  - The input string is reduced to the start non terminal 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 parsing trees is generate by connecting the child nodes to their parent node

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

abbcde

aAbcde

aAde

aABe

S

substring on which we can apply

$A \rightarrow b$

$B \rightarrow d$

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 is needed in general)

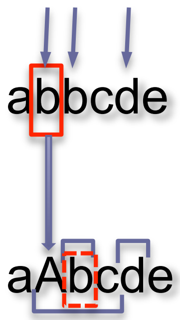
Bottom-up parsing– selecting reductions

- 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 selection we find an intermediate result in which there are no substrings that match the right side of one production rule

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$



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

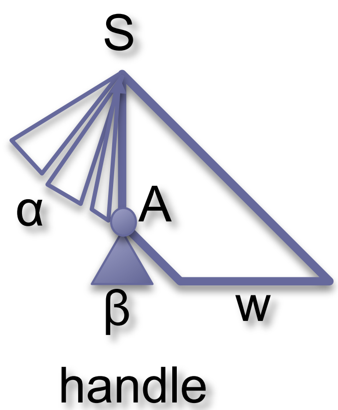
aAAcde

aAAcBe

is no more reducible….

Handle substrings

- A handle substring
  - corresponds to the right side 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 $S \stackrel{*}{\Rightarrow} \alpha A w \stackrel{rm}{\Rightarrow} \alpha\beta w$ (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



the reduction of β into 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

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

  - 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

- A stack is used to store the intermediate results (the tree frontier)
  - the parser pushes symbols into the stack starting 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 symbol A associated to the handle
  - the parser halts with success when the stack contains only the symbol S and the input string is empty
- The action that the parser can execute are
  - 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 halting of the parse
  - ERRORE – the parser outputs a syntax error

Bottom-up parsing- example

E → id

E → (E)

E → E + E

E → 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
  - Also REDUCE E → E+E could have been selected

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
- 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
    - They are a 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)

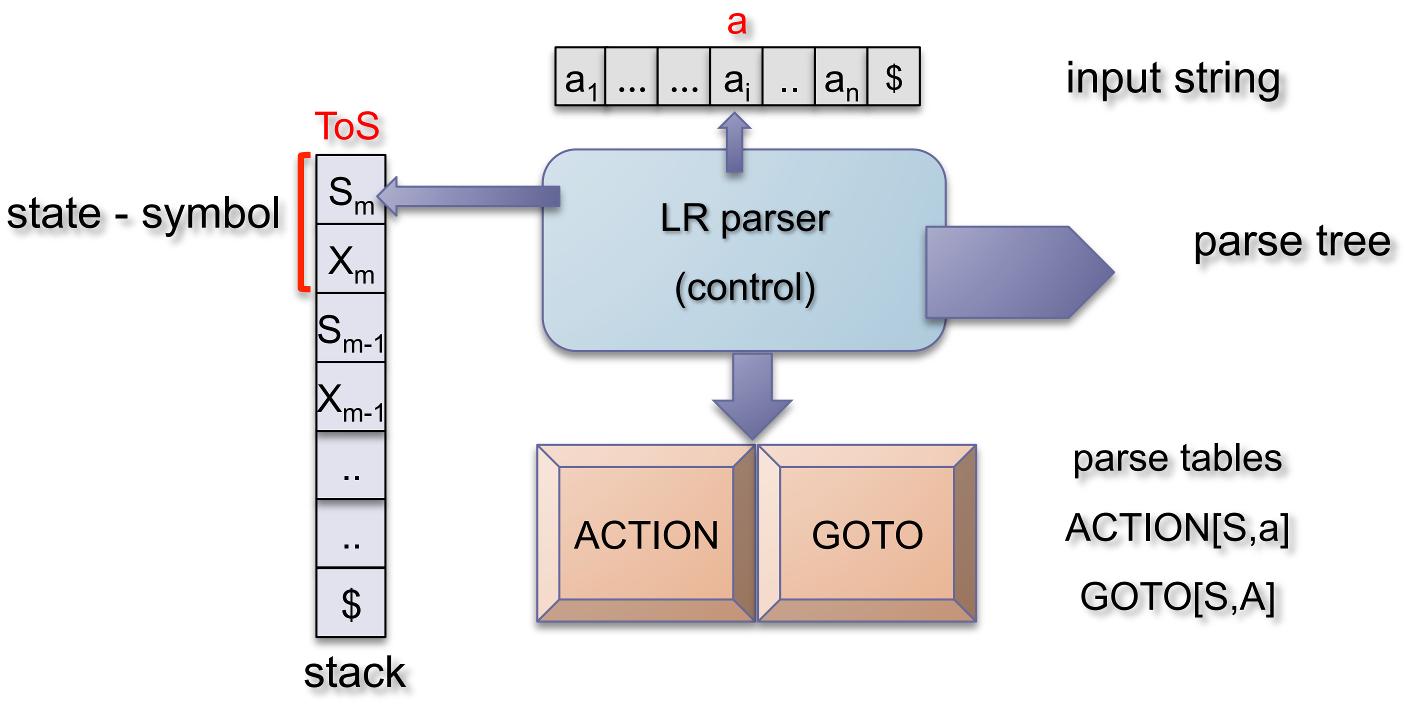
LR(k) parsers

- 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 are able to recognize all the statements of programming languages
  - It is the 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 canning of the input string

LR(k) parsers- structure

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



LR parsers – processing scheme 1

- The stack stores a string of pairs symbol-state

$$S_0X_1S_1X_2S_2....X_mS_m$$

  - 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

already expanded

$$X_1X_2....X_m a_i a_{i+1}.....a_n\$$$

to be expanded

  - 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

a

$$S_0X_1S_1X_2S_2....X_mS_m a_i S\, a_{i+1}.....a_n\$$$

ToS

LR parsers – processing scheme 3

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

a

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

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

- A LR grammar us a 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 they appear at the top of the stack (only the state is needed to perform this check)
    - The recognizer for a handle 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 up to the top of the stack

LR grammars - properties

- The 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

- How can we fill the parsing tables?
  - We consider the case of Simple LR (SLR) grammars that is a proper subset of LR grammars

SLR grammars

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

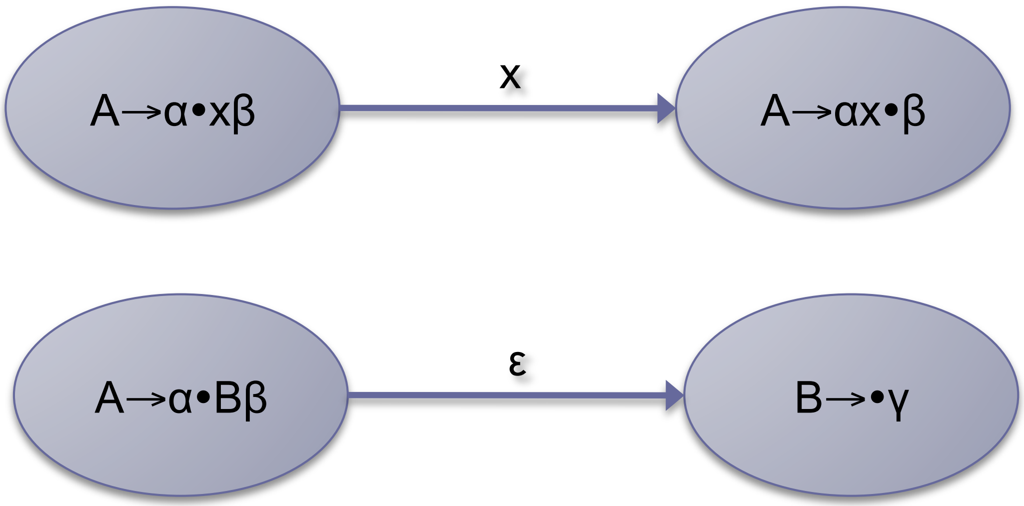
$$\#i \quad A \rightarrow xyz \quad \begin{cases} A \rightarrow \bullet xyz & (i,0) \\ A \rightarrow x \bullet yz & (i,1) \\ A \rightarrow xy \bullet z & (i,2) \\ A \rightarrow xyz \bullet & (i,3) \end{cases}$$

  - An element is defined by a pair of indexes (#production,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 a the states of a finite state automaton

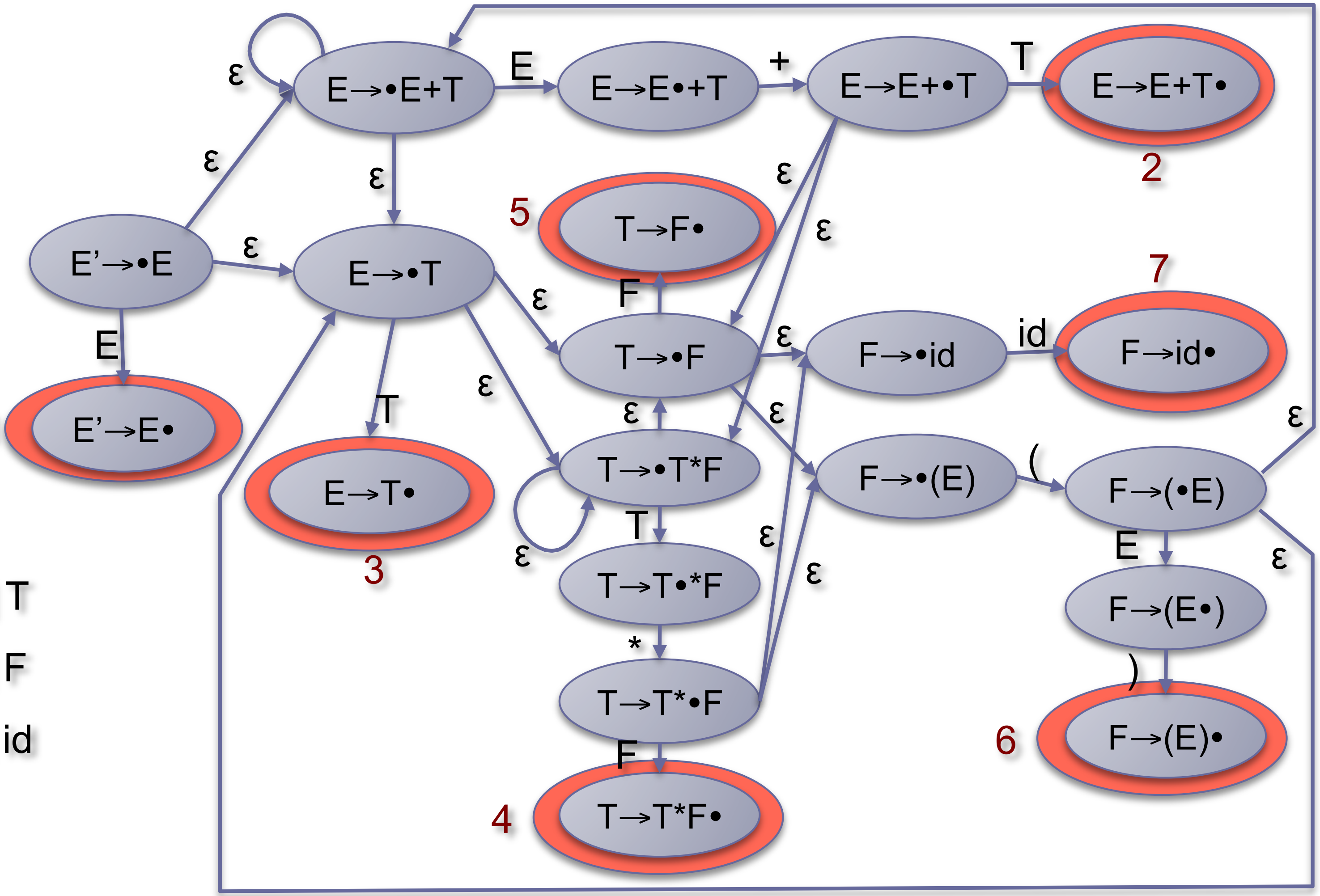


the input x is accepted to move one step forward in the recognition of αxβ

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)
  - We build the FSA recognizing the right sides of the production rules

SLR grammars– example 1



1 E’ → E

2-3 E → E+T| T

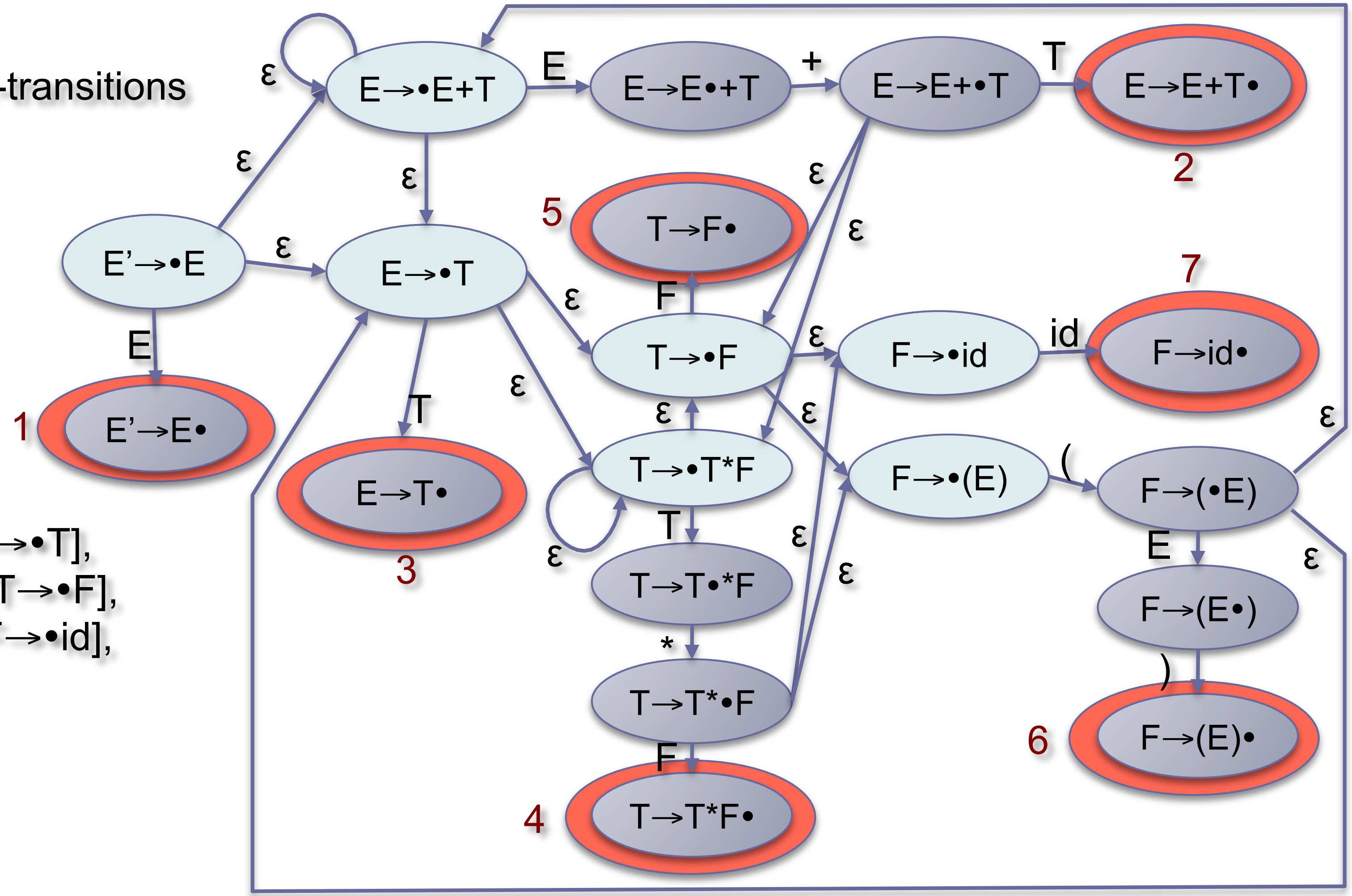
4-5 T → T*F | F

6-7 F → (E) | id

Marco Maggini Language processing technologies

SLR grammars- example 2

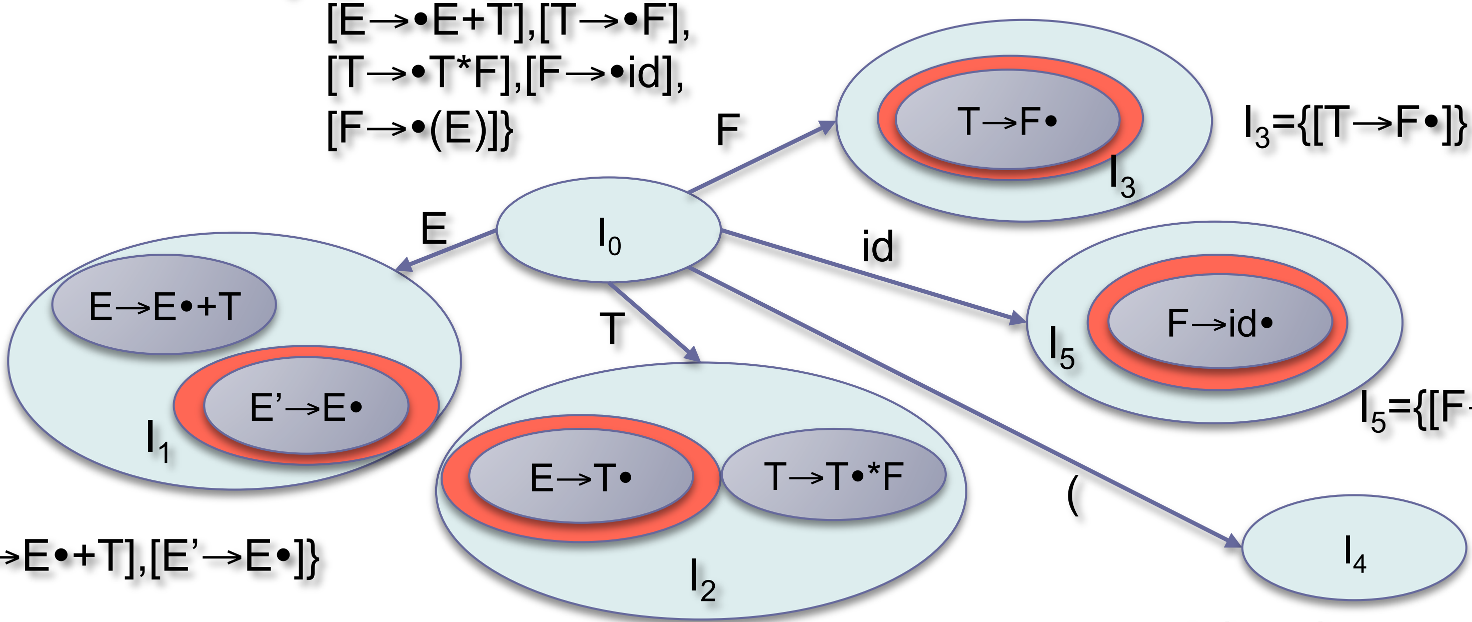
removal of ε-transitions



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

SLR grammars – example 3

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



I_3={[T→F•]}

I_5={[F→id•]}

I_1={[E→E•+T],[E'→E•]}

I_2={[E→T•],[T→T•*F]}

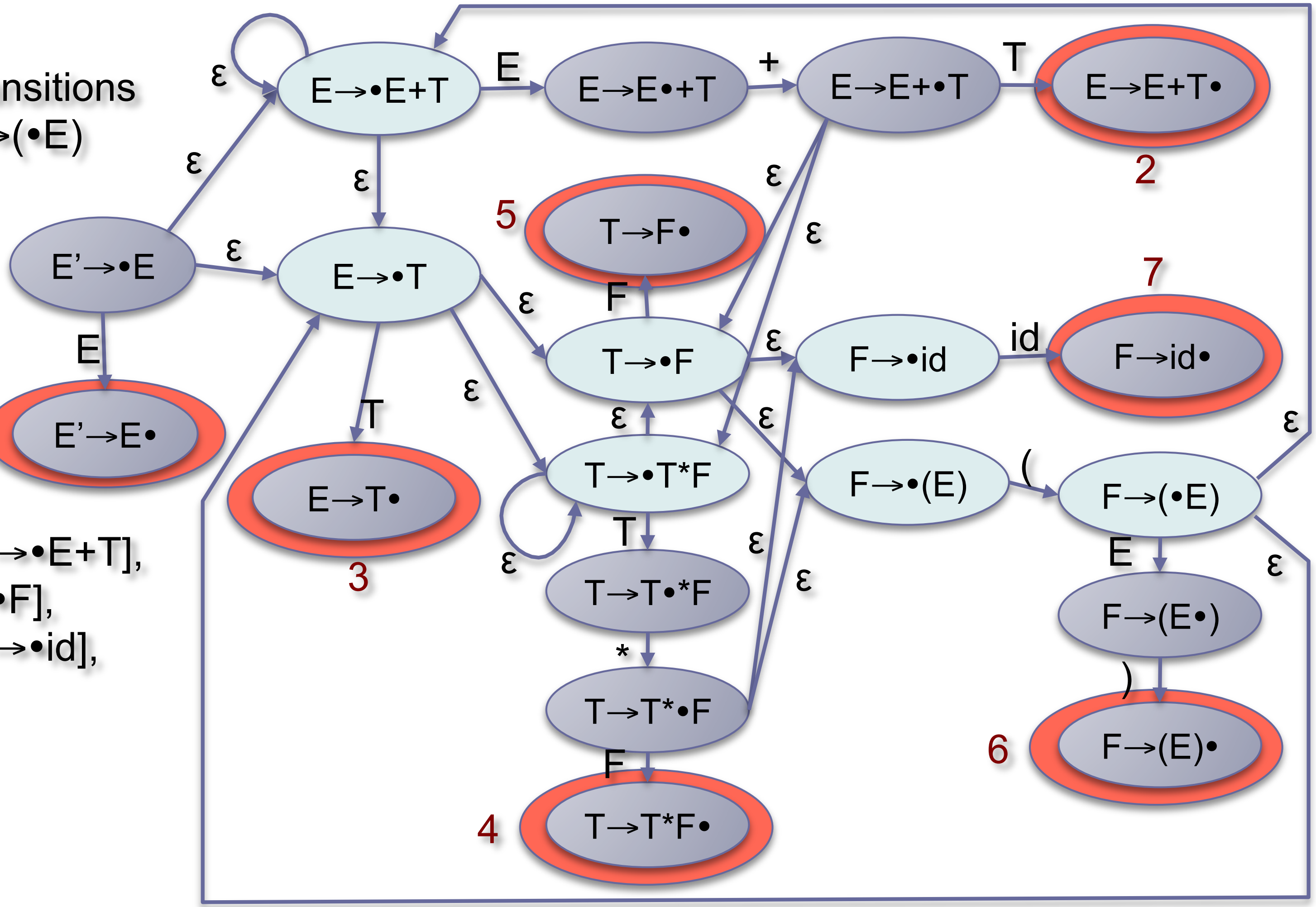
there are no ε-transitions from the states in I_1,I_2, I_3, I_5

I_4 is to be determined with the removal of ε-transitions

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)]}



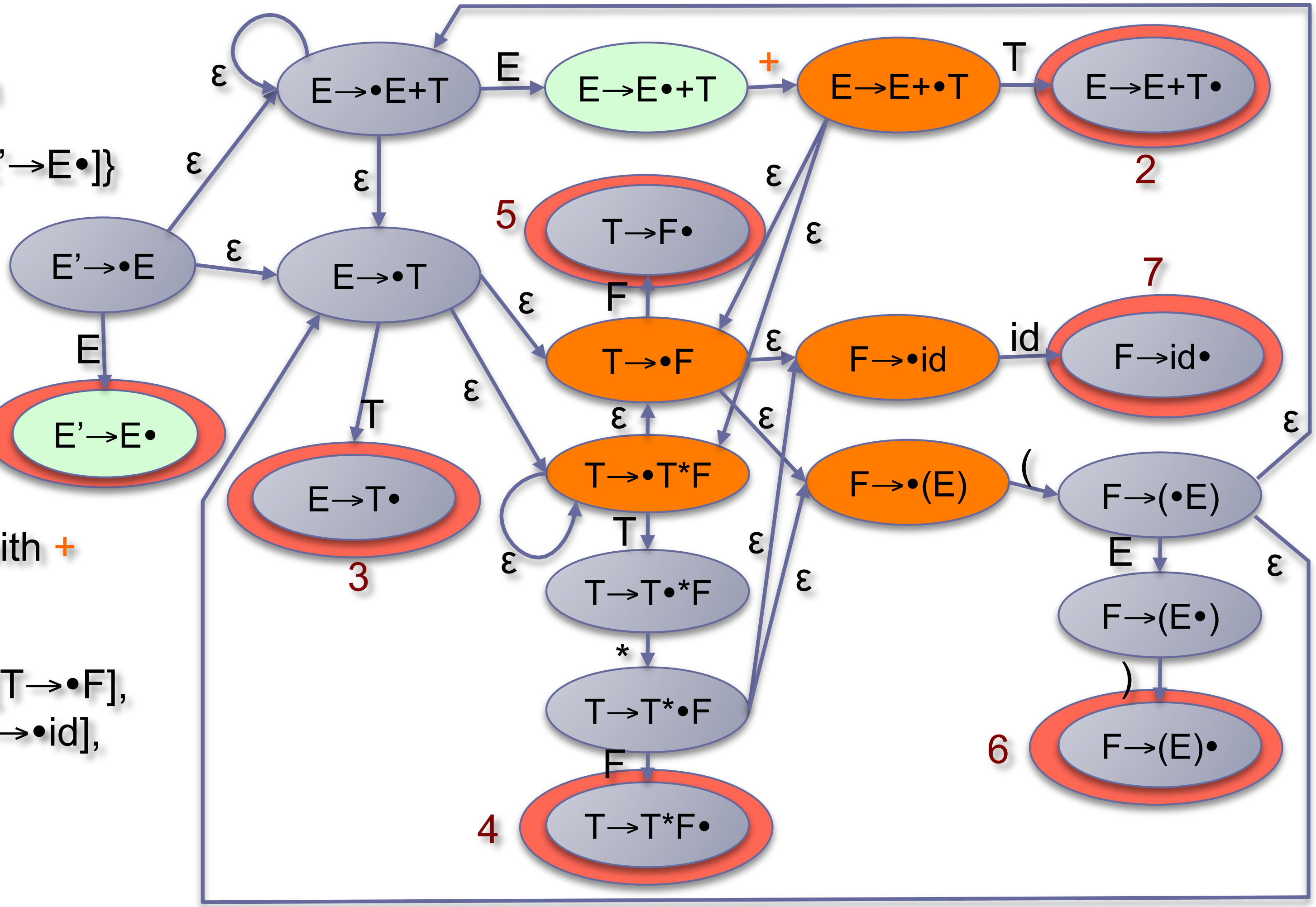
SLR grammars- example 5

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)]}



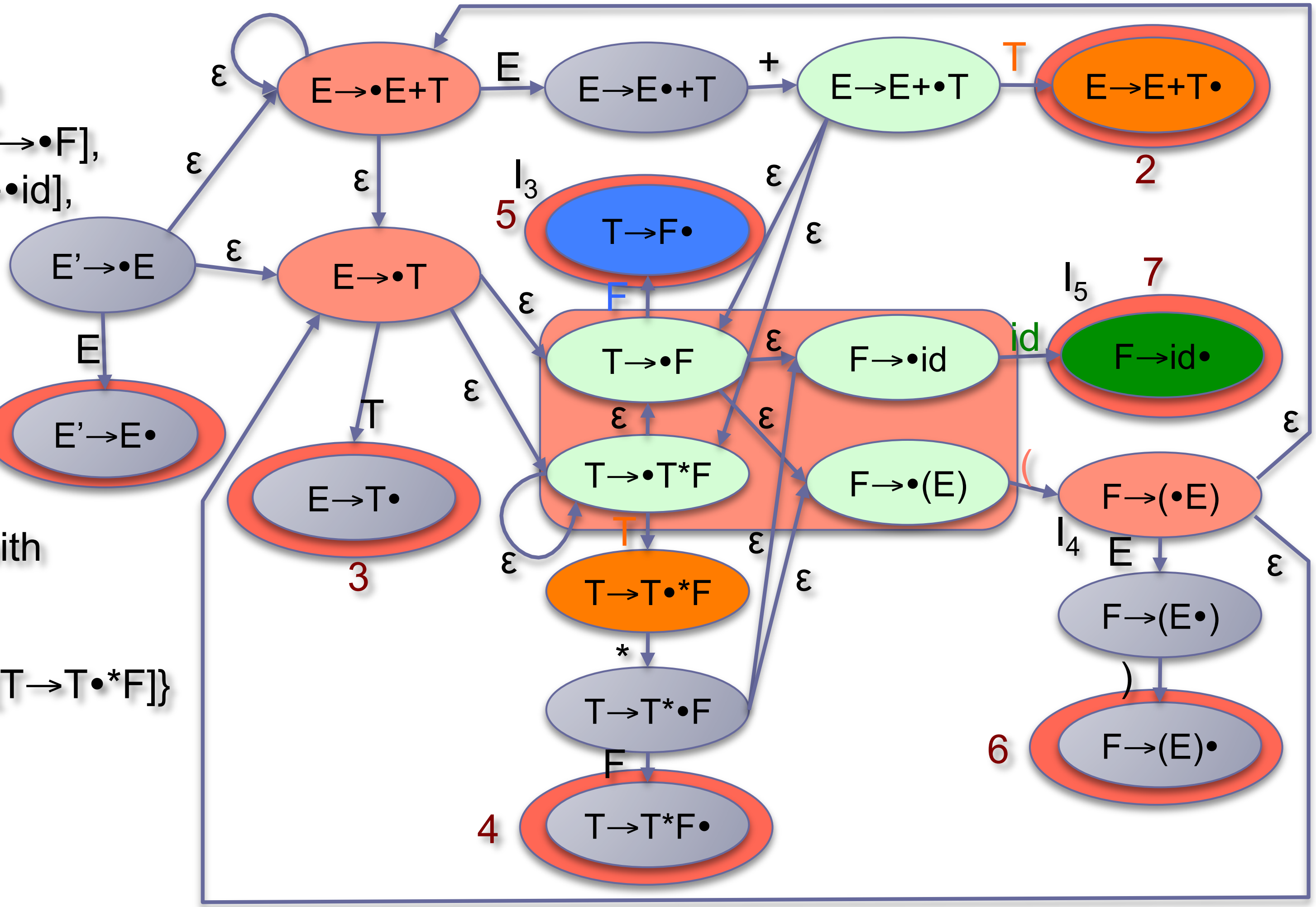
SLR grammars- example 6

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]}



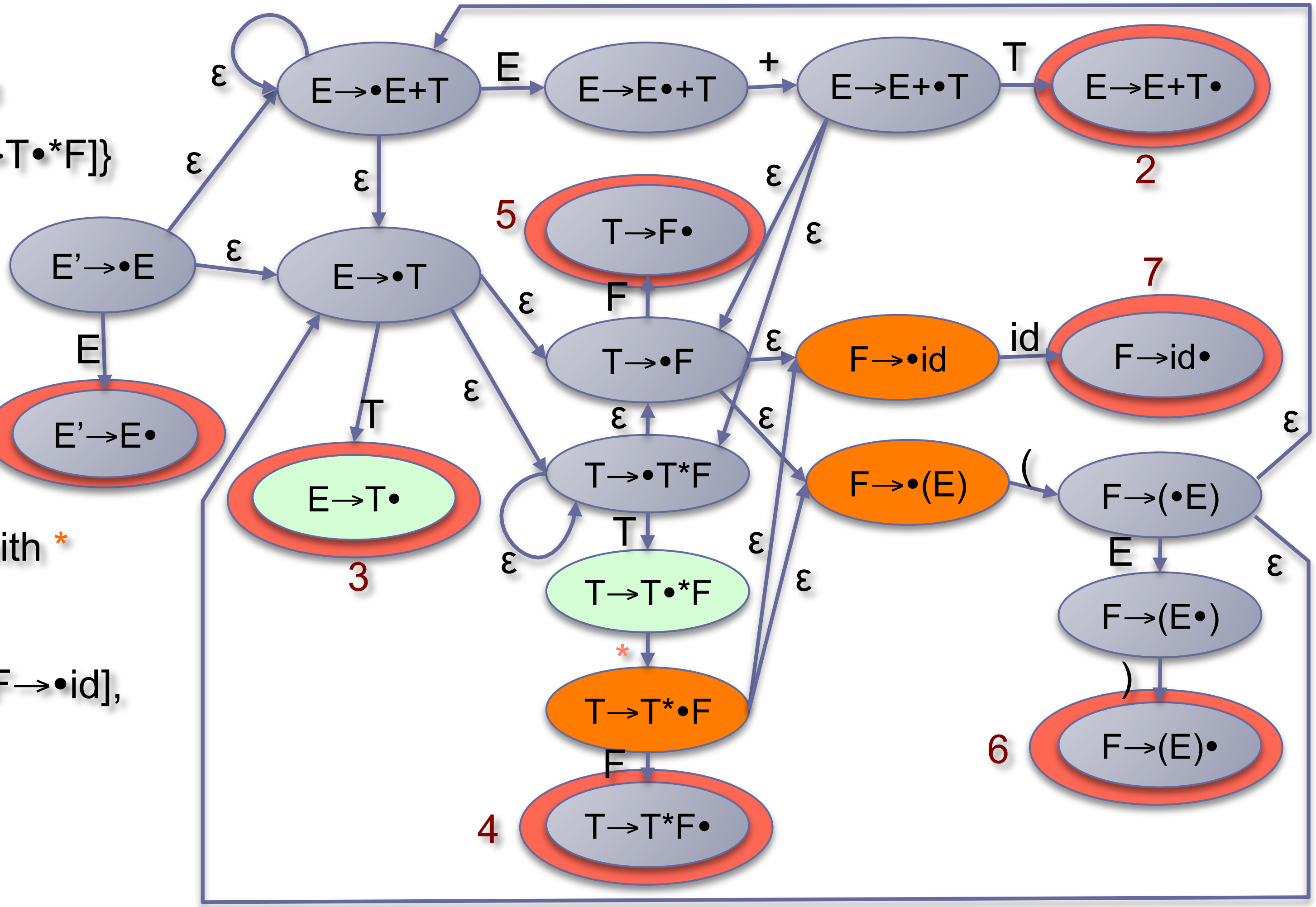
SLR grammars- example 7

transitions from

I_2={[E→T•],[T→T•*F]}

The transition with * defines

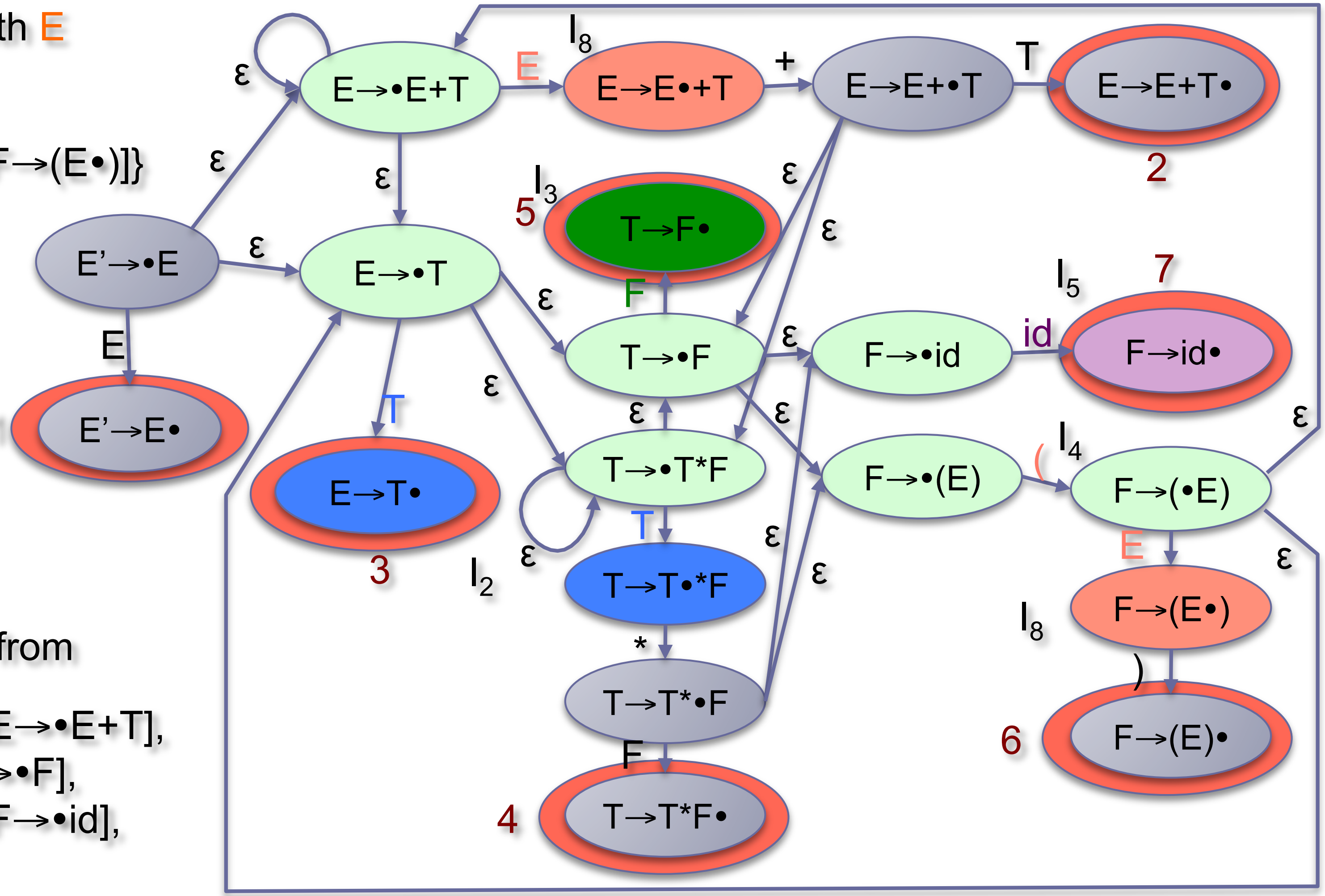
I_7={ [T→T*•F],[F→•id], [F→•(E)]}



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)]}

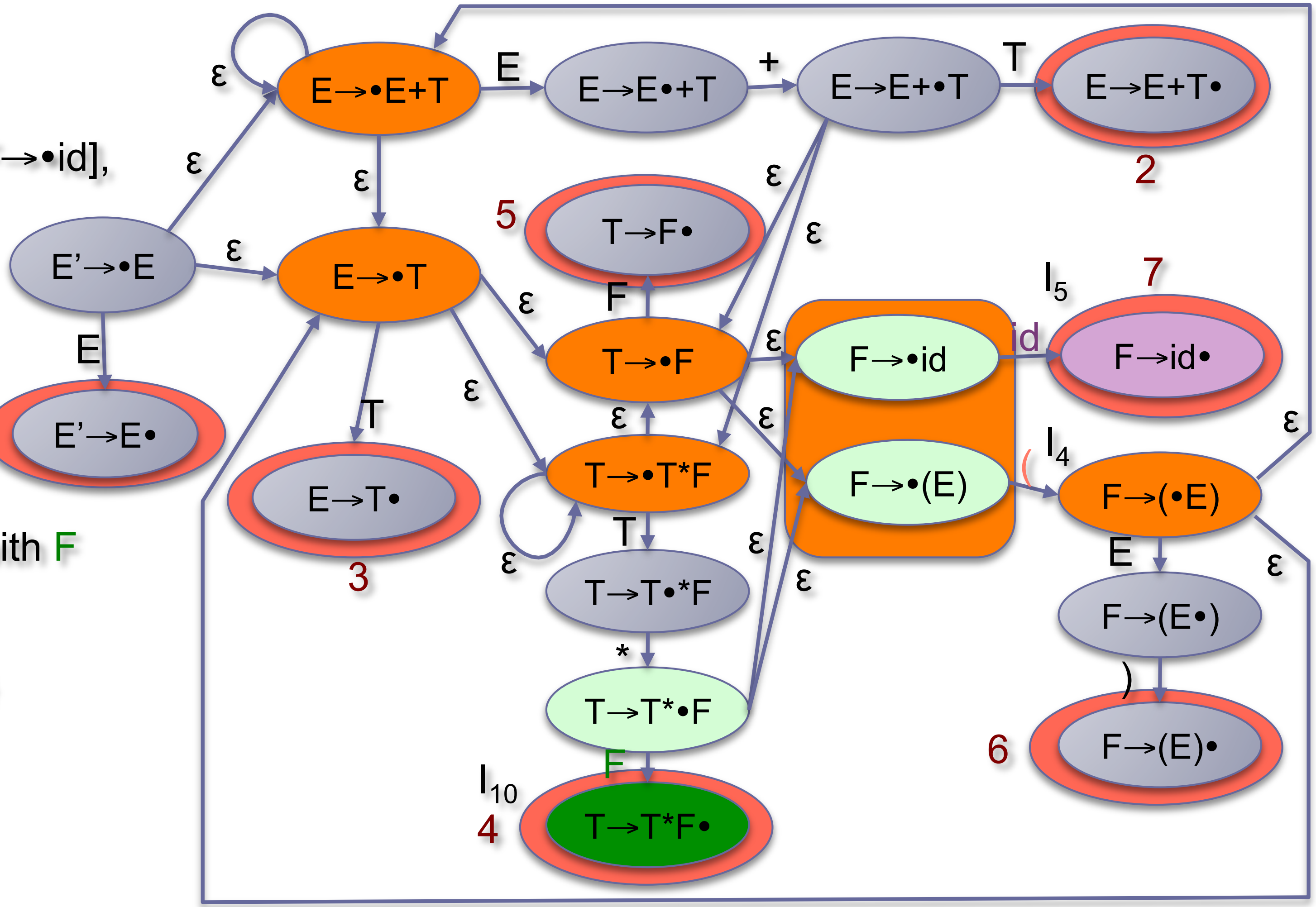
SLR grammars- example 9

transitions from

I_7={ [T→T*•F],[F→•id],
[F→•(E)]}

The transition with F defines

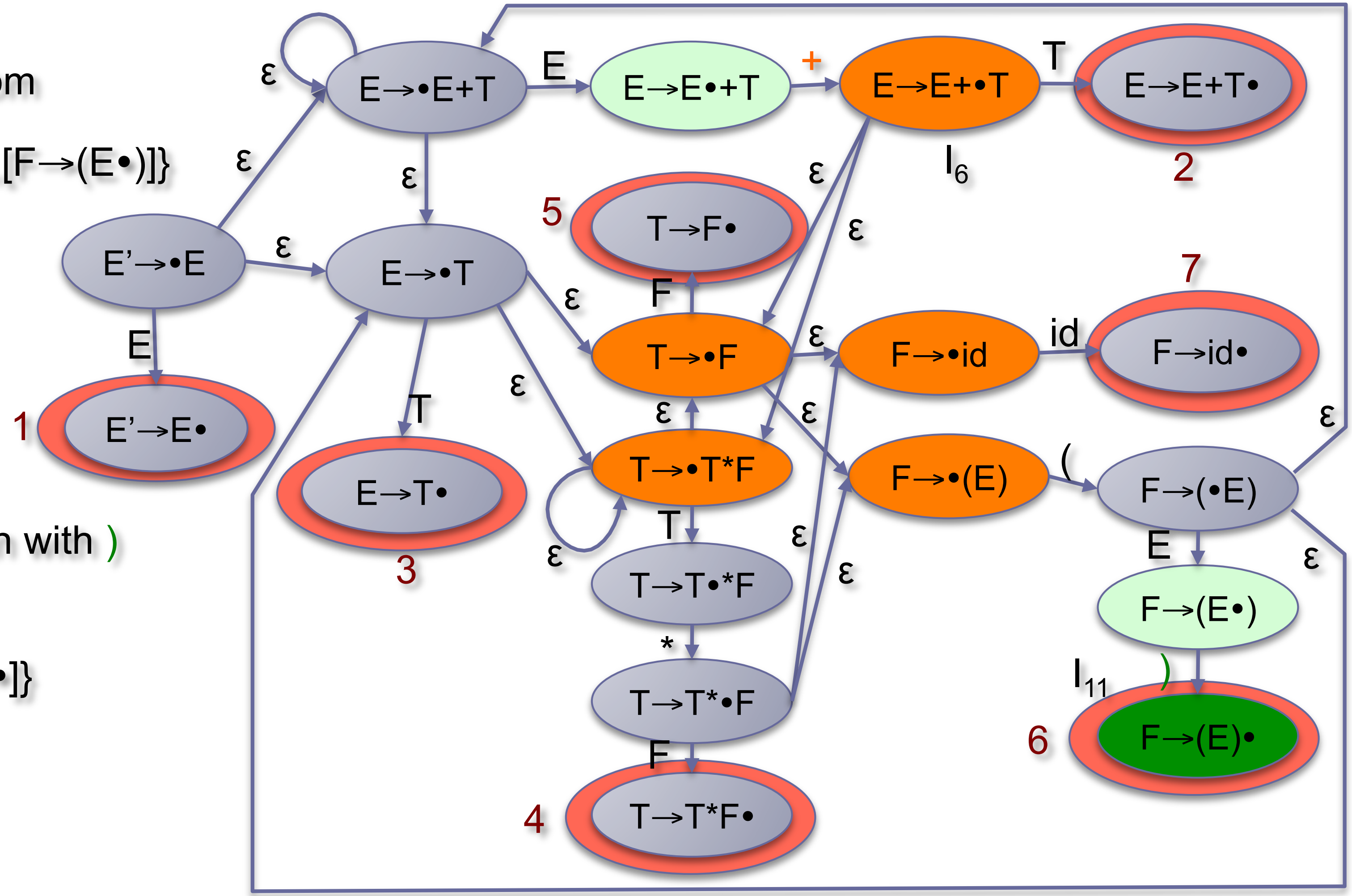
I_{10}={ [T→T*F•]}



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

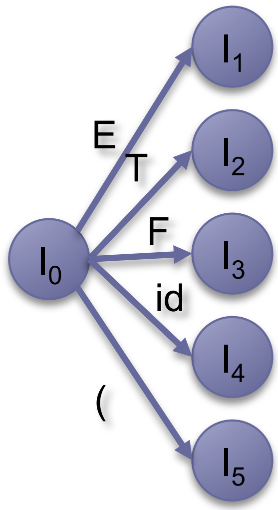
- Given the deterministic automaton that recognizes the prefixes of the right sides of the production rules, it is possible to fill the ACTION and GOTO parse tables
  - The automaton states $C=\{I_0,I_1,\ldots,I_n\}$ correspond to the states 0,1,..n in the scanner
- The actions for state i are defined as follows
  - If $[A\rightarrow\alpha\bullet a\gamma] \in I_i$ 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 $[A\rightarrow\alpha\bullet] \in I_i$ then ACTION[i,a]=REDUCE $A\rightarrow\alpha$ for any terminal symbol a in FOLLOW(A)
  - If $[S'\rightarrow S\bullet] \in I_i$ theb ACTION[i,$]=ACCEPT

Parse tables– filling GOTO

- The GOTO table is filled considering the transitions produced by non terminal symbols for each state
  - If there exists the transition $I_i \rightarrow I_j$ for input $A \in N$ then GOTO[i,A]=j
- The start state contains [S'→•S]
- 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

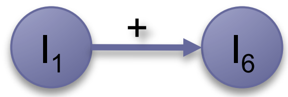
ACTION[0,(] = SHIFT 5

GOTO[0,E] = 1

GOTO[0,T] = 2

GOTO[0,F] = 3

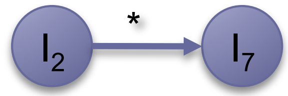
I_1={[E→E•+T],[E'→E•]}



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 languages more general 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

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

  - 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 had much more many states than SLR and LALR parsers
- The automatic generators of CF parsers yield LALR parsers