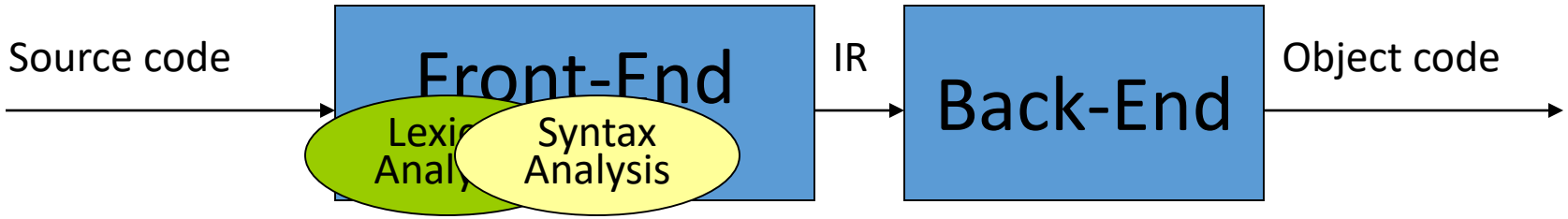
Parsing
(Syntax Analysis)

Introduction to Parsing (Syntax Analysis)



Lexical Analysis:

- Reads characters of the input program and produces tokens.

  But: Are they syntactically correct? Are they valid sentences of the input language?

Outline

- What is syntax analysis?

- **Specification of programming languages**: context-free grammars
- **Parsing context-free languages**: push-down automata

- **Top-down parsing**: **LL(1)** and **recursive-descent** parsing
- **Bottom-up parsing**: **LR**-parsing

Grammars

- Every programming language has **precise grammar rules** that describe **the syntactic structure of well-formed programs**
  - E.g., In C, the rules state how functions are made out of parameter lists, declarations, and statements; how statements are made of expressions, etc.

- Grammars are easy to understand, and **parsers for programming languages** can be **constructed automatically from certain classes of grammars**

- Context-free grammars are usually used for ***syntax specification*** of ***programming languages***

What is syntax analysis/parsing

- A parser for a grammar of a programming language
  - **verifies** that the string of tokens for a program in that language can indeed be generated from that grammar
  - **reports** any syntax errors in the program
  - **constructs** a parse tree representation of the program (not necessarily explicit)
  - usually calls the lexical analyzer to supply a token to it when necessary
  - could be hand-written or automatically generated
  - is based on context-free grammars

- Pushdown automata are machines recognizing context-free languages (like FSA for RL)

Context Free Grammars

- A CFG is denoted as $G = (N, T, P, S)$
  - N: Finite set of non-terminals
  - T: Finite set of terminals
  - $S \in N$: The start symbol
  - P: Finite set of productions, each of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$
- Usually, only P is specified and the first production corresponds to that of the start symbol
- Examples

(1)
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow id$

(2)
$S \rightarrow 0S0$
$S \rightarrow 1S1$
$S \rightarrow 0$
$S \rightarrow 1$
$S \rightarrow \epsilon$

(3)
$S \rightarrow aSb$
$S \rightarrow \epsilon$

(4)
$S \rightarrow aB \mid bA$
$A \rightarrow a \mid aS \mid bAA$
$B \rightarrow b \mid bS \mid aBB$

Derivations

- $E \Rightarrow^{E \to E+E} E + E \Rightarrow^{E \to id} id + E \Rightarrow^{E \to id} id + id$ is a derivation of the terminal string $id + id$ from E
- In a derivation, a production is applied at each step, to replace a nonterminal by the right-hand side of the corresponding production
- In the above example, the productions $E \to E + E$, $E \to id$, and $E \to id$, are applied at steps 1,2, and, 3 respectively
- The above derivation is represented in short as, $E \Rightarrow^* id + id$, and is read as S **derives** $id + id$

Context Free Languages

- Context-free grammars generate context-free languages (grammar and language resp.)
- The *language generated by G*, denoted $L(G)$, is
  $L(G) = \{w \mid w \in T^*, \text{ and } S \Rightarrow^* w\}$
  i.e., a string is in $L(G)$, if
  1. the string consists solely of terminals
  2. the string can be derived from S

- A string $\alpha \in (N \cup T)^*$ is a **sentential form** if $S \Rightarrow^* \alpha$
- Two grammars G_1 and G_2 are equivalent, if $L(G_1) = L(G_2)$

Examples

(1)

$E \to E + E$

$E \to E * E$

$E \to (E)$

$E \to id$

(2)

$S \to 0S0$

$S \to 1S1$

$S \to 0$

$S \to 1$

$S \to \epsilon$

Examples

(1)

$E \to E + E$

$E \to E * E$

$E \to (E)$

$E \to id$

$L(G_1)$ = Set of all expressions with +, *, names, and balanced ’(’ and ’)’

(2)

$S \to 0S0$

$S \to 1S1$

$S \to 0$

$S \to 1$

$S \to \epsilon$

$L(G_2)$ = Set of palindromes over 0 and 1

Examples (2)

(3)

$S \rightarrow aSb$

$S \rightarrow \epsilon$

(4)

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA$

$B \rightarrow b \mid bS \mid aBB$

Examples (2)

(3)

$S \rightarrow aSb$

$S \rightarrow \epsilon$

$L(G_3) = \{a^n b^n \mid n>=1\}$

(4)

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA$

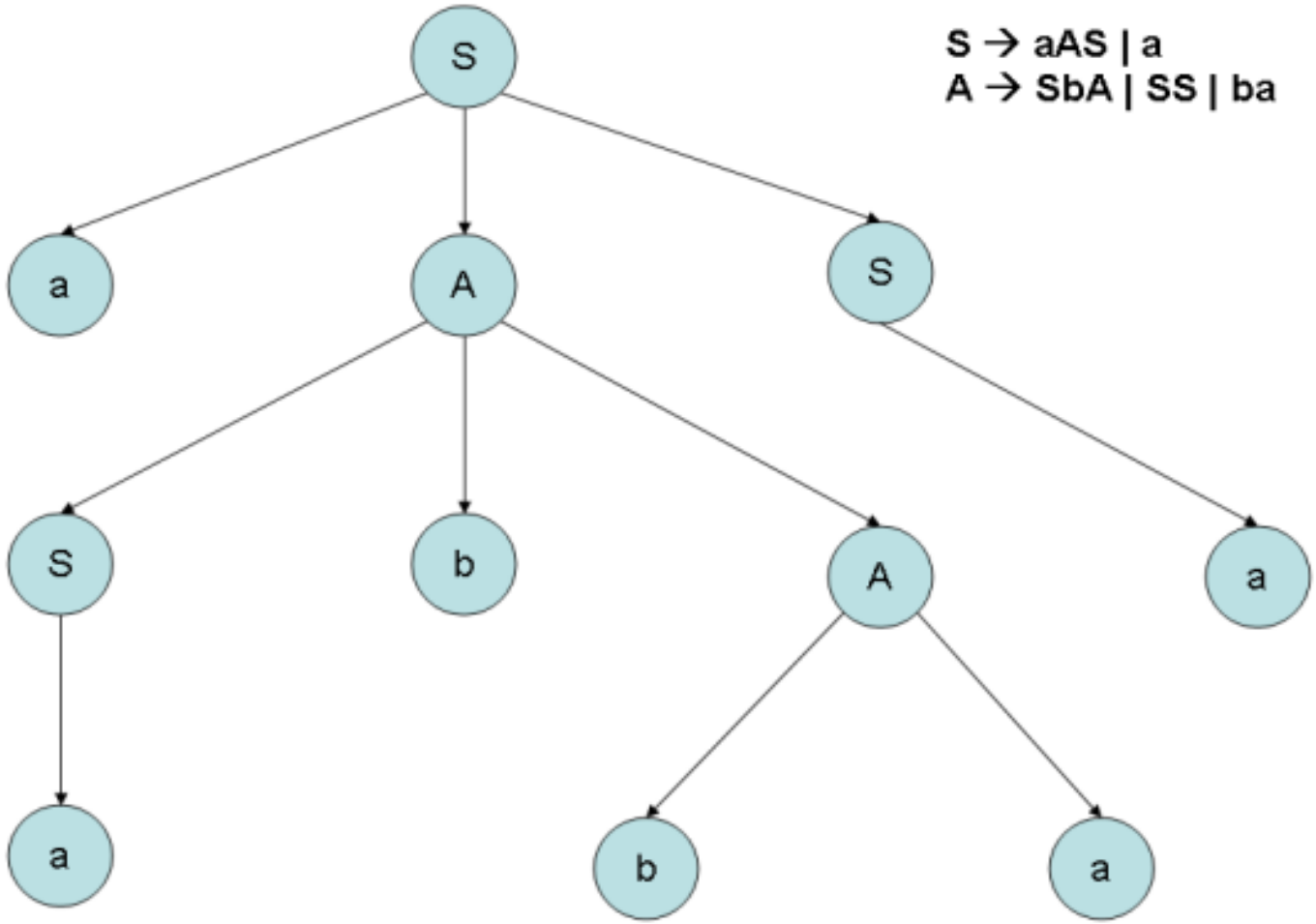
$B \rightarrow b \mid bS \mid aBB$

$L(G_4)$ = {x| x has equal number of a's and b's}

Derivation Trees

- Derivations can be displayed as trees
- The internal nodes of the tree are all nonterminals and the leaves are all terminals
- Corresponding to each internal node A, there exists a production $\in$ P, with the RHS of the production being the list of children of A, read from left to right
- The **yield** of a derivation tree is the list of the labels of all the leaves read from left to right
- If α is the yield of some derivation tree for a grammar G, then $S \Rightarrow^* \alpha$ and conversely

Example: Derivation Tree



S => aAS => aSbAS => aabAS => aabbaS => aabbaa

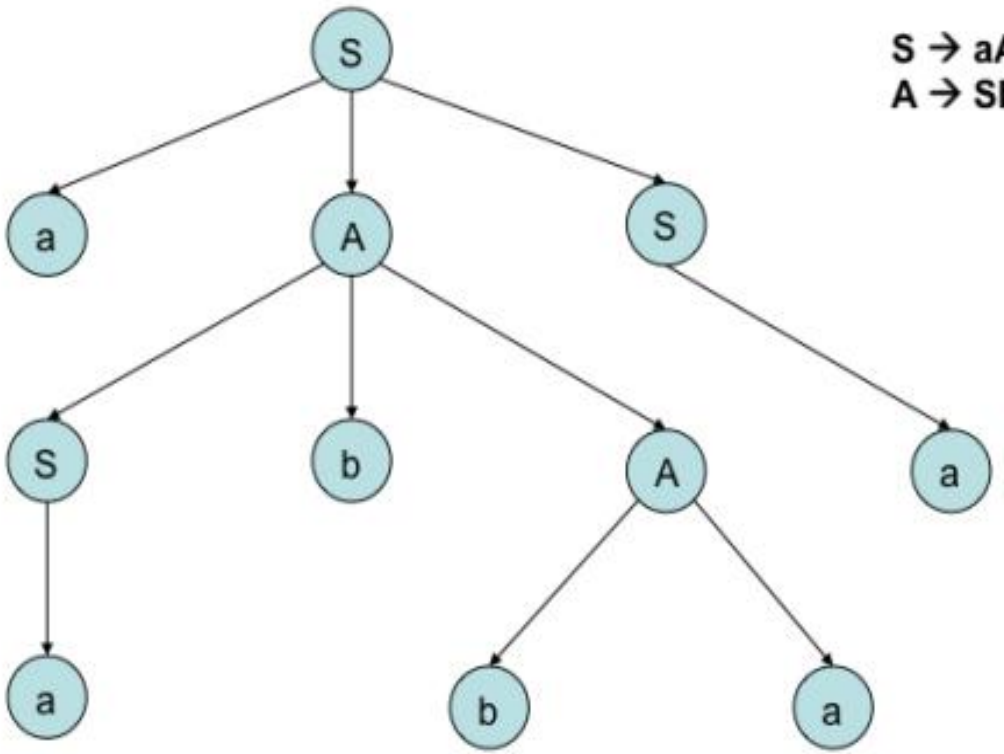
Leftmost and Rightmost Derivations

- Leftmost: at each step in a derivation, a production is applied to the **leftmost nonterminal**.

- Rightmost: at each step in a derivation, a production is applied to the **rightmost nonterminal**.

- If **w ∈ L(G)** for some **G**, then **w** has at least one parse tree and corresponding to a parse tree, **w** has unique *leftmost* and *rightmost* derivations

Example- Leftmost and Rightmost Derivations



S → aAS | a
A → SbA | SS | ba

Leftmost derivation: **S => aAS => aSbAS => aabAS => aabbaS => aabbaa**

Rightmost derivation: **S => aAS => aAa => aSbAa => aSbbaa => aabbaa**

Find parse tree, leftmost, rightmost derivation for: x-2*y

1. Goal → Expr
2. Expr → Expr op Expr
3. | number
4. | id
5. Op → +
6. | -
7. | *
8. | /

Ambiguity

- If some word **w** in ***L(G)*** has two or more parse trees, then **G** is said to be ambiguous

- A **CFL** for which every **G** is ambiguous, is said to be an inherently ambiguous **CFL**

Ambiguity - Example

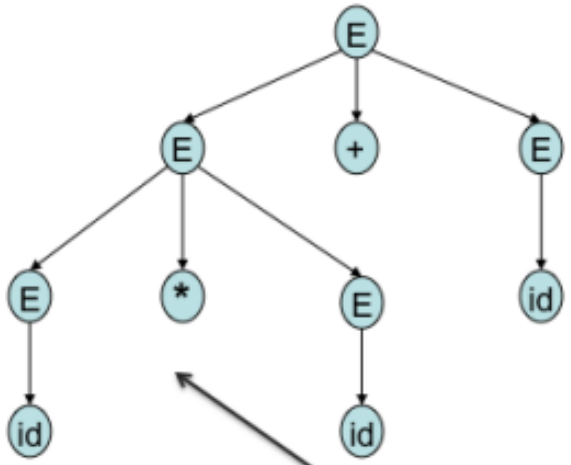
Is this grammar ambiguous?

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

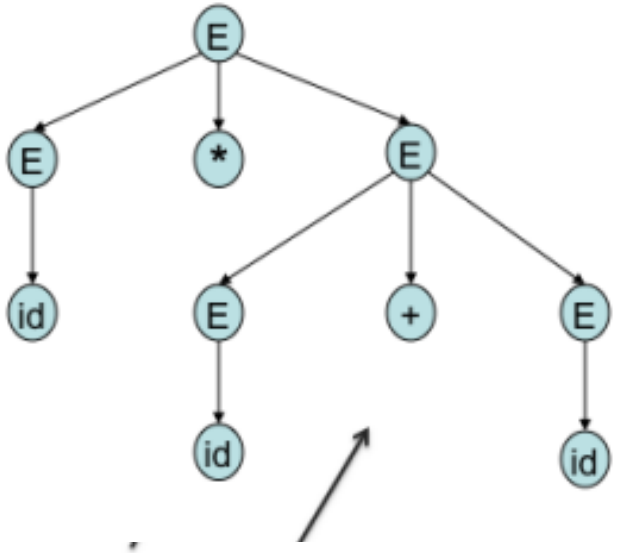
Ambiguity - Example

Is this grammar ambiguous?

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



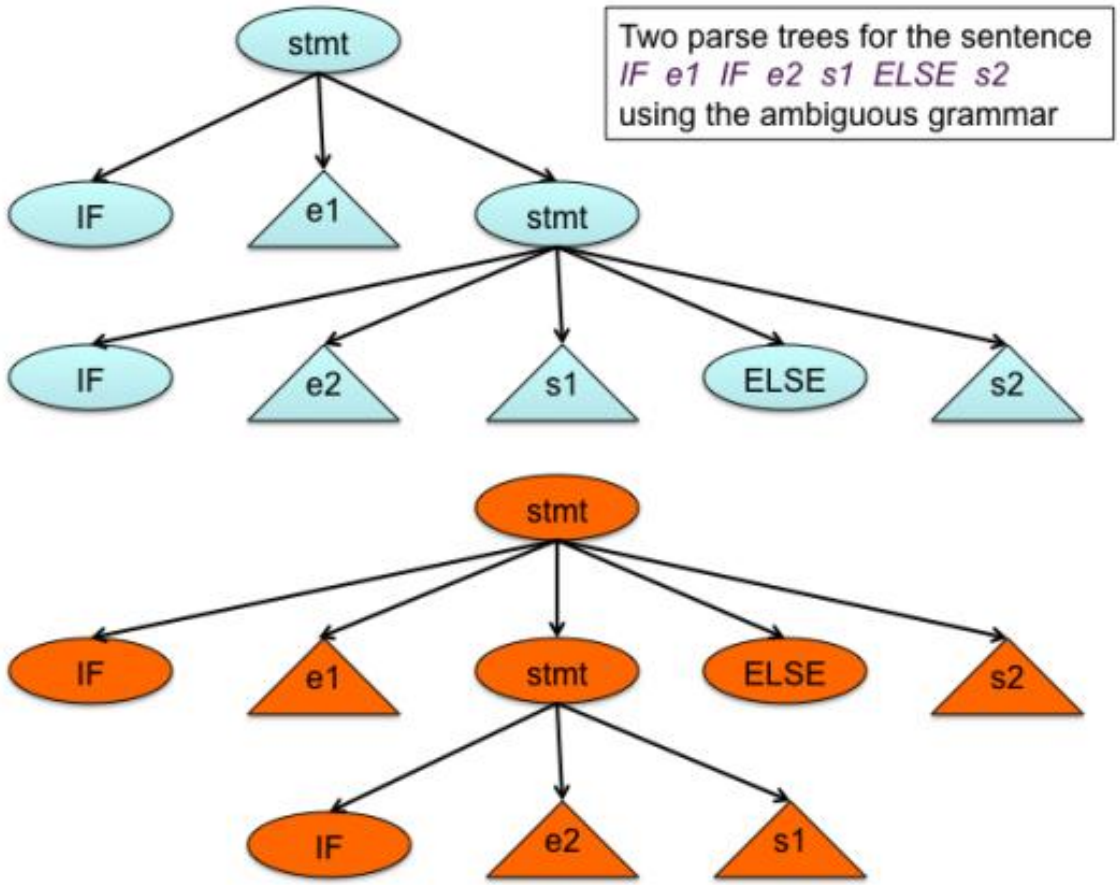
E => E+E => E*E+E => id*E+E => id*id+E => id*id+id



E => E*E => id*E => id*E+E => id*id+E => id*id+id

Ambiguity- Example

$stmt \rightarrow IF\ expr\ stmt | IF\ expr\ stmt\ ELSE\ stmt | other\_stmt$



Eliminating Ambiguity

- For most parsers, desirable that the grammar is made unambiguous (cannot uniquely determine which parse-tree to select otherwise)

- Use *disambiguating rules* (to discard undesirable parse trees), leaving only one tree for each sentence.

Eliminating Ambiguity

- An ambiguous grammar can be rewritten to eliminate ambiguity
- Eg:

$$\begin{aligned} E &\rightarrow E + E \\ &\mid E * E \\ &\mid id \end{aligned}$$

- Parse tree(s) for id+id+id, and for id+id*id
- *Associativity* and *precedence* not taken into account (restrict recursion, introduce levels).

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T*F \mid F \\ F &\rightarrow id \end{aligned}$$

Unambiguous Grammar

Ambiguity

A grammar that produces more than one parse tree for some sentence is ambiguous.

Exercise:

- Stmt $\rightarrow$ if Expr then Stmt | if Expr then Stmt else Stmt | …other…
- What are the derivations of:
  - **if E1 then if E2 then S1 else S2**
- Rewrite the grammar to avoid the problem
- Match each else to innermost unmatched if

Left-Recursive Grammars

- **Definition**: A grammar is ***left-recursive*** if it has a non-terminal symbol ***A***, such that there is a derivation $A \Rightarrow Aa$, for some string a.
- A left-recursive grammar can cause a (top-down) parser to go into an ***infinite loop***.
- **Eliminating left-recursion**: In many cases, it is sufficient to replace

  $A \rightarrow Aa \mid b$ with $A \rightarrow bA'$ and $A' \rightarrow aA' \mid \varepsilon$

- **Example**:

  $Sum \rightarrow Sum+number \mid number$

  would become:

  $Sum \rightarrow number\ Sum'$

  $Sum' \rightarrow +number\ Sum' \mid \varepsilon$

Eliminating Left Recursion

General algorithm: works for non-cyclic, no ε-productions grammars

1. Arrange the non-terminal symbols in order: A_1, A_2, A_3, …, A_n
2. For i=1 to n do
for j=1 to i-1 do
I) replace each production of the form $A_i \rightarrow A_j \gamma$ with
the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$
where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current A_j productions
II) eliminate the immediate left recursion among the A_i

Example- Eliminating Left Recursion

Example:

1. ***Goal*** $\rightarrow$ *Expr*
2. ***Expr*** $\rightarrow$ *Expr + Term*
3. | *Expr – Term*
4. | *Term*
5. ***Term*** $\rightarrow$ *Term * Factor*
6. | *Term / Factor*
7. | *Factor*
8. ***Factor*** $\rightarrow$ *number*
9. | *id*

Applying the transformation:

Expr $\rightarrow$ *Term Expr'*
Expr' $\rightarrow$ *+Term Expr'* | *– Term Expr'* | ε
Term $\rightarrow$ *Factor Term'*
Term' $\rightarrow$ **Factor Term'* | */ Factor Term'* | ε
(***Goal*** $\rightarrow$ *Expr* and ***Factor*** $\rightarrow$ *number | id* remain unchanged)

Left Factoring

- Useful for transforming a grammar to be suitable for (predictive) top-down parsing

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt
\end{array}
$$

- **Example**: On seeing input "if" we cannot immediately tell which production to choose to expand *stmt*.

- When the choice between two A-productions in unclear, we may be able to **defer the decision** until sufficient input is seen (to make correct choice)

- If $\boldsymbol{A} \rightarrow xB_1 + xB_2$ and the input begins with "x" should $\boldsymbol{A}$ be expanded to $\boldsymbol{xB_1}$ or $\boldsymbol{xB_2}$ is unclear
- Left Factored: $\boldsymbol{A} \rightarrow xA'$ $\quad A' \rightarrow B_1 + B_2$

Left Factoring

Algorithm:

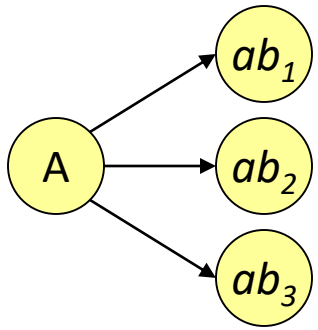
1. For each non-terminal A, find the longest prefix, say α, common to two or more of its alternatives

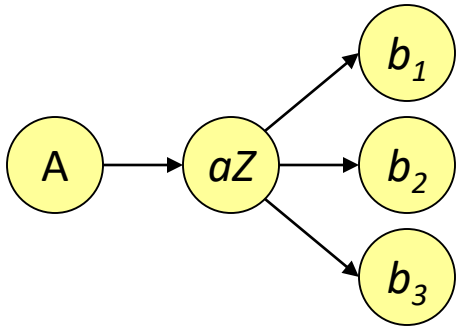
2. if $\alpha \neq \varepsilon$ then replace all the A productions, $A \rightarrow \alpha b_1 | \alpha b_2 | \alpha b_3 | \ldots | \alpha b_n | \gamma$, where γ is anything that does not begin with α, with $A \rightarrow \alpha Z \mid \gamma$ and $Z \rightarrow b_1 | b_2 | b_3 | \ldots | b_n$

Repeat the above until no common prefixes remain

Example: $A \rightarrow ab_1 \mid ab_2 \mid ab_3$ would become $A \rightarrow aZ$ and $Z \rightarrow b_1 | b_2 | b_3$

graphical representation:





Parsing techniques

- **Top-down parsers**:
  - Construct the top node of the tree and then the rest in pre-order. (depth-first)
  - Pick a production & try to match the input; if you fail, **backtrack**.
  - Essentially, we try to find a **leftmost** derivation for the input string (which we scan *left-to-right*).
  - **predictive** parsing (backtrack-free).

- **Bottom-up parsers**:
  - Construct the tree for an input string, beginning at the leaves and working up towards the top (root).
  - Bottom-up parsing, using *left-to-right* scan of the input, tries to construct a **rightmost** derivation in **reverse**.
  - Handle a large class of grammars.

Top-down paring

Top-down Parsing

- Constructing a parse-tree for the input string starting from the root
- At each step of a top-down parser:
  - Determine the production to be applied for a non-terminal (say A)
  - Matching the terminal symbols in the production body with the input string

- Recursive-descent parsing (general form of top-down parsing)
  - **May require backtracking** to find the correct production to be applied

- Predictive parsing
  - Chooses correct production by looking ahead of the input a fixed number of symbols
  - **No backtracking required**

Recursive-Descent Parsing

- Consists of a set of procedures one for each non-terminal.
- Execution begins with the procedure for the start symbol.

```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

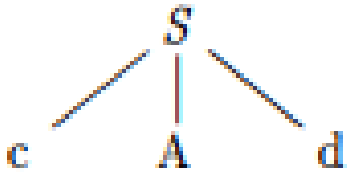
Typical procedure for a non-terminal in a top-down parser

- General recursive-descent parser may require ***backtracking***
- Unique ***A*** production cannot be chosen (must try different productions)
- Failure at line 7 (return to line 1, try another ***A*** production)

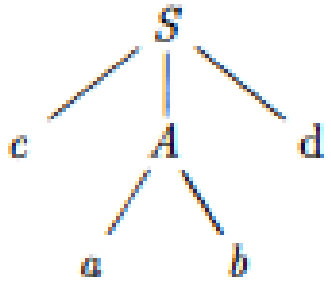
Example- Recursive-Descent Parsing

$$S \rightarrow c\ A\ d$$
$$A \rightarrow a\ b \mid a$$

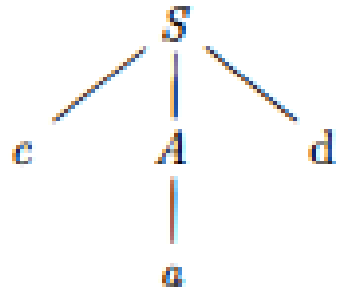
- To construct a parse tree top-down for the input string ***"cad"***
- Input pointer pointing to "***c***" initially



- Advance input pointer to "a"



- Advance input pointer to "d"
- Does not match with "b"
- Reset input pointer to position 2,
- go back, check another alternative for A



- Leaf "a" matches 2nd inp symbol
- Leaf "d" matches 3rd symbol
- **Halt**, announce ***successful***

Example (2)- Recursive-Descent Parsing

Example:

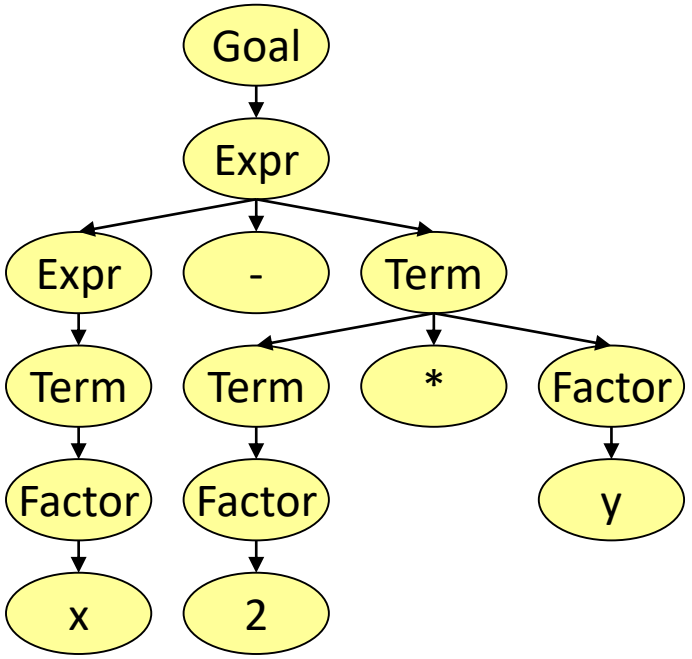
1. *Goal* → *Expr*
2. *Expr* → *Expr + Term*
3. | *Expr – Term*
4. | *Term*
5. *Term* → *Term * Factor*
6. | *Term / Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

Example: Parse *x-2*y*

Example: Parse *x-2*y*

Steps

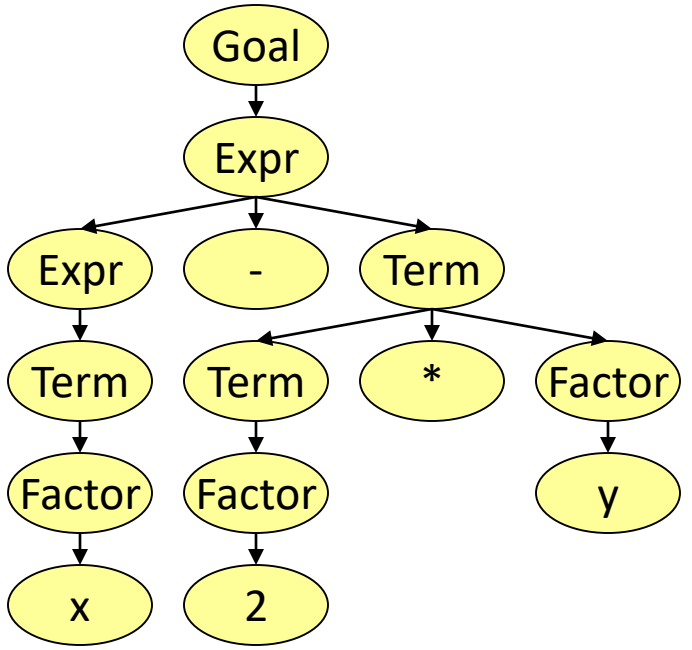
Rule	Sentential Form	Input
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Expr + Term*	| x – 2*y
4	*Term + Term*	| x – 2*y
7	*Factor + Term*	| x – 2*y
9	*id + Term*	| x – 2*y
Fail	*id + Term*	x | – 2*y
Back	*Expr*	| x – 2*y
3	*Expr – Term*	| x – 2*y
4	*Term – Term*	| x – 2*y
7	*Factor – Term*	| x – 2*y
9	*id – Term*	| x – 2*y
Match	*id – Term*	x – | 2*y
7	*id – Factor*	x – | 2*y
9	*id – num*	x – | 2*y
Fail	*id – num*	x – 2 | *y
Back	*id – Term*	x – | 2*y
5	*id – Term * Factor*	x – | 2*y
7	*id – Factor * Factor*	x – | 2*y
8	*id – num * Factor*	x – | 2*y
match	*id – num * Factor*	x – 2* | y
9	*id – num * id*	x – 2* | y
match	*id – num * id*	x – 2*y |



Example: Parse *x-2*y*

Steps

Rule	Sentential Form	Input
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Expr + Term*	| x – 2*y
4	*Term + Term*	| x – 2*y
7	*Factor + Term*	| x – 2*y
9	*id + Term*	| x – 2*y
Fail	*id + Term*	x | – 2*y
Back	*Expr*	| x – 2*y
3	*Expr – Term*	| x – 2*y
4	*Term – Term*	| x – 2*y
7	*Factor – Term*	| x – 2*y
9	*id – Term*	| x – 2*y
Match	*id – Term*	x – | 2*y
7	*id – Factor*	x – | 2*y
9	*id – num*	x – | 2*y
Fail	*id – num*	x – 2 | *y
Back	*id – Term*	x – | 2*y
5	*id – Term * Factor*	x – | 2*y
7	*id – Factor * Factor*	x – | 2*y
8	*id – num * Factor*	x – | 2*y
match	*id – num * Factor*	x – 2* | y
9	*id – num * id*	x – 2* | y
match	*id – num * id*	x – 2*y |



Other choices for expansion are possible:

Rule	Sentential Form	Input
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Expr + Term*	| x – 2*y
2	*Expr + Term + Term*	| x – 2*y
2	*Expr + Term + Term + Term*	| x – 2*y
2	*Expr + Term + Term + … + Term*	| x – 2*y

- Wrong choice leads to non-termination!
- This is a bad property for a parser!
- Parser must make the right choice!

Left-recursive grammar: Wrong choice can lead to non-termination

Where are we?

- We can produce a top-down parser, but:
  - if it picks the wrong production rule it has to backtrack.

- **Idea**: look ahead in input and use context to pick correctly.

- How much ***lookahead*** is needed?
  - Fortunately, most programming language constructs fall into subclasses of context-free grammars that can be parsed with limited lookahead.

Predictive Parsing (First Sets)

- ***FIRST*** sets:
  - For any symbol A, *FIRST(A)* is defined as *the set of terminal symbols that appear as the first symbol of one or more strings derived from A*.

    E.g.:

Goal $\rightarrow$ *Expr*
Expr $\rightarrow$ *Term Expr'*
Expr' $\rightarrow$ *+Term Expr' | – Term Expr' |* ε
Term $\rightarrow$ *Factor Term'*
Term' $\rightarrow$ **Factor Term' | / Factor Term' |* ε
Factor $\rightarrow$ *number | id*

- *FIRST(Expr')={+,-,*ε*}*
- *FIRST(Term')={*,/,*ε*}*
- *FIRST(Factor)={number, id}*

FIRST Sets

If α is any string of grammar symbols ($\alpha \in (N \cup T)^*$), then
$FIRST(\alpha) = \{a \mid a \in T, \text{ and } \alpha \Rightarrow^* ax, x \in T^*\}$
$FIRST(\epsilon) = \{\epsilon\}$

Computation of FIRST

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) = $\{X\}$.
2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), ..., FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$), and so on.
3. If $X \to \epsilon$ is a production, then add ϵ to FIRST(X).

Examples: Computing First Set

Consider the following grammar

$S' \rightarrow S\$,\ S \rightarrow aAS \mid c,\ A \rightarrow ba \mid SB,\ B \rightarrow bA \mid S$

- FIRST (S') = ?
- FIRST (A) = ?

Examples: Computing First Set

Consider the following grammar

$S' \rightarrow S\$,\ S \rightarrow aAS \mid c,\ A \rightarrow ba \mid SB,\ B \rightarrow bA \mid S$

- FIRST (S') = ?
- FIRST (A) = ?

$FIRST(S') = FIRST(S) = \{a, c\}$ because
$S' \Rightarrow S\$ \Rightarrow \underline{c}\$$, and $S' \Rightarrow S\$ \Rightarrow \underline{a}AS\$ \Rightarrow \underline{a}baS\$ \Rightarrow \underline{a}bac\$$

$FIRST(A) = \{a, b, c\}$ because
$A \Rightarrow \underline{b}a$, and $A \Rightarrow SB$, and therefore all symbols in $FIRST(S)$ are in $FIRST(A)$

FOLLOW

For non-terminal **A**;
Follow(A) is the set of terminals **a**
that can appear immediately to the right of **A** in some ***sentential form***.

If A is any nonterminal, then

$$FOLLOW(A) = \{a \mid S \Rightarrow^* \alpha A a \beta,\ \alpha, \beta \in (N \cup T)^*,\ a \in T \cup \{\$\}\}$$

Examples: Computing Follow Set

Consider the following grammar

$S' \rightarrow S\$,\ S \rightarrow aAS \mid c,\ A \rightarrow ba \mid SB,\ B \rightarrow bA \mid S$

- Follow(S) = ?
- Follow(A) = ?

Examples: Computing Follow Set

Consider the following grammar

$S' \to S\$,\ S \to aAS \mid c,\ A \to ba \mid SB,\ B \to bA \mid S$

- Follow(S) = ?
- Follow(A) = ?

$FOLLOW(S) = \{a, b, c, \$\}$ because
$S' \Rightarrow \underline{S}\$$,
$S' \Rightarrow^* aAS\$ \Rightarrow a\underline{S}BS\$ \Rightarrow aS\underline{b}AS\$$,
$S' \Rightarrow^* aSBS\$ \Rightarrow a\underline{S}SS\$ \Rightarrow aS\underline{a}ASS\$$,
$S' \Rightarrow^* a\underline{S}SS\$ \Rightarrow aS\underline{c}S\$$

$FOLLOW(A) = \{a, c\}$ because
$S' \Rightarrow^* a\underline{A}S\$ \Rightarrow aA\underline{a}AS\$$,
$S' \Rightarrow^* a\underline{A}S\$ \Rightarrow aA\underline{c}$

Predictive Parsing

- **Basic idea**:
  - For any production $A \rightarrow \alpha \mid \beta$ we would like to have a distinct way of choosing the correct production to expand.
- ***FIRST* sets**:
  - For any symbol A, ***FIRST(A)*** is defined as the set of terminal symbols that appear as the first symbol of one or more strings derived from A.

    E.g. (grammar in prev. slide): ***FIRST***$(Expr')=\{+,-,\varepsilon\}$, ***FIRST***$(Term')=\{*,/,\varepsilon\}$, ***FIRST***$(Factor)=\{number, id\}$
- **The LL(1) property**:
  - If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like to have: $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$.
  - This would allow the parser to make a correct choice with a ***lookahead*** of exactly ***one*** symbol!

Parsing- LL(1) grammars

- Parsing is the process of constructing a parse tree for a sentence generated by a given grammar

- Predictive parsing using LL(1) grammars (top-down parsing method)

LL(1) grammars

- Predictive parsers (that require no back-tracking) can be constructed for a class of grammars called **LL(1)** grammars
  - L: Scanning input from **left to right**
  - L: produce a **left-most derivation**
  - "1": use **1 input symbol as lookahead** at each step

- LL(1) grammars covers most programming constructs

- No ambiguous or left-recursive grammar can be LL(1)

LL(1) grammars

- A grammar **G** is **LL(1)** *iff* whenever A → α and A → β are two distinct productions of **G** the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

LL(1) grammars

- A grammar **G** is **LL(1)** *iff* whenever **A → α** and **A → β** are two distinct productions of **G** the following conditions hold:

  Condition 1 &2: **First(α)** and **First(β)** are disjoint sets

  Condition 3: if ε is in **First(β)**, then **First(α)** and **Follow(A)** are disjoint sets (likewise if ε in First(**α**))

- For LL(1) grammars, predictive parsers can be constructed

  (by looking only at current input symbol, production to apply for a non-terminal can be selected)

Predictive Parsing Table

- Predictive parsing table M[A,a], a two-dimensional array
  - **A**: non-terminal
  - **a**: terminal or **$** (input endmark)

Idea:

- Choose production **A → α** if the next input symbol is in **First(α)**
- Only issue is when **α= ε** (or ε can be derived from α)
  - Again choose **A → α** if the current input symbol is in **Follow(A)**, or if **$** is reached and **$** is in **Follow(A)**

Constructing Predictive Parsing table (using First/Follow)

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal "**a**" in **FIRST(α)**, add **A-> α** to **M[A,a]**

2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

Example- Predictive Parsing Table

$E \rightarrow T\,E'$

$E' \rightarrow +T\,E' \mid \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *F\,T' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Non Terminal	id	+	*	(	)	$
E	$E \rightarrow T\,E'$			$E \rightarrow T\,E'$		
E’						
T						
T’						
F						

Example- Predictive Parsing Table

$E \rightarrow T\,E'$

$E' \rightarrow +T\,E' \mid \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *F\,T' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Non Terminal	id	+	*	(	)	$
E	$E \rightarrow T\,E'$			$E \rightarrow T\,E'$		
E’		$E' \rightarrow +T\,E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T						
T’						
F						

Example- Predictive Parsing Table

$E \rightarrow T\ E'$

$E' \rightarrow +T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

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

$F \rightarrow (E) \mid id$

Non Terminal	id	+	*	(	)	$
E	$E \rightarrow T\ E'$			$E \rightarrow T\ E'$		
E’		$E' \rightarrow +T\ E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F\ T'$			$T \rightarrow F\ T'$		
T’						
F						

Example- Predictive Parsing Table

$E \rightarrow T\,E'$

$E' \rightarrow +T\,E' \mid \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *F\,T' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Non Terminal	id	+	*	(	)	$
E	$E \rightarrow T\,E'$			$E \rightarrow T\,E'$		
E’		$E' \rightarrow +T\,E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F\,T'$			$T \rightarrow F\,T'$		
T’		$T' \rightarrow \varepsilon$	$T' \rightarrow *F\,T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F						

Example- Predictive Parsing Table

$E \rightarrow T\ E'$
$E' \rightarrow +T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *F\ T' \mid \varepsilon$
$F \rightarrow (E) \mid id$

Non Terminal	id	+	*	(	)	$
E	$E \rightarrow T\ E'$			$E \rightarrow T\ E'$		
E’		$E' \rightarrow +T\ E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F\ T'$			$T \rightarrow F\ T'$		
T’		$T' \rightarrow \varepsilon$	$T' \rightarrow *F\ T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Example- Predictive Parsing Table

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

- **Non-blanks indicate production to use to expand a non-terminal**
- **Blanks are error entries**

Parsing table, LL(1) grammar

- For every **LL(1)** grammar, each parse table entry uniquely identifies a production or signals an error

- For any grammar **G**, parse table can be constructed (**M** may have entries multiply defined)
  - E.g., if G is ambiguous or left-recursive, there will be at least one multiply defined entry in M !!

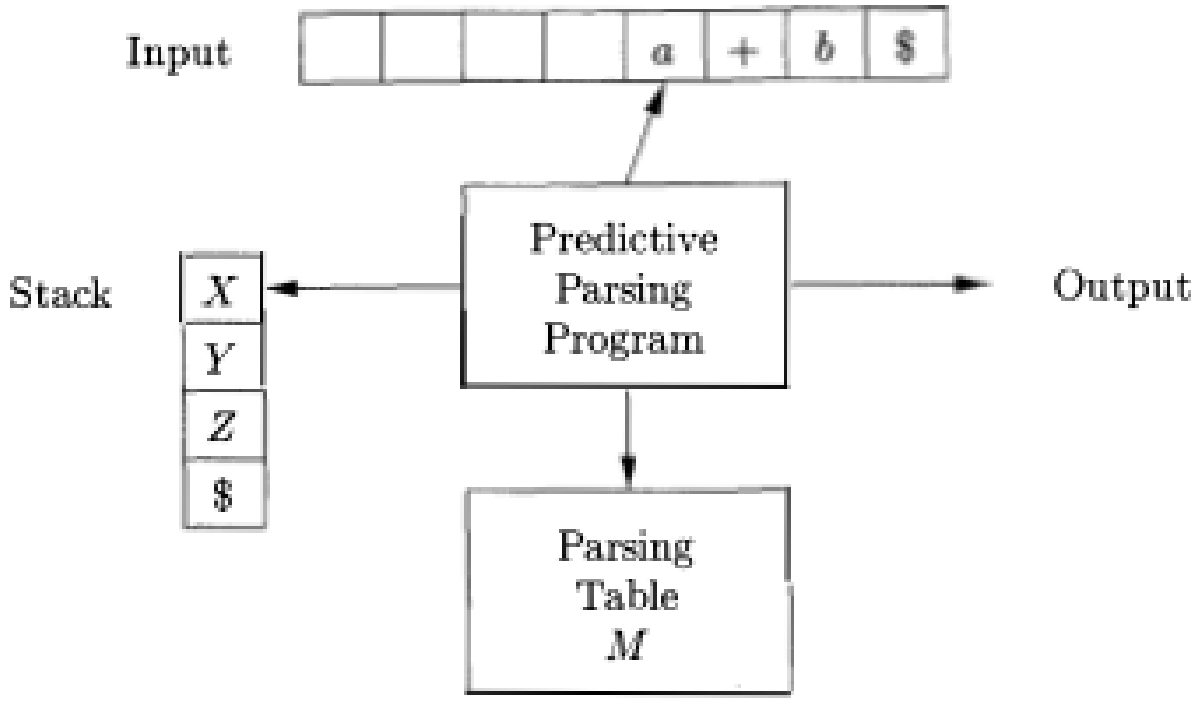
Non-recursive predictive parsing

- Non-recursive predictive parser built by maintaining an ***explicit stack***
- Parser mimics a *leftmost derivation*
- Let "***w***" denote the sequence of input that has been matched so far. Then the stack holds a sequence of grammar symbols **α** such that:

$$S \overset{*}{\underset{lm}{\Rightarrow}} w\alpha$$

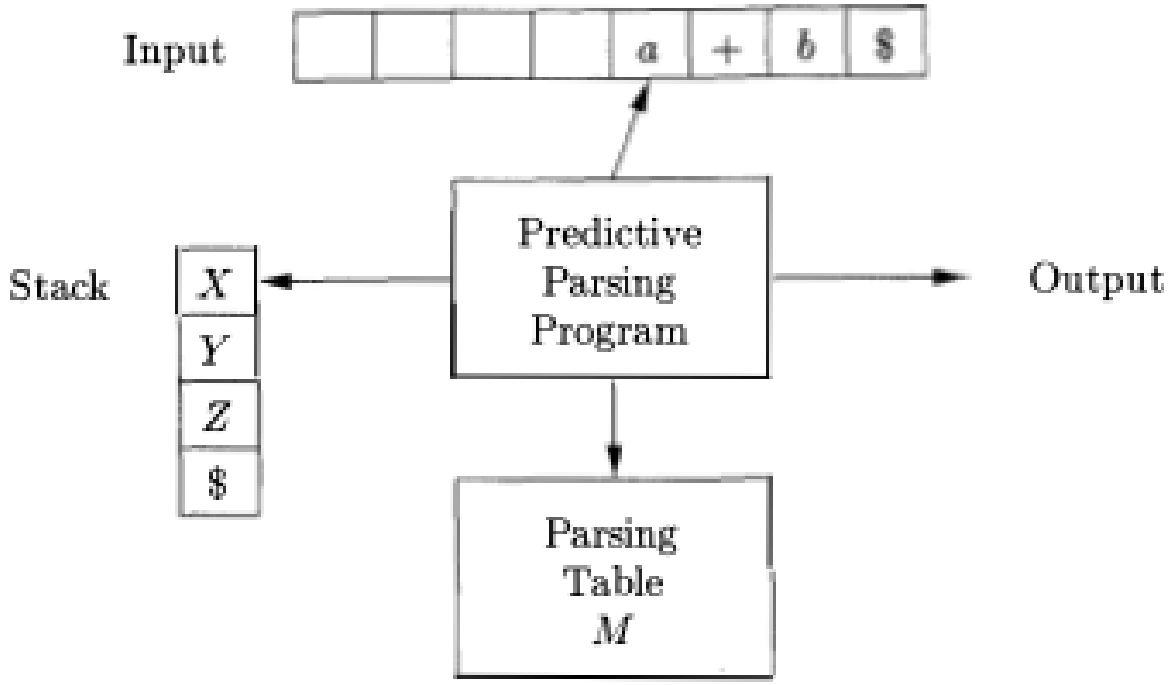
Non-recursive predictive parsing

- **Table driven parser**
  - **Input buffer** (input to be parsed followed by an end marker $)
  - **Stack** containing a sequence of grammar symbols ($ marks bottom of stack)
  - **Parsing table** (constructed using the prev. algo)



Non-recursive predictive parsing

- Take "**x**" symbol on top of the stack, and the current input "**a**".
  - **X is non-terminal**: Apply rule as per entry **M[X,a]**
  - **X is terminal**: Check for match between **X** and current input symbol **a**



Predictive parsing algorithm

- Behavior of parser can be described in terms of its configurations (***stack content*** and ***remaining input***)

- **Initial configuration**: stack containing S (start symbol) above **$**, remaining input is complete input (w$)

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X is a ) pop the stack and advance ip;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1 Y2 ··· Yk ) {
                output the production X → Y1 Y2 ··· Yk;
                pop the stack;
                push Yk, Yk−1, ..., Y1 onto the stack, with Y1 on top;
        }
        set X to the top stack symbol;
}
```

Moves made by predictive parser for: id+id*id

$E \rightarrow T\ E'$
$E' \rightarrow +T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *F\ T' \mid \varepsilon$
$F \rightarrow (E) \mid id$

ProcessedInput	StackContent	RemainingInput	Action
	E$	id+id*id	

Non Terminal	id	+	*	(	)	$
E	$E \rightarrow T\ E'$			$E \rightarrow T\ E'$		
E’		$E' \rightarrow +T\ E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F\ T'$			$T \rightarrow F\ T'$		
T’		$T' \rightarrow \varepsilon$	$T' \rightarrow *F\ T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Moves made by predictive parser for: **id+id*id**

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+\ TE'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	output $E' \to +\ TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id} +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$*\ FT'E'\$$	$* \mathbf{id}\$$	output $T' \to *\ FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Recap

- Ambiguity, left-recursion, left-factoring

- **Top-down parsing**
  - Recursive-Descent Parsing (with backtracking)
  - Predictive parsing – LL(1)

Bottom-Up Parsing

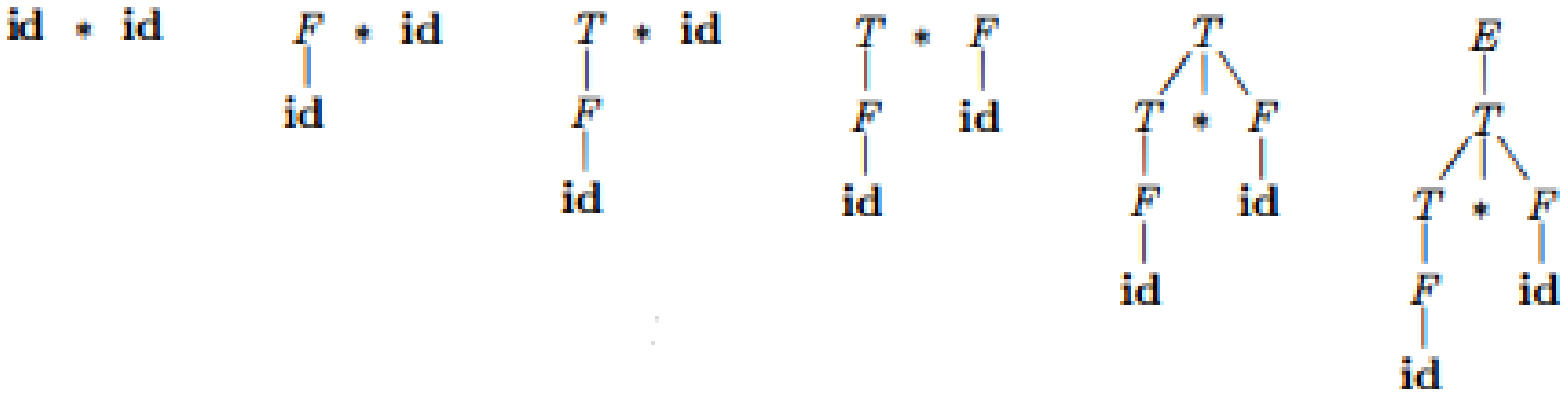
Bottom-Up Parsing

- Begin at the leaves, build the parse tree in small segments, combine the small trees to make bigger trees, until the root is reached

- This process is called *reduction* of **the sentence to the start symbol** of the grammar
- **Reduction**: reverse of a step in a derivation
- One of the ways of "reducing" a sentence is to follow the rightmost derivation of the sentence in ***reverse***
  - **Shift-Reduce parsing** implements such a strategy
  - It uses the concept of a ***handle*** to detect when to perform reductions
- Bottom-up parsing during left-right scan of the input constructs a right-most derivation in reverse.

A bottom-up parse for **id*id**

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



Handle

- Informally, a handle is
  - a substring that matches the body of a production, and
  - whose **reduction** represents **one step** along the **reverse of a right-most derivation**

Example: Handle (during a parse of $\mathbf{id_1 * id_2}$)

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

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id_1} * \mathbf{id_2}$	$\mathbf{id_1}$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id_2}$	F	$T \rightarrow F$
$T * \mathbf{id_2}$	$\mathbf{id_2}$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

NOTE: The leftmost substring that matches the body of some production need not be a handle
(e.g, replacing **T** with **E** in $\mathbf{T*id_2}$ we get $\mathbf{E*id_2}$ which cannot be derived from the start symbol).

Handle (Formally)

- **Handle**: A handle of a right sentential form **γ**, is a production **A → β** and a **position in γ**, where the string **β** may be found and replaced by **A**, to produce the previous right sentential form in a rightmost derivation of **γ**

- That is, if $S \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta w$, then **A → β** in the position following **α** is a handle of **αβw**

- A handle will always eventually appear on the top of the stack (never submerged inside the stack)
- In Shift-Reduce parsing, we locate the handle and reduce it by the LHS of the production repeatedly, to reach the start symbol

- These reductions, in fact, trace out a **rightmost derivation of the sentence in reverse**. This process is called handle pruning

Example (1)

S → aAcBe,
A → Ab | b,
B → d

For the string "***abbcde***" provide rightmost derivation marked with handles.

S ⇒ aAcBe (aAcBe, S → aAcBe)
⇒ aAc**d**e (d, B → d)
⇒ a**Ab**cde (Ab, A → Ab)
⇒ a**b**bcde (b, A → b)

The handle is unique if the grammar is unambiguous!

Example (2)

S → aAS | c,
A → ba | SB, B → bA | S

For the string "***acbbac***" provide rightmost derivation marked with handles.

Example (2)

S → aAS | c,
A → ba | SB, B → bA | S

For the string "***acbbac***" provide rightmost derivation marked with handles.

S ⇒ aAS (aAS, S → aAS)
⇒ aAc (c, S → c)
⇒ aSBc (SB, A → SB)
⇒ aSbAc (bA, B → bA)
⇒ aSbbac (ba, A → ba)
⇒ acbbac (c, S → c)

Example (3)

$$E \rightarrow E + E,\ E \rightarrow E * E,\ E \rightarrow (E),\ E \rightarrow \text{id}$$

- For the string "*id + id * id*" is the right-most derivation unique?

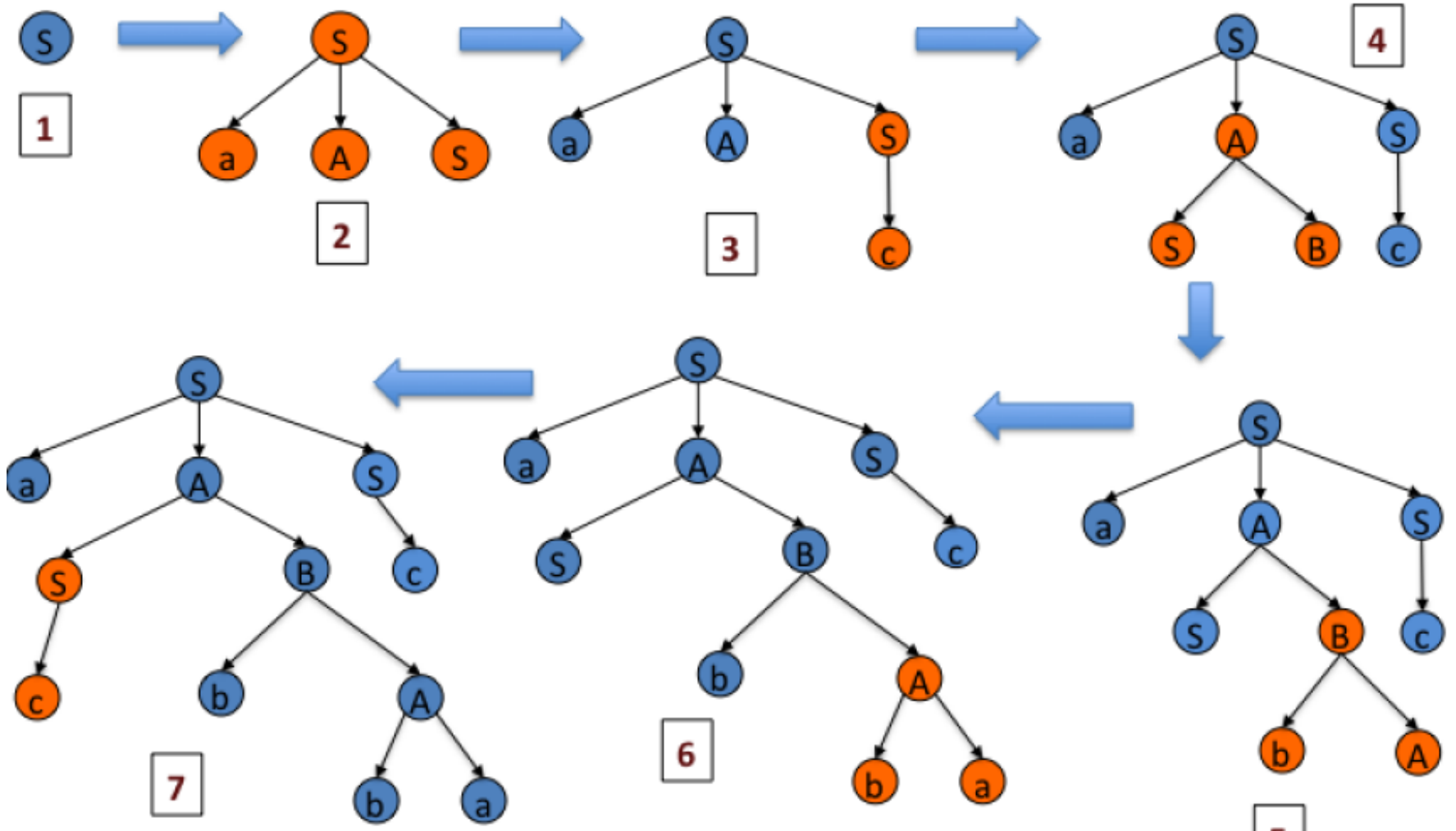
Right-most derivation

S → aAS | c
A → ba | SB
B → bA | S

Rightmost derivation of the string *acbbac*

S => aAS => aAc => aSBc => aSbAc => aSbbac => acbbac

1 2 3 4 5 6 7



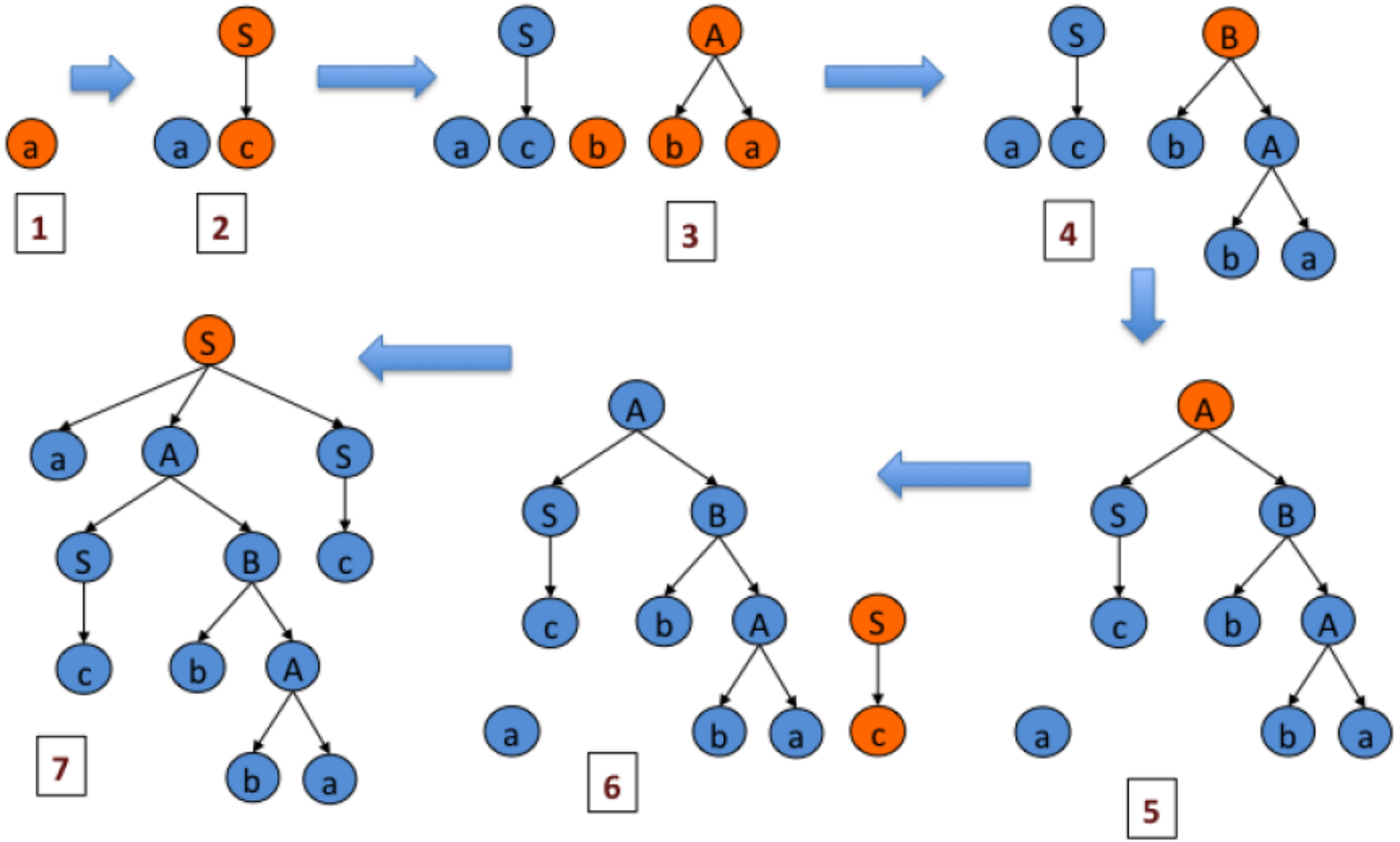
Rightmost Derivation and Bottom-up Parsing

S → aAS | c
A → ba | SB
B → bA | S

Rightmost derivation of the string *acbbac* in reverse

S <= a**A**S <= a**A**c <= aS**B**c <= aS**b**Ac <= a**S**bbac <= acbbac

7 6 5 4 3 2 1



Shift-Reduce Parsing

- Form of bottom-up parsing
- Stack: holds grammar symbols ($: bottom of stack)
- Input buffer: holds rest of the input to be parsed ($: end of the input)
- Top of the stack on the right
- Initial configuration:

STACK	INPUT
$	w $

- **While** scanning input from ***left-to-right***:
  - **Shift**: input symbols onto the stack (until ready to reduce)
  - **Reduce**: a string on top of the stack reduced to the head of appropriate production

- Repeat **Shift-reduce** until the stack and input is empty, or until an error is detected.

Implementing a shift-reduce parser

```
push $ onto the stack
token = next_token()
repeat
   if the top of the stack is a handle A → β
            then /* reduce β to A */
                     pop the symbols of β off the stack
                     push A onto the stack
            elseif (token != eof) /* eof: end-of-file = end-of-input */
                     then /* shift */
                              push token
                              token=next_token()
                     else /* error */
                              call error_handling()
until (top_of_stack == Goal && token==eof)
```

Configuration of a Shift-Reduce Parser (id_1*id_2)

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

STACK	INPUT	ACTION
\$	$\mathbf{id}_1 * \mathbf{id}_2$ \$	shift
\$ $\mathbf{id}_1$	$* \mathbf{id}_2$ \$	reduce by $F \rightarrow \mathbf{id}$
\$ F	$* \mathbf{id}_2$ \$	reduce by $T \rightarrow F$
\$ T	$* \mathbf{id}_2$ \$	shift
\$ $T *$	$\mathbf{id}_2$ \$	shift
\$ $T * \mathbf{id}_2$	\$	reduce by $F \rightarrow \mathbf{id}$
\$ $T * F$	\$	reduce by $T \rightarrow T * F$
\$ T	\$	reduce by $E \rightarrow T$
\$ E	\$	accept

Example

Configuration of the shift-reduce parser for "***x – 2*y***"

1. *Goal* → *Expr*
2. *Expr* → *Expr + Term*
3. | *Expr – Term*
4. | *Term*
5. *Term* → *Term * Factor*
6. | *Term / Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

Example

Configuration of the shift-reduce parser for "***x – 2*y***"

1. *Goal* → *Expr*
2. *Expr* → *Expr + Term*
3. | *Expr – Term*
4. | *Term*
5. *Term* → *Term * Factor*
6. | *Term / Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

Stack	Input	Handle	Action
$	id – num * id	None	Shift
$ id	– num * id	9,1	Reduce 9
$ Factor	– num * id	7,1	Reduce 7
$ Term	– num * id	4,1	Reduce 4
$ Expr	– num * id	None	Shift
$ Expr –	num * id	None	Shift
$ Expr – num	* id	8,3	Reduce 8
$ Expr – Factor	* id	7,3	Reduce 7
$ Expr – Term	* id	None	Shift
$ Expr – Term *	id	None	Shift
$ Expr – Term * id		9,5	Reduce 9
$ Expr – Term * Factor		5,5	Reduce 5
$ Expr – Term		3,3	Reduce 3
$ Expr		1,1	Reduce 1
$ Goal		none	Accept

What can go wrong?

(think about the highlighted steps in the previous example)

- **Shift/reduce conflicts**: the parser cannot decide whether to shift or to reduce.

  Example: usually due to ambiguous grammars (the dangling-else grammar).

  Solution: a) modify the grammar; b) resolve in favour of a shift.

- **Reduce/reduce conflicts**: the parser cannot decide which of several reductions to make.

  May be difficult to tackle.

Key to efficient bottom-up parsing: the handle-finding mechanism.

Conflicts during Shift-Reduce Parsing

- Shift reduce parsing cannot be used for all context-free grammars
- Suitable for grammars that belong to the class LR
- For non-LR grammars, the parser may reach a config in which it
  - Cannot decide to shift or to reduce (**shift/reduce conflict**)
  - Cannot decide which of the several reductions to apply (**reduce/reduce conflict**)

LR(1) grammars

A grammar is LR(1) if, given a rightmost derivation, we can
(I) isolate the handle of each right-sentential form, and
(II) determine the production by which to reduce,
by scanning the sentential form from left-to-right, going at most **1** symbol beyond the right-end of the handle.

- LR(1) grammars are widely used to construct (automatically) efficient and flexible parsers:
  - LR grammars are the most general grammars parsable by a non-backtracking, shift-reduce parser.
  - LR parsers detect an error as soon as possible in a left-to-right scan of the input.

L stands for left-to-right scanning of the input; **R** for constructing a rightmost derivation in reverse; **1** for the number of input symbols for lookahead.

LR Parsing

- **LR(k)** - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens
  - k = 0, 1 are of practical interest
- LR parsers are also automatically generated using parser generators
- LR grammars are a subset of CFGs for which LR parsers can be constructed
- LR(1) grammars can be written easily for practically all programming language constructs for which CFGs can be written
- LR parsing is the most general non-backtracking shift-reduce parsing method
- **LL grammars** are a **strict subset of LR grammars** - an LL(k) grammar is also LR(k), but not vice-versa

LR Parsing: Background

- Read tokens from an input buffer (same as with shift-reduce parsers)
- **Add an extra state information** after each symbol in the stack. The state summarises the information contained in the stack below it. The stack would look like:

  $\$ \; S_0 \; Expr \; S_1 \; - \; S_2 \; num \; S_3$

- Use a table that consists of two parts:
  - **action**[state_on_top_of_stack, input_symbol]: returns one of: **shift** s (push a symbol and a state); **reduce** by a rule; **accept**; **error**.
  - **goto**[state_on_top_of_stack, non_terminal_symbol]: returns a new state to push onto the stack ***after a reduction***.

Skeleton code for an LR Parser

```
Push $ onto the stack
push s0
token=next_token()
repeat
  s=top_of_the_stack  /* not pop! */
  if ACTION[s,token]=='reduce A→b'
      then  pop 2*(symbols_of_b) off the stack
            s=top_of_the_stack  /* not pop! */
            push A; push GOTO[s,A]
      elseif ACTION[s,token]=='shift sx'
                then push token; push sx
                     token=next_token()
                elseif ACTION[s,token]=='accept'
                     then break
                     else report_error
end repeat
report_success
```

Example: the expression grammar

1. *Goal* → *Expr*
2. *Expr* → *Expr + Term*
3. | *Expr – Term*
4. | *Term*
5. *Term* → *Term * Factor*
6. | *Term / Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

STATE	ACTION							GOTO		
	eof	+	–	*	/	num	id	Expr	Term	Factor
0						S 4	S 5	1	2	3
1	Acc	S 6	S 7							
2	R 4	R 4	R 4	S 8	S 9					
3	R 7	R 7	R 7	R 7	R 7					
4	R 8	R 8	R 8	R 8	R 8					
5	R 9	R 9	R 9	R 9	R 9					
6						S 4	S 5		10	3
7						S 4	S 5		11	3
8						S 4	S 5			12
9						S 4	S 5			13
10	R 2	R 2	R 2	S 8	S 9					
11	R 3	R 3	R 3	S 8	S 9					
12	R 5	R 5	R 5	R 5	R 5					
13	R 6	R 6	R 6	R 6	R 6					

*Apply the algorithm to the expression **x-2*y***
The result is the rightmost derivation but …
…no conflicts now: state information makes it fully deterministic!

LR() – Table Generation

LR Grammars

- Consider a rightmost derivation:
  $S \Rightarrow^*_{rm} \phi Bt \Rightarrow_{rm} \phi\beta t$,
  where the production $B \rightarrow \beta$ has been applied
- A grammar is said to be **LR(k)**, if for any given input string, at each step of any rightmost derivation, the handle β can be detected by examining the string $\phi\beta$ and scanning *at most*, first *k* symbols of the unused input string *t*

LR Grammars (Viable Prefix)

- A viable prefix of a sentential form **ϕβt**, where **β** denotes the handle, is any prefix of **ϕβ**.
- A viable prefix cannot contain symbols to the right of the handle

Example: $S \rightarrow E\#,\ E \rightarrow E + T \mid E - T \mid T,\ T \rightarrow id \mid (E)$

$S \Rightarrow E\# \Rightarrow E + T\# \Rightarrow E + (E)\# \Rightarrow E + (T)\#$

$\Rightarrow E + (id)\#$

$E,\ E+,\ E + (,$ and $E + (id,$ are all viable prefixes of the right sentential form $E + (id)\#$

- Viable prefixes characterize the **prefixes of sentential forms that can occur on the stack** of an LR parser

LR Grammars (Viable Prefix)

- **Theorem**: The set of all viable prefixes of all the right sentential forms of a grammar is a regular language

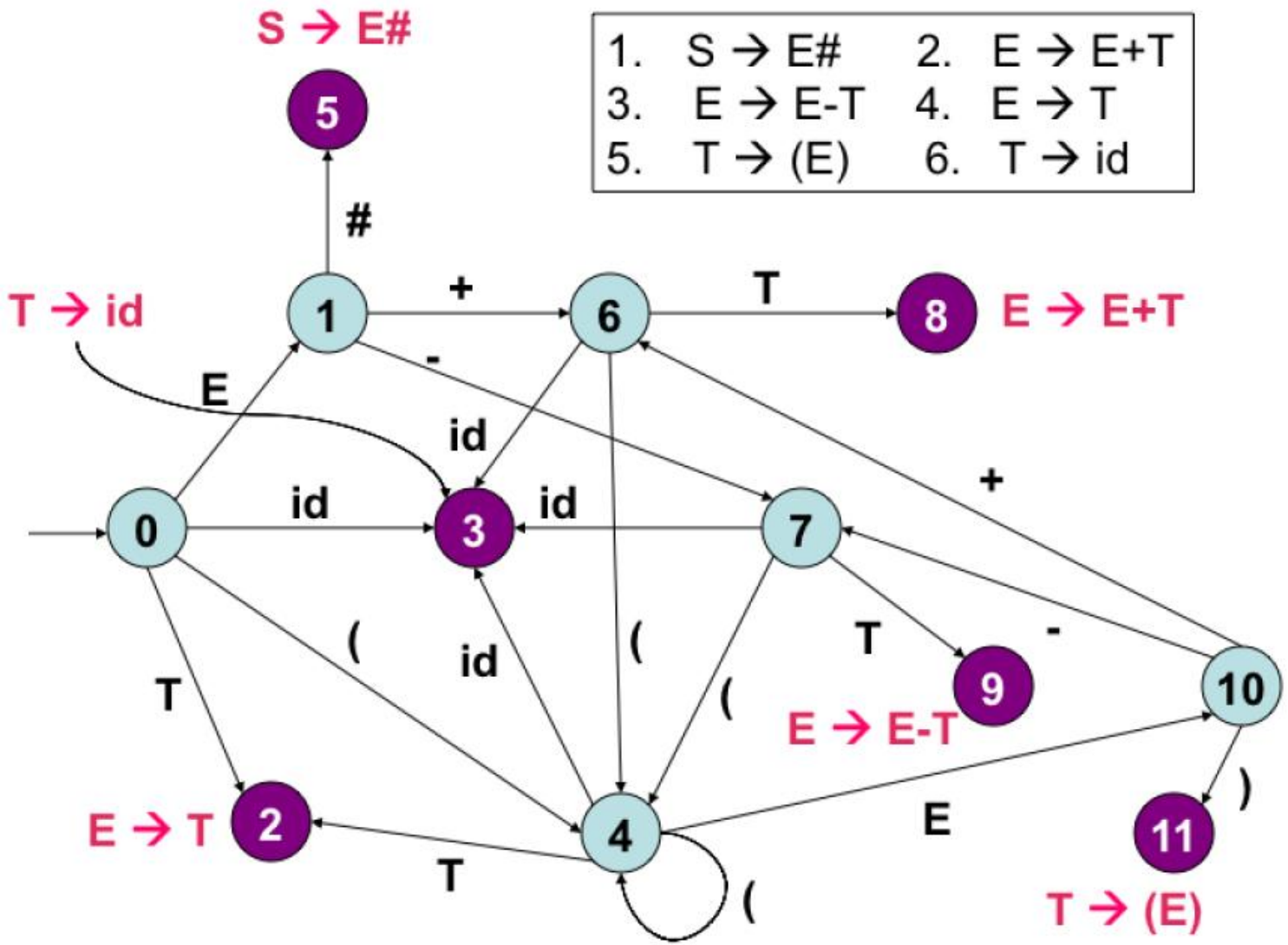
- The DFA of this regular language can **detect handles** during LR parsing

- When this DFA reaches a "reduction state", the corresponding viable prefix cannot grow further and thus **signals a reduction**

- This DFA can be constructed by the compiler using the grammar

- All **LR** parsers have such a DFA incorporated in them

Example: DFA for Viable Prefixes



LR Parsers: How do they work?

- **Key**: language of handles is regular
  - build a handle-recognising DFA
  - **Action** and **Goto** tables encode the DFA

- How do we generate the **Action** and **Goto** tables?
  - Use the grammar to build a model of the **DFA**
  - Use the model to build **Action** and **Goto** tables
  - If construction succeeds, the grammar is LR().

- Three commonly used algorithms to build tables:
  - LR(): full set of LR() grammars; large tables; slow, large construction.
  - SLR(): smallest class of grammars; smallest tables; simple, fast construction.
  - LALR(): intermediate sized set of grammars; smallest tables; very common.

Building Automaton from the Grammar

Items

- An (**LR(0)**) item of a grammar **G** is a production of **G** with a dot "**.**" at some position in the body

- An item indicates how much of a production we have seen at a given point in the parsing process

- **Example**: Consider production $A \rightarrow XYZ$
  - Four possible items ($A \rightarrow .XYZ$, $A \rightarrow X.YZ$, $A \rightarrow XY.Z$, $A \rightarrow XYZ.$)

- LR parser makes shift-reduce decisions by maintaining states (keep track of where we are in the parsing process)
- States represent **sets of "items"**

Constructing canonical LR(0) collection for a grammar

- We construct an augmented grammar for which we construct the **DFA**
  - If **S** is the start symbol of **G**, then **G′** contains all productions of **G** and also a new production **S′ → S**
  - New start production indicates when to stop parsing (***accept*** *if the parser is about to reduce* **S′ → S**)

- We define and use two functions **CLOSURE**, and **GOTO**

Closure of Item Sets

If ***I*** is a set of items for a grammar ***G***, then **CLOSURE(I)** is the set of items constructed from *I* by the two rules:

1. Initially, add every item in ***I*** to **CLOSURE(I)**.

2. If $A \rightarrow \alpha \cdot B\beta$ is in **CLOSURE(I)** and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot\gamma$ to **CLOSURE(I)**, if it is not already there. Apply this rule until no more new items can be added to **CLOSURE(I)**.

Example- Closure of Item Sets

- Consider the following grammar.

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

- Let **I** = {E’ → .E}. Compute CLOSURE(I).

Computation of CLOSURE

```
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for ( each item A → α·Bβ in J )
            for ( each production B → γ of G )
                if ( B → ·γ is not in J )
                    add B → ·γ to J;
    until no more items are added to J on one round;
    return J;
}
```

The GOTO Function

- Used to define the transitions in the LR() automaton.

- **GOTO(I,X)**
  - **I**: set of items
  - **X**: grammar symbol (terminal/ non-terminal)
  - **Closure** of set of all items [**A** → α**X.**β] such that [**A** → α**.X**β] is in **I**

- The states of the LR() automaton correspond to sets of items and **GOTO(I,X)** specifies the transition from the state **I** under input **X**

Example- GOTO

- Consider the following grammar.

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

- Let I = {[E’ → E.], [E → E.+T]}
- Compute **GOTO(I,+)**

Example- GOTO

- Consider the following grammar.

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

- Let I = {[E’ → E.], [E → E.+T]}
- Compute **GOTO(I,+)**
- Examine items with “+” immediately after the “.”
- **GOTO(I,+)** = CLOSURE ({[E → E+.T]})

Construction of Sets of Canonical LR(0) Items

```
void items(G') {
   C = CLOSURE({[S' → .S]});
   repeat
      for (each set of items I in C)
         for (each grammar symbol X)
            if (GOTO(I, X) is not empty and not in C)
               add GOTO(I, X) to C;
   until no new sets of items are added to C on a round;
}
```

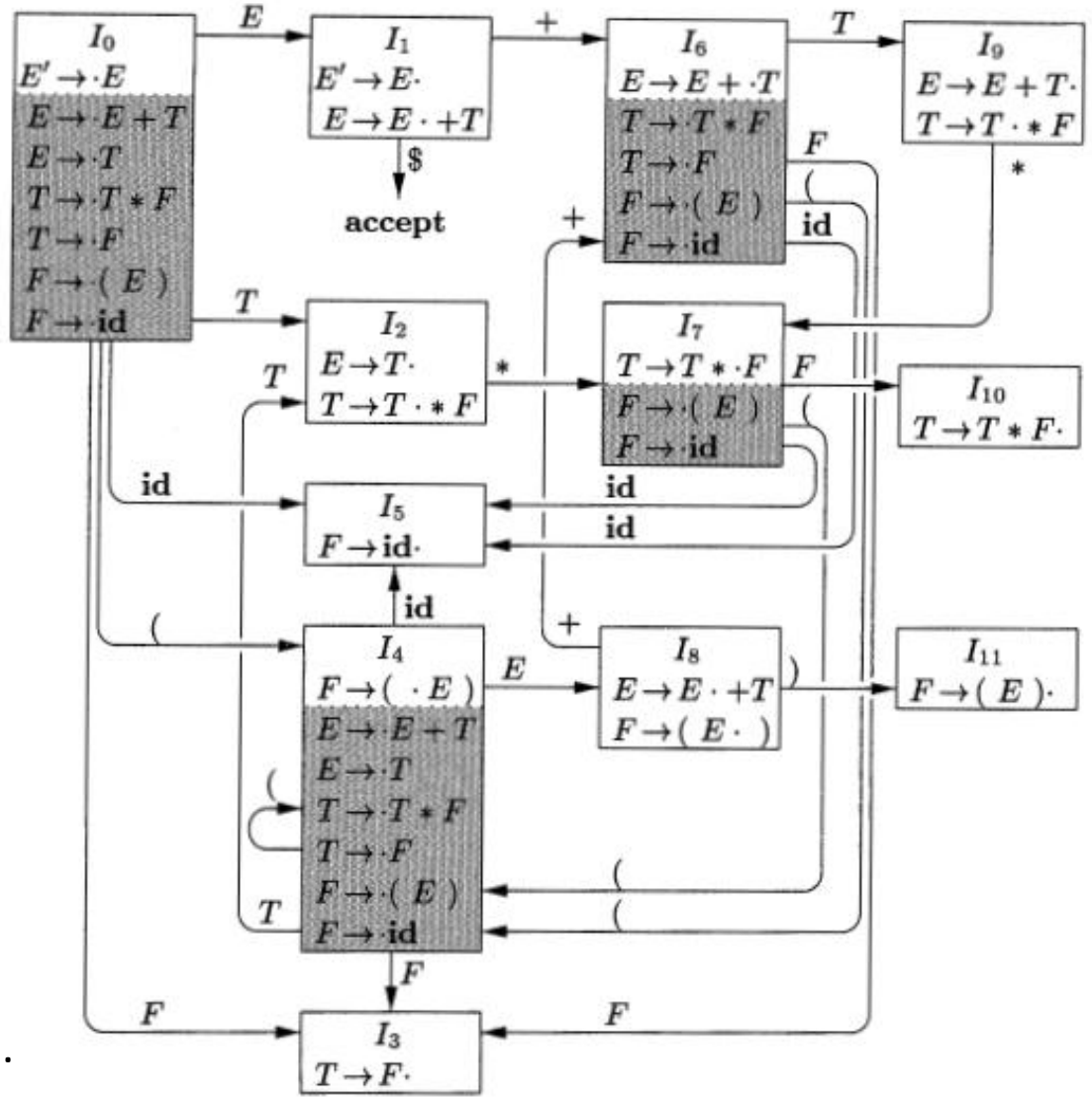
- Each set in *C* (above) corresponds to a state of a DFA (LR(0) DFA)
- This is the DFA that recognizes viable prefixes

Example- Canonical collection of sets of LR(0) Items

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

Example- Canonical collection of sets of LR(0) Items

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



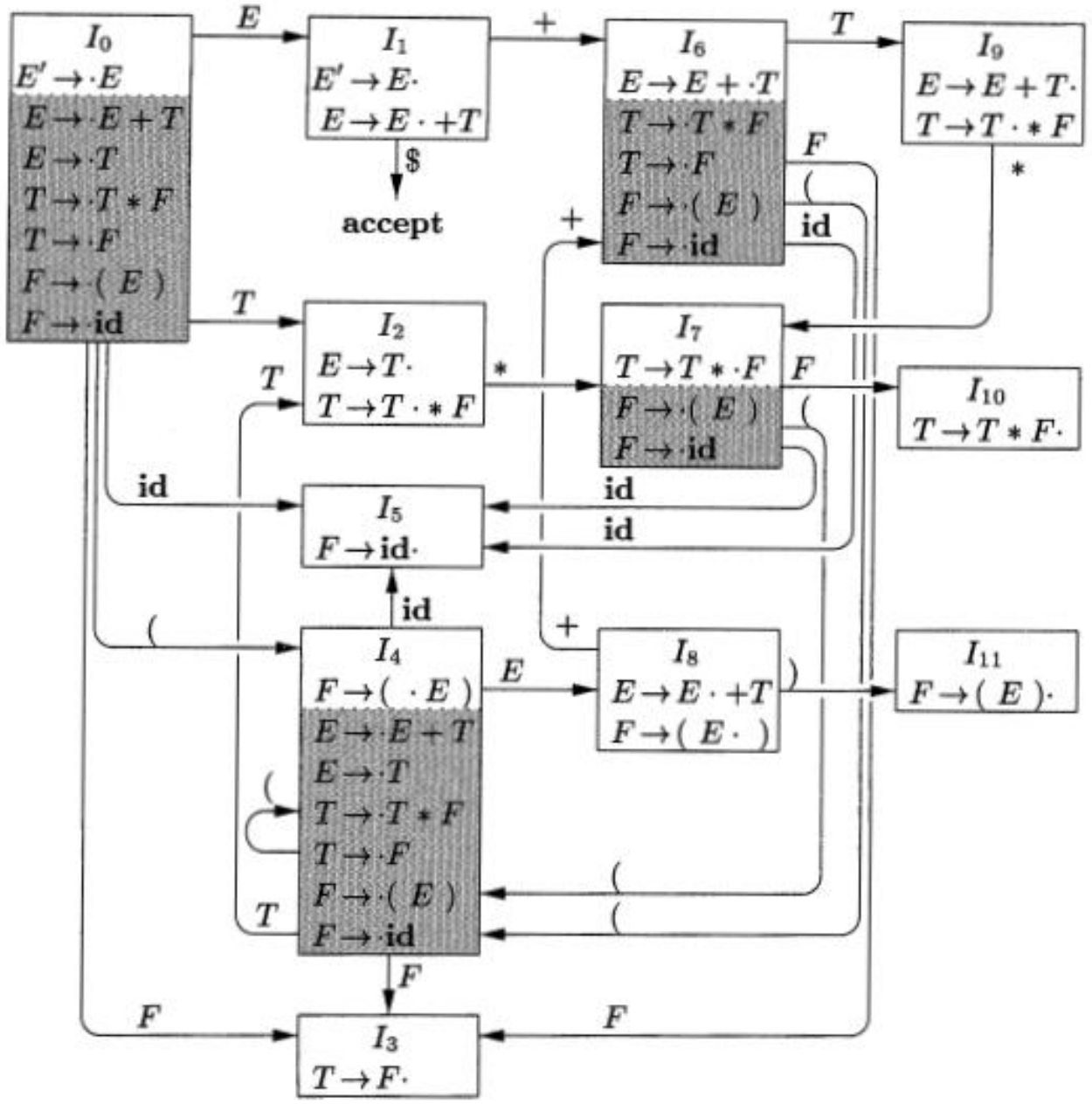
Sets of LR(0) items and GOTO (encoded by the transitions).

Shift Reduce Decisions using LR(0) Automaton

- Shift-reduce decisions using **LR(0)** automaton
  - Let string α take the automaton from state **0** (initial) to some state ***j***
  - Consider the next input symbol "a":
    - if state ***j*** has a transition on "**a**" (**SHIFT**)
    - otherwise (**REDUCE**): (items in state **j** tell us which production to use)

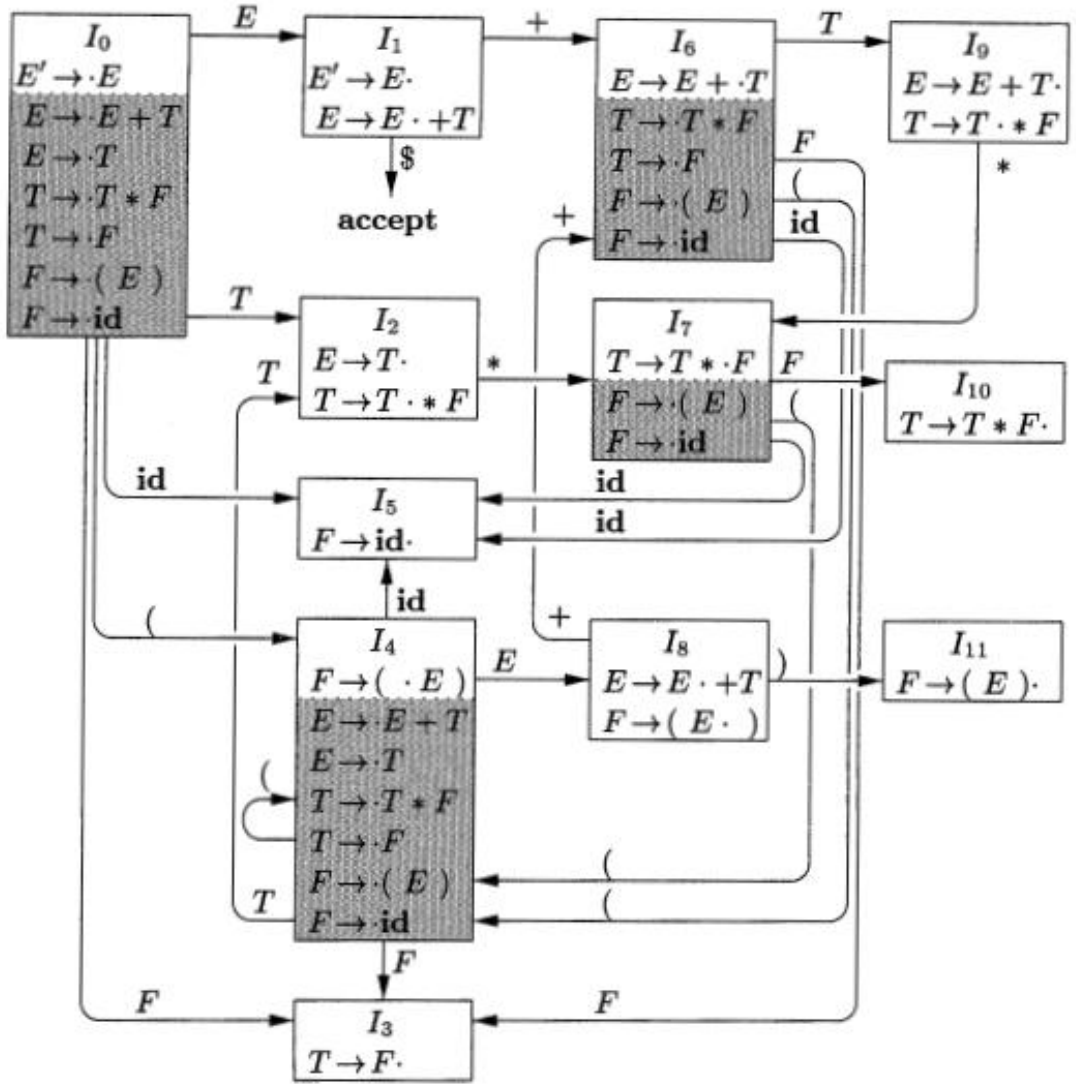
Example- Actions of a parser using LR(0) automaton for input "**id * id**"

Line	Stack	Symbols	Input	Action
1	0	$	id*id$	shift to 5



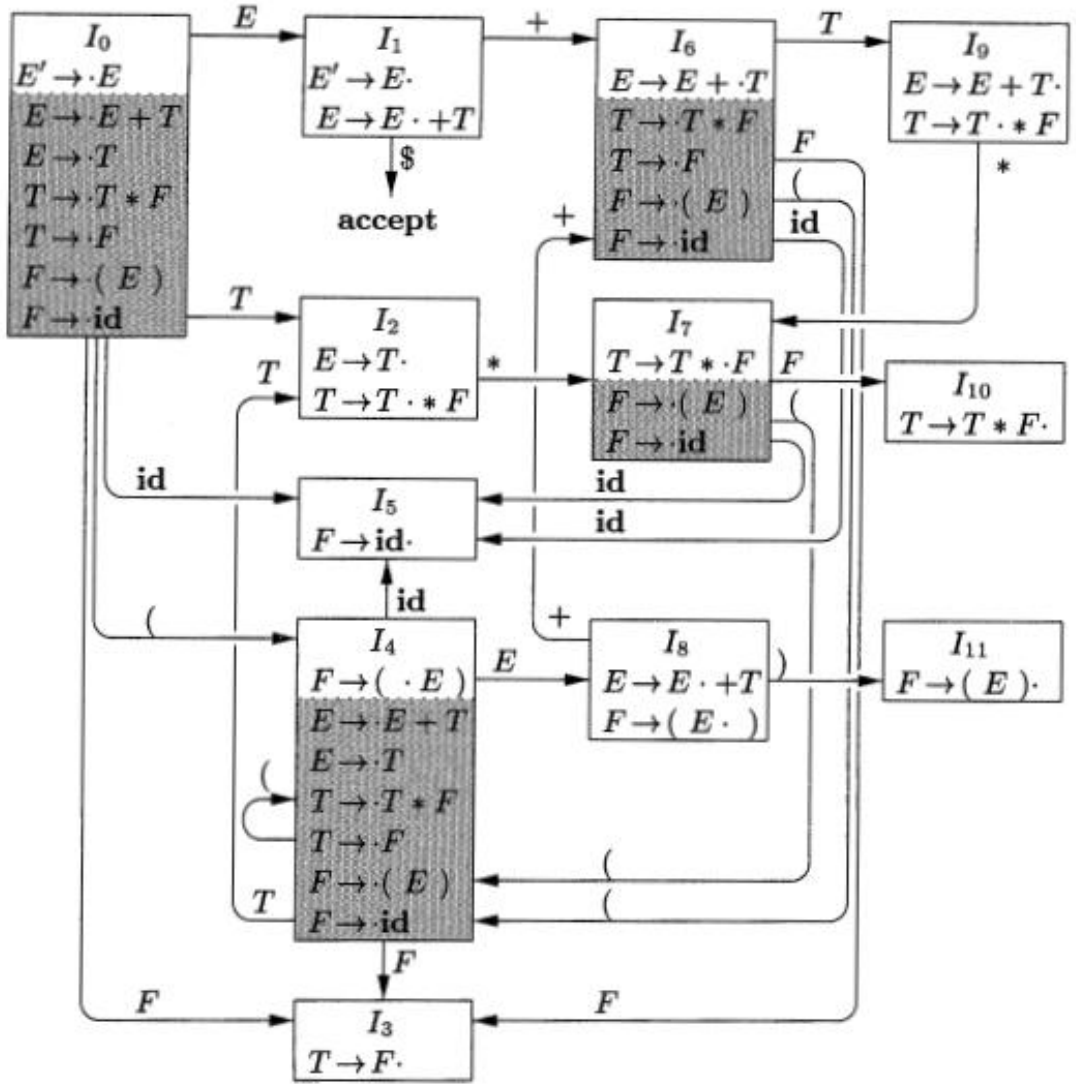
Example- Actions of a parser using LR(0) automaton for input "**id * id**"

Line	Stack	Symbols	Input	Action
1	0	$	id * id $	shift to 5
2	0 5	$ id	* id $	Reduce by F -> id



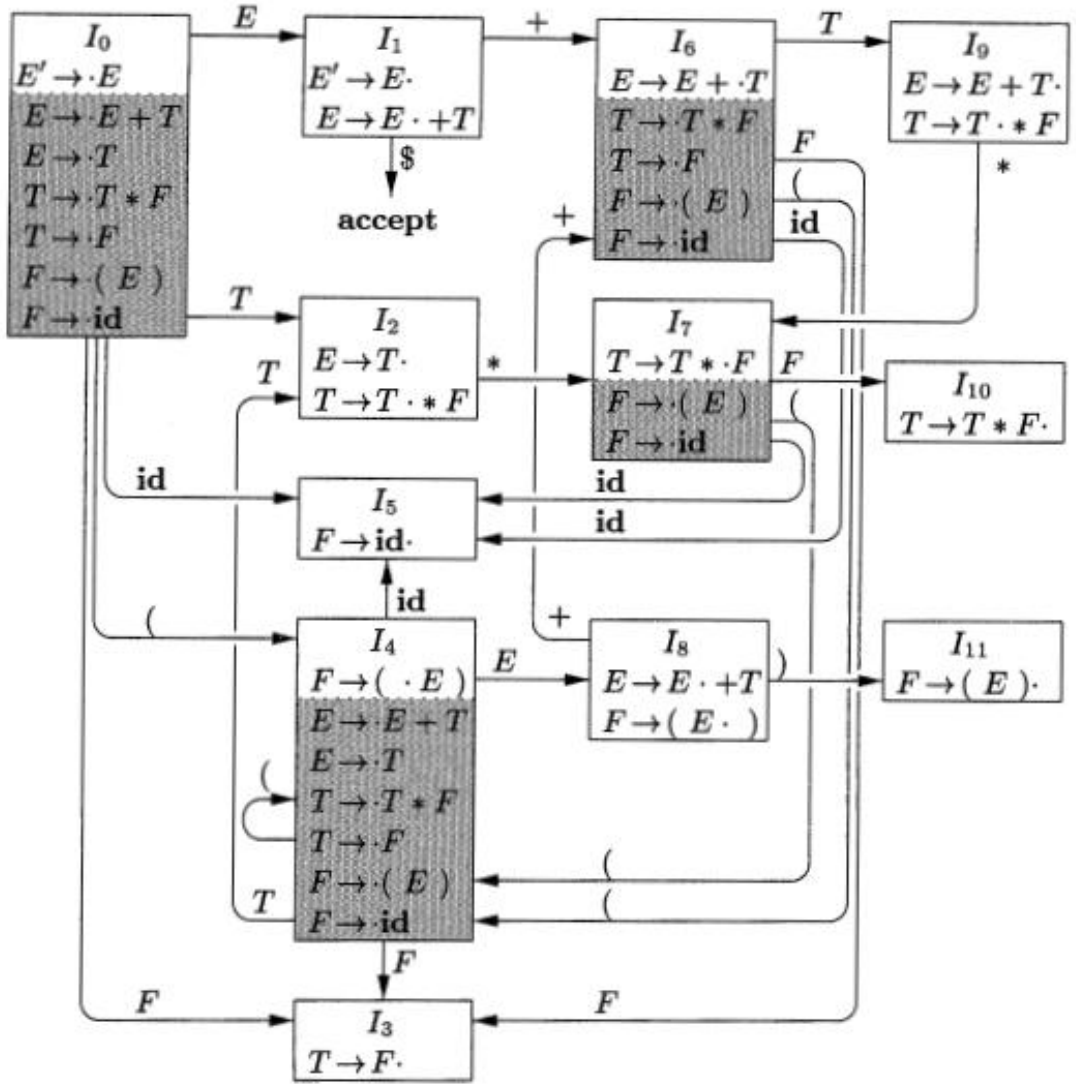
Example- Actions of a parser using LR(0) automaton for input "**id * id**"

Line	Stack	Symbols	Input	Action
1	0	$	id * id $	shift to 5
2	0 5	$ id	* id $	reduce by F -> id
3	0 3	$ F	* id $	reduce by T -> F

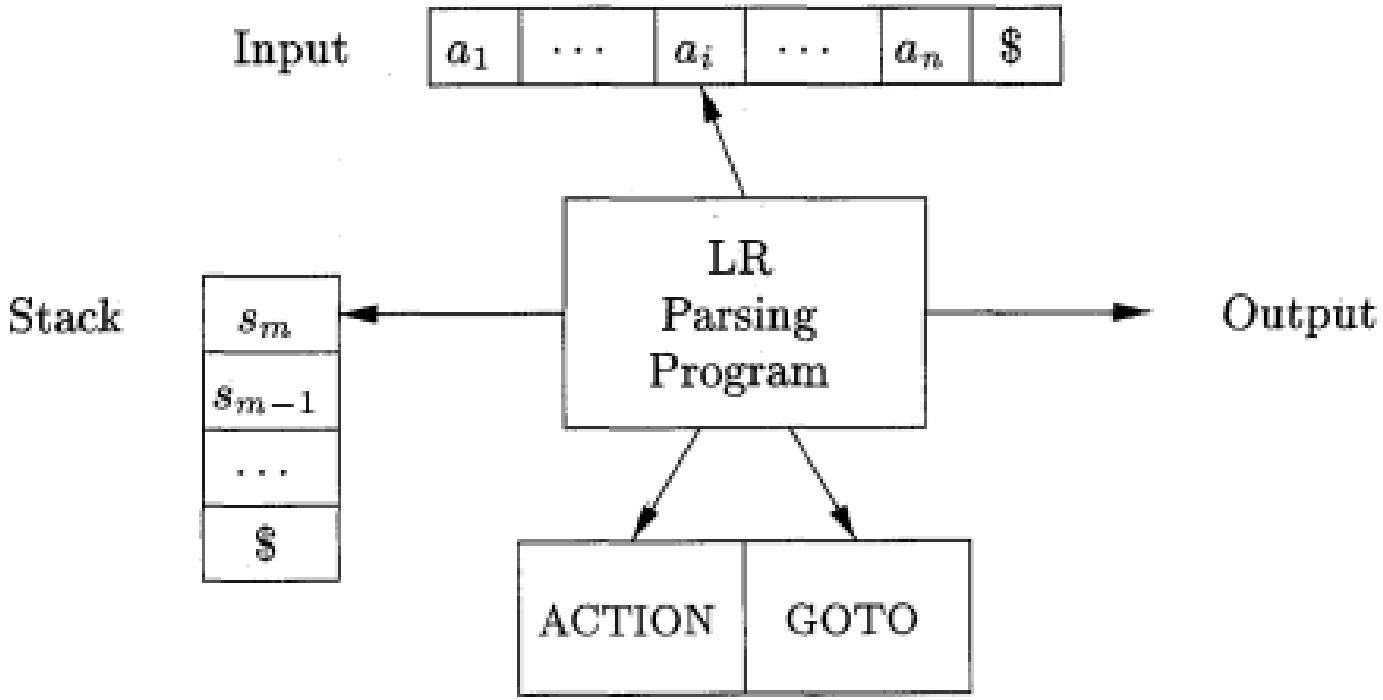


Example- Actions of a parser using LR(0) automaton for input "**id * id**"

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	**id** * **id** \$	shift to 5
(2)	0 5	\$ **id**	* **id** \$	reduce by $F \to$ **id**
(3)	0 3	\$ F	* **id** \$	reduce by $T \to F$
(4)	0 2	\$ T	* **id** \$	shift to 7
(5)	0 2 7	\$ T *	**id** \$	shift to 5
(6)	0 2 7 5	\$ T * **id**	\$	reduce by $F \to$ **id**
(7)	0 2 7 10	\$ T * F	\$	reduce by $T \to T * F$
(8)	0 2	\$ T	\$	reduce by $E \to T$
(9)	0 1	\$ E	\$	accept



The LR- Parsing Algorithm



The LR driver program is the same for all LR parsers
(only change in parsing table from one parser to another)

LR Parsing Algorithm

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
    let s be the state on top of the stack;
    if ( ACTION[s, a] = shift t ) {
        push t onto the stack;
        let a be the next input symbol;
    } else if ( ACTION[s, a] = reduce A → β ) {
        pop |β| symbols off the stack;
        let state t now be on top of the stack;
        push GOTO[t, A] onto the stack;
        output the production A → β;
    } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
    else call error-recovery routine;
}
```

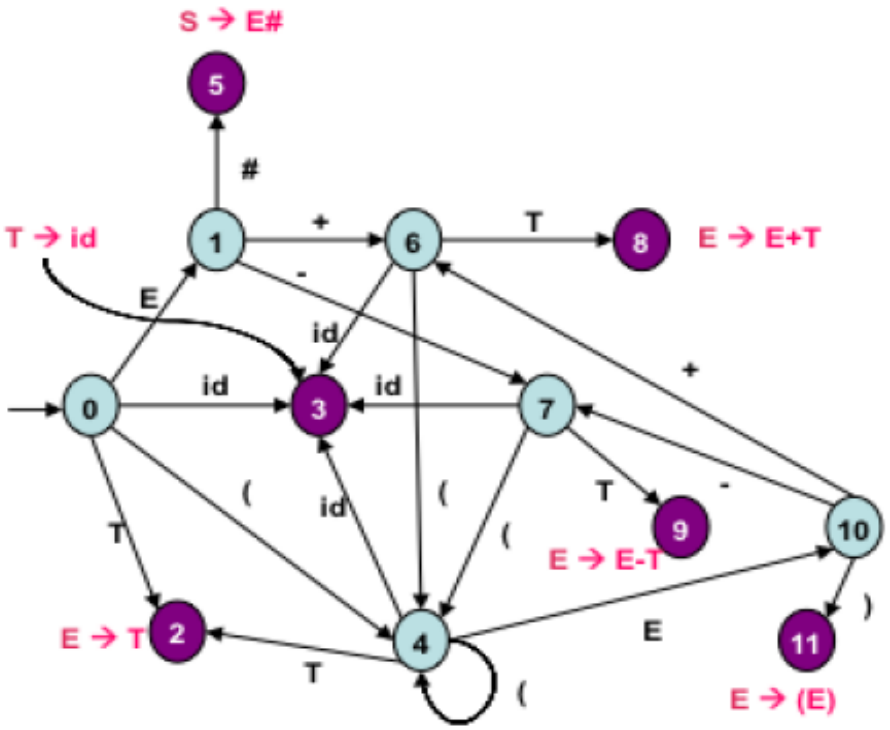
LR Parsing Table Structure

- Consists of two parts (parsing-action function ACTION, and function GOTO)

- ACTION[i, a]
  - **i**: State
  - **a**: terminal symbol (or $)
  - Possible actions
    - **Shift *j***: shift input a to the stack (use state **j** to represent **a**)
    - **Reduce A->** β: reduces β on top of the stack to **A**
    - Accept
    - Error
- GOTO[i, A]
  - **i**: State
  - **A**: Non-terminal
  - Maps state **i** and a non-terminal **A** to state **j**

Construction of LR(0) Parse Table

STATE	ACTION						GOTO		
	+	-	(	)	id	#	S	E	T
0									
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									



1. S → E#
2. E → E+T
3. E → E-T
4. E → T
5. T → (E)
6. T → id

State 0
S → .E#
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 1
S → E.#
E → E.+T
E → E.-T

State 2
E → T.

State 3
T → id.

State 4
T → (.E)
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 5
S → E#.

State 6
E → E+.T
T → .(E)
T → .id

State 7
E → E-.T
T → .(E)
T → .id

State 8
E → E+T.

State 9
E → E-T.

State 10
T → (E.)
E → E.+T
E → E.-T

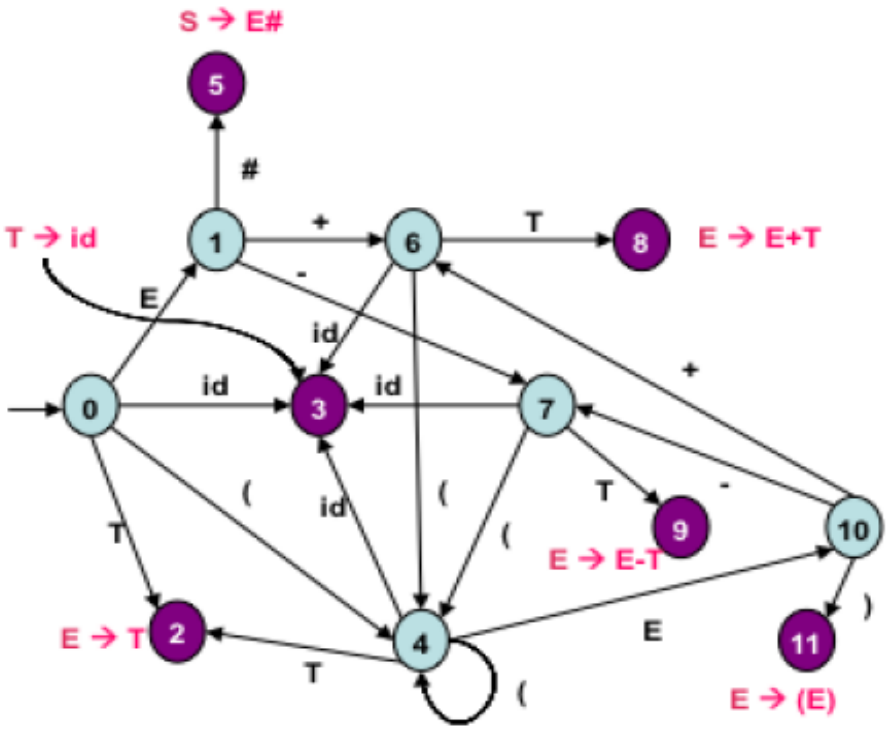
State 11
T → (E).

indicates closure items

indicates kernel items

Construction of LR(0) Parse Table

STATE	ACTION						GOTO		
	+	-	(	)	id	#	S	E	T
0			S4		S3			1	2
1	S6	S7				S5			
2	R4	R4	R4	R4	R4	R4			
3	R6	R6	R6	R6	R6	R6			
4			S4		S3			10	2
5	R1 acc	R1 acc	R1 acc	R1 acc	R1 acc	R1 acc			
6			S4		S3				8
7			S4		S3				9
8	R2	R2	R2	R2	R2	R2			
9	R3	R3	R3	R3	R3	R3			
10	S6	S7		S11					
11	R5	R5	R5	R5	R5	R5			



1. S → E#
2. E → E+T
3. E → E-T
4. E → T
5. T → (E)
6. T → id

State 0
S → .E#
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 1
S → E.#
E → E.+T
E → E.-T

State 2
E → T.

State 3
T → id.

State 4
T → (.E)
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 5
S → E#.

State 6
E → E+.T
T → .(E)
T → .id

State 7
E → E-.T
T → .(E)
T → .id

State 8
E → E+T.

State 9
E → E-T.

State 10
T → (E.)
E → E.+T
E → E.-T

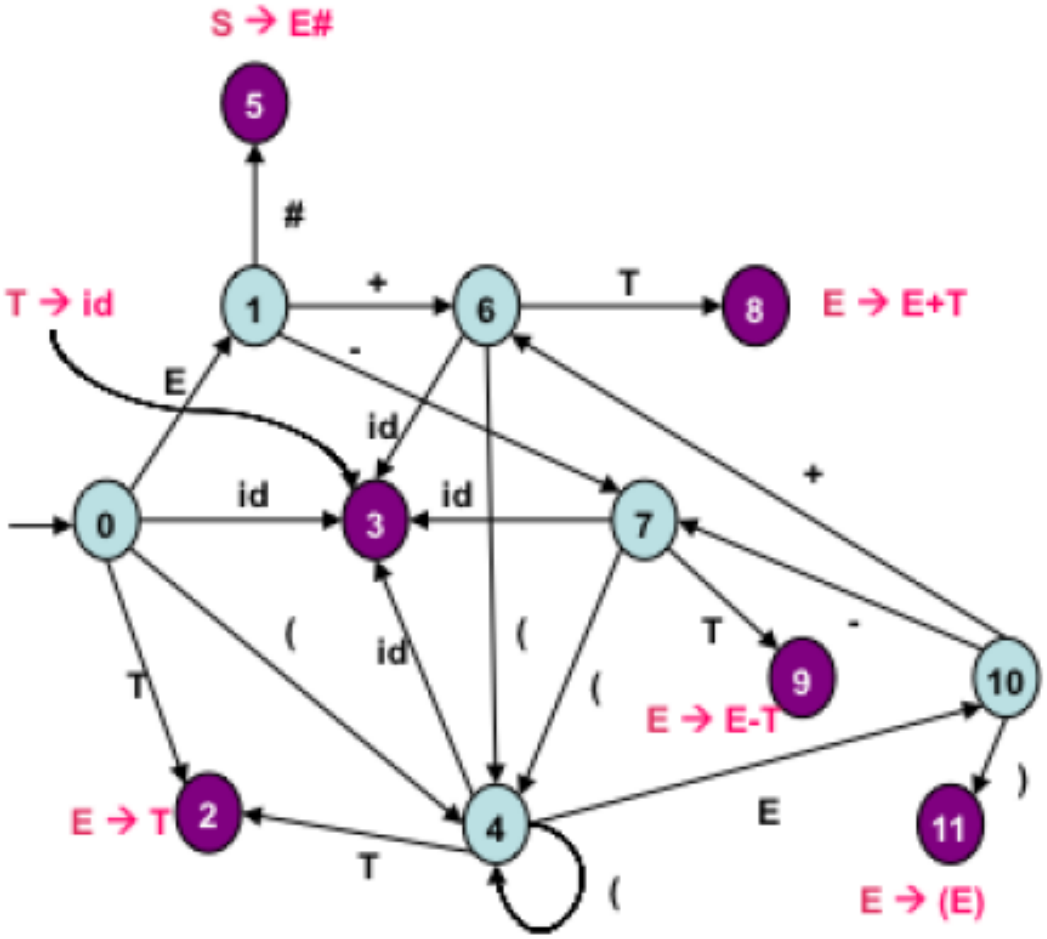
State 11
T → (E).

indicates closure items

indicates kernel items

Construction of LR(0) Parse Table

STATE	ACTION						GOTO		
	+	-	(	)	id	#	S	E	T
0			S4		S3			1	2
1	S6	S7				S5			
2	R4	R4	R4	R4	R4	R4			
3	R6	R6	R6	R6	R6	R6			
4			S4		S3			10	2
5	R1 acc	R1 acc	R1 acc	R1 acc	R1 acc	R1 acc			
6			S4		S3				8
7			S4		S3				9
8	R2	R2	R2	R2	R2	R2			
9	R3	R3	R3	R3	R3	R3			
10	S6	S7		S11					
11	R5	R5	R5	R5	R5	R5			



LR(0) Grammars

- There could be **shift-reduce** conflicts or **reduce-reduce** conflicts in a state
- Both shift and reduce items are present in the same state (S-R conflict), or
- More than one reduce item is present in a state (R-R conflict)

- If there are no S-R or R-R conflicts in any state of an LR(0) DFA, then the grammar is LR(0), otherwise, it is not LR(0)

Example- LR(0) Grammar

1. S → E#
2. E → E+T
3. E → E-T
4. E → T
5. T → (E)
6. T → id

State 0
S → .E#
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 1
S → E.#
E → E.+T
E → E.-T

State 2
E → T.

State 3
T → id.

State 4
T → (.E)
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 5
S → E#.

State 6
E → E+.T
T → .(E)
T → .id

State 7
E → E-.T
T → .(E)
T → .id

State 8
E → E+T.

State 9
E → E-T.

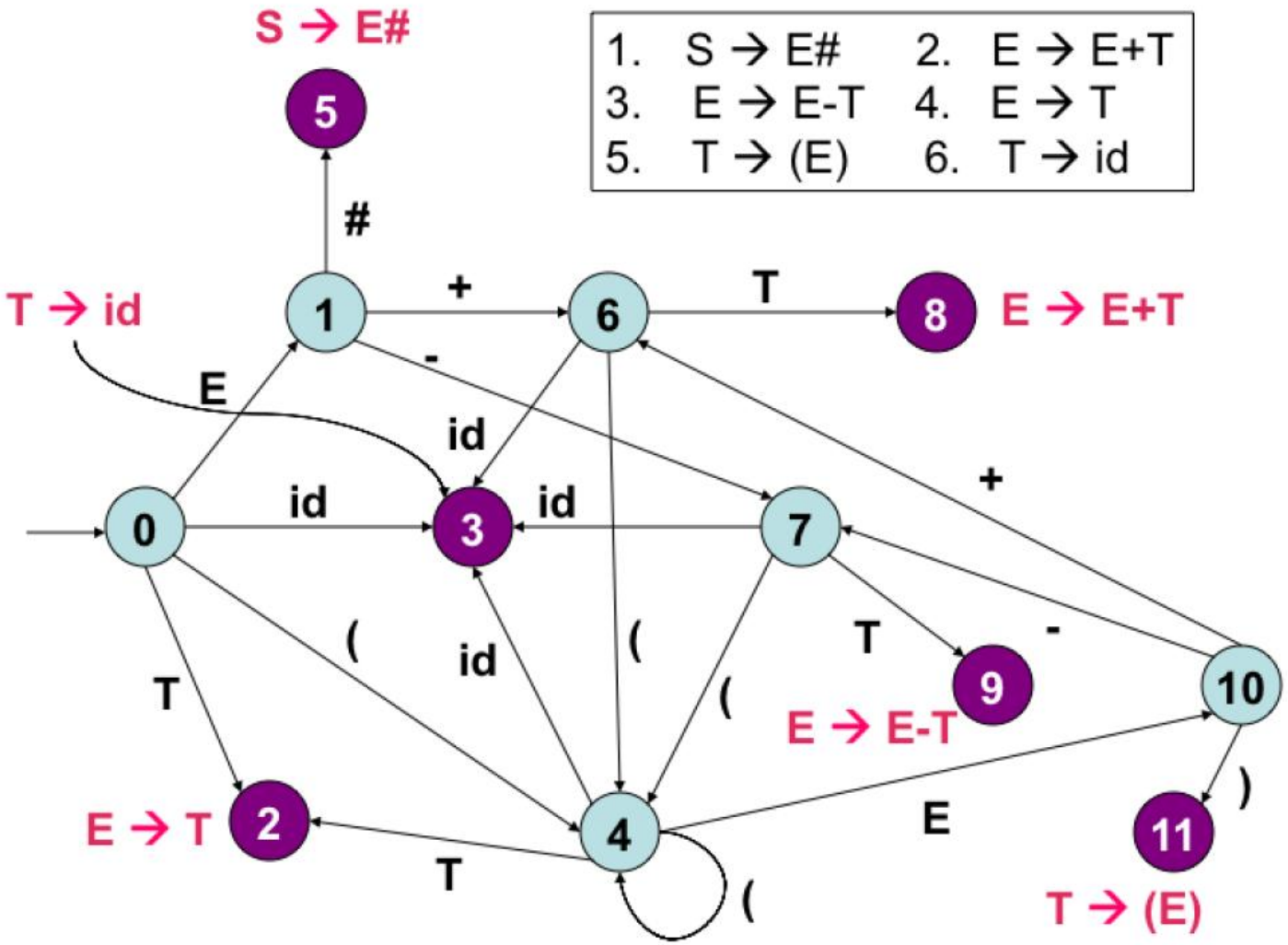
State 10
T → (E.)
E → E.+T
E → E.-T

State 11
T → (E).

indicates closure items

indicates kernel items

Example- LR(0) Grammar



Another Example- Compute LR(0) Automaton

S-> E

E-> E+T

E-> E-T

E-> T

T-> (E)

T-> id

Example- Grammar that is not LR(0)

State 0
S → .E
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 1
S → E.
E → E.+T
E → E.-T

shift-reduce conflicts in state 1

State 2
E → T.

State 3
T → id.

State 4
T → (.E)
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 5
E → E+.T
T → .(E)
T → .id

State 6
E → E-.T
T → .(E)
T → .id

State 7
E → E+T.

State 8
E → E-T.

State 9
T → (E.)
E → E.+T
E → E.-T

State 10
T → (E).

● indicates closure items

● indicates kernel items

follow(S) = {$}, where $ is EOF
Reduction on $, and shifts on + and - , will resolve the conflicts
This is similar to having an end marker such as #

Example- Grammar that is not LR(0)

State 0
S → .E
E → .E+T
E → .T
T → .F*T
T → .F
F → .(E)
F → .id

State 1
S → E.
E → E.+T
Shift-reduce conflict

State 2
E → T.

State 3
T → F.*T
T → F.
Shift-reduce conflict

State 4
F → (.E)
E → .E+T
E → .T
T → .F*T
T → .F
F → .(E)
F → .id

State 5
F → id.

State 6
E → E+.T
T → .F*T
T → .F
F → .(E)
F → .id

State 7
T → F*.T
T → .F*T
T → .F
F → .(E)
F → .id

State 8
F → (E.)
E → E.+T

State 9
E → E+T.

State 10
E → F*T.

State 11
F → (E).

SLR(1) Parsers

- If the grammar is not LR(0), we **try to resolve conflicts** in the states **using one look-ahead symbol**

- **Example**: The expression grammar that is not LR(0)

The state containing the items **[T → F .]** and **[T → F . * T]** has **S-R conflicts**

- Consider the reduce item **[T → F .]** and the symbols in FOLLOW(T)

- **FOLLOW(T) = {+,), $},** and **reduction by T → F** can be performed on seeing one of these symbols in the input (look-ahead), since **shift** requires seeing ***** in the input

- Recall from the definition of **FOLLOW(T)** that symbols in **FOLLOW(T)** are the only symbols that can legally follow **T** in any sentential form, and hence reduction by T → F when one of these symbols is seen, is correct

- If the **S-R conflicts can be resolved** using the **FOLLOW set**, the grammar is said to be **SLR(1)**

Constructing an SLR(1) Parsing table

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'.

2. State i is constructed from I_i. The parsing actions for state i are determined as follows:

   (a) If $[A \to \alpha \cdot a\beta]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift j." Here a must be a terminal.

   (b) If $[A \to \alpha \cdot]$ is in I_i, then set $\text{ACTION}[i, a]$ to "reduce $A \to \alpha$" for all a in $\text{FOLLOW}(A)$; here A may not be S'.

   (c) If $[S' \to S \cdot]$ is in I_i, then set $\text{ACTION}[i, \$]$ to "accept."

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.

4. All entries not defined by rules (2) and (3) are made "error."

5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S]$.

If any conflicting actions result in Step 2, then the grammar is not SLR(1). Cannot produce SLR parser in this case.

Example- grammar that is not LR(0). Is it SLR(1) ?

State 0
S → .E
E → .E+T
E → .T
T → .F*T
T → .F
F → .(E)
F → .id

State 1
S → E.
E → E.+T
Shift-reduce conflict

State 2
E → T.

State 3
T → F.*T
T → F.
Shift-reduce conflict

State 4
F → (.E)
E → .E+T
E → .T
T → .F*T
T → .F
F → .(E)
F → .id

State 5
F → id.

State 6
E → E+.T
T → .F*T
T → .F
F → .(E)
F → .id

State 7
T → F*.T
T → .F*T
T → .F
F → .(E)
F → .id

State 8
F → (E.)
E → E.+T

State 9
E → E+T.

State 10
E → F*T.

State 11
F → (E).

SLR(1) ??

Example- grammar that is not LR(0). Is it SLR(1) ?

State 0
S → .E
E → .E+T
E → .T
T → .F*T
T → .F
F → .(E)
F → .id

State 1
S → E.
E → E.+T
Shift-reduce conflict

State 2
E → T.

State 3
T → F.*T
T → F.
Shift-reduce conflict

State 4
F → (.E)
E → .E+T
E → .T
T → .F*T
T → .F
F → .(E)
F → .id

State 5
F → id.

State 6
E → E+.T
T → .F*T
T → .F
F → .(E)
F → .id

State 7
T → F*.T
T → .F*T
T → .F
F → .(E)
F → .id

State 8
F → (E.)
E → E.+T

State 9
E → E+T.

State 10
E → F*T.

State 11
F → (E).

SLR(1)

follow(S) = {$}, Reduction on $ and shift on +, eliminates conflicts
follow(T) = {$,), +}, where $ is EOF
Reduction on $,), and +, and shift on *, eliminates conflicts

Exercise: Construct SLR(1) parse table

1. S -> E
2. E -> E+ T
3. E -> T
4. T -> F*T
5. T-> F
6. F -> (E)
7. F-> id

Exercise- Is the following grammar LR(0)?

1. S’ -> S
2. S -> aSb
3. S -> ε

Exercise- Is the following grammar LR(0)?

1. S’ -> S
2. S -> aSb
3. S -> ε

State 0
S’ → .S
S → .aSb
S → .

State 1
S’ → S.

State 2
S → a.Sb
S → .aSb
S → .

State 3
S → aS.b

State 4
S → aSb.

Shift-reduce conflicts in states 0, 2

Grammar is not LR(0)

Exercise- Is the grammar SLR(1)?

1. S’ -> S
2. S -> aSb
3. S -> ε

State 0
S’ → .S
S → .aSb
S → .

State 1
S’ → S.

State 2
S → a.Sb
S → .aSb
S → .

State 3
S → aS.b

State 4
S → aSb.

Grammar is not LR(0), but is it SLR(1) ??

Exercise- Is the grammar SLR(1)?

1. S’ -> S
2. S -> aSb
3. S -> ε

State 0
S’ → .S
S → .aSb
S → .

State 1
S’ → S.

State 2
S → a.Sb
S → .aSb
S → .

State 3
S → aS.b

State 4
S → aSb.

follow(S) = {$, b}

	a	b	$	S
0	S2	reduce S → ε	reduce S → ε	1
1			accept	
2	S2	reduce S → ε	reduce S → ε	3
3		S4		
4		reduce S → aSb	reduce S → aSb	

Grammar is not LR(0), but is SLR(1)

Example- Is this grammar SLR(1) ?

GRAMMAR

1. S’ -> S
2. S -> aSb
3. S -> ab
4. S -> ε

Example- Is this grammar SLR(1) ?

GRAMMAR

1. S’ -> S
2. S -> aSb
3. S -> ab
4. S -> ε

State 0
S’ → .S
S → .aSb
S → .ab
S → .

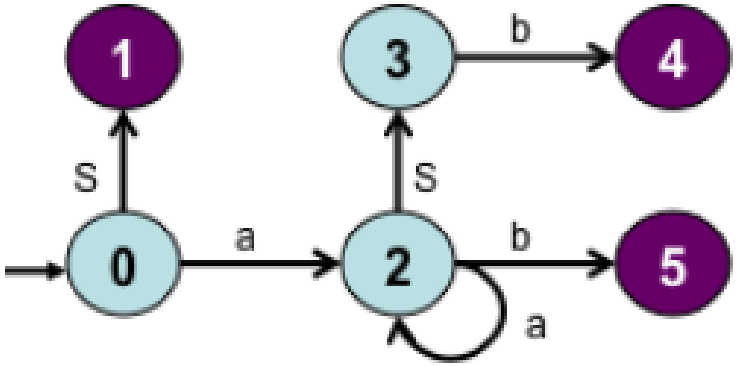
State 1
S’ → S.

State 2
S → a.Sb
S → a.b
S → .aSb
S → .ab
S → .

State 3
S → aS.b

State 4
S → aSb.

State 5
S → ab.



	a	b	$	S
0	S2	R: S → ε	R: S → ε	1
1			accept	
2	S2	S5, R: S → ε	R: S → ε	3
3		S4		
4		R: S → aSb	R: S → aSb	
5		R: S → ab	R: S → ab	

follow(S) = {$, b}
State 0: Reduction on *$* and *b*, by S → ε, and shift on *a* resolves conflicts
State 2: S-R conflict on *b* still remains

- Shift-reduce conflicts in states 0, 2
- Grammar is neither LR(0) nor SLR(1)

Example 2- Is this grammar SLR(1) ?

GRAMMAR

1. S’ -> S
2. S -> L = R
3. S -> R
4. L -> *R
5. L -> id
6. R -> L

Example 2- Is this grammar SLR(1) ?

GRAMMAR

1. S’ -> S
2. S -> L = R
3. S -> R
4. L -> *R
5. L -> id
6. R -> L

- **Follow(R) = {$,=}**
- S-R conflict cannot be resolved
- Grammar is not SLR(1)

State 0
S’ → .S
S → .L=R
S → .R
L → .*R
L → .id
R → .L

State 1
S’ → S.

State 3
S → R.

State 2
S → L .=R
R → L .
shift-reduce
conflict

State 4
L → *.R
R → .L
L → .*R
L → .id

State 5
L → id.

State 6
S → L=.R
R → .L
L → .*R
L → .id

State 7
L → *R.

State 8
R → L .

State 9
S → L=R.

LR(1) Parsers

LR(1) Parsers

- LR(1) items are of the form **[A → α.β, a]**, **a** being the "***lookahead***" symbol

- Lookahead symbols have no part to play in ***shift*** items, but in reduce items of the form [**A → α., a]**, reduction by **A → α** is valid only if the next input symbol is '**a**'

Constructing LR(1) sets of items

Closure of a set of LR(1) items

```
SetOfItems CLOSURE(I) {
        repeat
                for ( each item [A → α·Bβ, a] in I )
                        for ( each production B → γ in G' )
                                for ( each terminal b in FIRST(βa) )
                                        add [B → ·γ, b] to set I;
        until no more items are added to I;
        return I;
}
```

EXAMPLE

Grammar

S′ → S

S → aSb | ε

State 0

S′ → .S , $

S → .aSb , $

S → . , $

GOTO Set Computation

SetOfItems GOTO(I, X) {
    initialize J to be the empty set;
    for (each item $[A \to \alpha \cdot X \beta, a]$ in I)
        add item $[A \to \alpha X \cdot \beta, a]$ to set J;
    return CLOSURE(J);
}

EXAMPLE

Grammar
S' → S
S → aSb | ε

State 0
S' → .S , $
S → .aSb , $
S → . , $

State 1
S' → S. , $

State 2
S → a.Sb , $
S → .aSb , b
S → . , b

State 4
S → a.Sb , b
S → .aSb , b
S → . , b

GOTO(0,S) = 1; GOTO(0,a)=2; GOTO(2,a)=4

Construction of the sets of LR(1) items

```
void items(G') {
        initialize C to CLOSURE({[S' → ·S, $]});
        repeat
                for ( each set of items I in C )
                        for ( each grammar symbol X )
                                if ( GOTO(I, X) is not empty and not in C )
                                        add GOTO(I, X) to C;
        until no new sets of items are added to C;
}
```

LR(1) DFA - Each set in C (above) corresponds to a state of a DFA

This is the DFA that recognizes viable prefixes

Example- LR(1) DFA construction

GRAMMAR

1. S’ -> S
2. S -> CC
3. C -> cC | d

Example- LR(1) DFA construction

GRAMMAR

1. S’ -> S
2. S -> CC
3. C -> cC | d

I_0
S’ -> .S, $
S -> .CC, $
C -> .cC, c|d
C -> .d, c|d

I_1
S’ -> S., $

I_2
S -> C.C, $
C -> .cC, $
C -> .d, $

I_3
C -> c.C, c|d
C -> .cC, c|d
C -> .d, c|d

I_4
C -> d., c|d

I_5
S -> CC., $

I_6
C -> c.C, $
C -> .cC, $
C -> .d, $

I_7
C -> d., $

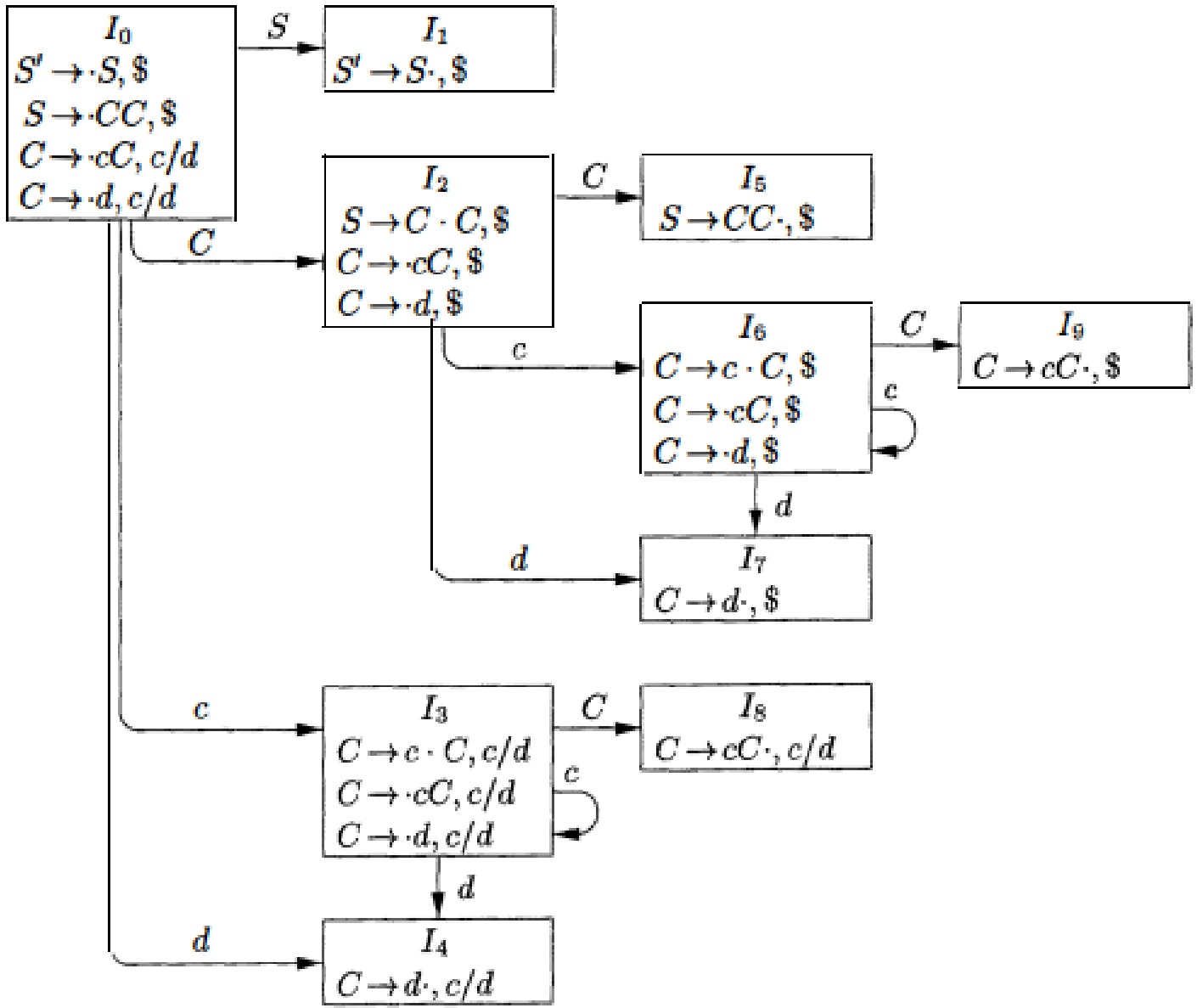
I_8
C -> cC., c|d

I_9
C -> cC., $

Example- LR(1) DFA construction

GRAMMAR

1. S’ -> S
2. S -> CC
3. C -> cC | d



Exercise- Construct LR(1) DFA for the following grammar

Grammar
S' → S, S → aSb, S → ε

Construction of an LR(1) parsing table

Let $C = \{I_0, I_1, ..., I_i, ..., I_n\}$ be the canonical LR(1) collection of items, with the corresponding states of the parser being 0, 1, ... , i, ... , n
Without loss of generality, let 0 be the initial state of the parser (containing the item $[S' \rightarrow .S,\ \$]$)

Parsing actions for state *i* are determined as follows

1. If $([A \rightarrow \alpha.a\beta,\ b] \in I_i)$ && $([A \rightarrow \alpha a.\beta,\ b] \in I_j)$
   set ACTION[i, a] = *shift j* /* *a* is a terminal symbol */
2. If $([A \rightarrow \alpha.,\ a] \in I_i)$
   set ACTION[i, a] = *reduce* $A \rightarrow \alpha$
3. If $([S' \rightarrow S.,\ \$] \in I_i)$ set ACTION[i, $] = *accept*

S-R or R-R conflicts in the table imply grammar is not LR(1)

GOTO

If $([A \rightarrow \alpha.A\beta,\ a] \in I_i)$ && $([A \rightarrow \alpha A.\beta,\ a] \in I_j)$
set GOTO[i, A] = *j* /* *A* is a nonterminal symbol */

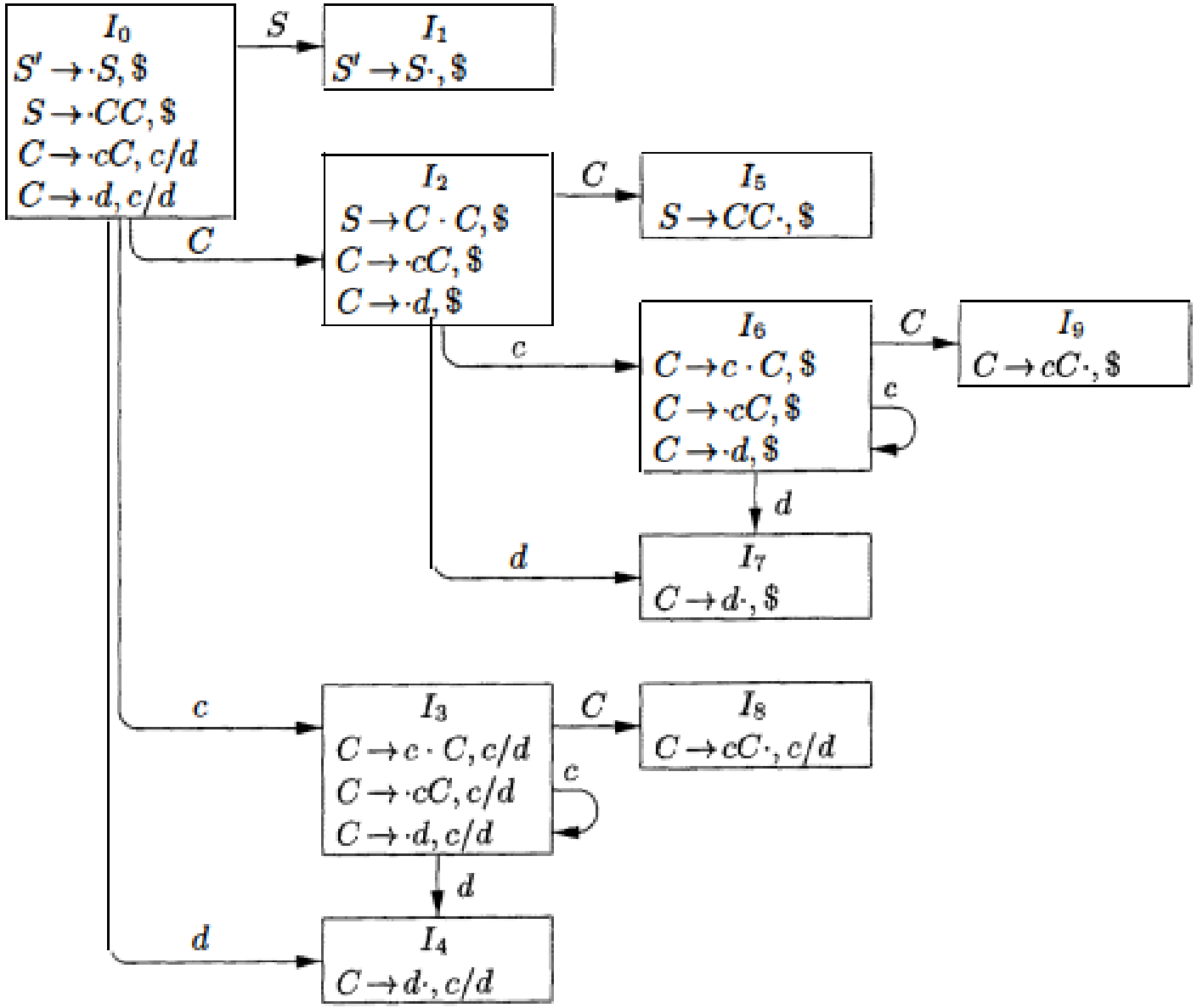
All other entries not defined by the rules above are made *error*

Example- LR(1) parse table

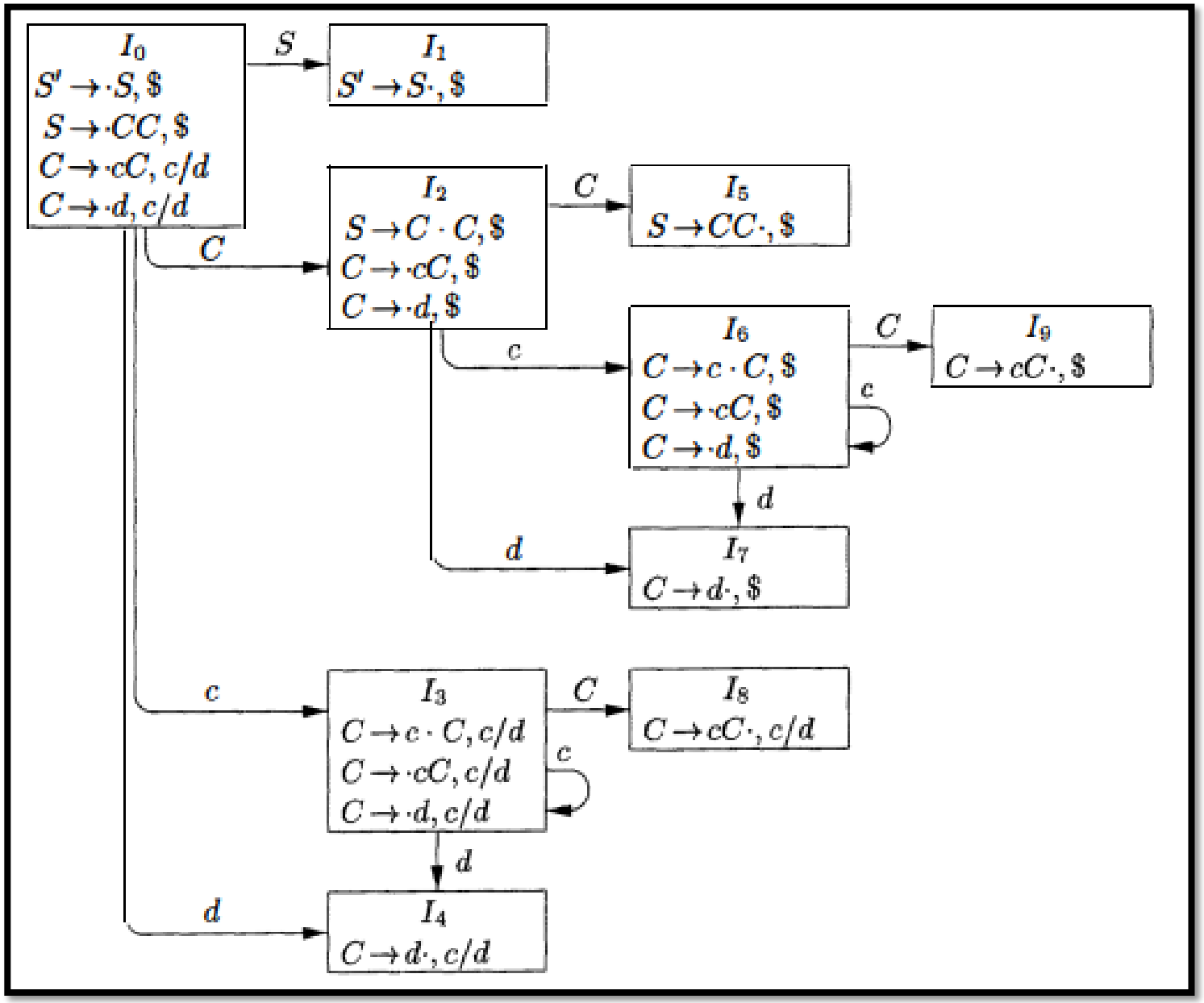
GRAMMAR

1. S’ -> S
2. S -> CC
3. C -> cC | d

STATE	ACTION			GOTO	
	c	d	$\$$	S	C



Example- LR(1) parse table



GRAMMAR

1. S' -> S
2. S -> CC
3. C -> cC | d

STATE	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Exercise

- Construct **LR(1)** automaton and **LR(1)** parse-table for the following grammar:

Grammar

S' → S

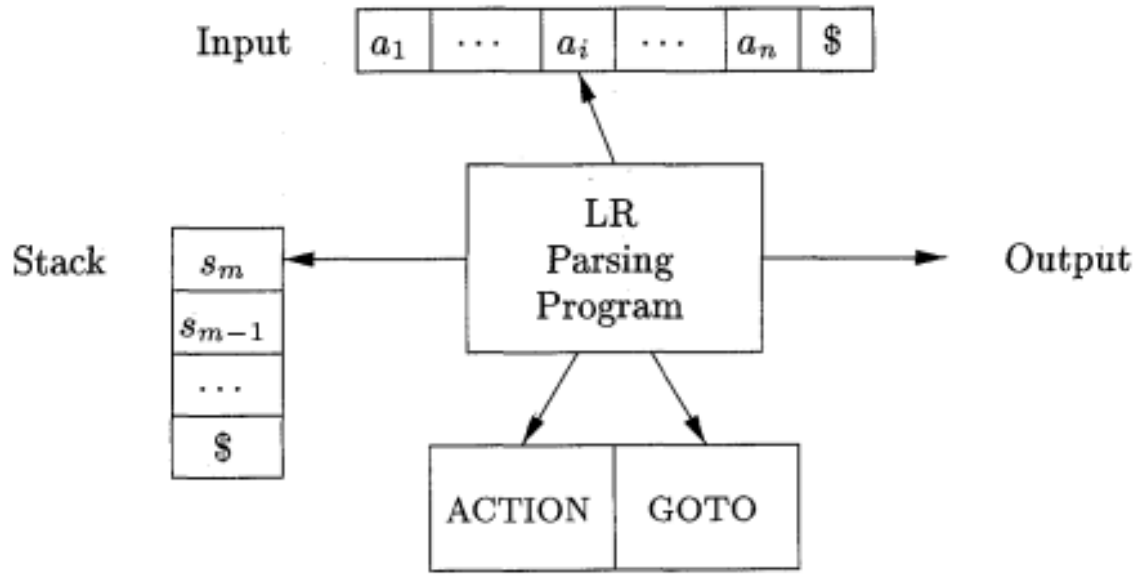
S → L=R | R

L → *R | id

R → L

- Is this grammar LR(1)?

RECAP- The LR- Parsing Algorithm



The LR driver program is the same for all LR parsers (**only change in parsing table from one parser to another**)

RECAP- LR Parsing Algorithm

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Example- Actions of a parser for input "**id * id**"

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to \mathbf{id}$

STATE	ACTION						GOTO		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
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			

Example- Actions of a parser for input "**id * id**"

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to \mathbf{id}$

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
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			

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	$	**id** * **id** $	shift to 5
(2)	0 5	$ **id**	* **id** $	reduce by $F \to \mathbf{id}$
(3)	0 3	$ F	* **id** $	reduce by $T \to F$
(4)	0 2	$ T	* **id** $	shift to 7
(5)	0 2 7	$ T *	**id** $	shift to 5
(6)	0 2 7 5	$ T * **id**	$	reduce by $F \to \mathbf{id}$
(7)	0 2 7 10	$ T * F	$	reduce by $T \to T * F$
(8)	0 2	$ T	$	reduce by $E \to T$
(9)	0 1	$ E	$	accept

Is the following grammar LR(1)?

Grammar
$S' \rightarrow S$
$S \rightarrow aSb$
$S \rightarrow ab$
$S \rightarrow \varepsilon$

A non-LR(1) grammar

Grammar
S' → S
S → aSb
S → ab
S → ε

	a	b	$	S
0	S2		R: S → ε	1
1			accept	
2	S5	S3, R: S → ε		4
3			R: S → ab	
4		S6		
5	S5	S9, R: S → ε		7
6			R: S → aSb	
7		S8		
8		R: S → aSb		
9		R: S → ab		

This grammar is neither SLR(1) nor LR(1), because it is ambiguous.

LALR(1) Parsers

LALR(1) Parsers

- **LR(1)** parsers have a ***large number of states***
  - For C, many thousand states
  - An **SLR(1)** parser (or **LR(0) DFA**) for **C** will have a few hundred states (with many conflicts)

- **LALR(1)** parsers have exactly the same number of states as **SLR(1)** parsers for the same grammar, and are derived from **LR(1)** parsers
  - **SLR(1)** parsers may have ***many conflicts***, but **LALR(1)** parsers may have ***very few conflicts***
  - If the **LR(1)** parser had no **S-R** conflicts, then the corresponding derived **LALR(1)** parser will also have none
  - However, this is not true regarding **R-R** conflicts

LALR(1) parsers are as compact as **SLR(1)** parsers and are ***almost*** as powerful as **LR(1)** parsers

Constructing LALR(1) Parsers

- The core part of **LR(1) items** (the part after leaving out the *lookahead* symbol) is the **same** for several **LR(1) states**

  (the ***loohahead*** symbols will be different)
  - Merge the states with the same core, along with the lookahead symbols, and rename them
- The **ACTION** and **GOTO** parts of the parser table will be modified
  - Merge the rows of the parser table corresponding to the merged states, replacing the old names of states by the corresponding new names for the merged states
  - **EXAMPLE**: If states **2** and **4** are merged into a new state **24**, and states **3** and **6** are merged into a new state **36**,

    all references to states **2, 4, 3, and 6** will be replaced by **24, 24, 36, and 36**, respectively
- LALR(1) parsers *may perform a few more reductions* (but not shifts) than an LR(1) parser before detecting an error

LALR(1) Parser Construction – Example

Grammar
S' → S, S → aSb, S → ε

State 0
S' → .S , $
S → .aSb , $
S → . , $

State 1
S' → S. , $

State 2
S → a.Sb , $
S → .aSb , b
S → . , b

State 3
S → aS.b , $

State 4
S → a.Sb , b
S → .aSb , b
S → . , b

State 5
S → aSb. , $

State 6
S → aS.b , b

State 7
S → aSb. , b

	a	b	$	S
0	S2		R: S → ε	1
1			accept	
2	S4	R: S → ε		3
3		S5		
4	S4	R: S → ε		6
5			R: S → aSb	
6		S7		
7		R: S → aSb		

LR(1) Parser Table

	a	b	$	S
0	S24		R: S → ε	1
1			accept	
24	S24	R: S → ε		36
36		S57		
57		R: S → aSb	R: S → aSb	

LALR(1) Parser Table

LALR(1) Parser Construction – Example

Grammar
S' → S, S → aSb, S → ε

State 0
S' → .S , $
S → .aSb , $
S → . , $

State 1
S' → S. , $

State 2
S → a.Sb , $
S → .aSb , b
S → . , b

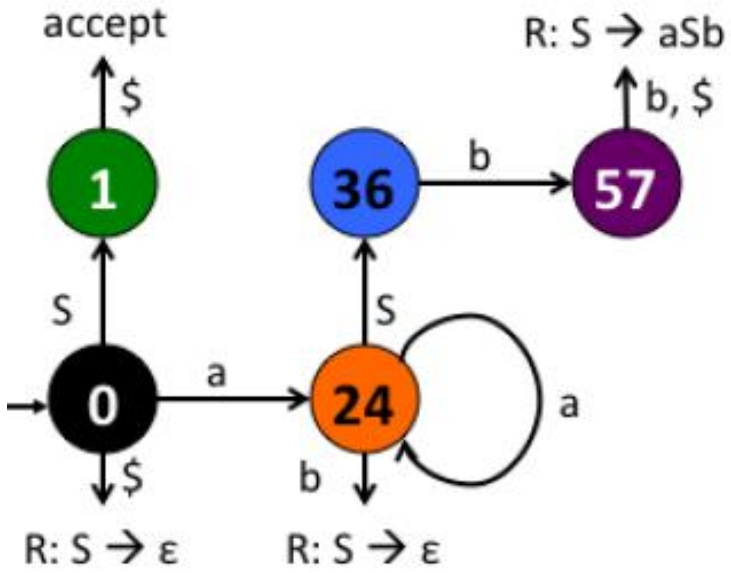
State 3
S → aS.b , $

State 4
S → a.Sb , b
S → .aSb , b
S → . , b

State 5
S → aSb. , $

State 6
S → aS.b , b

State 7
S → aSb. , b



	a	b	$	S
0	S24		R: S → ε	1
1			accept	
24	S24	R: S → ε		36
36		S57		
57		R: S → aSb	R: S → aSb	

LALR(1) Parser Table

Example- LALR(1) Parser Vs. LR(1) Parser: Actions for input “ab”

	a	b	$	S
0	S2		R: S → ε	1
1			accept	
2	S4	R: S → ε		3
3		S5		
4	S4	R: S → ε		6
5			R: S → aSb	
6		S7		
7		R: S → aSb		

	a	b	$	S
0	S24		R: S → ε	1
1			accept	
24	S24	R: S → ε		36
36		S57		
57		R: S → aSb	R: S → aSb	

Actions for input “ab”

Example- LALR(1) Parser Vs. LR(1) Parser: Actions for input “ab”

LR(1) Parser

0	ab$	shift
0 a 2	b$	S → ε
0 a 2 S 3	b$	shift
0 a 2 S 3 b 5	$	S → aSb
0 S 1	$	**accept**

LALR(1) Parser

0	ab$	shift
0 a 24	b$	S → ε
0 a 24 S 36	b$	shift
0 a 24 S 36 b 57	$	S → aSb
0 S 1	$	**accept**

Actions for input “ab”

Example- LALR(1) Parser Vs. LR(1) Parser: Error detection

Actions for input “aa”, and for input “aab”

0	aa$	shift
0 a 2	a$	shift
0 a 2 a 4	$	error

0	aa$	shift
0 a 24	a$	shift
0 a 24 a 24	$	error

0	aab$	shift
0 a 2	ab$	shift
0 a 2 a 4	b$	S → ε
0 a 2 a 4 S 6	b$	shift
0 a 2 a 4 S 6 b 7	$	error

0	aab$	shift
0 a 24	ab$	shift
0 a 24 a 24	b$	S → ε
0 a 24 a 24 S 36	b$	shift
0 a 24 a 24 S 36 b 57	$	S → aSb
0 a 24 S 36	$	error

Characteristics of LALR(1) parsers

- If an **LR(1)** parser has no **S-R** conflicts, then the corresponding derived **LALR(1)** parser will also have none

Merging of states with common core can never introduce a new S-R conflict, because shift depends only on core, not on lookahead

Characteristics of LALR(1) parsers (cont.)

- However, merger of states may introduce a **new R-R conflict** in the LALR(1) parser even though the original LR(1) parser had none
- Such grammars are rare in practice

Example:

Construct LR(1) automaton for the following grammar:

S' → S
S → aAd | bBd | aBe | bAe
A → c
B → c

Characteristics of LALR(1) parsers (cont.)

Construct LR(1) automaton for the following grammar:

$$S' \rightarrow S$$
$$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$$
$$A \rightarrow c$$
$$B \rightarrow c$$

Characteristics of LALR(1) parsers (cont.)

Construct LR(1) automaton for the following grammar:

$S' \rightarrow S\$,\ S \rightarrow aAd \mid bBd \mid aBe \mid bAe$
$A \rightarrow c,\ B \rightarrow c$

Characteristics of LALR(1) parsers (cont.)

- However, merger of states may introduce a **new R-R conflict** in the LALR(1) parser even though the original LR(1) parser had none
- Such grammars are rare in practice

Example:

Construct LR(1) automaton for the following grammar:

$$S' \rightarrow S\$,\ S \rightarrow aAd \mid bBd \mid aBe \mid bAe$$
$$A \rightarrow c,\ B \rightarrow c$$

- We will notice that two states contain the following items:

$$\{[A \rightarrow c.,\ d], [B \rightarrow c.,\ e]\} \text{ and}$$
$$\{[A \rightarrow c.,\ e], [B \rightarrow c.,\ d]\}$$

- The above two states will be merged to a single LALR(1) state which has a reduce-reduce conflict

$$\{[A \rightarrow c.,\ d/e], [B \rightarrow c.,\ d/e]\}$$

Error Recovery in LR Parsers