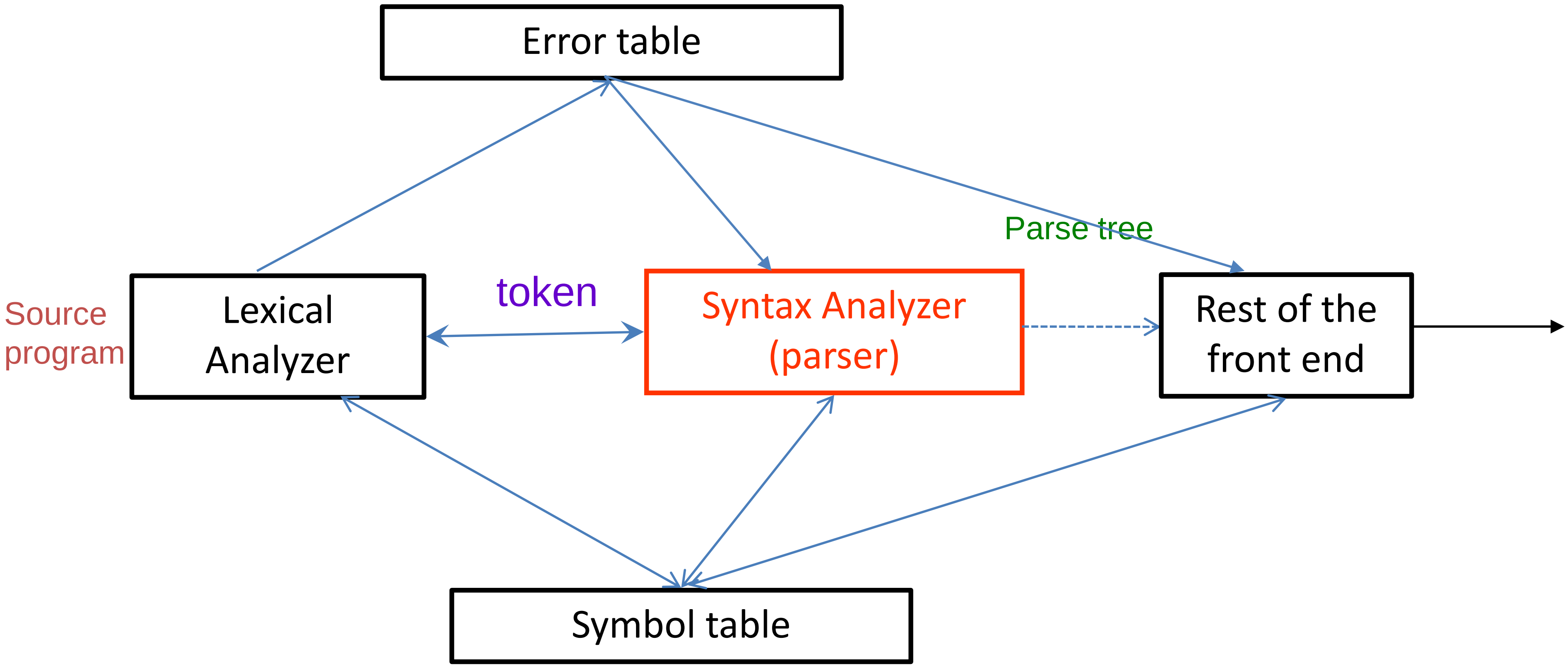
CSE 4801

Syntax Analysis

(Parsing)

Role of the Syntax Analyzer



Syntax Error Handling

Syntactic Errors: invalid order of tokens.

Example:

- Arithmetic expression with unbalanced parenthesis
- Two consecutive operands or operators
- A statement without end marker (; for TC), etc.

Goals of syntax error handler

- Report the presence of syntactic errors clearly and accurately
- Recover from each error quickly (optional)
- Should not slow down the entire compilation

Syntax Error Recovery

Recovery Strategies

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction

Formal Definition of a Context-Free Grammar

TMC @ CSE.IUT

A context-free grammar (grammar for short) consists of a set of terminals, a set of non-terminals, a start symbol, and a set of productions.

- **Terminals** are the basic symbols from which strings are formed.
- **Nonterminals** are syntactic variables that denote sets of strings.
- In a grammar, one nonterminal is distinguished as the **start symbol**, and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.
- The **productions** of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings.

Productions

1. E → E + E
2. E → E * E
3. E → -E
4. E → (E)
5. E → **id**
6. E → **num**

- Terminals: +, *, -, (,), **id**, **num**
- Non-terminals: E
- Start Symbol: E
- Productions: 1..6

A production can be treated as a rewriting rule in which the nonterminal on the left can be replaced by the string on the right side of the production or vice-versa.

This replacement process is known as derivation.

Notational conventions

(context-free grammars)

1. **Terminals**

- early lower case letters: *a*,*b*,*c*,..
- Operator symbols: +,-,*….
- Punctuation symbols: comma, parenthesis,….
- Digits: 0,1,2,….,9
- Boldface strings: **id, if**

Notational conventions

(context-free grammars)

2. **Non-terminals**
   - early upper case letters: *A,B,C*,..
   - The letter *S,* usually the start symbol.
   - Lower case italic names such as *expr* or *stmt*

3. **Upper case letters, late in the alphabet, such as X, Y, Z, represent a grammar symbol, either *terminal* or *non-terminal*.**

4. **Lower case letters, late in the alphabet, such as x, y, z, represent string of terminals.**

Notational conventions

(context-free grammars)

5. **Lower case Greek letters, α, β, γ, represent strings of grammar symbols.**

6. **If $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$,…. $A \rightarrow \alpha_k$ are all productions with A on the left side,**

   then we may write $A \rightarrow \alpha_1 | \alpha_2 | \ldots\ldots | \alpha_k$

7. **Unless stated otherwise, the left side of the first production is the start symbol.**

Derivation and Parse Tree

For input **id+id*id**

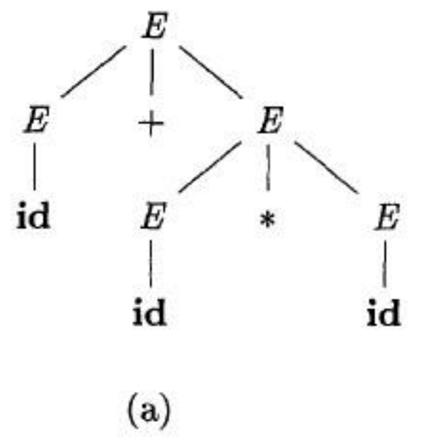
Grammar

1. E → E + E
2. E → E * E
3. E → -E
4. E → (E)
5. E → **id**
6. E → **num**

Derivations

E → E + E
→ **id** + E
→ **id** + E * E
→ **id** + **id** * E
→ **id** + **id** * **id**

Parse Tree



Parse tree is a graphical representation for a set of derivation. A parse tree for input id+id*id is presented above.

Grammar Ambiguity

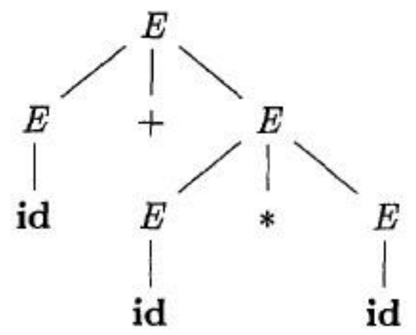
Grammar

1. E → E + E
2. E → E * E
3. E → -E
4. E → (E)
5. E → **id**
6. E → **num**

Derivations

E → E + E
→ **id** + E
→ **id** + E * E
→ **id** + **id** * E
→ **id** + **id** * **id**

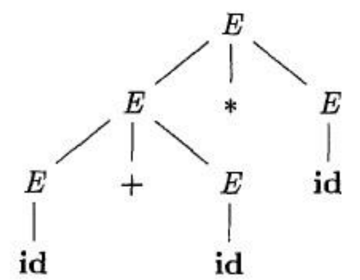
Parse Tree



(a)

Alternate Parse Tree

E → E * E
→ E + E * E
→ **id** + E * E
→ **id** + **id** * E
→ **id** + **id** * **id**



(b)

Input: id+id*id

Ambiguity in branching statement

Stmt -> if expr then stmt

| if expr then stmt else stmt

| other

If E1 then if E2 then S1 else S2

Stmt -> if expr then **stmt** (by production 1)
-> if expr then *if expr then stmt else stmt* (by production 2)

Stmt -> if expr then **stmt** else stmt (by production 2)
-> if expr then *if expr then stmt* else stmt (by production 1)

Eliminating Ambiguity

TMC @ CSE.IUT

stmt -> matched_stmt

| unmatched_stmt

matched_stmt -> if expr then matched_stmt else matched_stmt

| other

unmatched_stmt -> if expr then stmt

| if expr then matched_stmt else unmatched_stmt

Types of Parsing

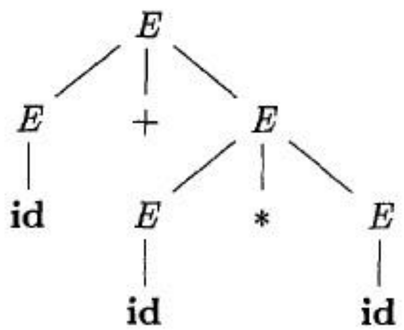
- **Top-Down Parsing (LL)**
  - **Recursive Descent Parsing**
  - **Nonrecursive Predictive Parsing**
- **Bottom-Up Parsing (LR)**
  - **Simple LR (SLR)**
  - **Canonical LR (CLR)**
  - **Look Ahead LR (LALR)**

LL: Scan the input from Left to Right, Use Left most derivation

LR: Scan the input from Left to Right, Use Right most derivation in Reverse order

Types of Parsing

Top-Down Parsing
(LL Parser)



(a)

Bottom-Up Parsing
(LR Parser)

LL Parser

LL

Scan or read source/input text from *Left to Right*

Use *Left most derivation*

Grammar

1. E → E + E
2. E → E * E
3. E → -E
4. E → (E)
5. E → **id**
6. E → **num**

Input: id+id*id

Derivations

E → E + E
→ **id** + E
→ **id** + E * E
→ **id** + **id** * E
→ **id** + **id** * **id**

LR Parser

TMC @ CSE.IUT

L R

Scan source/input text from *Left to Right*

Use *Right most derivation in reverse order*

Grammar

1. E → E + E
2. E → E * E
3. E → -E
4. E → (E)
5. E → **id**
6. E → **num**

Input: id+id*id

Derivations

E → E + E
→ **id** + E
→ **id** + E * E
→ **id** + **id** * E
→ **id** + **id** * **id**

Sentential Form

- A **sentential form** is any string derivable from the start symbol. Note that this includes the forms with non-terminals at intermediate steps as well.

- A **right-sentential** form is a sentential form that occurs in a step of rightmost derivation (RMD).

- A **left-sentential** form is a sentential form that occurs in a step of leftmost derivation (RMD).

- A **sentence** is a sentential form consisting only of terminals.

Top-Down Parsing (LL)

Examples of Top-Down Parsing (LL)

- **Recursive Descent Parsing**
- **Nonrecursive Predictive Parsing**

LL Parser: Recursive Descent Parsing

Select a production to derive a non-terminal randomly during parsing. If any dead end is there before fully parsing the input do backtrack.

Grammar

1. E → E + E
2. E → E * E
3. E → -E
4. E → (E)
5. E → **id**
6. E → **num**

Input: id+id*id

E → E * E (using production no. 2; taken randomly)

.

.

.

21

LL Parser: Recursive Descent Parsing

Select a production to derive a non-terminal randomly during parsing. If any dead end is there before fully parsing the input do backtrack.

Grammar

1. E → E + E
2. E → E * E
3. E → -E
4. E → (E)
5. E → **id**
6. E → **num**

Input: id+id*id

Mismatch

E → E * E (using production no. 2)
E → id * E (using production no. 5; backtrack needed)
E → E * E (undo derivation by production no. 5)
E → E + E * E (using production no. 1)
E → id + E * E (using production no. 5)
E → id + id * E (using production no. 5)
E → id + id * id (using production no. 5)

Lots of backtracking may be needed in between these steps.

22

LL Parser: Predictive Parsing

TMC @ CSE.IUT

During parsing select a unique production for derivation based on present input token and present symbol to derive.

Input: id+id*id

Grammar

1. E → E + E
2. E → E * E
3. E → -E
4. E → (E)
5. E → **id**
6. E → **num**

E → ??

Modify a grammar for predictive parsing

- **Eliminate Left Recursion**
- **Do Left Factoring**

Left Recursion

$A \rightarrow A +$

Input: id+id*id

Grammar

1. $E \rightarrow E + E$
2. $E \rightarrow E * E$
3. $E \rightarrow -E$
4. $E \rightarrow (E)$
5. $E \rightarrow$ **id**
6. $E \rightarrow$ **num**

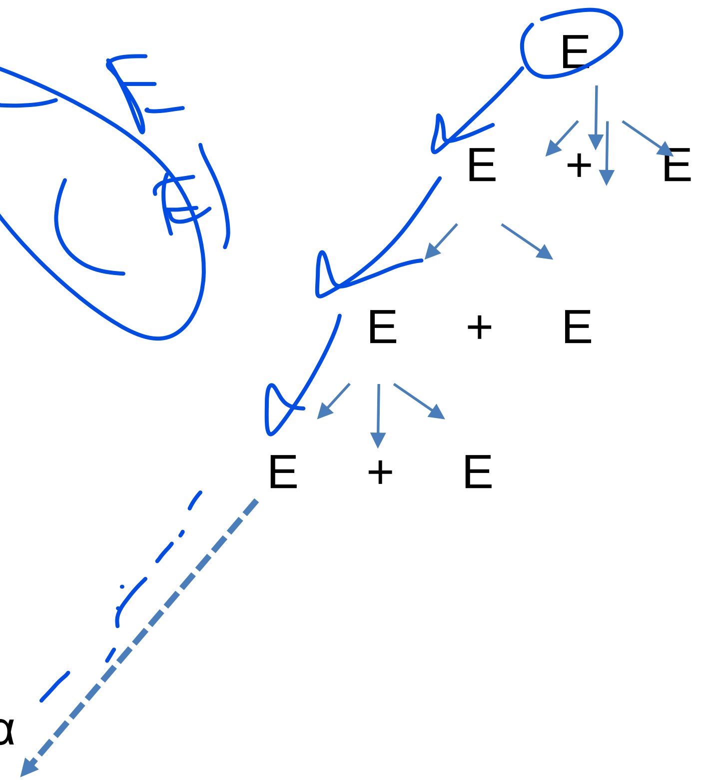
$E \rightarrow E+E$

$E \rightarrow E+E+E$

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

E

(E)



To implement top-down parsing left recursion need to be eliminated without affecting the language represented by the grammar.

Elimination of Left Recursion

TMC @ CSE.IUT

A pair of "A productions" with left recursion:

$A \rightarrow A\alpha$
$A \rightarrow \beta$

Modified productions without left recursions keeping the recognized language intact:

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A'$
$A' \rightarrow \varepsilon$

$A \rightarrow A\alpha$
$A \rightarrow \beta$

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A''$
$A' \rightarrow \epsilon$

Elimination of Left Recursion(2)

Grammar:
$A \rightarrow A\alpha$
$A \rightarrow \beta$

Above grammar can generate strings of format: $\beta\alpha^*$ (β, $\beta\alpha$, $\beta\alpha\alpha$, $\beta\alpha\alpha\alpha$, ……)

Same set of strings will be generated by the following grammar:

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A'$
$A' \rightarrow \varepsilon$

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

>>

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

Elimination of Left Recursion (3)

A set of “A productions” with left recursion:

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots\ldots\ldots\ldots \mid A\alpha_n$
$A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots\ldots\ldots \mid \beta_k$

Modified productions without left recursions keeping the recognized language intact:

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots\ldots \mid \beta_k A'$
$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots\ldots\ldots\ldots \mid \alpha_n A'$
$A' \rightarrow \varepsilon$

Left Factoring

Consider a set of A-productions where first symbol(s) of the right side are exactly same:

$A \rightarrow bC$		$A \rightarrow baC$
$A \rightarrow bD$	or	$A \rightarrow baD$
$A \rightarrow bE$		$A \rightarrow baE$

.

These form of ambiguity must be removed by "left factoring" process as follows:

$A \rightarrow bH$ or $A \rightarrow baH$

$H \rightarrow C \mid D \mid E$

Predictive Parser (Top-down or LL)

TMC @ CSE.IUT

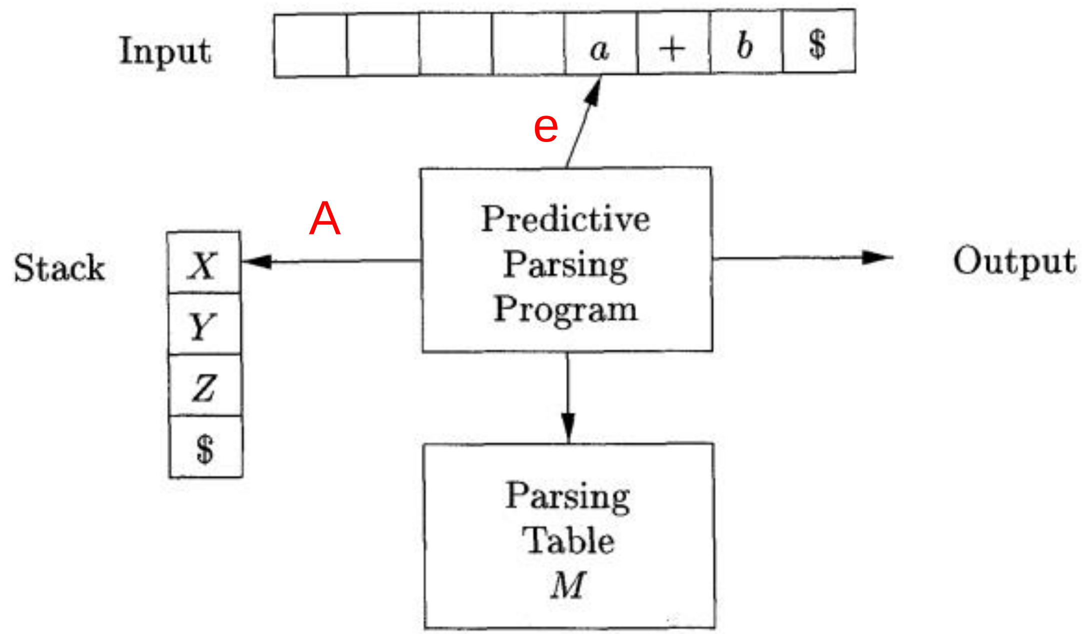


Fig: General structure of a predictive parser

Parsing Algorithm

do while 1
1. If A = e = $, parsing is complete
2. else if A = e <> $ pop A from stack and advance the input pointer
3. else if A is a nonterminal, check entry at M[A,e]. If the entry is an A production then replace A by right side of A production in reverse order.
4. else report an error to error handler for present input symbol e.

Predictive Parser: Parse Table, M

TMC @ CSE.IUT

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	$F \to \mathbf{id}$			$F \to (E)$		

Predictive parser: parsing steps

Modified Grammar

1. E → T E’
2. E’ → +TE’
3. E’ → ϵ
4. T → FT’
5. T’ → *FT’ | ϵ
6. T’ → ϵ
7. F → (E)
8. F → **id**

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	$F \to \mathbf{id}$			$F \to (E)$		

Parse Table

Stack

E'$

Output

id

A e

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 $E' \to \epsilon$

Parsing Steps

Parsing Algorithm

do while 1
1. If A = e = $, parsing is complete

2. else if A = e <> $ pop **A** from the stack and advance the input pointer

3. else if A is a nonterminal, check entry at **M[A,e]**. If the entry is an A production then replace A by right side of A production in reverse order.

4. else report an error to error handler for present input symbol e.

Construction of Parse Table

Steps:

- Eliminate *Left-Recursion* from the Grammar
- If necessary perform *Left Factoring*
- Find set of *First(X) and Follow(X)*
- Build the parse table

Predictive parser: Eliminate left recursion

TMC @ CSE.IUT

Grammar

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → **id**

Modified Grammar

1. E → T E’
2. E’ → +TE’
3. E’ → ϵ
4. T → FT’
5. T’ → *FT’ | ϵ
6. T’ → ϵ
7. F → (E)
8. F → **id**

Sentential Form

A **sentential form** is any string derivable from the start symbol of a grammar. Note that this includes the forms with non-terminals at intermediate steps as well.

1.	$E \rightarrow E + T$
2.	$E \rightarrow T$
3.	$T \rightarrow T * F$
4.	$T \rightarrow F$
5.	$F \rightarrow (E)$
6.	$F \rightarrow \textbf{id}$

$$
\left.\begin{aligned}
E &\rightarrow E + T \\
&\rightarrow T + T \\
&\rightarrow T * F + T \\
&\rightarrow F * F + T \\
&\rightarrow id * F + T \\
&\rightarrow id * id + T \\
&\rightarrow id * id + F \\
&\rightarrow id * id + id
\end{aligned}\right\}
$$

Right Sentential Form

A **right sentential form** is a sentential form that occurs in steps of rightmost derivation (RMD).

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → **id**

E → E + T
→ E + F
→ E + id
→ T + id
→ T * F + id
→ T * id + id
→ F * id + id
→ id * id + id

Right Sentential Form

Left Sentential Form

A **left sentential form** is a sentential form that occurs in steps of leftmost derivation (LMD).

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow \textbf{id}$

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

Left Sentential Form

First (X) and Follow(X)

TMC
@
CSE.IUT

If X is a non-terminal then First(X) is the set of terminals that can appear at the beginning of its sentential form. if X is terminal, First (X) is X itself.

A → bC | Ce
C → d | f

First(bC) = {b}
First(Ce) = {d,f}

A = {b, d, f}
C = {d, f}

A → bC → bd
A → Ce → de
A → Ce → fe

First(A) = {b,d,f}
First(C) = {d,f}

bd

Follow(X) is a set of terminals that may appear after X in a sentential form.

A → Ce → de
A → Ce → fe

Follow(C) = {e}

First (X)

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

Predictive parser: First (X)

Modified Grammar

1. E → T E’
2. E’ → +TE’
3. E’ → ϵ
4. T → FT’
5. T’ → *FT’
6. T’ → ϵ
7. F → (E)
8. F → **id**

First
E = { (, id }
E' = { +, ϵ }
T = { (, id }
T' = { *, ϵ }
F = { (, id }

First (E) = First (T) = First (F) = { (, **id** }
First (E’) = { +, ϵ }
First (T’) = { *, ϵ }

Predictive parser: First (X)

For a production A → α; First(A) and First(α) may not be equal. Because the grammar may contain multiple A productions like follows-

A → α

A → β

…..

$\boxed{\text{First}(A) = \text{First}(\alpha) \cup \text{First}(\beta)}$

So, for a production A → α; First(α) ⊆ First(A).

Follow (X)

TMC
@
CSE.IUT

Follow(X) to be the set of terminals that can appear immediately to the right of Non-Terminal X in some sentential form.

Algorithm:

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where FIRST($\beta$) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).
4. Follow(X) never contains ϵ.

Predictive parser: Follow (X)

TMC @ CSE.IUT

E = {), $ }
E' = {), $ }
T = { +,), $ }
T' = { +,), $ }

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.
2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

follow

Modified Grammar

1. E → TE'
2. E' → +TE'
3. E' → ϵ
4. T → FT'
5. T' → *FT' | ϵ
6. T' → ϵ
7. F → (E)
8. F → **id**

F = { *, +,), $ }

Follow(E) = { \$, …. }
Follow(T) ← First (E') except ϵ [+]
Follow(F) ← First (T') except ϵ [*]
Follow(E) ← ')'

Follow(E') ← Follow(E)
Follow(T') ← Follow(T)
Follow(T) ← Follow(E)
Follow(F) ← Follow(T)

E → T E'
First(E')
E → T
E → T F
E → T ( )

First (E) = First (T) = First (F) = { (, **id** }
First (E') = { +, ϵ }
First (T') = { *, ϵ }

Follow (E) = Follow (E') = {), \$ }
Follow (T) = Follow (T') = { +,), \$ }
Follow(F) = {+, *,), \$ }

Predictive parser: Generating parse table

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

E → (E

For each production A → α, do the following.

1. For each terminal a in FIRST(α), add $A \to \alpha$ 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.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table).

Algorithm to generate parse table

First (E) = First (T) = First (F) = { (, **id** }
First (E') = { +, ϵ }
First (T') = { *, ϵ }

(, id

Follow (E) = Follow (E') = {), \$ }
Follow (T) = Follow (T') = { +,), \$ }
Follow(F) = {+, *,), \$ }

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	$F \to \mathbf{id}$			$F \to (E)$		

Types of Parsing

TMC @ CSE.IUT

- **Top-Down Parsing (LL)**
  - **Recursive Descent Parsing**
  - **Nonrecursive Predictive Parsing**
- **Bottom-Up Parsing (LR)**
  - **Simple LR (SLR)**
  - **Canonical LR (CLR)**
  - **Look Ahead LR (LALR)**

LL: Scan the input from Left to Right, Use Left most derivation

LR: Scan the input from Left to Right, Use Right most derivation in Reverse order

Some Definitions for LR Parser

- Viable Prefix
- Handle
- Handle Pruning

Viable Prefix

TMC @ CSE.IUT

The prefix of a right sentential form which may appear in parser stack is known as **viable prefix**.

Handle and handle pruning

TMC @ CSE.IUT

A Handle is a substring of a right sentential form that matches the body (right side) of a production. Handle always appears at top of the stack.

Handle reduction is a step in the reverse of rightmost derivation. A rightmost derivation in reverse can be obtained by handle pruning.

Grammar

1. E → E + E
2. E → E * E
3. E → -E
4. E → (E)
5. E → **id**
6. E → **num**

Input: id+id*id

E → id + id * id

E → id + id * E *handle*

E → id + E * E

E → id + E *handle pruning*

Bottom-up parsing (LR)

L = Scan the input from Left to Right
R = Rightmost derivation in Reverse Order

Types of LR Parsing

- Operator-precedence parsing
- Simple LR (SLR)
- Canonical LR (CLR)
- Look Ahead LR (LALR)

LR Parser

TMC @ CSE.IUT

An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse. States are generated from the grammar.

Shift - shift to another state.

***Reduce* –** apply a reduction operation using handle.

States represent sets of "items."

LR Parser States

States represent sets of "items.“

An LR(0) item (item for short) of a grammar G is a production of G with a dot at some position of the body. Thus, production A → XYZ yields the four items-

A → .XYZ
A → X.YZ
A → XY.Z
A → XYZ.

The production A → *empty* generates only one item, A → .

SLR Parser

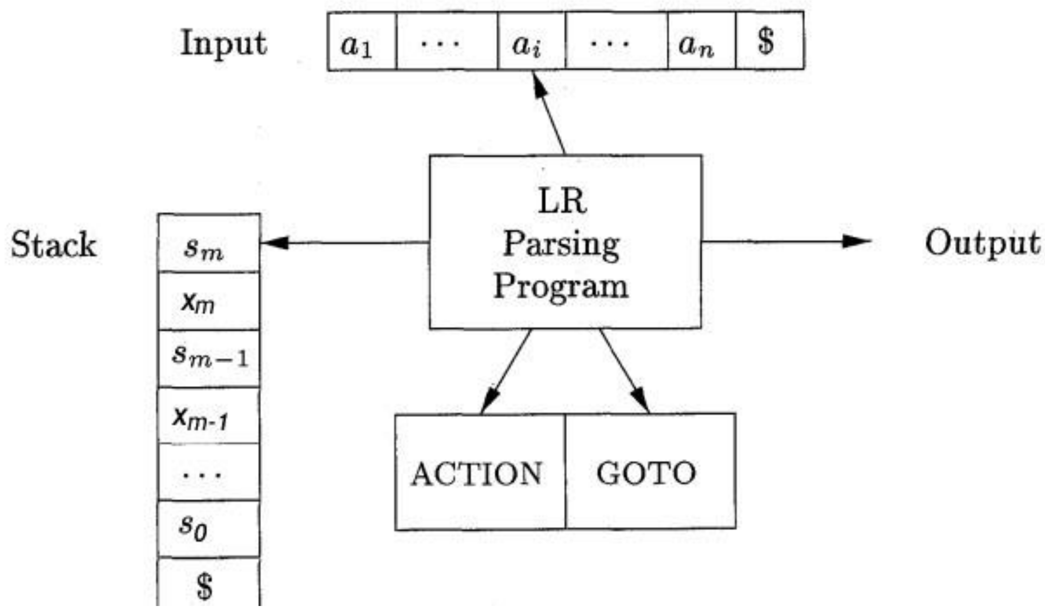
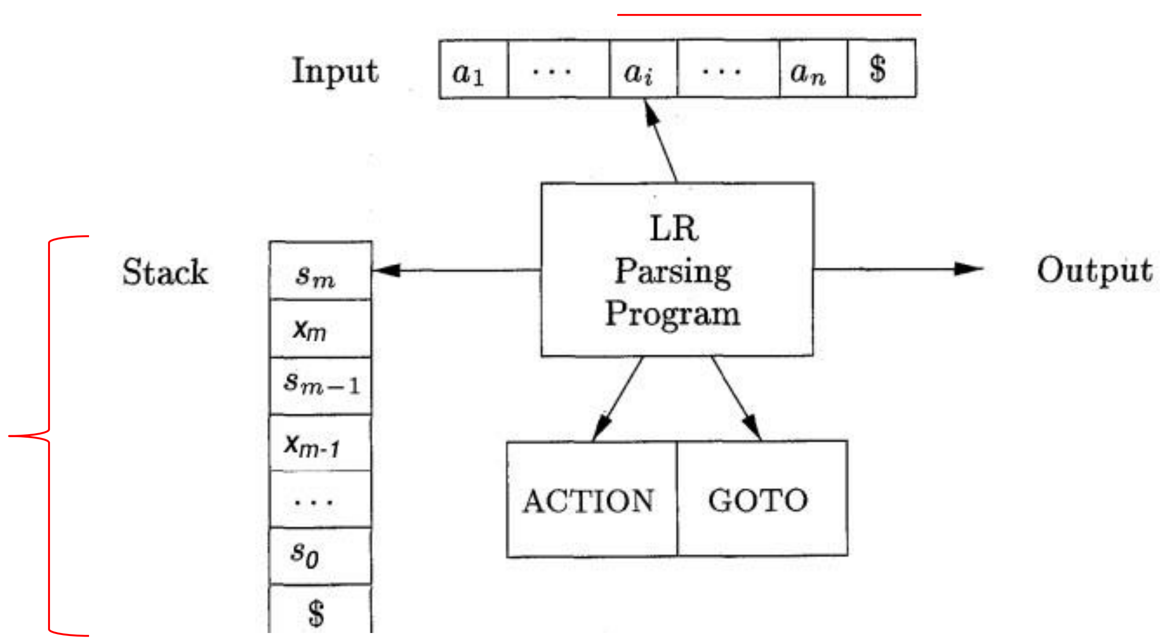


Fig: General Diagram of an SLR parser

Configuration of an SLR parser

A pair whose first component is the stack and second component is the unprocessed (unexpanded) input.



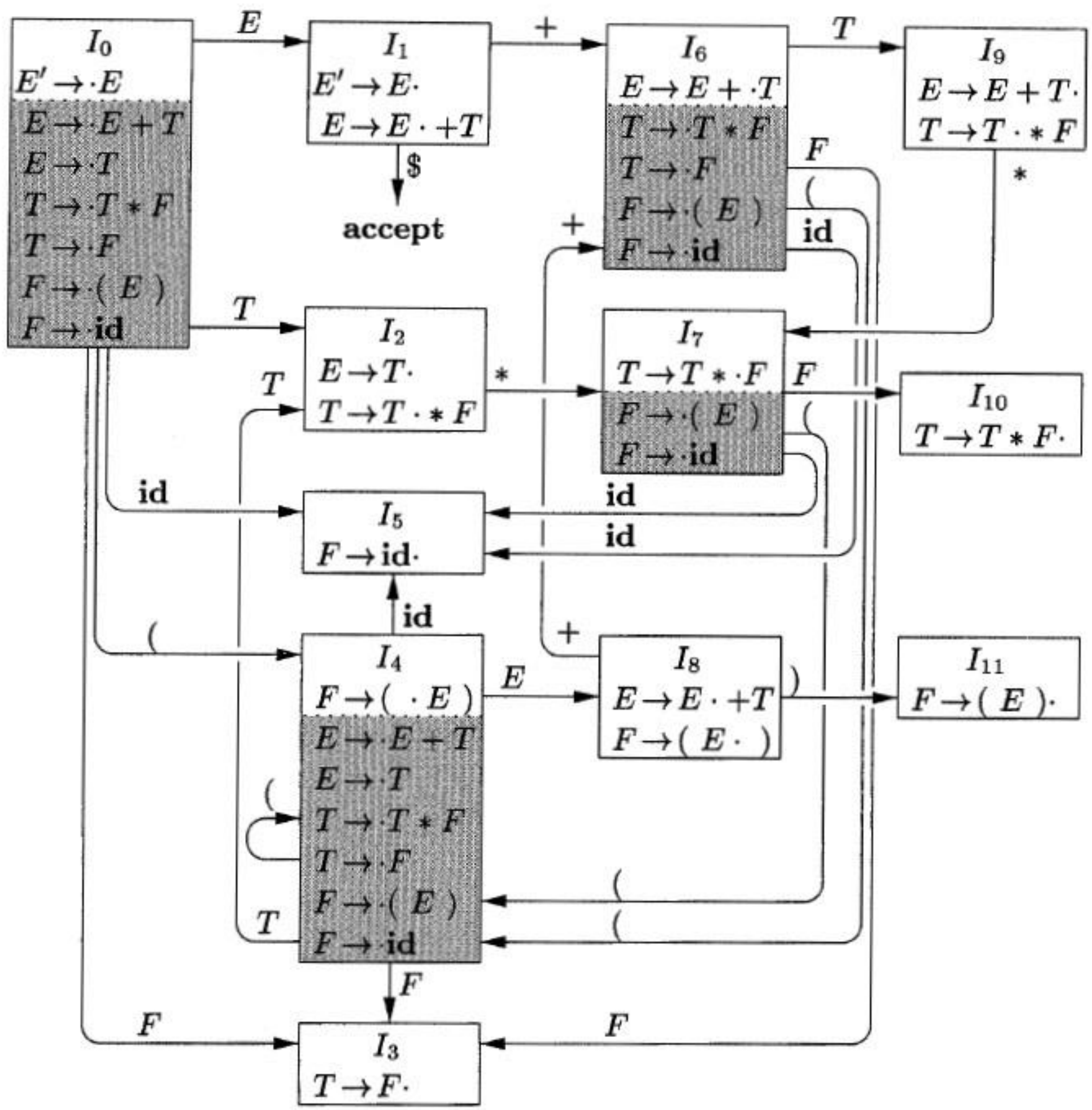
SLR: Transition diagram for LR(0) items

G

E → E + T
E → T
T → T * F
T → F
F → (E)
F → **id**

G'

E' → E
E → E + T
E → T
T → T * F
T → F
F → (E)
F → **id**



Closure of Item Sets

TMC @ CSE.IUT

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 $B \rightarrow .\gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

SLR Parse Table Generation

TMC @ CSE.IUT

1. Construct $C = \{I_0, I_1, \dots, I_n\}$, 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."

   If any conflicting actions result from the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

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 $[S' \to \cdot S]$.

SLR parse table

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → **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			

s6: Shift present token to stack and move to state 6
r6: reduce by production 6 and move to a new state (use goto section)

SLR 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;
}
```

Figure 4.36: LR-parsing program

Moves of SLR parser on id*id+id$

TMC @ CSE.IUT

	STACK	SYMBOLS	INPUT	ACTION
(1)	0	$	$\mathbf{id} * \mathbf{id} + \mathbf{id}\ \$$	shift
(2)	0 5	$ **id**	$* \mathbf{id} + \mathbf{id}\ \$$	reduce by $F \to \mathbf{id}$
(3)	0 3	$ F	$* \mathbf{id} + \mathbf{id}\ \$$	reduce by $T \to F$
(4)	0 2	$ T	$* \mathbf{id} + \mathbf{id}\ \$$	shift
(5)	0 2 7	$ $T *$	$\mathbf{id} + \mathbf{id}\ \$$	shift
(6)	0 2 7 5	$ $T * \mathbf{id}$	$+ \mathbf{id}\ \$$	reduce by $F \to \mathbf{id}$
(7)	0 2 7 10	$ $T * F$	$+ \mathbf{id}\ \$$	reduce by $T \to T * F$
(8)	0 2	$ T	$+ \mathbf{id}\ \$$	reduce by $E \to T$
(9)	0 1	$ E	$+ \mathbf{id}\ \$$	shift
(10)	0 1 6	$ $E +$	$\mathbf{id}\ \$$	shift
(11)	0 1 6 5	$ $E + \mathbf{id}$	$	reduce by $F \to \mathbf{id}$
(12)	0 1 6 3	$ $E + F$	$	reduce by $T \to F$
(13)	0 1 6 9	$ $E + T$	$	reduce by $E \to E + T$
(14)	0 1	$ E	$	accept

Moves of SLR parser on id*id+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			

Stack: Symbols:

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → **id**

	STACK	SYMBOLS	INPUT	ACTION
(1)	0	$	**id** * **id** + **id** $	shift
(2)	0 5	$ **id**	* **id** + **id** $	reduce by $F \to$ **id**
(3)	0 3	$ F	* **id** + **id** $	reduce by $T \to F$
(4)	0 2	$ T	* **id** + **id** $	shift
(5)	0 2 7	$ $T *$	**id** + **id** $	shift
(6)	0 2 7 5	$ $T *$ **id**	+ **id** $	reduce by $F \to$ **id**
(7)	0 2 7 10	$ $T * F$	+ **id** $	reduce by $T \to T * F$
(8)	0 2	$ T	+ **id** $	reduce by $E \to T$
(9)	0 1	$ E	+ **id** $	shift
(10)	0 1 6	$ $E +$	**id** $	shift
(11)	0 1 6 5	$ $E +$ **id**	$	reduce by $F \to$ **id**
(12)	0 1 6 3	$ $E + F$	$	reduce by $T \to F$
(13)	0 1 6 9	$ $E + T$	$	reduce by $E \to E + T$
(14)	0 1	$ E	$	accept

Weakness of SLR Parser

TMC @ CSE.IUT

Kernel item

Grammar

1. S → L = R
2. S → R
3. L → * R
4. L → **id**
5. R → L

I_0: $\boxed{S' \rightarrow \cdot S}$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S \cdot$

I_2: $S \rightarrow L \cdot = R$
$R \rightarrow L \cdot$

I_3: $S \rightarrow R \cdot$

I_4: $L \rightarrow * \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_5: $L \rightarrow \mathbf{id} \cdot$

I_6: $\boxed{S \rightarrow L = \cdot R}$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_7: $L \rightarrow * R \cdot$

I_8: $R \rightarrow L \cdot$

I_9: $S \rightarrow L = R \cdot$

=; s6

=; r5

state	action			
	=	*	id	$
0	……			
1	…….			
2	s6/r5			
.	…….			
.	…….			

shift-reduce conflict

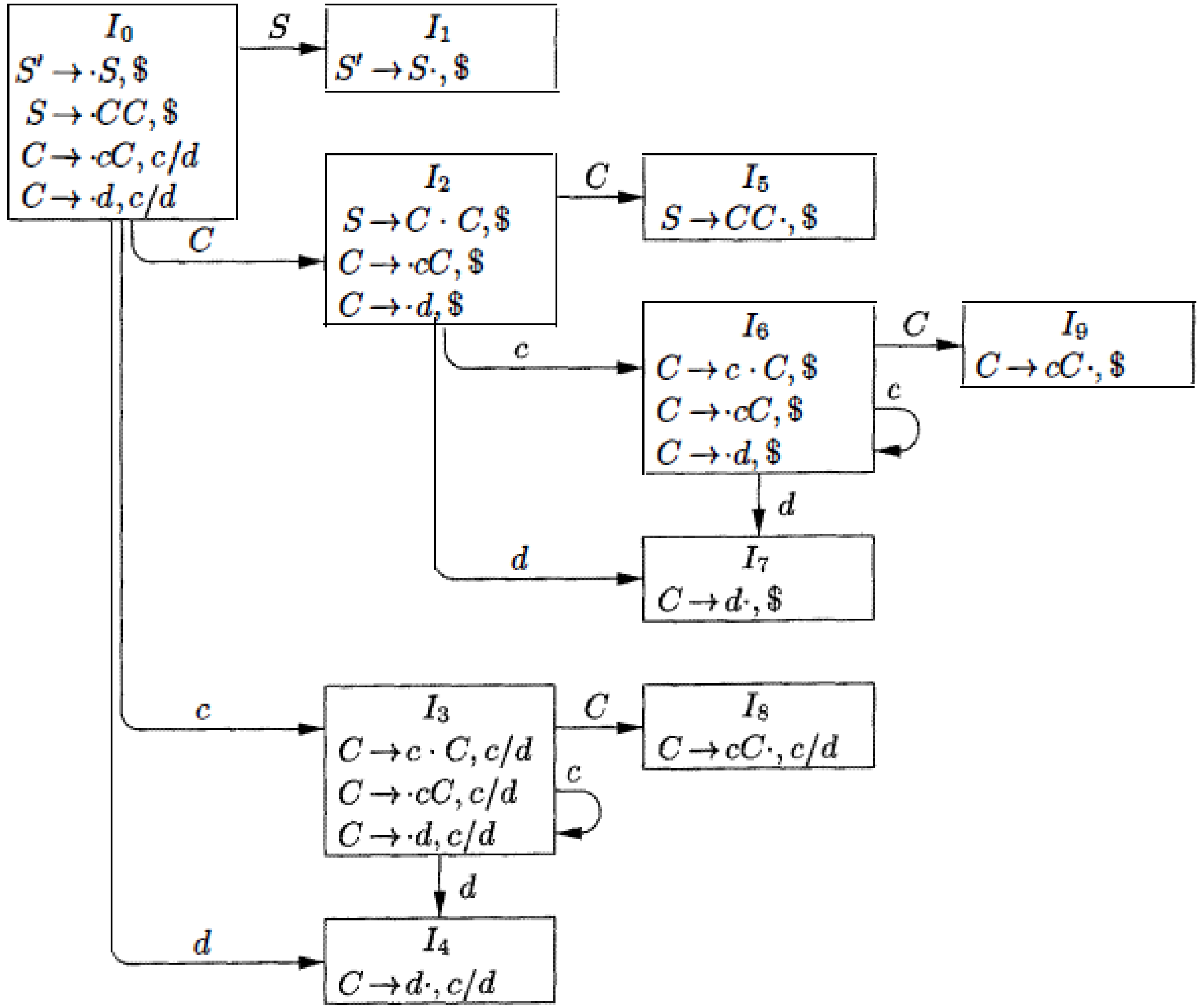
Fig: Canonical LR(0) Collections

CLR States and Transition diagram

A → α.Bβ, a

closure:

B → .γ, First(βa)



Grammar

1. S → CC
2. C → cC
3. C → d

Fig: Canonical LR(1) Collections / GOTO Graph

CLR Parse Table

TMC @ CSE.IUT

Grammar
1. S → CC
2. C → cC
3. C → d

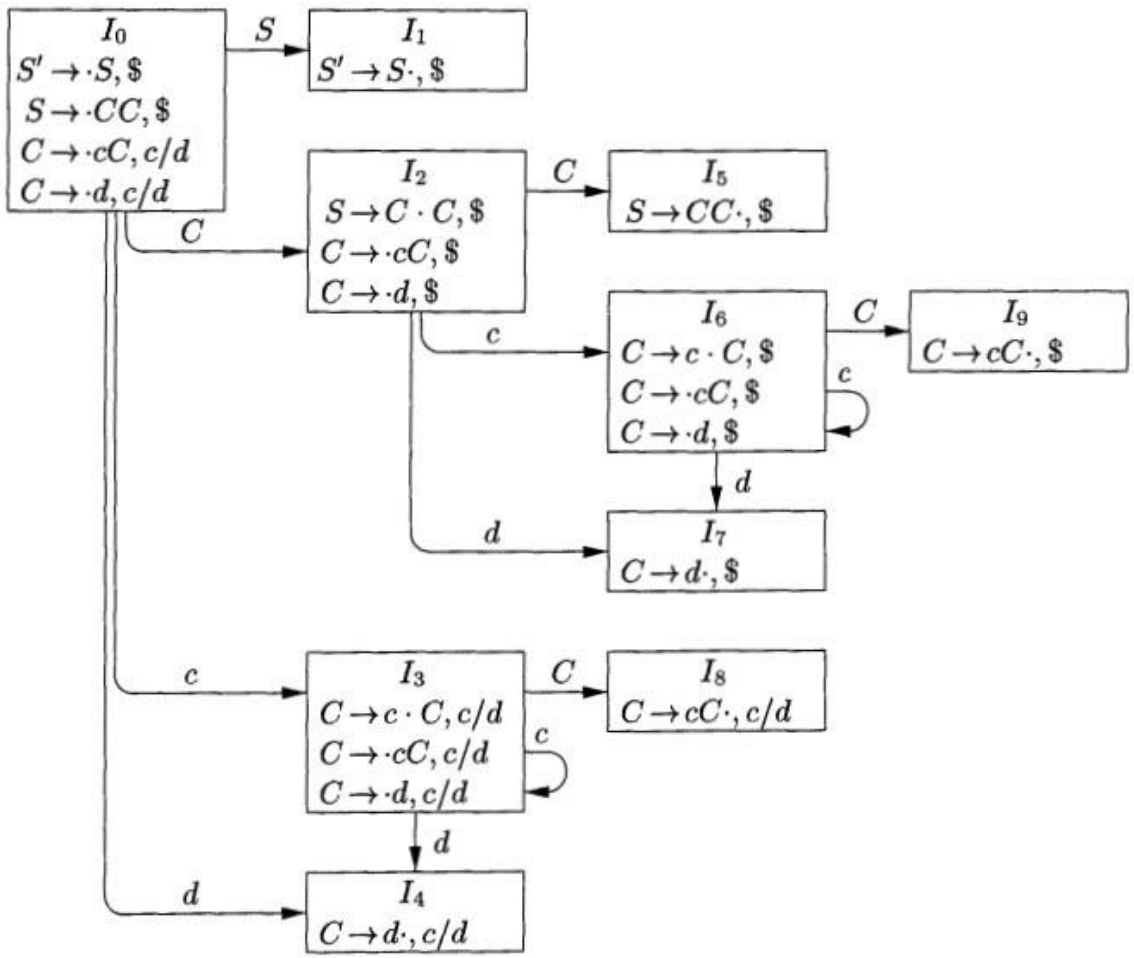
STATE	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Fig: CLR Parse Table

LALR from CLR states

Grammar

1. $S \rightarrow CC$
2. $C \rightarrow cC$
3. $C \rightarrow d$



$$
\begin{aligned}
I_{36}: \quad & C \rightarrow c\cdot C, \; c/d/\$ \\
& C \rightarrow \cdot cC, \; c/d/\$ \\
& C \rightarrow \cdot d, \; c/d/\$ \\
I_{47}: \quad & C \rightarrow d\cdot, \; c/d/\$ \\
I_{89}: \quad & C \rightarrow cC\cdot, \; c/d/\$
\end{aligned}
$$

Fig: Canonical LR(1) Collections

LALR Parse Table

Grammar
1. S → CC
2. C → cC
3. C → d

STATE	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Fig: LALR parse table