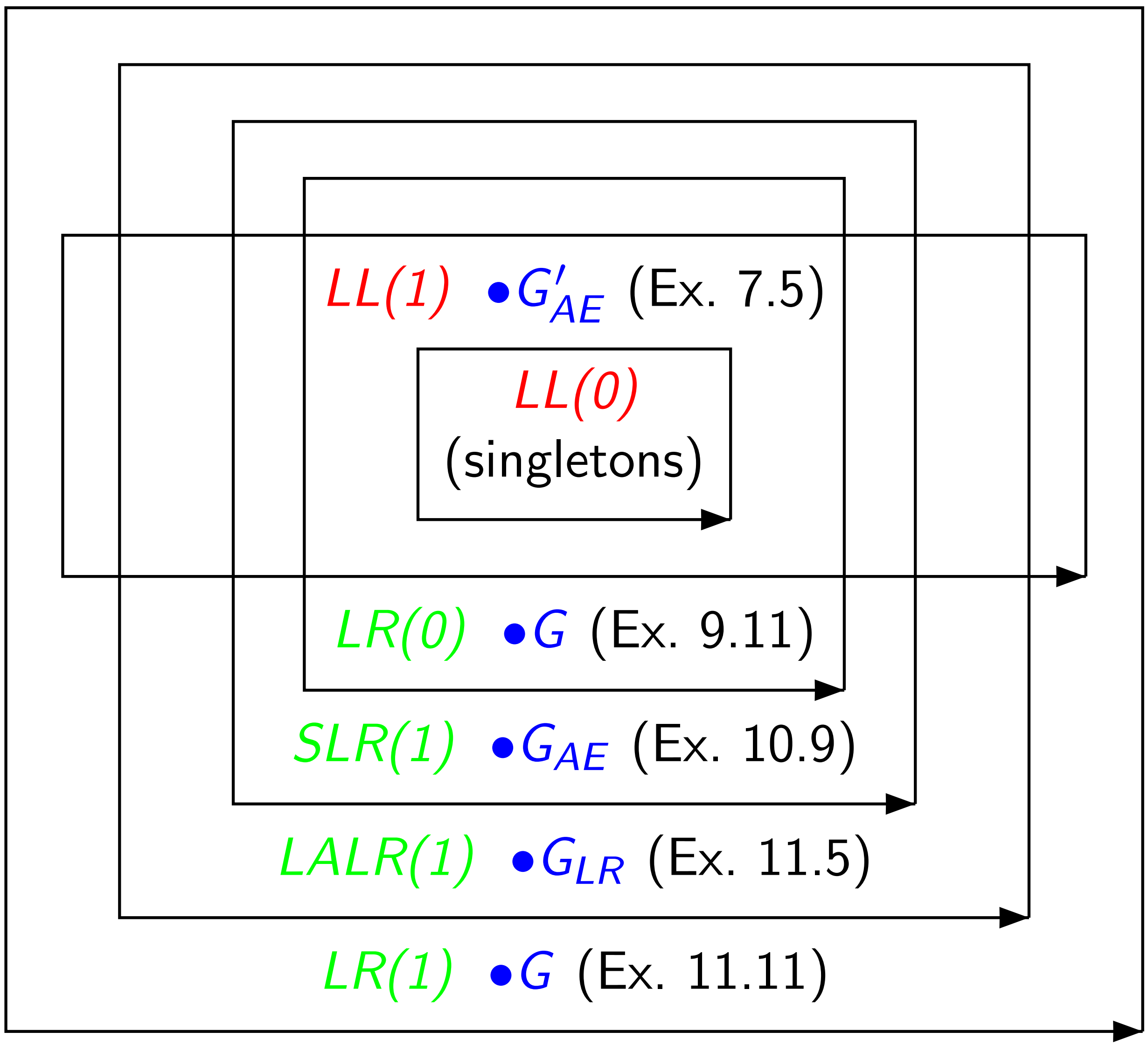
بسم الله الرحمن الرحيم

Parsing:

Bottom-Up Parsing
Error Recovery

Recall

- LL(1)
- SLR
- LR(0)
- LR(1)
- LALR

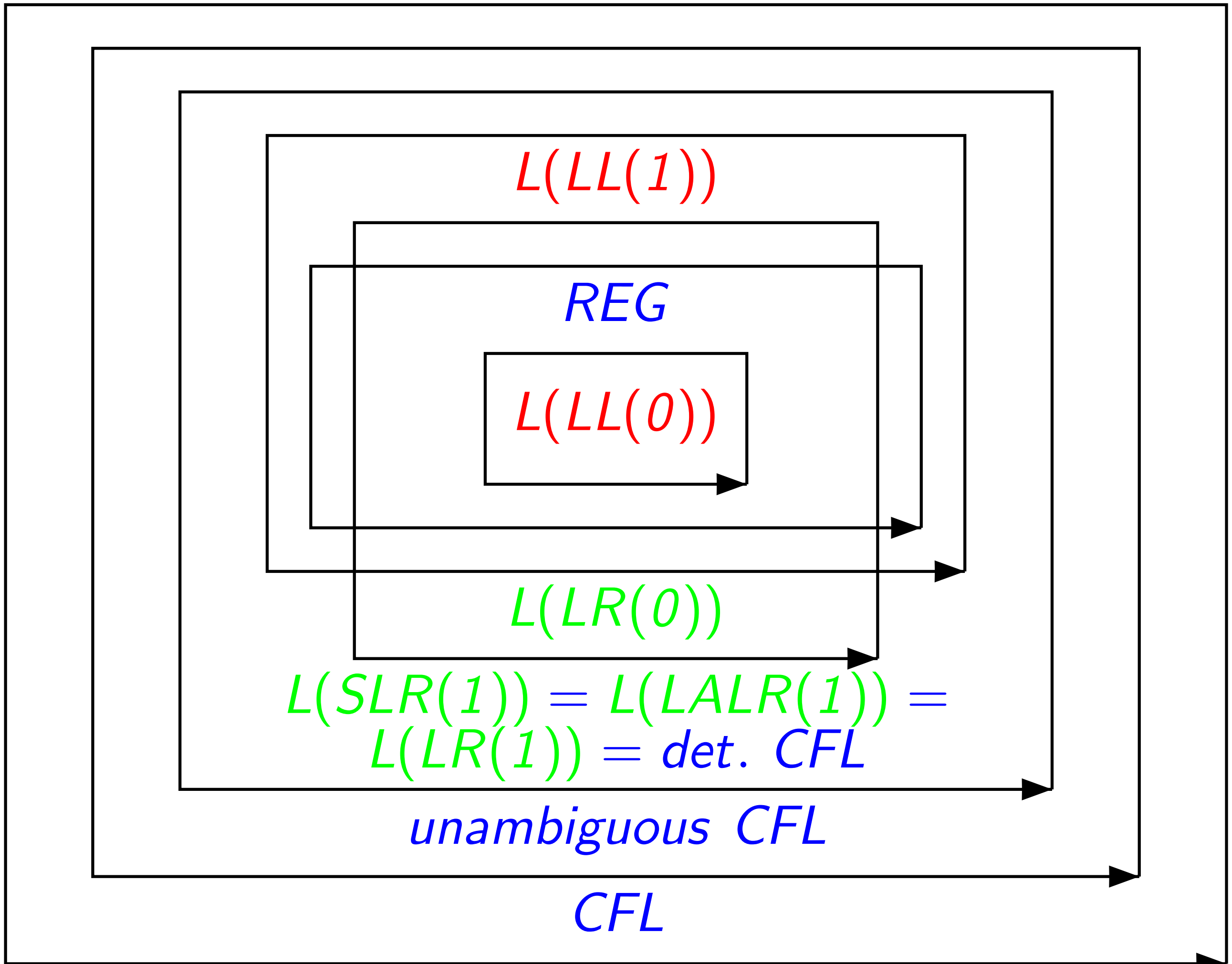


Moreover:

- $LL(k) \subsetneq LL(k+1)$ for every $k \in \mathbb{N}$
- $LR(k) \subsetneq LR(k+1)$ for every $k \in \mathbb{N}$
- $LL(k) \subseteq LR(k)$ for every $k \in \mathbb{N}$

Overview of Language Classes

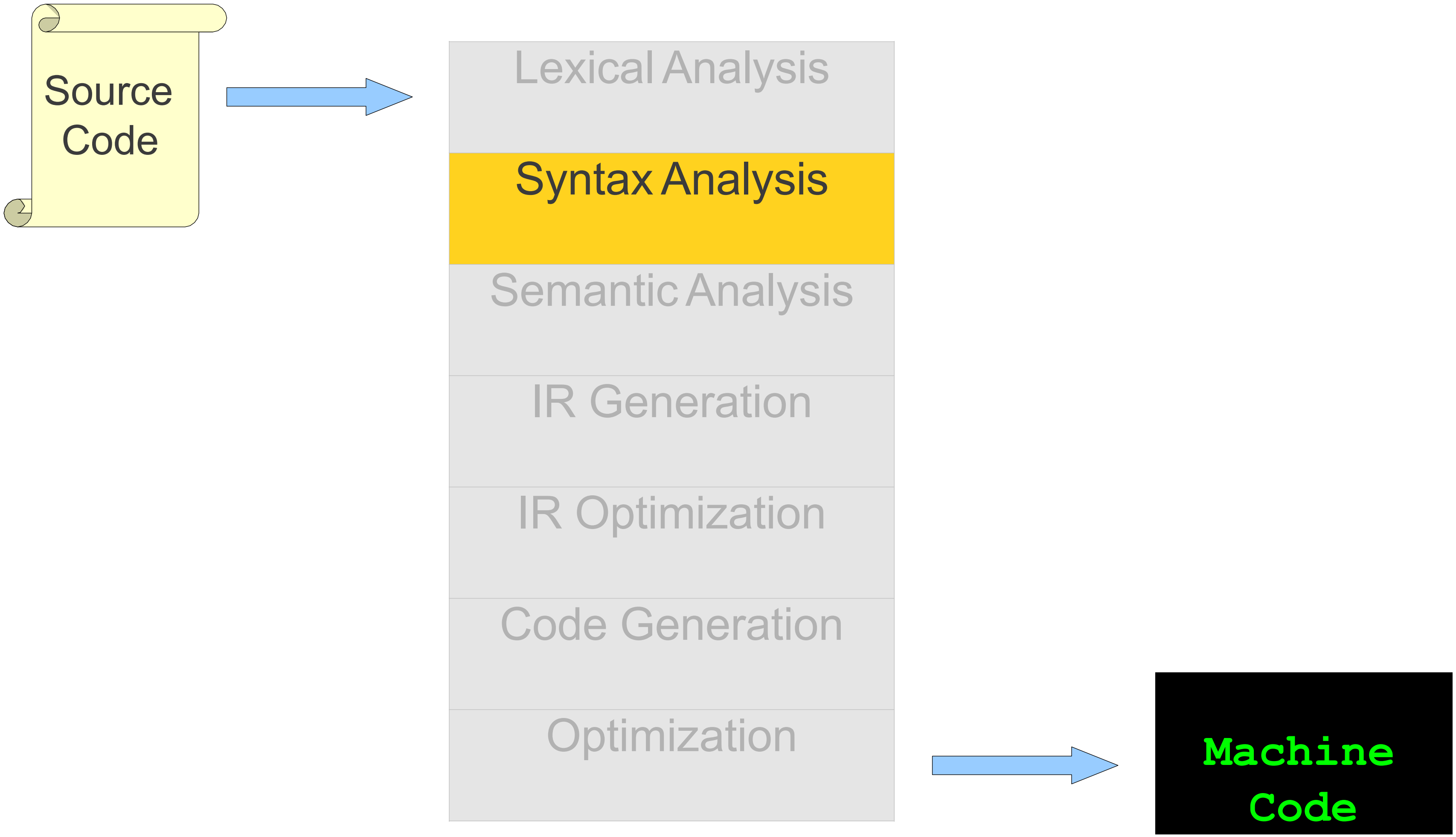
(cf. O. Mayer: *Syntaxanalyse*, BI-Verlag, 1978, p. 409ff)



Moreover:

- $L(LL(k)) \subsetneq L(LL(k+1)) \subsetneq L(LR(1))$ for every $k \in \mathbb{N}$
- $L(LR(k)) = L(LR(1))$ for every $k \geq 1$

Where are we?


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Error Recovery

Why Error Handling?

Why Error Handling?

- It is very simple to design compilers just for parsing correct programs.

Why Error Handling?

- It is very simple to design compilers just for parsing correct programs.
- It is expected to **assist** programmers in **locating** and **tracking** errors.

Why Error Handling?

- It is very simple to design compilers just for parsing correct programs.
- It is expected to **assist** programmers in **locating** and **tracking** errors.
- It is important in Parsing:
  - Finding error as soon as possible.
  - Most of errors occurs in parsing phase. (Actually most of errors we can find efficiently!)

Types of Errors

- **Lexical Errors**: misspellings of identifiers, keywords, or operators and etc.
- **Syntactic Errors**: e.g. misplaced semicolons or extra or missing braces and etc.
- **Semantic Errors**: type mismatches between operators and operands, or actual and formal parameters.
- **Logical Errors**: any part of code which does not reflect the intent of programmer.

Goals of Error Handling

- **Report** the presence of error (As accurate and clear as possible).
- **Recover** from each error to detect subsequent errors.

- However we shouldn’t make parsing less efficient.

Strategies

- Let’s learn some strategies and common solutions.

Strategies

- Let’s learn some strategies and common solutions.
- We will work on follows:
  - Panic Mode
  - Phrase-level
  - Error Productions
  - Global Correction

Strategies

- Let’s learn some strategies and common solutions.
- We will work on follows:
  - Panic Mode
  - Phrase-level
  - Error Productions
  - Global Correction
- Be careful: no strategy is universally acceptable.

Panic Mode Recovery

- There is a synchronizing set of tokens:
  - Consist of unambiguous-role tokens such as delimiters such as semicolon or }.
  - Compiler designer must choose this set.

Panic Mode Recovery

- There is a synchronizing set of tokens:
  - Consist of unambiguous-role tokens such as delimiters such as semicolon or }.
  - Compiler designer must choose this set.
- The effectiveness of this method rely on choosing this set.

Panic Mode Recovery

- There is a synchronizing set of tokens:
  - Consist of unambiguous-role tokens such as delimiters such as semicolon or }.
  - Compiler designer must choose this set.
- The effectiveness of this method rely on choosing this set.

- On discovering error, parser discard input **symbols** until one of synchronizing set's token.

Panic Mode Recovery

- Pros:
  - Simplicity
  - Not go into infinite loop

- Cons:
  - Skip a considerable amount of input

Top-Down Parsers with Panic Mode

- In LL(1) an empty parse table entry shows **error**!
- E.g. if top stack is A and input token is a, and `M[A, a]` is empty, we face an error.

Top-Down Parsers with Panic Mode

- In LL(1) an empty parse table entry shows **error**!
- E.g. if top stack is A and input token is a, and `M[A, a]` is empty, we face an error.
- We have same situation in RD approach in a function.
- We can have synchronizing set for each variable.

Heuristics

- Sync(A) = follow (A).

Heuristics

- Sync(A) = follow (A).
  - Good when some extra symbols used wrongly.
  - Not enough: What if a semicolon is forgotten? So…

Heuristics

- Sync(A) = follow (A).
  - Good when some extra symbols used wrongly.
  - Not enough: What if a semicolon is forgotten? So…
- Add keywords begin an Statement.

Heuristics

- Sync(A) = follow (A).
  - Good when some extra symbols used wrongly.
  - Not enough: What if a semicolon is forgotten? So…
- Add keywords begin an Statement.
- Sync(A) += first (A).
  - Enables continue parsing as the chance arise!

Heuristics

- Sync(A) = follow (A).
  - Good when some extra symbols used wrongly.
  - Not enough: What if a semicolon is forgotten? So…
  - Add keywords begin an Statement.
- Sync(A) += first (A).
  - Enables continue parsing as the chance arise!
  - Pop, terminal from stack!

LL(1): Example

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

	id	+	*	(	)	$
E	1			1	**sync**	**sync**
E’		2			3	3
T	4	**sync**		4	**sync**	**sync**
T’		6	5		6	6
F	8	**sync**	**sync**	7	**sync**	**sync**

Stack	Input	Note
E $	) id * + id $	

LL(1): Example

STACK	INPUT	REMARK
$E\,\$$	$)\,\mathbf{id} * + \mathbf{id}\,\$$	error, skip)
$E\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	**id** is in FIRST(E)
$TE'\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$FT'E'\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$\mathbf{id}\ T'E'\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$T'E'\,\$$	$* + \mathbf{id}\,\$$	
$* FT'E'\,\$$	$* + \mathbf{id}\,\$$	
$FT'E'\,\$$	$+ \mathbf{id}\,\$$	error, $M[F,+] =$ synch
$T'E'\,\$$	$+ \mathbf{id}\,\$$	F has been popped
$E'\,\$$	$+ \mathbf{id}\,\$$	
$+ TE'\,\$$	$+ \mathbf{id}\,\$$	
$TE'\,\$$	$\mathbf{id}\,\$$	
$FT'E'\,\$$	$\mathbf{id}\,\$$	
$\mathbf{id}\,T'E'\,\$$	$\mathbf{id}\,\$$	
$T'E'\,\$$	$\$$	
$E'\,\$$	$\$$	
$\$$	$\$$	

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

	id	+	*	(	)	$
E	1			1	**sync**	**sync**
E’		2			3	3
T	4	**sync**		4	**sync**	**sync**
T’		6	5		6	6
F	8	**sync**	**sync**	7	**sync**	**sync**

LR-Parsers with Panic Mode

- Choose some non-terminals like A, representing major program pieces, such as an expr, stmt, or block.

LR-Parsers with Panic Mode

- Choose some non-terminals like A, representing major program pieces, such as an expr, stmt, or block.
- On face error, go to PS and pop states, until find a **GOTO** with a non-terminal A which was chosen before.
- We discard input until we find a $\in$ follow(A) (LA).
- So we resume parsing.
- E.g. `A = stmt` implies `a = semicolon`, which indicate end of statement.

Phrase-Level Recovery

- When an error is discovered, parser perform a local correction which enables parser to continue. Such as:
  - Replace comma with semicolon.
  - Insert/Delete a semicolon.

Phrase-Level Recovery

- Pros:
  - It can correct many errors.
  - It is very effective for continue parsing.

- Cons:
  - We may go into infinite loop:
    - Consider we always add some string.
  - Major drawback: when the error occurs before the point of detection. (Why?)
    - Example: ?

Top-Down Error Recovery with Phrase-Level

- Fill the blank entries of parse table with pointer to **error recovery routines**.
  - They change symbols, or add or remove them on input and print error massages.
  - They may pop sth from stack.
  - (change stack is not recommended, why?)
- At end we should check, routines do not result an infinite loop.

Bottom-Up Parsing with Phrase Level

- We should fill the blank cells with routine again.
- Try to *guess* what makes this problem.
- Consequently, routines can correct it.

Example: missing operator

STATE	ACTION						GOTO
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

$$
\begin{array}{ll}
I_0: & E' \to \cdot E \\
 & E \to \cdot E + E \\
 & E \to \cdot E * E \\
 & E \to \cdot (E) \\
 & E \to \cdot \mathbf{id}
\end{array}
$$

$$
\begin{array}{ll}
I_1: & E' \to E \cdot \\
 & E \to E \cdot + E \\
 & E \to E \cdot * E
\end{array}
$$

$$
\begin{array}{ll}
I_2: & E \to (\cdot E) \\
 & E \to \cdot E + E \\
 & E \to \cdot E * E \\
 & E \to \cdot (E) \\
 & E \to \cdot \mathbf{id}
\end{array}
$$

$$
\begin{array}{ll}
I_3: & E \to \mathbf{id} \cdot
\end{array}
$$

$$
\begin{array}{ll}
I_4: & E \to E + \cdot E \\
 & E \to \cdot E + E \\
 & E \to \cdot E * E \\
 & E \to \cdot (E) \\
 & E \to \cdot \mathbf{id}
\end{array}
$$

$$
\begin{array}{ll}
I_5: & E \to E * \cdot E \\
 & E \to \cdot E + E \\
 & E \to \cdot E * E \\
 & E \to \cdot (E) \\
 & E \to \cdot \mathbf{id}
\end{array}
$$

$$
\begin{array}{ll}
I_6: & E \to (E \cdot) \\
 & E \to E \cdot + E \\
 & E \to E \cdot * E
\end{array}
$$

$$
\begin{array}{ll}
I_7: & E \to E + E \cdot \\
 & E \to E \cdot + E \\
 & E \to E \cdot * E
\end{array}
$$

$$
\begin{array}{ll}
I_8: & E \to E * E \cdot \\
 & E \to E \cdot + E \\
 & E \to E \cdot * E
\end{array}
$$

$$
\begin{array}{ll}
I_9: & E \to (E) \cdot
\end{array}
$$

Ref: adapted from Dr.Ghassemi’s Slides, University of Tehran

Example: missing operand

STATE	ACTION						GOTO
	id	$+$	$*$	(	)	\$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

I_0: $E' \to \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_1: $E' \to E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_2: $E \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_3: $E \to \mathbf{id} \cdot$

I_4: $E \to E + \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_5: $E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_6: $E \to (E \cdot)$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_7: $E \to E + E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_8: $E \to E * E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_9: $E \to (E) \cdot$

Ref: adapted from Dr.Ghassemi's Slides, University of Tehran

Example: unbalanced parenthesis

STATE	ACTION						GOTO
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

I_0: $E' \to \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_1: $E' \to E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_2: $E \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_3: $E \to \mathbf{id} \cdot$

I_4: $E \to E + \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_5: $E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_6: $E \to (E \cdot)$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_7: $E \to E + E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_8: $E \to E * E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_9: $E \to (E) \cdot$

Ref: adapted from Dr.Ghassemi's Slides, University of Tehran

Error Production Rule Recovery

- It is just for grammar-based parsers.

Error Production Rule Recovery

- It is just for grammar-based parsers.
- Compiler designer, **anticipate** common errors that might occur and add a production rule for them.

Error Production Rule Recovery

- It is just for grammar-based parsers.
- Compiler designer, **anticipate** common errors that might occur and add a production rule for them.
- It has almost no overhead and can generate appropriate error diagnostics about the erroneous parts.

Error Production Rule Recovery

- It is just for grammar-based parsers.
- Compiler designer, **anticipate** common errors that might occur and add a production rule for them.
- It has almost no overhead and can generate appropriate error diagnostics about the erroneous parts.
- This approach is used by YACC/Bison.

LR Parsers with Production Rule

- This approach is used by YACC/Bison.
- First, major non-terminals are selected.
- Then we add $A \rightarrow \boldsymbol{error}\ \alpha$ as a production rule for common errors.
- Rules semantic action may contain input changing or print error messages.
- Examples:
  - $stmt \rightarrow \boldsymbol{error}\ ;$
  - $stmt \rightarrow \boldsymbol{error}\ '\backslash n'$
  - $block \rightarrow \boldsymbol{error}\ '\}'$

Production Rule Recovery Procedure

- On Error, we pop from stack until found state s contains item $A \rightarrow .\ \boldsymbol{error}\ \alpha.$

Production Rule Recovery Procedure

- On Error, we pop from stack until found state s contains item $A \rightarrow . \; \boldsymbol{error} \; \alpha.$
- Then, parser shifts a mock **error** token.

Production Rule Recovery Procedure

- On Error, we pop from stack until found state s contains item $A \rightarrow .\ \boldsymbol{error}\ \alpha.$
- Then, parser shifts a mock **error** token.
- Case 1: $\alpha = \epsilon$
  - After the reduction, a routine can be called (we talk about call routines later!).

Production Rule Recovery Procedure

- On Error, we pop from stack until found state s contains item $A \rightarrow .\ \boldsymbol{error}\ \alpha$.
- Then, parser shifts a mock **error** token.
- Case 1: $\alpha = \epsilon$
  - After the reduction, a routine can be called (we talk about call routines later!).
  - The parser then discards input symbols until it finds an input symbol on which normal parsing can proceed.

Production Rule Recovery Procedure

- Case 2: $\alpha \neq \epsilon$
  - Skip input for a substring can be reduced to α . (Note: it may contains non-terminals also)

Production Rule Recovery Procedure

- Case 2: $\alpha \neq \epsilon$
  - Skip input for a substring can be reduced to α . (Note: it may contains non-terminals also)
  - If it contains only terminals to reduce α also to A.
  - After the reduction, a routine can be called
- Continue Parsing!

Global Correction

- Given incorrect program **P** and grammar **G**, a global correction algorithm, must find a program **Q**, with minimum **edit distance** in term of insertion, deletion or token change.
- It is too expensive!
- It is a standard for evaluating other techniques.