UNIT-III

Bottom-Up Parsing

Contents

- *Bottom up Parsing*
- *Reductions*
- *Handle Pruning*
- *Shift Reduce Parsing*
- *Problems related to Shift Reduce Parsing*
- *Conflicts during Shift Reduce Parsing*
- *Operator Precedence Parser*
- *Computation of LEADING*
- *Computation of TRAILING*
- *Problems related to LEADING AND TRAILING*
- *LR Parsers – Why LR Parsers*
- *Items and LR(0) Automation*
- *Closure of Item sets*
- *LR Parsing Algorithm*
- *SLR Grammars*
- *SLR Parsing Tables*
- *Problems related to SLR*
- *Construction of Canonical LR(1) and LALR*
- *Construction of LALR*
- *Problems related to Canonical LR(1) and LALR Parsing Table*

Bottom-Up Parsing
Introduction

- Bottom up parsing works in the opposite direction from top down.
- A top down parser begins with the start symbol at the top of the parse tree and works downward, driving productions in forward order until it gets to the terminal leaves.
- A bottom up parse starts with the string of terminals itself and builds from the leaves upward, working backwards to the start symbol by applying the productions in reverse.
- Along the way, a bottom up parser searches for substrings of the working string that match the right side of some production. When it finds such a substring, it reduces it, i.e., substitutes the left side non-terminal for the matching right side.
- The goal is to reduce all the way up to the start symbol and report a successful parse.

Contd…

- A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).
- It is the process of "reducing" a Input string *w* to the start symbol of the grammar.
- At each reduction step, a specific substring matching the body of a production is replaced by the non terminal at the head of that production
- A reduction is the reverse of a step in a derivation, therefore bottom up parser is rightmost derivation in reverse.
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parser proceeds.

Example for reduction

Example:

- Consider the grammar:

  $S \rightarrow aABe$

  $A \rightarrow Abc \mid b$

  $B \rightarrow d$

- The sentence/Input string(w) to be recognized is abbcde.

REDUCTION (LEFTMOST)		RIGHTMOST DERIVATION
a**b**bcde	(A → b)	**S** → a**AB**e
a**Abc**de	(A → Abc)	→ a**A**de
a**A**de	(B → d)	→ a**Abc**de
aABe	(S → aABe)	→ abbcde
S		

The reductions trace out the right-most derivation in reverse.

Handle

- A "handle" is a substring that matches the right side of a production, and whose reduction to a non terminal in the left side represents previous step in a rightmost derivation in reverse.

handle of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of 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.

Example

Consider the following grammar

(1) $E \rightarrow E + E$
(2) $E \rightarrow E * E$
(3) $E \rightarrow (E)$
(4) $E \rightarrow \mathbf{id}$

Contd…

And the input string $id_1+id_2*id_3$

The rightmost derivation is :

$E \rightarrow \underline{\mathbf{E+E}}$

$\rightarrow E+\underline{\mathbf{E*E}}$

$\rightarrow E+E*\underline{\mathbf{id_3}}$

$\rightarrow E+\underline{\mathbf{id_2}}*id_3$

$\rightarrow \underline{\mathbf{id_1}}+id_2*id_3$

In the above derivation the underlined substrings are called **handles.**

Contd…

Right Sentential Form	Handle	Reducing Production
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E + E$	$E \rightarrow E + E$
$E * E$	$E * E$	$E \rightarrow E * E$
E		

$E \rightarrow E+E \mid E*E \mid id$

Handle Pruning

- A Handle is a substring that matches the body of a production.
- Handle reduction is a step in the reverse of rightmost derivation.
- A rightmost derivation in reverse can be obtained by handle pruning.
- The implementation of handle pruning involves the following data-structures:-
  - a stack - to hold the grammar symbols;
  - an input buffer that contains the remaining input and a table to decide handles.

Contd…

A rightmost derivation in reverse can be obtained by "handle pruning." That is, we start with a string of terminals w that we wish to parse. If w is a sentence of the grammar at hand, then $w = \gamma_n$, where γ_n is the nth right-sentential form of some as yet unknown rightmost derivation

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

To reconstruct this derivation in reverse order, we locate the handle β_n in γ_n and replace β_n by the left side of some production $A_n \rightarrow \beta_n$ to obtain the $(n-1)$st right-sentential form γ_{n-1}.

If by continuing this process we produce a right-sentential form consisting only of the start symbol S, then we halt and announce successful completion of parsing.

Shift-Reduce Parsing

- Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- Initially, the stack is empty, and the string *w* is on the input, as follows:
- **Stack** **Input Buffer**

  $** **w$
- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until the handle appears on top of the stack.
- It then reduces by left side of the appropriate production.
- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

  Stack **Input Buffer**

  $S **$**
- Upon entering this configuration, the parser halts and announces successful completion of parsing.

Example

- Consider the following grammar and the Input string **w= i d 1 * i d 2 $.**

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

Shift reduce parsing for w= i d 1 * i d 2 $.

STACK	INPUT	ACTION
$	id 1 * id 2 $	shift
$id	id 2 * id 2 $	reduce by *F* -> id
$*F*	* id 2 $	reduce by *T* -> *F*
$*T*	* id 2 $	shift
$ *T* *	id 2$	shift
$ *T* * id 2	$	reduce by *F* -> id
$ *T* * *F*	$	reduce by *T* -> *T* * *F*
$*T*	$	reduce by *E* -> *T*
$*E*	$	accept

Contd…

- The primary operations are shift and reduce
- There are actually four possible actions a shift-reduce parser can make: (1) shift, (2) reduce, (3) accept, and (4) error.

1. ***Shift.*** Shift the next input symbol onto the top of the stack.
2. ***Reduce.*** If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of production rule is popped out of stack and LHS of production rule is pushed onto the stack.
3. **Accept.** Announce successful completion of parsing.
4. ***Error.*** Discover a syntax error and call an error recovery routine.

Conflicts During Shift-Reduce Parsing

- A shift-reduce parser for a grammar can reach a configuration in which the parser

1. cannot decide whether to shift or to reduce (a shift/reduce conflict)
2. cannot decide which of several reductions to make (a reduce/reduce conflict). Reduce-reduce conflicts are rare and usually indicate a problem in the grammar definition.

Example for shift/reduce conflict

stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other**

STACK	INPUT
· · · **if** *expr* **then** *stmt*	**else** · · · $

Here we do not know whether to perform reduce or shift action.

Example for reduce/reduce conflict

Example for reduce/reduce conflict

(1)	*stmt*	→	**id** (*parameter_list*)
(2)	*stmt*	→	*expr* := *expr*
(3)	*parameter_list*	→	*parameter_list* , *parameter*
(4)	*parameter_list*	→	*parameter*
(5)	*parameter*	→	**id**
(6)	*expr*	→	**id** (*expr_list*)
(7)	*expr*	→	**id**
(8)	*expr_list*	→	*expr_list* , *expr*
(9)	*expr_list*	→	*expr*

STACK	INPUT
· · · **id** (**id**	, **id**) · · ·

Here we do not know whether to reduce by parameter or expr.

Bottom-Up Parsing

- A **bottom-up parser** creates the parse tree of the given input starting from leaves towards the root.
- A bottom-up parser tries to find the right-most derivation of the given input in the reverse order.

$S \Rightarrow ... \Rightarrow \omega$ (the right-most derivation of ω)

$\leftarrow$ (the bottom-up parser finds the right-most derivation in the reverse order)

Operator-Precedence Parser

- **Operator grammar**
  - small, but an important class of grammars
  - we may have an efficient operator precedence parser (a shift-reduce parser) for an operator grammar.
- In an *operator grammar*, no production rule can have:
  - ε at the right side
  - two adjacent non-terminals at the right side.

- Ex:

E→AB	E→EOE	E→E+E |
A→a	E→id	E*E |
B→b	O→+|*|/	E/E | id
not operator grammar	not operator grammar	operator grammar

Precedence Relations

- In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

  $a \lessdot b$ b has higher precedence than a

  $a \doteq b$ b has same precedence as a

  $a \gtrdot b$ b has lower precedence than a

- The determination of correct precedence relations between terminals are based on the traditional notions of associativity and precedence of operators. (Unary minus causes a problem).

Using Operator-Precedence Relations

- The intention of the precedence relations is to find the handle of a right-sentential form,
  $\lessdot$ with marking the left end,
  $\doteq$ appearing in the interior of the handle, and
  $\gtrdot$ marking the right hand.

- In our input string $\$a_1a_2...a_n\$$, we insert the precedence relation between the pairs of terminals (the precedence relation holds between the terminals in that pair).

Using Operator -Precedence Relations

E → E+E | E-E | E*E | E/E | E^E | (E) | -E | id

The partial operator-precedence table for this grammar

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

- Then the input string id+id*id with the precedence relations inserted will be:

$ <· id ·> + <· id ·> * <· id ·> $

To Find The Handles

1. Scan the string from left end until the first $\cdot>$ is encountered.
2. Then scan backwards (to the left) over any $=\cdot$ until a $<\cdot$ is encountered.
3. The handle contains everything to left of the first $\cdot>$ and to the right of the $<\cdot$ is encountered.

$ <· id ·> + <· id ·> * <· id ·> $	E → id	$ id + id * id $
$ <· + <· id ·> * <· id ·> $	E → id	$ E + id * id $
$ <· + <· * <· id ·> $	E → id	$ E + E * id $
$ <· + <· * ·> $	E → E*E	$ E + E * E $
$ <· + ·> $	E → E+E	$ E + E $
$ $		$ E $

Operator-Precedence Parsing Algorithm

- The input string is w$, the initial stack is $ and a table holds precedence relations between certain terminals

Algorithm:

```
set p to point to the first symbol of w$ ;
repeat forever
   if ( $ is on top of the stack and p points to $ ) then return
   else {
      let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by p;
      if ( a <· b or a =· b ) then {        /* SHIFT */
         push b onto the stack;
         advance p to the next input symbol;
      }
      else if ( a ·> b ) then              /* REDUCE */
         repeat pop stack
         until ( the top of stack terminal is related by <· to the terminal most recently popped );
      else error();
   }
```

Operator-Precedence Parsing Algorithm -- Example

stack	***input***	***action***		
$	id+id*id$	$ $\lessdot$ id	shift	
$id	+id*id$	id $\gtrdot$ +	reduce	E → id
$	+id*id$	shift		
$+	id*id$	shift		
$+id	*id$	id $\gtrdot$ *	reduce	E → id
$+	*id$	shift		
$+*	id$	shift		
$+*id	$	id $\gtrdot$ $	reduce	E → id
$+*	$	* $\gtrdot$ $	reduce	E → E*E
$+	$	+ $\gtrdot$ $	reduce	E → E+E
$	$	accept		

How to Create Operator-Precedence Relations

- We use associativity and precedence relations among operators.

1. If operator O_1 has higher precedence than operator O_2,
   $O_1 \cdot> O_2$ and $O_2 <\cdot O_1$

2. If operator O_1 and operator O_2 have equal precedence,
   they are left-associative $\Rightarrow$ $O_1 \cdot> O_2$ and $O_2 \cdot> O_1$
   they are right-associative $\Rightarrow$ $O_1 <\cdot O_2$ and $O_2 <\cdot O_1$

3. For all operators O,
   $O <\cdot id$, $id \cdot> O$, $O <\cdot ($, $(<\cdot O$, $O \cdot>)$, $) \cdot> O$, $O \cdot> \$$, and $\$ <\cdot O$

4. Also, let

$(=\cdot)$	$\$ <\cdot ($	$id \cdot>)$	$) \cdot> \$$
$(<\cdot ($	$\$ <\cdot id$	$id \cdot> \$$	$) \cdot>)$
$(<\cdot id$			

Operator-Precedence Relations

	+	-	*	/	^	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
-	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
^	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=·	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Handling Unary Minus

- Operator-Precedence parsing cannot handle the unary minus when we also the binary minus in our grammar.
- The best approach to solve this problem, let the lexical analyzer handle this problem.
  - The lexical analyzer will return two different operators for the unary minus and the binary minus.
  - The lexical analyzer will need a lookhead to distinguish the binary minus from the unary minus.
- Then, we make

$O \lessdot$ unary-minus	for any operator
unary-minus $\gtrdot O$	if unary-minus has higher precedence than O
unary-minus $\lessdot O$	if unary-minus has lower (or equal) precedence than O

Precedence Functions

- Compilers using operator precedence parsers do not need to store the table of precedence relations.
- The table can be encoded by two precedence functions f and g that map terminal symbols to integers.
- For symbols a and b.

  $f(a) < g(b)$ whenever $a <\cdot b$

  $f(a) = g(b)$ whenever $a =\cdot b$

  $f(a) > g(b)$ whenever $a \cdot> b$

Disadvantages of Operator Precedence Parsing

- **Disadvantages**:
  - It cannot handle the unary minus (the lexical analyzer should handle the unary minus).
  - Small class of grammars.
  - Difficult to decide which language is recognized by the grammar.

- **Advantages**:
  - simple
  - powerful enough for expressions in programming languages

Error Recovery in Operator-Precedence Parsing

Error Cases:

1. No relation holds between the terminal on the top of stack and the next input symbol.
2. A handle is found (reduction step), but there is no production with this handle as a right side

Error Recovery:

3. Each empty entry is filled with a pointer to an error routine.
4. Decides the popped handle “looks like” which right hand side. And tries to recover from that situation.

Computation of LEADING & TRAILING

- Two Methods to determine a precedence relation between a pair of terminals
  - Based on associativity and precedence relations of operators
  - Using Operator Precedence Grammar
- Implementation of Operator-Precedence Parser:
  - An operator-precedence parser is a simple shift-reduce parser that is capable of parsing a subset of LR(1) grammars.
  - The operator-precedence parser can parse all LR(1) grammars where two consecutive non-terminals and epsilon never appear in the right-hand side of any rule.

Computation of LEADING & TRAILING

- Steps involved in Parsing:
  1. Ensure the grammar satisfies the pre-requisite.
  2. Computation of the function LEADING()
  3. Computation of the function TRAILING()
  4. Using the computed leading and trailing ,construct the operator Precedence Table
  5. Parse the given input string based on the algorithm
  6. Compute Precedence Function and graph.

Computation of LEADING

Leading is defined for every non-terminal. Terminals that can be the first terminal in a string derived from that non-terminal.

Compute LEADING (A)

- LEADING (A) = {a| A → γaδ, where γ is ε or a single non-terminal.}
- Rule 1: a is in LEADING (A) if there is a production of the form A → γaδ, Where γ is ε or a single non-terminal
- Rule 2: a is in LEADING (B) and if there is a production of the form A → Bα, then a is in LEADING (A)

Computation of TRAILING

Trailing is defined for every non-terminal. • Terminals that can be the last terminal in a string derived from that non-terminal.

Compute TRAILING (A)

• TRAILING (A) = {a| A → γaδ, where δ is ε or a single non-terminal.}

• Rule 1: a is in TRAILING (A) if there is a production of the form A → γaδ, Where δ is ε or a single non-terminal

• Rule 2: a is in TRAILING (B) and if there is a production of the form A → αB, then a is in TRAILING (A)

Computation of LEADING & TRAILING--Example

Example 1: Consider the unambiguous grammar,

E→E + T

E→T

T→T * F

T→F

F→(E)

F→**id**

Step 1: Compute LEADING and TRAILING:

LEADING(E)= { +,LEADING(T)} ={+ , * , (, id}

LEADING(T)= { *,LEADING(F)} ={* , (, id}

LEADING(F)= { (, id}

TRAILING(E)= { +, TRAILING(T)} ={+ , * ,) , id}

TRAILING(T)= { *, TRAILING(F)} ={* ,) , id}

TRAILING(F)= {) , id}

Algorithm for constructing Precedence Relation Table

Step 2: After computing LEADING and TRAILING, the table is constructed between all the terminals in the grammar including the '$' symbol.

PARSINGTABLE(Grammar G, LEADING(), TRAILING())

{

For each production A $\rightarrow X_1X_2X_3 \ldots X_n$

for i = 1 to n-1

1. if X_i and X_{i+1} are terminals
   set $X_i = \cdot X_{i+1}$
2. if $i \leq n-2$ and X_i and X_{i+2} are terminals and X_{i+1} is a non-terminal
   set $X_i = \cdot X_{i+2}$
3. if X_i is a terminal and X_{i+1} is a non-terminal then for all 'a' in Leading(X_{i+1}) set $X_i <\cdot$ **a**
4. if X_i is a non-terminal and X_{i+1} is a terminal then for all 'a' in Trailing(X_i) set a $\cdot> X_{i+1}$

}

Precedence Relation Table

	+	*	id	(	)	$
+	>	<	<	<	>	>
*	>	>	<	<	>	>
id	>	>	e	e	>	>
(	<	<	<	<	=	e
)	>	>	e	e	>	>
$	<	<	<	<	e	Accept

Parsing Algorithm

Algorithm:

set p to point to the first symbol of w$;
repeat forever
if $ is on top of the stack **and** p points to $ **then return**
else begin
let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by p;
if (a $\lessdot$ b or a $\doteq$ b) **then**
push b onto the stack;
advance p to the next input symbol;
end
else if a $\gtrdot$ b **then**
repeat pop stack
until the top of stack terminal is related by $\lessdot$ to the terminal most recently popped
else error();
end

Parse the given input string (id+id)*id$

- Step 3

STACK	REL.	INPUT	ACTION
$	$ < (	(id+id)*id$	Shift (
$(	(< id	id+id)*id$	Shift id
$(id	id > +	+id)*id$	Pop id
$(	(< +	+id)*id$	Shift +
$(+	+ < id	id)*id$	Shift id
$(+id	id >)	)*id$	Pop id
$(+	+ >)	)*id$	Pop +
$(	(=)	)*id$	Shift)
$() $(	) > *	*id $	Pop) Pop (
$	$ < *	*id $	Shift *
$*	* < id	id$	Shift id
$*id	id > $	$	Pop id
$*	* > $	$	Pop *
$		$	Accept

Precedence Functions

- Compilers using operator-precedence parsers need not store the table of precedence relations. In most cases, the table can be encoded by two precedence functions f and g that map terminal symbols to integers. We attempt to select f and g so that, for symbols a and b.

1. f (a) < g(b) whenever a<·b.
2. f (a) = g(b) whenever a = b. and
3. f(a) > g(b) whenever a ·> b.

Algorithm for Constructing Precedence Functions

1. Create functions f_a for each grammar terminal a and for the end of string symbol.
2. Partition the symbols in groups so that f_a and g_b are in the same group if $a = b$ (there can be symbols in the same group even if they are not connected by this relation).
3. Create a directed graph whose nodes are in the groups, next for each symbols a and b do: place an edge from the group of g_b to the group of f_a if $a <\cdot b$, otherwise if $a \cdot> b$ place an edge from the group of f_a to that of g_b.
4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of f_a and g_b respectively.

There are no cycles,so precedence function exist. As f$ and g$ have no out edges,f($)=g($)=0.The longest path from g+ has length 1,so g(+)=1.There is a path from g_{id} to f* to g* to f+ to g+ to f$,so g(id)=5.The resulting precedence functions are:

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

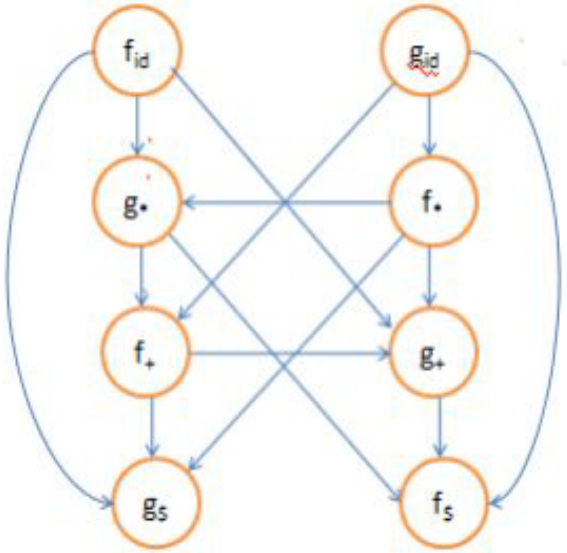


Fig Precedence Graph

Example 2:

Consider the following grammar, and construct the operator precedence parsing table and check whether the input string (i) *id=id (ii)id*id=id are successfully parsed or not?

S→L=R

S→R

L→*R

L→id

R→L

1.Computation of LEADING:

LEADING(S) = {=, * , id}

LEADING(L) = {* , id}

LEADING(R) = {* , id}

2.Computation of TRAILING:

TRAILING(S) = {= , * , id}

TRAILING(L)= {* , id}

TRAILING(R)= {* , id}

3.Precedence Table:

	=	*	id	$
=	e	<·	<·	·>
*	·>	<·	<·	·>
id	·>	e	e	·>
$	<·	<·	<·	accept

* All undefined entries are error (e).

4.Parsing the given input string:

1. ***id = id**

STACK	INPUT STRING	ACTION
$	*id=id$	$<·* Push
$*	id=id$	*<·id Push
$*id	=id$	id·>= Pop
$*	=id$	*·>= Pop
$	=id$	$<·= Push
$=	id$	=<·id Push
$=id	$	id·>$ Pop
$=	$	=·>$ Pop
$	$	Accept

2. **id*id=id**

STACK	INPUT STRING	ACTION
$	id*id=id$	$<·idPush
$id	*id=id$	Error

Introduction to LR Parsing

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(k) parsing. The 'L' is for left-to-right scanning of the input, the 'R' for constructing a rightmost derivation in reverse, and the 'k' for the number of input symbols. When 'k' is omitted, it is assumed to be 1.

Advantages of LR parsing:

- ✓ It recognizes virtually all programming language constructs for which CFG can be written.
- ✓ It is an efficient non-backtracking shift-reduce parsing method.
- ✓ A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
- ✓ It detects a syntactic error as soon as possible.

Drawbacks of LR method:

It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

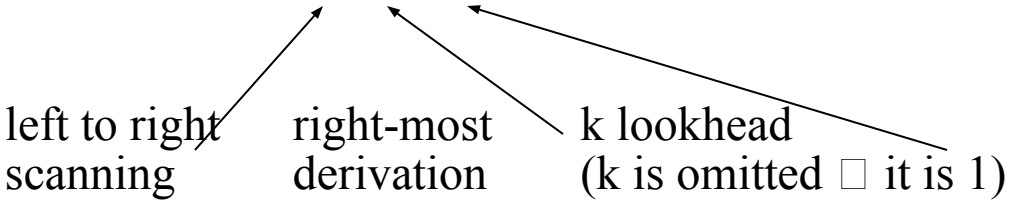
Why LR Parsers?

- LR parsing is attractive for a variety of reasons:
- LR parsers can be constructed to recognize all programming language constructs.
- The LR-parsing method is the most general non backtracking shift-reduce parsing method it can be implemented efficiently.
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods. LR grammars can describe more languages than LL grammars.
- The principal drawback of the LR method is that it is too much work to construct an LR parser for a typical programming-language grammar.

LR Parsers

- The most powerful shift-reduce parsing (yet efficient) is:

LR(k) parsing.



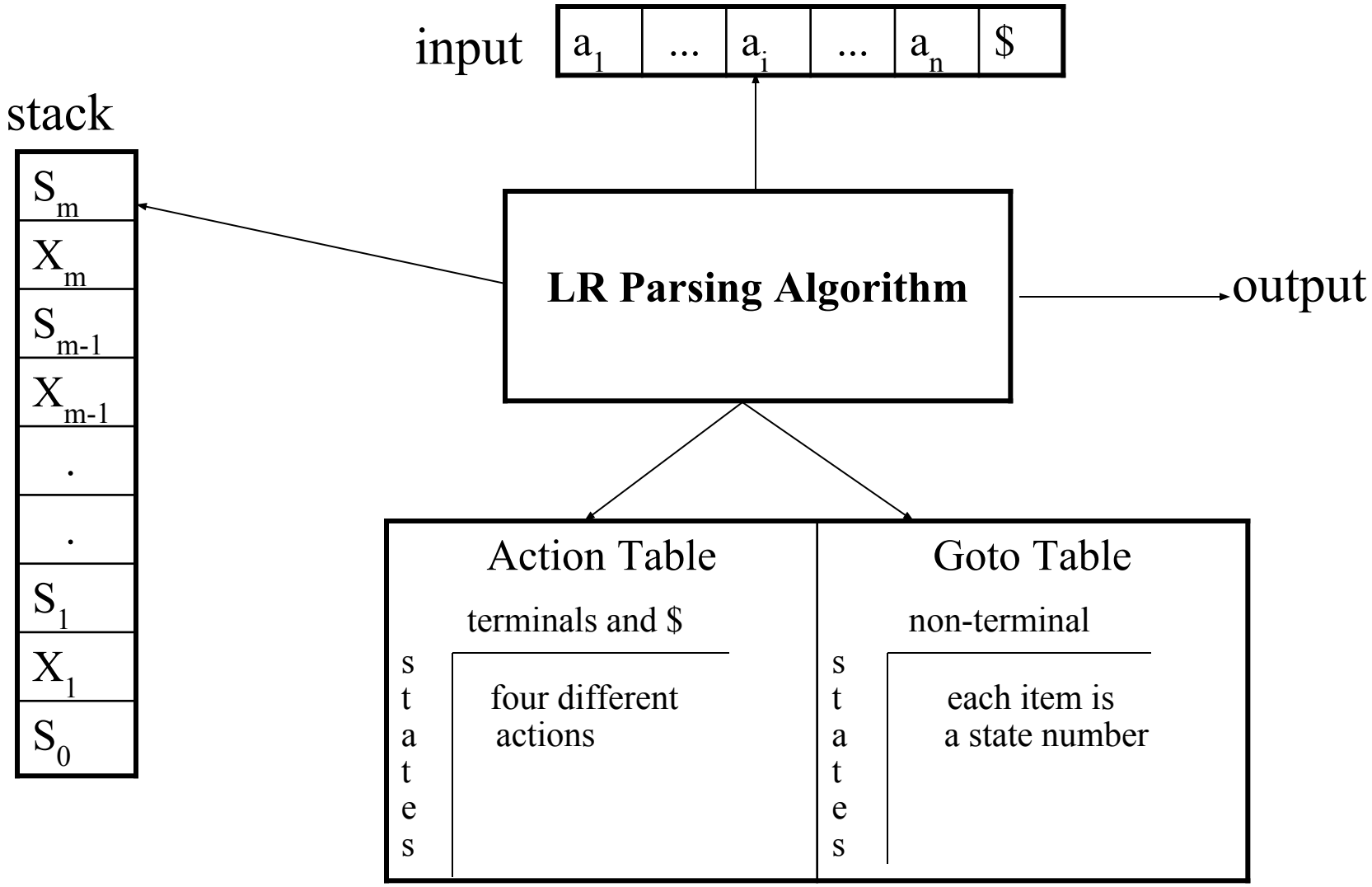
- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

    LL(1)-Grammars ⊂ LR(1)-Grammars
  - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.

LR Parsers

- **LR-Parsers**
  - covers wide range of grammars.
  - SLR – simple LR parser
  - LR – most general LR parser
  - LALR – intermediate LR parser (look-head LR parser)
  - SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

LR Parsing Algorithm



A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:

$$(S_o X_1 S_1 \ldots X_m S_m, \; a_i a_{i+1} \ldots a_n \$)$$

- S_m and a_i decides the parser action by consulting the parsing action table. (*Initial Stack* contains just S_o)

- A configuration of a LR parsing represents the right sentential form:

$$X_1 \ldots X_m a_i a_{i+1} \ldots a_n \$$$

Actions of A LR-Parser

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack

   $(S_o X_1 S_1 \ldots X_m S_m, a_i a_{i+1} \ldots a_n \$) \Rightarrow (S_o X_1 S_1 \ldots X_m S_m a_i s, a_{i+1} \ldots a_n \$)$

2. **reduce A→β** (or **rn** where n is a production number)
   - pop $2|\beta|$ (=r) items from the stack;
   - then push **A** and **s** where **s=goto[s_{m-r},A]**

   $(S_o X_1 S_1 \ldots X_m S_m, a_i a_{i+1} \ldots a_n \$) \Rightarrow (S_o X_1 S_1 \ldots X_{m-r} S_{m-r} A s, a_i \ldots a_n \$)$

   - Output is the reducing production reduce A→β

3. **Accept** – Parsing successfully completed

4. **Error** -- Parser detected an error (an empty entry in the action table)

Reduce Action

- pop $2|\beta|$ (=r) items from the stack; let us assume that $\beta = Y_1 Y_2 \ldots Y_r$
- then push **A** and **s** where **s=goto[s_{m-r},A]**

$(S_o X_1 S_1 \ldots X_{m-r} S_{m-r} Y_1 S_{m-r} \ldots Y_r S_m, a_i a_{i+1} \ldots a_n \$)$

$\Rightarrow (S_o X_1 S_1 \ldots X_{m-r} S_{m-r} A s, a_i \ldots a_n \$)$

- In fact, $Y_1 Y_2 \ldots Y_r$ is a handle.

$X_1 \ldots X_{m-r} A a_i \ldots a_n \$ \Rightarrow X_1 \ldots X_m Y_1 \ldots Y_r a_i a_{i+1} \ldots a_n \$$

(SLR) Parsing Tables for Expression Grammar

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

Action Table | Goto Table

state	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				

Actions of A (S)LR-Parser -- Example

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

Constructing SLR Parsing Tables – LR(0) Item

- An **LR(0) item** of a grammar G is a production of G a dot at the some position of the right side.
- Ex: $A \rightarrow aBb$ Possible LR(0) Items: $A \rightarrow \bullet aBb$

  (four different possibility) $A \rightarrow a \bullet Bb$

  $A \rightarrow aB \bullet b$

  $A \rightarrow aBb \bullet$
- Sets of LR(0) items will be the states of action and goto table of the SLR parser.
- A collection of sets of LR(0) items (**the canonical LR(0) collection**) is the basis for constructing SLR parsers.
- *Augmented Grammar*:

  G' is G with a new production rule S'→S where S' is the new starting symbol.

The Closure Operation

- If ***I*** is a set of LR(0) items for a grammar G, then ***closure(I)*** is the set of LR(0) items constructed from I by the two rules:

1. Initially, every LR(0) item in I is added to closure(I).
2. If $A \rightarrow \alpha \bullet B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow \bullet \gamma$ will be in the closure(I).
   We will apply this rule until no more new LR(0) items can be added to closure(I).

The Closure Operation -- Example

E' → E

E → E+T

E → T

T → T*F

T → F

F → (E)

F → id

closure({E' → •E}) =

{ E' → •E kernel items

E → •E+T

E → •T

T → •T*F

T → •F

F → •(E)

F → •id }

Goto Operation

- If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha \bullet X\beta$ in I

    then every item in **closure({$A \rightarrow \alpha X \bullet \beta$})** will be in goto(I,X).

Example:

I ={ $E' \rightarrow \bullet E$, $E \rightarrow \bullet E+T$, $E \rightarrow \bullet T$,
$T \rightarrow \bullet T*F$, $T \rightarrow \bullet F$,
$F \rightarrow \bullet (E)$, $F \rightarrow \bullet id$ }

goto(I,E) = { $E' \rightarrow E \bullet$, $E \rightarrow E \bullet +T$ }

goto(I,T) = { $E \rightarrow T \bullet$, $T \rightarrow T \bullet *F$ }

goto(I,F) = {$T \rightarrow F \bullet$ }

goto(I,() = { $F \rightarrow (\bullet E)$, $E \rightarrow \bullet E+T$, $E \rightarrow \bullet T$, $T \rightarrow \bullet T*F$, $T \rightarrow \bullet F$,
$F \rightarrow \bullet (E)$, $F \rightarrow \bullet id$ }

goto(I,id) = { $F \rightarrow id \bullet$ }

Construction of The Canonical LR(0) Collection

- To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.

- ***Algorithm***:

  C is { closure({S'→ • S}) }

  repeat the followings until no more set of LR(0) items can be added to ***C***.

  for each I in ***C*** and each grammar symbol X

  if goto(I,X) is not empty and not in ***C***

  add goto(I,X) to ***C***

- goto function is a DFA on the sets in C.

The Canonical LR(0) Collection -- Example

I_0: E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_1: E' → E.
E → E.+T

I_2: E → T.
T → T.*F

I_3: T → F.

I_4: F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_5: F → id.

I_6: E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

I_7: T → T*.F
F → .(E)
F → .id

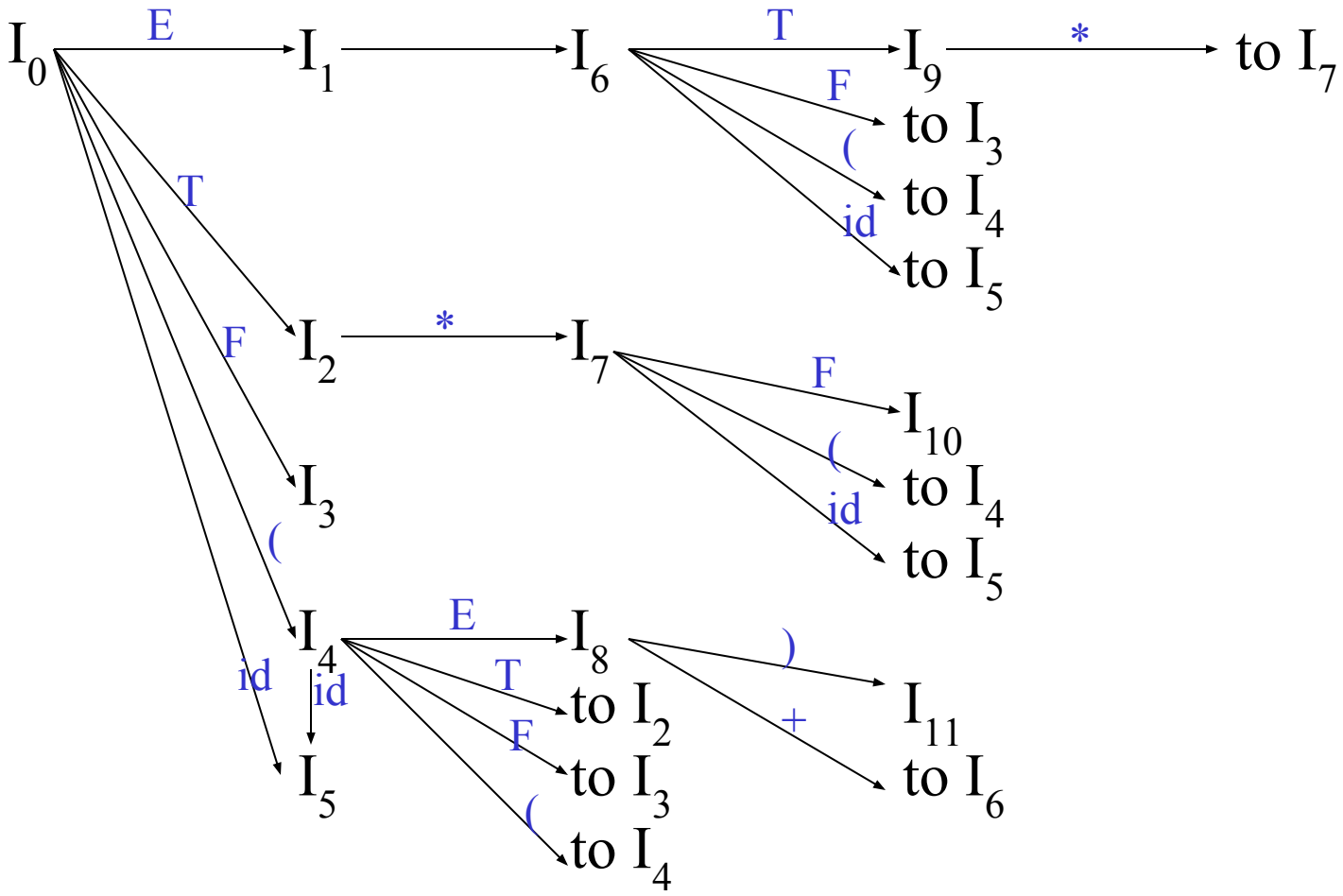
I_8: F → (E.)
E → E.+T

I_9: E → E+T.
T → T.*F

I_{10}: T → T*F.

I_{11}: F → (E).

Transition Diagram (DFA) of Goto Function



Constructing SLR Parsing Table

(of an augumented grammar G')

1. Construct the canonical collection of sets of LR(0) items for G'. C← $\{I_0,...,I_n\}$
2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha.a\beta$ in I_i and goto(I_i,a)=I_j then action[i,a] is ***shift j.***
   - If $A \rightarrow \alpha.$ is in I_i, then action[i,a] is ***reduce*** $A \rightarrow \alpha$ for all a in FOLLOW(A) where $A \neq S'$.
   - If $S' \rightarrow S.$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).
3. Create the parsing goto table
   - for all non-terminals A, if goto(I_i,A)=I_j then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains $S' \rightarrow .S$

Parsing Tables of Expression Grammar

	Action Table						Goto Table		
state	**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			

Another Example

- GivenGrammar - G

  S → AA

  A → aA | b
- Step 1:
- Add Augment Production and insert '•' symbol at the first position for every production in G

  S` → •S

  S → •AA

  A → •aA

  A → •b
- Step 2:
- Add Augment production to the I0 State and Compute the Closure

  I0 = Closure (S` → •S)

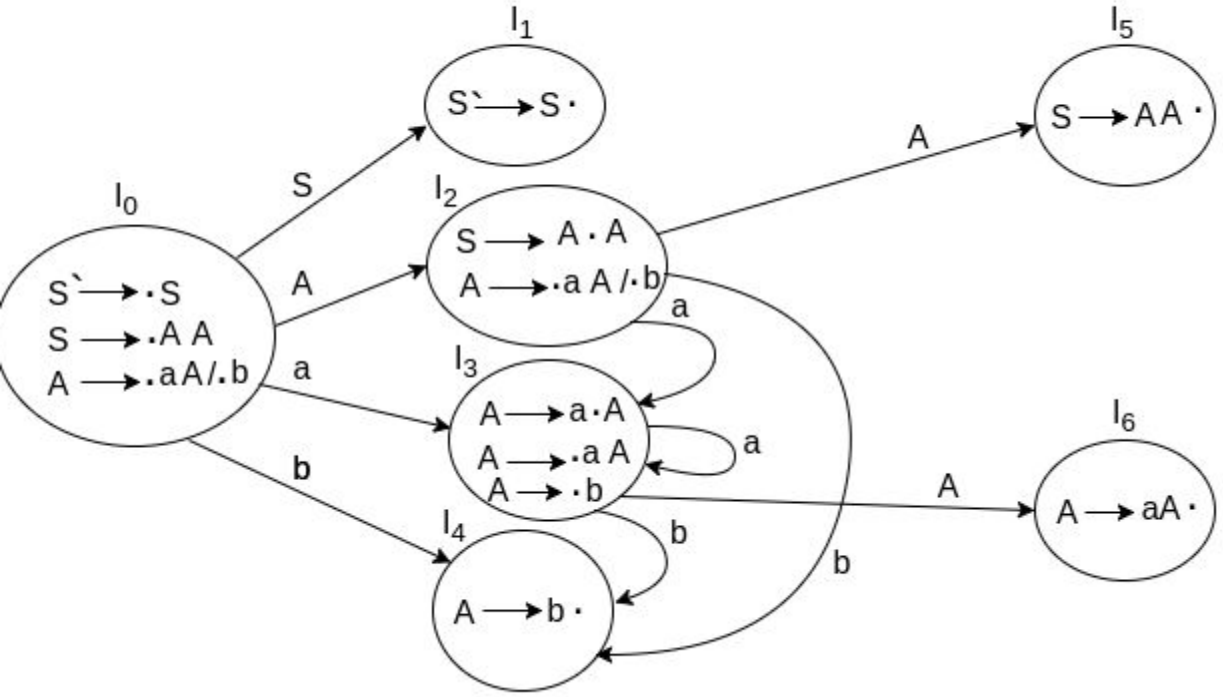
Contd…

- Add all productions starting with "A" in modified I0 State because "•" is followed by the non-terminal. So, the I0 State becomes.
- **I0=** S` → •S
  S → •AA
  A → •aA
  A → •b
- **I1=** Go to (I0, S) = closure (S` → S•) = S` → S•
- Here, the Production is reduced so close the State.
- **I1=** S` → S•
- **I2=** Go to (I0, A) = closure (S → A•A)
- Add all productions starting with A in to I2 State because "•" is followed by the non-terminal. So, the I2 State becomes
- **I2 =**S→A•A
  A → •aA
  A → •b

Contd…

- **I3=** Go to (I0,a) = Closure (A → a•A)
- Add productions starting with A in I3.
- A → a•A
  A → •aA
  A → •b
- **I4=** Go to (I0, b) = closure (A → b•) = A → b•
- **I5=** Go to (I2, A) = Closure (S → A • A) = S → AA•
- Go to (I2,a) = Closure (A → a•A) = (same as I3)
- Go to (I2, b) = Closure (A → b•) = (same as I4)
- Go to (I3, a) = Closure (A → a•A) = (same as I3)
  Go to (I3, b) = Closure (A → b•) = (same as I4)
- **I6=** Go to (I3, A) = Closure (A → aA•) = A → aA•

The DFA contains the 7 states I0 to I6.



States	Action			Go to	
	a	b	$	A	S
I_0	S3	S4		2	1
I_1			accept		
I_2	S3	S4		5	
I_3	S3	S4		6	
I_4	r3	r3	r3		
I_5	r1	r1	r1		
I_6	r2	r2	r2		

LR(0) Table

- If a state is going to some other state on a terminal then it correspond to a Shift move.
- If a state is going to some other state on a variable then it correspond to Go to move.
- If a state contain the final item in the particular row then write the Reduce move completely.

States	Action			Go to	
	a	b	S	A	S
I_0	S3	S4		2	1
I_1			accept		
I_2	S3	S4		5	
I_3	S3	S4		6	
I_4	r3	r3	r3		
I_5	r1	r1	r1		
I_6	r2	r2	r2		

LR(0) Table Entry Explanation

- Productions are numbered as follows:

  S → AA ... (1)

  A → aA ... (2)

  A → b ... (3)
- I0 on S is going to I1 so write it as 1.
- I0 on A is going to I2 so write it as 2.
- I2 on A is going to I5 so write it as 5.
- I3 on A is going to I6 so write it as 6.
- I0, I2and I3on a are going to I3 so write it as S3 which means that shift 3.
- I0, I2 and I3 on b are going to I4 so write it as S4 which means that shift 4.
- I4, I5 and I6 all states contains the final item because they contain • in the right most end. So rate the production as production number.

Contd…

- I1 contains the final item which drives(S` → S•), so action {I1, $} = Accept.
- I4 contains the final item which drives A → b• and that production corresponds to the production number 3 so write it as r3 in the entire row.
- I5 contains the final item which drives S → AA• and that production corresponds to the production number 1 so write it as r1 in the entire row.
- I6 contains the final item which drives A → aA• and that production corresponds to the production number 2 so write it as r2 in the entire row.

SLR(1) Grammar

- An LR parser using SLR(1) parsing tables for a grammar G is called as the SLR(1) parser for G.
- If a grammar G has an SLR(1) parsing table, it is called SLR(1) grammar (or SLR grammar in short).
- Every SLR grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.

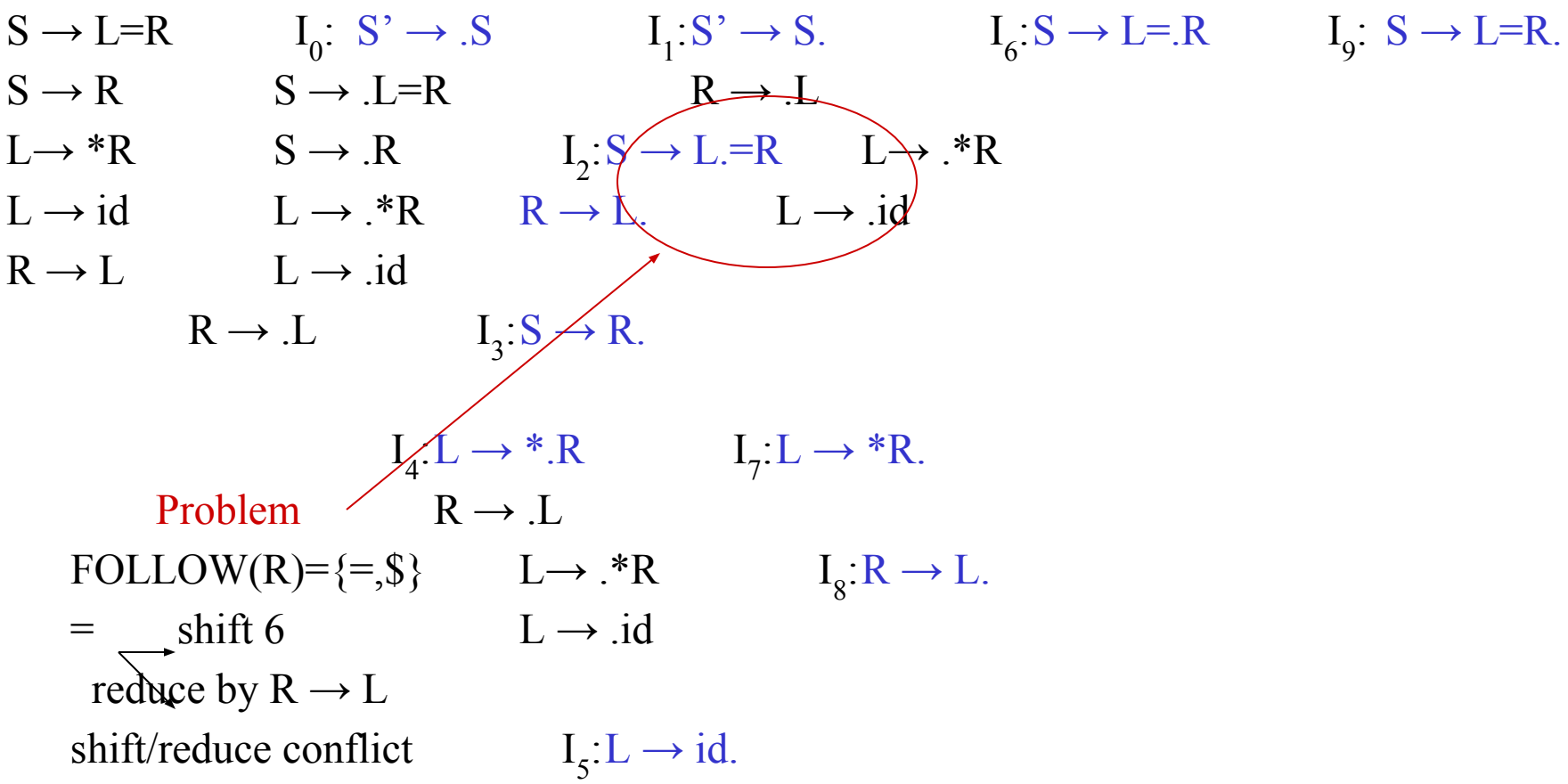
shift/reduce and reduce/reduce conflicts

- If a state does not know whether it will make a shift operation or reduction for a terminal, we say that there is a **shift/reduce conflict**.

- If a state does not know whether it will make a reduction operation using the production rule `i` or `j` for a terminal, we say that there is a **reduce/reduce conflict**.

- If the SLR parsing table of a grammar G has a conflict, we say that that grammar is not SLR grammar.

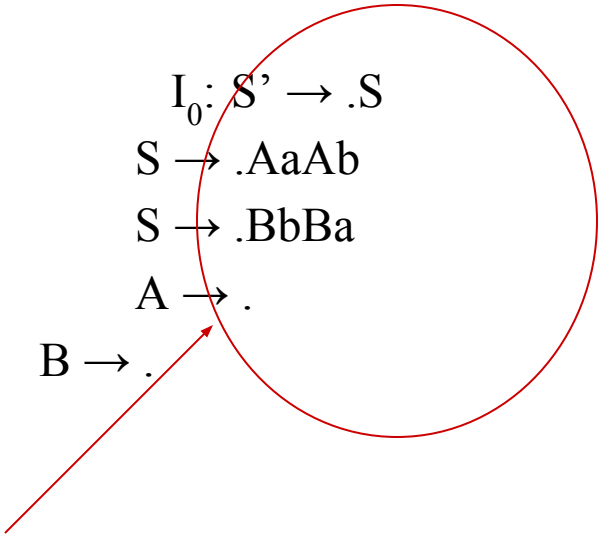
Conflict Example



Conflict Example2

S → AaAb
S → BbBa
A → ε
B → ε

I_0: S' → .S
S → .AaAb
S → .BbBa
A → .
B → .



Problem

FOLLOW(A)={a,b}
FOLLOW(B)={a,b}

a	reduce by A → ε	b	reduce by A → ε
	reduce by B → ε		reduce by B → ε
	reduce/reduce conflict		reduce/reduce conflict

Constructing Canonical LR(1) Parsing Tables

- In SLR method, the state i makes a reduction by $A \rightarrow \alpha$ when the current token is a:
  - if the $A \rightarrow \alpha.$ in the I_i and a is FOLLOW(A)
- In some situations, βA cannot be followed by the terminal a in a right-sentential form when $\beta\alpha$ and the state i are on the top stack. This means that making reduction in this case is not correct.

$S \rightarrow AaAb$ $\quad S \Rightarrow AaAb \Rightarrow Aab \Rightarrow ab$ $\quad S \Rightarrow BbBa \Rightarrow Bba \Rightarrow ba$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$ $\quad Aab \Rightarrow \varepsilon\, ab$ $\quad Bba \Rightarrow \varepsilon\, ba$

$B \rightarrow \varepsilon$ $\quad AaAb \Rightarrow Aa\, \varepsilon\, b$ $\quad BbBa \Rightarrow Bb\, \varepsilon\, a$

LR(1) Item

- To avoid some of invalid reductions, the states need to carry more information.
- Extra information is put into a state by including a terminal symbol as a second component in an item.
- A LR(1) item is:

$A \rightarrow \alpha \bullet \beta, a$ where **a** is the look-head of the LR(1) item

(**a** is a terminal or end-marker.)

LR(1) Item (cont.)

- When β (in the LR(1) item $A \rightarrow \alpha \bullet \beta, a$) is not empty, the look-head does not have any affect.
- When β is empty ($A \rightarrow \alpha \bullet, a$), we do the reduction by $A \rightarrow \alpha$ only if the next input symbol is **a** (not for any terminal in FOLLOW(A)).
- A state will contain $A \rightarrow \alpha \bullet, a_1$ where $\{a_1, \ldots, a_n\} \subseteq$ FOLLOW(A)

  ...

  $A \rightarrow \alpha \bullet, a_n$

Canonical Collection of Sets of LR(1) Items

- The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that *closure* and *goto* operations work a little bit different.

closure(I) is: (where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I)
- if $A \rightarrow \alpha . B\beta, a$ in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma, b$ will be in the closure(I) for each terminal b in FIRST(βa) .

goto operation

- If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha.X\beta,a$ in I
    then every item in **closure({$A \rightarrow \alpha X.\beta,a$})** will be in goto(I,X).

Construction of The Canonical LR(1) Collection

- ***Algorithm***:

  C is { closure({S'→.S,$}) }

  repeat the followings until no more set of LR(1) items can be added to ***C***.

  for each I in ***C*** and each grammar symbol X

  if goto(I,X) is not empty and not in ***C***

  add goto(I,X) to ***C***

- goto function is a DFA on the sets in C.

A Short Notation for The Sets of LR(1) Items

- A set of LR(1) items containing the following items

$$A \rightarrow \alpha \bullet \beta, a_1$$

...

$$A \rightarrow \alpha \bullet \beta, a_n$$

can be written as

$$A \rightarrow \alpha \bullet \beta, a_1/a_2/.../a_n$$

Canonical LR(1) Collection -- Example

S → AaAb
S → BbBa
A → ε
B → ε

I_0: S' → .S ,$
S → .AaAb ,$
S → .BbBa ,$
A → . ,a
B → . ,b

I_1: S' → S. ,$ (S)

I_2: S → A.aAb ,$ (A)

I_3: S → B.bBa ,$ (B)

a → to I_4

b → to I_5

I_4: S → Aa.Ab ,$
A → . ,b

A → I_6: S → AaA.b ,$

a → I_8: aS → AaAb. ,$

I_5: S → Bb.Ba ,$
B → . ,a

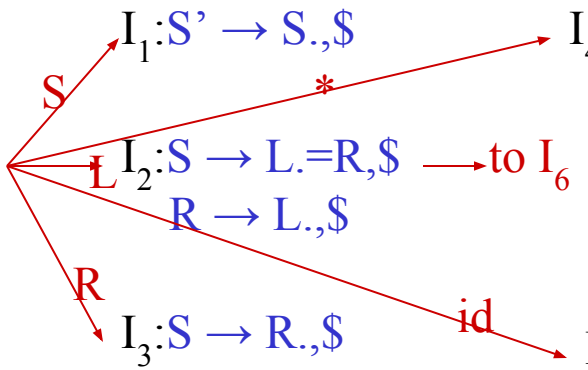
B → I_7: S → BbB.a ,$

b → I_9: bS → BbBa. ,$

Canonical LR(1) Collection – Example2

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

I_0: S' → .S,$
S → .L=R,$
S → .R,$
L → .*R,$/=
L → .id,$/=
R → .L,$

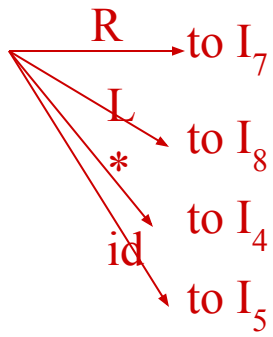


I_1: S' → S.,$

I_2: S → L.=R,$
R → L.,$

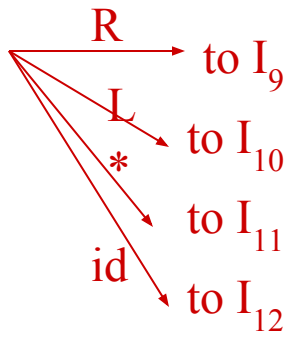
I_3: S → R.,$

I_4: L → *.R,$/=
R → .L,$/=
L→ .*R,$/=
L → .id,$/=



I_5: L → id.,$/=

I_6: S → L=.R,$
R → .L,$
L → .*R,$
L → .id,$



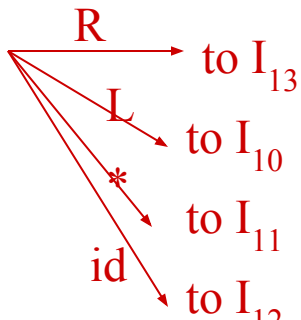
I_7: L → *R.,$/=

I_8: R → L.,$/=

I_9: S → L=R.,$

I_{10}: R → L.,$

I_{11}: L → *.R,$
R → .L,$
L→ .*R,$
L → .id,$



I_{12}: L → id.,$

I_{13}: L → *R.,$

I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

Construction of LR(1) Parsing Tables

1. Construct the canonical collection of sets of LR(1) items for G'. C← $\{I_0,...,I_n\}$
2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha \bullet a\beta,b$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j.***
   - If $A \rightarrow \alpha \bullet,a$ is in I_i, then action[i,a] is ***reduce*** $A \rightarrow \alpha$ where $A \neq S'$.
   - If $S' \rightarrow S \bullet,\$$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not LR(1).
3. Create the parsing goto table
   - for all non-terminals A, if $goto(I_i,A)=I_j$ then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains $S' \rightarrow .S,\$$

LR(1) Parsing Tables – (for Example2)

	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	s12	s11				10	13
12				r4			
13				r3			

no shift/reduce or
no reduce/reduce conflict

⇓

so, it is a LR(1) grammar

LALR Parsing Tables

- **LALR** stands for **LookAhead LR.**

- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
- The number of states in SLR and LALR parsing tables for a grammar G are equal.
- But LALR parsers recognize more grammars than SLR parsers.
- ***yacc*** creates a LALR parser for the given grammar.
- A state of LALR parser will be again a set of LR(1) items.

Creating LALR Parsing Tables

Canonical LR(1) Parser □ LALR Parser

shrink # of states

- This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser (so the grammar is NOT LALR)
- But, this shrik process does not produce a **shift/reduce** conflict.

The Core of A Set of LR(1) Items

- The core of a set of LR(1) items is the set of its first component.

Ex: $S \rightarrow L \bullet = R, \$$ $S \rightarrow L \bullet = R$ Core

$R \rightarrow L \bullet, \$$ $R \rightarrow L \bullet$

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

I_1: $L \rightarrow id \bullet, =$ A new state: I_{12}: $L \rightarrow id \bullet, =$

$L \rightarrow id \bullet, \$$

I_2: $L \rightarrow id \bullet, \$$ have same core, merge them

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

Creation of LALR Parsing Tables

- Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
- Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.

  $C=\{I_0,...,I_n\} \rightarrow C'=\{J_1,...,J_m\}$ where $m \leq n$
- Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
  - Note that: If $J=I_1 \cup ... \cup I_k$ since $I_1,...,I_k$ have same cores
    $\rightarrow$ cores of $goto(I_1,X),...,goto(I_2,X)$ must be same.
  - So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_1,X)$.
- If no conflict is introduced, the grammar is LALR(1) grammar. (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)

Shift/Reduce Conflict

- We say that we cannot introduce a shift/reduce conflict during the shrink process for the creation of the states of a LALR parser.
- Assume that we can introduce a shift/reduce conflict. In this case, a state of LALR parser must have:

$$A \rightarrow \alpha \bullet, a \text{ and } B \rightarrow \beta \bullet a\gamma, b$$

- This means that a state of the canonical LR(1) parser must have:

$$A \rightarrow \alpha \bullet, a \text{ and } B \rightarrow \beta \bullet a\gamma, c$$

But, this state has also a shift/reduce conflict. i.e. The original canonical LR(1) parser has a conflict.

(Reason for this, the shift operation does not depend on lookaheads)

Reduce/Reduce Conflict

- But, we may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser.

$I_1 : A \rightarrow \alpha\bullet,a$ $\quad$ $I_2: A \rightarrow \alpha\bullet,b$

$B \rightarrow \beta\bullet,b$ $\quad$ $B \rightarrow \beta\bullet,c$

$\Downarrow$

$I_{12}: A \rightarrow \alpha\bullet,a/b$ reduce/reduce conflict

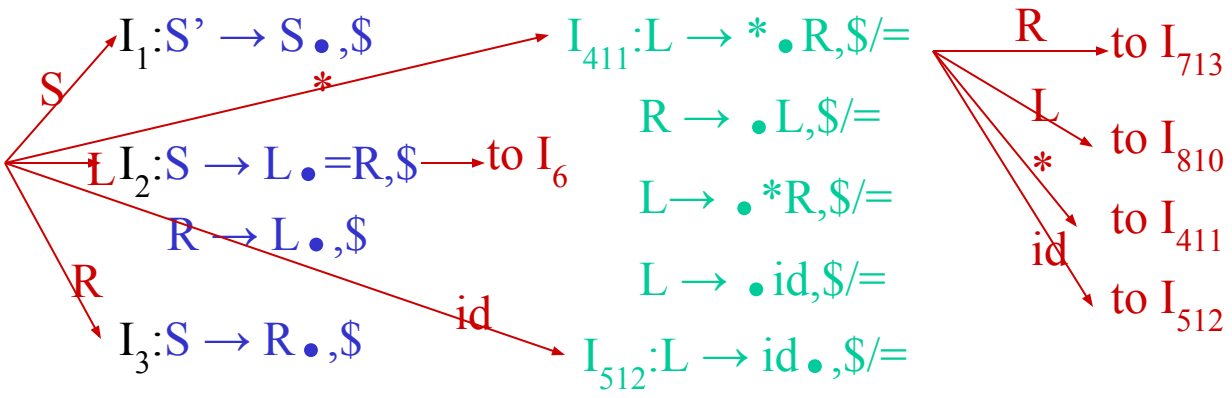
$B \rightarrow \beta\bullet,b/c$

Canonical LALR(1) Collection – Example2

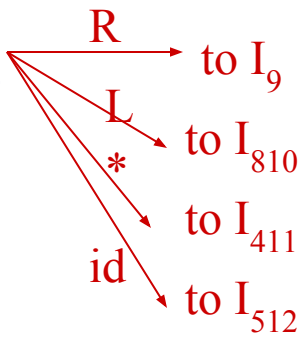
S' → S
1) S → L=R
2) S → R
3) L→ *R
4) L → id
5) R → L

I_0: S' → •S,$
S → •L=R,$
S → •R,$
L → •*R,$/=
L → •id,$/=
R → •L,$

I_1: S' → S•,$
I_2: S → L•=R,$
R → L•,$
I_3: S → R•,$
I_{411}: L → *•R,$/=
R → •L,$/=
L→ •*R,$/=
L → •id,$/=
I_{512}: L → id•,$/=



I_6: S → L=•R,$
R → •L,$
L → •*R,$
L → •id,$



I_9: S → L=R•,$

I_{713}: L → *R•,$/=

I_{810}: R → L•,$/=

Same Cores
I_4 and I_{11}
I_5 and I_{12}
I_7 and I_{13}
I_8 and I_{10}

LALR(1) Parsing Tables – (for Example2)

	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			

no shift/reduce or
no reduce/reduce conflict

so, it is a LALR(1) grammar

Using Ambiguous Grammars

- All grammars used in the construction of LR-parsing tables must be un-ambiguous.
- Can we create LR-parsing tables for ambiguous grammars ?
  - Yes, but they will have conflicts.
  - We can resolve these conflicts in favor of one of them to disambiguate the grammar.
  - At the end, we will have again an unambiguous grammar.
- Why we want to use an ambiguous grammar?
  - Some of the ambiguous grammars are **much natural**, and a corresponding unambiguous grammar can be very complex.
  - Usage of an ambiguous grammar may **eliminate unnecessary reductions**.
- Ex.

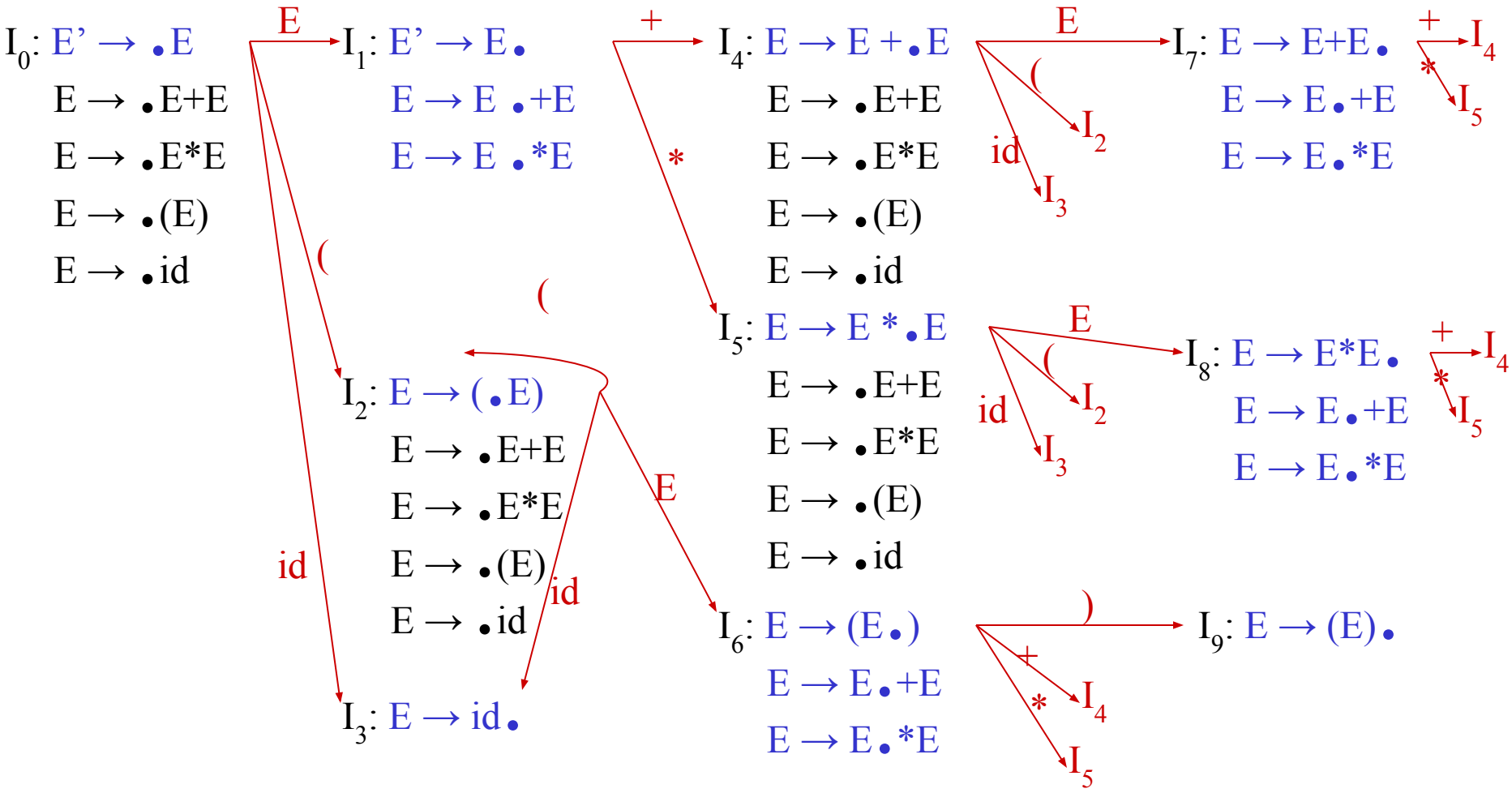
$E \rightarrow E+E \mid E*E \mid (E) \mid id$ □

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Sets of LR(0) Items for Ambiguous Grammar



SLR-Parsing Tables for Ambiguous Grammar

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

State I_7 has shift/reduce conflicts for symbols + and *.

$$I_0 \xrightarrow{E} I_1 \xrightarrow{+} I_4 \xrightarrow{E} I_7$$

when current token is +

shift □ + is right-associative

reduce □ + is left-associative

when current token is *

shift □ * has higher precedence than +

reduce □ + has higher precedence than *

SLR-Parsing Tables for Ambiguous Grammar

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

State I_8 has shift/reduce conflicts for symbols + and *.

$$I_0 \xrightarrow{E} I_1 \xrightarrow{*} I_5 \xrightarrow{E} I_7$$

when current token is *

shift → * is right-associative

reduce → * is left-associative

when current token is +

shift → + has higher precedence than *

reduce → * has higher precedence than +

SLR-Parsing Tables for Ambiguous Grammar

	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	