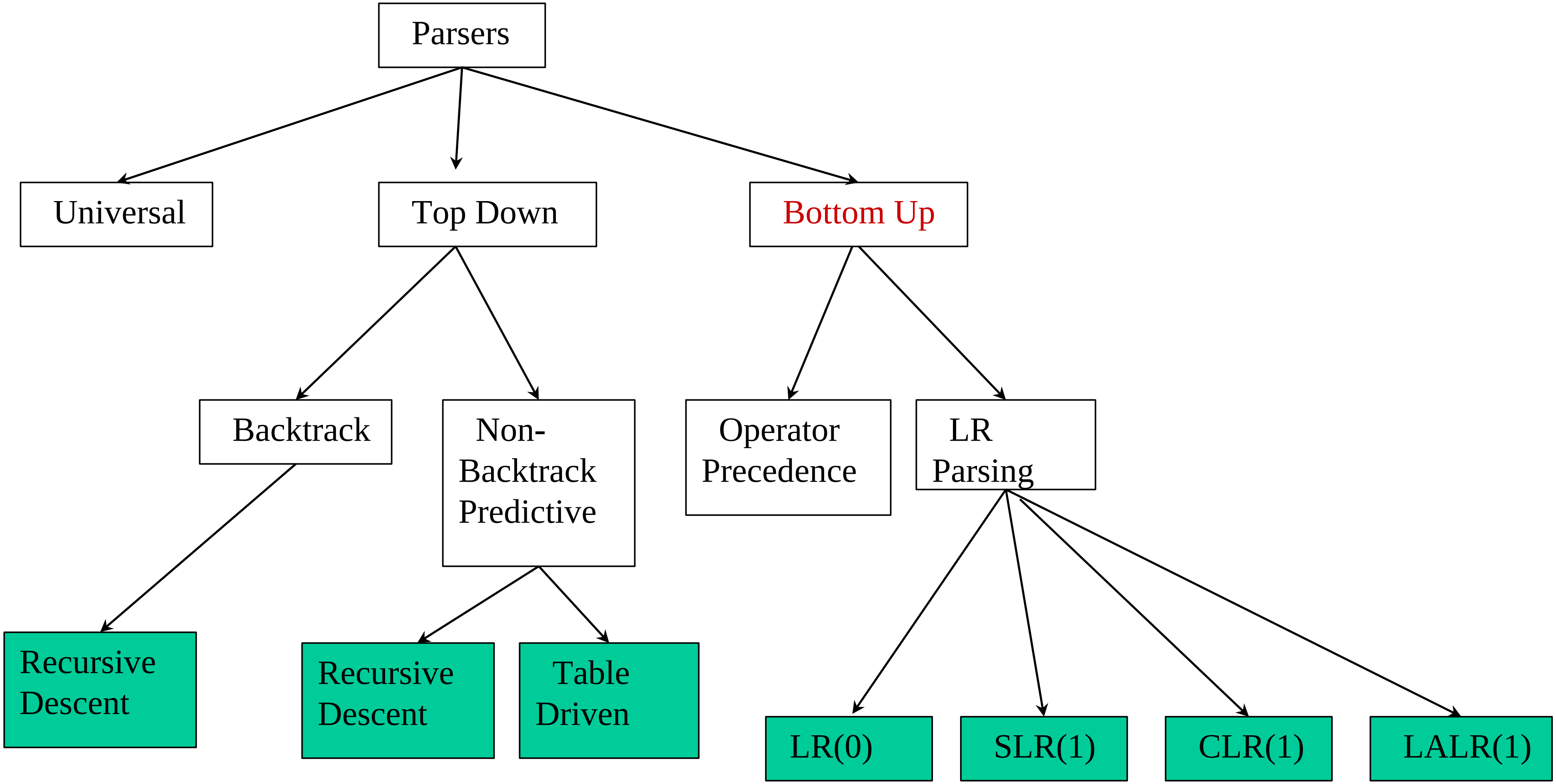
Types of Parsers



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.

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

  ← (the bottom-up parser finds the right-most derivation in the reverse order)
- Bottom-up parsing is also known as **shift-reduce parsing** because its two main actions are shift and reduce.
  - At each shift action, the current symbol in the input string is pushed to a stack.
  - At each reduction step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will replaced by the non-terminal at the left side of that production.
  - There are also two more actions: accept and error.

Shift-Reduce Parsing

- A shift-reduce parser tries to reduce the given input string into the starting symbol.

a string □ the starting symbol

reduced to

- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.
- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

Rightmost Derivation: $S \underset{rm}{\overset{*}{\Rightarrow}} \omega$

Shift-Reduce Parser finds: $S \underset{rm}{\Leftarrow} \ldots \underset{rm}{\Leftarrow} \omega$

Shift-Reduce Parsing -- Example

$S \rightarrow aABb$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

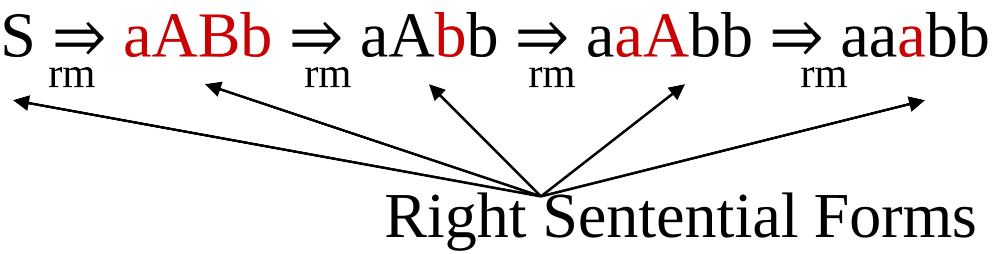
input string: `aaabb`

`aaAbb`

`aAbb` ⇓ reduction

`aABb`

`S`



$$S \underset{rm}{\Rightarrow} aABb \underset{rm}{\Rightarrow} aAbb \underset{rm}{\Rightarrow} aaAbb \underset{rm}{\Rightarrow} aaabb$$

Right Sentential Forms

- How do we know which substring to be replaced at each reduction step?

Handle

- Informally, a **handle** of a string is a substring that matches the right side of a production rule.
  - But not every substring matches the right side of a production rule is handle
- Every string of symbols in the derivation is a **sentential form** and a **sentence *is a sentential*** form that has only terminal symbols
- A **handle** of a right sentential form γ ($\equiv \alpha\beta\omega$) is

  a production rule A → β 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.

$$S \overset{*}{\overset{rm}{\Rightarrow}} \alpha A \omega \overset{rm}{\Rightarrow} \alpha\beta\omega$$

- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

Handle Pruning

- A right-most derivation in reverse can be obtained by **handle-pruning**.

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

input string

- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n in by A_n to get γ_{n-1}.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} in by A_{n-1} to get γ_{n-2}.
- Repeat this, until we reach S.

A Shift-Reduce Parser

E → E+T | T
T → T*F | F
F → (E) | id

Right-Most Derivation of `id+id*id`

E ⇒ E+T ⇒ E+T*F ⇒ E+T*id ⇒ E+F*id
⇒ E+id*id ⇒ T+id*id ⇒ F+id*id ⇒ id+id*id

Right-Most Sentential Form	Reducing Production (RMD in reverse)
`id+id*id`	F → id
`F+id*id`	T → F
`T+id*id`	E → T
`E+id*id`	F → id
`E+F*id`	T → F
`E+T*id`	F → id
`E+T*F`	T → T*F
`E+T`	E → E+T
`E`	

`Handles` are red and underlined in the right-sentential forms.

A Stack Implementation of A Shift-Reduce Parser

- There are four possible actions of a shift-reduce parser:

  1. **Shift** : The next input symbol is shifted onto the top of the stack.
  2. **Reduce**: Replace the handle on the top of the stack by the non-terminal.
  3. **Accept**: Successful completion of parsing.
  4. **Error**: Parser discovers a syntax error, and calls an error recovery routine.

- Initial stack just contains only the end-marker $ or starting state.
- The end of the input string is marked by the end-marker $.

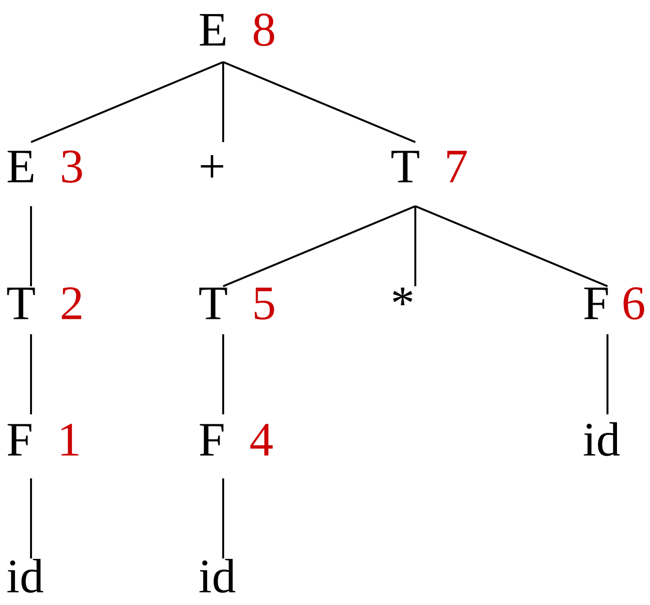
Shift-Reduce Parsing

```
push '$' onto the stack;
token = nextToken();
repeat
    if (there is a handle  A::=b on top of the stack){
            reduce b  to  A; /* reduce */
            pop b off the stack;
            push A onto the stack;
    } else {/* shift */
            shift token onto the stack;
            token = nextToken();
    }
until (top of stack is S and token is eof)
```

A Stack Implementation of A Shift-Reduce Parser

Stack	**Input**	**Action**
$	id+id*id$	shift
$id	+id*id$	reduce by F → id
$F	+id*id$	reduce by T → F
$T	+id*id$	reduce by E → T
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by F → id
$E+F	*id$	reduce by T → F
$E+T	*id$	shift
$E+T*	id$	shift
$E+T*id	$	reduce by F → id
$E+T*F	$	reduce by T → T*F
$E+T	$	reduce by E → E+T
$E	$	accept

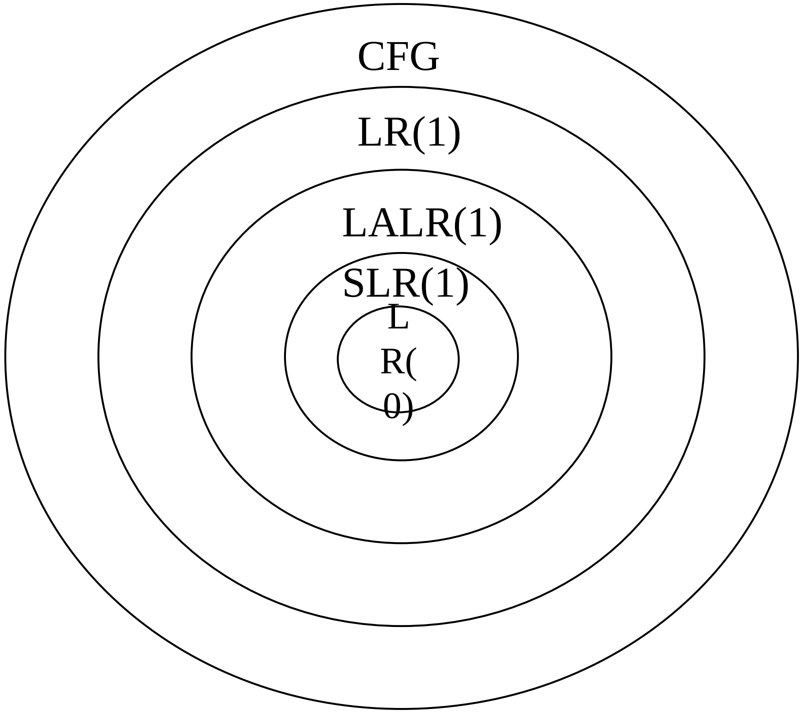
Parse Tree



Shift-Reduce Parsers

- There are two main categories of shift-reduce parsers

1. **Operator-Precedence Parser**
   - simple, but only a small class of grammars.



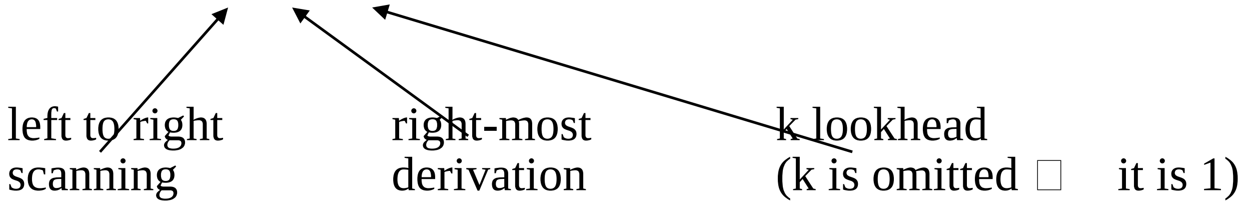
2. **LR-Parsers**
   - Main: LR(0), LR(1)
   - Some variations SLR and LALR(1)

   - LR(0) Items – LR(0),SLR(1)
   - LR(1) Items – LR(1), LALR(1)

LR Parsers

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

LR(k) parsing.



left to right scanning

right-most derivation

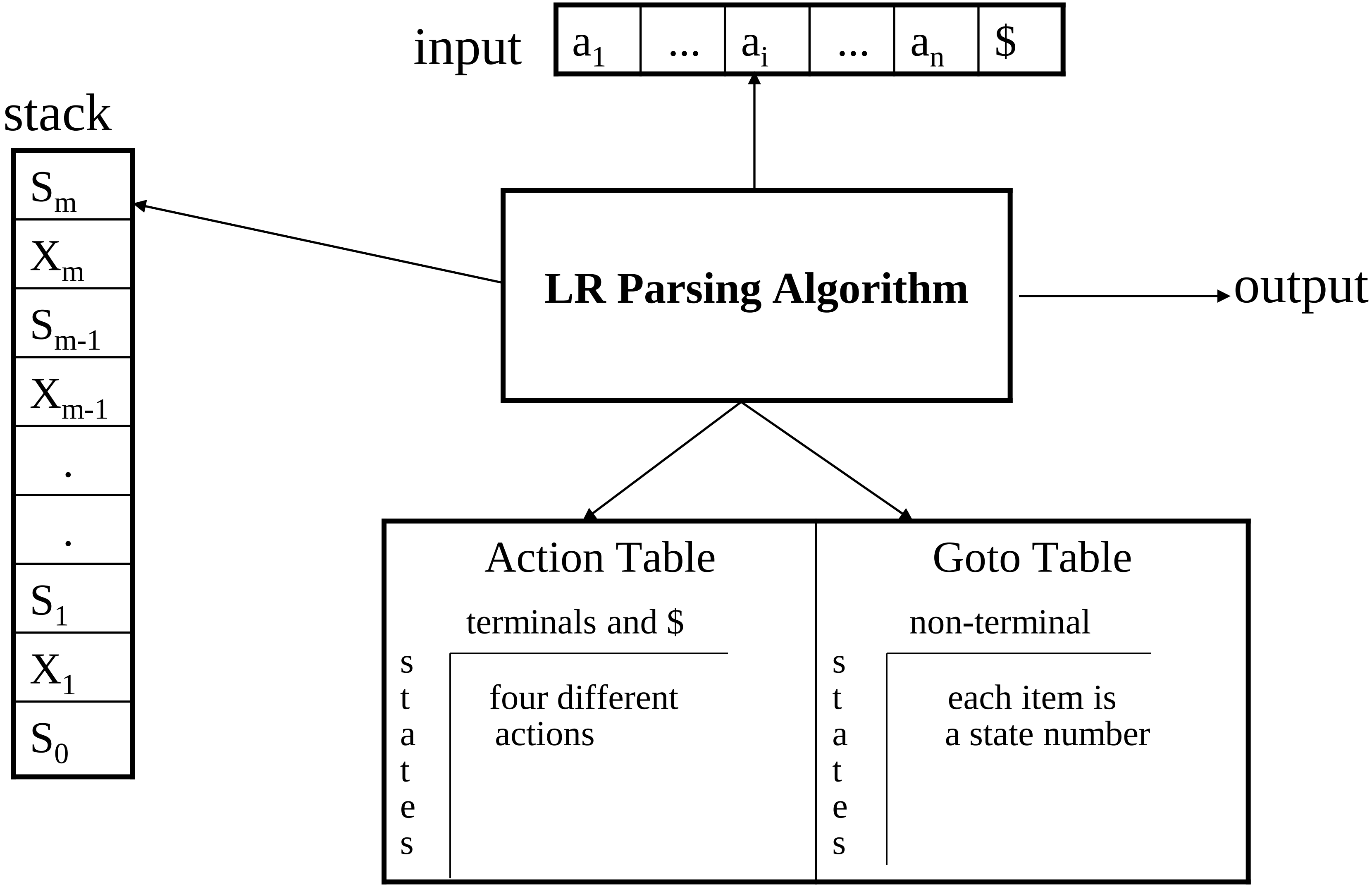
k lookhead (k is omitted □ it is 1)

- 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.

    $$\text{LL(1)-Grammars} \subset \text{LR(1)-Grammars}$$

  - 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 Parsing Algorithm



input

a_1 ... a_i ... a_n $

stack

S_m

X_m

S_{m-1}

X_{m-1}

.

.

S_1

X_1

S_0

LR Parsing Algorithm

output

Action Table

terminals and $

states

four different actions

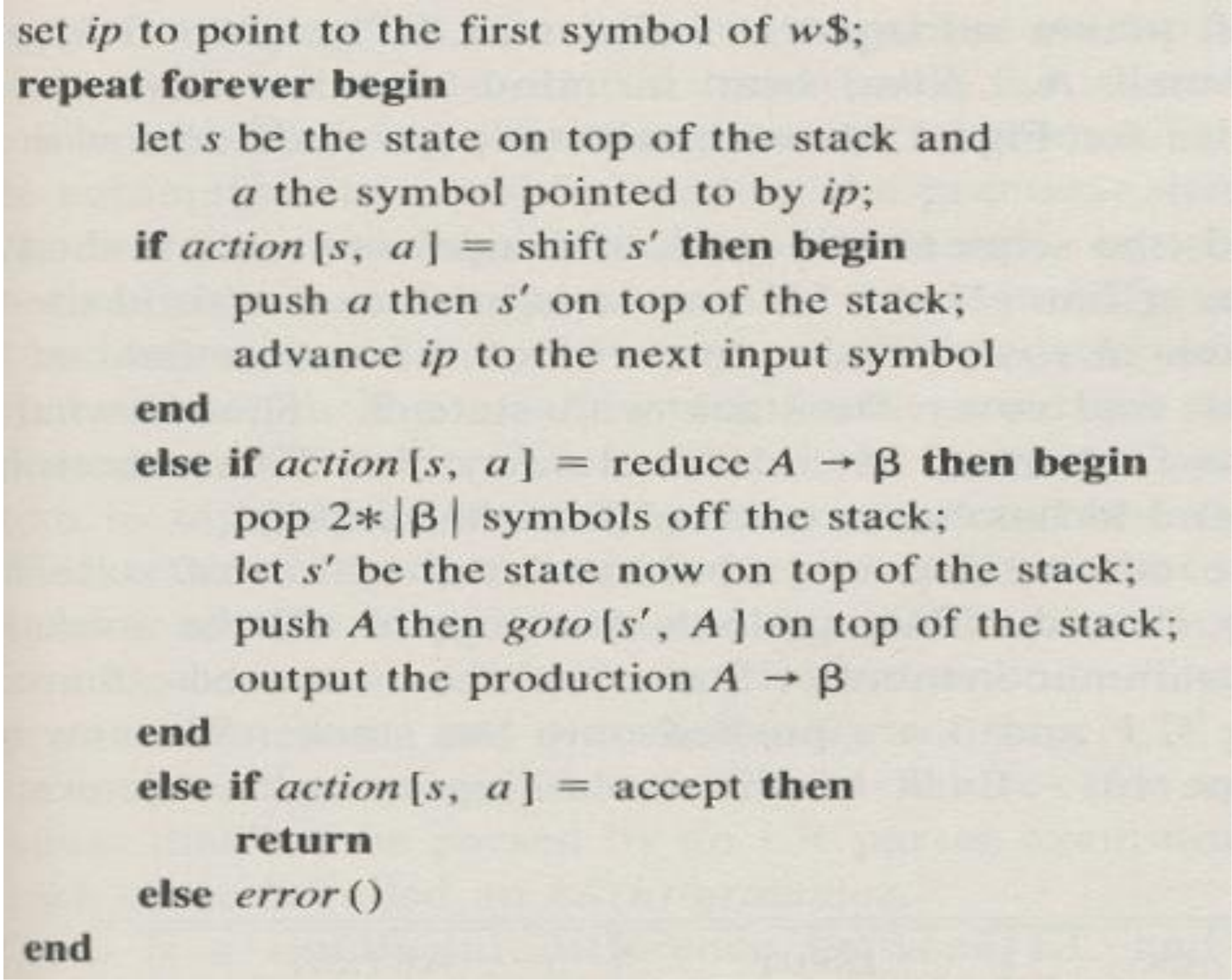
Goto Table

non-terminal

states

each item is a state number

LR Parsing Algorithm



```
set ip to point to the first symbol of $w\$$;
repeat forever begin
        let $s$ be the state on top of the stack and
                $a$ the symbol pointed to by $ip$;
        if $action[s, a]$ = shift $s'$ then begin
                push $a$ then $s'$ on top of the stack;
                advance $ip$ to the next input symbol
        end
        else if $action[s, a]$ = reduce $A \rightarrow \beta$ then begin
                pop $2 * |\beta|$ symbols off the stack;
                let $s'$ be the state now on top of the stack;
                push $A$ then $goto[s', A]$ on top of the stack;
                output the production $A \rightarrow \beta$
        end
        else if $action[s, a]$ = accept then
                return
        else error()
end
```

A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:

$$(\underbrace{S_o\ X_1\ S_1 \ldots X_m\ S_m}_{\text{Stack}},\ \underbrace{a_i\ a_{i+1} \ldots a_n\ \$}_{\text{Rest of Input}})$$

- 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

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

2. **reduce A→β** (or **`rn`** where n is a production number)
   - pop 2|**β**| (=r) items from the stack;
   - then push **A** and **s** where **s=goto[s$_{\mathbf{m-r}}$,A] ---** after removing r many items from stack whatever is available state on top of stack

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

   - Output is the reducing production reduce A→β

3. **Accept** – Parsing successfully completed

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

Parsing Table

- It has two types of entries: ACTION, GOTO
- ACTION : two dimensional array indexed by state and terminals symbols
- GOTO : two dimensional array indexed by the state number and a Non-terminals

(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

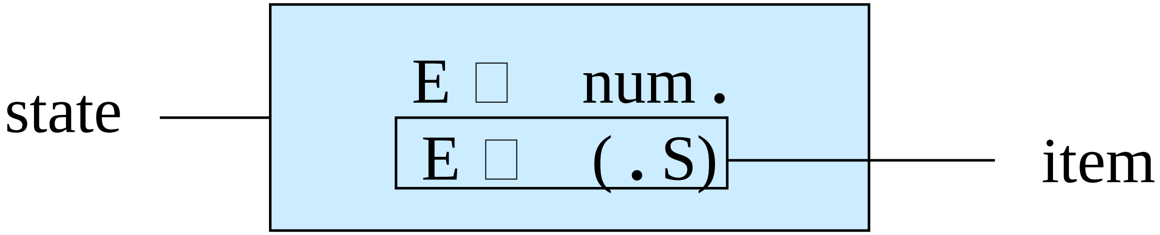
stack	**input**	**action**	**output**
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5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

LR(0) Parsers

  - Determine the actions without any lookahead
  - Will help us understand shift-reduce parsing
- To build the parsing table:
  - Define states of the parser
  - Build a DFA to describe transitions between states
  - Use the DFA to build the parsing table
- Each LR(0) state is a set of LR(0) items
  - An LR(0) item: X □ α . β where X □ αβ is a production in the grammar
  - The LR(0) items keep track of the progress on all of the possible upcoming productions
  - The item X □ α . β abstracts the fact that the parser already matched the string α at the top of the stack

Example LR(0) State

- An LR(0) item is a production from the language with a separator "." somewhere in the RHS of the production



- Sub-string before "." is already on the stack
- Sub-string after ".": what we might see next

LR(0) Items

- LR(0) item
  - A production with a dot at some position of the RHS

A ::= •XYZ	we are expecting XYZ
A ::= X •YZ	
A ::= XY •Z	
A ::= XYZ•	we have seen XYZ

Constructing LR(0) 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: $A \to aBb$ Possible LR(0) Items: $A \to \bullet aBb$

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

  $A \to aB \bullet b$

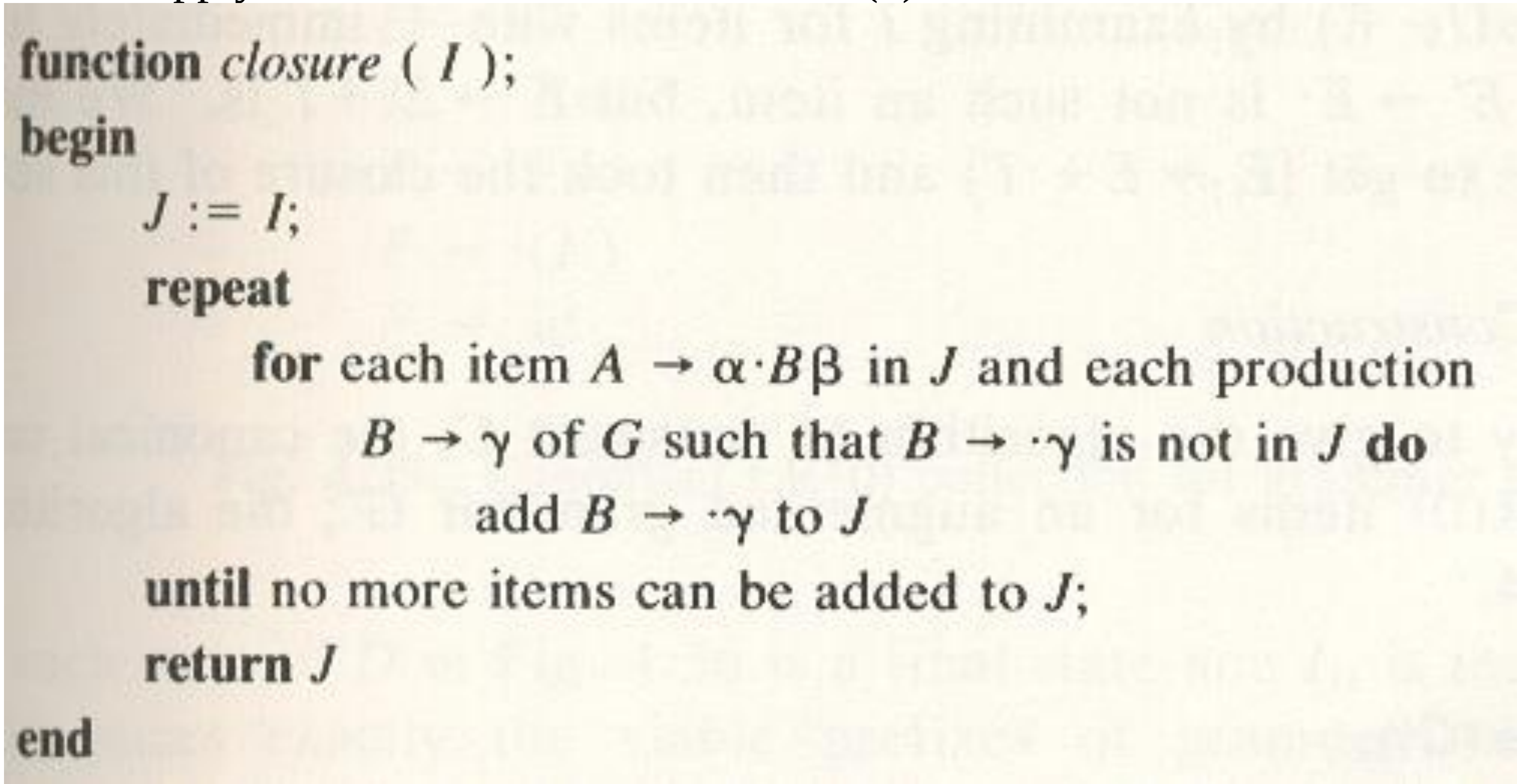
  $A \to aBb \bullet$
- Sets of LR(0) items will be the states of action and goto table of the SLR parser.
- A collection of sets of LR(0) items (**the canonical LR(0) collection**) is the basis for constructing SLR parsers.
- *Augmented Grammar*:

  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 → α.Bβ is in closure(I) and B→γ is a production rule of G; then B→.γ will be in the closure(I). We will apply this rule until no more new LR(0) items can be added to closure(I).



```
function closure ( I );
begin
    J := I;
    repeat
        for each item $A \rightarrow \alpha \cdot B\beta$ in J and each production
            $B \rightarrow \gamma$ of G such that $B \rightarrow \cdot\gamma$ is not in J do
                add $B \rightarrow \cdot\gamma$ to J
    until no more items can be added to J;
    return J
end
```

The Closure Operation -- Example

$E' \rightarrow E$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

closure($\{E' \rightarrow \bullet E\}$) =

$\{\ E' \rightarrow \bullet E$ ← kernel items

$E \rightarrow \bullet E+T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet T*F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet id\ \}$

Goto 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 → α.Xβ in I then every item in **closure({A → αX.β})** will be in goto(I,X).

Example:

I ={ E’ → .E, E → .E+T,

E → .T, T → .T*F, T → .F,

F → .(E), F → .id }

goto(I,E) = { E’ → E., E → E.+T }

goto(I,T) = { E → T., T → T.*F }

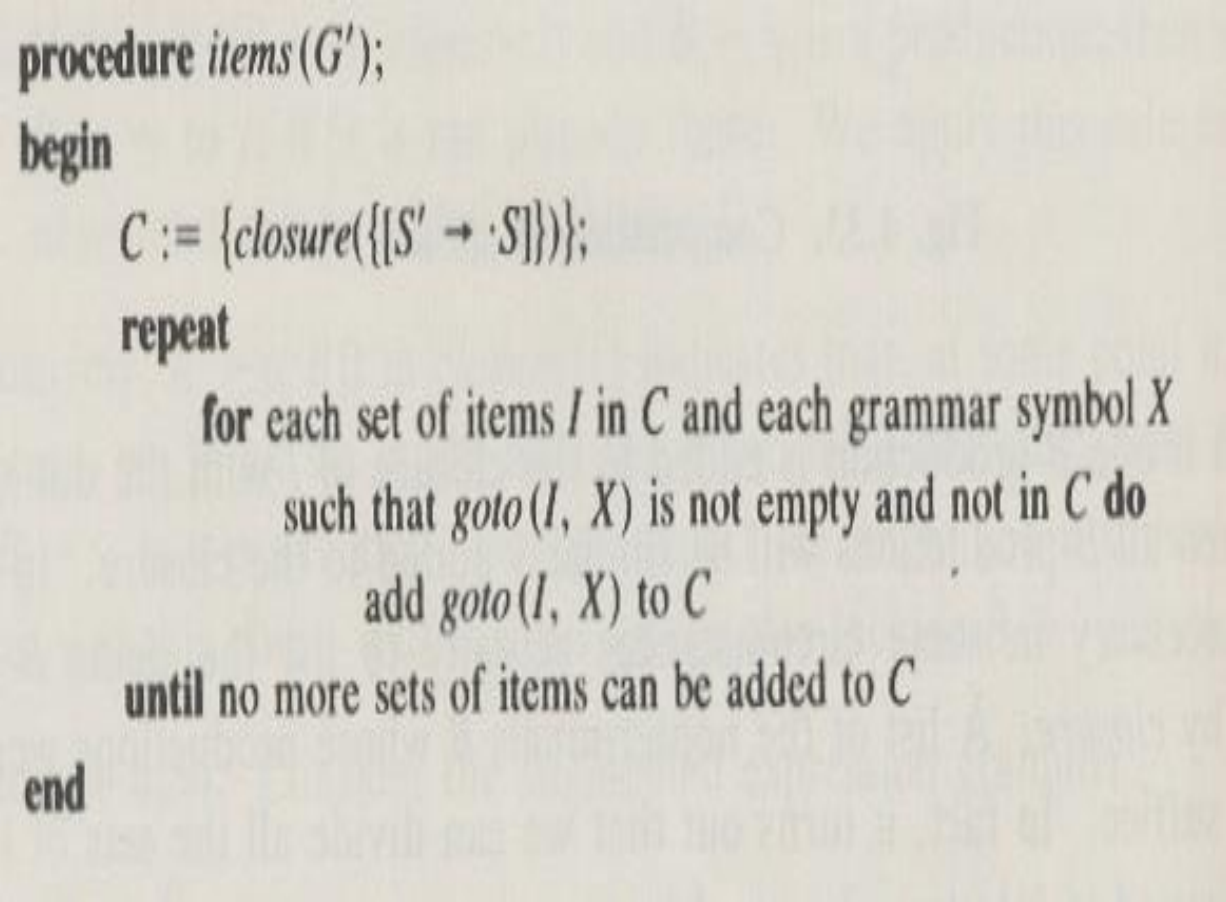
goto(I,F) = {T → F. }

goto(I,() = { F → (.E), E → .E+T,

E → .T, T → .T*F, T → .F,

F → .(E), F → .id }

goto(I,id) = { F → id. }



```
procedure items(G');
begin
    C := {closure({[S' → ·S]})};
    repeat
        for each set of items I in C and each grammar symbol X
            such that goto(I, X) is not empty and not in C do
                add goto(I, X) to C
    until no more sets of items can be added to C
end
```

Construction of The Canonical LR(0) Collection

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

Start State and Closure

- Start state
  - – Augment grammar with production: $S' \rightarrow S\ \$$
  - – Start state of DFA has empty stack: $S' \rightarrow .\ S\ \$$
- Closure of a parser state:
  - – Start with Closure(S) = S
  - – Then for each item in S:
    - $X \rightarrow \alpha\ .\ Y\ \beta$
    - Add items for all the productions $Y \rightarrow \gamma$ to the closure of S: $Y \rightarrow .\ \gamma$

Closure

S □ (L) | id

L □ S | L,S

DFA start state

S □ . S $

closure

S' □ . S $

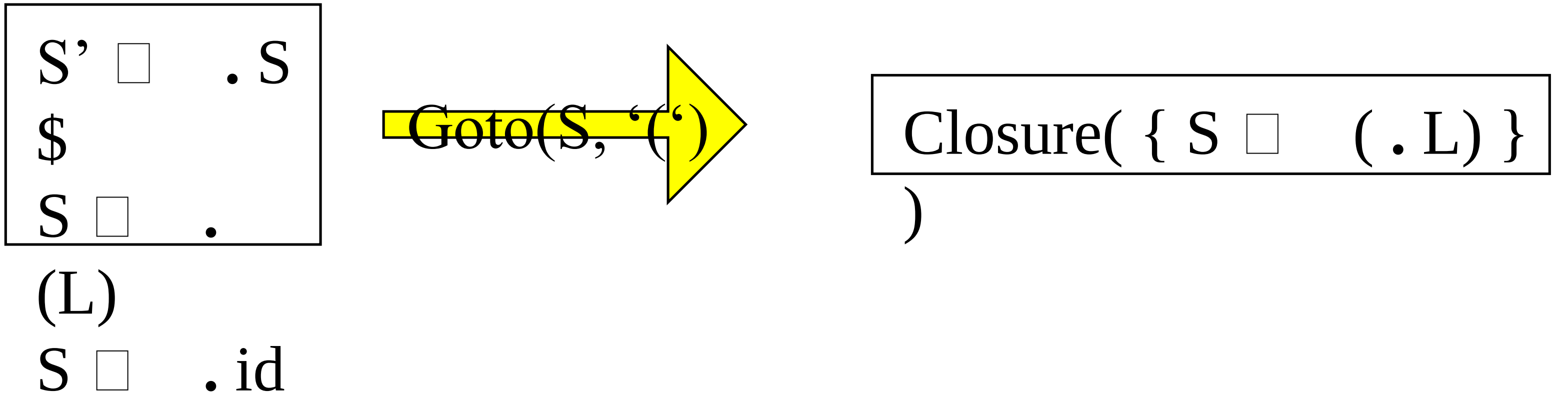
S □ . (L)

S □ . id

- Set of possible productions to be reduced next
- Closure of a parser state, S:
  - Start with Closure(S) = S
  - Then for each item in S:
    - X □ α . Y β
    - Add items for all the productions Y □ γ to the closure of S: Y □ . γ

The Goto Operation

- Goto operation = describes transitions between parser states, which are sets of items
- Algorithm: for state S and a symbol Y
  - If the item [X □ α . Y β] is in I, then
  - Goto(I, Y) = Closure([X □ α Y . β])

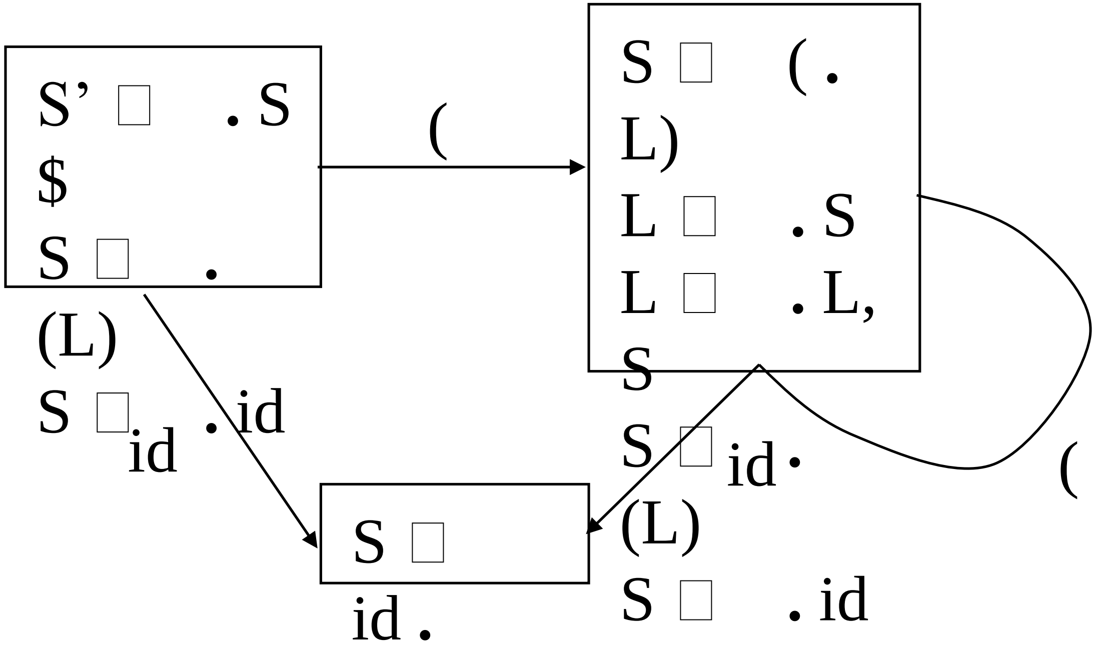


S' □ . S $
S □ . (L)
S □ . id

Goto(S, '(')

Closure({ S □ (. L) })

Goto: Terminal Symbols



S’ □ . S $

S □ . (L)

S □ . id

(

id

S □ (. L)

L □ . S

L □ . L, S

S □ . (L)

S □ . id

id

(

S □ id .

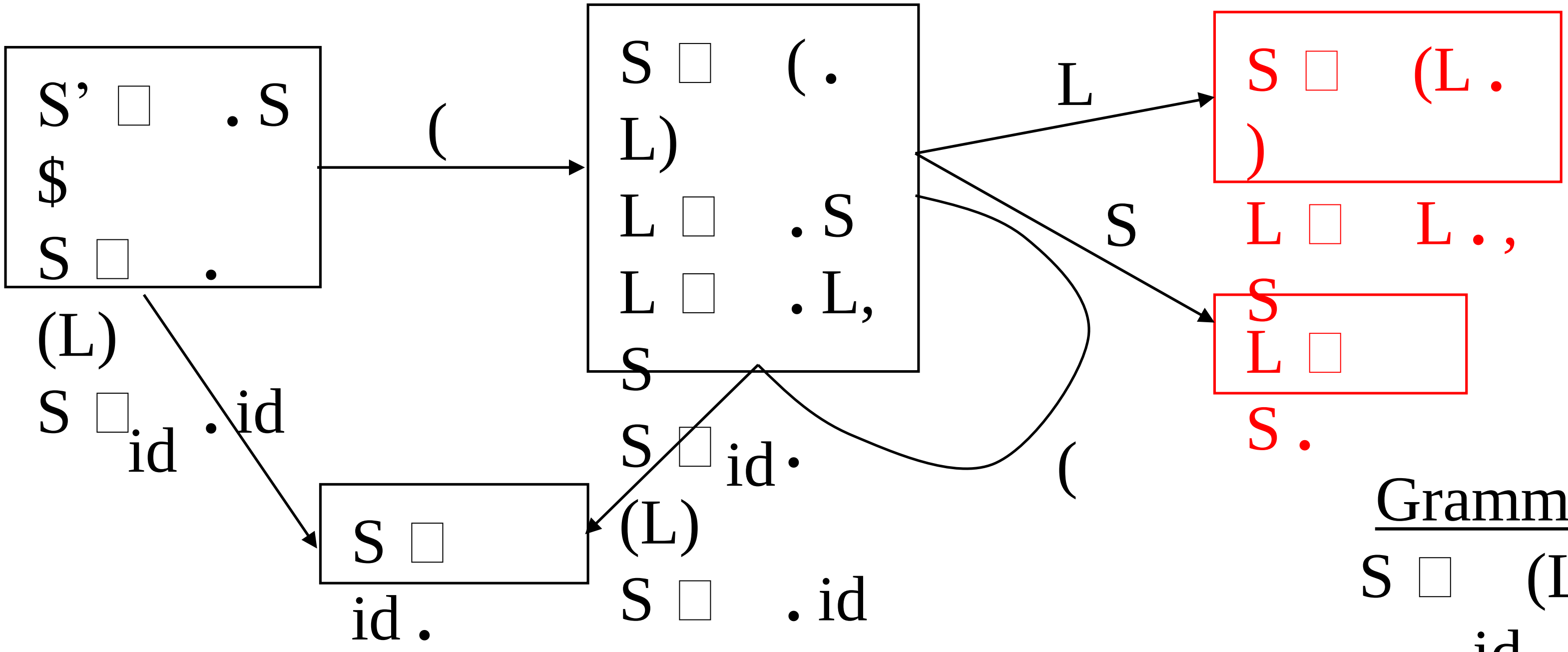
<u>Grammar</u>

S □ (L) | id

L □ S | L,S

In new state, include all items that have appropriate input symbol just after dot, advance do in those items and take closure

Goto: Non-terminal Symbols



S’ □ . S $
S □ . (L)
S □ . id

(

S □ (. L)
L □ . S
L □ . L, S
S □ . (L)
S □ . id

L

S □ (L .)
L □ L . , S

S

L □ S .

id

S □ id .

id

(

Grammar

S □ (L) | id
L □ S | L,S

same algorithm for transitions on non-terminals

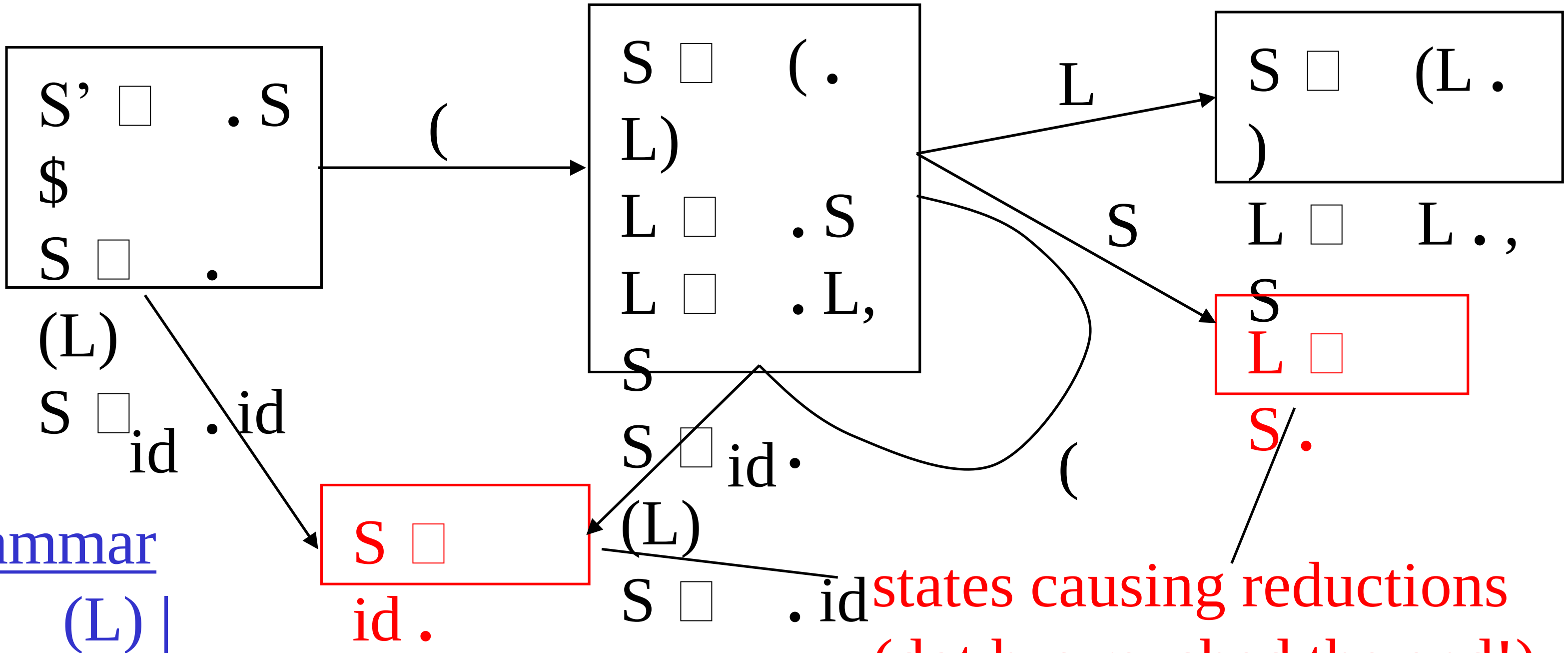
Example

E’ □ E
E □ E + T |
T
T □ T * F |
F
F □ (E) | id

1. If I = { [E’ □ . E]}, then Closure(I) = ??

1. If I = { [E’ □ E .], [E □ E . + T] }, then Goto(I,+) = ??

Applying Reduce Actions



states causing reductions
(dot has reached the end!)

Grammar

S □ (L) | id

L □ S | L,S

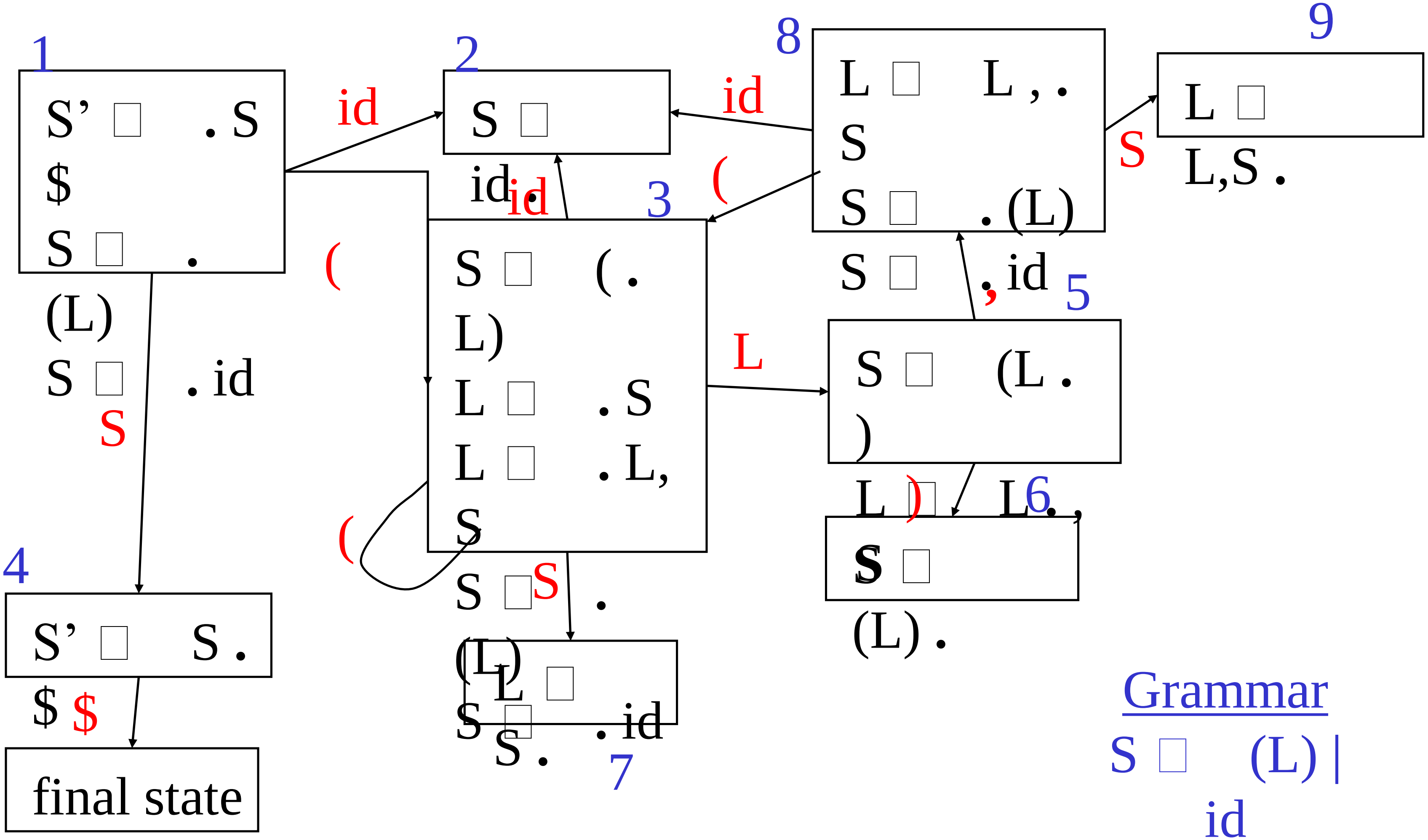
Pop RHS off stack, replace with LHS X (X □ β),
then rerun DFA (e.g., (x))

Reductions

- On reducing X □ β with stack αβ
  - – Pop β off stack, revealing prefix α and state
  - – Take single step in DFA from top state
  - – Push X onto stack with new DFA state
- Example

derivation	stack	input	action
((a),b) □	1 (3 (3	a),b)	shift, goto 2
((a),b) □ id	1 (3 (3 a 2	),b)	reduce S □
((S),b) □ S	1 (3 (3 S 7	),b)	reduce L □

Full DFA



1
S' □ . S $
S □ . (L)
S □ . id

2
S □ id .

3
S □ (. L)
L □ . S
L □ . L, S
S □ . (L)
S □ . id

4
S' □ S . $

final state

5
S □ (L .)
L □ L . , S

6
S □ (L) .

7
L □ S .

8
L □ L , . S
S □ . (L)
S □ . id

9
L □ L,S .

Grammar
S □ (L) | id
L □ S | L,S

Parsing Example ((a),b)

derivation	stack	input	action
((a),b) □	1	((a),b)$	shift, goto 3
((a),b) □	1(3	(a),b)	shift, goto 3
((a),b) □	1(3(3	a),b)	shift, goto 2
((a),b) □	1(3(3a2	),b)	reduce S□ id
((S),b) □	1(3(3(S7	),b)	reduce L□ S
((L),b) □	1(3(3(L5	),b)	shift, goto 6
((L),b) □	1(3(3L5)6	,b)	reduce S□ (L)
(S,b) □ L□ S	1(3S7	,b)	reduce
(L,b) □ 8	1(3L5	,b)	shift, goto
(L,b) □ 9	1(3L5,8	b)	shift, goto
(L,b) □ S□ id	1(3L5,8b2	)	reduce
(L,S) □ L□ L,S	1(3L8,S9	)	reduce
(L) □	1(3L5	)	shift, goto 6

S □ (L) | id

L □ S | L,S

S.$->((a),b)$

.dot Indicates already Scanned input

After dot – Expected input

Building the Parsing Table

- States in the table = states in the DFA
- For transition S □ S’ on terminal C:
  - Table[S,C] += Shift(S’)
- For transition S □ S’ on non-terminal N:
  - Table[S,N] += Goto(S’)
- If S is a reduction state X □ β then:
  - Table[S,*] += Reduce(X □ β)

Computed LR(0) Parsing Table

State	Input terminal					Non-terminals	
	(	)	id	,	$	S	L
1	s3		s2			g4	
2	S□ id	S□ id	S□ id	S□ id	S□ id		
3	s3		s2			g7	
	g5						
4					accept		
5		s6		s8			
6	S□ (L)	S□ (L)	S□ (L)	S□ (L)	S□ (L)		
7	L□ S	L□ S	L□ S	L□ S	L□ S		
8	s3		s2			g9	
9	L□ L,S	L□ L,S	L□ L,S	L□ L,S	L□ L,S		

blue = shift

red = reduce

LR(0) Summary

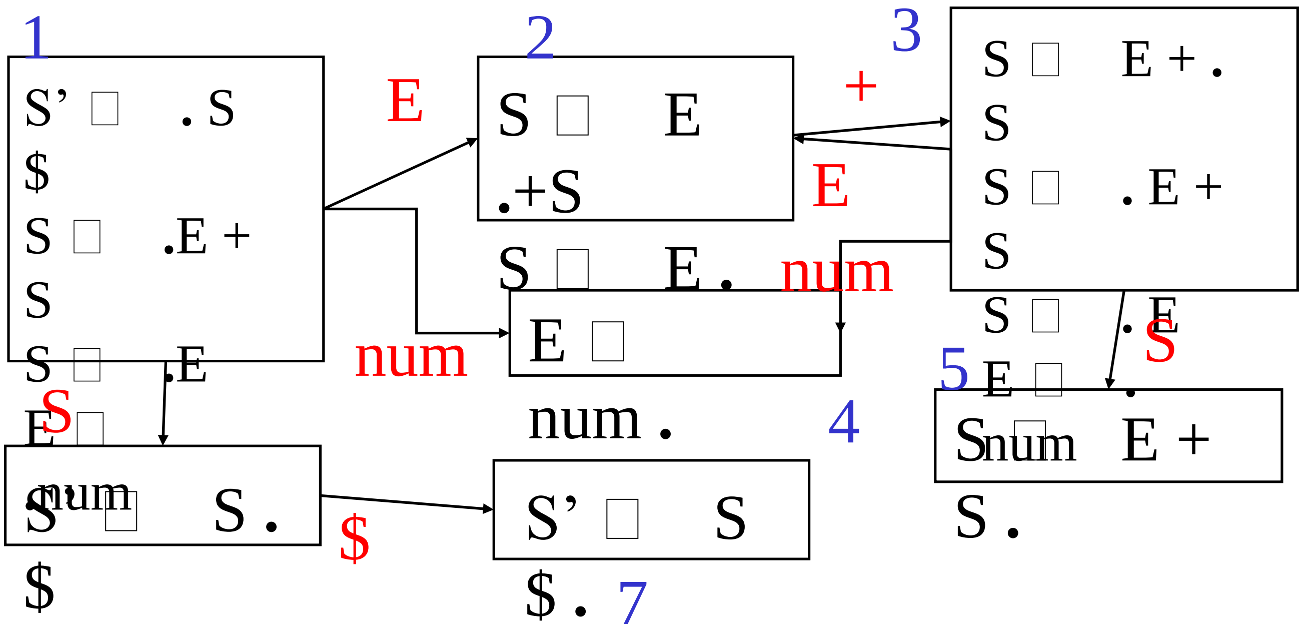
- LR(0) parsing recipe:
  - Start with LR(0) grammar
  - Compute LR(0) states and build DFA:
    - Use the closure operation to compute states
    - Use the goto operation to compute transitions
  - Build the LR(0) parsing table from the DFA

Example

Generate the DFA for the following grammar of addition of numbers

S □ E + S |
E
E □ num

LR(0) Parsing Table



Grammar

S □ E + S | E

E □ num

Shift or reduce in state 2?

	num	+	$	E	S
1	s4 g6			g2	
2	S□ E	s3/S□ E	S□ E		

Conflicts During Shift-Reduce Parsing

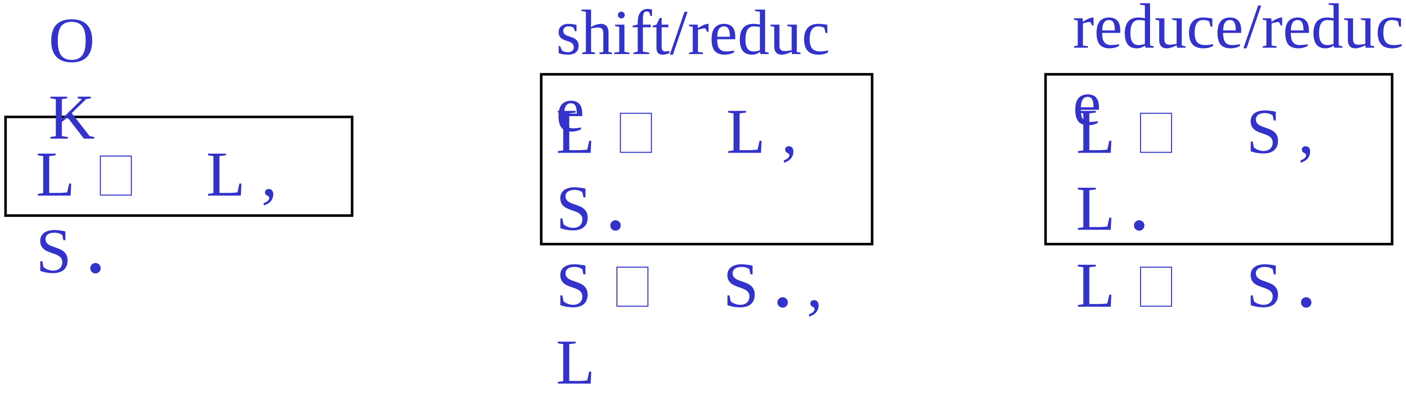
- There are context-free grammars for which shift-reduce parsers cannot be used.
- Stack contents and the next input symbol may not decide action:
  - **shift/reduce conflict**: Whether make a shift operation or a reduction.
  - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.

- An ambiguous grammar can never be a LR grammar.

- Grammar for addition of numbers
  - S □ S + E | E
  - E □ num
- Left-associative version is LR(0)
- While Right-associative is not LR(0) as seen.
  - S □ E + S | E
  - E □ num

LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a single reduce action
  - Always reduce regardless of lookahead
- With a more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Need to use lookahead to choose



OK

L □ L , S .

shift/reduce

L □ L , S .

S □ S . , L

reduce/reduce

L □ S , L .

L □ S .

Solve Conflict With Lookahead

- 3 popular techniques for employing lookahead of 1 symbol with bottom-up parsing
  - SLR – Simple LR
  - LALR – LookAhead LR
  - LR(1)
- Each as a different means of utilizing the lookahead
  - Results in different processing capabilities

Constructing SLR Parsing Table

(of an augumented grammar G')

1. Construct the canonical collection of sets of LR(0) items for G'.

   $C \leftarrow \{I_0,...,I_n\}$

1. 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→α*** for all a in FOLLOW(A) where A≠S'.(see example E→T.)
   - If S'→S. is in I_i , then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).

1. Create the parsing goto table
   - for all non-terminals A, if goto(I_i,A)=I_j then goto[i,A]=j

1. All entries not defined by (2) and (3) are errors.

1. Initial state of the parser contains S'→.S

SLR Parsing

- SLR Parsing = Easy extension of LR(0)
  - For each reduction X □ β, look at next symbol C
  - Apply reduction only if C is in FOLLOW(X)
- SLR parsing table eliminates some conflicts
  - Same as LR(0) table except reduction rows
  - Adds reductions X □ β only in the columns of symbols in FOLLOW(X)

Example: FOLLOW(S) = {$}

Grammar
S □ E + S |
E
E □ num

	num	+	$	E	S
1	s4			g2	g6
2		s3	S□ E		

SLR Parsing Table

- Reductions do not fill entire rows as before
- Otherwise, same as LR(0)

Grammar

S □ E + S | E

E □ num

	num	+	$	E	S
1	s4			g2	g6
2		s3	S□ E		
3	s4			g2	g5
4		E□ num	E□ num		
5			S□ E+S		
6			s7		
7			accept		

Example-1

Consider:

S → L = R
S → R
L → *R
L → ident
R → L

Think of L as l-value, R as r-value, and * as a pointer dereference

Cont.

$S \rightarrow L{=}R$
$S \rightarrow R$
$L \rightarrow {*}R$
$L \rightarrow id$
$R \rightarrow L$

I_0: $S' \rightarrow .S$
$S \rightarrow .L{=}R$
$S \rightarrow .R$
$L \rightarrow .{*}R$
$L \rightarrow .id$
$R \rightarrow .L$

I_1: $S' \rightarrow S.$

I_2: $S \rightarrow L.{=}R$
$R \rightarrow L.$

I_3: $S \rightarrow R.$

I_4: $L \rightarrow {*}.R$
$R \rightarrow .L$
$L \rightarrow .{*}R$
$L \rightarrow .id$

I_5: $L \rightarrow id.$

I_6: $S \rightarrow L{=}.R$
$R \rightarrow .L$
$L \rightarrow .{*}R$
$L \rightarrow .id$

I_7: $L \rightarrow {*}R.$

I_8: $R \rightarrow L.$

I_9: $S \rightarrow L{=}R.$

Problem

FOLLOW(R)={=,\$}

= → shift 6
↘ reduce by $R \rightarrow L$

shift/reduce conflict

Not SLR(1)

Example-2

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

I_0: $S' \rightarrow .S$
$S \rightarrow .AaAb$
$S \rightarrow .BbBa$
$A \rightarrow .$
$B \rightarrow .$

Problem

FOLLOW(A)={a,b}

FOLLOW(B)={a,b}

a → reduce by $A \rightarrow \varepsilon$
↘ reduce by $B \rightarrow \varepsilon$

reduce/reduce conflict

b → reduce by $A \rightarrow \varepsilon$
↘ reduce by $B \rightarrow \varepsilon$

reduce/reduce conflict

Not SLR(1)

Example-3

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

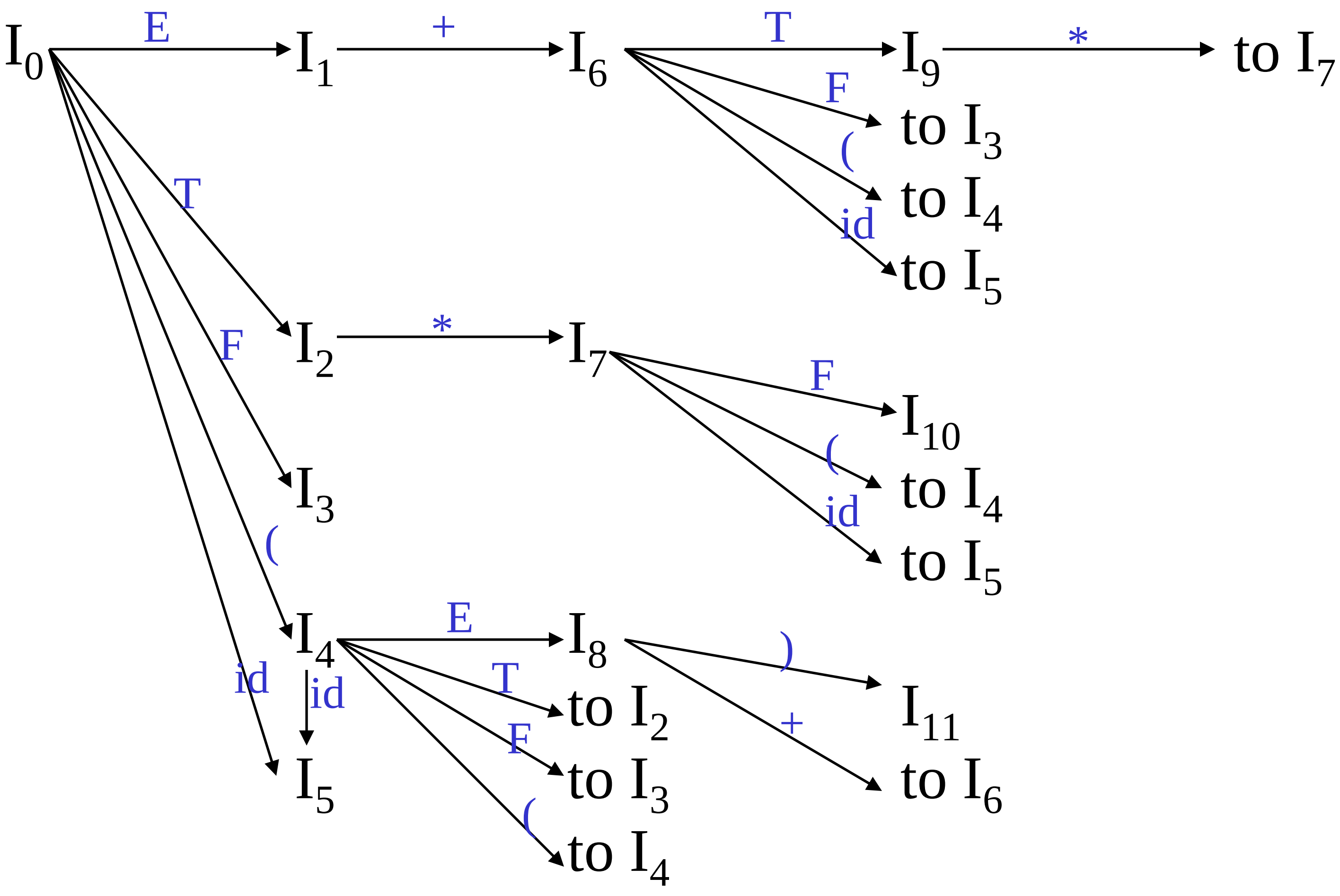
I_9: E → E+T.
T → T.*F

I_{10}: T → T*F.

I_{11}: F → (E).

Canonical collection of LR(0)

Transition Diagram (DFA) of Goto Function



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				

SLR(1)???

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.

LR(0) and SLR(1) Parsers

- SLR(1) Parse table= LR(0) parse table + **Follow** (for ex to find follow of T in the case of T->E. item)

Weaknesses of SLR(1)

- Using the only FOLLOW sets to drive reduce decisions throws away useful informations.
- Tokens in a FOLLOW set arise from different productions, but they get lumped together in the FOLLOW set with tokens arising from other productions.
- When it comes time to decide on a reduce move, we sometimes need to be more specific, and associate the look ahead with the particular production that added that token to the FOLLOW set.

A new way to build the parsing DFA

- To remember some of the context of new items added during the Closure operations
- For example in grammar below:

E -> E+(E)

E -> int

With the item E -> E+(.E)

- When we add the items E -> .E+(E) and E ->.int to the closure, we record the fact that the original E is followed by ‘)’, which creates the LR(1) items:

E -> .E+(E) ,)

E ->.int ,)

- The comma is just notation to separate the core of the item from lookahead

LR(1) Item

- An LR(1) item is a pair:

$X \text{ -> } \alpha.\beta$, a

Where $X \text{ -> } \alpha\beta$ is a production, **a** is a look ahead terminal and LR(1) means a look ahead of 1 terminal

- It also describes a context of the parser

➢ We are trying to find an X followed by an **a**

➢ We have (at least) α already at top of stack

➢ Thus we need to see next a prefix derivable from **βa**

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→α 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 $\{a_1,\ldots,a_n\} \subseteq \text{FOLLOW}(A)$

  ...

  $A \rightarrow \alpha\bullet,a_n$

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 $A \rightarrow \alpha \bullet B\beta, a$ 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

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

$$\ldots$$

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

can be written as

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

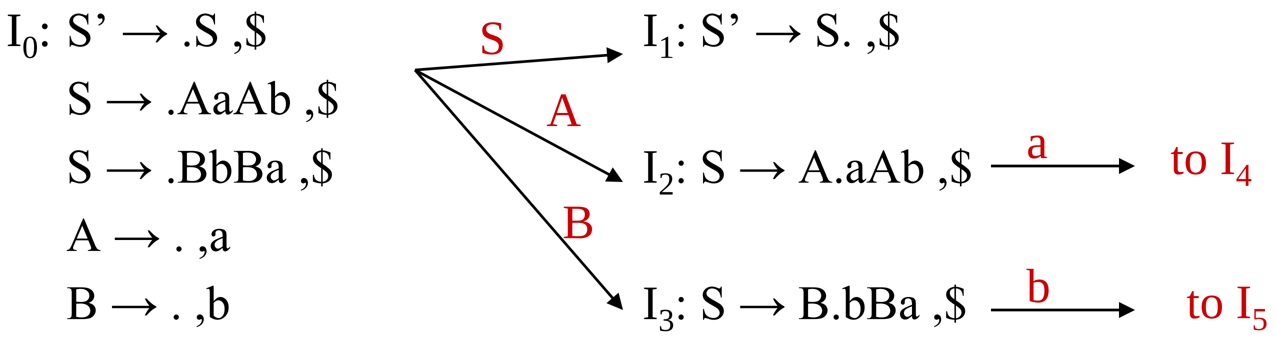
Canonical LR(1) Collection -- Example

S → AaAb

S → BbBa

A → ε

B → ε

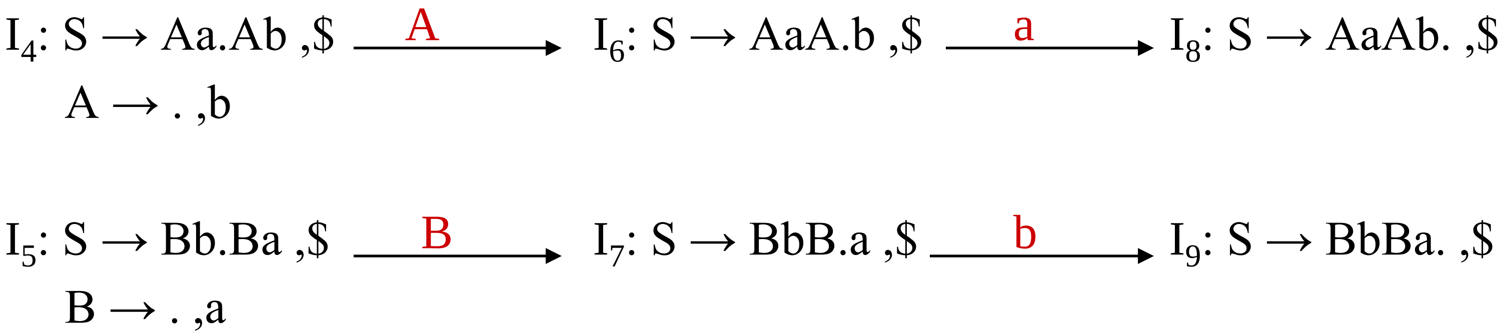


I_0: S’ → .S ,\$
S → .AaAb ,\$
S → .BbBa ,\$
A → . ,a
B → . ,b

I_1: S’ → S. ,\$

I_2: S → A.aAb ,\$

I_3: S → B.bBa ,\$



I_4: S → Aa.Ab ,\$
A → . ,b

I_6: S → AaA.b ,\$

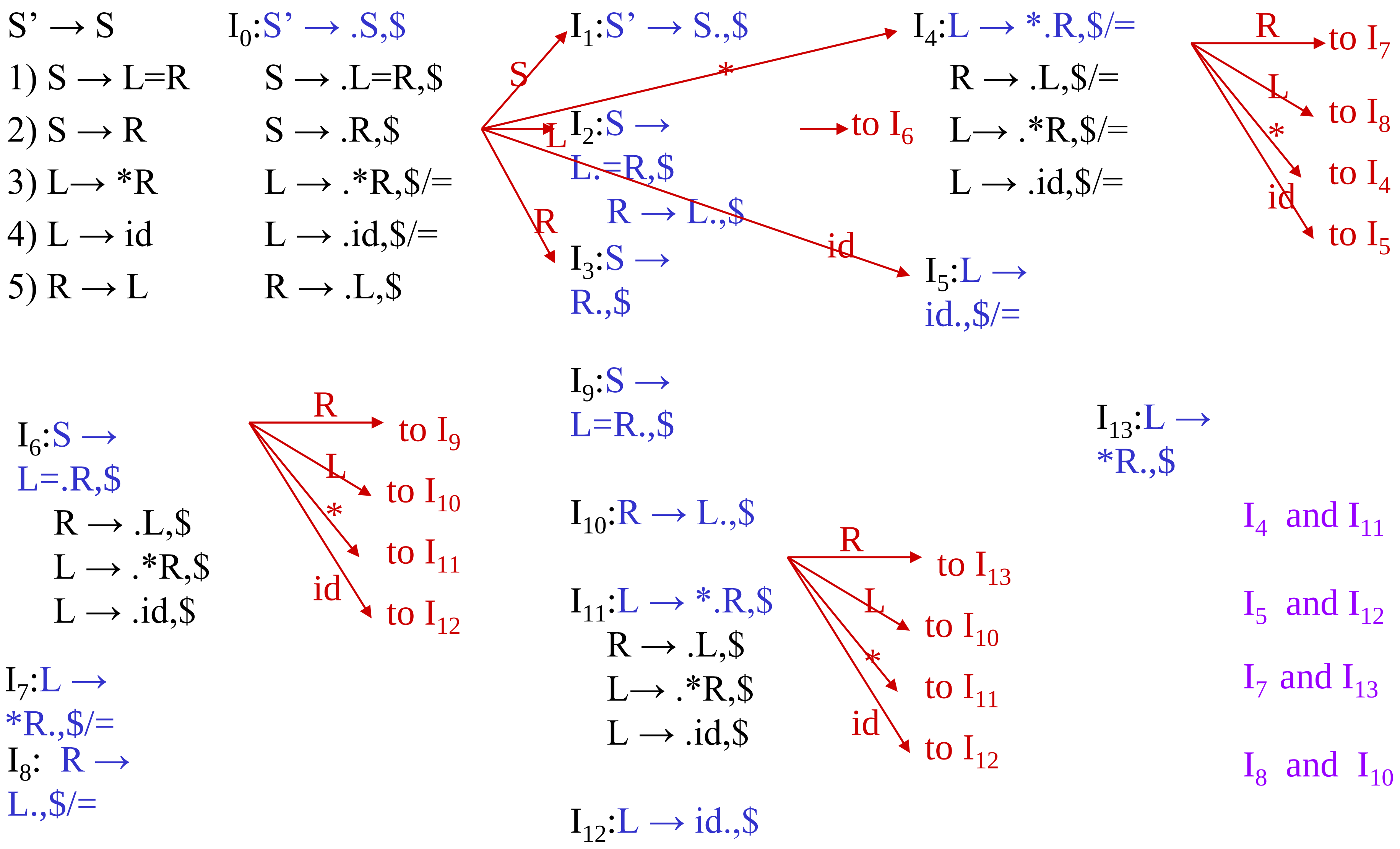
I_8: S → AaAb. ,\$

I_5: S → Bb.Ba ,\$
B → . ,a

I_7: S → BbB.a ,\$

I_9: S → BbBa. ,\$

Canonical LR(1) Collection – Example2



$S' \rightarrow S$

1) $S \rightarrow L=R$

2) $S \rightarrow R$

3) $L \rightarrow *R$

4) $L \rightarrow id$

5) $R \rightarrow L$

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

S

*

L

R

id

$I_1: S' \rightarrow S.,\$$

$I_2: S \rightarrow L.=R,\$$
$R \rightarrow L.,\$$

to I_6

$I_3: S \rightarrow R.,\$$

$I_4: L \rightarrow *.R,\$/=$
$R \rightarrow .L,\$/=$
$L \rightarrow .*R,\$/=$
$L \rightarrow .id,\$/=$

R

to I_7

L

to I_8

*

to I_4

id

to I_5

$I_5: L \rightarrow id.,\$/=$

$I_6: S \rightarrow L=.R,\$$
$R \rightarrow .L,\$$
$L \rightarrow .*R,\$$
$L \rightarrow .id,\$$

R

to I_9

L

to I_{10}

*

to I_{11}

id

to I_{12}

$I_7: L \rightarrow *R.,\$/=$

$I_8: R \rightarrow L.,\$/=$

$I_9: S \rightarrow L=R.,\$$

$I_{10}: R \rightarrow L.,\$$

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

R

to I_{13}

L

to I_{10}

*

to I_{11}

id

to I_{12}

$I_{12}: L \rightarrow id.,\$$

$I_{13}: L \rightarrow *R.,\$$

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(CLR(1))

1. Construct the canonical collection of sets of LR(1) items for G’.

   $C \leftarrow \{I_0,...,I_n\}$

1. Create the parsing action table as follows
   - If a is a terminal, $A \to \alpha \bullet a\beta, b$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j.***
   - If $A \to \alpha \bullet, a$ is in I_i , then action[i,a] is ***reduce A→α*** where $A \neq S’$.
   - If $S’ \to S \bullet, \$$ is in I_i , then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not LR(1).

1. Create the parsing goto table
   - for all non-terminals A, if $goto(I_i,A)=I_j$ then goto[i,A]=j

1. All entries not defined by (2) and (3) are errors.

1. Initial state of the parser contains $S’ \to .S,\$$

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(1) Grammars

- Problem with LR(1): too many states
- LALR(1) parsing (aka LookAhead LR)
  - Constructs LR(1) DFA and then merge any 2 LR(1) states whose items are identical except lookahead
  - Results in smaller parser tables, so are often used in practice
  - Theoretically less powerful than LR(1)
  - No of states in SLR = No of states in LALR

S □ id . , +
S □ E . , $

+

S □ id . , $
S □ E . , +

= ??

- LALR(1) grammar = a grammar whose LALR(1) parsing table has no conflicts

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 shrink 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.

  $C=\{I_0,...,I_n\}$ □ $C'=\{J_1,...,J_m\}$ 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

    □ 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)

Canonical LALR(1) Collection – Example2

$S' \rightarrow S$

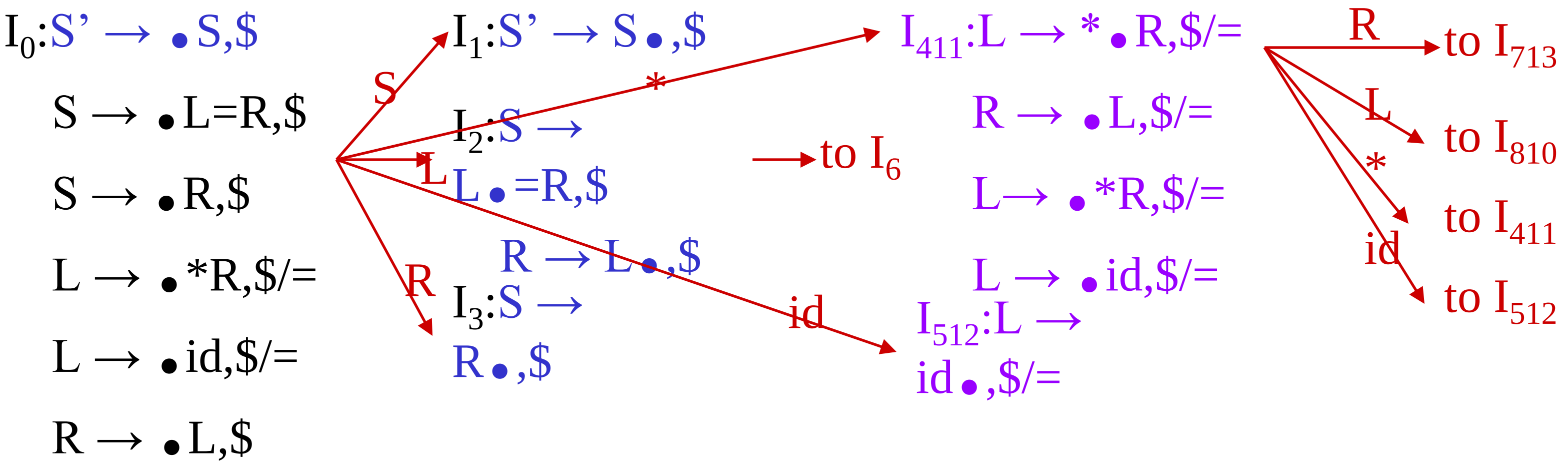
1) $S \rightarrow L=R$

2) $S \rightarrow R$

3) $L \rightarrow *R$

4) $L \rightarrow id$

5) $R \rightarrow L$



$I_0: S' \rightarrow \bullet S,\$$

$S \rightarrow \bullet L=R,\$$

$S \rightarrow \bullet R,\$$

$L \rightarrow \bullet *R,\$/=$

$L \rightarrow \bullet id,\$/=$

$R \rightarrow \bullet L,\$$

$I_1: S' \rightarrow S\bullet,\$$

$I_2: S \rightarrow L\bullet=R,\$$

$R \rightarrow L\bullet,\$$

$I_3: S \rightarrow R\bullet,\$$

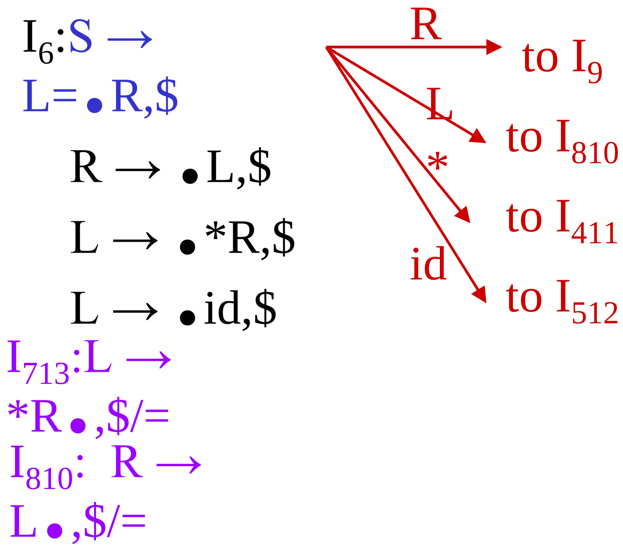
$I_{411}: L \rightarrow *\bullet R,\$/=$

$R \rightarrow \bullet L,\$/=$

$L \rightarrow \bullet *R,\$/=$

$L \rightarrow \bullet id,\$/=$

$I_{512}: L \rightarrow id\bullet,\$/=$



$I_6: S \rightarrow L=\bullet R,\$$

$R \rightarrow \bullet L,\$$

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

$L \rightarrow \bullet id,\$$

$I_{713}: L \rightarrow *R\bullet,\$/=$

$I_{810}: R \rightarrow L\bullet,\$/=$

$I_9: S \rightarrow L=R\bullet,\$$

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

⇓

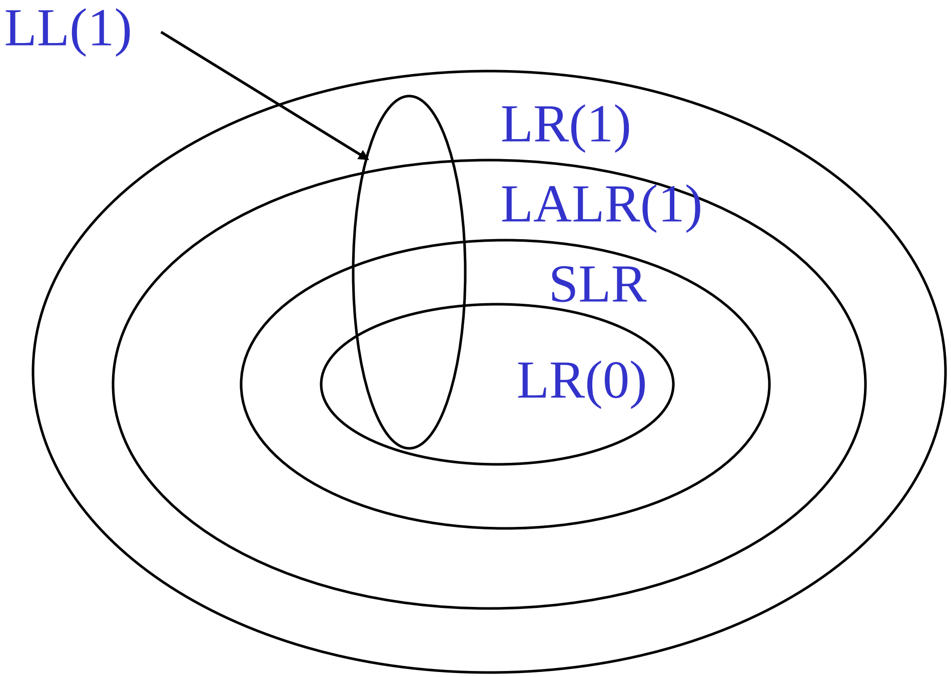
so, it is a LALR(1) grammar

Conflicts in LALR(1) parsing

- LALR(1) parsers cannot introduce shift/reduce conflicts
  - Such conflicts are caused when a lookahead is the same as a token on which we can shift. They depend on the core of the item. But we only merge states that had the same core to begin with. The only way for an LALR(1) parser to have a shift/reduce conflict is if one existed already in the LR(1) parser.
- LALR(1) parsers can introduce reduce/reduce conflicts
  - Here's a situation when this might happen:

$$\boxed{\begin{array}{l} A \to B\cdot, x \\ A \to C\cdot, y \end{array}} \text{ merges with } \boxed{\begin{array}{l} A \to B\cdot, y \\ A \to C\cdot, x \end{array}} \text{ to give: } \boxed{\begin{array}{l} A \to B\cdot, x/y \\ A \to C\cdot, x/y \end{array}}$$

Classification of Grammars



LR(k) ⊆ LR(k+1)
LL(k) ⊆ LL(k+0)

LL(k) ⊆ LR(k)
LR(0) ⊆ SLR
LALR(1) ⊆
LR(1)

Every LL(1) grammar is an LR(1) grammar, although there are LL(1) grammars that are not LALR(1)/SLR/LR(0)

Viable Prefixes

- Not all prefixes of right sentential forms can appear on the stack, however, since the parser must not shift past the handle. For example, suppose,
- E=>F*id=>(E)*id
- Then at various times during the parse, the stack will hold (, (E, and (E), but it must not hold (E)*, since (E) is a handle, which the parser must reduce to F before shifting *
- Def:

1) The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser
2) The prefixes of rsf that does not continue past the right end of the rightmost handle of that sentential form

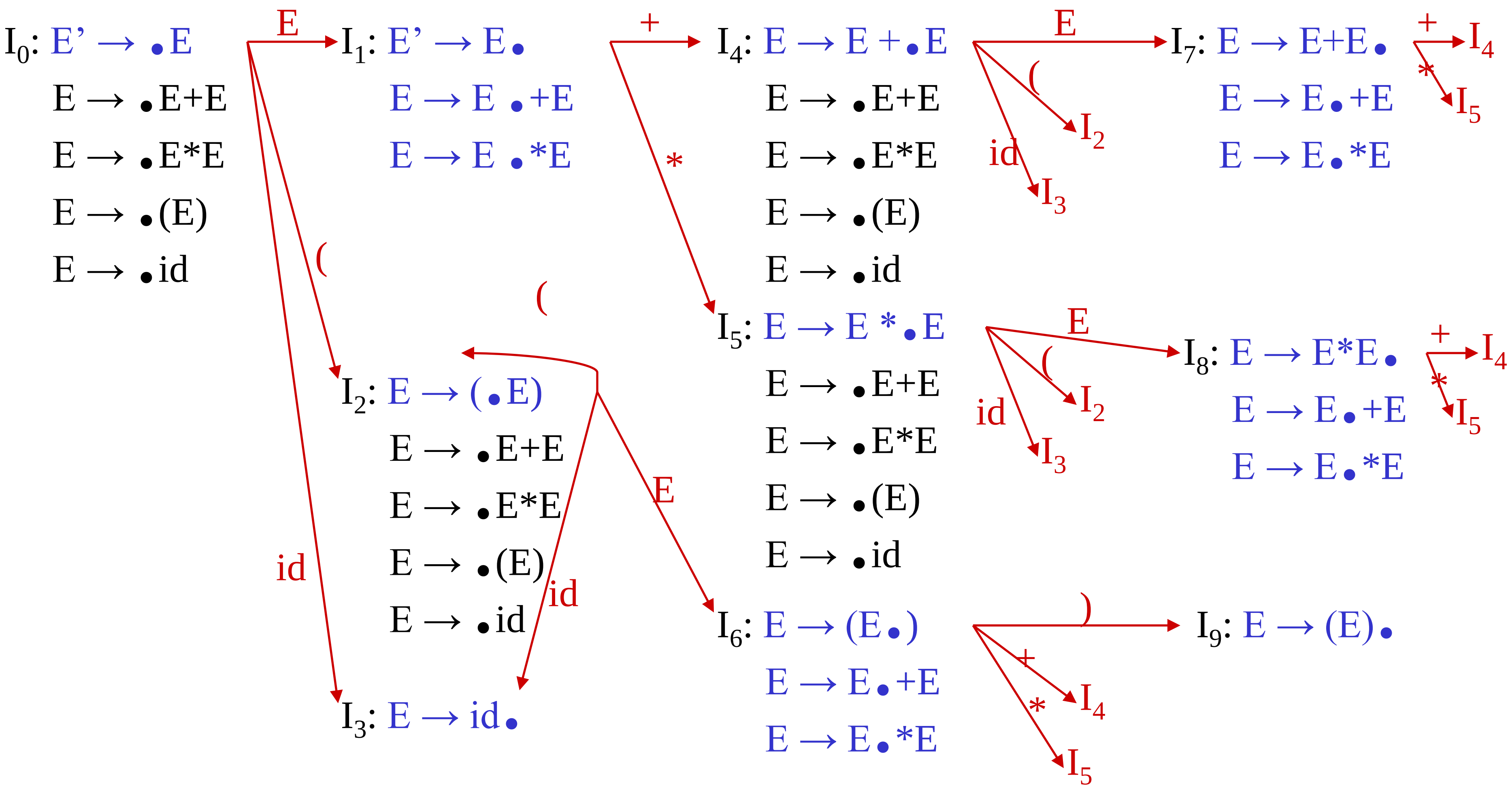
Using Ambiguous Grammars

- 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 → E+E | E*E | (E) | id □

E → E+T | T
T → T*F | F
F → (E) | id

Sets of LR(0) Items for Ambiguous Grammar



I_0: $E' \to \bullet E$
$E \to \bullet E+E$
$E \to \bullet E*E$
$E \to \bullet (E)$
$E \to \bullet id$

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

I_2: $E \to (\bullet E)$
$E \to \bullet E+E$
$E \to \bullet E*E$
$E \to \bullet (E)$
$E \to \bullet id$

I_3: $E \to id \bullet$

I_4: $E \to E + \bullet E$
$E \to \bullet E+E$
$E \to \bullet E*E$
$E \to \bullet (E)$
$E \to \bullet id$

I_5: $E \to E * \bullet E$
$E \to \bullet E+E$
$E \to \bullet E*E$
$E \to \bullet (E)$
$E \to \bullet id$

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

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

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

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

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		

Error detection in LR parsing

- Errors are discovered when a slot in the action table is blank.
- Canonical LR(1) parsers detect and report the error as soon as it is encountered
- LALR(1) parsers may perform a few reductions after the error has been encountered, and then detect it.
  - This is a result to the state merging.

Error recovery in LR parsing

- Phrase-level recovery
  - Associate error routines with the empty table slots. Figure out what situation may have cause the error and make an appropriate recovery.
- Panic-mode recovery
  - Discard symbols from the stack until a non-terminal is found. Discard input symbols until a possible lookahead for that non-terminal is found. Try to continue parsing.
- Error productions
  - Common errors can be added as rules to the grammar.

Error recovery in LR parsing

- Phrase-level recovery
  - Consider the table for grammar E→E+E | id

	+	id	$	E
0	e1	s2	e1	1
1	s3	e2	accept	
2	r(E→id)	e3	r(E→id)	
3	e1	s2	e1	4
4	r(E→E+E)	e2	r(E→E+E)	

Error e1: "missing operand inserted". Recover by inserting an imaginary identifier in the stack and shifting to state 2.

Error e2: "missing operator inserted". Recover by inserting an imaginary operator in the stack and shifting to state 3

Error e3: ??

Error recovery in LR parsing

- Error productions
  - Allow the parser to "recognize" erroneous input.
  - Example:
    - statement : expression SEMI

      ```
                  | error SEMI
                  | error NEWLINE
      ```

      - If the expression cannot be matched, discard everything up to the next semicolon or the next new line

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 <· b   b has higher precedence than a

  a =· b   b has same precedence as a

  a ·> 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,

  <· with marking the left end,

  =· appearing in the interior of the handle, and

  ·> marking the right hand.

- In 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 ·> is encountered.
2. Then scan backwards (to the left) until <· is encountered.
3. The handle contains everything to left of the first ·> and to the right of the <· 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 $

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

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 ( $a \lessdot b$ or $a \doteq b$ ) then {                 /* SHIFT */
       push b onto the stack;
       advance p to the next input symbol;
    }
    else if ( $a \gtrdot b$ ) then                  /* REDUCE */
       repeat pop stack
       until ( the top of stack terminal is related by $\lessdot$ to the terminal most
      recently popped );
    else error();
  }
```

Operator-Precedence Parsing Algorithm -- Example

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

stack	***input***	***action***	
$	id+id*id$	$ <· id	shift
$id	+id*id$	id ·> +	reduce E → id
$	+id*id$	shift	
$+	id*id$	shift	
$+id	*id$	id ·> *	reduce E → id
$+	*id$	shift	
$+*	id$	shift	
$+*id	$	id ·> $	reduce E → id
$+*	$	* ·> $	reduce E → E*E
$+	$	+ ·> $	reduce E → E+E
$	$	accept	

How to Create Operator-Precedence Relations

- We use associativity and precedence relations among operators.

1. If operator θ_1 has higher precedence than operator θ_2,
   □ $\theta_1 \gtrdot \theta_2$ and $\theta_2 \lessdot \theta_1$

1. If operator θ_1 and operator θ_2 have equal precedence,
   they are left-associative □ $\theta_1 \gtrdot \theta_2$ and $\theta_2 \gtrdot \theta_1$
   they are right-associative □ $\theta_1 \lessdot \theta_2$ and $\theta_2 \lessdot \theta_1$

1. For all operators θ, $\theta \lessdot \mathrm{id}$, $\mathrm{id} \gtrdot \theta$, $\theta \lessdot ($, $(\lessdot \theta$, $\theta \gtrdot)$, $) \gtrdot \theta$, $\theta \gtrdot \$$, and $\$ \lessdot \theta$

1. Also, let

$(\doteq)$	$\$ \lessdot ($	$\mathrm{id} \gtrdot)$	$) \gtrdot \$$
$(\lessdot ($	$\$ \lessdot \mathrm{id}$	$\mathrm{id} \gtrdot \$$	$) \gtrdot)$
$(\lessdot \mathrm{id}$			

Operator-Precedence Relations

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

Handling Unary Minus

- Operator-Precedence parsing cannot handle the unary minus when we also have 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 tokens 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

$\theta \lessdot$ unary-minus for any operator

unary-minus $\gtrdot \theta$ if unary-minus has higher precedence than θ

unary-minus $\lessdot \theta$ if unary-minus has lower (or equal) precedence than θ

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 $a \cdot> b$

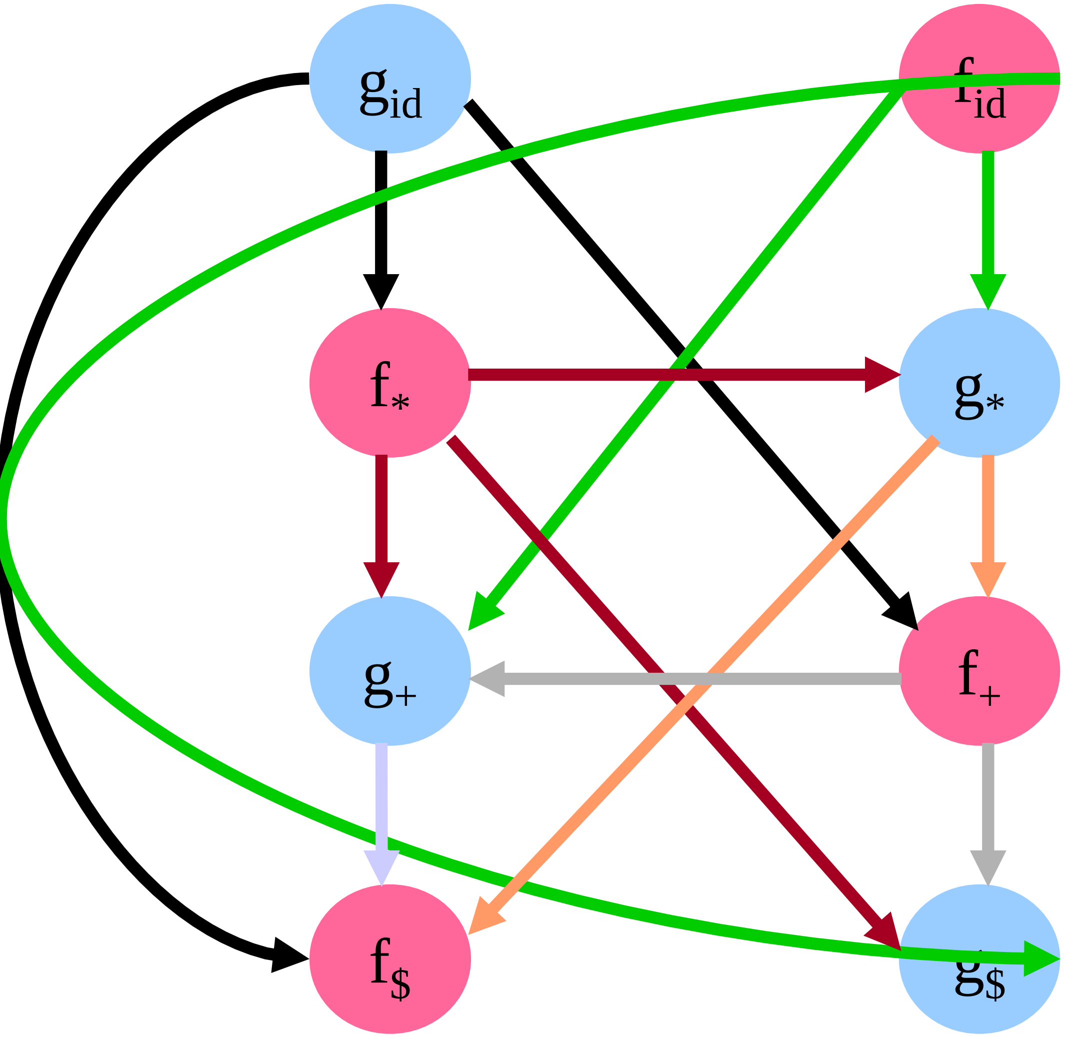
Algorithm Constructing precedence functions

Constructing precedence functions

Method:

1. Create symbols f_a and g_a for each **a** that is a terminal or $.
2. Partition the created symbols into as many groups as possible, in such a way that if a =. b, then f_a and g_b are in the same group.
3. Create a directed graph whose nodes are the groups found in (2). For any a and b, if a <.b , place an edge from the group of g_b to the group of f_a. Of a .> b, place an edge from the group of f_a to that of $g_{b.}$
4. If the graph constructed in (3) has a cycle, then no precedence functions exist. If there are no cycle, let f(a) be the length of the longest path beginning at the group of f_a; let g(a) be the length of the longest path beginning at the group of $g_{a.}$

Example



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

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

There are no = relationships, so each symbol is in a group by itself

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:

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

Handling Shift/Reduce Errors

When consulting the precedence matrix to decide whether to shift or reduce, we may find that no relation holds between the top stack and the first input symbol.

To recover, we must modify (insert/change)

1. Stack or
2. Input or
3. Both.

We must be careful that we don’t get into an infinite loop.

Example

	id	**(**	**)**	**$**
id	**e3**	**e3**	·>	·>
(	<·	<·	=.	**e4**
)	**e3**	**e3**	·>	·>
$	<·	<·	e2	e1

e1: Called when : issue diagnost whole expression is missing

insert **id** onto the input

ic: ‘missing operand’

e2: Called when : expression begins with a right parenthesis

delete) from the input

issue diagnostic: ‘unbalanced right parenthesis’

Example

	id	(	)	$
id	e3	e3	·>	·>
(	<·	<·	=.	e4
)	e3	e3	·>	·>
$	<·	<·	e2	e1

e3: Called when : id or) is followed by id or (

insert + onto the input

issue diagnostic: ‘missing operator’

e4: Called when : expression ends with a left parenthesis

pop (from the stack

issue diagnostic: ‘missing right parenthesis’