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)

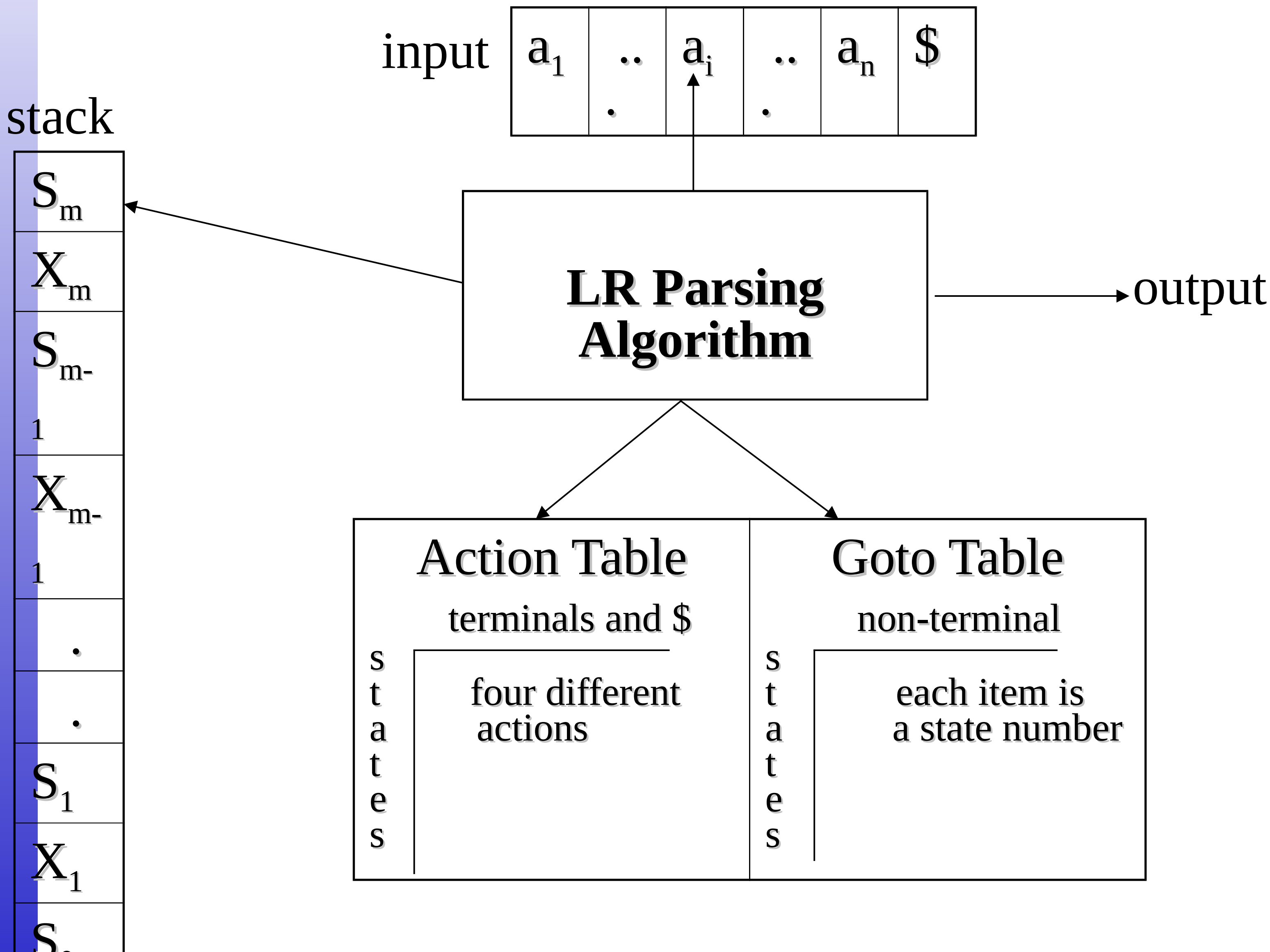
- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

    LL(1)-Grammars $\subset$ 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:

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

  Stack   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 ... X_m S_m, a_i a_{i+1} ... a_n \$) ➔ (S_o X_1 S_1 ... X_m S_m a_i s, a_{i+1} ... a_n \$)

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

   (S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n \$) ➔ (S_o X_1 S_1 ... X_{m-r} S_{m-r} A s, a_i ... 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 $2*|\boldsymbol{\beta}|$ (=r) items from the stack; let us assume that $\boldsymbol{\beta} = Y_1 Y_2 \ldots Y_r$
- 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 $S' \rightarrow S$ 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

$E' \rightarrow E$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

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

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

$E \rightarrow \bullet E+T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet T*F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet id \quad \}$

GOTO function

- **Definition.**
  Goto(I, X) = closure of the set of all items
  $A \rightarrow \alpha X.\beta$ where $A \rightarrow \alpha.X\beta$ belongs to I
- Intuitively: Goto(I, X) set of all items that "reachable" from the items of I once X has been "seen."
- E.g. consider I={**E'→ E.** , **E→ E.+T**} and compute Goto(I, +)

  Goto(I, +) = { **E→ E+.T**, **T → .T * F** , **T → .F** , **F → .(E)** , **F → .id** }

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: $E' \to .E$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_1: $E' \to E.$
$E \to E.+T$

I_2: $E \to T.$
$T \to T.*F$

I_3: $T \to F.$

I_4: $F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_5: $F \to id.$

I_6: $E \to E+.T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_7: $T \to T*.F$
$F \to .(E)$
$F \to .id$

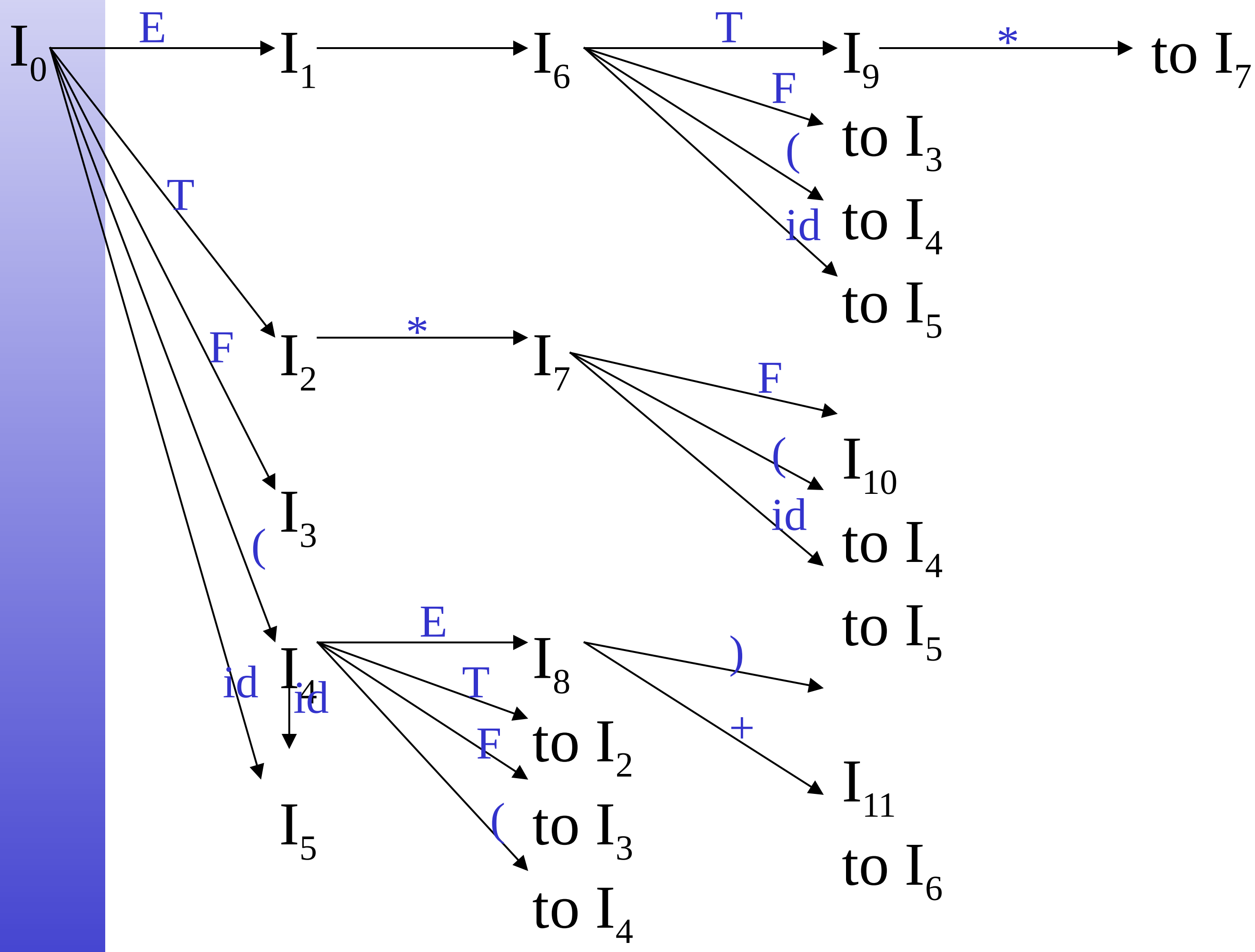
I_8: $F \to (E.)$
$E \to E.+T$

I_9: $E \to E+T.$
$T \to T.*F$

I_{10}: $T \to T*F.$

I_{11}: $F \to (E).$

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

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

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

$S \to L=R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

I_0: $S' \to .S$
$S \to .L=R$
$S \to .R$
$L \to .*R$
$L \to .id$
$R \to .L$

I_1: $S' \to S.$

I_2: $S \to L.=R$
$R \to L.$

I_3: $S \to R.$

I_4: $L \to *.R$
$R \to .L$
$L \to .*R$
$L \to .id$

I_5: $L \to id.$

I_6: $S \to L=.R$
$R \to .L$
$L \to .*R$
$L \to .id$

I_7: $L \to *R.$

I_8: $R \to L.$

I_9: $S \to L=R.$

Problem
FOLLOW(R)={=,$}
= shift 6
reduce by $R \to L$
shift/reduce conflict

Conflict Example2

$S \rightarrow AaAb$
$S \rightarrow BbBa$
$A \rightarrow \varepsilon$
$B \rightarrow \varepsilon$

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

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

a → reduce by $A \rightarrow \varepsilon$
a → reduce by $B \rightarrow \varepsilon$
reduce/reduce conflict

b → reduce by $A \rightarrow \varepsilon$
b → reduce by $B \rightarrow \varepsilon$
reduce/reduce conflict