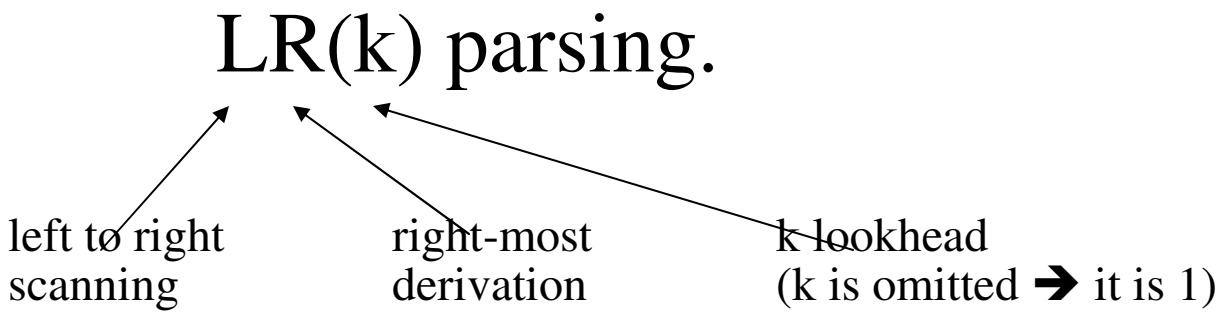
- The most powerful shift-reduce parsing (yet efficient) is:



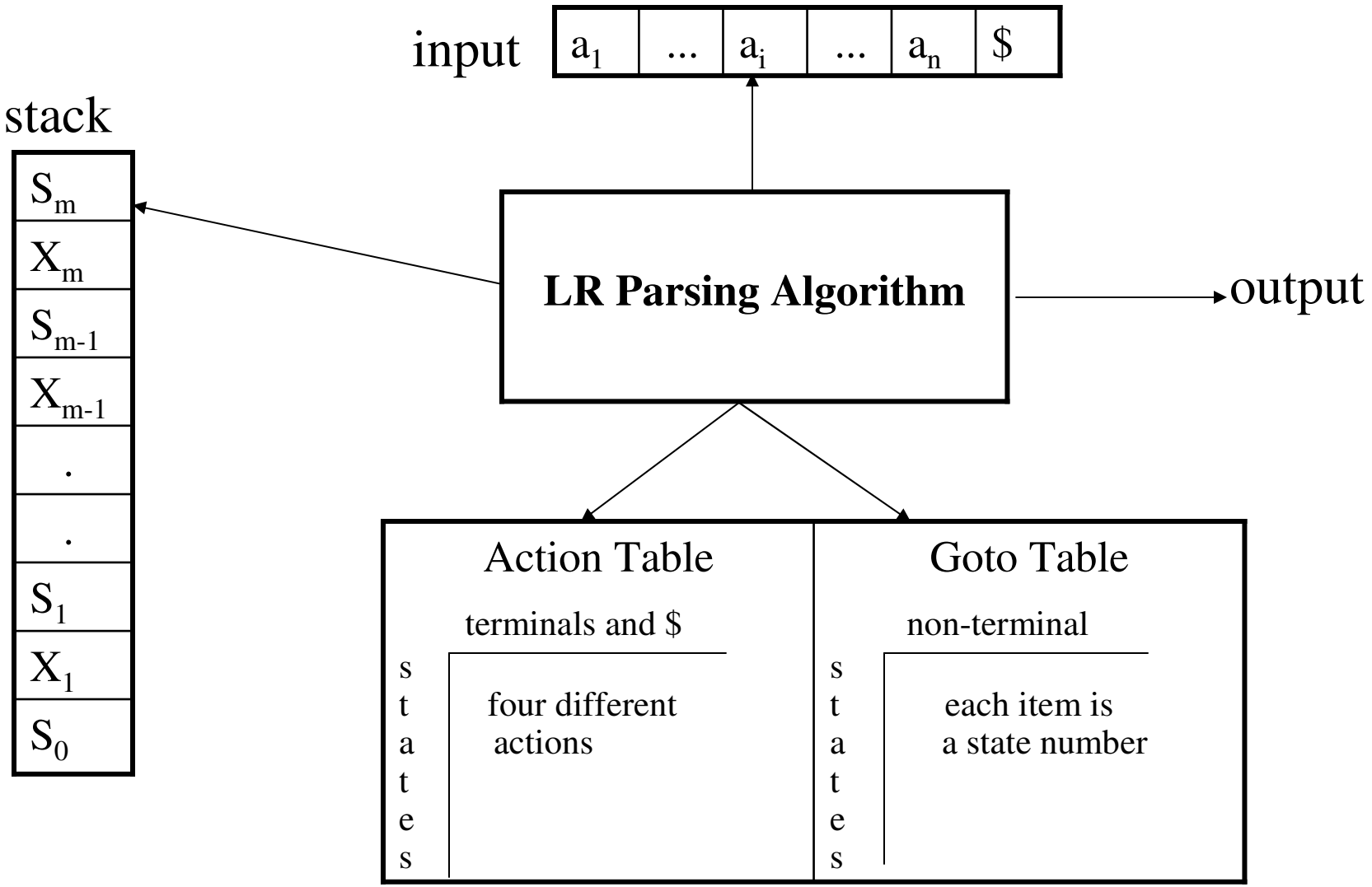
- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

    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

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

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

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

   - Output is the reducing production reduce $A \rightarrow \beta$

3. **Accept** – Parsing successfully completed

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

Reduce Action

- pop 2|β| (=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]**

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

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

- 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) $E \rightarrow E+T$
2) $E \rightarrow T$
3) $T \rightarrow T*F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow id$

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				

LR Parsing Algroithm

Set ip to point to the first symbol of w$;

Repeat forever begin

let s be the state on the top of the stack and a be the symbol pointed to by ip;

if action[s,a]=shift s' then begin

i. push a then s' on the top of the stack

ii. advance ip to the next input symbol

end

else if action[s,a]=reduce A->β then begin

i. pop 2*| β| symbols on the top of the stack

ii. let s' be the state now on the top of the stack

iii. Push A then goto[s',A] on the top of the stack

iv. Output the production A->β

end

else if action[s,a]=accept then

return

else

error()

end

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	