Syntax Analysis:

Context-free Grammars, Pushdown Automata and Parsing
Part - 4

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is syntax analysis? (covered in lecture 1)
- Specification of programming languages: context-free grammars (covered in lecture 1)
- Parsing context-free languages: push-down automata (covered in lectures 1 and 2)
- Top-down parsing: LL(1) parsing (covered in lectures 2 and 3)
- Recursive-descent parsing
- Bottom-up parsing: LR-parsing

Elimination of Left Recursion

- A *left-recursive* grammar has a non-terminal A such that $A \Rightarrow^+ A\alpha$
- Top-down parsing methods (LL(1) and RD) cannot handle reft-recursive grammars
- Left-recursion in grammars can be eliminated by transformations
- A simpler case is that of grammars with *immediate left recursion*, where there is a production of the form $A \rightarrow A\alpha$
  - Two productions $A \rightarrow A\alpha \mid \beta$ can be transformed to $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \epsilon$
  - In general, a group of productions:
    $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_n$
    can be transformed to
    $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_n A'$, $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid ... \mid \alpha_m A' \mid \epsilon$

Left Recursion Elimination - An Example

$A \to A\alpha \mid \beta \Rightarrow A \to \beta A', A' \to \alpha A' \mid \epsilon$

- The following grammar for regular expressions is ambiguous:
  $E \to E + E \mid E\ E \mid E* \mid (E) \mid a \mid b$
- Equivalent left-recursive but unambiguous grammar is:
  $E \to E + T \mid T, T \to T\ F \mid F, F \to F* \mid P, P \to (E) \mid a \mid b$
- Equivalent non-left-recursive grammar is:
  $E \to T\ E', E' \to +T\ E' \mid \epsilon, T \to F\ T', T' \to F\ T' \mid \epsilon,$
  $F \to P\ F', F' \to *F' \mid \epsilon, P \to (E) \mid a \mid b$

Left Factoring

- If two alternatives of a production begin with the same string, then the grammar is not LL(1)
- Example: $S \rightarrow 0S1 \mid 01$ *is not LL(1)*
  - After left factoring: $S \rightarrow 0S'$, $S' \rightarrow S1 \mid 1$ *is LL(1)*
- General method: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \Rightarrow A \rightarrow \alpha A'$, $A' \rightarrow \beta_1 \mid \beta_2$
- Another example: a grammar for logical expressions is given below
  $E \rightarrow T$ *or* $E \mid T$, $T \rightarrow F$ *and* $T \mid F$,
  $F \rightarrow$ *not* $F \mid (E) \mid$ *true* $\mid$ *false*
  - This grammar is not LL(1) but becomes LL(1) after left factoring
  - $E \rightarrow TE'$, $E' \rightarrow$ *or* $E \mid \epsilon$, $T \rightarrow FT'$, $T' \rightarrow$ *and* $T \mid \epsilon$,
    $F \rightarrow$ *not* $F \mid (E) \mid$ *true* $\mid$ *false*

Grammar Transformations may not help!

Original Grammar

S' → S$
S → if id S |
if id S else S |
a

LL(1) Parsing Table for modified grammar

	if	else	a	$
S'	S' → S$		S' → S$	
S	S → if id S S1		S → a	
S1		S1 → ε S1 → else S		S1 → ε

dirsymb(S$) = {if, a}; dirsymb (a) = {a}
dirsymb(if id S S1) = {if}
dirsymb(else S) = {else}
dirsymb(ε) = {else, $}

Grammar is not LL(1)

Left-Factored Grammar

S' → S$
S → if id S S1 | a
S1 → ε | else S

tokens: if, id, else, a

$$dirsymb(if\ id\ S\ S1) \bigcap dirsymb(a) = \varnothing$$
$$dirsymb(\varepsilon) \bigcap dirsymb(else\ S) \neq \varnothing$$

Choose $S1 \rightarrow$ *else S* instead of $S1 \rightarrow \epsilon$ on lookahead *else*. This resolves the conflict. Associates *else* with the innermost *if*

Recursive-Descent Parsing

- Top-down parsing strategy
- One function/procedure for each nonterminal
- Functions call each other recursively, based on the grammar
- Recursion stack handles the tasks of LL(1) parser stack
- LL(1) conditions to be satisfied for the grammar
- Can be automatically generated from the grammar
- Hand-coding is also easy
- Error recovery is superior

An Example

Grammar: $S' \rightarrow S\$,\ S \rightarrow aAS \mid c,\ A \rightarrow ba \mid SB,\ B \rightarrow bA \mid S$

```
/* function for nonterminal S' */
void main(){/* S' --> S$ */
   fS(); if (token == eof) accept();
         else error();
}
/* function for nonterminal S */
void fS(){/* S --> aAS | c */
   switch token {
     case a : get_token(); fA(); fS();
              break;
     case c : get_token(); break;
     others : error();
   }
}
```

An Example (contd.)

```
void fA(){/* A --> ba | SB */
    switch token {
      case b : get_token();
               if (token == a) get_token();
               else error(); break;
      case a,c : fS(); fB(); break;
      others : error();
    }
}
void fB(){/* B --> bA | S */
    switch token {
      case b : get_token(); fA(); break;
      case a,c : fS(); break;
      others : error();
    }
}
```

Automatic Generation of RD Parsers

- Scheme is based on structure of productions
- Grammar must satisfy LL(1) conditions
- function *get_token*() obtains the next token from the lexical analyzer and places it in the global variable *token*
- function *error*() prints out a suitable error message
- In the next slide, for each grammar component, the code that must be generated is shown

Automatic Generation of RD Parsers (contd.)

1. ϵ : ;
2. $a \in T$: if ($token == a$) $get\_token()$; else $error()$;
3. $A \in N$: $fA()$; /* function call for nonterminal A */
4. $\alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$:

```
switch token {
  case dirsym(α1): program_segment(α1); break;
  case dirsym(α2): program_segment(α2); break;
  ...
  others: error();
}
```

5. $\alpha_1\alpha_2 \ ... \ \alpha_n$:
   program_segment(α_1); program_segment(α_2); ... ;
   program_segment(α_n);
6. $A \rightarrow \alpha$: void $fA()$ { program_segment(α); }

Bottom-Up Parsing

- Begin at the leaves, build the parse tree in small segments, combine the small trees to make bigger trees, until the root is reached
- This process is called *reduction* of the sentence to the start symbol of the grammar
- One of the ways of "reducing" a sentence is to follow the rightmost derivation of the sentence in *reverse*
  - *Shift-Reduce* parsing implements such a strategy
  - It uses the concept of a *handle* to detect when to perform reductions

Shift-Reduce Parsing

- **Handle**: A *handle* of a right sentential form γ, is a production $A \rightarrow \beta$ *and* a position in 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
  That is, if $S \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta w$, then $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta w$
- A handle will always eventually appear on the top of the stack, never submerged inside the stack
- In S-R parsing, we locate the handle and reduce it by the LHS of the production repeatedly, to reach the start symbol
- These reductions, in fact, trace out a rightmost derivation of the sentence in reverse. This process is called handle pruning
- *LR-Parsing* is a method of shift-reduce parsing

Examples

1. $S \rightarrow aAcBe$, $A \rightarrow Ab \mid b$, $B \rightarrow d$
   For the string = *abbcde*, the rightmost derivation marked with handles is shown below

$$
\begin{aligned}
S &\Rightarrow \underline{aAcBe} \quad (aAcBe,\ S \rightarrow aAcBe) \\
&\Rightarrow aAc\underline{d}e \quad (d,\ B \rightarrow d) \\
&\Rightarrow a\underline{Ab}cde \quad (Ab,\ A \rightarrow Ab) \\
&\Rightarrow a\underline{b}bcde \quad (b,\ A \rightarrow b)
\end{aligned}
$$

The handle is unique if the grammar is unambiguous!

Examples (contd.)

2. $S \rightarrow aAS \mid c,\ A \rightarrow ba \mid SB,\ B \rightarrow bA \mid S$
   For the string = *acbbac*, the rightmost derivation marked with handles is shown below

$$
\begin{aligned}
S &\Rightarrow \underline{aAS} \ (aAS,\ S \rightarrow aAS) \\
&\Rightarrow aA\underline{c} \ (c,\ S \rightarrow c) \\
&\Rightarrow a\underline{SB}c \ (SB,\ A \rightarrow SB) \\
&\Rightarrow aS\underline{bA}c \ (bA,\ B \rightarrow bA) \\
&\Rightarrow aSb\underline{ba}c \ (ba,\ A \rightarrow ba) \\
&\Rightarrow a\underline{c}bbac \ (c,\ S \rightarrow c)
\end{aligned}
$$

Examples (contd.)

3. $E \rightarrow E + E$, $E \rightarrow E * E$, $E \rightarrow (E)$, $E \rightarrow id$
   For the string = $id + id * id$, two rightmost derivation marked with handles are shown below

$$
\begin{aligned}
E &\Rightarrow \underline{E + E} \quad (E + E,\ E \rightarrow E + E) \\
&\Rightarrow E + \underline{E * E} \quad (E * E,\ E \rightarrow E * E) \\
&\Rightarrow E + E * \underline{id} \quad (id,\ E \rightarrow id) \\
&\Rightarrow E + \underline{id} * id \quad (id,\ E \rightarrow id) \\
&\Rightarrow \underline{id} + id * id \quad (id,\ E \rightarrow id) \\
E &\Rightarrow \underline{E * E} \quad (E * E,\ E \rightarrow E * E) \\
&\Rightarrow E * \underline{id} \quad (id,\ E \rightarrow id) \\
&\Rightarrow \underline{E + E} * id \quad (E + E,\ E \rightarrow E + E) \\
&\Rightarrow E + \underline{id} * id \quad (id,\ E \rightarrow id) \\
&\Rightarrow \underline{id} + id * id \quad (id,\ E \rightarrow id)
\end{aligned}
$$

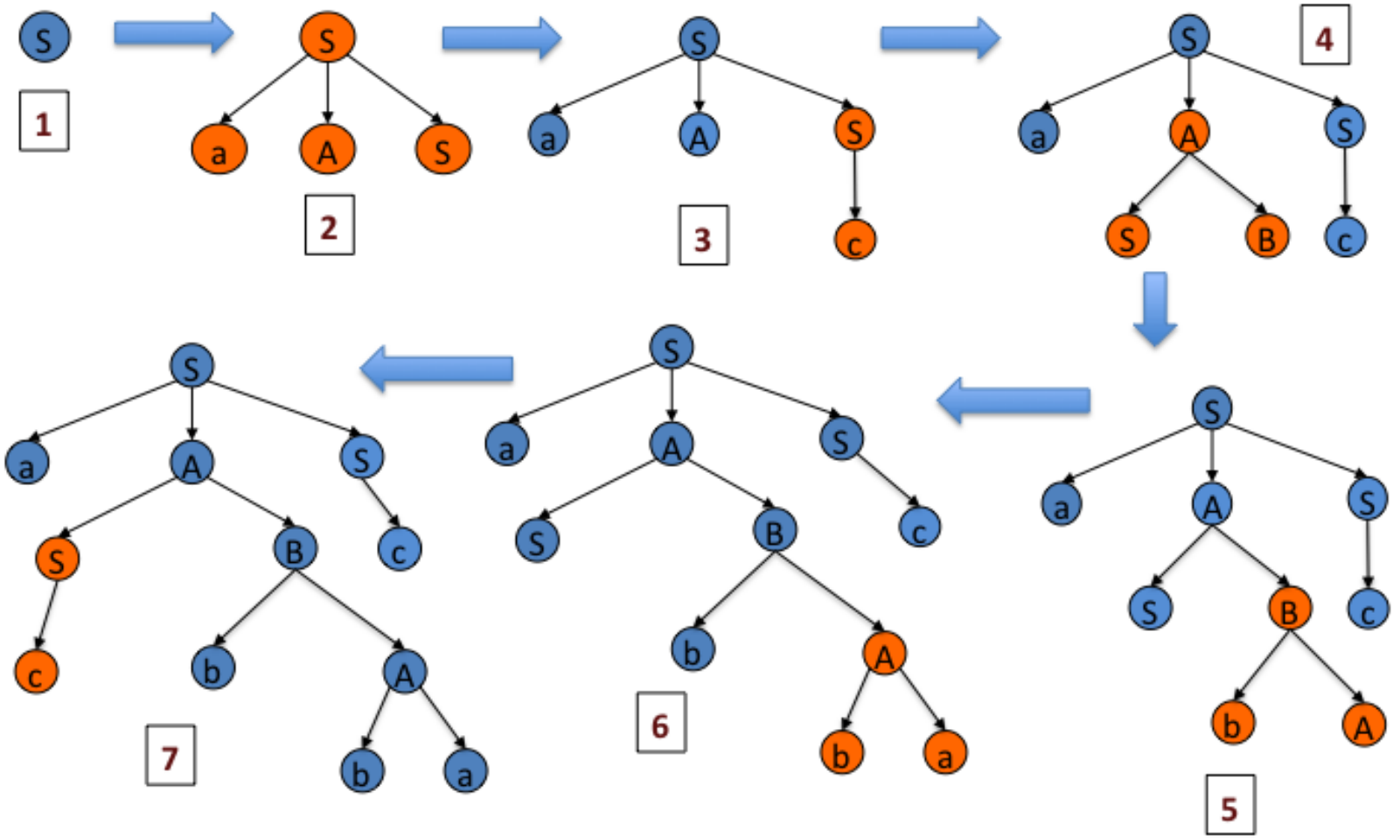
Rightmost Derivation and Bottom-UP Parsing

S → aAS | c
A → ba | SB
B → bA | S

Rightmost derivation of the string *acbbac*

S => aA**S** => a**A**c => aS**B**c => aS**b**Ac => a**S**bbac => acbbac

1 2 3 4 5 6 7



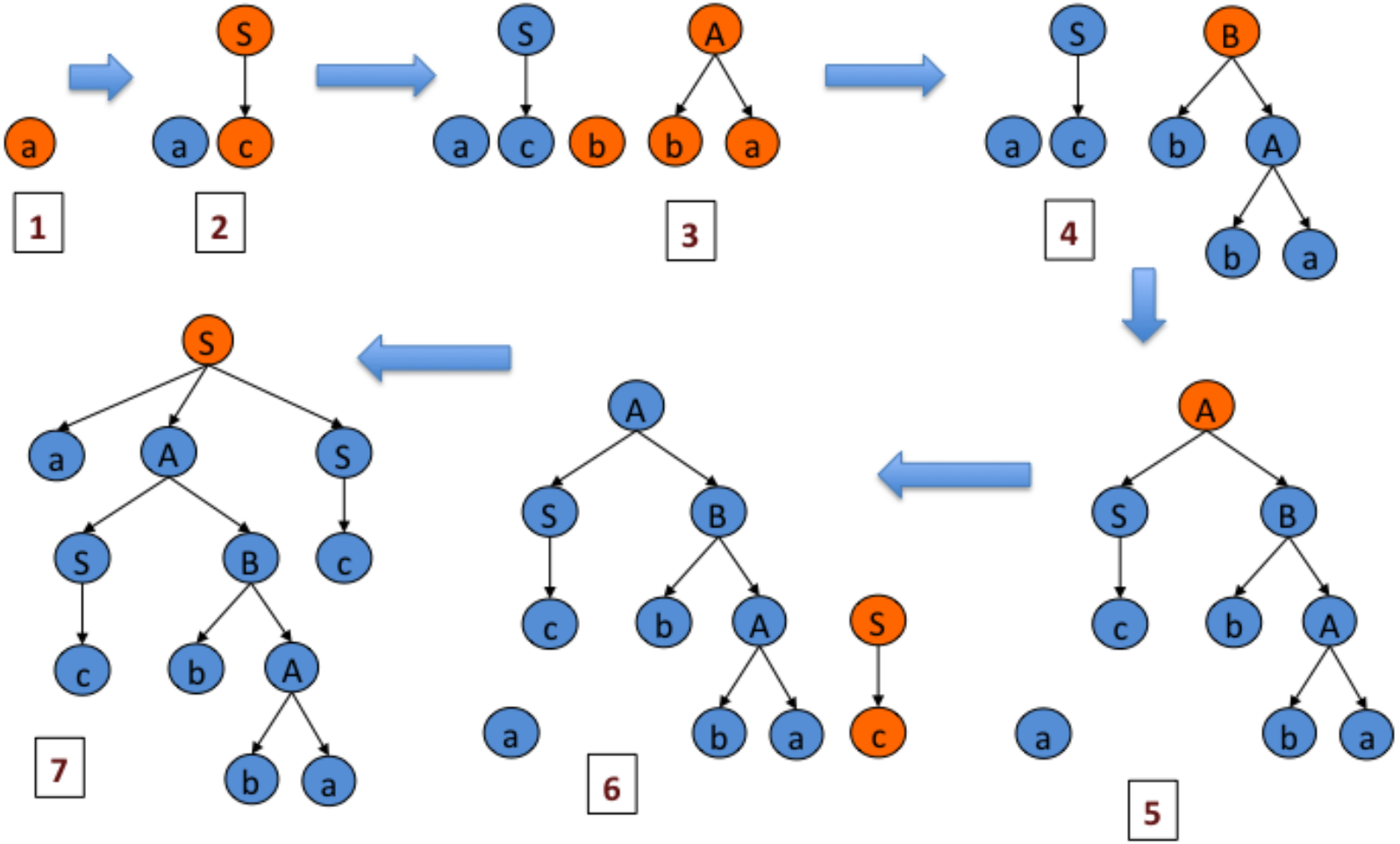
Rightmost Derivation and Bottom-UP Parsing (contd.)

S → aAS | c
A → ba | SB
B → bA | S

Rightmost derivation of the string *acbbac* in reverse

S <= a**A**S <= a**A**c <= a**S**B**B**c <= a**S**b**A**c <= a**S**bbac <= acbbac

7 6 5 4 3 2 1



Shift-Reduce Parsing Algorithm

- How do we locate a handle in a right sentential form?
  - An LR parser uses a DFA to detect the condition that a handle is now on the stack
- Which production to use, in case there is more than one with the same RHS?
  - An LR parser uses a parsing table similar to an LL parsing table, to choose the production
- A stack is used to implement an S-R parser, The parser has four actions
  1. **shift**: the next input symbol is shifted to the top of stack
  2. **reduce**: the right end of the handle is the top of stack; locates the left end of the handle inside the stack and replaces the handle by the LHS of an appropriate production
  3. **accept**: announces successful completion of parsing
  4. **error**: syntax error, error recovery routine is called

S-R Parsing Example 1

$ marks the bottom of stack and the right end of the input

Stack	**Input**	**Action**
$	*acbbac*$	shift
$ *a*	*cbbac*$	shift
$ *ac*	*bbac*$	reduce by $S \rightarrow c$
$ *aS*	*bbac*$	shift
$ *aSb*	*bac*$	shift
$ *aSbb*	*ac*$	shift
$ *aSbba*	*c*$	reduce by $A \rightarrow ba$
$ *aSbA*	*c*$	reduce by $B \rightarrow bA$
$ *aSB*	*c*$	reduce by $A \rightarrow SB$
$ *aA*	*c*$	shift
$ *aAc*	$	reduce by $S \rightarrow c$
$ *aAS*	$	reduce by $S \rightarrow aAS$
$ *S*	$	accept

S-R Parsing Example 2

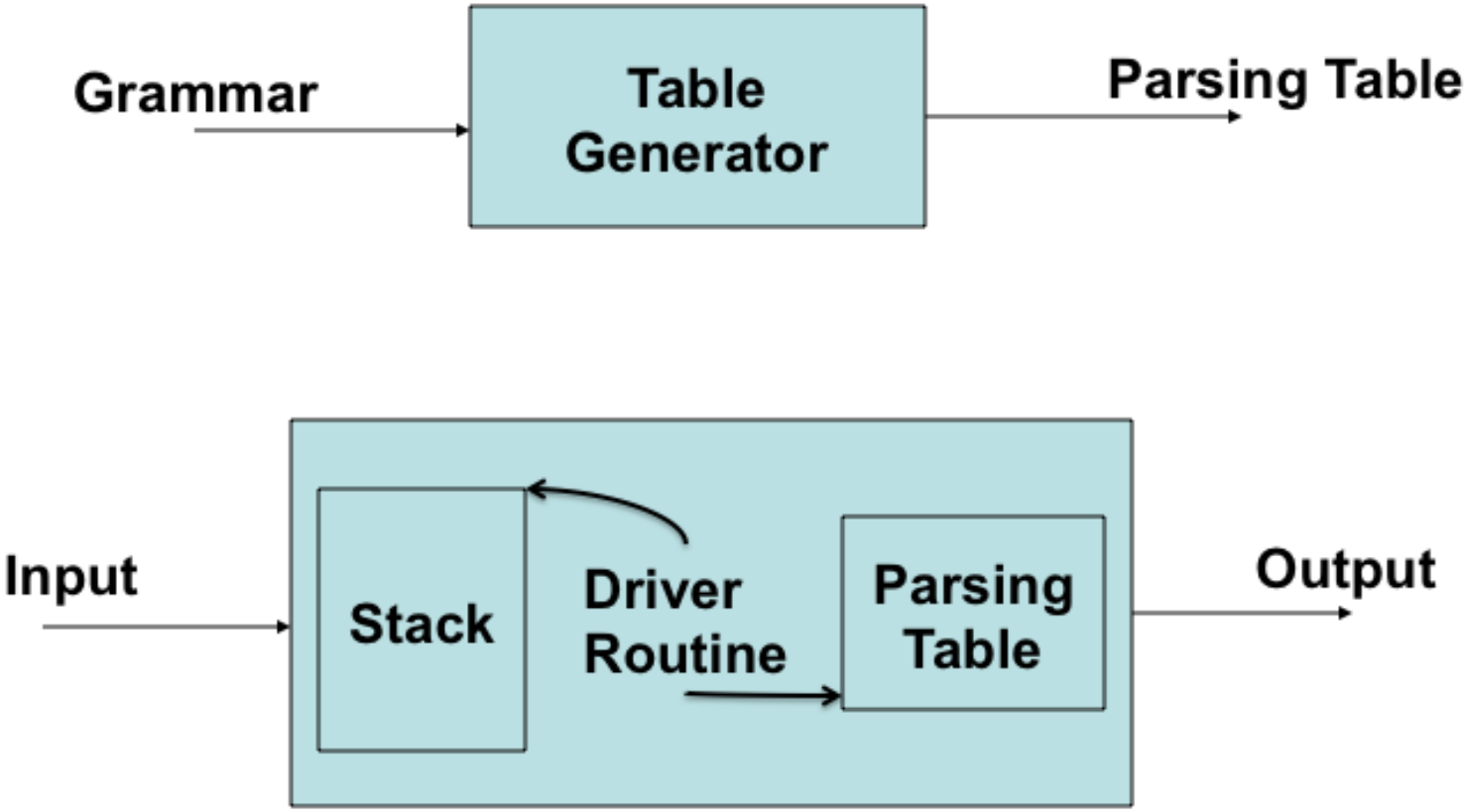
$ marks the bottom of stack and the right end of the input

Stack	Input	Action
$	$id_1 + id_2 * id_3$	shift
$ id_1	$+id_2 * id_3$	reduce by $E \rightarrow id$
$ E	$+id_2 * id_3$	shift
$ $E+$	$id_2 * id_3$	shift
$ $E + id_2$	$*id_3$	reduce by $E \rightarrow id$
$ $E + E$	$*id_3$	shift
$ $E + E*$	id_3	shift
$ $E + E * id_3$	$	reduce by $E \rightarrow id$
$ $E + E * E$	$	reduce by $E \rightarrow E * E$
$ $E + E$	$	reduce by $E \rightarrow E + E$
$ E	$	accept

LR Parsing

- LR(k) - *L*eft to right scanning with *R*ightmost derivation in reverse, k being the number of lookahead tokens
  - $k = 0, 1$ are of practical interest
- LR parsers are also automatically generated using parser generators
- LR grammars are a subset of CFGs for which LR parsers can be constructed
- LR(1) grammars can be written quite easily for practically all programming language constructs for which CFGs can be written
- LR parsing is the most general non-backtracking shift-reduce parsing method (known today)
- LL grammars are a strict subset of LR grammars - an LL(k) grammar is also LR(k), but not vice-versa

LR Parser Generation

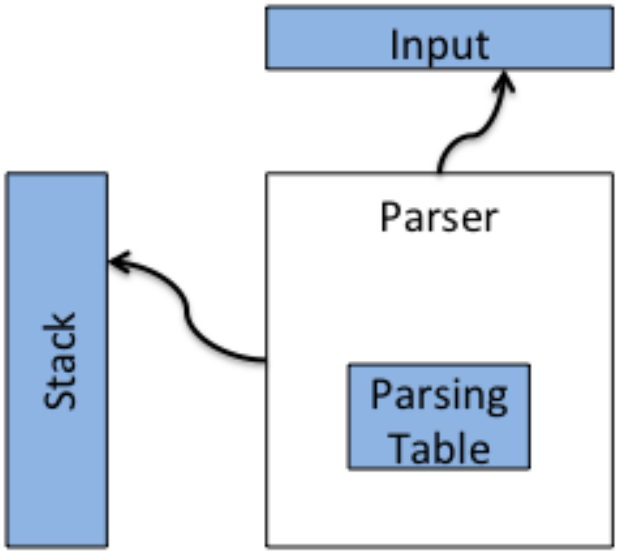


LR Parser Generator

LR Parser Configuration

- A configuration of an LR parser is:
  $(s_0 X_1 s_2 X_2 ... X_m s_m, \quad a_i a_{i+1} ... a_n \ \$)$, where,
  stack **unexpended input**
  $s_0, s_1, ..., s_m$, are the states of the parser, and $X_1, X_2, ..., X_m$, are grammar symbols (terminals or nonterminals)
- Starting configuration of the parser: $(s_0, a_1 a_2 ... a_n \$)$, where, s_0 is the initial state of the parser, and $a_1 a_2 ... a_n$ is the string to be parsed
- Two parts in the parsing table: *ACTION* and *GOTO*
  - The *ACTION* table can have four types of entries: **shift, reduce, accept**, or **error**
  - The *GOTO* table provides the next state information to be used after a *reduce* move

LR Parsing Algorithm



Initial configuration: Stack = *state 0*, Input = *w$*,
a = first input symbol;
repeat {
  let *s* be the top stack state;
  let *a* be the next input symbol;
  if (*ACTION[s, a] == shift p*) {
    push *a* and *p* onto the stack (in that order);
    advance input pointer;
  } else if (*ACTION[s,a] == reduce A* → *α*) then {
    pop *2*|α|* symbols off the stack;
    let *s'* be the top of stack state now;
    push *A* and *GOTO[s', A]* onto the stack
    (in that order);
  } else if (*ACTION[s, a] == accept*) break;
    /* parsng is over */
    else *error()*;
} until true; /* for ever */

LR Parsing Example 1 - Parsing Table

STATE	ACTION				GOTO		
	a	b	c	$	S	A	B
0	S2		S3		1		
1				R1 acc			
2	S2	S6	S3		8	4	
3	R3	R3	R3	R3			
4	S2		S3		5		
5	R2	R2	R2	R2			
6	S7						
7	R4	R4	R4	R4			
8	S2	S10	S3		12		9
9	R5	R5	R5	R5			
10	S2	S6	S3		8	11	
11	R6	R6	R6	R6			
12	R7	R7	R7	R7			

1. S' → S
2. S → aAS
3. S → c
4. A → ba
5. A → SB
6. B → bA
7. B → S