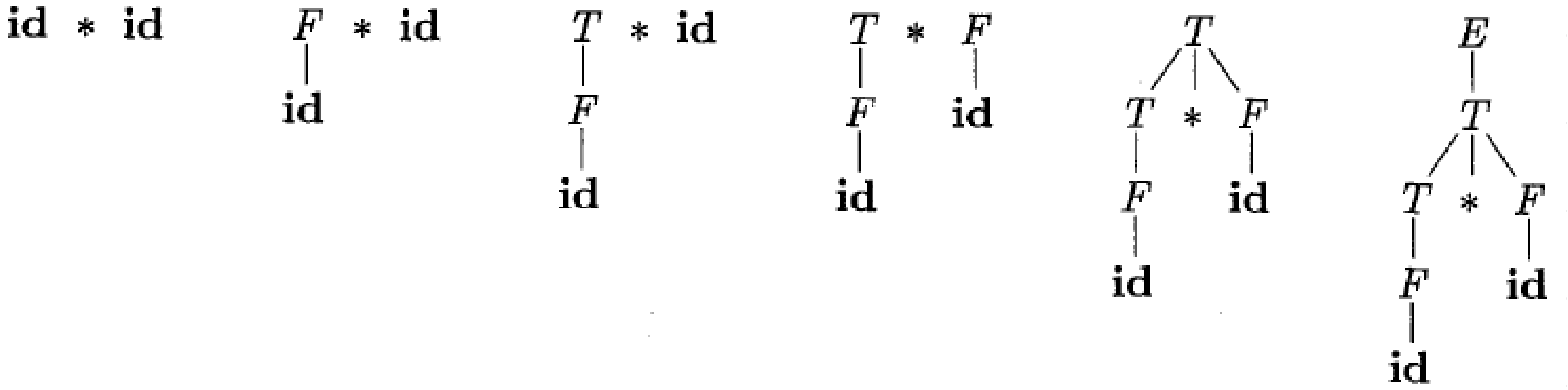
Chapter 6:
Bottom-Up Parsing (Shift-Reduce)

陳奇業 成功大學資訊工程系

Overview

- We study bottom-up (also called LR) parsers, whose operation can be compared with top-down parsers as follows:
  - A bottom-up parser begins with the parse tree's leaves and moves toward its root. A top-down parser moves the parse tree's root toward its leaves.
  - A bottom-up parser traces a rightmost derivation in reverse. A top-down parser traces a leftmost derivation.
  - A bottom-up parser uses a grammar rule to replace the rule's right-hand side (RHS) with its left-hand side (LHS). A top-down parser does the opposite, replacing a rule's LHS with its RHS.

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$



$E \rightarrow T \rightarrow T * F \rightarrow T * id \rightarrow F * id \rightarrow id * id$

An Example

Grammar :

$S \;\rightarrow\; aABe$

$A \;\rightarrow\; Abc \mid b$

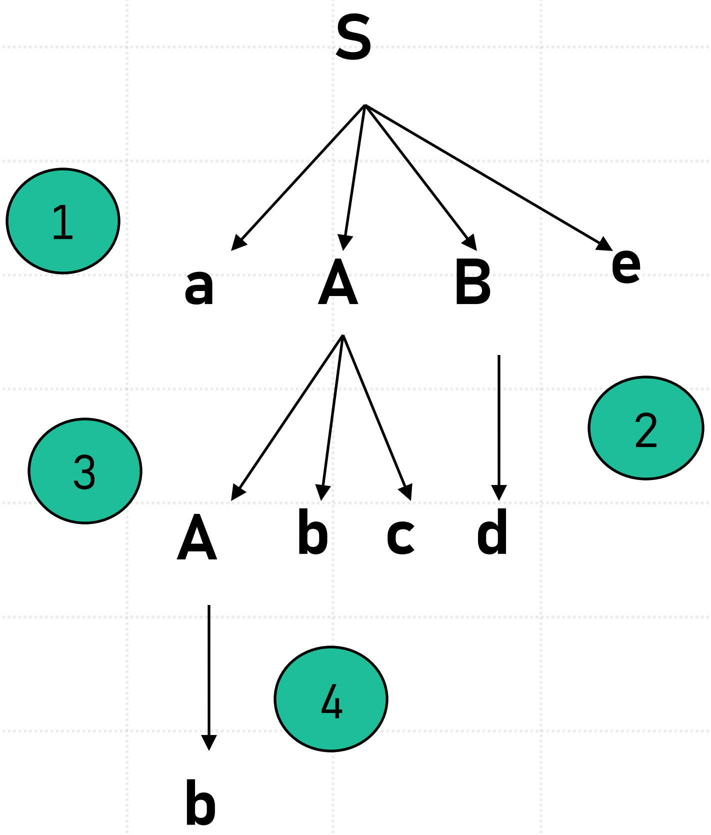
$B \;\rightarrow\; d$

Input: $w \;=\; \boldsymbol{abbcde}$

$S \Rightarrow_{\mathrm{rm}} aABe \Rightarrow_{\mathrm{rm}} aAde \Rightarrow_{\mathrm{rm}} aAbcde \Rightarrow_{\mathrm{rm}} abbcde$ (rightmost derivation)

LR parsing:

$abbcde \Rightarrow aAbcde \Rightarrow aAde \Rightarrow aABe \Rightarrow S$ (rightmost derivation in reverse)



LR parsing:

$$abbcde \Longrightarrow aAbcde \Longrightarrow aAde \Longrightarrow aABe \Longrightarrow S$$

Overview

- The style of parsing considered in this chapter is known by the following names:
  - Bottom-up, because the parser works its way from the terminal symbols to the grammar's goal symbol
  - Shift-reduce, because the two most prevalent actions taken by the parser are to shift symbols onto the parse stack and to reduce a string of such symbols located at the top-of-stack to one of the grammar's non-terminals
  - LR(k), because such parsers scan the input from the left (the "L" in LR) producing a rightmost derivation (the "R" in LR) in reverse, using k symbols of lookahead

Handle Pruning

- Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse.
- Informally, a "handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.
- Given a sentential form, the handle is defined as the sequence of symbols that will next be replaced by reduction.

Example

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

Handle Pruning

- Formally, if $S \Rightarrow^*_{\mathrm{rm}} \alpha A w \Rightarrow_{\mathrm{rm}} \alpha\beta w$, then production $A \to \beta$ in the position following α is a handle of $\alpha\beta w$. Notice that the string w to the right of the handle must contain only terminal symbols.
- For convenience, we refer to the body β rather than $A \to \beta$ as a handle.
- Note we say "a handle" rather than "the handle," because the grammar could be ambiguous, with more than one rightmost derivation of $\alpha\beta w$.
- If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

Handle Pruning

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

$$S = \gamma_0 \Rightarrow_{\mathrm{rm}} \gamma_1 \Rightarrow_{\mathrm{rm}} \gamma_2 \cdots \Rightarrow_{\mathrm{rm}} \gamma_{n-1} \Rightarrow_{\mathrm{rm}} \gamma_n = w$$

Shift-Reduce Parsing

- There are four actions a parser can make:
  - Shift. Shift the next input symbol onto the top of the stack.
  - Reduce. The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
  - Accept. Announce successful completion of parsing.
  - Error. Discover a syntax error and call an error recovery routine.

Stack Implementation of Bottom-Up Parsing

- There is an important fact that justifies the use of a stack in shift-reduce parsing: the handle will always eventually appear on top of the stack, never inside.

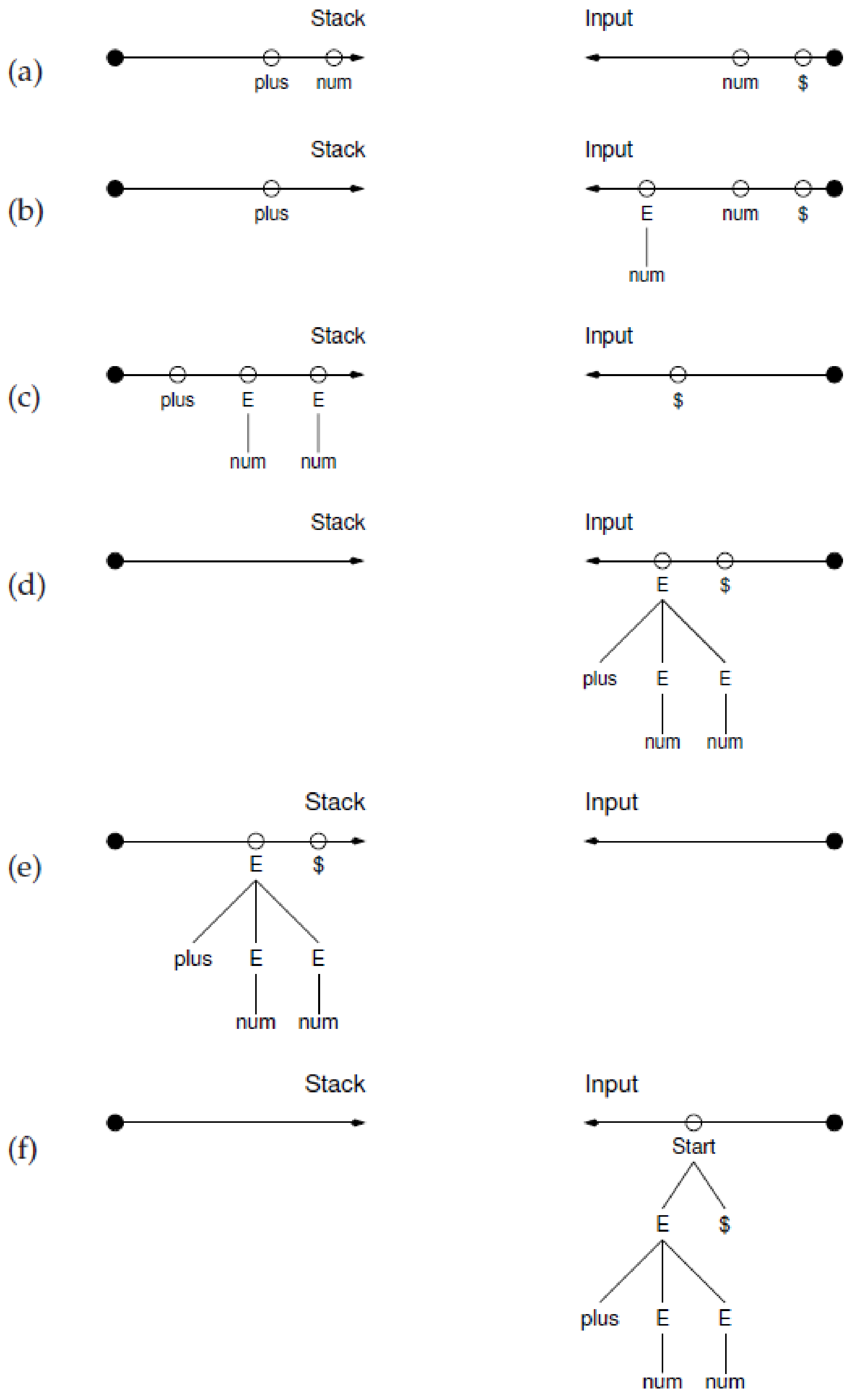
Initially,	(stack) $	$w$$ (input buffer)
⋮		⋮
Finally,	(stack) $$S$	$ (input buffer) // S is a start symbol of grammar G

Example (from 龍書)

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \,\$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2 \,\$$	reduce by $F \to \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2 \,\$$	reduce by $T \to F$
$\$\,T$	$* \mathbf{id}_2 \,\$$	shift
$\$\,T *$	$\mathbf{id}_2 \,\$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \to T * F$
$\$\,T$	$\$$	reduce by $E \to T$
$\$\,E$	$\$$	accept

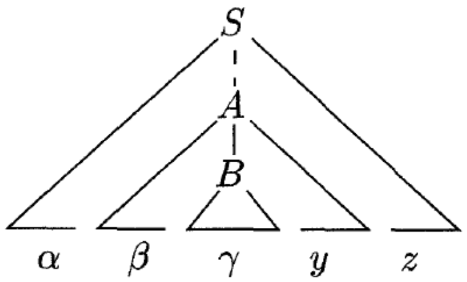
1	Start	$\rightarrow$	E \$
2	E	$\rightarrow$	plus E E
3		|	num

Rule	Derivation
1	Start $\Rightarrow_{rm}$ E \$
2	$\Rightarrow_{rm}$ plus E E \$
3	$\Rightarrow_{rm}$ plus E num \$
3	$\Rightarrow_{rm}$ plus num num \$

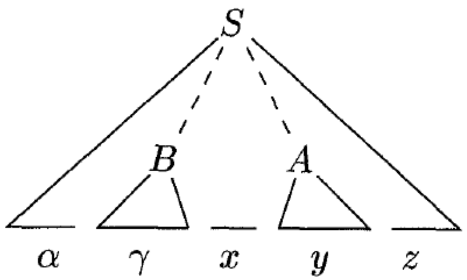


Shift-Reduce Parsing

- The use of a stack in shift-reduce parsing is justified by an important fact: the handle will always eventually appear on top of the stack, never inside.
- This fact can be shown by considering the possible forms of two successive steps in any rightmost derivation. In case (I), A is replaced by $\beta B y$, and then the rightmost nonterminal B in the body $\beta \gamma y$ is replaced by γ. In case (2), A is again expanded first, but this time the body is a string y of terminals only. The next rightmost nonterminal B will be somewhere to the left of y.



Case (1)



Case (2)

Shift-Reduce Parsing

- In other words:

(1) $S \Rightarrow_{\text{rm}}^{*} \alpha Az \Rightarrow_{\text{rm}} \alpha\beta Byz \Rightarrow_{\text{rm}} \alpha\beta\gamma yz$

(2) $S \Rightarrow_{\text{rm}}^{*} \alpha BxAz \Rightarrow_{\text{rm}} \alpha Bxyz \Rightarrow_{\text{rm}} \alpha\gamma xyz$

Consider case (1) in reverse

STACK	INPUT	ACTION
$\$\alpha\beta\gamma$	$yz\$$	reduce by $B \rightarrow \gamma$
$\$\alpha\beta B$	$yz\$$	shift
$\$\alpha\beta By$	$z\$$	reduce by $A \rightarrow \beta By$

Consider case (2)

STACK	INPUT	ACTION
$\$\alpha\gamma$	$xyz\$$	reduce by $B \rightarrow \gamma$
$\$\alpha B$	$xyz\$$	shift
$\$\alpha Bx$	$yz\$$	shift
$\$\alpha Bxy$	$z\$$	reduce by $A \rightarrow$ y

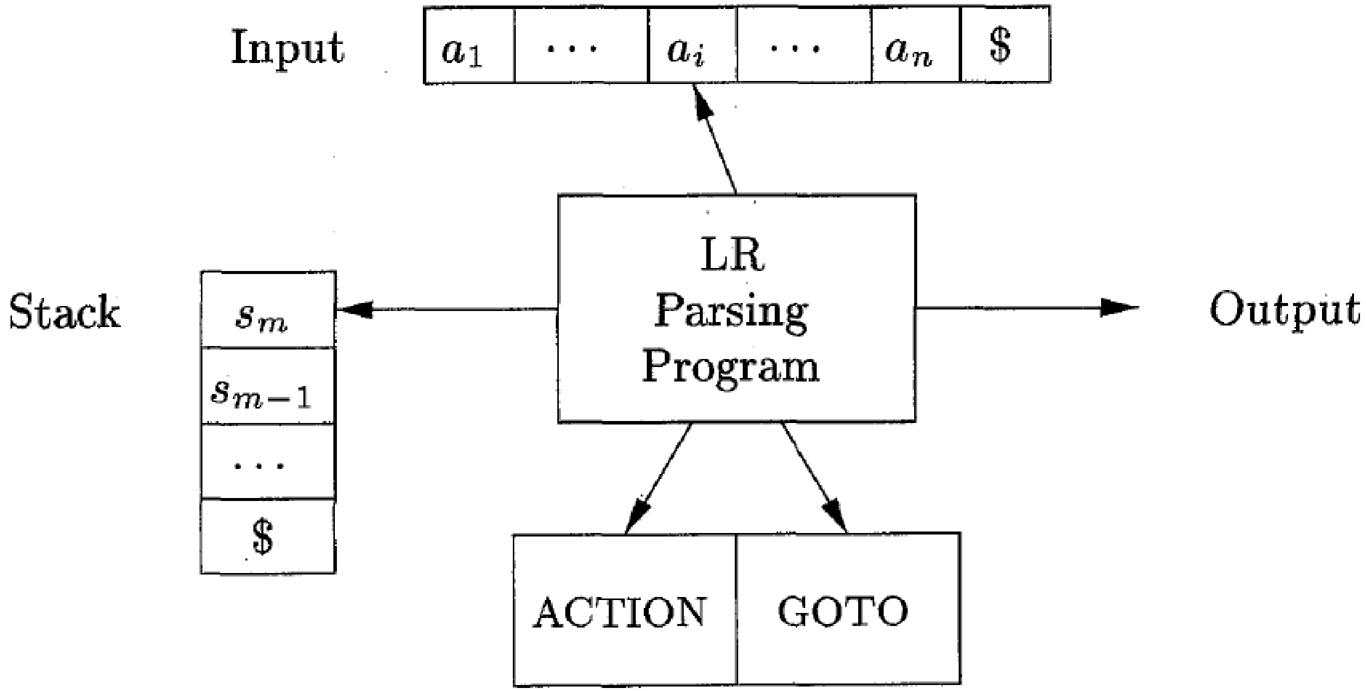
LR Parsers

- Advantages:
  - LR parsers can be constructed to recognize all programming language construct for which context-free grammars can be written.
  - The LR-parsing method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other, more primitive shift-reduce methods
  - The class of grammars that can be parsed by LR parser is the proper superset of the class of grammars that can be parsed by predictive parsers.
  - LR parsers can detect errors in syntax as soon as possible
- Drawbacks:
  - Too much work to do

LR Parsing Engine

```
call Stack.PUSH( StartState )
accepted ← false
while not accepted do
    action ← Table[Stack.TOS( )][InputStream.PEEK( )]        1
    if action = shift s
    then
        call Stack.PUSH( s )                                  2
        if s ∈ AcceptStates                                   3
        then   accepted ← true
        else   call InputStream.ADVANCE( )
    else
        if action = reduce A → γ
        then
            call Stack.POP( |γ| )                             4
            call InputStream.PREPEND( A )                     5
        else
            call ERROR( )                                     6
```

LR Parsing Engine



Structure of the LR Parsing Table

- The parsing table consists of two parts: a parsing-action function ACTION and a goto function GOTO
  1. The ACTION function takes as arguments a state i and a terminal a (or \$, the input endmarker). The value of ACTION$[i, a]$ can have one of four forms:
     (a) Shift j, where j is a state. The action taken by the parser effectively shifts input a to the stack, but uses state j to represent a.
     (b) Reduce $A \to \beta$. The action of the parser effectively reduces β on the top of the stack to head A.
     (c) Accept. The parser accepts the input and finishes parsing.
     (d) Error. The parser discovers an error in its input and takes some corrective action.
  2. We extend the GOTO function, defined on sets of items, to states: if GOTO$[I_i, A]$= I_j then GOTO also maps a state i and a nonterminal A to state j.

1	Start	$\rightarrow$	S \$
2	S	$\rightarrow$	A C
3	C	$\rightarrow$	c
4		$\mid$	λ
5	A	$\rightarrow$	a B C d
6		$\mid$	B Q
7	B	$\rightarrow$	b B
8		$\mid$	λ
9	Q	$\rightarrow$	q
10		$\mid$	λ

Rule	Derivation
1	Start $\Rightarrow_{rm}$ S \$
2	$\Rightarrow_{rm}$ A C \$
3	$\Rightarrow_{rm}$ A c \$
5	$\Rightarrow_{rm}$ a B C d c \$
4	$\Rightarrow_{rm}$ a B d c \$
7	$\Rightarrow_{rm}$ a b B d c \$
7	$\Rightarrow_{rm}$ a b b B d c \$
8	$\Rightarrow_{rm}$ a b b d c \$

State	a	b	c	d	q	\$	Start	S	A	B	C	Q
0	$\boxed{3}$	$\boxed{2}$	8		8	8	accept	$\boxed{4}$	$\boxed{1}$	$\boxed{5}$		
1			$\boxed{11}$			4					$\boxed{14}$	
2		$\boxed{2}$	8	8	8	8				$\boxed{13}$		
3		$\boxed{2}$	8	8						$\boxed{9}$		
4						$\boxed{8}$						
5			10		$\boxed{7}$	10						$\boxed{6}$
6			6			6						
7			9			9						
8						1						
9			$\boxed{11}$	4							$\boxed{10}$	
10				$\boxed{12}$								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

State	a	b	c	d	q	$	Start	S	A	B	C	Q
0	$\boxed{3}$	$\boxed{2}$	8		8	8	accept	$\boxed{4}$	$\boxed{1}$	$\boxed{5}$		
1			$\boxed{11}$			4					$\boxed{14}$	
2		$\boxed{2}$	8	8	8	8				$\boxed{13}$		
3		$\boxed{2}$	8	8						$\boxed{9}$		
4						$\boxed{8}$						
5			10		$\boxed{7}$	10						$\boxed{6}$
6			6			6						
7			9			9						
8						1						
9			$\boxed{11}$	4							$\boxed{10}$	
10				$\boxed{12}$								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

1 Start → S $
2 S → A C
3 C → c
4 | λ
5 A → a B C d
6 | B Q
7 B → b B
8 | λ
9 Q → q
10 | λ

Stack	Action	Input
0	Initial Configuration	a b b d c $
0, a 3	shift a	b b d c $
0, a 3, b 2	shift b	b d c $
0, a 3, b 2, b 2	shift b	d c $
0, a 3, b 2, b 2	Reduce λ to B	B d c $
0, a 3, b 2, b 2, B 13	shift B	d c $
0, a 3, b 2	Reduce b B to B	B d c $
0, a 3, b 2, B 13	shift B	d c $
0, a 3	Reduce b B to B	B d c $
0, a 3, B 9	shift B	d c $
0, a 3, B 9	Reduce λ to C	C d c $
0, a 3, B 9, C 10	shift C	d c $
0, a 3, B 9, C 10, d 12	shift d	c $
0	Reduce a B C d to A	A c $

State	a	b	c	d	q	$	Start	S	A	B	C	Q
0	$\boxed{3}$	$\boxed{2}$	8		8	8	accept	$\boxed{4}$	$\boxed{1}$	$\boxed{5}$		
1			$\boxed{11}$			4					$\boxed{14}$	
2		$\boxed{2}$	8	8	8	8				$\boxed{13}$		
3		$\boxed{2}$	8	8						$\boxed{9}$		
4						$\boxed{8}$						
5			10		$\boxed{7}$	10						$\boxed{6}$
6			6			6						
7			9			9						
8						1						
9			$\boxed{11}$	4							$\boxed{10}$	
10				$\boxed{12}$								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

1 Start → S $
2 S → A C
3 C → c
4 | λ
5 A → a B C d
6 | B Q
7 B → b B
8 | λ
9 Q → q
10 | λ

0	*(continued from Figure 6.6)*	A c $
0 · A 1	shift A	c $
0 · A 1 · c 11	shift c	$
0 · A 1	Reduce c to C	C $
0 · A 1 · C 14	shift C	$
0	Reduce A C to S	S $
0 · S 4	shift S	$
0 · S 4 · $ 8	shift $	$
0	Reduce S $ to Start	Start $
0 · Start 0	shift Start	$
	Accept	

LR(k) Parsing

- As is the case with LL parsers, LR parsers are parameterized by the number of lookahead symbols that are consulted to determine the appropriate parser action.
- An LR(k) parser can peek at the next k tokens.
- This notion of "peeking" and the term LR(0) are confusing, because even an LR(0) parser must refer to the next input token, for the purpose of indexing the parse table to determine the appropriate action. The "0" in LR(0) refers not to the lookahead at parse time, but rather to the lookahead used in constructing the parse table.
- At parse-time, LR(0) and LR(1) parsers index the parse table using one token of lookahead; for $k \geq 2$, an LR(k) parser uses k tokens of lookahead.

LR(k) Parsing

- The number of columns in an LR(k) parse table grows dramatically with k.
- For example, an LR(3) parse table is indexed by the parse state to select a row, and by the next 3 input tokens to select a column.
- If the terminal alphabet has n symbols, then the number of distinct three-token sequences is n^3. More generally, an LR(k) table has n^k columns for a token alphabet of size n.
- To keep the size of parse tables within reason, most parser generators are limited to one token of lookahead.

LR(k) Parsing

- LR(k) parsing decide the next action by examining the tokens already shifted and at most k lookahead tokens
- A grammar is LR(k) if, and only if, it is possible to construct an LR parse table such that k tokens of lookahead allows the parser to recognize exactly those strings in the grammar's language.

LR(0) Table Construction

- To keep track of the parser's progress, we introduce the notion of an LR(0) item—a grammar production with a bookmark that indicates the current progress through the production's RHS.

LR(0) item	Progress of rule in this state
$E \rightarrow \bullet\, \text{plus}\ E\ E$	Beginning of rule
$E \rightarrow \text{plus} \bullet E\ E$	Processed a plus, expect an E
$E \rightarrow \text{plus}\ E \bullet E$	Expect another E
$E \rightarrow \text{plus}\ E\ E\, \bullet$	Handle on top-of-stack, ready to reduce

LR(0) Table Construction

- Definition: An LR(0) item of a grammar G is a production of G with a dot (●) at some position of the right side. e.g. $A \rightarrow XYZ$ has 4 items

$$A \rightarrow \bullet XYZ$$
$$A \rightarrow X \bullet YZ$$
$$A \rightarrow XY \bullet Z$$
$$A \rightarrow XYZ \bullet$$

- $A \rightarrow \lambda$ has one item $A \rightarrow \bullet$

- Items can be denoted by pairs of integers in computer.

- Items can be viewed as the states of an NFA recognizing viable prefixes.

Closure of Item Sets

- If I is a set of items for a grammar G, then CLOSURE(I)s the set of items constructed from I by the two rules:
  - Initially, add every item in I to CLOSURE(I)
  - If $A \rightarrow \alpha \bullet B\beta$ is in CLOSURE(I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \bullet\gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

```
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for ( each item A → α·Bβ in J )
            for ( each production B → γ of G )
                if ( B → ·γ is not in J )
                    add B → ·γ to J;
    until no more items are added to J on one round;
    return J;
}
```

Example

- Consider the augmented expression grammar

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

If I is the set of one item $\{[E' \to \bullet E]\}$, then CLOSURE(I) contains the set of items:

$E' \to \bullet E$, $E \to \bullet E + T$, $E \to \bullet T$, $T \to \bullet T * F$, $T \to \bullet F$, $F \to \bullet (E)$, $F \to \bullet id$

LR(0) items

- We divide all the sets of items of interest into two classes:
  - Kernel items: the initial item, $S' \to \bullet S$, and all items whose dots are not at the left end.
  - Nonkernel items: all items with their dots at the left end, except for $S' \to \bullet S$.

- We now define a parser state as a set of LR(0) items. While each state is formally a set of items.

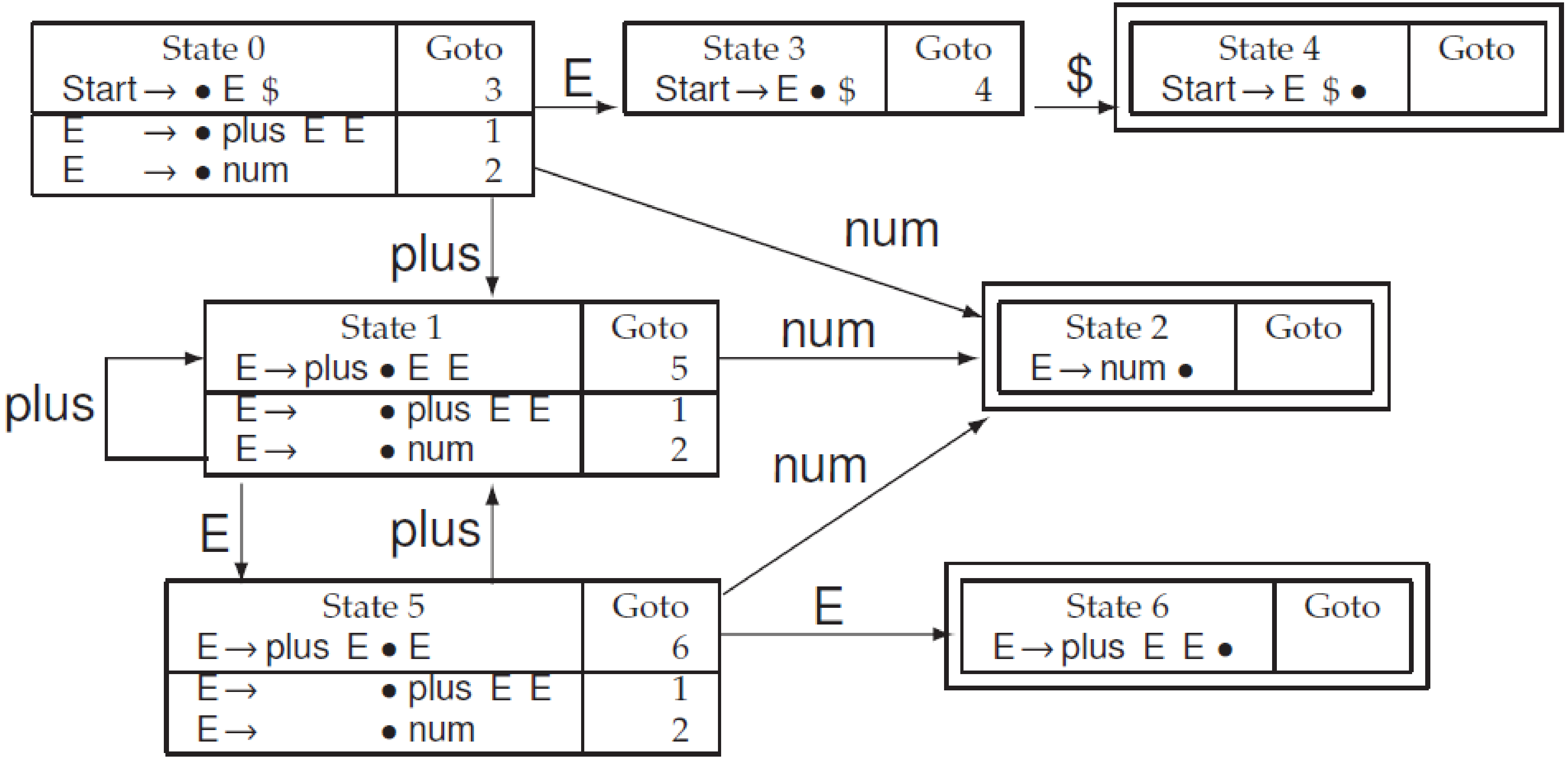
The Function GOTO

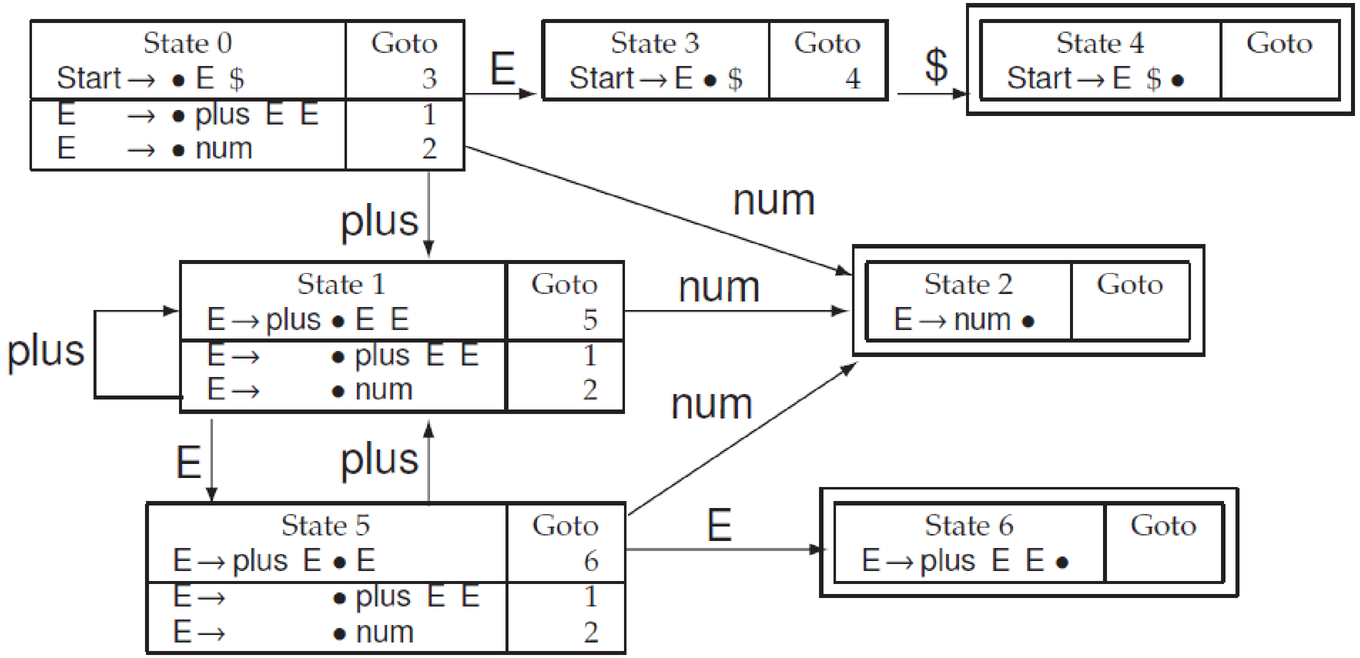
- GOTO(I, X) is defined to be the closure of the set of all items $[A \to \alpha X \bullet \beta\ ]$ such that $[A \to \alpha \bullet X \beta\ ]$ is in I where I is a set of items and X is a grammar symbol.
- Intuitively, the GOTO function is used to define the transitions in the LR(0) automaton for a grammar.
- The states of the automaton correspond to sets of items, and GOTO(I, X) specifies the transition from the state for I under input X.

```
function ComputeLR0( Grammar ) returns (Set, State)
    States ← ∅
    StartItems ← { Start→ • RHS(p) | p ∈ ProductionsFor( Start ) }    7
    StartState ← AddState( States, StartItems )
    while (s ← WorkList.ExtractElement( )) ≠ ⊥ do    8
        call ComputeGoto( States, s )
    return ((States, StartState))
end
function AddState( States, items ) returns State
    if items ∉ States    9
    then
        s ← newState(items)    10
        States ← States ∪ { s }
        WorkList ← WorkList ∪ { s }    11
        Table[s][★] ← error    12
    else  s ← FindState(items)
    return (s)
end
function AdvanceDot( state, X ) returns Set
    return ({ A→αX • β | A→α • Xβ ∈ state })    13
end

function Closure( state ) returns Set
    ans ← state
    repeat    14
        prev ← ans
        foreach A→α • Bγ ∈ ans do    15
            foreach p ∈ ProductionsFor( B ) do
                ans ← ans ∪ { B→ • RHS(p) }    16
    until ans = prev
    return (ans)
end
procedure ComputeGoto( States, s )
    closed ← Closure( s )    17
    foreach X ∈ (N ∪ Σ) do    18
        RelevantItems ← AdvanceDot( closed, X )    19
        if RelevantItems ≠ ∅
        then
            Table[s][X] ← shift AddState( States, RelevantItems )    20
end
```

1 Start → E $
2 E → plus E E
3 | num





Sentential Prefix	Transitions	Resulting Sentential Form
		plus plus num num num $
plus plus num	States 1, 1, and 2	plus plus E num num $
plus plus E num	States 1, 1, 5, and 2	plus plus E E num $
plus plus E E	States 1, 1, 5, and 6	plus E num $
plus E num	States 1, 5, and 2	plus E E $
plus E E	States 1, 5, and 6	E $
E $	States 1, 3, and 4	Start

Characteristic Finite-State Machine (CFSM)

- The basis for LR parsing is a deterministic finite automaton (DFA), called the characteristic finite-state machine (CFSM).
- A viable prefix of a right sentential form is any prefix that does not extend beyond its handle.
- Formally, a CFSM recognizes its grammar's viable prefixes.
- When the automaton arrives in a double-boxed state, it has processed a viable prefix that ends with a handle.

Example 5.2 Consider the following augmented grammar for rudimentary arithmetic expressions (no parentheses and one operation):

$$E' \to E$$
$$E \to E \,\mathbf{+}\, \boldsymbol{n} \mid \boldsymbol{n}$$

A bottom-up parse of the string $\boldsymbol{n} + \boldsymbol{n}$ using this grammar is given in Table 5.2.

Table 5.2
Parsing actions of a bottom-up parser for the grammar of Example 5.2

	Parsing stack	Input	Action
1	$\$$	$\boldsymbol{n} + \boldsymbol{n}\ \$$	shift
2	$\$\ \boldsymbol{n}$	$+ \boldsymbol{n}\ \$$	reduce $E \to \boldsymbol{n}$
3	$\$\ E$	$+ \boldsymbol{n}\ \$$	shift
4	$\$\ E\ +$	$\boldsymbol{n}\ \$$	shift
5	$\$\ E + \boldsymbol{n}$	$\$$	reduce $E \to E + \boldsymbol{n}$
6	$\$\ E$	$\$$	reduce $E' \to E$
7	$\$\ E'$	$\$$	accept

§

$$E' \Rightarrow E \Rightarrow E + n \Rightarrow n + n$$

$E, E+, E+n$ are all **viable prefixes** of the right-sentential form $\underline{\boldsymbol{E+n}}$.

Completing an LR(0) Parse Table

```
procedure CompleteTable(Table, grammar)
    call ComputeLookahead( )
    foreach state ∈ Table do
        foreach rule ∈ Productions(grammar) do
            call TryRuleInState(state, rule)
    call AssertEntry(StartState, GoalSymbol, accept)
end
procedure AssertEntry(state, symbol, action)
    if Table[state][symbol] = error
    then  Table[state][symbol] ← action
    else
        call ReportConflict(Table[state][symbol], action)
end
```

```
procedure TryRuleInState(s, r)
    if LHS(r)→RHS(r) • ∈ s
    then
        foreach X ∈ (Σ ∪ N) do  call AssertEntry(s, X, reduce r)
end
```

State	num	plus	$	Start	E
0	$\boxed{2}$	$\boxed{1}$		accept	$\boxed{3}$
1	$\boxed{2}$	$\boxed{1}$			$\boxed{5}$
2	reduce 3				
3			$\boxed{4}$		
4	reduce 1				
5	$\boxed{2}$	$\boxed{1}$			$\boxed{6}$
6	reduce 2				

LR(0) Parse (from 龍書)

Line	Stack	Symbols	Input	Action
(1)	0	$\$$	$\mathbf{id} * \mathbf{id}\ \$$	shift to 5
(2)	0 5	$\$\ \mathbf{id}$	$* \mathbf{id}\ \$$	reduce by $F \to \mathbf{id}$
(3)	0 3	$\$\ F$	$* \mathbf{id}\ \$$	reduce by $T \to F$
(4)	0 2	$\$\ T$	$* \mathbf{id}\ \$$	shift to 7
(5)	0 2 7	$\$\ T *$	$\mathbf{id}\ \$$	shift to 5
(6)	0 2 7 5	$\$\ T * \mathbf{id}$	$\$$	reduce by $F \to \mathbf{id}$
(7)	0 2 7 10	$\$\ T * F$	$\$$	reduce by $T \to T * F$
(8)	0 2	$\$\ T$	$\$$	reduce by $E \to T$
(9)	0 1	$\$\ E$	$\$$	accept

LR(0) Parse (from 龍書)

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Conflict Diagnosis

- If we consider the possibilities for multiple table-cell entries, only the following two cases are troublesome for LR(k) parsing:
  - shift/reduce conflicts exist in a state when table construction cannot use the next k tokens to decide whether to shift the next input token or call for a reduction.
  - reduce/reduce conflicts exist when table construction cannot use the next k tokens to distinguish between multiple reductions.

Conflict Diagnosis

- Conflicts arise for one of the following reasons:
  - The grammar is ambiguous. No (deterministic) table-construction method can resolve conflicts that arise due to ambiguity.
  - The grammar is not ambiguous, but the current table-building approach could not resolve the conflict. In this case, the conflict might disappear if one or more of the following approaches is taken:
    - The current table-construction method is given more lookahead.
    - A more powerful table-construction method is used.

Ambiguous Grammars

1	Start	→	E $
2	E	→	E plus E
3		|	num

State 0	Goto
Start → • E $	2
E → • E plus E	2
E → • num	1

State 1	Goto
E → num •	

State 2	Goto
E → E • plus E	3
Start → E • $	4

State 3	Goto
E → E plus • E	5
E → • E plus E	5
E → • num	1

State 4	Goto
Start → E $ •	

State 5	Goto
E → E plus E •	
E → E • plus E	3

1 Start → E $
2 E → E plus E
3 | num

State 0	Goto
Start → • E $	2
E → • E plus E	2
E → • num	1

State 1	Goto
E → num •	

State 2	Goto
E → E • plus E	3
Start → E • $	4

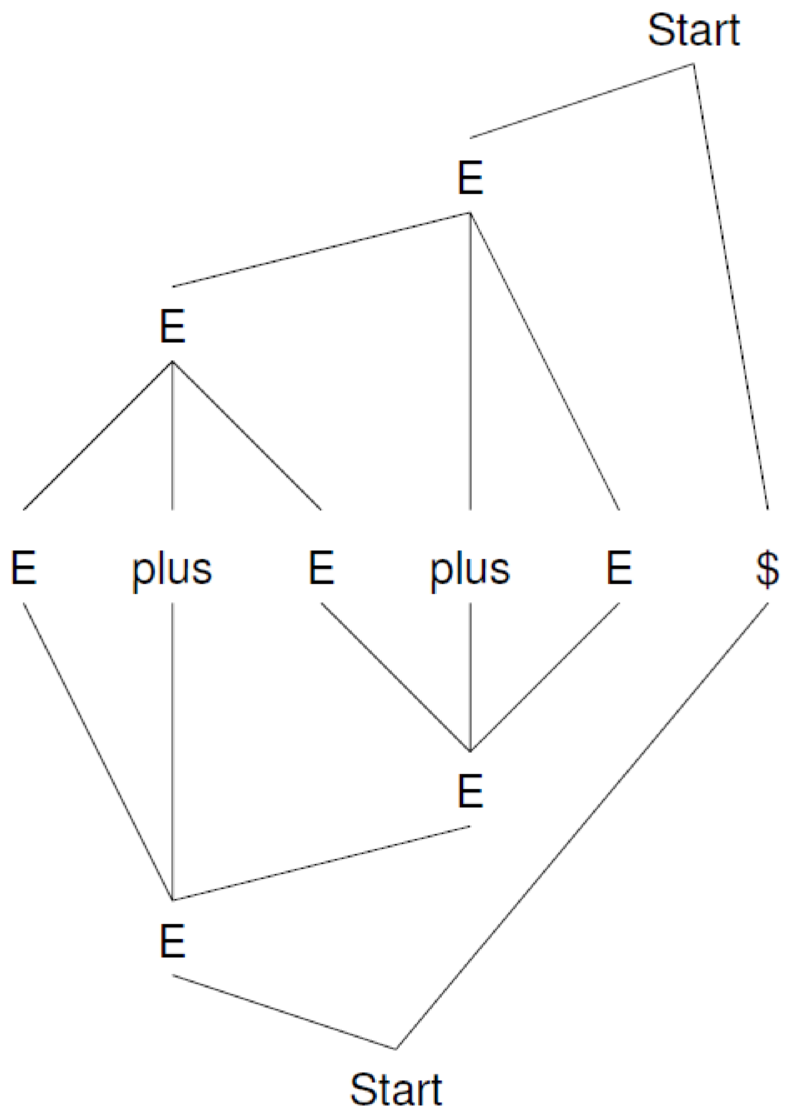
State 3	Goto
E → E plus • E	5
E → • E plus E	5
E → • num	1

State 4	Goto
Start → E $ •	

State 5	Goto
E → E plus E •	
E → E • plus E	3

Ambiguous Grammars

The parse tree that favors the reduction in State 5 corresponds to a left-associative grouping for addition, while the shift corresponds to a right-associative grouping.



Ambiguous Grammars

1	Start	→	E $
2	E	→	E plus E
3		|	num

State 0	Goto
Start → • E $	2
E → • E plus E	2
E → • num	1

State 1	Goto
E → num •	

State 2	Goto
E → E • plus E	3
Start → E • $	4

State 3	Goto
E → E plus • E	5
E → • E plus E	5
E → • num	1

State 4	Goto
Start → E $ •	

State 5	Goto
E → E plus E •	
E → E • plus E	3

Ambiguous

1	Start	→	E $
2	E	→	E plus num
3		|	num

State 0	Goto
Start → • E $	2
E → • E plus num	2
E → • num	1

State 1	Goto
E → num •	

State 2	Goto
E → E • plus num	3
Start → E • $	4

State 3	Goto
E → E plus • num	5

State 4	Goto
Start → E $ •	

State 5	Goto
E → E plus num •	

Unambiguous

Ambiguous Grammars

- A statement beginning with $p(i, j)$ would appear as the token stream $id(id, id)$ to the parser. After shifting the first three tokens onto the stack, a shift-reduce parser would be in configuration

(stack) \$… $id(id$ $, id)$ …\$ (input buffer)

- Make things as easy as possible for the parser. It should be left to scanner to determine if id is a procedure or an array.

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

array

procedure

Grammars that are not LR(k)

1	Start	$\rightarrow$	Exprs \$
2	Exprs	$\rightarrow$	E a
3		|	F b
4	E	$\rightarrow$	E plus num
5		|	num
6	F	$\rightarrow$	F plus num
7		|	num

$$
\begin{aligned}
\text{Start} &\Rightarrow_{\text{rm}} \text{Exprs \$} \\
&\Rightarrow_{\text{rm}} \text{E a \$} \\
&\Rightarrow_{\text{rm}} \text{E plus num a \$} \\
&\Rightarrow^{\star}_{\text{rm}} \text{E plus ... plus num a \$} \\
&\Rightarrow_{\text{rm}} \text{num plus ... plus num a \$}
\end{aligned}
$$

State 0	Goto
Start $\rightarrow$ • Exprs \$	1
Exprs $\rightarrow$ • E a	4
Exprs $\rightarrow$ • F b	3
E $\rightarrow$ • E plus num	4
E $\rightarrow$ • num	2
F $\rightarrow$ • F plus num	3
F $\rightarrow$ • num	2

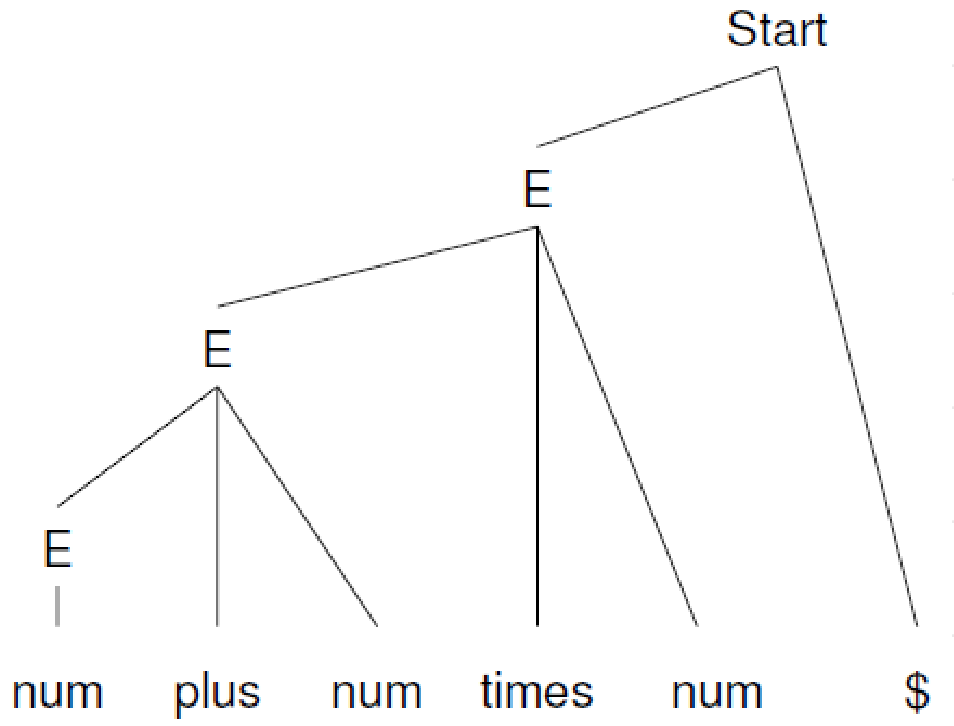
State 2	Goto
E $\rightarrow$ num •	
F $\rightarrow$ num •	

must know the last character of input

SLR(k) Table Construction

The SLR(k) (Simple LR with k tokens of lookahead) method attempts to resolve inadequate states using grammar analysis methods.

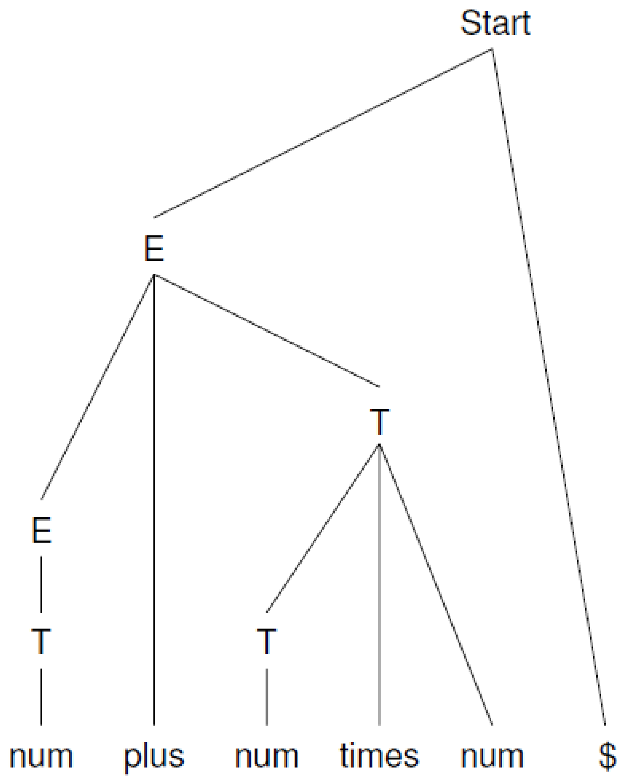
1	Start	$\rightarrow$	E \$
2	E	$\rightarrow$	E plus num
3		|	E times num
4		|	num



this grammar is LR(0)

SLR(k) Table Construction

1	Start	→	E $
2	E	→	E plus T
3		|	T
4	T	→	T times num
5		|	num



State 0	Goto
Start → • E $	3
E → • E plus T	3
E → • T	1
T → • T times num	1
T → • num	2

State 1	Goto
E → T •	
T → T • times num	7

shift/reduce conflict

State 2	Goto
T → num •	

State 3	Goto
Start → E • $	5
E → E • plus T	4

State 4	Goto
E → E plus • T	6
T → • T times num	6
T → • num	2

State 5	Goto
Start → E $ •	

State 6	Goto
E → E plus T •	
T → T • times num	7

shift/reduce conflict

State 7	Goto
T → T times • num	8

State 8	Goto
T → T times num •	

Stack	Action	Input
0	Initial Configuration	E plus num times num $
0, E 3	shift E	plus num times num $
0, E 3, plus 4	shift plus	num times num $
0, E 3, plus 4, num 2	shift num	times num $
0, E 3, plus 4	Reduce num to T	T times num $
0, E 3, plus 4, T 6	shift T	times num $

$$\xrightarrow{reduce} \mathrm{E} \xrightarrow{shift} E\ times \xrightarrow{shift} E\ times\ num$$

Error!!

SLR(k) Table Construction

- With the item $E \rightarrow E\ plus\ T\bullet$ in State 6, reduction by $E \rightarrow E\ plus\ T$ must be appropriate under some conditions. If we examine the sentential forms $E\ plus\ T\ \$$ and $E\ plus\ T\ plus\ num\ \$$, we see that the $E \rightarrow E\ plus\ T$ must be applied in State 6 when the next input symbol is $plus$ or $\$$, but not $times$.
- If the reduction to E can lead to a successful parse, then $plus$ (or \$) can appear next to E in some valid sentential form. An equivalent statement is $plus \in Follow(E)$
- For our example, States 1 and 6 are resolved by computing $Follow(E) = \{\ plus, \$\ \}$.

SLR(k) Table Construction

```
procedure CompleteTable( Table, grammar )
    call ComputeLookahead( )
    foreach state ∈ Table do
        foreach rule ∈ Productions(grammar) do
            call TryRuleInState( state, rule )
    call AssertEntry( StartState, GoalSymbol, accept )
end
procedure AssertEntry( state, symbol, action )
    if Table[state][symbol] = error
    then  Table[state][symbol] ← action
    else
        call ReportConflict( Table[state][symbol], action )
end
```

```
procedure TryRuleInState( s, r )
    if LHS( r ) → RHS( r ) • ∈ s
    then
        foreach X ∈ (Σ ∪ N) do  call AssertEntry( s, X, reduce r )
end
```

```
procedure TryRuleInState( s, r )
    if LHS( r ) → RHS( r ) • ∈ s
    then
        foreach X ∈ Follow( LHS( r ) ) do
            call AssertEntry( s, X, reduce r )
end
```

SLR(k) Table Construction

State 0	Goto
Start → • E $	3
E → • E plus T	3
E → • T	1
T → • T times num	1
T → • num	2

State 1	Goto
E → T •	
T → T • times num	7

State 2	Goto
T → num •	

State 3	Goto
Start → E • $	5
E → E • plus T	4

State 4	Goto
E → E plus • T	6
T → • T times num	6
T → • num	2

State 5	Goto
Start → E $ •	

State 6	Goto
E → E plus T •	
T → T • times num	7

State 7	Goto
T → T times • num	8

State 8	Goto
T → T times num •	

State	num	plus	times	$	Start	E	T
0	$\boxed{2}$				accept	$\boxed{3}$	$\boxed{1}$
1		3	$\boxed{7}$	3			
2		5	5	5			
3		$\boxed{4}$		$\boxed{5}$			
4	$\boxed{2}$						$\boxed{6}$
5				1			
6		2	$\boxed{7}$	2			
7	$\boxed{8}$						
8		4	4	4			

- Exercises 4, 5, 6, 10