



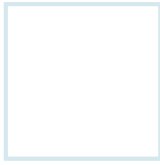
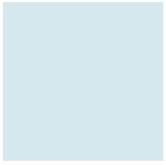
COMPILER CONSTRUCTION

Bottom-Up Parsing

Chia-Heng Tu

Dept. of Computer Science and Information
Engineering

National Cheng Kung University
Spring 2018



Chapter 6

Bottom-Up Parsing



Bottom-Up Parsing

- This chapter discusses the techniques and tools for automatically constructing bottom-up parsers
- Especially, the constructions of the parsing table for LR parsers are introduced in this chapter



Why Bottom-Up Parsing?

- Bottom-up parsers are commonly used in the syntax-checking phase of a compiler
 - because of their power, efficiency, and ease of construction
- **Bottom-up parsing can accommodate the grammar features,**
 - which are problematic for top-down parsing,
 - e.g., left recursive productions and common prefixes (Fig. 5.12)
- In fact, bottom-up parsers can handle the largest class of grammars that allow parsing to proceed **deterministically**

```
1 Stmt    → if Expr then StmtList endif
2         | if Expr then StmtList else StmtList endif
3 StmtList → StmtList ; Stmt
4         | Stmt
5 Expr     → var + Expr
6         | var
```

Figure 5.12: A grammar with common prefixes.



Why Bottom-Up Parsing? (Cont'd)

- Those problems would be addressed by rewriting the grammar,
 - which can be used to construct a top-down parser
 - E.g., grammar in Fig. 5.12 can be converted into another in Fig. 5.16
- Unfortunately, that grammar does not clearly articulate the **language's syntax**

```

1 Stmt    → if Expr then StmtList V1
2 V1     → endif
3         | else StmtList endif
4 StmtList → X Y
5 X       → Stmt
6 Y       → ; Stmt Y
7         | λ
8 Expr    → var V2
9 V2     → + Expr
10        | λ
    
```

Figure 5.16: LL(1) version of the grammar in Figure 5.14.



Reprise: Top-Down Parsing (Ch. 5)

- We had learned how to construct top-down parsers based on context-free grammars (CFGs) that had certain properties
- The fundamental concern of an LL parser is
 - **which production to choose in expanding a given nonterminal**
 - This choice is based on **the parser's current state** and on a peek at the unconsumed portion of **the parser's input string**
- The **derivations** and **parse trees** produced by LL parsers are constructed as follows:
 - the **leftmost nonterminal** is expanded at each step, and **the parse tree grows systematically – top-down**, from left to right
 - The LL parser **begins with the tree's root**, which is labeled with the grammar's start symbol
 - Suppose that A is the next nonterminal to be expanded, and that the parser chooses the production $A \rightarrow \gamma$
 - In the parse tree, the node corresponding to this A is supplied with children that are labeled with the symbols in γ



Bottom-Up Parsing

- We compare the high-level concepts (e.g., derivations and parse trees) of bottom-up parsers against top-down parsers
 - An LR 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
 - An LR parser traces a **rightmost derivation in reverse**
 - A top-down parser traces a leftmost derivation
 - An LR 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
 - An LR parser can concurrently **anticipate the eventual success of multiple nonterminals**
 - In an LL parser, each state is committed to expand a particular nonterminal
 - This flexibility makes LR parsers more general than LL parsers



Bottom-Up Parsing (Cont'd)

- You should know the difference between derivation and parsing by now
- If you don't, please
 - write down the left-most and right-most derivation for the input: "begin simplestmt ; simplestmt ; end \$" and
 - compare the derivations against the parsing steps in Fig. 4.5 and 4.6 respectively

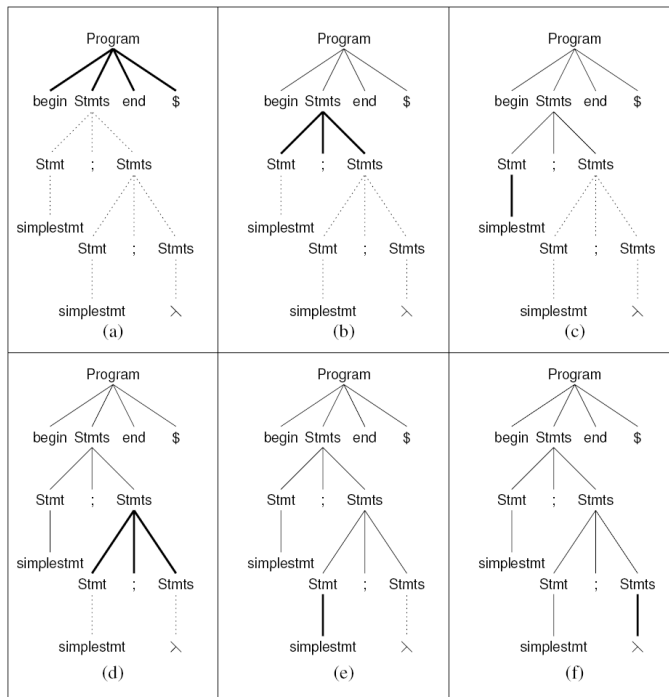


Figure 4.5: Parse of "begin simplestmt ; simplestmt ; end \$" using the top-down technique. Legend explained on page 126.

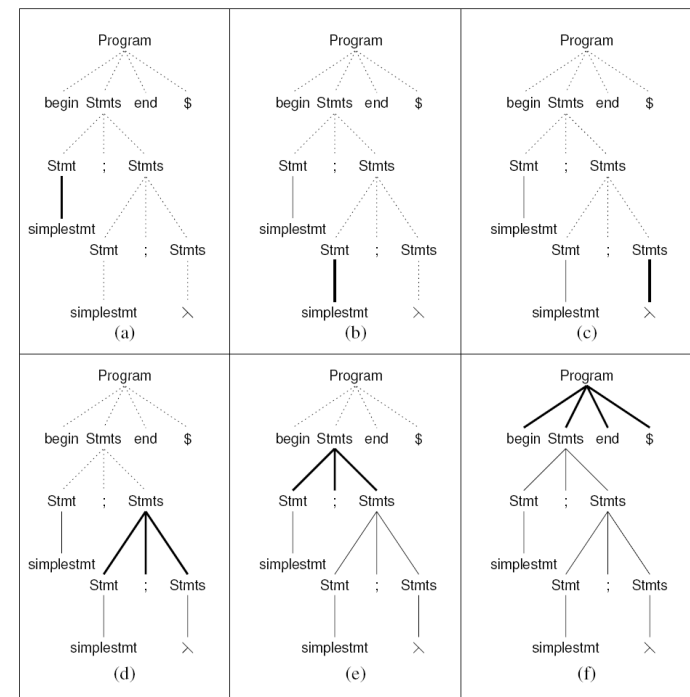


Figure 4.6: Parse of "begin simplestmt ; simplestmt ; end \$" using the bottom-up technique. Legend explained on page 126.



Bottom-Up Parsing (Cont'd)

- The style of parsing considered in Ch. 6 is known by the following names:
- **Bottom-up**
 - The parser works its way **from the terminal symbols to the grammar's goal symbol**
- **Shift-reduce**
 - The two most prevalent actions taken by the bottom-up parsers 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 nonterminals
- **LR(k)**
 - The bottom-up 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
 - It should be clear in context which meaning is intended; LR denotes both:
 - (1) the **generic bottom-up parsing engine**
 - (2) as well as a **particular technique for constructing the engine's tables**



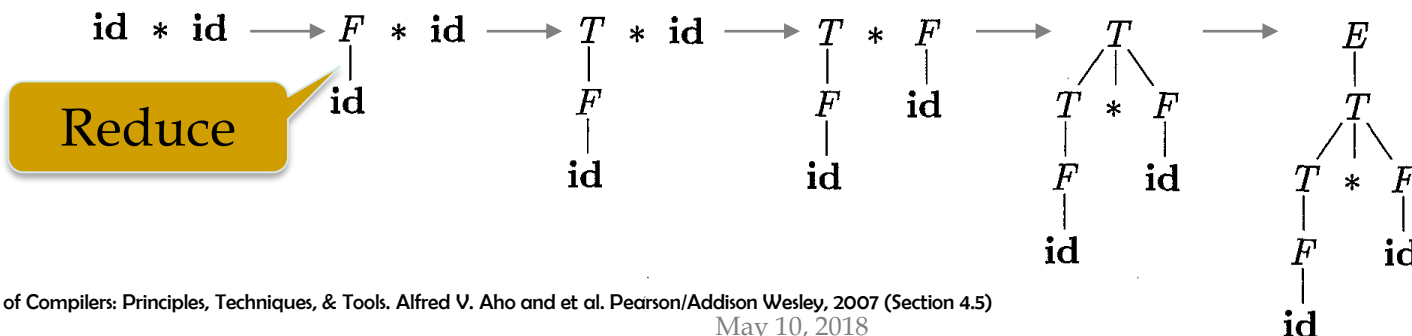
Shift-Reduce Parsers

- Shift-reduce parsing is a **form of bottom-up parsing**
- The **bottom-up parsing** is the process of *reducing* a token string to the start symbol of the grammar
 - At each reduction, the token string matching the **RHS** of a production is replaced by the **LHS** non-terminal of that production
 - The following slides illustrate the **parsing** procedure and the sequence of **derivation**



An Illustration of Bottom-Up Parsing

- Given the grammar:
 - $E \Rightarrow T$
 - $T \Rightarrow T * F$
 - $T \Rightarrow F$
 - $F \Rightarrow \text{id}$
- The **bottom-up parse** for the input string:
 - $\text{id} * \text{id}$





An Illustration of Bottom-Up Parsing (Cont'd)

- Given the grammar:
 - $E \Rightarrow T$
 - $T \Rightarrow T * F$
 - $T \Rightarrow F$
 - $F \Rightarrow \text{id}$
- The **rightmost derivation** for the input string:

$$\begin{aligned} & - \text{id} * \text{id} \\ & E \Rightarrow_{\text{rm}} T \\ & \Rightarrow_{\text{rm}} T * F \\ & \Rightarrow_{\text{rm}} T * \text{id} \\ & \Rightarrow_{\text{rm}} F * \text{id} \\ & \Rightarrow_{\text{rm}} \text{id} * \text{id} \end{aligned}$$



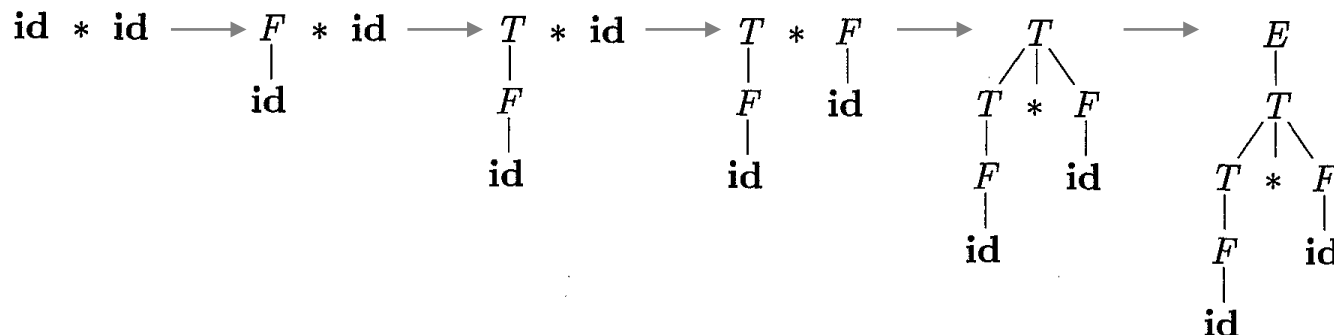
An Illustration of Bottom-Up Parsing (Cont'd)

- Given the grammar:

- $E \Rightarrow T$
- $T \Rightarrow T * F$
- $T \Rightarrow F$
- $F \Rightarrow \text{id}$

$$\begin{aligned}
 E &\Rightarrow_{\text{rm}} T \\
 &\Rightarrow_{\text{rm}} T * F \\
 &\Rightarrow_{\text{rm}} T * \text{id} \\
 &\Rightarrow_{\text{rm}} F * \text{id} \\
 &\Rightarrow_{\text{rm}} \text{id} * \text{id}
 \end{aligned}$$

To understand an LR parsing sequence is to appreciate that such parses construct **rightmost derivations in reverse**

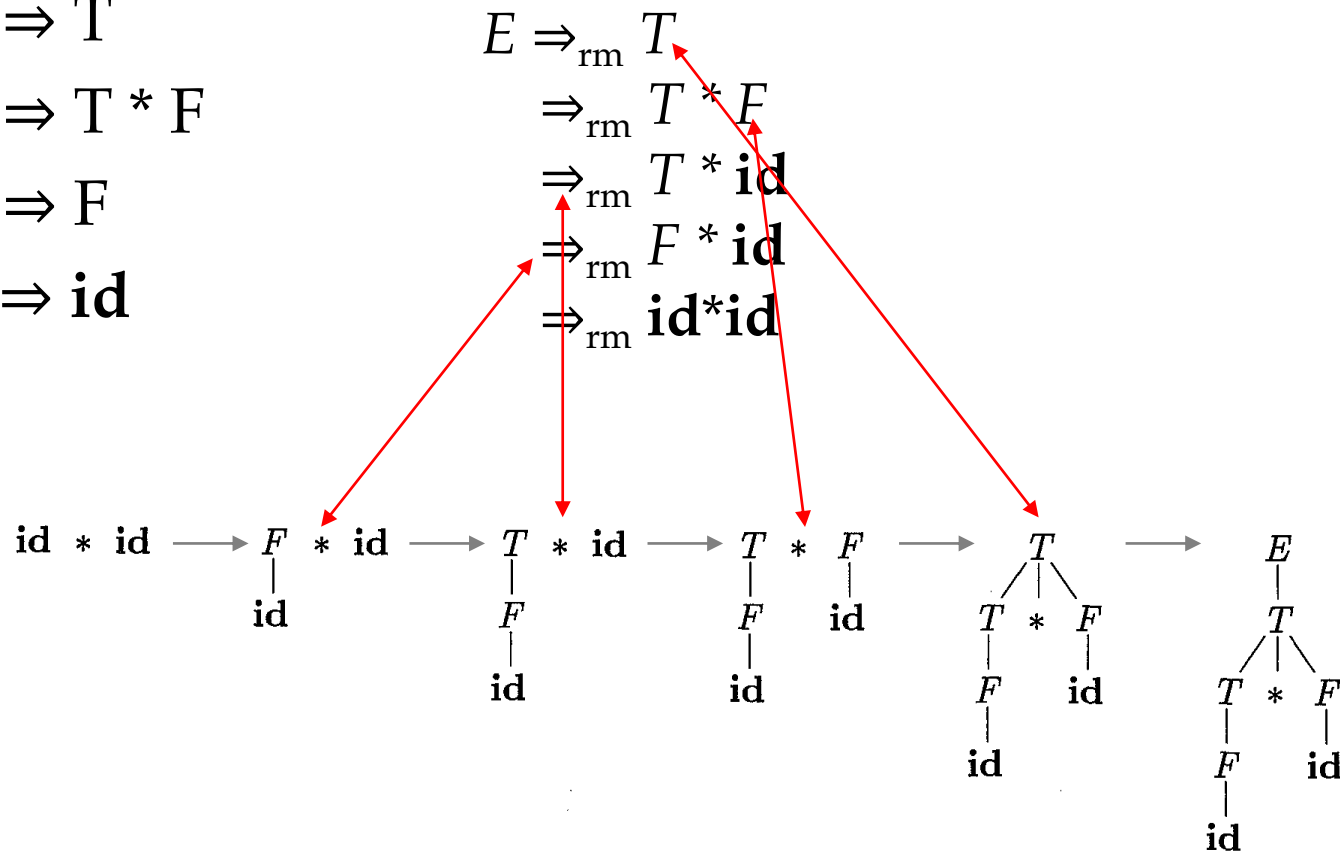




An Illustration of Bottom-Up Parsing (Cont'd)

- Given the grammar:

- $E \Rightarrow T$
- $T \Rightarrow T * F$
- $T \Rightarrow F$
- $F \Rightarrow \text{id}$





Shift-Reduce Parsing as Knitting

- Two important *needles* for the parsing:
 - a **stack** holds grammar symbols and
 - an **input buffer** holds the rest of the tokens to be parsed
 - We use **\$** to mark the end of the input (and also the bottom of the stack)
- During a left-to-right scan of the input tokens,
 - the parser **shifts** zero or more input tokens into the stack,
 - until it is ready to **reduce a string β of grammar symbols on top of the stack**
 - It then reduces β to the LHS of the appropriate production
- The parser repeats this cycle
 - until it has detected an error or
 - until the stack contains the start symbol and the input is empty (\$)





Actions of Shift-Reduce Parsers

- **Shift:** shift the next input token onto the top of the stack
- **Reduce:** 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 what non-terminal to replace that string
- **Accept:** announce successful completion of parsing
- **Error:** discover a syntax error and call an error recovery routine



*Illustration of LR Parsing Steps

- We examine **how** the RHS of a production is found so that a reduction can occur
- The parser halts and announces **successful** completion of parsing, upon entering the configuration below:

Stack Input
\$ Start \$

- The right figure steps through the actions that
 - a shift-reduce parser takes in parsing the input string $id_1 * id_2$
- Please do the exercise with the grammar and string in Fig. 6.2, and the parse progress in Fig. 6.1

Given the grammar:

$E \Rightarrow T$

$T \Rightarrow T * F$

$T \Rightarrow F$

$F \Rightarrow id$

The input string:

$id_1 * id_2$

Configurations of a shift-reduce parser on input $id_1 * id_2$

STACK	INPUT	ACTION
\$	$id_1 * id_2$ \$	shift
\$ id_1	$* id_2$ \$	reduce by $F \rightarrow id$
\$ F	$* id_2$ \$	reduce by $T \rightarrow F$
\$ T	$* id_2$ \$	shift
\$ $T *$	id_2 \$	shift
\$ $T * id_2$	\$	reduce by $F \rightarrow id$
\$ $T * F$	\$	reduce by $T \rightarrow T * F$
\$ T	\$	reduce by $E \rightarrow T$
\$ E	\$	accept





*Illustration of LR Parsing Steps (Cont'd)

- We examine **how** the RHS of a production is found so that a reduction can occur
- The right figure steps through the actions that
 - a shift-reduce parser takes in parsing the input string $\text{id}_1 * \text{id}_2$
 - Animations reveal the details
 - It is fine...
The representation style of this example after the reduction is different from what we have in the textbook

Given the grammar:

$E \Rightarrow T$

$T \Rightarrow T * F$

$T \Rightarrow F$

$F \Rightarrow \text{id}$

The input string:

$\text{id}_1 * \text{id}_2$

Configurations of a shift-reduce parser on input $\text{id}_1 * \text{id}_2$

STACK	INPUT	ACTION
\$	$\text{id}_1 * \text{id}_2$	\$① shift
\$ id_1	$* \text{id}_2$	\$② reduce by $F \rightarrow \text{id}$
\$ F	$* \text{id}_2$	\$③ reduce by $T \rightarrow F$
\$ T	$* \text{id}_2$	\$④ shift
\$ $T *$	id_2	\$⑤ shift
\$ $T * \text{id}_2$		\$⑥ reduce by $F \rightarrow \text{id}$
\$ $T * F$		\$⑦ reduce by $T \rightarrow T * F$
\$ T		\$⑧ reduce by $E \rightarrow T$
\$ E		\$⑨ accept



LR Parsing and Derivations

- Given a grammar and a rightmost derivation of some string in its language,
- **the sequence of productions applied by an LR parser is the sequence used by the rightmost derivation, but played backwards**
- It is all about the **order** in which productions are applied to perform a bottom-up parse



Another Example of LR Parsing and Derivation

- Fig. 6.2 shows a grammar and the rightmost derivation of a string in the grammar's language
- Each step of the derivation is annotated with the production number used at that step
- The **derivation** of the string **plus num num \$** is achieved by applying **Rules 1, 2, 3, and 3**

```

1 Start → E $
2 E    → 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 \$

Figure 6.2: Grammar and rightmost derivation of plus num num \$.



Another Example of LR Parsing and Derivation (Cont'd)

- A bottom-up (LR) **parse**
 - is accomplished by playing this sequence backwards: **Rules 3, 3, 2, and 1**
 - In contrast to LL parsing, an LR parser finds the RHS of a production and replaces it with the production's LHS
 - First, the leftmost **num** is **reduced** to an **E** by the rule **$E \rightarrow \text{num}$**
 - This rule is applied again to obtain **plus E E \$**
 - The rule is then reduced by **$E \rightarrow \text{plus E E}$** to obtain **E \$**
 - This can then be reduced by **Rule 1 to the goal symbol Start**

```

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 \$

Figure 6.2: Grammar and rightmost derivation of plus num num \$.

Another Example of LR Parsing and Derivation (Cont'd)

- LR parsing sequence is the rightmost derivation in reverse
- This slide illustrates the matched step for derivation and parsing

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 \$

Some actions are omitted here (b) to (c)

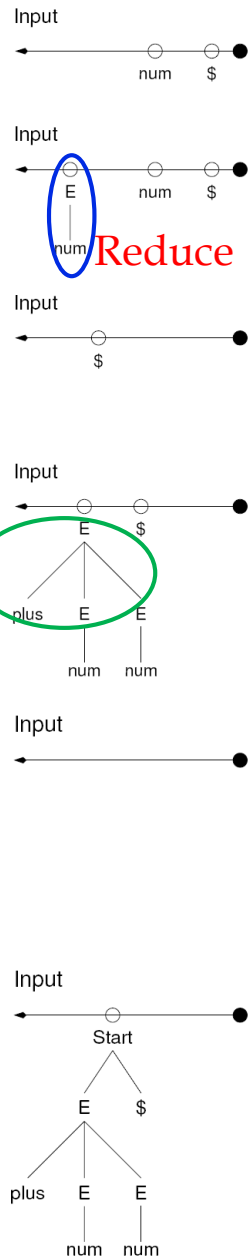
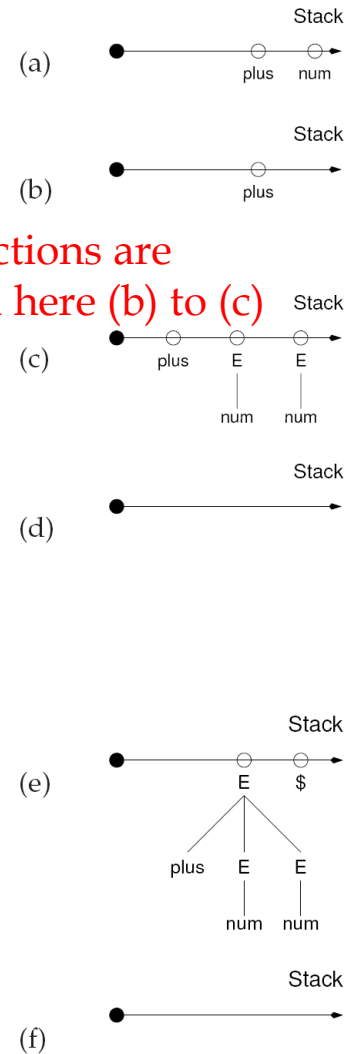


Figure 6.2: Grammar and rightmost derivation of plus num num \$.

Figure 6.1: Bottom-up parsing resembles knitting.



The Engine of Shift-Reduce Parsers

- A systematic method for the shift-reduce parsing is shown in Fig. 6.3
- The engine is driven by a **parse table**,
 - which records the **action** under the certain **parser's current state** and **the next (unprocessed) input symbol**
 - The **current state** of the parser is defined by the contents of the parser's stack, especially the top of the stack
- More about the table is discussed later (Sec. 6.2.4)

```

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

①

②

③

④

⑤

⑥

Figure 6.3: Driver for a bottom-up parser.



The Engine of Shift-Reduce Parsers (Cont'd)

- Given the `Stack.TOS()` and `InputStream.PEEK()`, the next action to be taken is (**Marker 1**):
 - shift**. It performs a shift of the next input symbol to state s (**Marker 2**)
 - reduce**. The RHS of a production is popped off the stack and its LHS symbol is **prepended** to the input (**Marker 4 & 5**)
 - NOTE: the prepend operation is not illustrated in the previous slide (Illustration of LR Parsing Steps); in the next step of the prepending, the prepended symbol/state will be shifted to stack first**
 - the operations affect the execution result of the table-driven parsing**
 - Fig. 6.1 illustrates the parsing steps**
- The parser continues to perform shift and reduce actions until one of the following situations occurs:
 - The input is reduced to the grammar's goal symbol (**Marker 3**); The input string is **accepted**
 - No valid action is found (**Marker 1**); in this case, the input string has a syntax **error** (**Marker 6**)

```

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

①

②

③

④

⑤

⑥

Figure 6.3: Driver for a bottom-up parser.



LR Parse Table

- An LR parse constructs a rightmost derivation in reverse
 - Each reduction step in the LR parse uses a grammar rule such as $A \rightarrow \gamma$ to replace γ by A
 - Given the **sentential forms** constructed during parsing, the **handle** is defined as **the sequence of symbols that will next be replaced by reduction**
- The difficulties of the parsing lie in:
 - identifying the **handle** and
 - in knowing **which production to employ** in the reduction (when there are multiple productions with the same RHS)
 - These activities are arranged by the parse table
- More about the shift and reduce operation
 - **Tokens are shifted until a handle appears at the top of the parse stack**, at which time the **next reduction** in the reverse derivation can be applied
 - **Shift** actions are essentially implied by the inability to perform a useful reduction
 - The shifted tokens must make progress toward developing a handle

More about LR Parsing

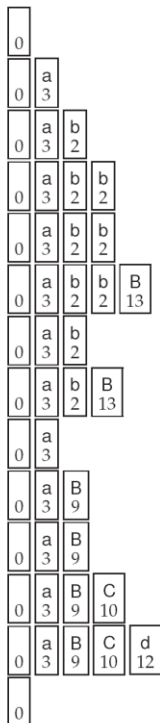
- Fig. 6.6 and 6.7 show the steps of a bottom-up parsing w/ the parse table in Fig. 6.5 and the grammar in Fig. 6.4
- The parser accepts when the **Start** symbol is shifted in the parser's starting state

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

Figure 6.5: Parse table for the grammar shown in Figure 6.4.

Stack

Input



Initial Configuration

a b b d c \$

shift a

b b d c \$

shift b

b d c \$

shift b

dc\$

Reduce λ to B

B d c \$

shift B

dc\$

Reduce $b \ B$ to B

B d c \$

shift B

dc\$

Reduce $b \ B$ to B

B d c \$

shift B

dc\$

Reduce λ to C

C d c \$

shift C

dc\$

shift d

C \$

Reduce a B C d to A

Ac\$

(continue to Figure 6.7)

- A **shift** to State s is denoted by s
 - **Reduction** by rule r is indicated by an unboxed entry of r
 - Blank entries are error actions
-
- Each stack cell is shown as two elements:

a

n
 - The bottom element n is the parser state entered when the cell is pushed
 - The top symbol a is the symbol causing the cell to be pushed
 - The parsing engine in Fig. 6.3 keeps track only of the state

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

Figure 6.4: Grammar and rightmost derivation of a b b d c \$.

Using LR Parse Table (1/2)

- The table determines when to shift and reduce
 - For example, if the next input is b for State 0, 2, and 3, it shifts the input b's all the way
 - When **reduce** is applied, the state remains the same

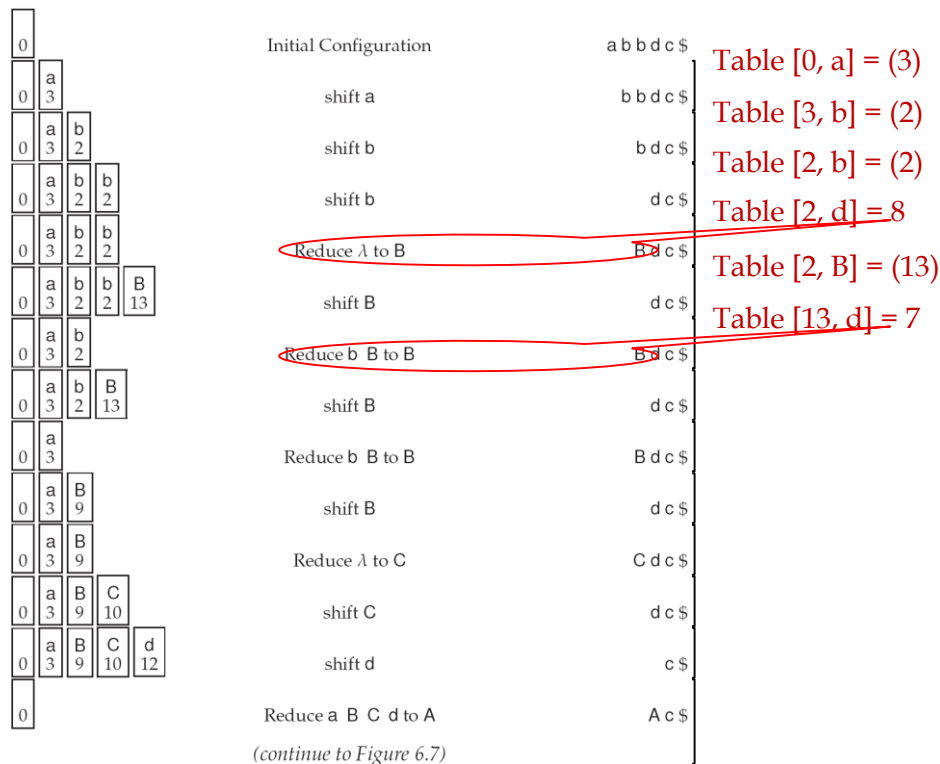


Figure 6.6: Bottom-up parse of a b b d c \$.

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

Figure 6.5: Parse table for the grammar shown in Figure 6.4.

- Start \rightarrow S \$
- S \rightarrow A C
- C \rightarrow c
- $\mid \lambda$
- A \rightarrow a B C d
- \mid B Q
- B \rightarrow b B
- $\mid \lambda$
- Q \rightarrow q
- $\mid \lambda$

- Rule Derivation
- Start \Rightarrow_{rm} S \$
 - \Rightarrow_{rm} A C \$
 - \Rightarrow_{rm} A c \$
 - \Rightarrow_{rm} a B C d c \$
 - \Rightarrow_{rm} a B d c \$
 - \Rightarrow_{rm} a b B d c \$
 - \Rightarrow_{rm} a b b B d c \$
 - \Rightarrow_{rm} a b b d c \$

Figure 6.4: Grammar and rightmost derivation of a b b d c \$.

- You should finish the following steps by yourself

\$

Figure 6.5: Parse table for the grammar shown in Figure 6.4.

28



What's Next?

- The above slides illustrate the high-level concepts of LR parsers
- Now, we take a look at the key properties that an $LR(k)$ parser must possess
- Later, we will learn the $LR(k)$ parsers:
 - $LR(0)$, $SLR(1)$, $LR(1)$, and $LALR(1)$



LR(k) Parsers

- An LR(k) parser, guided by its parse table,
 - must decide whether to shift or reduce,
 - knowing only the symbols already shifted (left context) and the next k lookahead symbols (right context)
- 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
- An important property of an **LR parse table**
 - is that **each cell accommodates only one entry**
 - In other words, the LR(k) parser is **deterministic** – exactly one action can occur at each step
- Please refer to Sec. 6.2.5 for more about LR(k) parsing



LR(0) Table Construction

- The table-construction methods discussed in this chapter
 - analyze a grammar to devise **a parse table suitable for use in the generic parser** presented in Fig. 6.3
- Determination of **inadequate states**
 - An important outcome of the LR construction methods
 - **States** that lack sufficient information to place at most one parsing action in each table entry



Preliminaries – Reduction

- Given a production rule r ,
 - prior to reducing the $\text{RHS}(r)$ to $\text{LHS}(r)$, each component of the RHS must be found

Example

- Consider the rule $E \Rightarrow \text{plus } E E$
- A **plus** must be identified, then two **E**s must be found
- Once these three symbols are on top-of-stack,
 - then it is possible for the parser to **apply the reduction** and **replace** the three symbols with the left-hand side (LHS) symbol **E**



Preliminaries – Item and Bookmark (1/2)

- An LR parser makes shift-reduce decisions
 - by maintaining **states** to keep track of *where we are in a parse*
- States represent sets of **items**
- An LR(0) **item** (item for short)
 - is **a grammar production with a bookmark**
 - that indicates the **current progress** through the production's RHS



Preliminaries – Item and Bookmark (2/2)

- The bookmark is analogous to
 - the *progress bar* present in many applications, which indicates the completed fraction of a task
- An **item** of the form:
 - $A \Rightarrow X_1 \dots X_i \cdot X_{i+1} \dots X_j$
 - The **bookmark symbol** `·', in an item may appear anywhere in the RHS of a production



Preliminaries – Items

- Four items for the rule $A \Rightarrow XYZ$
 1. $A \Rightarrow \cdot XYZ$
 2. $A \Rightarrow X \cdot YZ$
 3. $A \Rightarrow XY \cdot Z$
 4. $A \Rightarrow XYZ \cdot$
- One item for the rule $A \Rightarrow \lambda$
 1. $A \Rightarrow \cdot$
 - λ denotes that there is nothing on this rule's RHS
- Fig. 6.8 shows the progress of the bookmark symbol \cdot
 - through all of the possible LR(0) items for the production $E \rightarrow \text{plus } E E$

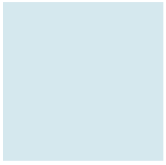
LR(0) item	Progress of rule in this state
$E \rightarrow \bullet \text{plus } E E$	Beginning of rule
$E \rightarrow \text{plus } \bullet E E$	Processed a plus, expect an E
$E \rightarrow \text{plus } E \bullet E$	Expect another E
$E \rightarrow \text{plus } E E \bullet$	Handle on top-of-stack, ready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

Figure 6.8: LR(0) items for production $E \rightarrow \text{plus } E E$.



Preliminaries – Fresh and Reducible Items

- A **fresh** item has its bookmark at the extreme left
 - E.g., $E \rightarrow \cdot \text{plus } E E$
- The item is **reducible** when the bookmark is at the extreme right
 - e.g., as in $E \rightarrow \text{plus } E E \cdot$



Preliminaries – Parser State

- A parser state is **a set of LR(0) items**
 - While each state is formally a set,
 - we drop the usual braces notation and
 - simply list the set's elements (items)
- The start state for our parsers is **state 0**



*Preliminaries – Closure (Items)

- Consider I as **a set of items** for a grammar G
 - **CLOSURE(I)** is the set of items constructed from I by the 2 rules:
 1. Initially, add every item in I to **CLOSURE(I)**
 2. If $A \rightarrow \alpha \cdot B \beta$ is in **CLOSURE(I)**, and $B \rightarrow \gamma$ is a production, then add $B \rightarrow \cdot \gamma$ to **CLOSURE(I)**, if it is not already there
- Apply this until no more new items can be added



*Preliminaries – Closure (Items) (Cont'd)

- Intuitively, $A \rightarrow \alpha \cdot B \beta$ in $\mathbf{CLOSURE}(I)$ indicates:
 - at some point in the parsing process, we think we might next see a substring derivable from $B\beta$ as input
- The substring **derivable from $B \beta$**
 - will have a prefix derivable from B by applying one of the B -productions
 - E.g., $B \rightarrow \cdot \gamma$, $B \rightarrow \cdot C$
- We therefore add items for all the B -productions
 - that is, if $B \rightarrow \gamma$ is a production, we also include $B \rightarrow \cdot \gamma$ in $\mathbf{CLOSURE}(I)$



Preliminaries – Closure (State)

- The closure of state s is computed at **Marker 14** in Fig. 6.10

- More about the **Closure(state)**

for the nonterminal B of each item $A \rightarrow \alpha \cdot B \beta$ in s (**Marker 15**)

for each production of B

add the $\cdot \text{RHS}(B)$ into the *ans* set (**Marker 16**)

repeat the above until the *ans* set is converged

$B \rightarrow \gamma$

The major difference between the definition of **CLOSURE(I)** and of **Closure(state)** in Fig. 6.10 is the input argument, **items** or the **state for the items**

```
function CLOSURE(state) returns Set
  ans ← state
  repeat
    prev ← ans
    foreach  $A \rightarrow \alpha \cdot B \gamma \in ans$  do
      foreach  $p \in \text{PRODUCTIONS\_FOR}(B)$  do
        ans ← ans  $\cup \{B \rightarrow \cdot \text{RHS}(p)\}$ 
    until ans = prev
  return (ans)
end
```

Figure 6.10: LR(0) closure and transitions.

(14)

(15)

(16)

$\cdot \gamma$



LR(0) Closure Example

- Given the grammar as follows:
 - $E' \Rightarrow E$
 - $E \Rightarrow E + T \mid T$
 - $T \Rightarrow T * F \mid F$
 - $F \Rightarrow (E) \mid \text{id}$
 - Where I is the set of one item $\{E' \Rightarrow \cdot E\}$
- We compute the **CLOSURE(I)**



LR(0) Closure Example (Cont'd)

1. $E' \Rightarrow \cdot E$ is put in **CLOSURE(I)** by rule 1
2. Fresh items for **E** (E-productions with bookmarks at the left end) are added:
 $E \Rightarrow \cdot E + T$ and $E \Rightarrow \cdot T$
3. As there is a **T** immediately to the right of a bookmark (i.e., $E \Rightarrow \cdot T$),
 so we add $T \Rightarrow \cdot T * F$ and $T \Rightarrow \cdot F$
4. $T \Rightarrow \cdot F$ forces us to add
 $F \Rightarrow \cdot (E)$ and $F \Rightarrow \cdot id$



Another LR(0) Closure Example

- Given the grammar as follows:

$$S \Rightarrow E \$$$

$$E \Rightarrow E + T \mid T$$

$$T \Rightarrow ID \mid (E)$$

CLOSURE($\{S \rightarrow \cdot E \$\}) =$

$\{ S \rightarrow \cdot E \$,$

$E \rightarrow \cdot E + T,$

$E \rightarrow \cdot T,$

$T \rightarrow \cdot ID,$

$T \rightarrow \cdot (E) \}$

The five items above forms an item set called **state s0**



Preliminaries - AdvanceDot

Be aware of the difference between state and item

- Consider a state s (an item set) and a grammar symbol X

```
function ADVANCEDOT( $state, X$ ) returns Set
    return ( $\{ A \rightarrow \alpha X \bullet \mid A \rightarrow \alpha \bullet X \beta \in state \}$ )
end
```

Figure 6.9: LR(0) construction.

- $AdvanceDot(s, X) = s'$
- $AdvanceDot(s, X)$ computes the next state s' (the item set reachable from s via X)
- $AdvanceDot(s, X)$ is defined to be
 - the closure of the set of all items $[A \rightarrow \alpha X \bullet \beta]$ such that $[A \rightarrow \alpha \bullet X \beta]$ is in s
 - This functions defines the transitions in the LR(0) automaton for a grammar



AdvanceDot Example

- Given the grammar, G
 - $E' \Rightarrow E$
 - $E \Rightarrow E + T \mid T$
 - $T \Rightarrow T * F \mid F$
 - $F \Rightarrow (E) \mid \text{id}$
- and the state s (item set)
 - s refers to $\{E \Rightarrow E \cdot + T\}$
- Please find **AdvanceDot($s, +$) = s'**



AdvanceDot Example (Cont'd)

- The next state s' should be:

$E \Rightarrow E + \cdot T$ (Move the bookmark one step)

$T \Rightarrow \cdot T * F$ (by closure)

$T \Rightarrow \cdot F$ (by closure)

$F \Rightarrow \cdot (E)$ (by closure)

$F \Rightarrow \cdot id$ (by closure)

- An example of shift from one state to another
 - Note state s' is comprised of the five items above
- Repeating the above steps, we can build all the states and construct the transition diagram for G



Preliminaries - ComputeGoto

- **ComputeGoto(States, s)**
 - is responsible for computing **all possible states reachable from s for all the grammar symbol X**
 - The parameter **States** is a global variable, recording all the states of the parse table
 - This function reflects the parser's progress after shifting across every item in this state s with X after the bookmark

- **ComputeGoto(States, s)** works as follows:

get all the items (*closed*) from the given state s (**Marker 17**)

for each grammar symbol X (**Marker 18**)

find all items (*RelevantItems*) reachable from the state s (*closed*) via X (**Marker 19**)

if *RelevantItems* is an empty set we **continue** the loop
add the items we found (*RelevantItems*) to the (new) state Y
set parse table entry $Table[s][X] = \text{shift } Y$ (**Marker 20**)

- All such items indicate transition to the same state since the parsers we construct must operate deterministically
 - In other words, the parse table has only one entry for a given state and symbol

```

function CLOSURE(state) returns Set
  ans ← state  ← Obtain the items of the given state
  repeat
    prev ← ans
    foreach A → α • B γ ∈ ans do
      foreach p ∈ PRODUCTIONSFOR(B) do
        ans ← ans ∪ { B → • RHS(p) }
    until ans = prev
  return (ans)
end
procedure COMPUTEGOTO(States, s)
  closed ← CLOSURE(s)
  foreach X ∈ (N ∪ Σ) do
    RelevantItems ← ADVANCEDOT(closed, X)
    if RelevantItems ≠ ∅
    then
      Table[s][X] ← shift ADDSTATE(States, RelevantItems)
  end

```

Figure 6.10: LR(0) closure and transitions.

```

function ADDSTATE(States, items) returns State
  if items ∉ States
  then
    s ← newState(items)
    States ← States ∪ { s }
    WorkList ← WorkList ∪ { s }
    Table[s][★] ← error
  else s ← FindState(items)
  return (s)
end
function ADVANCEDOT(state, X) returns Set
  return ( { A → α X • β | A → α • X β ∈ state } )
end

```

Figure 6.9: LR(0) construction.



Construction of LR(0) Parse Table I

- **ComputeLR0(*Grammar*)**
 - Given the grammar *G*, it returns the *States* and Start State *Start* for *G*
- **ComputeLR0(*Grammar*)** works as follows:

initialize table states *States*

find the **starting items** (*StarItems*) with the **Start** symbol of *G* (Marker 7)

add a new state *Start* with the *StarItems*

get the state *s* from the *WorkList* (Marker 8)

call **ComputeGoto**(*States*, *s*) to find the next states reachable from *s*

go to **Marker 8** until all of the states in *WorkList* has been processed

return the *States* and *Start*

```

function COMPUTELR0(Grammar) returns (Set, State)
    States  $\leftarrow \emptyset$ 
    StartItems  $\leftarrow \{ \text{Start} \rightarrow \bullet \text{ RHS}(p) \mid p \in \text{PRODUCTIONS\_FOR}(\text{Start}) \}$  ⑦
    StartState  $\leftarrow \text{ADDSTATE}(\text{States}, \text{StartItems})$ 
    while (s  $\leftarrow \text{WorkList.EXTRACTELEMENT}()$ )  $\neq \perp$  do ⑧
        call COMPUTEGOTO(States, s)
        return ((States, StartState))
    end
function ADDSTATE(States, items) returns State
    if items  $\notin \text{States}$  ⑨
        then
            s  $\leftarrow \text{newState}(\text{items})$  ⑩
            States  $\leftarrow \text{States} \cup \{s\}$ 
            WorkList  $\leftarrow \text{WorkList} \cup \{s\}$  ⑪  $\leftarrow$  Add the new state to the list
            Table[s][ $\star$ ]  $\leftarrow \text{error}$  ⑫
        else s  $\leftarrow \text{FindState}(\text{items})$ 
        return (s)
    end
function ADVANCEDOT(state, X) returns Set
    return ( $\{ A \rightarrow \alpha X \bullet \beta \mid A \rightarrow \alpha \bullet X \beta \in \text{state} \}$ ) ⑬
end
    
```

Figure 6.9: LR(0) construction.



The Transition Diagram for the Grammar (1/2)

- Fig. 6.11 shows the transitions among the states for the grammar in Fig. 6.2
 - Each state is shown as a separate box
- The **kernel** of state s
 - is the set of items explicitly represented in the state
 - E.g., **Start** $\rightarrow \cdot E \$$ in State 0
 - In States 0, 1, and 5, the horizontal line within a box (state) separates the **kernel** and **closure** items
 - In the other states, **no item contains a \cdot before a nonterminal**, so no closure items are indicated for those states, i.e., State 2, 3, 4, & 6
- Transitions
 - Next to each item in each state is the state number reached by shifting the symbol next to the item's bookmark
 - The transitions are also shown with labeled edges between the states

1 Start $\rightarrow E \$$
 2 E $\rightarrow \text{plus } E E$
 3 | num

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

Figure 6.2: Grammar and rightmost derivation of plus num num \$.

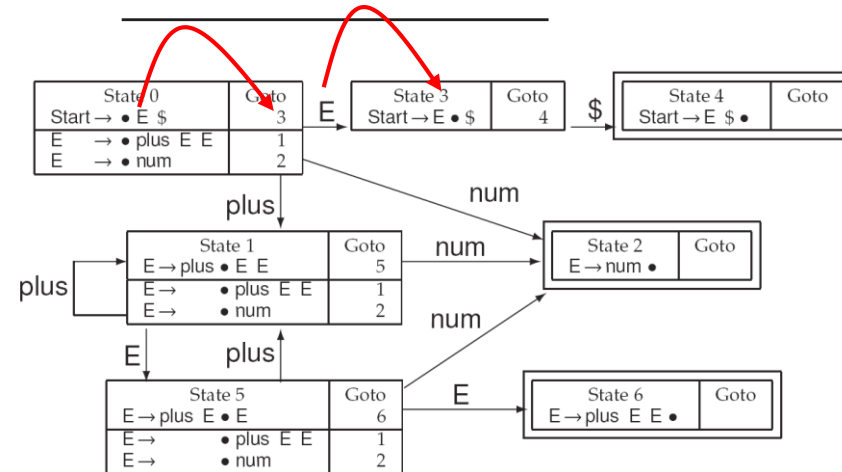


Figure 6.11: LR(0) computation for Figure 6.2, shown as a characteristic finite-state machine. State 0 is the initial state, and the double-boxed states are accept states.



The Transition Diagram for the Grammar (2/2)

- If a state contains a reducible item, then the state is **double-boxed**
- From the (double-boxed) states and edges, the basis for LR parsing is
 - A **deterministic finite automaton** (DFA)
 - called the **characteristic finite-state machine** (CFSM)
- Each transition
 - **shifts** the symbols of a valid sentential form
- When the automaton arrives in a double-boxed state
 - a **reduction** can be performed
- This process can be repeated until
 - the grammar's **goal symbol** is shifted (successful parse)
 - or the CFSM **blocks** (an input error)

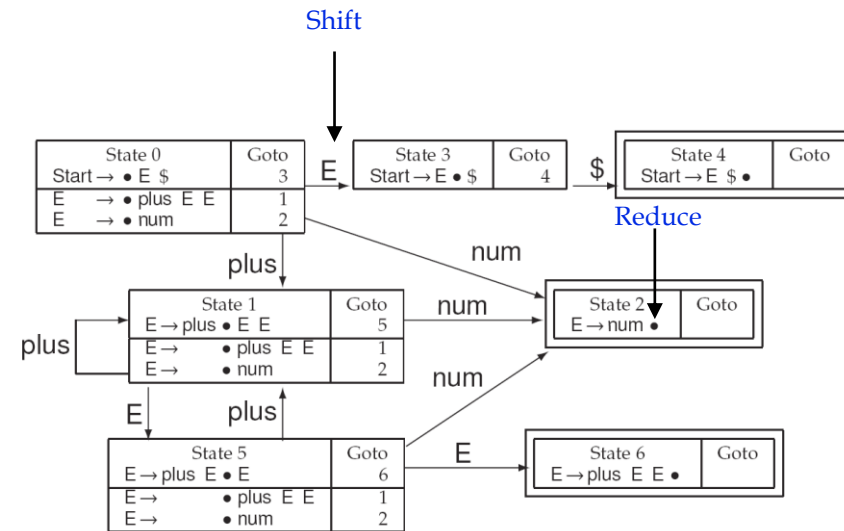


Figure 6.11: LR(0) computation for Figure 6.2, shown as a characteristic finite-state machine. State 0 is the initial state, and the double-boxed states are accept states.

NOTE:

- A **viable prefix** of a right sentential form is any prefix that does not extend beyond its handle
- The handle is the RHS of the (unique) reducible item in the state; an example is illustrated in Fig. 6.8



Construction of LR(0) Parse Table II (1/2)

- The decision to call for a **reduce** is reflected in the code of Fig. 6.14
 - Arrival in a **double-boxed state** signals a reduction **irrespective of the next input token**
- CompleteTable(Table, Grammar)**
 - Given the grammar G , and the *Table* constructed by **ComputeLRO(Grammar)**
- CompleteTable(Table, Grammar)** adds the reduce operations:

search for **the reducible items** from the states in the *Table* (Marker 1-3)

if the item of the rule r in the state s is **reducible** (i.e., $LHS(r) \rightarrow RHS(r) \cdot$) (Marker 8), then

call **AssertEntry** ($s, X, \text{reduce } r$) to add the entry in the *Table*

```

procedure COMPLETETABLE(Table, grammar)
    call COMPUTELOOKAHEAD()
    ① foreach state  $\in$  Table do
        ② foreach rule  $\in$  Productions(grammar) do
            ③ call TRYRULEINSTATE(state, rule)
            ④ call ASSERTENTRY(StartState, GoalSymbol, accept)
        end
    procedure ASSERTENTRY(state, symbol, action)
        ⑤ if Table[state][symbol] = error
            ⑥ then Table[state][symbol]  $\leftarrow$  action
        else
            ⑦ call REPORTCONFLICT(Table[state][symbol], action)
        end
    end
    
```

Figure 6.13: Completing an LR(0) parse table.

```

procedure COMPUTELOOKAHEAD()
    /* Reserved for the LALR(k) computation given in Section 6.5.2 */
    end
    procedure TRYRULEINSTATE(s, r)
        if  $LHS(r) \rightarrow RHS(r) \cdot \in s$  ← The items in double-checked boxes
        then
            ⑧ foreach  $X \in (\Sigma \cup N)$  do call ASSERTENTRY(s,  $X$ , reduce r)
        end
    end
    
```

Figure 6.14: LR(0) version of TRYRULEINSTATE.



Construction of LR(0) Parse Table II (2/2)

- As reduce actions are inserted,
 - AssertEntry** reports any **conflicts** that arise when a given state and grammar symbol call for multiple parsing actions (Marker 7)
 - Marker 6** allows an action to be asserted only if the relevant table cell was previously **undefined** (cells are initialized to the value of **error** at Marker 12 in Fig. 6.9)
- Finally, **Marker 4** establishes the State **accept**; Later, the parser calls for acceptance when the goal symbol is shifted in the table's start state
- Given the construction in Fig. 6.9 & 6.13 and the grammar in Figure 6.2, LR(0) analysis yields the parse table as shown in Fig. 6.15

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

```

procedure COMPLETETABLE(Table, grammar)
    call COMPUTELOOKAHEAD()
    ① foreach state ∈ Table do
        ② foreach rule ∈ Productions(grammar) do
            ③ call TRYRULEINSTATE(state, rule)
            ④ call ASSERTENTRY(StartState, GoalSymbol, accept)
        end
    procedure ASSERTENTRY(state, symbol, action)
        ⑤ if Table[state][symbol] = error
            ⑥ then Table[state][symbol] ← action
        else
            ⑦ call REPORTCONFLICT(Table[state][symbol], action)
        end
    
```

Figure 6.13: Completing an LR(0) parse table.

```

procedure COMPUTELOOKAHEAD()
    /* Reserved for the LALR(k) computation given in Section 6.5.2 */
    end
    procedure TRYRULEINSTATE(s, r)
        if LHS(r) → RHS(r) • ∈ s
        then
            ⑧ foreach X ∈ (Σ ∪ N) do call ASSERTENTRY(s, X, reduce r)
        end
    
```

Figure 6.14: LR(0) version of TRYRULEINSTATE.

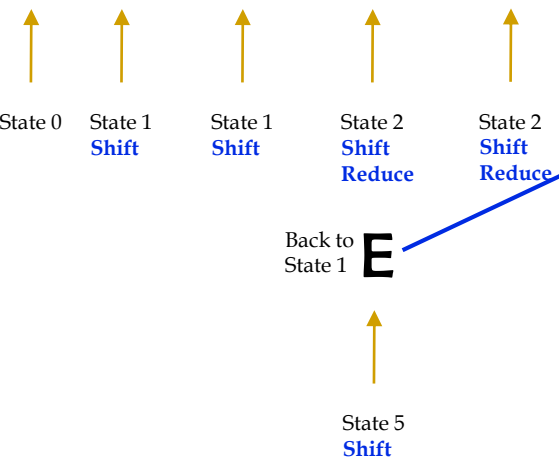
Figure 6.15: LR(0) parse table for the grammar in Figure 6.2.

Example of Processing the Input w/ the LR(0) Machine (1/4)



- The input string:
plus plus num num num \$

- Processing the input from the left
plus plus num num num \$



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

Figure 6.2: Grammar and rightmost derivation of plus num num \$.

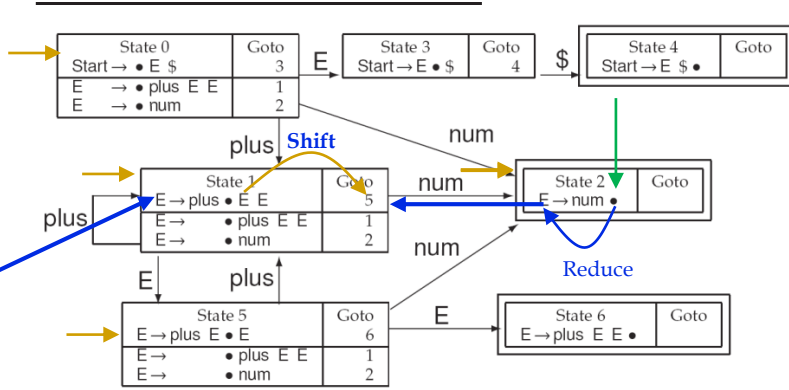


Figure 6.11: LR(0) computation for Figure 6.2, shown as a characteristic finite-state machine. State 0 is the initial state, and the double-boxed states are accept states.

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

Figure 6.12: Processing of plus plus num num num \$ by the LR(0) machine in Figure 6.11.

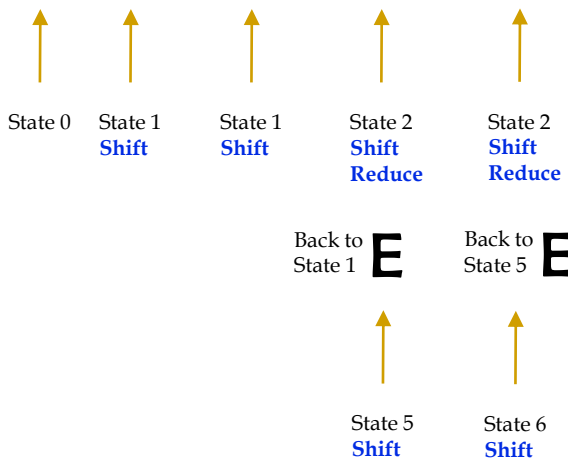
Example of Processing the Input w/ the LR(0) Machine (2/4)



- The input string:
plus plus num num num \$

- Processing the input from the left

plus plus num num num \$



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

Figure 6.2: Grammar and rightmost derivation of plus num num \$.

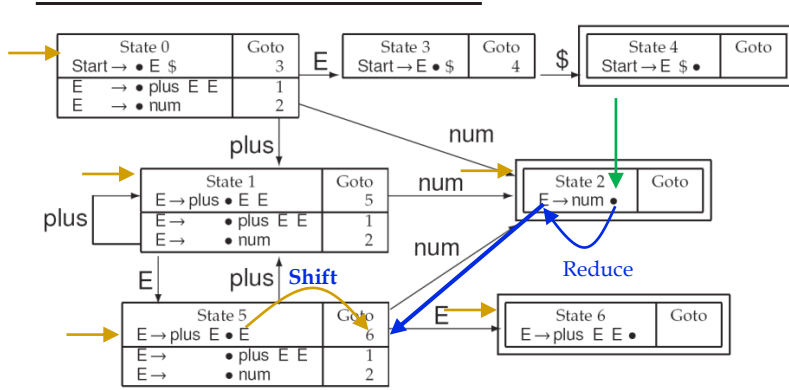


Figure 6.11: LR(0) computation for Figure 6.2, shown as a characteristic finite-state machine. State 0 is the initial state, and the double-boxed states are accept states.

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

Figure 6.12: Processing of plus plus num num num \$ by the LR(0) machine in Figure 6.11.

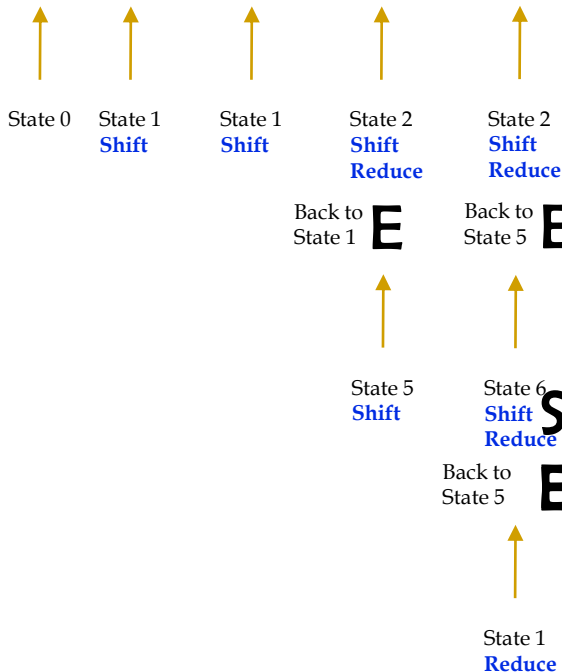
Example of Processing the Input w/ the LR(0) Machine (3/4)



- The input string:
plus plus num num num \$

- Processing the input from the left

plus plus num num num \$



- 1 Start $\rightarrow E \$$
- 2 E $\rightarrow \text{plus } E E$
- 3 \mid num

Figure 6.2: Grammar and rightmost derivation of plus num num \$.

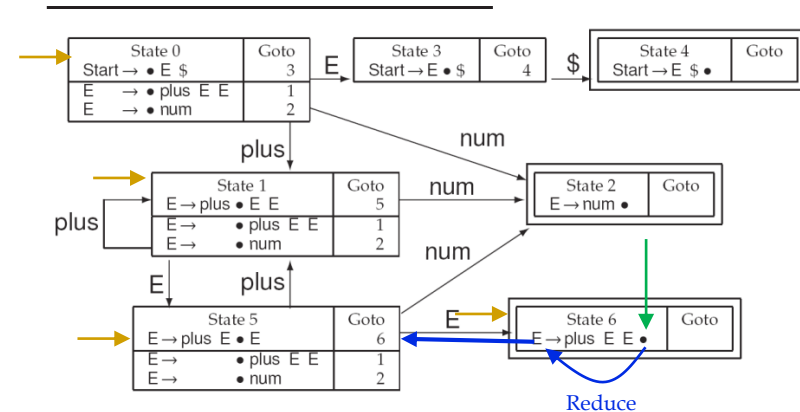


Figure 6.11: LR(0) computation for Figure 6.2, shown as a characteristic finite-state machine. State 0 is the initial state, and the double-boxed states are accept states.

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

Figure 6.12: Processing of plus plus num num num \$ by the LR(0) machine in Figure 6.11.

Example of Processing the Input w/ the LR(0) Machine (4/4)



- The input string:
plus plus num num num \$

- You should exercise the transitions in Fig. 6.11 on your own

- What is missing in the example in Fig. 6.12?

- Shift and **Reduce** operations are applied implicitly
- You could follow the **Knitting mechanism** introduced in the beginning of this chapter to derive the input string

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

Figure 6.2: Grammar and rightmost derivation of plus num num \$.

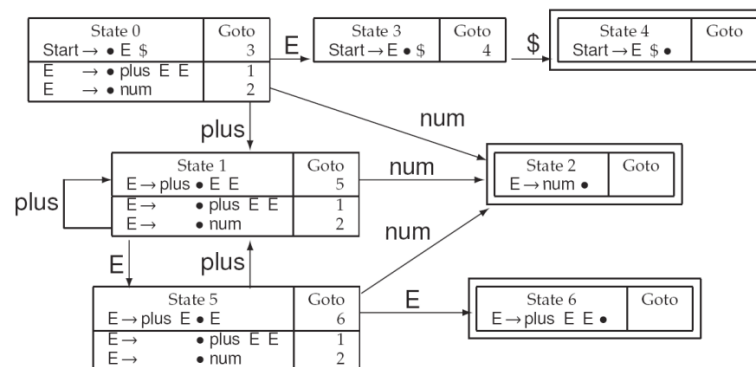


Figure 6.11: LR(0) computation for Figure 6.2, shown as a characteristic finite-state machine. State 0 is the initial state, and the double-boxed states are accept states.

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

Figure 6.12: Processing of plus plus num num num \$ by the LR(0) machine in Figure 6.11.



Conflict Diagnosis

- We consider table-construction methods,
 - which are more powerful than LR(0), thereby accommodating a much larger class of grammars
- A parse table **conflict** arises
 - when the table-construction method cannot decide between **multiple alternatives** for some table-cell entry
 - Then, the associated state (row of the parse table) is **inadequate** for that method
 - An inadequate state for a weaker table-construction algorithm can sometimes be resolved by **a stronger algorithm**



Conflict Diagnosis (Cont'd)

- Now, we examine why **conflicts** arise during LR table construction
 - and develops approaches for understanding and resolving such conflicts
- We use figures below to depict the **different characteristics** of
 - the parse tables for a weaker LR(0) (Fig. 6.15) and
 - a stronger method (Fig. 6.5)
 - E.g., the grammar of Fig. 6.4 is not LR(0) since a mix of shift and reduce actions can be seen in State 0 (shown in Fig. 6.5); simple reduce in State 2, 4, & 6 in LR(0) table in Fig. 6.15
 - Fortunately, the table-construction algorithms introduced in Sec. 6.5.1 resolve the LR(0) conflicts for the grammar

A mix of shift and reduce ops

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

Figure 6.5: Parse table for the grammar shown in Figure 6.4.

Simple reduce op

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

Figure 6.15: LR(0) parse table for the grammar in Figure 6.2.



What Are Those Conflicts?

- Two possible the conflict types during LR(k) parsing

1. 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**
 - The bookmark symbol must occur **before a terminal symbol** t in one of the state's items, so that a shift of t could be appropriate
 - The bookmark symbol must also occur **at the end of some other item**, so that a reduction in this state is also possible

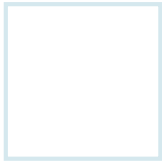
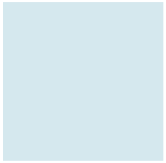
2. reduce/reduce conflicts exist

- when table construction cannot use the next k tokens to distinguish between **multiple reductions that could be applied in the inadequate state**
- Of course, a state with such a conflict must have **at least two reducible items**
 - Recall that State 2, 4, and 6 in Fig. 6.11 have single item for reduction



What Are Those Conflicts? (Cont'd)

- Other combinations of actions in a table cell do not make sense
- Example
 - It cannot be the case that some terminal t could be **shifted** but also cause an **error**
 - There cannot be a **shift/shift error**
 - A terminal symbol which might shift the current state to more than one target state
 - if a state admits the shifting of terminal symbols t and u , then the target state for the two shifts is different, and there is no conflict



Sources of the Conflicts I

- Two reasons for the arisen conflicts:

I. The grammar is **ambiguous**

- No (deterministic) table-construction method, e.g., LR(k), can resolve conflicts that arise due to ambiguity
- **If a grammar is ambiguous, then some input string has at least two distinct parse trees**
(Recall the ambiguity grammar definition in Sec. 4.2.2)
- A program specified in a computer language should have an unambiguous interpretation
- Handling ambiguous grammars is given in Sec. 6.4.1

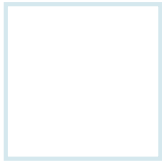
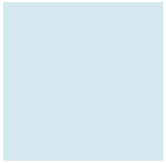


Sources of the Conflicts II

- Two reasons for the arisen conflicts:

II. The grammar is not ambiguous, but the current table-building approach **could not resolve the conflict** (Limitation of LR(k) gives an example)

- In this case, the **conflict might disappear** if one or more of the following approaches is taken:
 - a) The current table-construction method is given **more lookahead**
 - b) A **more powerful table-construction method** is used
 - It is possible that no amount of lookahead or table-building power can resolve the conflict, even if the grammar is unambiguous
 - We consider such a grammar in Sec. 6.4.2



Identify the Source(s) of Conflicts

- Given an inadequate state during the LR(k) construction
 - it is an unfortunate but important fact that **it is impossible to decide automatically which of the above problems afflicts the grammar**
- This follows from the impossibility of an algorithm to determine **if an arbitrary CFGs is ambiguous** [HU79, GJ79]
- It is therefore also impossible to determine generally whether a bounded amount of lookahead can resolve an inadequate state
- As a result, **human** (rather than mechanical) **reasoning** is required to understand and repair grammars for which conflicts arise
- Sec. 6.4.1 and 6.4.2 develop **intuition** and **strategies** for such reasoning



Conflicts Cased by Ambiguous Grammars

- Consider the grammar and its LR(0) construction shown in Fig. 6.16
- An inadequate state: **State 5**
 - A **plus** can be **shifted** to arrive in State 3
 - A **reduction** can also be applied, $E \rightarrow E \text{ plus } E$
- This inadequate state exhibits a **shift/reduce conflict** for LR(0)

1 Start $\rightarrow E \$$
 2 E $\rightarrow E \text{ plus } E$
 3 | num

State 0	
Start $\rightarrow \bullet E \$$	Goto 2
E $\rightarrow \bullet E \text{ plus } E$	2
E $\rightarrow \bullet \text{ num}$	1

State 1	Goto
E $\rightarrow \text{ num } \bullet$	

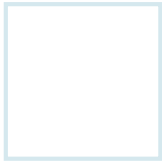
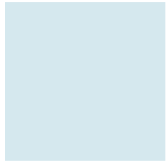
State 2	Goto
E $\rightarrow E \bullet \text{ plus } E$	3
Start $\rightarrow E \bullet \$$	4

State 3	Goto
E $\rightarrow E \text{ plus } \bullet E$	5
E $\rightarrow \bullet E \text{ plus } E$	5
E $\rightarrow \bullet \text{ num}$	1

State 4	Goto
Start $\rightarrow E \$ \bullet$	

State 5	Goto
E $\rightarrow E \text{ plus } E \bullet$	
E $\rightarrow E \bullet \text{ plus } E$	3

Figure 6.16: An ambiguous expression grammar.



Is the Grammar Ambiguous?

- While there is no automatic method for determining if an arbitrary grammar is ambiguous,
 - the **inadequate states** can provide valuable assistance in finding a **string with multiple derivations**
- The **bookmark** symbol shows the progress made thus far
 - Symbols appearing after the bookmark are symbols that can be shifted to make progress toward a successful parse

Approach

- While our ultimate goal is the **discovery of an input string with multiple derivations**
 - we begin by trying to find an **ambiguous sentential form**
 - Once identified, the sentential form can easily be **extended** into a terminal string by **replacing nonterminals using the grammar's productions**



Is the Grammar Ambiguous? (Cont'd)

- Using State 5 in Fig. 6.16 as an example, the steps taken to understand conflicts are as follows:
 - Using the parse table or CFSM, determine a sequence of vocabulary symbols that cause the parser to **move from the start state to the inadequate state**
 - The simplest such sequence is **E plus E**
 - which passes through States 0, 2, 3, and 5
 - In **State 5** we have **E plus E** on the top-of-stack
 - One option is a reduction by **E → E plus E •**
 - However, with the item **E → E • plus E**, it is also possible to shift a plus and then an E

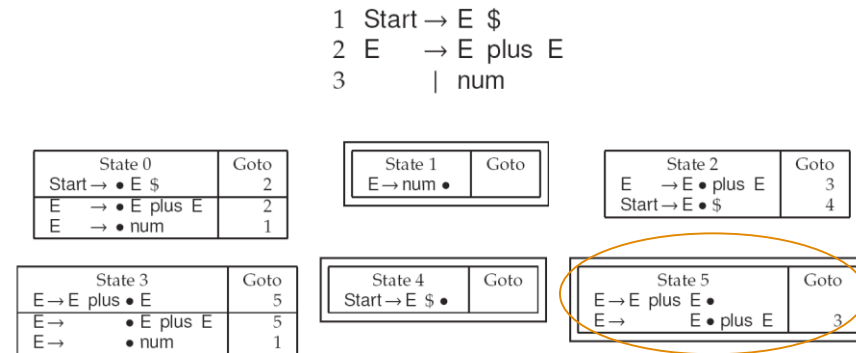


Figure 6.16: An ambiguous expression grammar.



Is the Grammar Ambiguous? (Cont'd)

- Using State 5 in Fig. 6.16 as an example, the steps taken to understand conflicts are as follows:

2. We **line up the dots of these two items**

- we obtain a snapshot of:
what is **on the stack** upon arrival in this state and
what may be successfully **shifted in the future**
- Here we obtain the sentential form prefix:
E plus E • plus E
- The shift/reduce conflict tells us that there are two potentially successful parses
 - We therefore try to construct **two derivation trees** for **E plus E plus E**, one assuming the **reduction** at the bookmark symbol and one assuming the **shift**
- Completing either derivation
 - may require extending this sentential prefix so that it becomes a sentential form:
 - a string of vocabulary symbols** derivable (in two different ways) from the goal symbol

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

State 0	Goto
Start \rightarrow • E \$	2
E \rightarrow • E plus E	2
E \rightarrow • num	1

State 1	Goto
E \rightarrow num •	

State 2	Goto
E \rightarrow E • plus E	3
Start \rightarrow E • \$	4

State 3	Goto
E \rightarrow E plus • E	5
E \rightarrow • E plus E	5
E \rightarrow • num	1

State 4	Goto
Start \rightarrow E \$ •	

State 5	Goto
E \rightarrow E plus E •	3
E \rightarrow E • plus E	3

Figure 6.16: An ambiguous expression grammar.

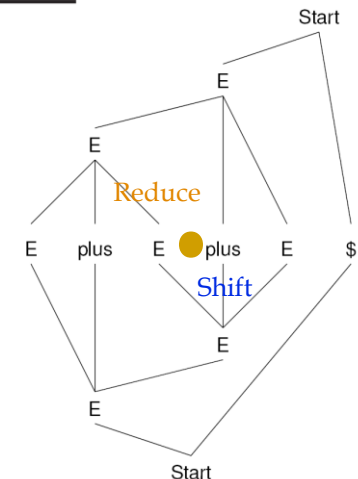


Figure 6.17: Two derivations for E plus E plus E \$. The parse tree on top favors reduction in State 5; the parse tree on bottom favors a shift.



Eliminate the Ambiguity

- Having analyzed the ambiguity in the grammar of Fig. 6.16,
 - we eliminate the ambiguity by **creating a grammar in Fig. 6.18**
- The grammars in Fig. 6.16 and 6.18 generate the same language
 - In fact, the language is regular, denoted by the regular expression: **num (plus num)* \$**
- We see that even simple languages can have ambiguous grammars
- In practice, diagnosing ambiguity can be more difficult**
- In particular, finding the ambiguous sentential form may require significant extensions

```

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

Figure 6.16: An ambiguous expression grammar.

```

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 •	

Figure 6.18: Unambiguous grammar for infix sums and its LR(0) construction.



Limitation of LR(k)

- This example shows:
 - the grammar in Fig. 6.19 is not **ambiguous**,
 - but the current LR(0) table-building approach **can not resolve the conflict**
- Fig. 6.19 shows
 - a grammar and
 - a portion of its LR(0) construction for a language similar to infix addition,
 - where expressions end in either **a** or **b**

```

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

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

State 2		Goto
E → num •		
F → num •		

Figure 6.19: A grammar that is not LR(k).



Limitation of LR(k): Not Ambiguous Grammar

- State 2 contains a **reduce/reduce conflict**
 - LR(0) fails to handle the grammar
 - It is not clear whether **num** should be reduced to an **E** or an **F**
 - The viable prefix (possible combinations of prefix) that takes us to State 2 is simply **num**
- To obtain a sentential form,
 - which could lead to **State 2** from State 0,
 - this must be extended either to **num a \$** or **num b \$**
- Could we obtain **more than one derivation**?
 - If we use the **former sentential form**, then **F** cannot be involved in the derivation
 - Similarly, if we use **the latter sentential form**, **E** is not involved
 - We cannot; progress past **num** cannot involve more than one derivation
 - it means it is **not ambiguous grammar**

```

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

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

State 2	Goto
E → num •	
F → num •	

Figure 6.19: A grammar that is not LR(k).



Limitation of LR(k): LR(k) also Fails (1/3)

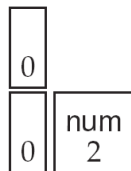
- Does a more ambitious table-construction method work?
 - Since LR(0) construction failed for the grammar in Fig. 6.19
 - we could try a more ambitious table-construction method from among those discussed in Sec. 6.5.1, 6.5.2, and 6.5.4
- It turns out that **none can succeed**
 - All LR(k) constructions analyze grammars using k lookahead symbols
 - If a grammar is LR(k), then there is **some value of k** ,
 - for which **all states are adequate in the LR(k)** construction described in Sec. 6.5.4



Limitation of LR(k): LR(k) also Fails (2/3)

- The grammar in Fig. 6.19 is not LR(k) for any k
 - To see this, consider the following rightmost derivation of a sufficiently long string:
num plus ... plus num a
 - A bottom-up parse must play the above derivation backwards
 - Thus, the first few steps of the parse will be as follows

$\text{Start} \Rightarrow_{\text{rm}} \text{Exprs } \$$
 $\Rightarrow_{\text{rm}} E a \$$
 $\Rightarrow_{\text{rm}} E \text{ plus num } a \$$
 $\Rightarrow_{\text{rm}} E \text{ plus ... plus num } a \$$
 $\Rightarrow_{\text{rm}} \text{num plus ... plus num } a \$$



Initial Configuration

num plus ... plus num a \$

shift num

plus ... plus num a \$



Limitation of LR(k): LR(k) also Fails (3/3)

- We are in **State 2** as **num** is on top-of-stack
- A deterministic, bottom-up parser must decide at this point
 - whether **to reduce num to an E or an F**
 - which we cannot do at this point
- If the decision were **delayed**,
 - then the reduction would have to take place in the middle of the stack, and this is not allowed
- To resolve the reduce/reduce conflict,
 - parser should know the symbol appears just before the **\$** symbol, i.e., **a** or **b**
- Unfortunately, the relevant **a** or **b** could be **arbitrarily far ahead** in the input,
 - because strings derived from **E** or **F** can be arbitrarily long

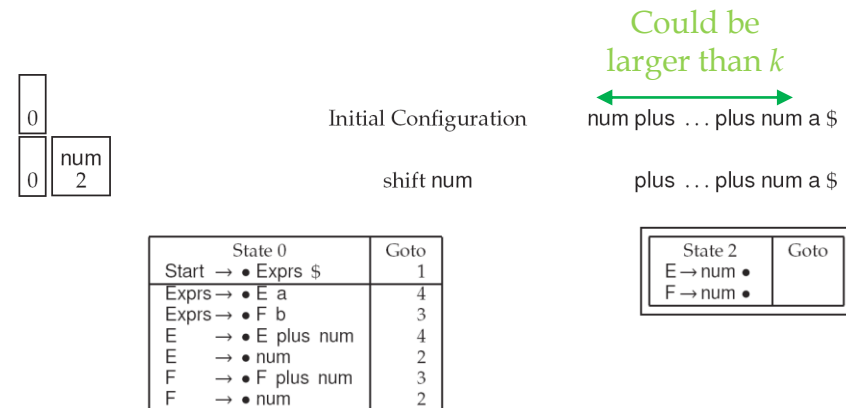


Figure 6.19: A grammar that is not LR(k).



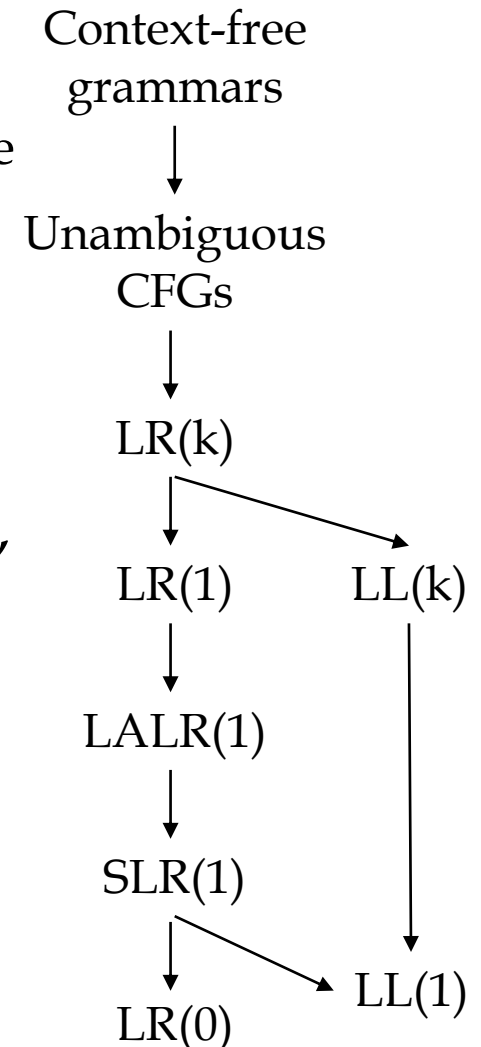
Summary of the Conflicts

- Simple grammars and languages can have subtle problems
 - that prohibit generation of a bottom-up parser
 - It follows that a top-down parser would also fail on such grammars
- The LR(0) construction can provide **important clues** for diagnosing a grammar's inadequacies
 - understanding and resolving such conflicts requires **human intelligence**
 - Also, the LR(0) construction forms the basis of the more advanced constructions considered next



Conflict Resolution and Table Construction

- While LR(0) construction succeeded for the grammar in Fig. 6.18,
 - most grammars require some lookahead during table construction
- Inclusion hierarchy for the context-free grammars is illustrated on the right
- The advanced parse table construction methods,
 - while based on the LR(0) construction, use increasingly sophisticated **lookahead** techniques to resolve conflicts
- If you are interested in the advanced methods,
 - you could find the related information by yourself
 - We are not going to cover them all, given the course schedule





SLR(k) Table Construction

- SLR(k) method
 - Simple LR with k tokens of lookahead
 - attempts to resolve inadequate states
- To demonstrate the SLR(k) construction
 - We begin by extending the grammar in Fig. 6.18 to accommodate expressions involving **sums and products**
 - Fig. 6.20 shows such a grammar along with a parse tree for the string: **num plus num times num \$**

```

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

Figure 6.18: Unambiguous grammar for infix sums and its LR(0) construction.

```

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

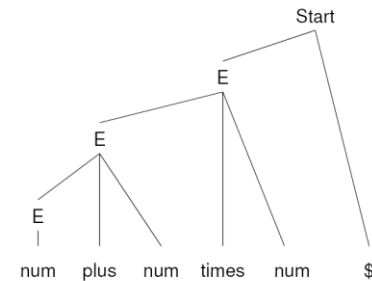


Figure 6.20: Expressions with sums and products.



Awareness of Operator Precedence

- The parse tree shown in Fig. 6.20 structures the computation by
 - adding the first two **nums** and then
 - multiplying that sum by the third **num**
 - As such, the input string **3 + 4 * 7** would produce a value of **49**
 - if evaluation were guided by the computation's parse tree

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

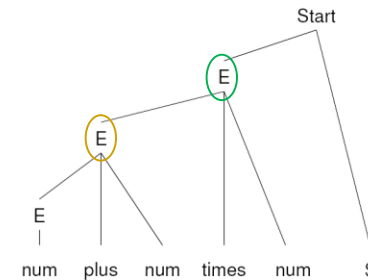


Figure 6.20: Expressions with sums and products.



Awareness of Operator Precedence (Cont'd)

- A common convention in mathematics is that **multiplication has precedence over addition**
 - Thus, the computation $3 + 4 * 7$ should be viewed as adding 3 to the product $4 * 7$, resulting in the value 31
 - Such conventions are typically adopted in programming language design, in an effort to simplify program authoring and readability
- We therefore seek a parse tree
 - that appropriately structures expressions involving multiplication and addition



Awareness of Operator Precedence (Cont'd)

- To achieve the desired effect, we first observe that
 - a string in the language of Fig. 6.20 should be regarded as a **sum of products**
 - The grammar in Fig. 6.18 generates **sums of nums**
 - A common technique to expand a language involves
 - replacing a **terminal symbol** in the grammar by a **nonterminal** whose role in the grammar is equivalent
 - To produce a **sum of Ts** rather than a sum of **nums**,
- We need only **replace num with T** to obtain the rules for **E** shown in Fig. 6.21

```

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

Figure 6.20: Expressions with sums and products.

```

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

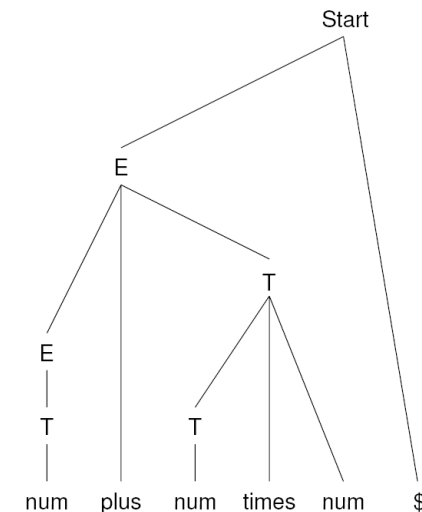


Figure 6.21: Grammar for sums of products.



Awareness of Operator Precedence (Cont'd)

- To achieve **a sum of products**,
 - each **T** can now derive a product, with the simplest product consisting of a single **num**
 - Thus, the rules for **T** are based on the rules for **E**, substituting **times** for **plus**
- Fig. 6.21 shows a parse tree for the input string from Fig. 6.20,
 - with multiplication having precedence over addition

```

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

Figure 6.20: Expressions with sums and products.

```

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

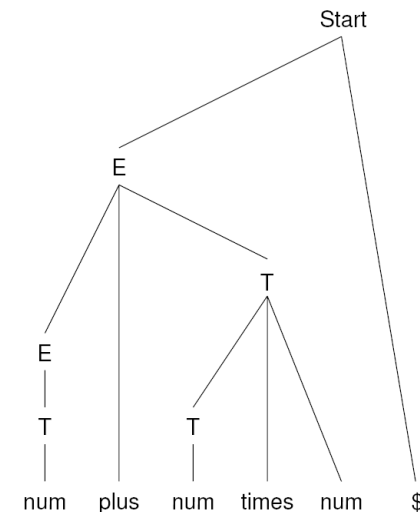


Figure 6.21: Grammar for sums of products.

LR(0) for the Precedence-Aware Grammar? (1/3)

- Fig. 6.22 shows a portion of the LR(0) construction for precedence respecting grammar
- Fig. 6.22**
 - shows a sequence of **parser actions** for the sentential form:
 - E plus num times num \$**, which blocks in **State 6**
- In fact, **States 1 and 6** are inadequate for LR(0)
 - because in each of these states, there is the possibility of **shifting a times** or **applying a reduction to E**

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

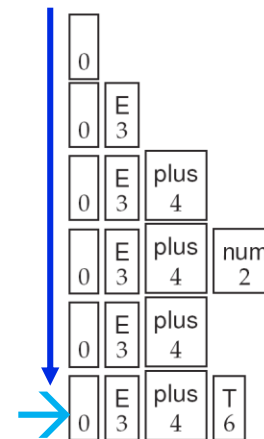
State 4		Goto
E → E plus • T		6
T → • T times num		6
T → • num		2

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 •		



Initial Configuration	E plus num times num \$
shift E	plus num times num \$
shift plus	num times num \$
shift num	times num \$
Reduce num to T	T times num \$
shift T	times num \$

Figure 6.22: LR(0) construction and parse leading to inadequate State 6.

LR(0) for the Precedence-Aware Grammar? (2/3)

- We first determine if the grammar in Fig. 6.21 is **ambiguous**
- We turn to the methods described in Sec. 6.4 (Is the Grammar Ambiguous?)
 - by analyzing the sentential form: **E plus T • times num \$** (obtained from merging the items in State 6)
 - In particular, we check if we can have more than one derivation from **State 6**
- The possible **shift** and **reduce** operations when we're at **State 6** are listed as below

- What to **shift**?
→ Shift **times** into stack
(E plus T • times num \$)
- What to **reduce**?
→ Reduce by Rule 2 $E \rightarrow E \text{ plus } T$
(E plus T • times num \$)

```

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

State 2	Goto
T → num •	

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

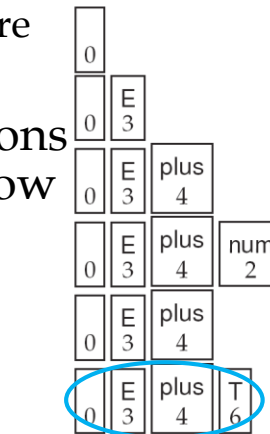
State 4	Goto
E → E plus • T	6
T → • T times num	6
T → • num	2

State 5	Goto
Start → E \$ •	

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

State 7	Goto
T → T times • num	8

State 8	Goto
T → T times num •	



Initial Configuration	E plus num times num \$
shift E	plus num times num \$
shift plus	num times num \$
shift num	times num \$
Reduce num to T	T times num \$
shift T	times num \$

Figure 6.22: LR(0) construction and parse leading to inadequate State 6.

LR(0) for the Precedence-Aware Grammar? (3/3)

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

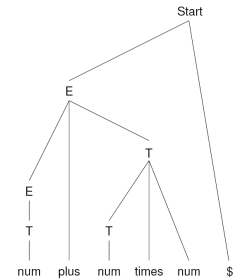
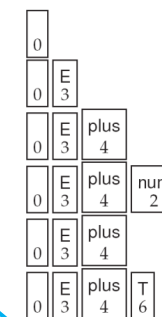


Figure 6.21: Grammar for sums of products.

<table> <tr><td>State 0</td><td>Goto</td></tr> <tr><td>Start \rightarrow • E \$</td><td>3</td></tr> <tr><td>E \rightarrow • E plus T</td><td>3</td></tr> <tr><td>E \rightarrow • T</td><td>1</td></tr> <tr><td>T \rightarrow • T times num</td><td>1</td></tr> <tr><td>T \rightarrow • num</td><td>2</td></tr> </table>	State 0	Goto	Start \rightarrow • E \$	3	E \rightarrow • E plus T	3	E \rightarrow • T	1	T \rightarrow • T times num	1	T \rightarrow • num	2	<table> <tr><td>State 1</td><td>Goto</td></tr> <tr><td>E \rightarrow T •</td><td>3</td></tr> <tr><td>T \rightarrow T • times num</td><td>7</td></tr> </table>	State 1	Goto	E \rightarrow T •	3	T \rightarrow T • times num	7
State 0	Goto																		
Start \rightarrow • E \$	3																		
E \rightarrow • E plus T	3																		
E \rightarrow • T	1																		
T \rightarrow • T times num	1																		
T \rightarrow • num	2																		
State 1	Goto																		
E \rightarrow T •	3																		
T \rightarrow T • times num	7																		
<table> <tr><td>State 2</td><td>Goto</td></tr> <tr><td>T \rightarrow num •</td><td></td></tr> </table>	State 2	Goto	T \rightarrow num •		<table> <tr><td>State 3</td><td>Goto</td></tr> <tr><td>Start \rightarrow E • \$</td><td>5</td></tr> <tr><td>E \rightarrow E • plus T</td><td>4</td></tr> </table>	State 3	Goto	Start \rightarrow E • \$	5	E \rightarrow E • plus T	4								
State 2	Goto																		
T \rightarrow num •																			
State 3	Goto																		
Start \rightarrow E • \$	5																		
E \rightarrow E • plus T	4																		
<table> <tr><td>State 4</td><td>Goto</td></tr> <tr><td>E \rightarrow E plus • T</td><td>6</td></tr> <tr><td>T \rightarrow • T times num</td><td>6</td></tr> <tr><td>T \rightarrow • num</td><td>2</td></tr> </table>	State 4	Goto	E \rightarrow E plus • T	6	T \rightarrow • T times num	6	T \rightarrow • num	2	<table> <tr><td>State 5</td><td>Goto</td></tr> <tr><td>Start \rightarrow E • \$</td><td></td></tr> </table>	State 5	Goto	Start \rightarrow E • \$							
State 4	Goto																		
E \rightarrow E plus • T	6																		
T \rightarrow • T times num	6																		
T \rightarrow • num	2																		
State 5	Goto																		
Start \rightarrow E • \$																			
<table> <tr><td>State 6</td><td>Goto</td></tr> <tr><td>E \rightarrow E plus T •</td><td>7</td></tr> <tr><td>T \rightarrow T • times num</td><td></td></tr> </table>	State 6	Goto	E \rightarrow E plus T •	7	T \rightarrow T • times num		<table> <tr><td>State 7</td><td>Goto</td></tr> <tr><td>T \rightarrow T times • num</td><td>8</td></tr> </table>	State 7	Goto	T \rightarrow T times • num	8								
State 6	Goto																		
E \rightarrow E plus T •	7																		
T \rightarrow T • times num																			
State 7	Goto																		
T \rightarrow T times • num	8																		
<table> <tr><td>State 8</td><td>Goto</td></tr> <tr><td>T \rightarrow T times num •</td><td></td></tr> </table>	State 8	Goto	T \rightarrow T times num •																
State 8	Goto																		
T \rightarrow T times num •																			



Initial Configuration E plus num times num \$

shift E plus num times num \$

shift plus num times num \$

shift num times num \$

Reduce num to T T times num \$

shift T times num \$

 times num \$

Figure 6.22: LR(0) construction and parse leading to inadequate State 6.

- The sentential form: **E plus T • times num \$**
- If the **shift** is taken (i.e., **E plus T times • num \$**),
 - then we can continue the parse in Fig. 6.22 to obtain the parse tree shown in Fig. 6.21
- If the **reduction** is taken
 - by Rule 2 **E \rightarrow E plus T** for the sentential form: **E plus T • times num \$**,
 - which yields **E • times num \$** after performing reduction by Rule 2 and shifting **E** to the stack
 - which causes the CFSM in Fig. 6.22 to block in **State 3** with no progress possible, i.e., impossible to handle **times** in the next token in input

LR(0) for the Precedence-Aware Grammar? (3/3)

- The sentential form: **E plus T · times num \$**

- If the **shift** is taken (i.e., **E plus T times · num \$**),
 - then we can continue the parse in Fig. 6.22 to obtain the parse tree shown in Fig. 6.21

- If the **reduction** is taken

- by Rule 2 **E → E plus T** (**E plus T · times num \$**),
- We obtain **E · times num \$** at **State 3**
- If we insist to further shift **times** and **num** into stack regardless of the transitions listed in the diagram,
- we obtain **E times num · \$** and
- we obtain **E times T · \$** after applying Rule 5 **T → num**
- we further obtain **E times E · \$** after applying Rule 3 **E → T**
- The parsing will fail at here since we don't have rule for **it**

- Thus, a reduction in **State 6** for this sentential form is inappropriate
 - Neither of the two phrases (**E times num \$** and **E times E \$**) can be further reduced to the goal symbol
- Also, Fig. 6.21 is not an ambiguous grammar since we have one possible derivation

```

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

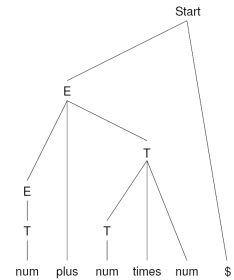


Figure 6.21: Grammar for sums of products.

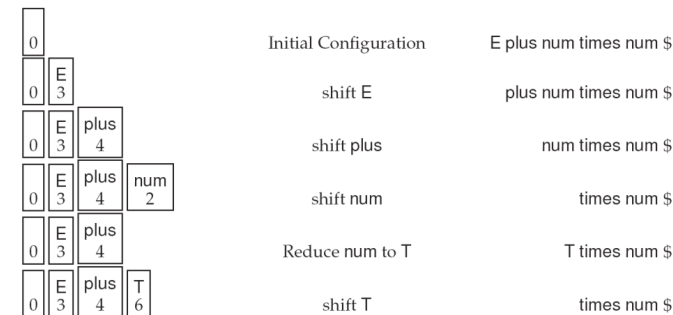
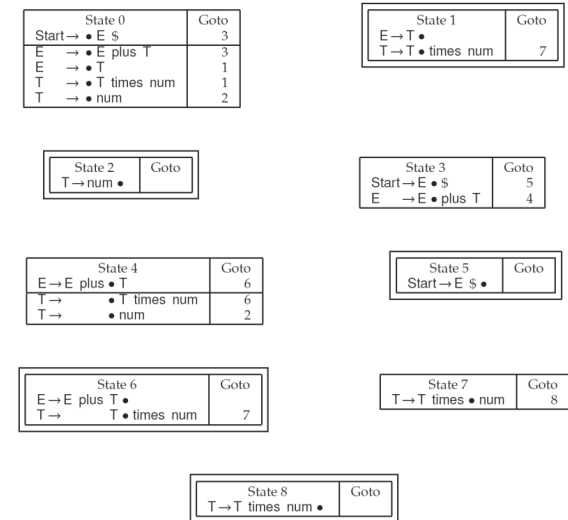


Figure 6.22: LR(0) construction and parse leading to inadequate State 6.



Further Discussion of the Grammar

- With the item $E \rightarrow E \text{ plus } T \bullet$ in State 6,
 - reduction by $E \rightarrow E \text{ plus } T$ must be appropriate **under some conditions**
 - In previous slide, we know that $E \rightarrow E \text{ plus } T \text{ times}$ is not working
 - What if we have other terminals after $E \rightarrow E \text{ plus } T$?
- For example, if we examine the sentential forms:
 - $E \text{ plus } T \$$ and $E \text{ plus } T \text{ plus num } \$$
 - We found that the $E \rightarrow E \text{ plus } T \bullet$ can be reduced in State 6 and
 - the parsing can go on if the next input symbol is **plus** or **\$**, but not **times**
- Specifically, we consider the sequence of parser actions
 - that could be applied between a reduction by $E \rightarrow E \text{ plus } T$ and the next shift of a terminal symbol
 - 1) **E** must be shifted onto the stack following the reduction
 - 2) At this point, assume terminal symbol **plus** is the next input symbol
 - If the **reduction** to **E** can lead to a successful parse, then **plus** can appear next to **E** in **some valid sentential form**



Further Discussion of the Grammar (Cont'd)

- What do we have so far?
 - E plus T • **times** num \$ **Shift** could be applied
 - E plus T • **plus** num \$ **Reduce** could be applied
- The action is context (**next token**) dependent

- **Nevertheless, LR(0) could not selectively call for a reduction in any state** (see Fig. 6.15)
 - however, methods that can **consult lookahead** information in **TryRuleInState** can resolve this conflict

Only reduction is allowed in State 2, 4, and 6

State	num	plus	\$	Start	E
0	<input type="text" value="2"/>	<input type="text" value="1"/>		accept	<input type="text" value="3"/>
1	<input type="text" value="2"/>	<input type="text" value="1"/>			<input type="text" value="5"/>
2	reduce 3				
3			<input type="text" value="4"/>		
4	reduce 1				
5	<input type="text" value="2"/>	<input type="text" value="1"/>			<input type="text" value="6"/>
6	reduce 2				

Figure 6.15: LR(0) parse table for the grammar in Figure 6.2.



Lookahead Information

- The shift/reduce action is context (**next token**) dependent
 - If the reduction to **E** can lead to a successful parse, then **plus** can appear next to **E** in **some valid sentential form**,
 - which could be checked by the **FOLLOW** information: **plus** \in **Follow(E)**
- SLR(k) parsing uses **Follow_k(A)** to determine if we should
 - call for a reduction to **A** in any state containing a reducible item for **A**
- Key idea:
 - A handle (RHS) should NOT be reduced to **N**, if the look ahead token is NOT in **Follow_k(N)**



SLR(1)

- SLR(1) has the same transition as LR(0)
- SLR(1) parse table construction is similar to LR(0)'s
 - But, with different parse table (finer-grained actions)
- Specifically, we obtain SLR(1) by
 - performing the LR(0) construction in Fig. 6.9;
 - the only change is to the method **TryRuleInState**, whose SLR(1) version is shown in Fig 6.23



SLR(1) Table Construction

- **TryRuleInState(s, r)**
 - Given the state s and the rule r , it adds the reduction op to the table entry $[s][X]$ when X is within the **FOLLOW(LHS(r))**
- In the previous example,
 - **States 1 and 6** are resolved by computing **Follow(E) = { plus, \$ }**
- The SLR(1) parse table that results from this analysis is shown in Figure 6.24

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

```

procedure TRYRULEINSTATE( $s, r$ )
  if LHS( $r$ )  $\rightarrow$  RHS( $r$ )  $\bullet \in s$ 
  then
    foreach  $X \in (\Sigma \cup N)$  do call ASSERTENTRY( $s, X, \text{reduce } r$ )
  end
  
```

Figure 6.14: LR(0) version of TRYRULEINSTATE.

```

procedure TRYRULEINSTATE( $s, r$ )
  if LHS( $r$ )  $\rightarrow$  RHS( $r$ )  $\bullet \in s$  For items in double-checked boxes
  then
    foreach  $X \in \text{Follow}(\text{LHS}(r))$  do
      call ASSERTENTRY( $s, X, \text{reduce } r$ )
  end
  
```

Figure 6.23: SLR(1) version of TRYRULEINSTATE.

SLR(1) Table Construction (Cont'd)

- In the previous example,
 - States 1** and **6** are resolved by computing $\text{Follow}(E) = \{ \text{plus}, \$ \}$

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

Figure 6.21: Grammar for sums of products.

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

Figure 6.24: SLR(1) parse table for the grammar in Figure 6.21.

	
State 0	State 1
Goto	Goto
3	7

Rule 3

State 2	Goto
T \rightarrow num •	

State 3	Goto
Start \rightarrow E • \$	5
E \rightarrow E • plus T	4

State 4	Goto
E \rightarrow E plus • T	6
T \rightarrow • T times num	6
T \rightarrow • num	2

State 5	Goto
Start \rightarrow E \$ •	

Rule 2

State 6	Goto
E \rightarrow E plus T •	7
T \rightarrow T • times num	

State 7	Goto
T \rightarrow T times • num	8

State 8	Goto
T \rightarrow T times num •	

0				
0	E			
0	E	plus		
0	E	plus	num	
0	E	plus	4	2
0	E	plus	4	
0	E	plus	4	T

Initial Configuration

E plus num times num \$

shift E

plus num times num \$

shift plus

num times num \$

shift num

times num \$

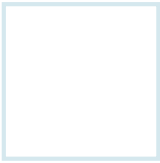
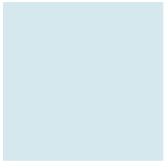
Reduce num to T

T times num \$

shift T

times num \$

Figure 6.22: LR(0) construction and parse leading to inadequate State 6.



QUESTIONS?