

COMPILER CONSTRUCTION

Bottom-Up Parsing

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Spring 2020

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
- Shift-Reduce Parsers
- LR Parsers
- Conflict Diagnose
- Conflict Resolution



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	$\rightarrow$	if Expr then StmtList endif
2		|	if Expr then StmtList else StmtList endif
3	StmtList	$\rightarrow$	StmtList ; Stmt
4		|	Stmt
5	Expr	$\rightarrow$	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 V_1
2	V_1	→	endif
3		|	else StmtList endif
4	StmtList	→	X Y
5	X	→	Stmt
6	Y	→	; Stmt Y
7		|	λ
8	Expr	→	var V_2
9	V_2	→	+ 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

Parsing and Derivation

- 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

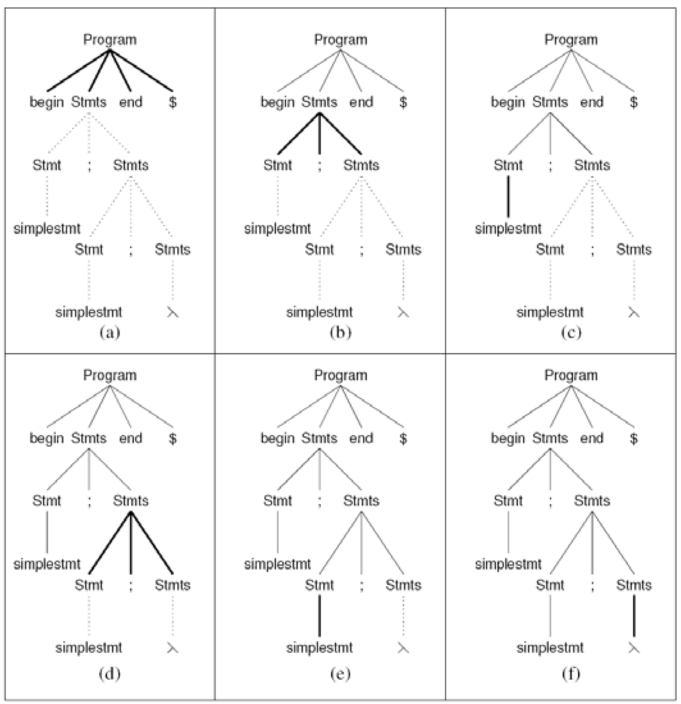


Figure 4.5: Parse of "begin simplestmt ; simplestmt ; end $" using the top-down technique. Legend explained on page 126.

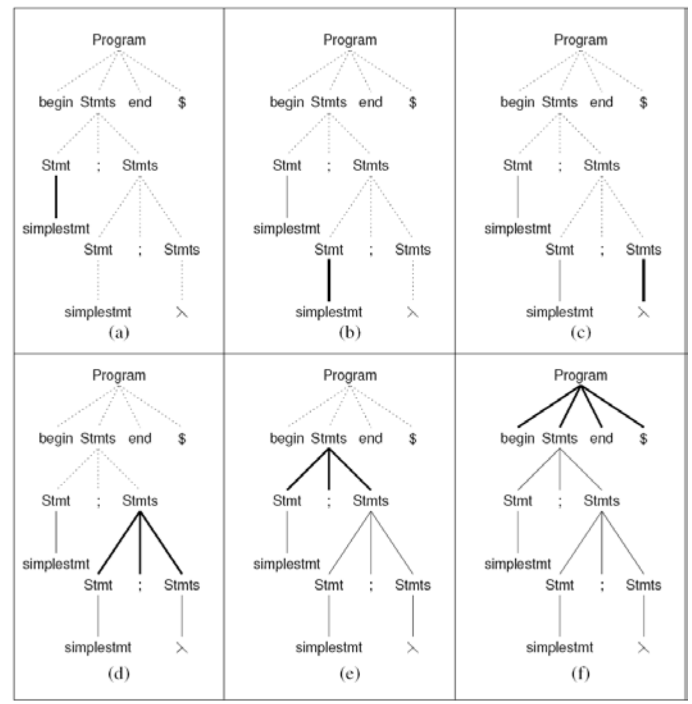


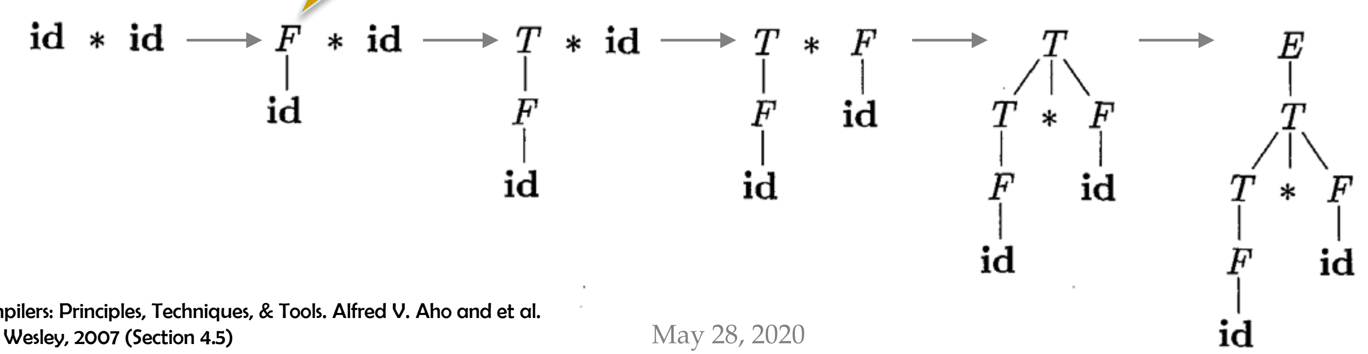
Figure 4.6: Parse of "begin simplestmt ; simplestmt ; end $" using the bottom-up technique. Legend explained on page 126.

An Illustration of Bottom-Up Parsing

- Given the grammar:
  - E ⇒ T
  - T ⇒ T * F
  - T ⇒ F
  - F ⇒ **id**

Reduce!!! Parser gets an terminal **id**, where the LHS of the matched rule for **id** is used to create a new node, **F**.

- The **bottom-up parsing** for the input string:
  - **id * id**



*Courtesy of Compilers: Principles, Techniques, & Tools. Alfred V. Aho and et al. Pearson/Addison Wesley, 2007 (Section 4.5)

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

- Given the grammar:
  - E ⇒ T
  - T ⇒ T * F
  - T ⇒ F
  - F ⇒ **id**
- The **rightmost derivation** for the input string:
  - **id * id**

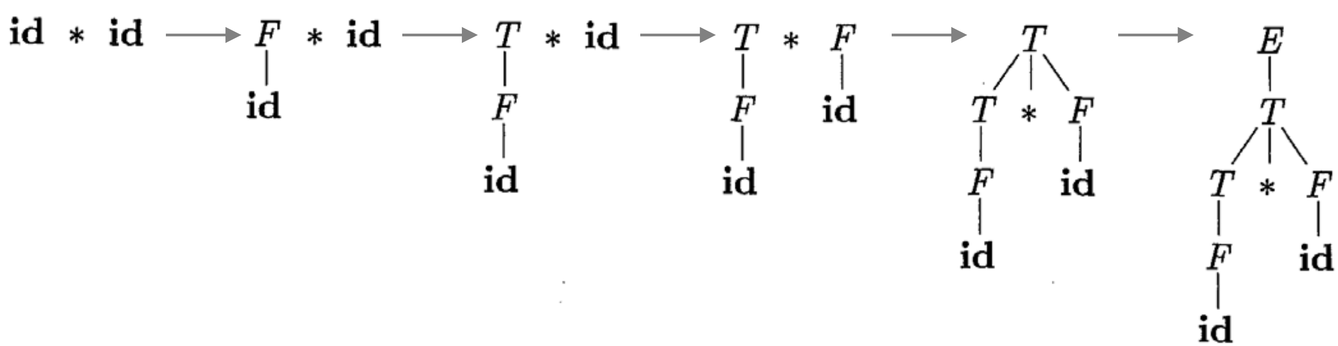
$$
\begin{aligned}
E &\Rightarrow_{\mathrm{rm}} T \\
&\Rightarrow_{\mathrm{rm}} T * F \\
&\Rightarrow_{\mathrm{rm}} T * \mathbf{id} \\
&\Rightarrow_{\mathrm{rm}} F * \mathbf{id} \\
&\Rightarrow_{\mathrm{rm}} \mathbf{id} * \mathbf{id}
\end{aligned}
$$

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

- Given the grammar:
  - E ⇒ T
  - T ⇒ T * F
  - T ⇒ F
  - F ⇒ **id**

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

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

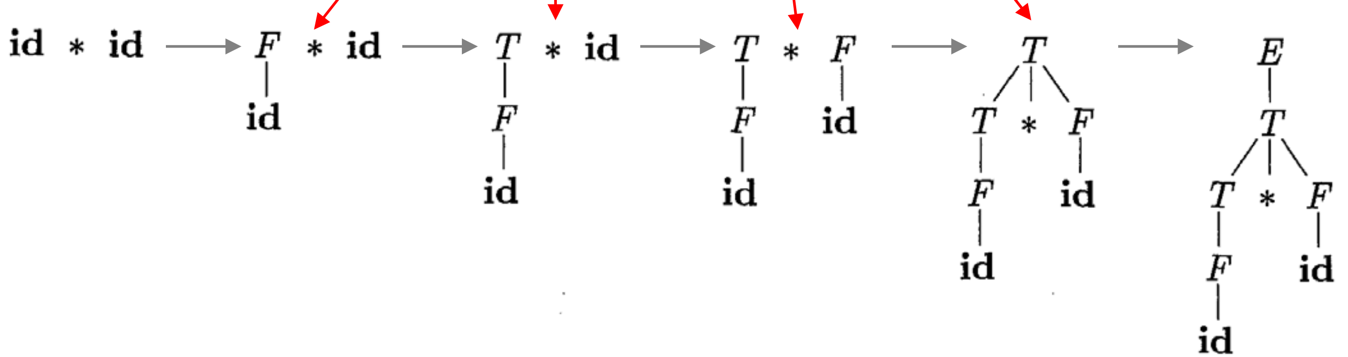




An Illustration of Bottom-Up Parsing (Cont'd)

- Given the grammar:
  - E ⇒ T
  - T ⇒ T * F
  - T ⇒ F
  - F ⇒ **id**

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





Bottom-Up Parsers

- The style of parsing 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**

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 $\mathbf{id}_1 * \mathbf{id}_2$
- 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 ⇒ T
T ⇒ T * F
T ⇒ F
F ⇒ **id**

The input string:

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

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

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

*Courtesy of Compilers: Principles, Techniques, & Tools. Alfred V. Aho and et al. Pearson/Addison Wesley, 2007 (Fig. 4.28 in Section 4.5)



*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 **$\mathbf{id_1}$ *$\mathbf{id_2}$**
  - 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 ⇒ T
T ⇒ T * F
T ⇒ F
F ⇒ **id**

The input string:

$\mathbf{id_1}$ * $\mathbf{id_2}$

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

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

*Courtesy of Compilers: Principles, Techniques, & Tools. Alfred V. Aho and et al. Pearson/Addison Wesley, 2007 (Fig. 4.28 in Section 4.5)

Revisiting 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	$\rightarrow$	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)

- A bottom-up (LR) **parsing**
  - 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→num**
    - This rule is applied again to obtain **plus E E $**
  - The rule is then reduced by **E→plus E E** to obtain **E $**
  - This can then be reduced by **Rule 1 to the goal symbol Start**

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)

- **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 $

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

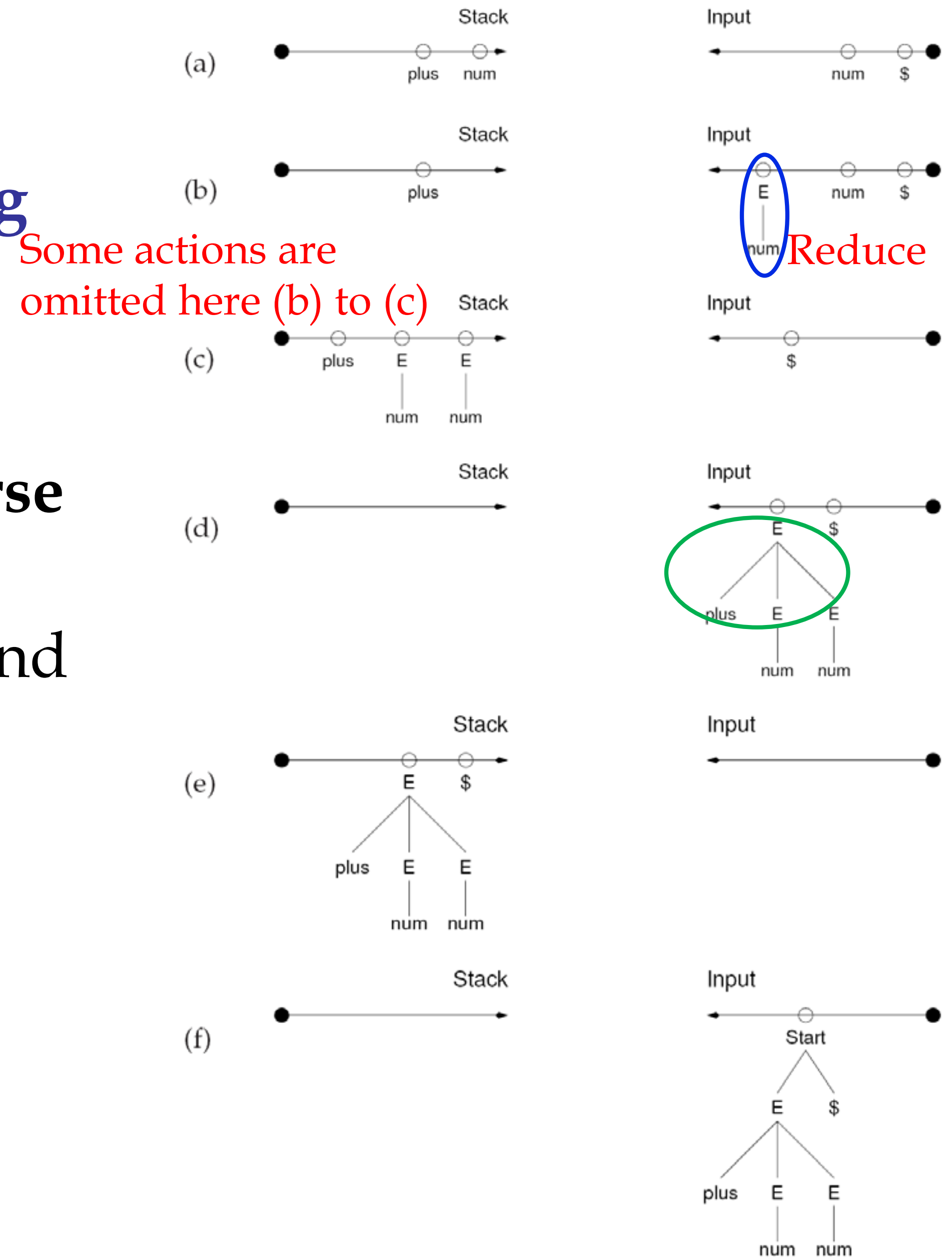


Figure 6.1: Bottom-up parsing resembles knitting.



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( )]          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
```

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()]                  (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)
```

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→γ 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 operations
  - **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

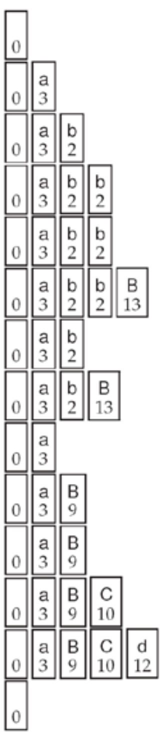
More about LR Parsing

- 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	$\boxed{3}$	$\boxed{2}$	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						

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	d c $
	Reduce λ to B	B d c $
	shift B	d c $
	Reduce b B to B	B d c $
	shift B	d c $
	Reduce b B to B	B d c $
	shift B	d c $
	Reduce λ to C	C d c $
	shift C	d c $
	shift d	c $
	Reduce a B C d to A	A c $

(continue to Figure 6.7)

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

- A **shift** to State s is denoted by $\boxed{s}$
- **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: $\boxed{\begin{matrix} a \\ n \end{matrix}}$
- The top symbol a is the **symbol** causing the cell to be pushed
- The bottom element n is the **parser state** entered when the cell is pushed
- The parsing engine in Fig. 6.3 keeps track only of the state

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

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
  - After **reduce** is applied, the resulting state is the state of TOS

Stack		Input
0	Initial Configuration	a b b d c $
0 a3	shift a	b b d c $
0 a3 b2	shift b	b d c $
0 a3 b2 b2	shift b	d c $
0 a3 b2 b2	Reduce λ to B	B d c $
0 a3 b2 b2 B13	shift B	d c $
0 a3 b2	Reduce b B to B	B d c $
0 a3 b2 B13	shift B	d c $
0 a3	Reduce b B to B	B d c $
0 a3 B9	shift B	d c $
0 a3 B9	Reduce λ to C	C d c $
0 a3 B9 C10	shift C	d c $
0 a3 B9 C10 d12	shift d	c $
0	Reduce a B C d to A	A c $

(continue to Figure 6.7)

Table [0, a] = (3)
Table [3, b] = (2)
Table [2, b] = (2)
Table [2, d] = 8
Table [2, B] = (13)
Table [13, d] = 7

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	$\boxed{3}$	$\boxed{2}$	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						

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

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 $

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

Using LR Parse Table (2/2)

- You should finish the following steps by yourself

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						

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

Stack	Action	Input
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	

Figure 6.7: Continued bottom-up parse of a b b d c $.

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

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 $.

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), and SLR(1)
  - LR(1), and LALR(1) are important, but will not cover here



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
- The following slides show
  - the construction of the ***parser states*** for the LR(0) table
  - the transitions among the table states

Preliminaries – Reduction

- Given a production rule *r*,
  - **prior to reducing the RHS(*r*) to LHS(*r*), each component of the RHS must be found**

Example

- Consider the rule **E ⇒ 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:
  - $A \Rightarrow X_1 \ldots X_i \cdot X_{i+1} \ldots X_j$
  - The **bookmark symbol `·'**, in an item may appear anywhere in the RHS of a production



Preliminaries – Items

- Four items for the rule **A ⇒ XYZ**
  1. **A ⇒ ·XYZ**
  2. **A ⇒ X · YZ**
  3. **A ⇒ XY · Z**
  4. **A ⇒ XYZ ·**
- One item for the rule **A ⇒ λ**
  1. **A ⇒ ·**
  – λ denotes that there is nothing on this rule's RHS

- Fig. 6.8 shows the progress of the bookmark symbol `•'
  – through all of the possible LR(0) items for the production **E→plus E E**

LR(0) item	Progress of rule in this state
E→ • plus E E	Beginning of rule
E→plus • E E	Processed a plus, expect an E
E→plus E • E	Expect another E
E→plus E E •	Handle on top-of-stack, ready to reduce

Figure 6.8: LR(0) items for production E→plus E E.

Preliminaries – Fresh and Reducible Items

- A **fresh** item has its bookmark at the extreme left
  - E.g., **E→ · plus E E**

- The item is **reducible** when the bookmark is at the extreme right
  - e.g., as in **E→plus E E ·**

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 following two rules:

1. Initially, add every item in I to **CLOSURE(I)**
2. If $\mathbf{A} \rightarrow \alpha \cdot \mathbf{B}\ \beta$ is in **CLOSURE(I)**, and
   $\mathbf{B} \rightarrow \gamma$ is a production, then
   add $\mathbf{B} \rightarrow \cdot\gamma$ to **CLOSURE(I)**, if it is not already there

   Apply this until no more new items can be added

*Courtesy of Compilers: Principles, Techniques, & Tools. Alfred V. Aho and et al. Pearson/Addison Wesley, 2007 (Section 4.6.2)



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

- Intuitively, $A \rightarrow \alpha \cdot B\ \beta$ in 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 CLOSURE(I)

*Courtesy of Compilers: Principles, Techniques, & Tools. Alfred V. Aho and et al. Pearson/Addison Wesley, 2007 (Section 4.6.2)

Preliminaries – Closure (State)

- The closure of state ***s*** is computed at **Marker 14** in Fig. 6.10
- More about the **Closure(*state*)**

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**

for each item associated with the nonterminal B
A → α·**B** β in ***s*** (**Marker 15**)

$\mathbf{B} \rightarrow \gamma$

for each production of **B**

add the ·RHS(**B**) into the ***ans*** set (**Marker 16**)

·γ

repeat the above until the ***ans*** set is converged

```
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
```

Figure 6.10: LR(0) closure and transitions.



LR(0) Closure Example

- Given the grammar as follows:
  - E’ ⇒ E
  - E ⇒ E + T | T
  - T ⇒ T * F | F
  - F ⇒ (E) | id
  - Where *I* is the set of one item {**E’ ⇒ ·E**}

- We compute the **CLOSURE(*I*)**



LR(0) Closure Example (Cont'd)

Given that *I* is the set of one item {**E' ⇒ ·E**}, compute **CLOSURE(*I*)**

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

E' ⇒ E
E ⇒ E + T | T
T ⇒ T * F | F
F ⇒ (E) | id

Another LR(0) Closure Example

- Given the grammar as follows:

  $S \Rightarrow E\ \$$

  $E \Rightarrow E + T \ \ | \ \ T$

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

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

$\{ S \longrightarrow \cdot E\$,$

$E \longrightarrow \cdot E+T,$

$E \longrightarrow \cdot T,$

$T \longrightarrow \cdot ID,$

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

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*
  - AdvanceDot(*s*, *X*) = *s′*
  - AdvanceDot(*s*, *X*) computes the next state *s′*
    - *s′* is **the item set** reachable from *s* via *X*

```
function AdvanceDot(state, X) returns Set
    return ({ A→αX • β | A→α • Xβ ∈ state })
end
```

Figure 6.9: LR(0) construction.

- AdvanceDot(*s*, *X*) is defined to be
  - the closure of the set of **all items** [A → α *X*·β] given that [A → α·*X* β] is in *s*
  - This functions defines the transitions in the LR(0) automaton for a grammar



AdvanceDot Example

- Given the grammar, G
  - E’ ⇒ E
  - E ⇒ E + T | T
  - T ⇒ T * F | F
  - F ⇒ (E) | id
- and the state *s* (item set)
  - *s* refers to {E ⇒ E · + 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*] = `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                                                        (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
```

Figure 6.10: LR(0) closure and transitions.

```
function AddState(States, items) returns State
    if items ∉ States                                             (9)
    then        ← Add a new state
        s ← newState(items)                                       (10)
        States ← States ∪ { s }
        WorkList ← WorkList ∪ { s }                               (11)
        Table[s][★] ← error                                       (12)
    else  s ← FindState(items)    ← Return the state of the items
    return (s)
end
function AdvanceDot(state, X) returns Set
    return ({ A→αX • β | A→α • Xβ ∈ state })                      (13)
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 ***StartItems***
(***Start*** is added into the ***WorkList*** in **AddState**)

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*)
    $States \leftarrow \emptyset$
    $StartItems \leftarrow \{\, \text{Start} \rightarrow \bullet\ \text{RHS}(p) \mid p \in \text{ProductionsFor}(\text{Start}) \,\}$ (7)
    $StartState \leftarrow$ AddState(*States*, *StartItems*)
    while $(s \leftarrow WorkList.\text{ExtractElement}(\,)) \neq \perp$ **do** (8)
        call ComputeGoto(*States*, *s*)
    return ((*States*, *StartState*))
end
function AddState(*States*, *items*) **returns** *State*
    if $items \notin States$ (9)
    then
        $s \leftarrow newState(items)$ (10)
        $States \leftarrow States \cup \{\, s \,\}$
        $WorkList \leftarrow WorkList \cup \{\, s \,\}$ ← Add the new state to the list (11)
        $Table[s][\star] \leftarrow$ error (12)
    else $s \leftarrow FindState(items)$
    return (*s*)
end
function AdvanceDot(*state*, $\mathcal{X}$) **returns** *Set*
    return $(\{\, \text{A} \rightarrow \alpha\mathcal{X} \bullet \beta \mid \text{A} \rightarrow \alpha \bullet \mathcal{X}\beta \in state \,\})$ (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 → · 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 · 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	→	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 $.

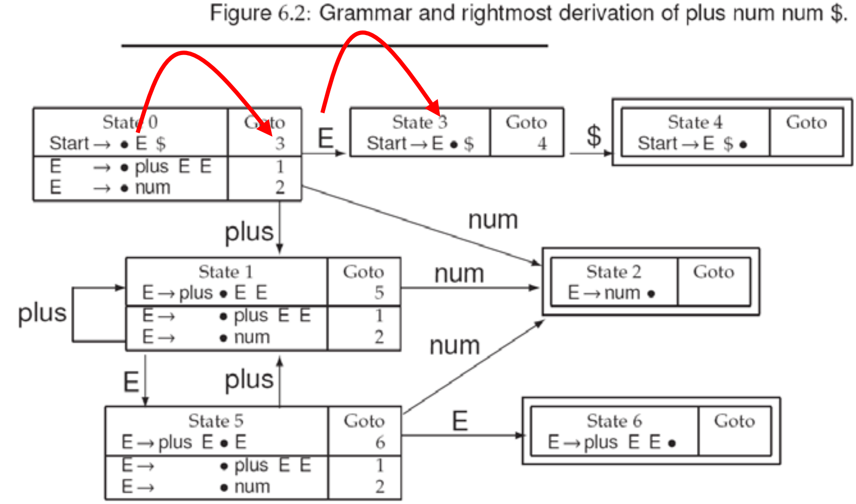


Figure 6.11: LR(0) computation for Figure 6.2, shown as a characteristic finite-state machine. State 0 is the initial state, and the double-boxed states are accept states.

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)

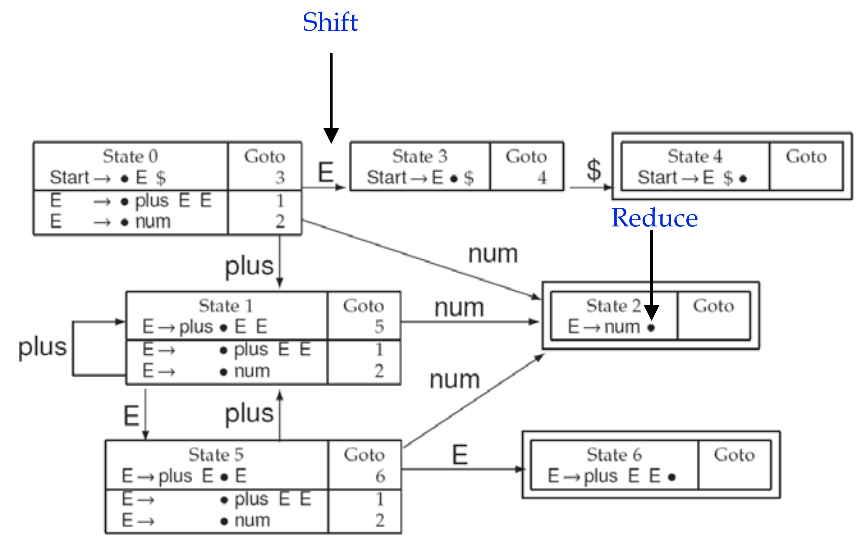


Figure 6.11: LR(0) computation for Figure 6.2, shown as a characteristic finite-state machine. State 0 is the initial state, and the double-boxed states are accept states.

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 **ComputeLR0(*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*) → RHS(*r*)** ·) then

call **AssertEntry** (***s***, ***X***, reduce ***r***) to add the entry in the ***Table*** (Marker 8)

```
   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)          (21)
   end
   procedure AssertEntry(state, symbol, action)
(5)    if Table[state][symbol] = error                           (22)
(6)    then  Table[state][symbol] ← action
       else
(7)        call ReportConflict(Table[state][symbol], action)     (23)
   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   ← The items in double-checked boxes
       then
(8)        foreach X ∈ (Σ ∪ N) do  call AssertEntry(s, X, reduce r)
   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 Fig. 6.2, LR(0) analysis yields the parse table as shown in Fig. 6.15

```
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)        (21)
end
procedure AssertEntry(state, symbol, action)
⑤ if Table[state][symbol] = error                          (22)
⑥ then  Table[state][symbol] ← action
    else
⑦     call ReportConflict(Table[state][symbol], action)    (23)
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.

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				

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 $

State 0

State 1 **Shift**

State 1 **Shift**

State 2 **Shift** **Reduce**

State 2 **Shift** **Reduce**

Back to State 1 **E**

State 5 **Shift**

1 Start → E $
2 E → plus E E
3 | num

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

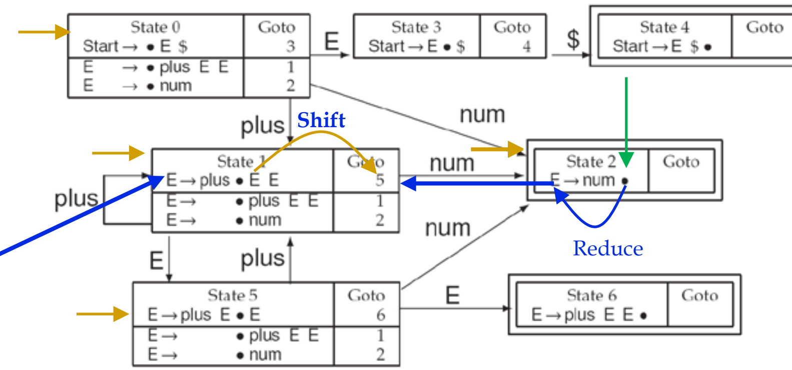


Figure 6.11: LR(0) computation for Figure 6.2, shown as a characteristic finite-state machine. State 0 is the initial state, and the double-boxed states are accept states.

Reduce

Shift

Shift E and 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

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 $

State 0 | State 1 **Shift** | State 1 **Shift** | State 2 **Shift** **Reduce** | State 2 **Shift** **Reduce**

Back to State 1 **E** | Back to State 5 **E**

State 5 **Shift** | State 6 **Shift**

1 Start → E $
2 E → plus E E
3 | num

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

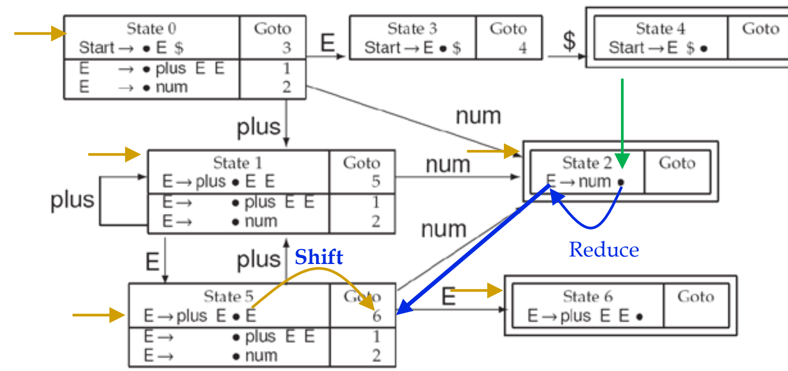


Figure 6.11: LR(0) computation for Figure 6.2, shown as a characteristic finite-state machine. State 0 is the initial state, and the double-boxed states are accept states.

Sentential Prefix	Transitions	Resulting Sentential Form
		plus plus num 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

Reduce

Shift

Shift E and num

Shift E

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 $

State 0	State 1	State 1	State 2	State 2
	Shift	**Shift**	**Shift**	**Shift**
			Reduce	**Reduce**
			Back to State 1 **E**	Back to State 5 **E**
			State 5	State 6
			Shift	**Shift**
				Reduce
				Back to State 5
				State 1
				Reduce

State 5

E → plus E E·

1 Start → E $
2 E → plus E E
3 | num

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

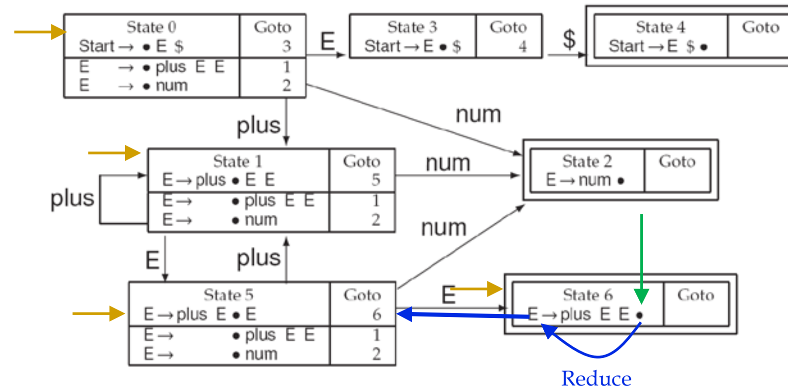


Figure 6.11: LR(0) computation for Figure 6.2, shown as a characteristic finite-state machine. State 0 is the initial state, and the double-boxed states are accept states.

Sentential Prefix	Transitions	Resulting Sentential Form
		plus plus num 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

Reduce Shift Shift E and num Shift E Reduce

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 → E $
2 E → plus E E
3 | num

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

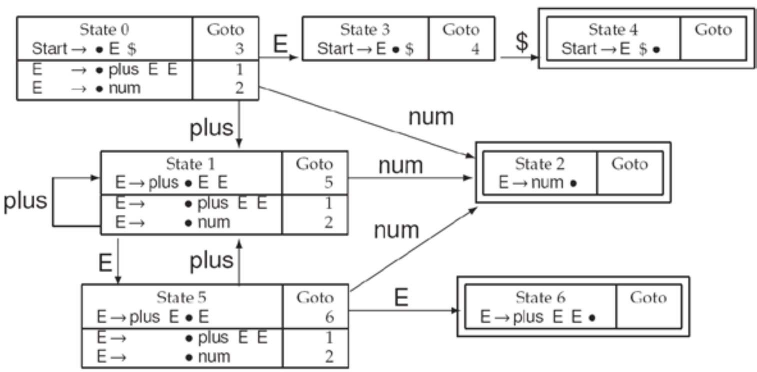


Figure 6.11: LR(0) computation for Figure 6.2, shown as a characteristic finite-state machine. State 0 is the initial state, and the double-boxed states are accept states.

Sentential Prefix	Transitions	Resulting Sentential Form
		plus plus num 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

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 develop 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	$	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						

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

Simple reduce op

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				

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**
  1. A **plus** can be **shifted** to arrive in State 3
  2. A **reduction** can also be applied, **E→E plus E**
- This inadequate state exhibits a **shift/reduce conflict** for LR(0)

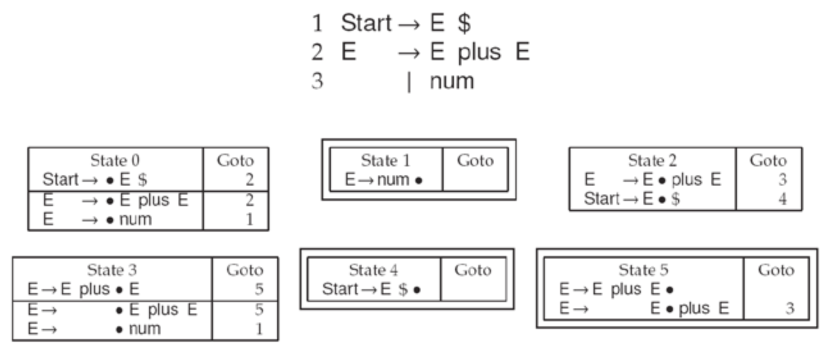


Figure 6.16: An ambiguous expression grammar.



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:

1. 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
  i. One option is a reduction by E→E plus E •
  ii. However, with the item E→E • plus E, it is also possible to shift a plus and then an E

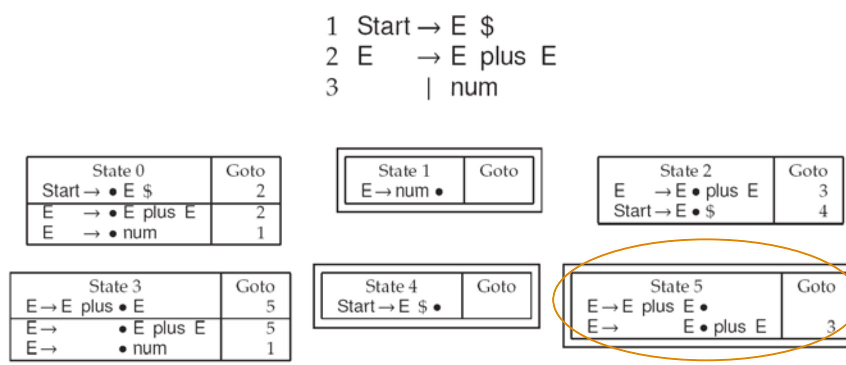


Figure 6.16: An ambiguous expression grammar.

Is the Grammar Ambiguous? (Cont'd)

1 Start → E $
2 E → E plus E
3 | num

- 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

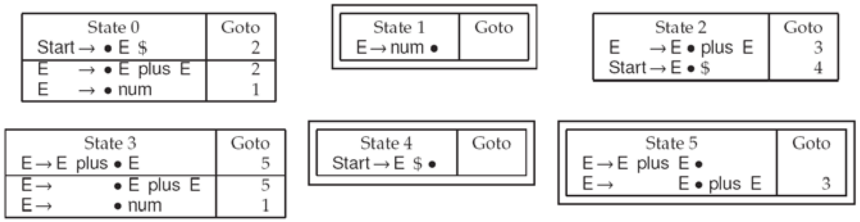


Figure 6.16: An ambiguous expression grammar.

- 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

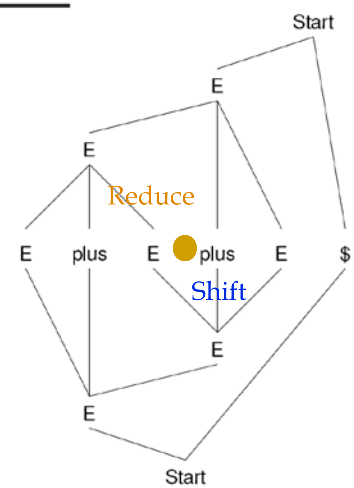


Figure 6.17: Two derivations for E plus E plus E $. The parse tree on top favors reduction in State 5; the parse tree on bottom favors a shift.



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

Start $\Rightarrow_{rm}$ **Exprs \$**
$\Rightarrow_{rm}$ **E a \$**
$\Rightarrow_{rm}$ **E plus num a \$**
$\Rightarrow_{rm}$ **E plus … plus num a \$**
$\Rightarrow_{rm}$ **num plus … plus num a \$**

Stack	Action	Input
0	Initial Configuration	num plus . . . plus num a \$
0 | num 2	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

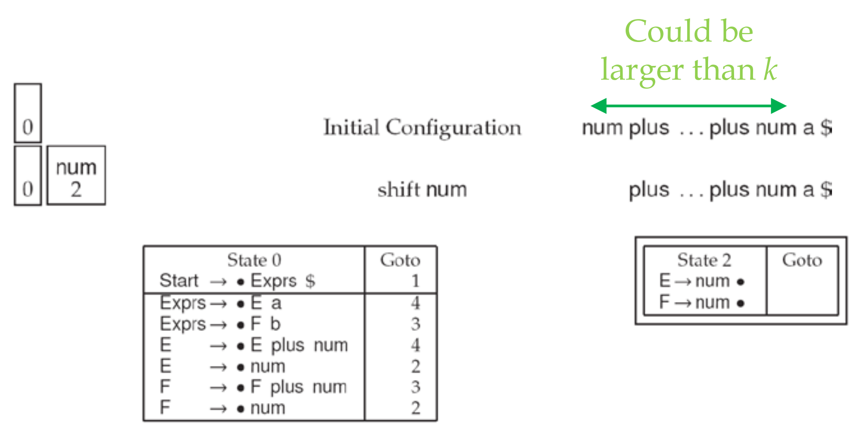


Figure 6.19: A grammar that is not LR(*k*).

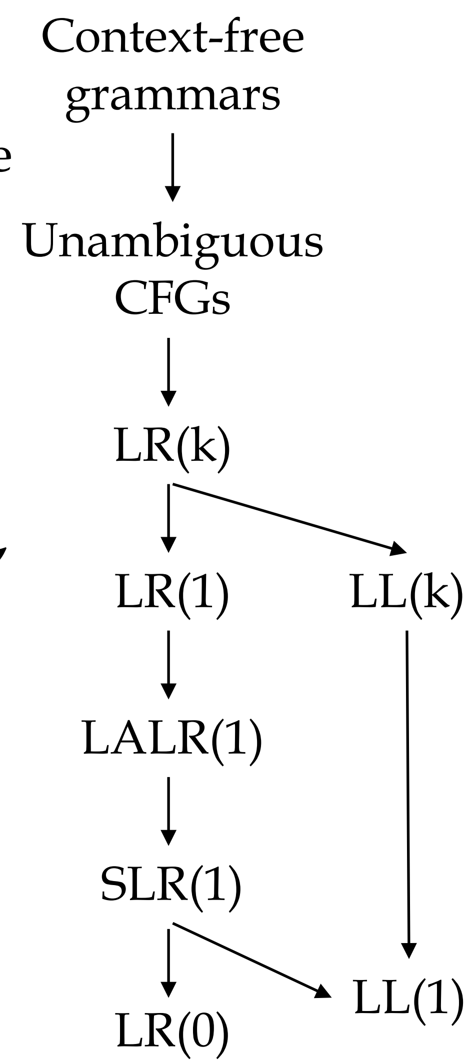


Summary of the Conflicts

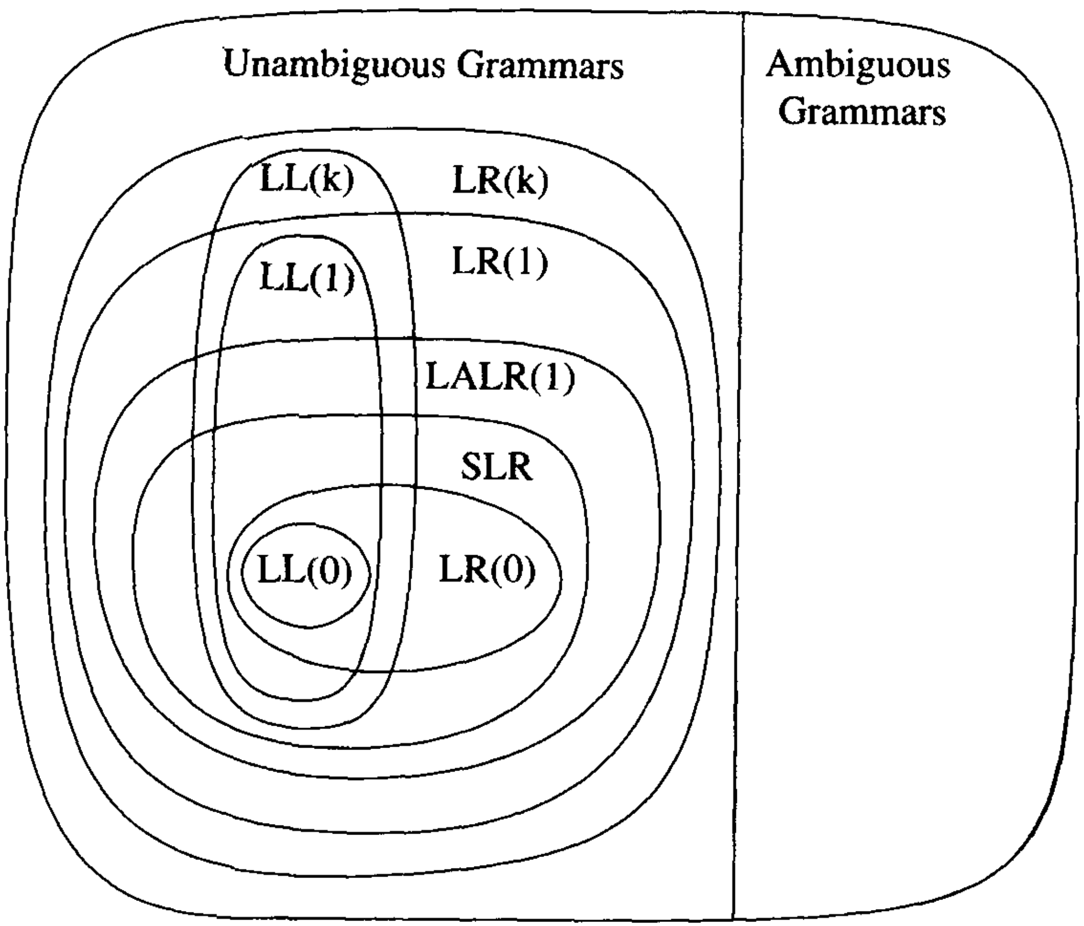
- Simple grammars and languages can have subtle problems
  - that prohibit generation of a bottom-up parser
  - It follows that a top-down parser would also fail on such grammars
- The LR(0) construction can provide **important clues** for diagnosing a grammar’s inadequacies
  - understanding and resolving 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



Grammar Class Hierarchy



Courtesy of: Appel, A. W. (1998c). *Modern Compiler Implementation in ML/Java/C.* Cambridge University Press.



SLR(*k*) Table Construction

- SLR(*k*) method
  - **S**imple **LR** with *k* tokens of lookahead
  - attempts to resolve inadequate states

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

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

- 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 | E times num
4 | num

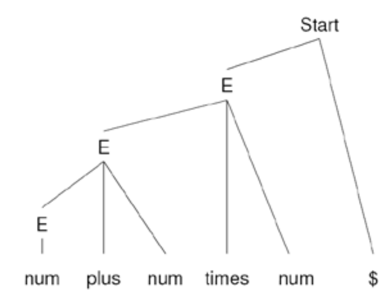


Figure 6.20: Expressions with sums and products.

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 → E $
2 E → E plus num
3 | E times num
4 | num

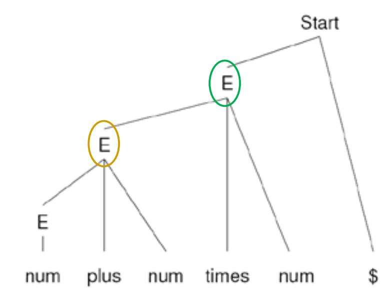


Figure 6.20: Expressions with sums and products.



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

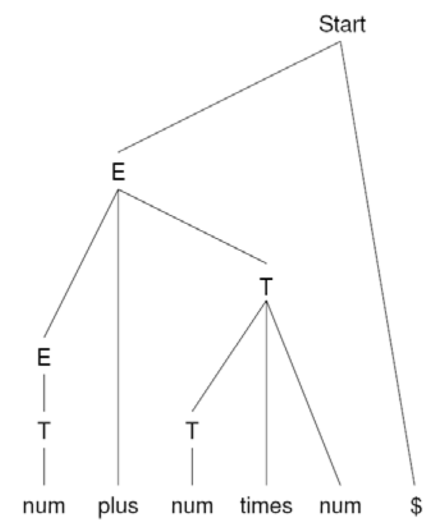


Figure 6.21: Grammar for sums of products.

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

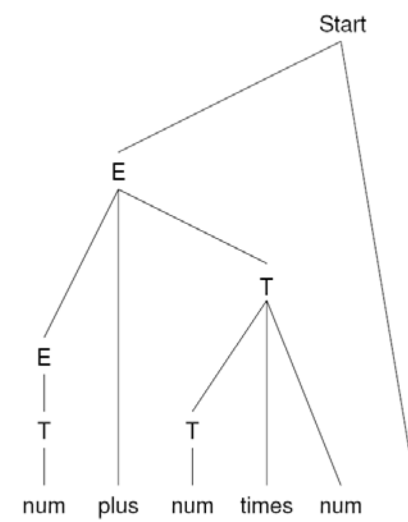


Figure 6.21: Grammar for sums of products.

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

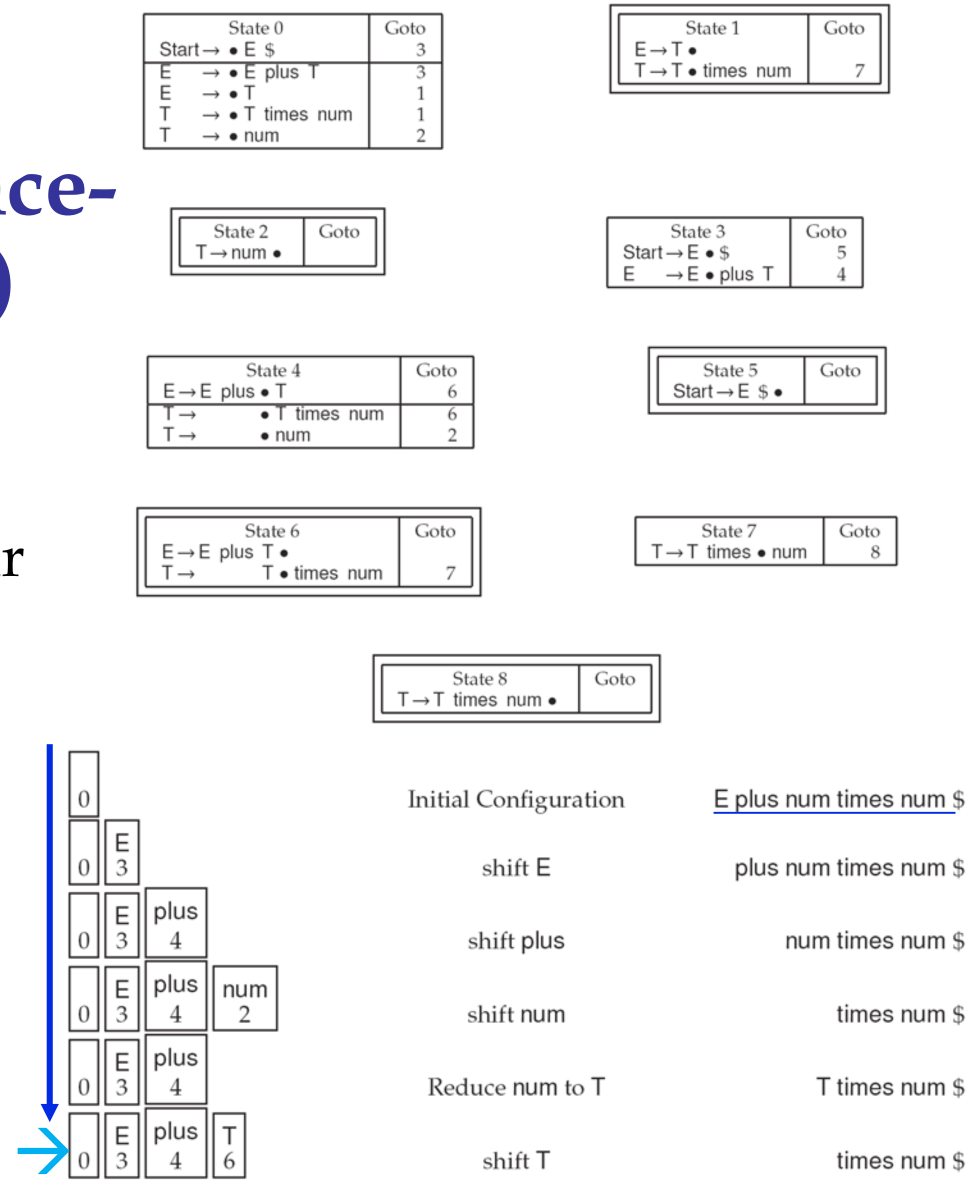


Figure 6.22: LR(0) construction and parse leading to inadequate State 6.

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
  1. What to **shift**?
     → Shift times into stack
     (**E plus T** · **times num $**)
  2. What to **reduce**?
     → Reduce by Rule 2 **E→ E plus T**
     (**E plus T** · **times 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 •	
T → T • times num	7

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 $

→ ?

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

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

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

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

1. 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
2. If the **reduction** is taken
   - by Rule 2 **E→E plus T** for the sentential form: **E plus T · times num $**,
   - which yields $\boxed{\textbf{E · times num \$}}$ 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

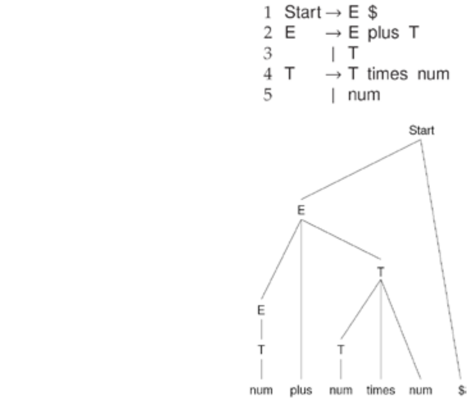


Figure 6.21: Grammar for sums of products.

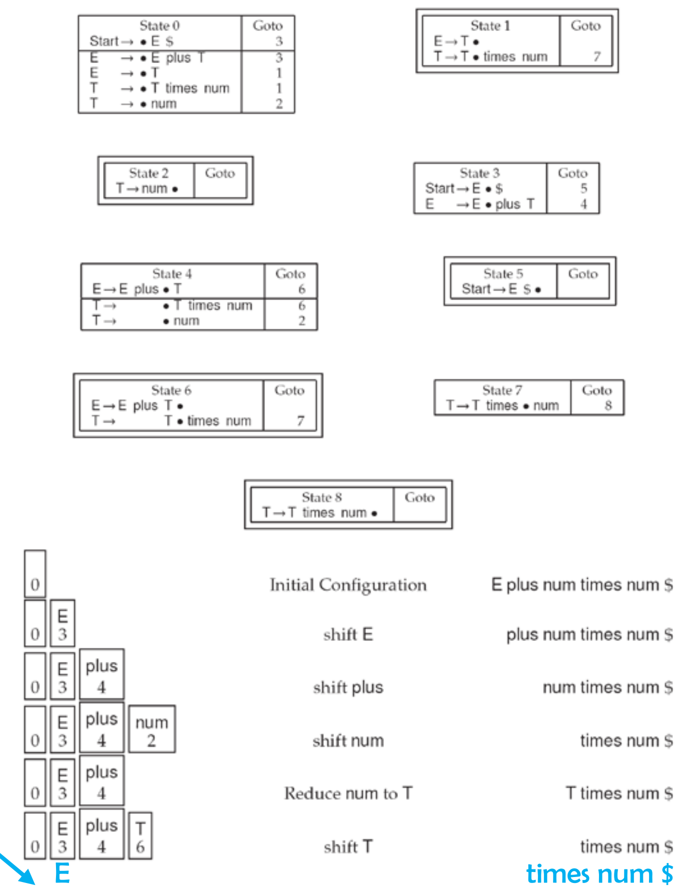


Figure 6.22: LR(0) construction and parse leading to inadequate State 6.

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

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

1. 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
2. If the **reduction** is taken
   - by Rule 2 **E→<u>E plus T</u>** (**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→ <u>num</u>**
   - we further obtain **E times E · $** after applying Rule 3 **E→ <u>T</u>**
   - 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
```

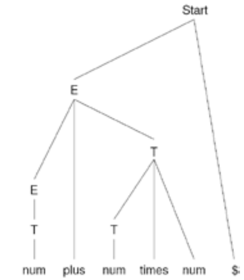


Figure 6.21: Grammar for sums of products.

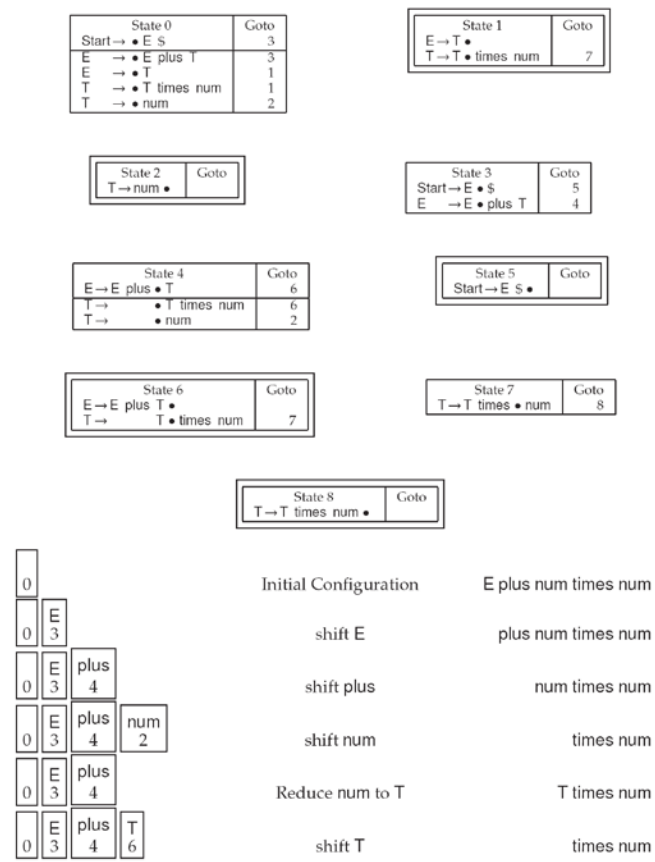


Figure 6.22: LR(0) construction and parse leading to inadequate State 6.



Further Discussion of the Grammar

- With the item **E→E plus T•** in **State 6**,
  - reduction by **E→E plus T** must be appropriate **under some conditions**
  - In previous slide, we know that **E→E plus T times** is not working
  - What if we have other terminals after **E→E plus T ?**
- For example, if we examine the sentential forms:
  - **E plus T $** and **E plus T plus num $**
  - We found that the **E→E plus T •** can be reduced in **State 6** and
  - the parsing can go on if the next input symbol is **plus** or **$**, but not **times**
- Specifically, we consider the sequence of parser actions
  - that could be applied between a reduction by **E→E 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	$\boxed{2}$	$\boxed{1}$		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				

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 ∈ 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

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

Figure 6.14: LR(0) version of TryRuleInState.

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			

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

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

For items in double-checked boxes

Figure 6.23: SLR(1) version of TryRuleInState.

SLR(1) Table Construction (Cont'd)

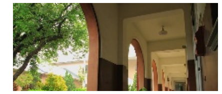
- In the previous example,
  - **States 1** and **6** are resolved by computing **Follow(*E*) = { plus, $ }**

```
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.

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			

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



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

Rule 3

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 $ •	

Rule 2

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 •	

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 $

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

Spurious Conflicts in SLR
(This example shows the previous trick does not work)

Grammar G

0. $S' \rightarrow S$
1. $S \rightarrow L = R$
2. $S \rightarrow R$
3. $L \rightarrow * R$
4. $L \rightarrow id$
5. $R \rightarrow L$

FIRST(S) = {*, id}
FIRST(L) = {*, id}
FIRST(R) = {*, id}
FOLLOW(S) = {$}
FOLLOW(L) = {$, =}
FOLLOW(R) = {$, =}

- Some grammars may cause conflicts when constructing SLR tables
- Given the grammar *G*, the FIRST and FOLLOW sets, and the constructed CFSM states

State 0:
$S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot id$
$R \rightarrow \cdot L$

State 1:
$S' \rightarrow S \cdot$

State 2:
$S \rightarrow L \cdot = R$
$R \rightarrow L \cdot$

State 3:
$S \rightarrow R \cdot$

State 4:
$L \rightarrow * \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot id$

State 5:
$L \rightarrow id \cdot$

State 6:
$S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot id$

State 7:
$L \rightarrow * R \cdot$

State 8:
$R \rightarrow L \cdot$

State 9:
$S \rightarrow L = R \cdot$

*Courtesy of Compilers: Principles, Techniques, & Tools. Alfred V. Aho and et al. Pearson/Addison Wesley, 2007 (Fig. 4.44 in Example 4.6.1)

Spurious Conflicts in SLR (Cont'd)

- We use the State 2 as the example to show the parsing conflict
  - LR(0) item **S → L · = R** calls for a **shift** on lookahead '='
  - But, LR(0) item **R → L ·** calls for a **reduce** on lookahead '=', since **FOLLOW(R)** contains '='
  - When lookahead is '=', there is a shift-reduce conflict
- However, in fact, '=' can only follow **L** in the context of the production **S → L = R**
  - This gives a hint for resolving the conflict by choosing the **shift** action in State 2
- The cause of the above problem is that **FOLLOW** sets are **global**, taking information from the entire grammar

State 2:
S → L · = R
R → L ·

FOLLOW(S) = {$}
FOLLOW(L) = {$, =}
FOLLOW(R) = {$, =}



Key Concept in LALR Parsers

- LR(k) and LALR(k) parsers further handle the situations by adding extra lookahead information in the *items*
  - E.g., the LR parsers adds lookahead information for each item
- LALR is a more efficient algorithm by adding lookahead information for *each reducible item*
  - Fig. 6.27 shows LALR uses the item-wise FOLLOW set when adding the **reduce** entries
  - The grammar *G* in the previous page is not in SLR(1), but in LALR(1)
  - If you are interested LALR(k) table construction, you can refer to Section 6.5.2 and 6.5.3 by yourself
- LALR(1) is the basis for most bottom-up parser generators
  - To achieve greater power, more lookahead can be applied, but this is rarely necessary

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

Figure 6.23: SLR(1) version of TryRuleInState.

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

Figure 6.27: LALR(1) version of TryRuleInState.

QUESTIONS?