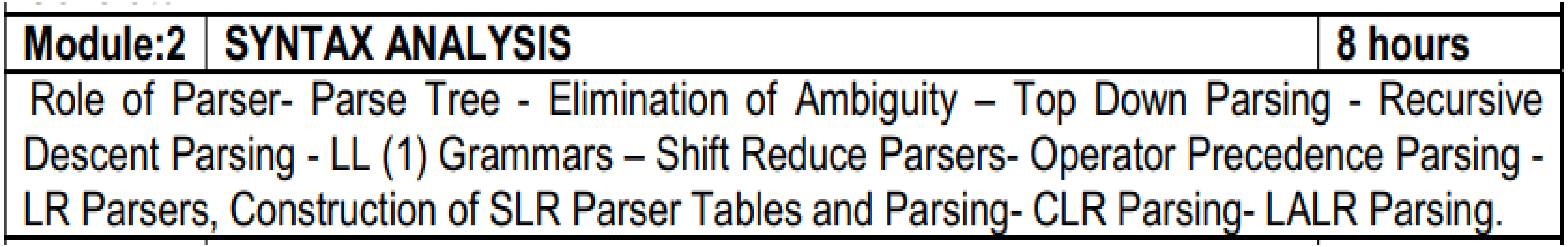
BCSE307L – COMPILER DESIGN

TEXT BOOK:

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques and Tools", Second Edition, Pearson Education Limited, 2014.



Module:2	**SYNTAX ANALYSIS**	**8 hours**
Role of Parser- Parse Tree - Elimination of Ambiguity – Top Down Parsing - Recursive Descent Parsing - LL (1) Grammars – Shift Reduce Parsers- Operator Precedence Parsing - LR Parsers, Construction of SLR Parser Tables and Parsing- CLR Parsing- LALR Parsing.		

Parsing Techniques

Bottom-Up Parsing:

- Construction of the parse tree starts at the leaves, and proceeds towards the root.
- Normally efficient bottom-up parsers are created with the help of some software tools.
- Bottom-up parsing is also known as shift-reduce parsing.
- Operator-Precedence Parsing – simple, restrictive, easy to implement
- LR Parsing – much general form of shift-reduce parsing, CLR, SLR, LALR

Bottom Down Parsing

- Reduction
- Handle Pruning
- Shift-Reduce Parsing
- Operator Precedence
- LR Parser
  - SLR (Simple LR)
  - CLR (Canonical LR)
  - LALR (Lookahead LR)

Bottom up Parsing

Unambiguous Expression Grammar :

E → E+T | T

T → T*F | F

F → (E)

F → id

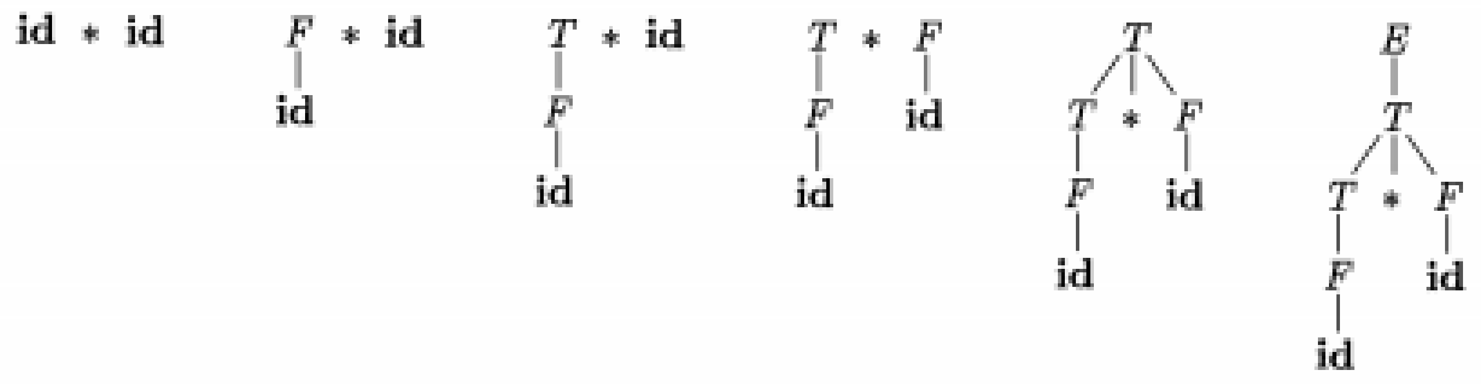
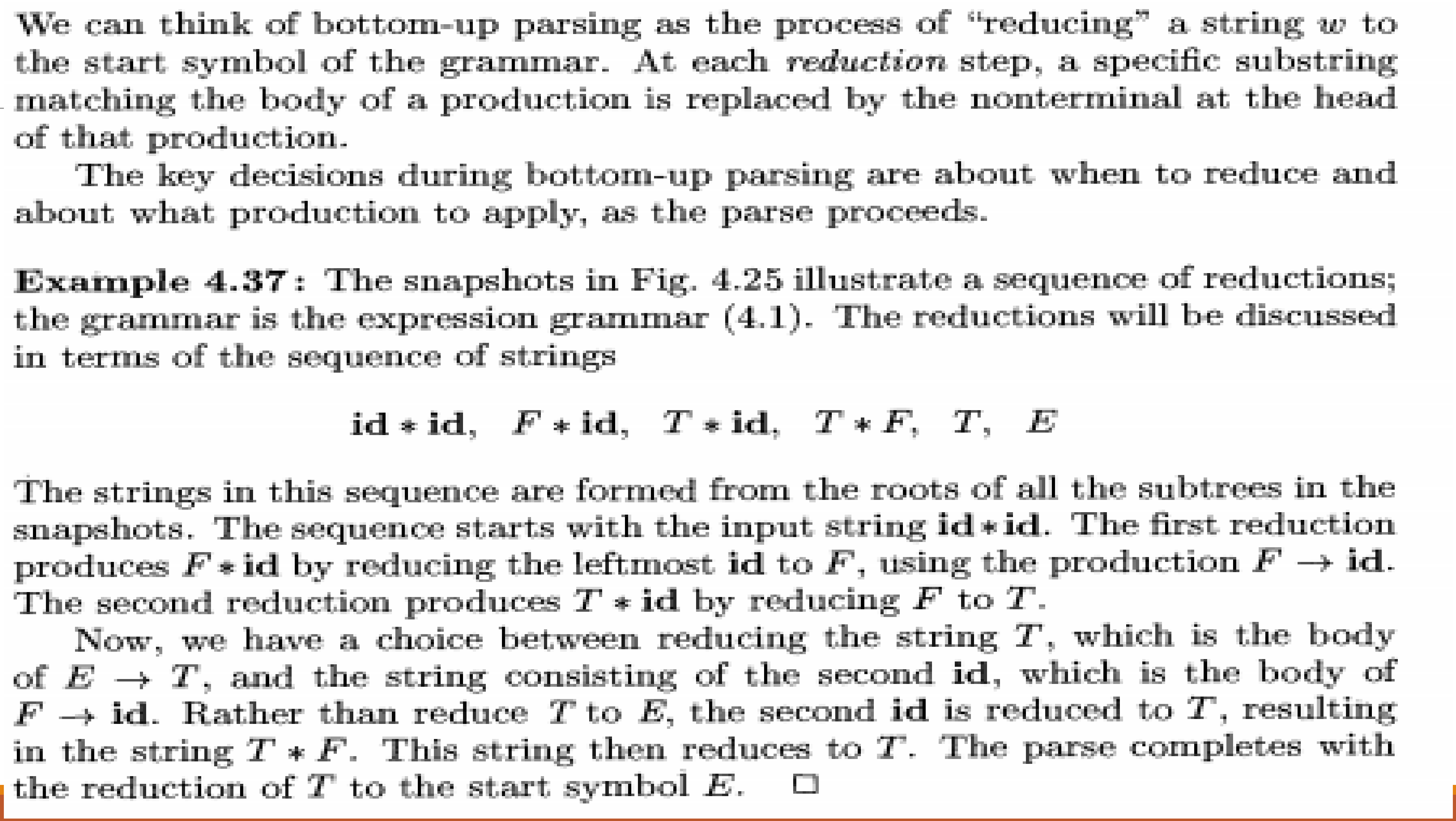


Figure 4.25: A bottom-up parse for **id** * **id**

Reduction



We can think of bottom-up parsing as the process of "reducing" a string w to the start symbol of the grammar. At each *reduction* step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

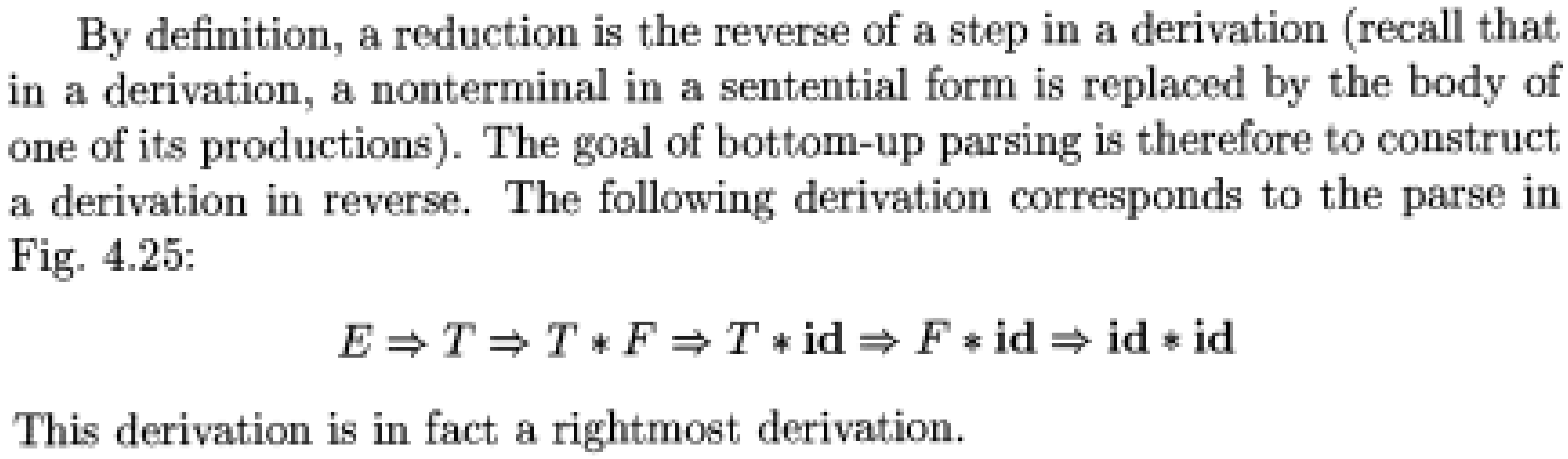
Example 4.37: The snapshots in Fig. 4.25 illustrate a sequence of reductions; the grammar is the expression grammar (4.1). The reductions will be discussed in terms of the sequence of strings

$$\mathbf{id} * \mathbf{id},\quad F * \mathbf{id},\quad T * \mathbf{id},\quad T * F,\quad T,\quad E$$

The strings in this sequence are formed from the roots of all the subtrees in the snapshots. The sequence starts with the input string $\mathbf{id} * \mathbf{id}$. The first reduction produces $F * \mathbf{id}$ by reducing the leftmost $\mathbf{id}$ to F, using the production $F \rightarrow \mathbf{id}$. The second reduction produces $T * \mathbf{id}$ by reducing F to T.

Now, we have a choice between reducing the string T, which is the body of $E \rightarrow T$, and the string consisting of the second $\mathbf{id}$, which is the body of $F \rightarrow \mathbf{id}$. Rather than reduce T to E, the second $\mathbf{id}$ is reduced to T, resulting in the string $T * F$. This string then reduces to T. The parse completes with the reduction of T to the start symbol E. □

Reduction

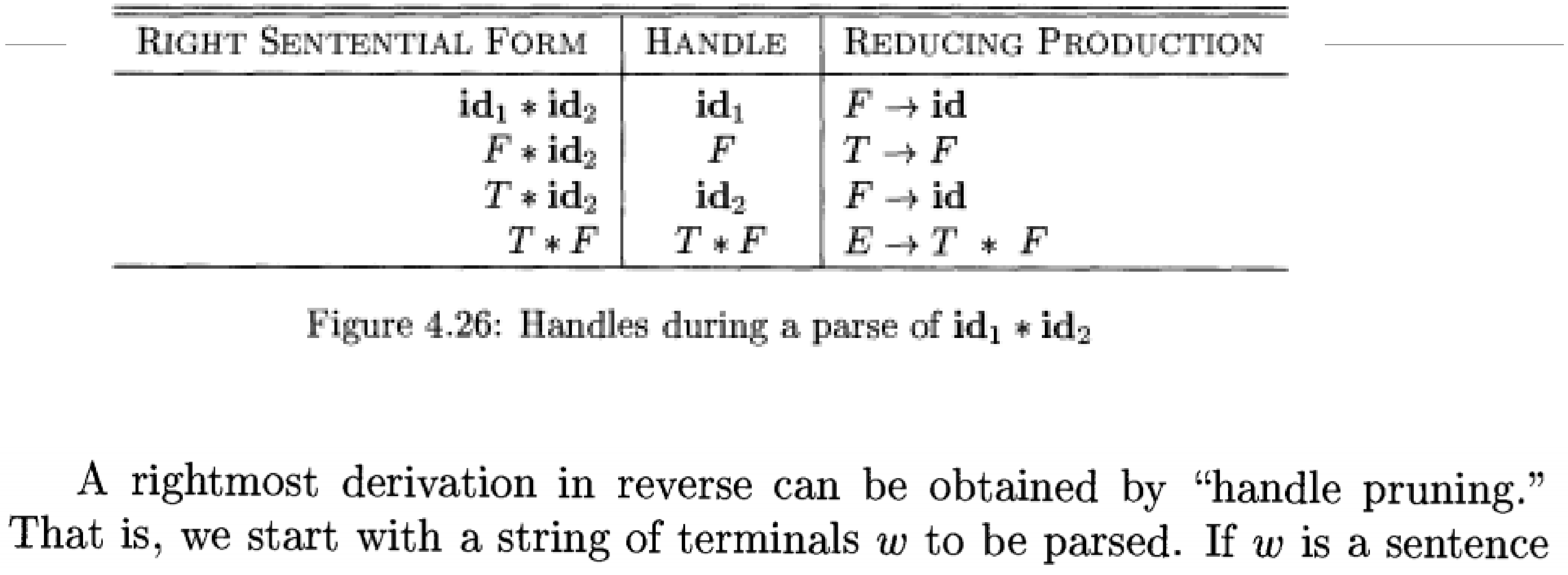


By definition, a reduction is the reverse of a step in a derivation (recall that in a derivation, a nonterminal in a sentential form is replaced by the body of one of its productions). The goal of bottom-up parsing is therefore to construct a derivation in reverse. The following derivation corresponds to the parse in Fig. 4.25:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

This derivation is in fact a rightmost derivation.

Handle Pruning



Right Sentential Form	Handle	Reducing Production
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

Figure 4.26: Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

A rightmost derivation in reverse can be obtained by "handle pruning." That is, we start with a string of terminals w to be parsed. If w is a sentence

Handle Pruning

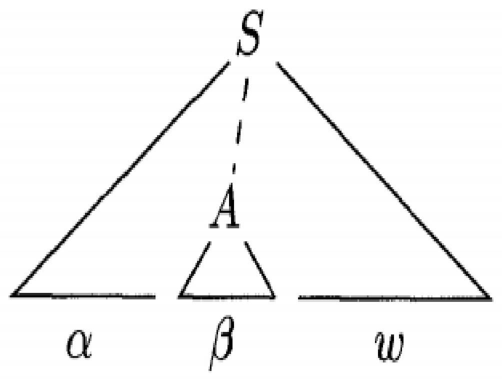
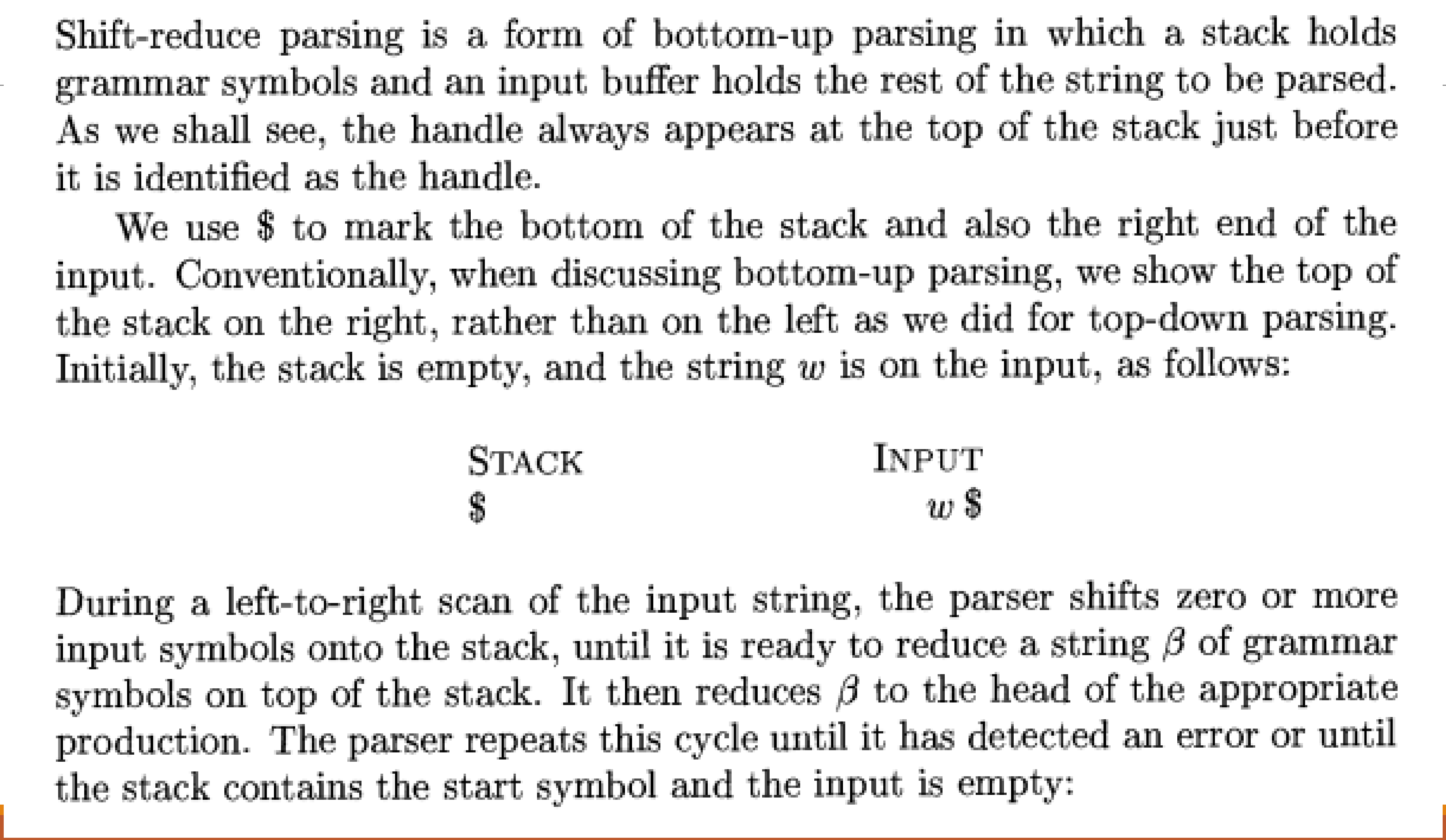


Figure 4.27: A handle $A \to \beta$ in the parse tree for $\alpha\beta w$

of the grammar at hand, then let $w = \gamma_n$, where γ_n is the nth right-sentential form of some as yet unknown rightmost derivation

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

Shift-Reduce Parsing



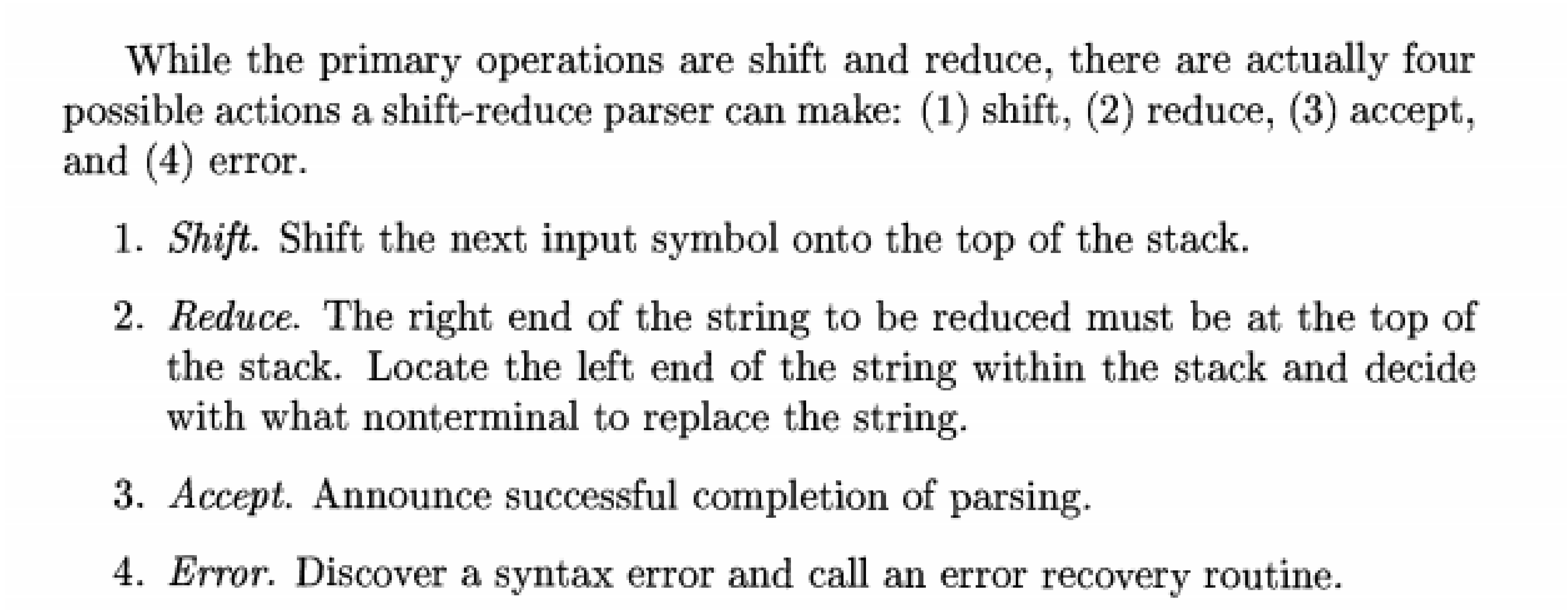
Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. As we shall see, the handle always appears at the top of the stack just before it is identified as the handle.

We use \$ to mark the bottom of the stack and also the right end of the input. Conventionally, when discussing bottom-up parsing, we show the top of the stack on the right, rather than on the left as we did for top-down parsing. Initially, the stack is empty, and the string w is on the input, as follows:

STACK	INPUT
\$	w \$

During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto 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 head 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:

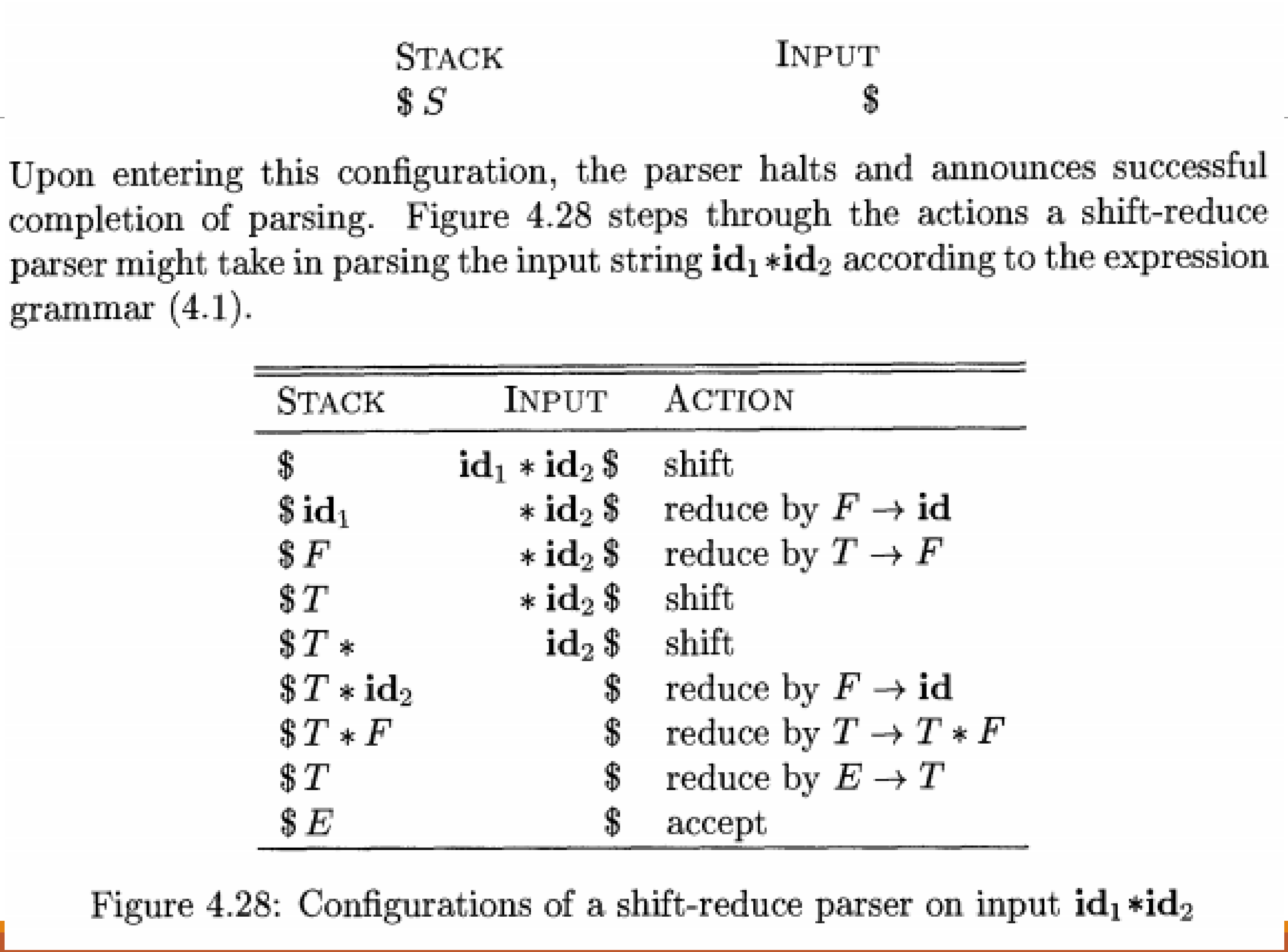
Shift-Reduce Parsing



While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make: (1) shift, (2) reduce, (3) accept, and (4) error.

1. *Shift*. Shift the next input symbol onto the top of the stack.
2. *Reduce*. The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. *Accept*. Announce successful completion of parsing.
4. *Error*. Discover a syntax error and call an error recovery routine.

Shift-Reduce Parsing



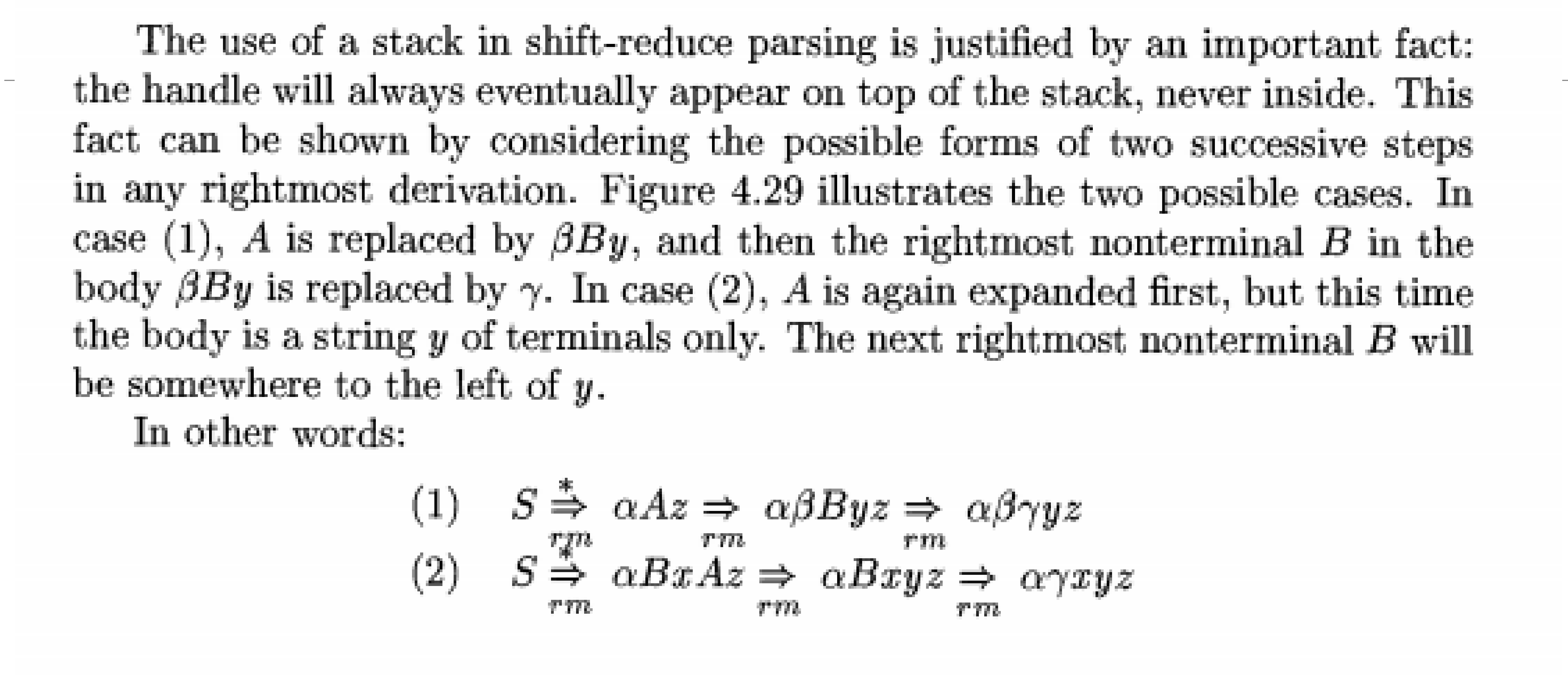
STACK	INPUT
$\$\,S$	$\$$

Upon entering this configuration, the parser halts and announces successful completion of parsing. Figure 4.28 steps through the actions a shift-reduce parser might take in parsing the input string $\mathbf{id}_1 * \mathbf{id}_2$ according to the expression grammar (4.1).

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

Figure 4.28: Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

Shift-Reduce Parsing



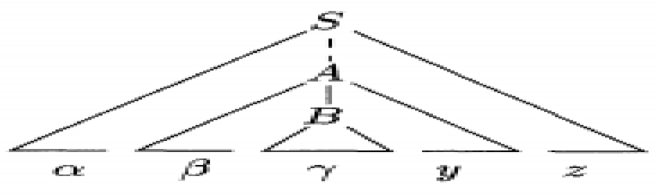
The use of a stack in shift-reduce parsing is justified by an important fact: the handle will always eventually appear on top of the stack, never inside. This fact can be shown by considering the possible forms of two successive steps in any rightmost derivation. Figure 4.29 illustrates the two possible cases. In case (1), A is replaced by $\beta B y$, and then the rightmost nonterminal B in the body $\beta B y$ is replaced by γ. In case (2), A is again expanded first, but this time the body is a string y of terminals only. The next rightmost nonterminal B will be somewhere to the left of y.

In other words:

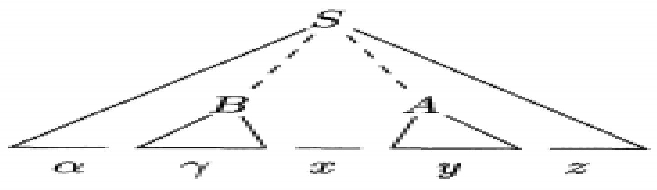
$$(1) \quad S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A z \underset{rm}{\Rightarrow} \alpha\beta B y z \underset{rm}{\Rightarrow} \alpha\beta\gamma y z$$

$$(2) \quad S \overset{*}{\underset{rm}{\Rightarrow}} \alpha B x A z \underset{rm}{\Rightarrow} \alpha B x y z \underset{rm}{\Rightarrow} \alpha\gamma x y z$$

Shift-Reduce Parsing



Case (1)



Case (2)

Figure 4.29: Cases for two successive steps of a rightmost derivation

Consider case (1) in reverse, where a shift-reduce parser has just reached the configuration

STACK	INPUT
$\$\alpha\beta\gamma$	$yz\$$

The parser reduces the handle γ to B to reach the configuration

$\$\alpha\beta B$	$yz\$$

The parser can now shift the string y onto the stack by a sequence of zero or more shift moves to reach the configuration

$\$\alpha\beta By$	$z\$$

with the handle βBy on top of the stack, and it gets reduced to A.

Now consider case (2). In configuration

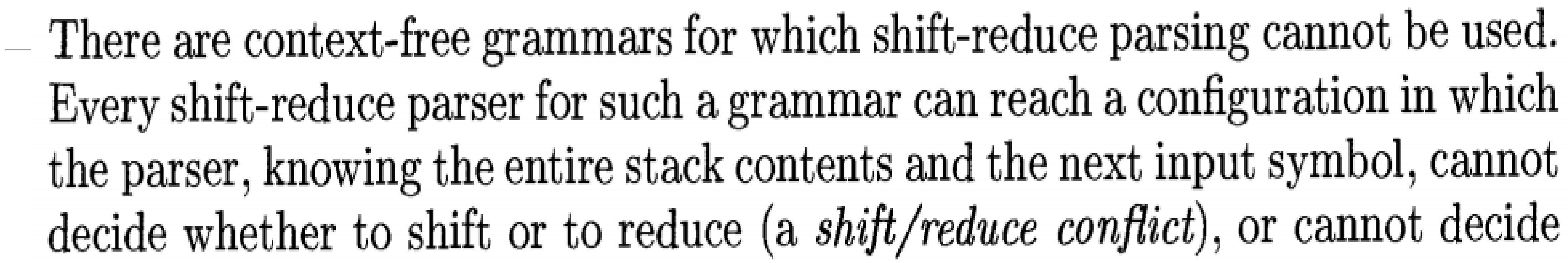
$\$\alpha\gamma$	$xyz\$$

the handle γ is on top of the stack. After reducing the handle γ to B, the parser can shift the string xy to get the next handle y on top of the stack, ready to be reduced to A:

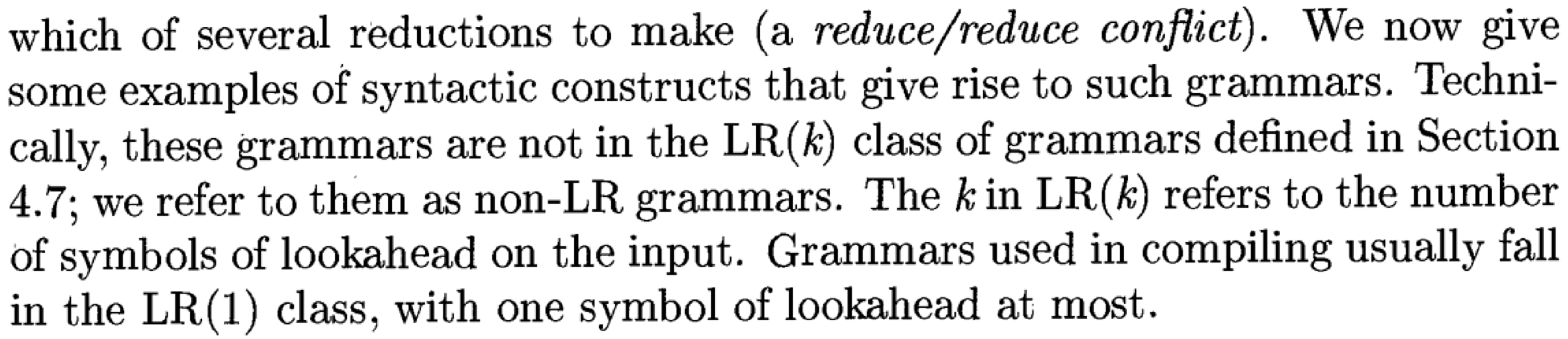
$\$\alpha Bxy$	$z\$$

In both cases, after making a reduction the parser had to shift zero or more symbols to get the next handle onto the stack. It never had to go into the stack to find the handle.

Conflicts During Shift-Reduce Parsing

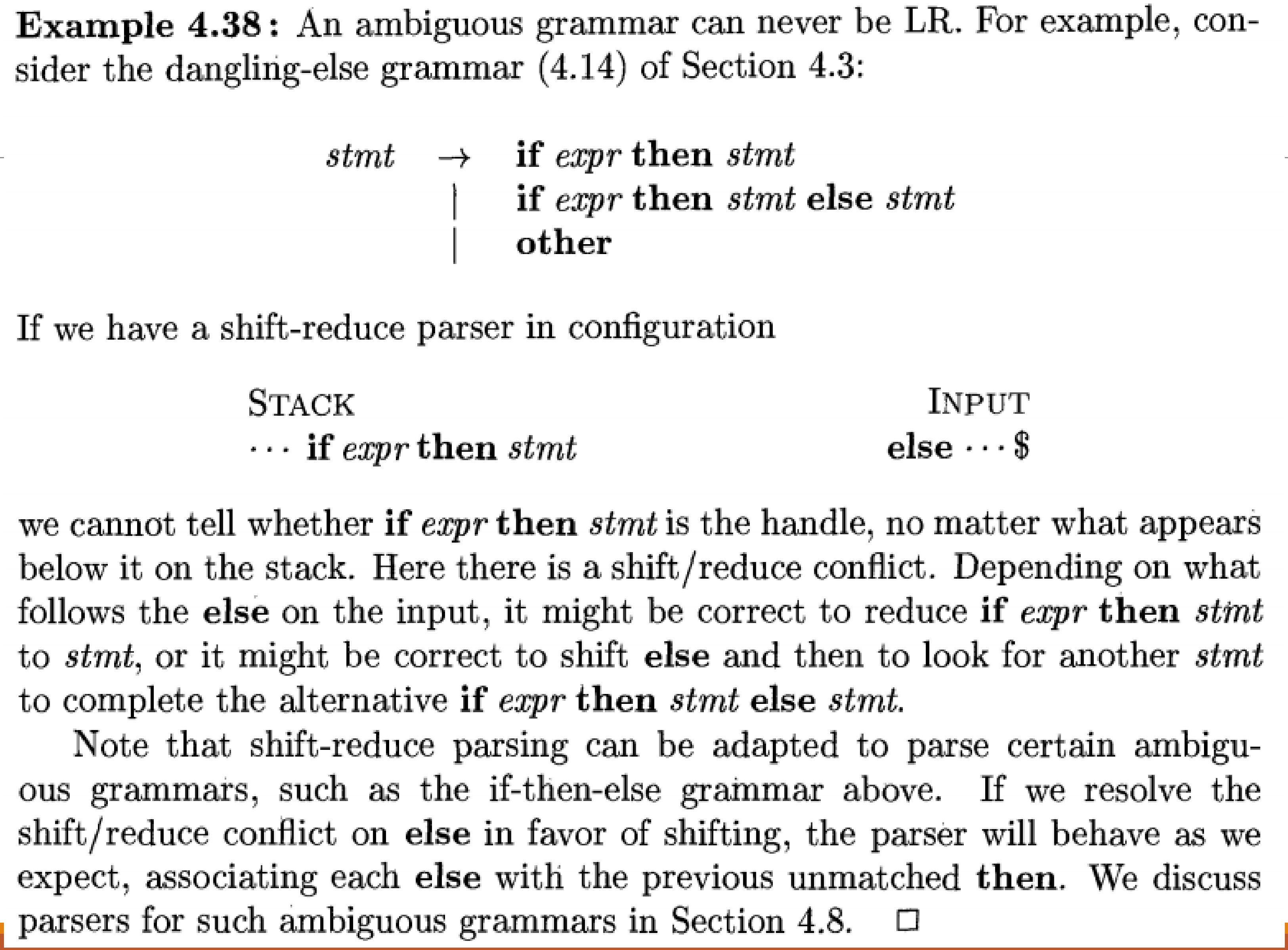


There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input symbol, cannot decide whether to shift or to reduce (a *shift/reduce conflict*), or cannot decide



which of several reductions to make (a *reduce/reduce conflict*). We now give some examples of syntactic constructs that give rise to such grammars. Technically, these grammars are not in the LR(k) class of grammars defined in Section 4.7; we refer to them as non-LR grammars. The k in LR(k) refers to the number of symbols of lookahead on the input. Grammars used in compiling usually fall in the LR(1) class, with one symbol of lookahead at most.

Conflicts During Shift-Reduce Parsing



Example 4.38 : An ambiguous grammar can never be LR. For example, consider the dangling-else grammar (4.14) of Section 4.3:

$$\begin{array}{lcl} stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\ & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ & | & \textbf{other} \end{array}$$

If we have a shift-reduce parser in configuration

STACK	INPUT
$\cdots$ **if** *expr* **then** *stmt*	**else** $\cdots$ \$

we cannot tell whether **if** *expr* **then** *stmt* is the handle, no matter what appears below it on the stack. Here there is a shift/reduce conflict. Depending on what follows the **else** on the input, it might be correct to reduce **if** *expr* **then** *stmt* to *stmt*, or it might be correct to shift **else** and then to look for another *stmt* to complete the alternative **if** *expr* **then** *stmt* **else** *stmt*.

Note that shift-reduce parsing can be adapted to parse certain ambiguous grammars, such as the if-then-else grammar above. If we resolve the shift/reduce conflict on **else** in favor of shifting, the parser will behave as we expect, associating each **else** with the previous unmatched **then**. We discuss parsers for such ambiguous grammars in Section 4.8. □

Thank You