CHAPTER 4

Deterministic Top-Down Parsing

From now on, compared to the previous chapters, this book becomes less theoretical and more practical. Regarding parsing, while the previous chapter has explained its basic methodology in general, this chapter and the next chapter give a more realistic insight into parsing because they discuss its deterministic methods, which fulfill a central role in practice.

A deterministic top-down parser verifies that the tokenized version of a source program is syntactically correct by constructing its parse tree. Reading the input string representing the tokenized program from left to the right, the parser starts from its root and proceeds down toward the frontier denoted by the input string. To put it alternatively in terms of derivations, it builds up the leftmost derivation of this tokenized program starting from the start symbol. Frequently, this parser is based upon *LL grammars*, where the first *L* stands for the *l*eft-to-right scan of tokens and the second *L* stands for the *l*eftmost derivations. By making use of *predictive sets* constructed for rules in these grammars, the parser makes a completely deterministic selection of an applied rule during every leftmost derivation.

Based on LL grammars, we concentrate our attention on *predictive parsing*, which is perhaps the most frequently used deterministic top-down parsing method in practice. More specifically, first, we return to the popular recursive descent method (see Section 3.2), which frees us from explicitly implementing a pushdown list, and create its deterministic version. Then, we use the LL grammars and their predictive sets to make a predictive table used by a deterministic predictive table-driven parser, which explicitly implements a pushdown list. In this parser, any grammatical change only leads to a modification of the table while its control procedure remains unchanged, which is its key pragmatic advantage. We also explain how this parser handles the syntax errors to recover from them.

Synopsis. Section 4.1 introduces and discusses predictive sets and LL grammars. Then, based upon the LL grammar, Section 4.2 discusses the predictive recursive-descent and table-driven parsing.

4.1 Predictive Sets and LL Grammars

Consider a grammar, $G = ({}_G\Sigma, {}_GR)$, and a G-based top-down parser working with an input string w (see Section 3.2). Suppose that the parser has already found the beginning of the leftmost derivation for w, $S\ {}_{lm}\Rightarrow^* tAv$, where t is a prefix of w. More precisely, let $w = taz$, where a is the current input symbol, which follows t in w, and z is the suffix of w, which follows a. In tAv, A is the leftmost nonterminal to be rewritten in the next step. Assume that there exist several different A-rules, so the parser has to select one of them to continue the parsing process, and if the parser works deterministically, it cannot revise this selection later on. A *predictive parser* selects the right rule by predicting whether its application gives rise to a leftmost derivation of a string starting with a. To make this prediction, every rule $r \in {}_GR$ is accompanied with its *predictive set* containing all terminals that can begin a string resulting from a derivation whose first step is made by r. If the A-rules have their predictive sets pairwise disjoint, the parser deterministically selects the rule whose predictive set contains a. To construct the predictive sets, we first need the *first* and *follow* sets, described next.

First. The predictive set corresponding to $r \in {}_GR$ obviously contains the terminals that occur as the *first* symbol in a string derived from ***rhs***(r).

Definition 4.1 ***first*****.** Let $G = ({}_G\Sigma, {}_GR)$ be a grammar. For every string $x \in {}_G\Sigma^*$,

$$first(x) = \{a|\ x \Rightarrow^* w, \text{ where either } w \in {}_G\Delta^+ \text{ with } a = symbol(w, 1) \text{ or } w = \varepsilon = a\},$$

where $symbol(w, 1)$ denotes the leftmost symbol of w (see Section 1.1).

■

In general, *first* is defined in terms of $\Rightarrow$. However, as for every $w \in {}_G\Delta^*$, $x \Rightarrow^* w$ if and only if $x\ {}_{lm}\Rightarrow^* w$ (see Theorem 3.20), we could equivalently rephrase this definition in terms of the leftmost derivations, which play a crucial role in top-down parsing, as

$$first(x) = \{a|\ x\ {}_{lm}\Rightarrow^* w, \text{ where either } w \in {}_G\Delta^+ \text{ with } a = symbol(w, 1) \text{ or } w = \varepsilon = a\}$$

Furthermore, observe that if $x \Rightarrow^* \varepsilon$, where $x \in \Sigma^*$, then ε is in $first(x)$; as a special case, for $x = \varepsilon$, $first(\varepsilon) = \{\varepsilon\}$.

Next, we will construct the *first* sets for all strings contained in ${}_G\Delta \cup \{\boldsymbol{lhs}(r)|\ r \in {}_GR\} \cup \{y|\ y \in suffixes(\boldsymbol{rhs}(r)) \text{ with } r \in {}_GR\}$. We make use of some subsets of these *first* sets later in this section (see Algorithm 4.4 and Definition 4.5).

Goal. Construct $first(x)$ for every $x \in {}_G\Delta \cup \{\boldsymbol{lhs}(r)|\ r \in {}_GR\} \cup \{y|\ y \in suffixes(\boldsymbol{rhs}(r)) \text{ with } r \in {}_GR\}$.

Gist. Initially, set $first(a)$ to $\{a\}$ for every $a \in {}_G\Delta \cup \{\varepsilon\}$ because $a\ {}_{lm}\Rightarrow^0 a$ for these *a*s. Furthermore, if $A \to uw \in {}_GR$ with $u \Rightarrow^* \varepsilon$, then $A \Rightarrow^* w$, so add the symbols of $first(w)$ to $first(A)$ (notice that $u\ {}_{lm}\Rightarrow^* \varepsilon$ if and only if u is a string consisting of ε-nonterminals, determined by Algorithm 3.33). Repeat this extension of all the *first* sets in this way until no more symbols can be added to any of the *first* sets.

Algorithm 4.2 ***First Sets*****.**

Input • a grammar $G = ({}_G\Sigma, {}_GR)$.

Output • $first(u)$ for every $u \in {}_G\Delta \cup \{\boldsymbol{lhs}(r)|\ r \in {}_GR\} \cup \{y|\ y \in suffixes(\boldsymbol{rhs}(r)) \text{ with } r \in {}_GR\}$.

Method

```
begin
  set $first(a) = \{a\}$ for every $a \in {}_G\Delta \cup \{\varepsilon\}$, and
  set all the other constructed first sets to $\emptyset$;
  repeat
    if $r \in {}_GR$, $u \in prefixes(\boldsymbol{rhs}(r))$, and $u\ {}_{lm}\Rightarrow^* \varepsilon$ then
      extend $first(suffix(\boldsymbol{rhs}(r), |\boldsymbol{rhs}(r)| - |u|))$ by $first(symbol(\boldsymbol{rhs}(r), |u| + 1))$ and
      extend $first(\boldsymbol{lhs}(r))$ by $first(suffix(\boldsymbol{rhs}(r), |\boldsymbol{rhs}(r)| - |u|))$
  until no change
end.
```

Follow. The ε-rules deserve our special attention because they may give rise to derivations that erase substrings of sentential forms, and this possible erasure makes the parser's selection of the next applied rule even more difficult. Indeed, consider a grammar $G = ({}_G\Sigma, {}_GR)$ and $A \to x \in {}_GR$ with $x\ {}_{lm}\Rightarrow^* \varepsilon$. At this point, the parser needs to decide whether from A, it should make either $A\ {}_{lm}\Rightarrow x\ {}_{lm}\Rightarrow^* \varepsilon$ or a derivation that produces a non-empty string. To make this decision, we need to determine the set *follow*(A) containing all terminals that can follow A in any sentential form; if this set contains the current input token that is out of *first*(x), the parser simulates $A\ {}_{lm}\Rightarrow x\ {}_{lm}\Rightarrow^* \varepsilon$. In addition, we include ◀ into *follow*(A) to express that a sentential form ends with A (we assume that $◀ \notin {}_G\Sigma$).

Definition 4.3 ***follow*.** Let $G = ({}_G\Sigma, {}_GR)$ be grammar. For every $A \in {}_GN$,

$$follow(A) = \{a \in {}_G\Delta \cup \{◀\} \mid Aa \in substrings(F(G)\{◀\})\},$$

where $F(G)$ denotes the set of G's sentential forms (see Definition 3.1).

■

Goal. Construct *follow*(A) for every $A \in N$.

Gist. S is a sentential form, so we initialize *follow*(S) with $\{◀\}$. Consider any $B \to uAv \in {}_GR$. If a is a terminal in *first*(v), then $Aa \in substrings(F(G))$, so we add a to *follow*(A). In addition, if ε is in *first*(v), then $v\ {}_{lm}\Rightarrow^* \varepsilon$ and, consequently, $follow(B) \subseteq follow(A)$, so we add all symbols from *follow*(B) to *follow*(A). Keep extending all the *follow* sets in this way until no symbol can be added to any of them.

Algorithm 4.4 ***Follow Sets.***

Input • a grammar $G = ({}_G\Sigma, {}_GR)$;
• *first*(u) for every $u \in \{y \mid y \in suffixes(\mathbf{rhs}(r))$ with $r \in {}_GR\}$ (see Algorithm 4.2).

Output • *follow*(A), for every $A \in {}_GN$.

Method

```
begin
    set $follow(S) = \{◀\}$, and
    set all the other constructed follow sets to $\emptyset$;
    repeat
        if $r \in {}_GR$ and $Au \in suffixes(\mathbf{rhs}(r))$, where $A \in {}_GN$, $u \in {}_G\Sigma^*$ then
        begin
            add the symbols in $(first(u) - \{\varepsilon\})$ to $follow(A)$;
            if $\varepsilon \in first(u)$ then
                add the symbols in $follow(\mathbf{lhs}(r))$ to $follow(A)$
        end
    until no change
end.
```

Predictive sets. Based on the *first* and *follow* sets, we define the predictive sets as follows.

Definition 4.5 ***Predictive Set.*** The *predictive set* of each $r \in {}_GR$, symbolically denoted by *predictive-set*(r), is defined in the following way

• if $\varepsilon \notin first(\mathbf{rhs}(r))$, $predictive\text{-}set(r) = first(\mathbf{rhs}(r))$;

• if $\varepsilon \in$ *first*(***rhs***(r)), *predictive-set*(r) = (*first*(***rhs***(r)) − $\{\varepsilon\}$) $\cup$ *follow*(***lhs***(r)).

■

LL grammars. Reconsider the parsing situation described in the beginning of the present section. That is, a top-down parser based on a grammar $G = ({}_G\Sigma, {}_GR)$ has found the beginning of the leftmost derivation $S \ {}_{lm}\!\Rightarrow^* tAv$ for an input string taz, where a is the current input token, and it needs to select one of several different A-rules to rewrite A in tAv and, thereby, make another step. If for an A-rule r, $a \in$ *predictive-set*(r) and for any other A-rule p, $a \notin$ *predictive-set*(p), the parser obviously selects r. This idea leads to the next definition of LL grammars.

Definition 4.6 ***LL Grammars.*** A grammar $G = ({}_G\Sigma, {}_GR)$ is an *LL grammar* if for each $A \in N$, any two different A-rules, $p, q \in {}_GR$ and $p \neq q$, satisfy *predictive-set*(p) $\cap$ *predictive-set*(q) = $\emptyset$.

■

As already noted, in *LL grammars*, the first *L* stands for a *l*eft-to-right scan of tokens and the other *L* stands for a *l*eftmost derivation. Sometimes, in greater detail, the literature refers to the LL grammars as *LL*(1) *grammars* to point out that the top-down parsers based on these grammars always look at one input token during each step of the parsing process. Indeed, these grammars represent a special case of *LL*(k) *grammars*, where $k \geq 1$, which underlie parsers that make a k-tokens lookahead. In this introductory textbook, however, we discuss only LL(1) grammars and simply refer to them as LL grammars for brevity.

Case Study 11/35 ***LL Grammar.*** Return to the grammar obtained in Case Study 9/35 in Section 3.2. The present section demonstrates that the grammar is an LL grammar; hence, we denote this grammar by ${}_{LL}H = ({}_{_{LL}H}\Sigma, {}_{_{LL}H}R)$. Recall that ${}_{_{LL}H}R$ contains the following rules

$E \rightarrow TA$
$A \rightarrow \vee TA$
$A \rightarrow \varepsilon$
$T \rightarrow FB$
$B \rightarrow \wedge FB$
$B \rightarrow \varepsilon$
$F \rightarrow (E)$
$F \rightarrow i$

First (Algorithm 4.2). For each rule in this grammar, we construct *first*(u) for every $u \in \Delta \cup \{$***lhs***(r))| $r \in {}_{_{LL}H}R\} \cup \{y|\ y \in$ *suffix*es(***rhs***(r)) with $r \in {}_{_{LL}H}R\}$ by Algorithms 4.2. First, we set *first*(a) = $\{a\}$ for every $a \in \{i, (,), \vee, \wedge\} \cup \{\varepsilon\}$. Consider $F \rightarrow i$. By the **repeat** loop of Algorithms 4.2, as *first*(i) = $\{i\}$, we include i into *first*(F). As $i \in$ *first*(F) and $T \rightarrow FB \in {}_{_{LL}H}R$, we add i to *first*(FB) as well. Complete this construction as an exercise. However, to construct *predictive-set*(r) for each $r \in {}_{_{LL}H}R$, we only need $\{$*first*(***rhs***(r))| $r \in {}_{_{LL}H}R\}$, which represents a subset of all the *first* sets constructed by Algorithm 4.2. The members of $\{$*first*(***rhs***(r))| $r \in {}_{_{LL}H}R\}$ are listed in the second column of the table given in Figure 4.1.

Follow (Algorithm 4.4). Consider *first*(u) for each $u \in \{y|\ y \in$ *suffix*es(***rhs***(r)) with $r \in {}_GR\}$. We construct *follow*(A), for every $A \in {}_{_{LL}H}N$ by Algorithm 4.4 as follow. We initially have *follow*(E) = $\{$◀$\}$. As $F \rightarrow (E) \in {}_{_{LL}H}R$ and) $\in$ *first*()), we add) to *follow*(E). As $E \rightarrow TA \in {}_{_{LL}H}R$ and *follow*(E) contains ◀ and), we add ◀ and) to *follow*(A), too. Since $\varepsilon \in$ *first*(A),) and ◀ are in *follow*(E), and $E \rightarrow TA \in {}_{_{LL}H}R$, we add) and ◀ to *follow*(T). As $\vee \in$ *first*(A), we add $\vee$ to *follow*(T), too.

Complete this construction as an exercise. The third column of the table in Figure 4.1 contains *follow*(***lhs***(r)) for each $r \in {}_{LL}HR$.

Predictive sets (Definition 4.5). The fourth column of the table in Figure 4.1 contains *predictive-set*(r) for each $r \in {}_{LL}HR$. Notice that *follow*(***lhs***(r)) is needed to determine *predictive-set*(r) if $\varepsilon \in$ *first*(***rhs***(r)) because at this point *predictive-set*(r) = (*first*(***rhs***(r)) − {ε}) ∪ *follow*(***lhs***(r)); if $\varepsilon \notin$ *first*(***rhs***(r)), it is not needed because *predictive-set*(r) = *first*(***rhs***(r)) (see Definition 4.5). Take, for instance, $E \rightarrow TA \in {}_{LL}HR$ with *first*(TA)) = {i, (}. As $\varepsilon \notin$ *first*(***rhs***(r)), *predictive-set*($E \rightarrow TA$) = *first*(TA) = {i, (}. Consider $A \rightarrow \varepsilon \in {}_{LL}HR$ with *first*(ε) = {ε} and *follow*(***lhs***(A)) = {), ◀}. As $\varepsilon \in$ *first*(***rhs***($A \rightarrow \varepsilon$)), *predictive-set*($A \rightarrow \varepsilon$) = *first*(ε) − {ε} ∪ *follow*(***lhs***(A)) = ∅ ∪ {), ◀} = {), ◀}. Complete this construction as an exercise.

LL grammar. Observe that *predictive-set*($A \rightarrow \vee\ TA$) ∩ *predictive-set*($A \rightarrow \varepsilon$) = {∨} ∩ {), ◀} = ∅. Analogously, *predictive-set*($B \rightarrow \wedge\ FB$) ∩ *predictive-set*($B \rightarrow \varepsilon$) = ∅ and *predictive-set*($F \rightarrow (E)$) ∩ *predictive-set*($F \rightarrow i$) = ∅. Thus, ${}_{LL}H$ is an LL grammar.

Rule *r* ***in*** ${}_{LL}H$	*first*(***rhs***(r))	*follow*(***lhs***(r))	*predictive-set*(r)
$E \rightarrow TA$	i, (	), ◀	i, (
$A \rightarrow \vee\ TA$	∨	), ◀	∨
$A \rightarrow \varepsilon$	ε	), ◀	), ◀
$T \rightarrow FB$	i, (	∨,), ◀	i, (
$B \rightarrow \wedge\ FB$	∧	∨,), ◀	∧
$B \rightarrow \varepsilon$	ε	∨,), ◀	∨,), ◀
$F \rightarrow (E)$	(	∧, ∨,), ◀	(
$F \rightarrow i$	i	∧, ∨,), ◀	i

Figure 4.1 *first*, *follow* ***and*** *predictive* **sets for** ${}_{LL}H$**.**

■

As demonstrated in the following section, the LL grammars underlie the most important deterministic top-down parsers, so their significance is indisputable. Not all grammars represent LL grammars, however. Some non-LL grammars can be converted to equivalent LL grammars by an algorithm that removes an undesirable grammatical phenomenon as illustrated next by a grammar transformation called *left factoring*. Unfortunately, in general, for some grammars, there exist no equivalent LL grammars (see Exercises 4.6 and 4.7).

Left Factoring. Consider a grammar $G = ({}_G\Sigma, {}_GR)$ such that R contains two different A-rules of the form $A \rightarrow zx$ and $A \rightarrow zy$ in R. If G derives a non-empty string of terminals from z, the predict sets of these rules have some terminals in common, so G is no LL grammar. Therefore, we transform G to an equivalent *left-factored* grammar in which any pair of different A-rules r and p, which are not ε-rules, satisfies *symbol*(***rhs***(r), 1) ≠ *symbol*(***rhs***(p), 1), so the above grammatical phenomenon is a priori ruled out.

Goal. Conversion of any grammar to an equivalent left-factored grammar.

Gist. Consider a grammar $G = ({}_G\Sigma, {}_GR)$. Let $A \rightarrow zx$ and $A \rightarrow zy$ be two different A-rules in ${}_GR$, where $z \neq \varepsilon$ and *prefixes*{x} ∩ *prefix*es{y} = {ε}. Then, replace these two rules with these three rules $A \rightarrow zA_{lf}$, $A_{lf} \rightarrow x$, and $A_{lf} \rightarrow y$, where A_{lf} is a new nonterminal. Repeat this modification until the resulting grammar is left-factored. In essence, the resulting grammar somewhat defers the original derivation process to select the right rule. More specifically, by $A \rightarrow zA_{lf}$, it first changes

A to zA_{lf}. Then, it derives a terminal string from z. Finally, based upon the current input symbol, it decides whether to apply $A_{lf} \to x$ or $A_{lf} \to y$ to continue the derivation.

Algorithm 4.7 ***Left-Factored Grammar.***

Input • a grammar $G = ({}_G\Sigma, {}_GR)$.

Output • a left-factored grammar $H = ({}_H\Sigma, {}_HR)$ equivalent to G.

Method

```
begin
    $_H\Sigma := {}_G\Sigma$;
    $_HR := {}_GR$;
    repeat
        if $p, r \in {}_GR, p \neq r$, $\textbf{\textit{lhs}}(p) = \textbf{\textit{lhs}}(r)$, and $\textit{prefixes}(\textbf{\textit{rhs}}(p)) \cap \textit{prefixes}(\textbf{\textit{rhs}}(r)) \neq \{\varepsilon\}$ then
        begin
            introduce a new nonterminal $A_{lf}$ to $_HN$;
            in $_HR$, replace $p$ and $r$ with these three rules
            $\textbf{\textit{lhs}}(p) \to xA_{lf}$, $A_{lf} \to \textit{suffix}(\textbf{\textit{rhs}}(p), |\textbf{\textit{rhs}}(p)| - |x|)$, and $A_{lf} \to \textit{suffix}(\textbf{\textit{rhs}}(r), |\textbf{\textit{rhs}}(r)| - |x|)$
                where $x$ is the longest string in $\textit{prefixes}(\textbf{\textit{rhs}}(p)) \cap \textit{prefixes}(\textbf{\textit{rhs}}(r))$
        end
    until no change;
end.
```

Case Study 12/35 ***Left Factoring.*** Consider the next grammar, which defines the ***integer*** and ***real*** FUN declarations.

⟨declaration part⟩ → ***declaration*** ⟨declaration list⟩
⟨declaration list⟩ → ⟨declaration⟩; ⟨declaration list⟩
⟨declaration list⟩ → ⟨declaration⟩
⟨declaration⟩ → ***integer*** ⟨variable list⟩
⟨declaration⟩ → ***real*** ⟨variable list⟩
⟨variable list⟩ → *i*, ⟨variable list⟩
⟨variable list⟩ → *i*

Because the predictive sets of ⟨declaration list⟩-rules and ⟨variable list⟩-rules are not disjoint, this grammar is not an LL grammar. Apply Algorithm 4.7 to this grammar. This algorithm replaces

⟨declaration list⟩ → ⟨declaration⟩; ⟨declaration list⟩
⟨declaration list⟩ → ⟨declaration⟩

with these three rules

⟨declaration list⟩ → ⟨declaration⟩⟨more declarations⟩
⟨more declarations⟩ → ; ⟨declaration list⟩
⟨more declarations⟩ → ε

where ⟨more declarations⟩ is a new nonterminal. Analogously, it replaces

⟨variable list⟩ → *i*, ⟨variable list⟩
⟨variable list⟩ → *i*

with

⟨variable list⟩ → *i* ⟨more variables⟩
⟨more variables⟩ → , ⟨variable list⟩
⟨more variables⟩ → ε

where ⟨more variables⟩ is a new nonterminal. As a result, we obtain this LL grammar:

⟨declaration part⟩ → ***declaration*** ⟨declaration list⟩
⟨declaration list⟩ → ⟨declaration⟩ ⟨more declarations⟩
⟨more declarations⟩ → ; ⟨declaration list⟩
⟨more declarations⟩ → ε
⟨declaration⟩ → ***integer*** ⟨variable list⟩
⟨declaration⟩ → ***real*** ⟨variable list⟩
⟨variable list⟩ → *i* ⟨more variables⟩
⟨more variables⟩ → , ⟨variable list⟩
⟨more variables⟩ → ε

■

4.2 Predictive Parsing

In this section, we first reconsider the recursive-descent parsing method (see Section 3.2). Specifically, based upon LL grammars, we create its deterministic version. Then, we use the LL grammars and their predictive sets to create predictive tables and deterministic top-down parsers driven by these tables. Finally, we explain how to handle errors in predictive parsing.

Predictive Recursive-Descent Parsing

The present section discusses the deterministic version of recursive-descent parsers, described in Section 3.2 with a detailed illustration given in Case Study 8/35 *Recursive-Descent Parser*. This determinism is achieved by the predictive sets constructed in the previous section, hence *predictive recursive-descent parsing*. That is, for an LL grammar $G = ({}_G\Sigma, {}_GR)$, we construct a *G-based predictive recursive-descent parser*, denoted by *G-rd-parser*. We describe this construction in the following specific case study, which makes use of these programming routines

- **ACCEPT** and **REJECT**
- **INPUT-SYMBOL**
- **RETURN-SYMBOL**

ACCEPT and **REJECT** are announcements described in Convention 1.8. In the next parser as well as in all the other parsers given in the rest of this book, **ACCEPT** announces a successful completion of the parsing process while **REJECT** announces a syntax error in the parsed program.

INPUT-SYMBOL has the meaning described in Definition 3.14—that is, it is a lexical-analysis function that returns the input symbol, representing the current token, after which the input string is advanced to the next symbol.

Definition 4.8 RETURN-SYMBOL. Procedure **RETURN-SYMBOL** returns the current token in **INPUT-SYMBOL** back to the lexical analyzer, which thus sets **INPUT-SYMBOL** to this returned token during its next call.

■

The following parser makes use of **RETURN-SYMBOL** when a rule selection is made by using a token from the *follow* set; at this point, this token is not actually used up, so when the next call of **INPUT-SYMBOL** occurs, the parser wants the lexical analyzer to set **INPUT-SYMBOL** to this very same token, which is accomplished by **RETURN-SYMBOL**.

Case Study 13/35 ***Predictive Recursive Descent Parser.*** Reconsider the LL grammar $_{LL}H$ discussed in Case Study 11/35. Figure 4.2 repeats its rules together with the corresponding predictive sets.

Rule** r **in $_{LL}H$	*predictive-set*(r)
$E \rightarrow TA$	i, (
$A \rightarrow \vee TA$	$\vee$
$A \rightarrow \varepsilon$	), ◀
$T \rightarrow FB$	i, (
$B \rightarrow \wedge FB$	$\wedge$
$B \rightarrow \varepsilon$	$\vee$,), ◀
$T \rightarrow F$	i, (
$F \rightarrow (E)$	(
$F \rightarrow i$	i

Figure 4.2 ***Rules in*** $_{LL}H$ ***and Their Predictive Sets.***

Just like in Case Study 8/35, we now construct an $_{LL}H$-based recursive-descent parser $_{LL}H$*-rd-parser* as a collection of Boolean *rd-functions* corresponding to the nonterminals in $_{LL}H$. In addition, however, we make use of the predictive sets of $_{LL}H$'s rules so this parser always selects the next applied rule deterministically.

Consider the start symbol E and the only E-rule $E \rightarrow TA$ with *predictive-set*($E \rightarrow TA$) = $\{i, (\}$. As a result, *rd-function* E has this form

```
function E: Boolean ;
begin
   E := false;
   if INPUT-SYMBOL in {i, (} then
      begin
         RETURN-SYMBOL;
         if T then
            if A then E := true
      end
end
```

Consider the two A-rules that include $A \rightarrow \vee TA$ with *predictive-set*($A \rightarrow \vee TA$) = $\{\vee\}$ and $A \rightarrow \varepsilon$ with *predictive-set*($A \rightarrow \varepsilon$) = {), ◀}. Therefore, *rd-function* A selects $A \rightarrow \vee TA$ if the input symbol is $\vee$ and this function selects $A \rightarrow \varepsilon$ if this symbol is in {), ◀}. If the input symbol differs from $\vee$,), or ◀, the syntax error occurs. Thus, *rd-function* A has this form

```
function A: Boolean;
begin
   A := false;
```

```
    case INPUT-SYMBOL of
    $\vee$:  if $T$ then {$A \to \vee TA$}
            if $A$ then
                $A$ := true;
    ), ◀:
        begin {$A \to \varepsilon$}
            RETURN-SYMBOL; {) and ◀ are in follow($A$)}
            $A$ := true
        end
    end {case};
end
```

There exists a single T-rule of the form $T \to FB$ in ${}_{LL}H$ with *predictive-set*($T \to FB$) = {i, (}. Its *rd-function* T, given next, is similar to *rd-function E*.

```
function $T$: Boolean ;
begin
    $T$ := false;
    if INPUT-SYMBOL in {$i$, (} then
        begin
            RETURN-SYMBOL;
            if $F$ then
                if $B$ then
                    $T$ := true
        end
end
```

Consider the two B-rules $B \to \wedge FB$ with *predictive-set*($B \to \wedge FB$) = {$\wedge$} and $B \to \varepsilon$ with *predictive-set*($B \to \varepsilon$) = {$\vee$,), ◀}. Therefore, *rd-function B* selects $B \to \wedge FB$ if the input symbol is $\wedge$, and it selects $B \to \varepsilon$ if this symbol is in {$\vee$,), ◀}. The *rd-function B* has thus this form

```
function $B$: Boolean ;
begin
    $B$ := false;
    case INPUT-SYMBOL of
    $\wedge$:  if $F$ then {$B \to \wedge FB$}
            if $B$ then
                $B$ := true;
    $\vee$, ), ◀:
        begin {$B \to \varepsilon$}
            RETURN-SYMBOL; {$\vee$, ), and ◀ are in follow($B$)}
            $B$ := true
        end
    end {case};
end
```

Finally, consider the F-rules $F \to (E)$ with *predictive-set*($F \to (E)$) = {(} and $F \to i$ with *predictive-set*($F \to i$) = {i}. Therefore, *rd-function F* selects $F \to (E)$ if the input symbol is (and this function selects $F \to i$ if the input symbol is i. If the input symbol differs from (or i, the syntax error occurs. Thus, *rd-function F* has this form

```
function $F$: Boolean ;
begin
    $F$ := false;
    case INPUT-SYMBOL of
    (:  if $E$ then {$F \rightarrow (E)$}
            if INPUT-SYMBOL = ')' then
                $F$ := true;
    $i$:  $F$ := true
    end {case};
end
```

Having these functions in ${}_{LL}H$*-rd-parser*, its main body is based on the following simple **if** statement, which decides whether the source program is syntactically correct by the final Boolean value of *rd-function E*:

```
if $E$ then
    ACCEPT
else
    REJECT
```

As an exercise, reconsider each application of **RETURN-SYMBOL** in the above functions and explain its purpose in detail.

■

Predictive Table-Driven Parsing

In the predictive recursive-descent parsing, for every nonterminal and the corresponding rules, there exists a specific Boolean function, so any grammatical change usually necessitates reprogramming several of these functions. Therefore, unless a change of this kind is ruled out, we often prefer an alternative *predictive table-driven parsing* based on a single general control procedure that is completely based upon a predictive table. At this point, a change of the grammar only implies an adequate modification of the table while the control procedure remains unchanged. As opposed to the predictive recursive-descent parsing, however, this parsing method maintains a pushdown explicitly, not implicitly via recursive calls like in predictive recursive-descent parsing.
Consider an LL grammar $G = ({}_G\Sigma, {}_GR)$. Like a general top-down parser (see Algorithm 3.12), a *G-based predictive table-driven parser*, denoted by *G-td-parser*, is underlain by a pushdown automaton, $M = ({}_M\Sigma, {}_MR)$ (see Definition 3.7). However, strictly mathematical specification of M, including all its rules in ${}_MR$, would be unbearably tedious and difficult-to-follow. Therefore, to make this specification brief, clear, and easy to implement, we describe *G-td-parser* as an algorithm together with a *G-based predictive table*, denoted by *G-predictive-table*, by which *G-td-parser* determines a move from every configuration. *G-predictive-table*'s rows and columns are denoted by the members of ${}_GN$ and ${}_G\Delta \cup \{◀\}$, respectively. Each of its entry contains a member of ${}_GR \cup \{☹\}$. More precisely, for each $A \in {}_GN$ and each $t \in {}_G\Delta$, if there exists $r \in R$ such that ***lhs***$(r) = A$ and $t \in$ *predictive-set(r)*, *G-predictive-table*$[A, t] = r$; otherwise, *G-predictive-table*$[A, t]$ = ☹, which means a syntax error. Making use of *G-predictive-table*, *G-td-parser M* works with the input string, representing the tokenized source program, as described next.

Goal. Construction of a *G-td-parser* $M = ({}_M\Sigma, {}_MR)$ for an LL grammar $G = ({}_G\Sigma, {}_GR)$.

Gist. A predictive table-driven parser always performs a computational step based on the pushdown top symbol X and the current input symbol a. This step consists in one of the following actions:

- if $X \in {}_G\Delta$ and $X = a$, the pushdown top symbol is a terminal coinciding with the input token, so *M pops* the pushdown by removing X from its top and, simultaneously, advances to the next input symbol, occurring behind a;
- if $X \in {}_G\Delta$ and $X \neq a$, the pushdown top symbol is a terminal that differs from the input token, so the parser announces an error by **REJECT**;
- if $X \in {}_GN$ and *G-predictive-table*[X, a] = r with $r \in {}_GR$, where ***lhs***(r) = X, *M expands* the pushdown by replacing X with *reversal*(***rhs***(r)).
- if $X \in {}_GN$ and *G-predictive-table*[X, a] = ☹, M announces an error by **REJECT**;
- if ▶ = X and a = ◀, the pushdown is empty and the input token string is completely read, so M halts and announces a successful completion of parsing by **ACCEPT**.

Before going any further in the predictive table-driven parsing, we introduce Convention 4.9 concerning the way we frequently describe the configuration of a parser throughout the rest of this book. Specifically, Algorithm 4.11, which represents *G-td-parser*, makes use of this convention several times.

Convention 4.9. Consider a parser's configuration, ▶u◆v◀, where u and v are the current *pd* and *ins*, respectively (see the notes following Convention 3.8 for *pd* and *ins*). For $i = 1, \dots, |u| + 1$, pd_i denotes the ith topmost symbol stored in the pushdown, so pd_1 refers to the very topmost pushdown symbol, which occurs as the rightmost symbol of *pd*, while $pd_{|u|+1}$ denotes ▶, which marks the *bottom* of the pushdown. Furthermore, for $h = 1, \dots, |v| + 1$, ins_h denotes the hth input symbol in *ins*, so ins_1 refers to the current input symbol, which occurs as the leftmost symbol of *ins*, while $ins_{|v|+1}$ denotes ◀, which marks the *ins* end. Making use of this notation, we can express ▶u◆v◀ as

$$\blacktriangleright pd_{|u|} \dots pd_2\, pd_1 \blacklozenge ins_1\, ins_2 \dots ins_{|v|} \blacktriangleleft$$

■

Throughout the rest of this section, we also often make use of operations **EXPAND** and **POP**, described next.

Definition 4.10 *Operations* EXPAND *and* POP. Operations **EXPAND** and **POP** modify *G-td-parser*'s current *pd* top as follows

EXPAND($A \to x$) expands the pushdown according to a rule $A \to x$; in terms of Algorithm 3.12, **EXPAND**($A \to x$) thus means an application of A◆ $\to$ *reversal*(x)◆.

POP pops the topmost symbol from the current pushdown.

■

Algorithm 4.11 *Predictive Table-Driven Parsing.*

Input • a *G-predictive-table* of an LL grammar $G = ({}_G\Sigma, {}_GR)$;
• an input string *ins* = w◀ with $w \in {}_G\Delta^*$.

Output • **ACCEPT** if $w \in L(G)$, and **REJECT** if $w \notin L(G)$.

Method

```
begin

    initialize pd with ▶$S$; {initially pd = ▶$S$}
```

```
set $i$ to 1; {initially the current input symbol is $ins_1$—the leftmost symbol of $w◀$}

repeat
    case $pd_1$ of

        in $_G\Delta$: if $pd_1 = ins_i$ then {$pd_1$ is a terminal}
            begin
                POP; {pop $pd$}
                $i := i + 1$ {advance to the next input symbol}
            end
            else
                REJECT; {$pd_1 \neq ins_i$}
        in $_GN$: if $G$-precedence-table[$pd_1$, $ins_i$] = $r$ with $r \in {}_GR$ then {$pd_1$ is a nonterminal}
                EXPAND($r$)
            else
                REJECT {the table entry is ☹};
        ▶: if $ins_i$ = ◀ then
                ACCEPT
            else
                REJECT {the pushdown is empty but the input string is not}
    end{case};

until ACCEPT or REJECT

end.
```

Case Study 14/35 ***Predictive Table-Driven Parser.*** Return to grammar $_{LL}H$ (see Case Study 12/35) defined as:

1: $E \rightarrow TA$
2: $A \rightarrow \vee TA$
3: $A \rightarrow \varepsilon$
4: $T \rightarrow FB$
5: $B \rightarrow \wedge FB$
6: $B \rightarrow \varepsilon$
7: $T \rightarrow F$
8: $F \rightarrow (E)$
9: $F \rightarrow i$

By using the predictive sets corresponding to these rules (see Figure 4.1), we construct the predictive table of this grammar (see Figure 4.3).

	$\vee$	$\wedge$	(	)	i	◀
E	☹	☹	1	☹	1	☹
A	2	☹	☹	3	☹	3
T	☹	☹	4	☹	4	☹
B	6	5	☹	6	☹	6
F	☹	☹	8	☹	9	☹
▶	☹	☹	☹	☹	☹	☺

Figure 4.3 ***$_{LL}H$-predictive-table.***

Observe that from Algorithm 4.11 and the $_{LL}H$*-predictive-table,* we could obtain a strictly mathematical specification of *G-td-parser* as a pushdown automaton $M = ({}_M\Sigma, {}_MR)$ according to Definition 3.7. In essence, M has the form of the parser constructed in Algorithm 3.12 in Section 3.2. That is, M makes **ACCEPT** by $\blacktriangleright\blacklozenge\blacktriangleleft \to \blacklozenge$. If a pair of the *pd* top X and the current input symbol b leads to **REJECT**, ${}_MR$ has no rule with $X\blacklozenge b$ on its left-hand side. It performs **POP** by rules of the form $a\blacklozenge a \to \blacklozenge$ for each $a \in {}_{LL}H\Delta$. Finally, if a pair of the *pd* top and the current input symbol leads to **EXPAND**(r), where r is a rule from ${}_GR$, M makes this expansion according to r by $A\blacklozenge \to reversal(x)\blacklozenge$. For instance, as ${}_{LL}H$*-predictive-table*$[E, i] = 1$ and $1\colon E \to TA \in {}_{LL}HR$, ${}_MR$ has $E\blacklozenge \to AT\blacklozenge$ to make an expansion according to 1. A completion of this mathematical specification of M is left as an exercise. However, not only is this completion a tedious task, but also the resulting parser M specified in this way is difficult to understand what it actually does with its incredibly many rules. That is why, as already pointed out, we always prefer the description of a parser as an algorithm together with a parsing table throughout the rest of this book.

Configuration	*Table Entry and Rule*	*Sentential Form*
		$\underline{E}$
$\blacktriangleright E\blacklozenge i \wedge i \vee i\blacktriangleleft$	$[E, i] = 1\colon E \to TA$	$\underline{T}A$
$\blacktriangleright AT\blacklozenge i \wedge i \vee i\blacktriangleleft$	$[T, i] = 4\colon T \to FB$	$\underline{F}BA$
$\blacktriangleright ABF\blacklozenge i \wedge i \vee i\blacktriangleleft$	$[F, i] = 9\colon F \to i$	$i\,\underline{B}A$
$\blacktriangleright ABi\blacklozenge i \wedge i \vee i\blacktriangleleft$		
$\blacktriangleright AB\blacklozenge \wedge i \vee i\blacktriangleleft$	$[B, \wedge] = 5\colon B \to \wedge FB$	$i \wedge \underline{F}BA$
$\blacktriangleright ABF\wedge\blacklozenge \wedge i \vee i\blacktriangleleft$		
$\blacktriangleright ABF\blacklozenge i \vee i\blacktriangleleft$	$[F, i] = 9\colon F \to i$	$i \wedge i\underline{B}A$
$\blacktriangleright ABi\blacklozenge i \vee i\blacktriangleleft$		
$\blacktriangleright AB\blacklozenge \vee i\blacktriangleleft$	$[B, \vee] = 6\colon B \to \varepsilon$	$i \wedge i\underline{A}$
$\blacktriangleright A\blacklozenge \vee i\blacktriangleleft$	$[A, \vee] = 2\colon A \to \vee TA$	$i \wedge i \vee \underline{T}A$
$\blacktriangleright AT\vee\blacklozenge \vee i\blacktriangleleft$		
$\blacktriangleright AT\blacklozenge i\blacktriangleleft$	$[T, i] = 7\colon T \to F$	$i \wedge i \vee \underline{F}A$
$\blacktriangleright AF\blacklozenge i\blacktriangleleft$	$[F, i] = 9\colon F \to i$	$i \wedge i \vee i\underline{A}$
$\blacktriangleright Ai\blacklozenge i\blacktriangleleft$		
$\blacktriangleright A\blacklozenge\blacktriangleleft$	$[A, \blacktriangleleft] = 3\colon A \to \varepsilon$	$i \wedge i \vee i$
$\blacktriangleright\blacklozenge\blacktriangleleft$	$[\blacktriangleright, \blacktriangleleft] = ☺$	

Figure 4.4 ***Predictive Table-Driven Parsing in Terms of Leftmost Derivation.***

Configuration	*Rule*	*Parse Tree*
		$\underline{E}$
$\blacktriangleright E\blacklozenge i \wedge i \vee i\blacktriangleleft$	1	$E\langle \underline{T}A\rangle$
$\blacktriangleright AT\blacklozenge i \wedge i \vee i\blacktriangleleft$	4	$E\langle T\langle \underline{F}B\rangle A\rangle$
$\blacktriangleright ABF\blacklozenge i \wedge i \vee i\blacktriangleleft$	9	$E\langle T\langle F\langle i\rangle \underline{B}\rangle A\rangle$
$\blacktriangleright ABi\blacklozenge i \wedge i \vee i\blacktriangleleft$		
$\blacktriangleright AB\blacklozenge \wedge i \vee i\blacktriangleleft$	5	$E\langle T\langle F\langle i\rangle B\langle \wedge \underline{F}B\rangle\rangle A\rangle$
$\blacktriangleright ABT\wedge\blacklozenge \wedge i \vee i\blacktriangleleft$		
$\blacktriangleright ABF\blacklozenge i \vee i\blacktriangleleft$	9	$E\langle T\langle F\langle i\rangle B\langle \wedge F\langle i\rangle \underline{B}\rangle\rangle A\rangle$
$\blacktriangleright ABi\blacklozenge i \vee i\blacktriangleleft$		
$\blacktriangleright AB\blacklozenge \vee i\blacktriangleleft$	6	$E\langle T\langle F\langle i\rangle B\langle \wedge F\langle i\rangle B\langle\varepsilon\rangle\rangle\rangle \underline{A}\rangle$
$\blacktriangleright A\blacklozenge \vee i\blacktriangleleft$	2	$E\langle T\langle F\langle i\rangle B\langle \wedge F\langle i\rangle B\langle\varepsilon\rangle\rangle\rangle A\langle \vee \underline{T}A\rangle\rangle$
$\blacktriangleright AT\vee\blacklozenge \vee i\blacktriangleleft$		
$\blacktriangleright AT\blacklozenge i\blacktriangleleft$	7	$E\langle T\langle F\langle i\rangle B\langle \wedge F\langle i\rangle B\langle\varepsilon\rangle\rangle\rangle A\langle \vee T\langle \underline{F}\rangle A\rangle\rangle$
$\blacktriangleright AF\blacklozenge i\blacktriangleleft$	9	$E\langle T\langle F\langle i\rangle B\langle \wedge F\langle i\rangle B\langle\varepsilon\rangle\rangle\rangle A\langle \vee T\langle F\langle i\rangle\rangle \underline{A}\rangle\rangle$
$\blacktriangleright Ai\blacklozenge i\blacktriangleleft$		
$\blacktriangleright A\blacklozenge\blacktriangleleft$	3	$E\langle T\langle F\langle i\rangle B\langle \wedge F\langle i\rangle B\langle\varepsilon\rangle\rangle\rangle A\langle \vee T\langle F\langle i\rangle\rangle A\langle\varepsilon\rangle\rangle\rangle$
$\blacktriangleright\blacklozenge\blacktriangleleft$	☺	

Figure 4.5 ***Predictive Table-Driven Parsing in Terms of Parse Tree.***

Consider $i \wedge i \vee i$ as the input string, which ${}_{LL}H$ generates in this leftmost way

$$\begin{array}{lll}
\underline{E} & {}_{lm}\Rightarrow \underline{T}A & [1] \\
 & {}_{lm}\Rightarrow \underline{F}BA & [4] \\
 & {}_{lm}\Rightarrow i\underline{B}A & [9] \\
 & {}_{lm}\Rightarrow i \wedge \underline{F}BA & [5] \\
 & {}_{lm}\Rightarrow i \wedge i\underline{B}A & [9] \\
 & {}_{lm}\Rightarrow i \wedge i\underline{A} & [6] \\
 & {}_{lm}\Rightarrow i \wedge i \vee \underline{T}A & [2] \\
 & {}_{lm}\Rightarrow i \wedge i \vee \underline{F}A & [7] \\
 & {}_{lm}\Rightarrow i \wedge i \vee i\underline{A} & [9] \\
 & {}_{lm}\Rightarrow i \wedge i \vee i & [3]
\end{array}$$

With $i \wedge i \vee i$, *${}_{LL}H$-td-parser* works as described in the tables given in Figures 4.4 and 4.5. In the first column of both tables, we give *${}_{LL}H$-td-parser*'s configurations, each of which has the form ▶*x*◆*y*◀, where ▶*x* and *y*◀ are the current *pd* and *ins*, respectively. In Figure 4.4, the second and the third columns describe the parsing of $i \wedge i \vee i$ in terms of the leftmost derivation. That is, its second column describes *${}_{LL}H$-predictive-table*'s entries and the rules they contain. The third column gives the sentential forms derived in the leftmost derivation. Figure 4.5 reformulates this description in terms of the parse tree $pt(E\ {}_{lm}\Rightarrow i \wedge i \vee i)$. In the second column of Figure 4.5, we give the labels of the applied rules. The third column gives a step-by-step construction of $pt(E\ {}_{lm}\Rightarrow i \wedge i \vee i)$ so that we always underline the node to which a rule tree is being attached during the corresponding step of the parsing process.

■

Handling Errors

Of course, a predictive parser struggles to handle each syntax error occurrence as best as it can. It carefully diagnoses the error and issues an appropriate error message. Then, it recovers from the error by slightly modifying *pd* or skipping a short prefix of *ins*. After the recovery, the parser resumes its analysis of the syntactically erroneous program, possibly discovering further syntax errors. Most crucially, no matter what it does during handling errors, the parser has to avoid any endlessly repeated routines so all the tokenized source string is eventually processed. Throughout the rest of this section, we sketch two simple and popular error recovery methods in terms of a *G-td-parser*, where $G = ({}_G\Sigma, {}_GR)$ is an LL grammar while leaving their straightforward adaptation for *G-rd-parser* as an exercise.

Panic-mode error recovery. For every nonterminal $A \in {}_GN$, we define a set of synchronizing input symbols as *synchronizing-set*(A) = *first*(A) ∪ *follow*(A). Supposing that *G-td-parser* occurs in a configuration with $pd_1 = X$ and $ins_1 = a$ when a syntax error occurs, this error recovery method handles the error, in essence, as follows.

- If $pd_1 \in {}_GN$ and *G-precedence-table*[pd_1, ins_1] = ☹, skip the input symbols until the first occurrence of t that is in *synchronizing-set*(pd_1). If $t \in$ *first*(pd_1), resume parsing according to *G-precedence-table*[pd_1, t] without any further change. If $t \in$ *follow*(pd_1), pop pd_1 from *pd* and resume parsing.

- If $pd_1 \in {}_G\Delta$ and $pd_1 \neq a$, pop pd_1 and resume parsing.

As an exercise, we discuss further variants of this method, most of which are based on alternative definitions of the synchronizing sets.

Ad-hoc recovery. In this method, we design a specific recovery routine, **RECOVER**[*X*, *t*], for every error-implying pair of a top pushdown symbol *X* and an input symbol *t*. That is, if *X* and *t* are two different input symbols, we make **RECOVER**[*X*, *t*]. We also design **RECOVER**[*X*, *t*] if $X \in N \cup \{▶\}$ and *G-predictive-table*[*X*, *t*] = ☹. **RECOVER**[*X*, *t*] figures out the most probable mistake that leads to the given syntax error occurrence, issues an error message, and decides on a recovery procedure that takes a plausible action to resume parsing. Typically, **RECOVER**[*X*, *t*] skips a tiny portion of *ins* or modifies *pd* by changing, inserting, or deleting some symbols; whatever it does, however, the recovery procedure needs to guarantee that this modification surely avoids any infinite loop so that the parser eventually proceeds its normal process. *G-td-parser* with this type of error recovery works just like Algorithm 4.11 except that if the error occurs in a configuration such that either *G-predictive-table*[pd_1, ins_1] = ☹ or $ins_1 \neq pd_1$ with $pd_1 \in {}_G\Delta$, it performs **RECOVER**[pd_1, ins_1]. In this way, *G-td-parser* detects and recovers from the syntax error, after which it resumes the parsing process. Of course, once *G-td-parser* detects a syntax error and recovers from it, it can never proclaim that the program is syntactically correct later during the resumed parsing process. That is, even if *G-td-parser* eventually reaches *G-predictive-table*[pd_1, ins_1] = *G-predictive-table*[▶,◀] = ☺ after a recovery from a syntax error, it performs **REJECT**, not **ACCEPT**.

Case Study 15/35 ***Error Recovery in Predictive Parsing.*** Return to the grammar ${}_{LL}H$ discussed in Case Study 14/35. Consider)*i* as the input string. As obvious, $)i \notin L({}_{LL}H)$. With)*i*, ${}_{LL}H$*-td-parser* immediately interrupts its parsing because entry ${}_{LL}H$*-predictive-table*[*E*,)] equals ☹. Reconsidering this interruption in detail, we see that if *E* occurs on the pushdown top and) is the input token, then the following error-recovery routine is appropriate.

name:	**RECOVER**[*E*,)]
diagnostic:	) starts *ins* or no expression occurs between parentheses
recovery:	If) starts *ins*, skip it. If no expression occurs between (and), remove *E* from *pd* and skip) in *ins*; in this way, *pd* and *ins* are changed so that the missing expression problem can be ignored. Resume parsing.

Designing the other error-recovery routines is left as an exercise. With)*i*, the parser works as described in Figure 4.6, in which the error-recovery information is pointed up.

Configuration	*Table Entry*	*Sentential Form*
		E
▶*E*◆) *i*◀	[*E*,)] = ☹, so **RECOVER**[*E*,)]	
▶*E*◆*i*◀	[*E*, *i*] = 1: $E \to TA$	*TA*
▶*AT*◆*i*◀	[*T*, *i*] = 4: $T \to FB$	*FBA*
▶*ABF*◆*i*◀	[*F*, *i*] = 9: $F \to i$	*iBA*
▶*AB i*◆*i*◀		
▶*AB*◆◀	[*B*,◀] = 6: $B \to \varepsilon$	*iA*
▶*A*◆◀	[*A*,◀] = 3: $A \to \varepsilon$	*i*
▶◆◀		

Figure 4.6 ***Predictive Table-Driven Parsing with Error Recovery.***

■

Even if a parser handles syntax errors very sophisticatedly, it sometimes repeatedly detects more and more errors within a relatively short, but heavily erroneous program construct, such as a single statement. At this point, it usually skips all the erroneous construct until its end delimited, for instance, by a semicolon and issues a message stating that there are too many errors to analyze this construct. Then, it continues with the syntax analysis right behind this construct.

Exercises

4.1$_{\textit{Solved}}$**.** Consider the next three grammars. Determine the languages they generate. Observe that the languages generated by grammars (a) and (b) represent simple programming languages while the language generated by grammar (c) is rather abstract. Construct the predictive tables corresponding to grammars (a) through (c).

(a)
1: ⟨program⟩ → ⟨statement⟩⟨statement list⟩.
2: ⟨statement list⟩ → ; ⟨statement⟩⟨statement list⟩
3: ⟨statement list⟩ → ε
4: ⟨statement⟩ → *i* = ⟨expression⟩
5: ⟨statement⟩ → **read** *i*
6: ⟨statement⟩ → **write** ⟨expression⟩
7: ⟨statement⟩ → **for** *i* = ⟨expression⟩ **to** ⟨expression⟩ **perform** ⟨statement⟩
8: ⟨statement⟩ → **begin** ⟨statement⟩⟨statement list⟩ **end**
9: ⟨expression⟩ → ⟨operand⟩⟨continuation⟩
10: ⟨continuation⟩ → ⟨operator⟩⟨expression⟩
11: ⟨continuation⟩ → ε
12: ⟨operand⟩ → (⟨expression⟩)
13: ⟨operand⟩ → *i*
14: ⟨operator⟩ → +
15: ⟨operator⟩ → –

(b)
1: ⟨program⟩ → **begin** ⟨statement⟩⟨statement list⟩ **end**
2: ⟨statement list⟩ → ; ⟨statement⟩⟨statement list⟩
3: ⟨statement list⟩ → ε
4: ⟨statement⟩ → *i* = ⟨boolean expression⟩
5: ⟨statement⟩ → **read** *i*
6: ⟨statement⟩ → **write** ⟨boolean expression⟩
7: ⟨statement⟩ → **if** ⟨boolean expression⟩ **then** ⟨statement⟩ **else** ⟨statement⟩
8: ⟨statement⟩ → **while** ⟨boolean expression⟩ **do** ⟨statement⟩
9: ⟨statement⟩ → **repeat** ⟨statement⟩⟨statement list⟩ **until** ⟨boolean expression⟩
10: ⟨statement⟩ → **begin** ⟨statement⟩⟨statement list⟩ **end**
11: ⟨boolean expression⟩ → ⟨operand⟩⟨continuation⟩
12: ⟨continuation⟩ → ⟨operator⟩⟨boolean expression⟩
13: ⟨continuation⟩ → ε
14: ⟨operand⟩ → (⟨boolean expression⟩)
15: ⟨operand⟩ → *i*
16: ⟨operand⟩ → **not** ⟨boolean expression⟩
17: ⟨operator⟩ → **and**
18: ⟨operator⟩ → **or**

(c)
1: $S \to ABb$
2: $A \to CD$
3: $B \to dB$
4: $B \to \varepsilon$
5: $C \to aCb$
6: $C \to \varepsilon$
7: $D \to cDd$
8: $D \to \varepsilon$

4.2$_{\mathit{Solved}}$**.** Prove that grammars (a) through (d), given next, are not LL grammars. Determine their languages. Construct proper LL grammars equivalent to them.

(a)
1: ⟨program⟩ → **begin** ⟨statement list⟩ **end**
2: ⟨statement list⟩ → ⟨statement⟩ ; ⟨statement list⟩
3: ⟨statement list⟩ → ⟨statement⟩
4: ⟨statement⟩ → **read** *i*
5: ⟨statement⟩ → **write** *i*
6: ⟨statement⟩ → *i* = **sum** (⟨item list⟩)
7: ⟨item list⟩ → *i* , ⟨item list⟩
8: ⟨item list⟩ → *i*

(b)
1: ⟨program⟩ → **begin** ⟨statement list⟩ **end**
2: ⟨statement list⟩ → ⟨statement⟩ ; ⟨statement list⟩
3: ⟨statement list⟩ → ⟨statement⟩
4: ⟨statement⟩ → **read** ⟨item list⟩
5: ⟨statement⟩ → **write** ⟨item list⟩
6: ⟨statement⟩ → *i* = ⟨expression⟩
7: ⟨item list⟩ → *i* , ⟨item list⟩
8: ⟨item list⟩ → *i*
9: ⟨expression⟩ → *i* + ⟨expression⟩
10: ⟨expression⟩ → *i* – ⟨expression⟩
11: ⟨expression⟩ → *i*

(c)
1: ⟨program⟩ → ⟨function⟩ ; ⟨function list⟩
2: ⟨function list⟩ → ⟨function⟩
3: ⟨function⟩ → ⟨head⟩⟨body⟩
4: ⟨head⟩ → **function** *i* ( ) : ⟨type⟩
5: ⟨head⟩ → **function** *i* (⟨parameter list⟩) : ⟨type⟩
6: ⟨parameter list⟩ → ⟨parameter⟩ , ⟨parameter list⟩
7: ⟨parameter list⟩ → ⟨parameter⟩
8: ⟨parameter⟩ → *i* : ⟨type⟩
9: ⟨type⟩ → **integer**
10: ⟨type⟩ → **real**
11: ⟨type⟩ → **string**
12: ⟨body⟩ → **begin** ⟨statement list⟩ **end**
13: ⟨statement list⟩ → ⟨statement⟩ ; ⟨statement list⟩
14: ⟨statement list⟩ → ⟨statement⟩
15: ⟨statement⟩ → *s*
16: ⟨statement⟩ → ⟨program⟩

(d)
1: ⟨block⟩ → **begin** ⟨declaration list⟩⟨execution part⟩ **end**
2: ⟨declaration list ⟩ → ⟨declaration list⟩ ; ⟨declaration⟩
3: ⟨declaration list⟩ → ⟨declaration⟩
4: ⟨declaration⟩ → **integer** ⟨variable list⟩
5: ⟨declaration⟩ → **real** ⟨variable list⟩
6: ⟨variable list⟩ → *i* , ⟨variable list⟩
7: ⟨variable list⟩ → *i*
8: ⟨execution part⟩ → **compute** ⟨statement list⟩
9: ⟨statement list⟩ → ⟨statement list⟩ ; ⟨statement⟩

10: $\langle$statement list$\rangle \rightarrow \langle$statement$\rangle$
11: $\langle$statement$\rangle \rightarrow s$
12: $\langle$statement$\rangle \rightarrow \langle$block$\rangle$

4.3*Solved*. Demonstrate that the following grammar generates the expressions in Polish postfix notation with operand i and operators + and *. Prove that it is not an LL grammar. Construct an equivalent LL grammar.

1: $\langle$expression$\rangle \rightarrow \langle$expression$\rangle\langle$expression$\rangle$ +
2: $\langle$expression$\rangle \rightarrow \langle$expression$\rangle\langle$expression$\rangle$ *
3: $\langle$expression$\rangle \rightarrow i$

4.4. Consider Definitions 4.12 and 4.13, given next. Design algorithms that construct $last(x)$ for every $x \in {}_G\Sigma^*$ and $precede(A)$ for every $A \in {}_GN$. Write programs to implement them.

Definition 4.12 ***last.*** Let $G = ({}_G\Sigma, {}_GR)$ be a grammar. For every string $x \in {}_G\Sigma^*$,

$$last(x) = \{a \mid x \Rightarrow^* w, \text{ where either } w \in {}_G\Delta^+ \text{ with } a = symbol(w, |w|) \text{ or } w = \varepsilon = a\},$$

where $symbol(w, |w|)$ denotes the rightmost symbol of w (see Section 1.1).

■

Definition 4.13 ***precede.*** Let $G = ({}_G\Sigma, {}_GR)$ be a grammar. For every $A \in {}_GN$,

$$precede(A) = \{a \in {}_G\Delta \cup \{\blacktriangleright\} \mid aA \in substrings(\{\blacktriangleright\}F(G))\},$$

where $F(G)$ denotes the set of G's sentential forms (see Definition 3.1).

■

4.5*Solved*. Consider Definition 4.5 *Predictive Set*. For a proper grammar, $G = ({}_G\Sigma, {}_GR)$, the definition of its rules' predictive sets can be simplified. Explain how and give this simplified definition.

4.6. Prove that no LL grammar generates $\{a^ib^i \mid i \geq 1\} \cup \{a^jb^{2j} \mid j \geq 1\}$.

4.7. Prove that every LL grammar is unambiguous (see Definition 3.6). Based on this result, demonstrate that no LL grammar can generate any inherently ambiguous language, such as $\{a^ib^jc^k \mid i, j, k \geq 1, \text{ and } i = j \text{ or } j = k\}$.

4.8. Consider the transformation of any grammar G to an equivalent proper grammar H described in the proof of Theorem 3.38. Prove or disprove that if G is an LL grammar, then H, produced by this transformation, is an LL grammar as well.

4.9. Consider grammars ${}_{cond}G$, ${}_{cond}H$, ${}_{expr}G$, and ${}_{expr}H$ (see Case Study 6/35). Prove that none of them represents an LL grammar. Turn them to equivalent LL grammars.

4.10. Generalize LL grammars to *LL*(k) *grammars*, where $k \geq 1$, which underlie parsers that make a k-token lookahead. Reformulate the notions introduced in Section 4.1, such as *first* and *follow* sets, in terms of these generalized grammars. Discuss the advantages and disadvantages of this generalization from a practical point of view.

4.11. By analogy with left factoring (see Section 4.1), discuss right factoring. Modify Algorithm

4.7 *Left-Factored Grammar* so it converts any grammar to an equivalent right-factored grammar.

4.12. Reconsider Case Study 12/35 *Left Factoring* in terms of right factoring. Make use of the solution to Exercise 4.4.

4.13. Write a program to implement predictive recursive-descent parser; including **RETURN-SYMBOL**, **EXPAND**, and **POP**; described in Section 4.2. In addition, write a program to implement Algorithm 4.11 *Predictive Table-Driven Parsing*.

4.14. Complete Case Study 15/35 *Error Recovery in Predictive Parsing*.

4.15. Consider the programs obtained in Exercise 4.13. Extend them so they perform the panic-mode error recovery described in Section 4.2. Alternatively, extend them so they perform the ad-hoc recovery described in Section 4.2.

4.16. Generalize a predictive table for any grammar G so for each $A \in {}_GN$ and each $t \in {}_G\Delta$, the corresponding entry contains the list of all rules r satisfying ***lhs***$(r) = A$ and $t \in$ *predictive-set*(r); if this list is empty, this entry contains ☹. Notice that an entry of this generalized table may contain more than one rule. Give a rigorous version of this generalization. Discuss the advantages and disadvantages of this generalization from a practical point of view.

4.17*Solved*. Consider this five-rule grammar

1: ⟨if-statement⟩ → **if** ⟨condition⟩ **then** ⟨if-statement⟩⟨else-part⟩
2: ⟨if-statement⟩ → *a*
3: ⟨condition⟩ → *c*
4: ⟨else-part⟩ → **else** ⟨if-statement⟩
5: ⟨else-part⟩ → ε

Construct *predictive-set*(r) for each rule $r = 1, \ldots, 5$. Demonstrate that both *predictive-set*(4) and *predictive-set*(5) contain **else**, so this grammar is no LL grammar. Construct its generalized predictive table, whose entry [⟨else-part⟩, **else**] contains 4 and 5 (see Exercise 4.16). Remove 5 from this multiple entry. Demonstrate that after this removal, Algorithm 4.11 *Predictive Table-Driven Parsing* works with the resulting table quite deterministically so it associates an **else** with the most recent unmatched **then**, which is exactly what most real high-level programming languages do.

Solution to Selected Exercises

4.1. We only give the predictive tables of grammars (a) and (c).

(a)

	.	;	*i*	=	**read**	**write**	**for**	**to**	**perform**	**begin**	**end**	(	)	+	–	◀
⟨program⟩	☹	☹	1	☹	1	1	1	☹	☹	1	☹	☹	☹	☹	☹	☹
⟨statement⟩	☹	☹	4	☹	5	6	7	☹	☹	8	☹	☹	☹	☹	☹	☹
⟨statement list⟩	3	2	☹	☹	☹	☹	☹	☹	☹	☹	3	☹	☹	☹	☹	☹
⟨expression⟩	☹	☹	9	☹	☹	☹	☹	☹	☹	☹	☹	9	☹	☹	☹	☹
⟨operand⟩	☹	☹	13	☹	☹	☹	☹	☹	☹	☹	☹	12	☹	☹	☹	☹
⟨continuation⟩	11	11	☹	☹	☹	☹	☹	11	11	☹	11	☹	11	10	10	☹
⟨operator⟩	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	14	15	☹
▶	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☺

(c)

	a	*b*	*c*	*d*	◀
S	1	1	1	1	☹
A	2	2	2	2	☹
B	☹	4	☹	3	☹
C	5	6	6	6	☹
D	☹	8	7	8	☹
▶	☹	☹	☹	☹	☺

4.2. We only give the solution in terms of grammar (a). An equivalent proper LL grammar to this grammar is

1: ⟨program⟩ → **begin** ⟨statement⟩ ⟨statement list⟩
2: ⟨statement list⟩ → ; ⟨statement⟩ ⟨statement list⟩
3: ⟨statement list⟩ → **end**
4: ⟨statement⟩ → **read** *i*
5: ⟨statement⟩ → **write** *i*
6: ⟨statement⟩ → *i* = **sum** (*i* ⟨item list⟩
7: ⟨item list⟩ → , *i* ⟨item list⟩
8: ⟨item list⟩ →)

4.3. An equivalent LL grammar and its predictive table follow next.

1: ⟨expression⟩ → *i* ⟨continuous⟩
2: ⟨continuous⟩ → ⟨expression⟩ ⟨operator⟩ ⟨continuous⟩
3: ⟨continuous⟩ → ε
4: ⟨operator⟩ → +
5: ⟨operator⟩ → *

	+	*	*i*	◀
⟨expression⟩	☹	☹	1	☹
⟨continuous⟩	3	3	2	3
⟨operator⟩	4	5	☹	☹
▶	☹	☹	☹	☺

4.5. No *follow* set is needed.

Definition 4.14 ***Predictive Sets for a Proper Grammar's Rules.*** Let $G = ({}_G\Sigma, {}_GR)$ be a proper grammar. The *predictive set* of each $r \in {}_GR$, *predictive-set*(r), is defined as *predictive-set*(r) = *first*(***rhs***(r)).

■

4.17. We only give the generalized predictive table:

	if	**then**	**else**	*c*	*a*	◀
⟨if-statement⟩	1	☹	☹	☹	2	☹
⟨condition⟩	☹	☹	☹	3	☹	☹
⟨else-part⟩	☹	☹	4, 5	☹	☹	5
▶	☹	☹	☹	☹	☹	☺