5

LL Grammars and Parsers

Chapter Chapter:global:two presents a *recursive-descent parser* for the syntax-analysis phase of a small compiler. Manual construction of such parsers is both time consuming and error prone—especially when applied at the scale of a real programming language. At first glance, the code for a recursive-descent parser may appear to be written *ad hoc*. Fortunately, there *are* principles at work. This chapter discusses these principles and their application in tools that automate the parsing phase of a compiler.

Recursive-descent parsers belong to the more general class of LL parsers, which were introduced in Chapter Chapter:global:four. In this chapter, we discuss LL parsers in greater detail, analyzing the conditions under which such parsers can be reliably and automatically constructed from **context-free grammars** (CFGs). Our analysis builds on the algorithms and grammar-processing concepts presented in Chapter Chapter:global:four. Because of their simplicity, performance, and excellent error diagnostics, recursive-descent parsers have been constructed for most modern programming languages.

In Section 5.2, we identify a subset of CFGs known as the LL(k) grammars. In Sections 5.3 and 5.4, we show how to construct recursive-descent and table-driven LL parsers from LL(1) grammars—an efficient subset of the LL(k) grammars. For grammars that are not LL(1), Section 5.5 considers grammar transformations that can eliminate non-LL(1) properties. Unfortunately, some languages have no LL(k) grammar, as discussed in Section 5.6. Section 5.7 establishes some useful properties of LL grammars and parsers. Parse-table representations are considered in Section 5.8. Because parsers are typically responsible for discovering syntax errors in programs, Section 5.9 considers how an LL(k) parser might respond to syntactically faulty inputs.

5.1 Overview

In this chapter, we study the following two forms of LL parsers.

- **recursive-descent parsers** contain a set of mutually recursive procedures that cooperate to parse a string. The appropriate code for these procedures is determined by the particular LL(k) grammar.
- **table-driven LL parsers** use a generic LL(k) parsing engine and a parse table that directs the activity of the engine. The entries for the parse table are determined by the particular LL(k) grammar.

Fortunately, CFGs with certain properties can be used to generate such parsers automatically. Tools that operate in this fashion are generally called **compiler-compilers** or **parser generators**. They take a grammar description file as input and attempt to produce a parser for the language defined by the grammar. The term "compiler-compiler" applies because the parser generator is itself a compiler—it accepts a high-level expression of a program (the grammar definition file) and generates an executable form of that program (the parser). This approach makes parsing one of the easiest and most reliable phases of compiler construction for the following reasons.

- When the grammar serves as a language's definition, parsers can be automatically constructed to perform syntax analysis in a compiler. The rigor of the automatic construction guarantees that the resulting parser is faithful to the language's syntactic specification.
- When a language is revised, updated, or extended, the associated modifications can be applied to the grammar description to generate a parser for the new language.
- When parser construction is successful through the techniques described in this chapter, the grammar is proved unambiguous. While devising an algorithmic test for grammar ambiguity is impossible, parsing techniques such as LL(k) are of great use to language designers in developing intuition as to why a grammar might be ambiguous.

As discussed in Chapters Chapter:global:two and Chapter:global:four, every string in a grammar's language can be generated by a derivation that begins with the grammar's start symbol. While it is relatively straightforward to use a grammar's productions to generate sample strings in its language, reversing this process does not seem as simple. That is, given an input string, how can we show why the string is or is not in the grammar's language? This is the **parsing problem**, and in this chapter, we consider a parsing technique that is successful with many CFGs. This parsing technique is known by the following names:

- Top-down, because the parser begins with the grammar's start symbol and grows a parse tree from its root to its leaves

- Predictive, because the parser must predict at each step in the derivation which grammar rule is to be applied next
- LL(k), because these techniques scan the input from left-to-right, produce a leftmost derivation, and use k symbols of lookahead
- Recursive-descent, because this kind of parser can be implemented by a collection of mutually recursive procedures

5.2 LL(k) Grammars

Following is a reprise from Chapter Chapter:global:two of the process for constructing a recursive-descent parser from a CFG.

- A **parsing procedure** is associated with each nonterminal A.
- The procedure associated with A is charged with accomplishing one step of a derivation by choosing and applying one of A's productions.
- The parser chooses the appropriate production for A by inspecting the next k tokens (terminal symbols) in the input stream. The **Predict set** for production $A \rightarrow \alpha$ is the set of tokens that trigger application of that production.
- The Predict set for $A \rightarrow \alpha$ is determined primarily by the detail in α—the **right-hand side** (RHS) of the production. Other CFG productions may participate in the computation of a production's Predict set.

Generally, the choice of production can be predicated on the next k tokens of input, for some constant k chosen before the parser is pressed into service. These k tokens are called the **lookahead** of an LL(k) parser. If it is possible to construct an LL(k) parser for a CFG such that the parser recognizes the CFG's language, then the CFG is an **LL(k) grammar**.

An LL(k) parser can peek at the next k tokens to decide which production to apply. However, the *strategy* for choosing productions must be established when the parser is constructed. In this section, we formalize this strategy by defining a function called $\mathsf{Predict}_k(p)$. This function considers the grammar production p and computes the set of length-k token strings that predict the application of rule p. We assume henceforth that we have one token of lookahead ($k = 1$). We leave the generalization as Exercise 18. Thus, for rule p, $\mathsf{Predict}(p)$ is the set of terminal symbols that call for applying rule p.

Consider a parser that is presented with the input string $\alpha \mathsf{a} \beta \in \Sigma^*$. Suppose the parser has constructed the derivation $\mathsf{S} \Rightarrow^*_{\mathrm{lm}} \alpha\, \mathsf{A} \mathcal{Y}_1 \ldots \mathcal{Y}_n$. At this point, α has been matched and A is the leftmost nonterminal in the derived sentential form. Thus *some* production for A must be applied to continue the leftmost derivation. Because the input string contains an a as the next input token, the parse must continue with a production for A that derives a as its first terminal symbol.

```
function Predict(p : A → X1 ... Xm) : Set
    ans ← First(X1 ... Xm)                          1
    if RuleDerivesEmpty(p)                          2
    then
        ans ← ans ∪ Follow(A)                       3
    return (ans)
end
```

Figure 5.1: Computation of Predict sets.

Recalling the notation from Section Subsection:four:gramrep, we must examine the set of productions

$$P = \{\, p \in \text{PRODUCTIONSFOR}(\mathsf{A}) \mid a \in \mathsf{Predict}(p) \,\}.$$

One of the following conditions must be true of the set P.

- P is the empty set. In this case, no production for A can cause the next input token to be matched. The parse cannot continue and a syntax error is issued, with a as the offending token. The productions for A can be helpful in issuing error messages that indicate which terminal symbols could be processed at this point in the parse. Section 5.9 considers error recovery and repair in greater detail.

- P contains more than one production. In this case, the parse could continue, but **nondeterminism** would be required to pursue the independent application of each production in P. For efficiency, we require that our parsers operate deterministically. Thus parser construction must ensure that this case cannot arise.

- P contains exactly one production. In this case, the leftmost parse can proceed deterministically by applying the only production in set P.

Fortunately, Predict sets are based on a grammar and not on any particular input string. We can analyze a grammar to determine whether each terminal predicts (at most) one of A's rules. In such cases, we can construct a deterministic parser and we call the associated grammar LL(1).

We next consider a rule p in greater detail and show how to compute Predict(p). Consider a production $p : \mathsf{A} \to \mathcal{X}_1 \ldots \mathcal{X}_m, m \geq 0$.[1] As shown in Figure 5.1, the set of symbols that predict rule p is drawn from one or both of the following:

- The set of possible terminal symbols that are first produced in some derivation from $\mathcal{X}_1 \ldots \mathcal{X}_m$

- Those terminal symbols that can follow A in some sentential form

[1] Recall that by convention, if $m = 0$, then we have the rule $\mathsf{A} \to \lambda$.

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

Figure 5.2: A CFG.

In the algorithm of Figure 5.1, Step **1** initializes the result to First($\mathcal{X}_1 \ldots \mathcal{X}_m$)—the set of terminal symbols that can appear leftmost in any derivation of $\mathcal{X}_1 \ldots \mathcal{X}_m$. The algorithm for computing this set is given in Figure Figure:four:computefirst. Step **2** detects when $\mathcal{X}_1 \ldots \mathcal{X}_m \Rightarrow^* \lambda$, using the results of the algorithm presented in Figure Figure:four:derivelambda. RuleDerivesEmpty(p) is true if, and only if, production p can derive λ. In this case, Step **3** includes those symbols in Follow(A), as computed by the algorithm in Figure Figure:four:computefollow. Such symbols can *follow* A after A $\Rightarrow^* \lambda$. Thus, the function shown in Figure 5.1 computes the set of length-1 token strings that predict rule p. By convention, λ is *not* a terminal symbol, so it does not participate in any Predict set.

In an LL(1) grammar, the productions for each A must have disjoint predict sets, as computed with one symbol of lookahead. Experience indicates that creating an LL(1) grammar is possible for most programming languages. However, *not all* CFGs are LL(1). For such grammars, the following may apply.

- More lookahead may be needed, in which case the grammar is LL(k) for some constant $k > 1$.
- A more powerful parsing method may be required. Chapter Chapter:global:six describes such methods.
- The grammar may be *ambiguous*. Such grammars cannot be accommodated by *any* deterministic parsing method.

We now apply the algorithm in Figure 5.1 to the grammar shown in Figure 5.2. Figure 5.3 shows the Predict calculation; for each production of the form A $\rightarrow \mathcal{X}_1 \ldots \mathcal{X}_m$, First($\mathcal{X}_1 \ldots \mathcal{X}_m$) is shown. The next column indicates whether $\mathcal{X}_1 \ldots \mathcal{X}_m \Rightarrow^* \lambda$. The rightmost column shows Predict(A $\rightarrow \mathcal{X}_1 \ldots \mathcal{X}_m$)—the set of symbols that predict the production A $\rightarrow \mathcal{X}_1 \ldots \mathcal{X}_m$. This set includes First($\mathcal{X}_1 \ldots \mathcal{X}_m$), as well as Follow(A) if $\mathcal{X}_1 \ldots \mathcal{X}_m \Rightarrow^* \lambda$.

The algorithm shown in Figure 5.4 determines whether a grammar is LL(1), based on the grammar's Predict sets. The Predict sets for each nonterminal A are checked for intersection. If no two rules for A have any symbols in common, then

Rule Number	A	$\mathcal{X}_1 \ldots \mathcal{X}_m$	First($\mathcal{X}_1 \ldots \mathcal{X}_m$)	Derives Empty?	Follow(A)	Answer
1	S	A C $	a,b,q,c,$	No		a,b,q,c,$
2	C	c	c	No		c
3		λ		Yes	d,$	d,$
4	A	a B C d	a	No		a
5		B Q	b,q	Yes	c,$	b,q,c,$
6	B	b B	b	No		b
7		λ		Yes	q,c,d,$	q,c,d,$
8	Q	q	q	No		q
9		λ		Yes	c,$	c,$

Figure 5.3: Predict calculation for the grammar of Figure 5.2.

```
function IsLL1(G) : Boolean
    foreach A ∈ N do
        PredictSet ← ∅
        foreach p ∈ ProductionsFor(A) do
            if Predict(p) ∩ PredictSet ≠ ∅                    4
            then return (false)
            PredictSet ← PredictSet ∪ Predict(p)
    return (true)
end
```

Figure 5.4: Algorithm to determine if a grammar G is LL(1).

the grammar is LL(1). The grammar of Figure 5.2 passes this test, and is therefore LL(1).

5.3 Recursive-Descent LL(1) parsers

We are now prepared to generate the procedures of a recursive-descent parser. The parser's input is a sequence of tokens provided by the stream *ts*. We assume that *ts* offers the following methods.

- PEEK, which examines the next input token without advancing the input.
- ADVANCE, which advances the input by one token

The parsers we construct rely on the MATCH method shown in Figure 5.5. This method checks the token stream *ts* for the presence of a particular token.

To construct a recursive-descent parser for an LL(1) grammar, we write a separate procedure for each nonterminal A. If A has rules $p_1, p_2, \ldots, p_n$, we formulate the procedure shown in Figure 5.6. The code constructed for each p_i is obtained

```
procedure MATCH(ts, token)
    if ts.PEEK( ) = token
    then call ts.ADVANCE( )
    else call ERROR(Expected token)
end
```

Figure 5.5: Utility for matching tokens in an input stream.

```
    procedure A(ts)
        switch ()
            case ts.PEEK( ) ∈ Predict(p1)
                /*                  Code for p1                  */
            case ts.PEEK( ) ∈ Predict(pi)
                /*                  Code for p2                  */
            /*                          .                        */
            /*                          .                        */
            /*                          .                        */
            case ts.PEEK( ) ∈ Predict(pn)
                /*                  Code for pn                  */
            case default
                /*                  Syntax error                 */
    end
```

Figure 5.6: A typical recursive-descent procedure.

by scanning the RHS of rule p_i $(\mathcal{X}_1 \ldots \mathcal{X}_m)$ from left to right. As each symbol is visited, code is inserted into the parsing procedure. For productions of the form $A \rightarrow \lambda$, $m = 0$ so there are no symbols to visit; in such cases, the parsing procedure simply returns immediately.

In considering $\mathcal{X}_i$, there are two possible cases, as follows.

1. $\mathcal{X}_i$ is a terminal symbol. In this case, a call to MATCH($ts, \mathcal{X}_i$) is placed in the parser to insist that $\mathcal{X}_i$ is the next symbol in the token stream. If the token is successfully matched, the token stream is advanced. Otherwise, the input string cannot be in the grammar's language and an error message is issued.

2. $\mathcal{X}_i$ is a nonterminal symbol. In this case, there is a procedure responsible for continuing the parse by choosing an appropriate production for $\mathcal{X}_i$. Thus, a call to $\mathcal{X}_i(ts)$ is placed in the parser.

Figure 5.7 shows the parsing procedures created for the LL(1) grammar shown in Figure 5.2. For presentation purposes, the default case—representing a syntax error—is not shown in the parsing procedures of Figure 5.7.

```
procedure S(ts)
    switch ()
        case ts.PEEK( ) ∈ { a, b, q, c, $ }
            call A( )
            call C( )
            call MATCH(ts, $)
end
procedure C(ts)
    switch ()
        case ts.PEEK( ) ∈ { c }
            call MATCH(ts, c)
        case ts.PEEK( ) ∈ { d, $ }
            return ()
end
procedure A(ts)
    switch ()
        case ts.PEEK( ) ∈ { a }
            call MATCH(ts, a)
            call B( )
            call C( )
            call MATCH(ts, d)
        case ts.PEEK( ) ∈ { b, q, c, $ }
            call B( )
            call Q( )
end
procedure B(ts)
    switch ()
        case ts.PEEK( ) ∈ { b }
            call MATCH(ts, b)
            call B( )
        case ts.PEEK( ) ∈ { q, c, d, $ }
            return ()
end
procedure Q(ts)
    switch ()
        case ts.PEEK( ) ∈ { q }
            call MATCH(ts, q)
        case ts.PEEK( ) ∈ { c, $ }
            return ()
end
```

Figure 5.7: Recursive-Descent Code.

5.4 Table-Driven LL(1) Parsers

The task of creating recursive-descent parsers as presented in Section 5.3 is mechanical and can therefore be automated. However, the size of the parser's code grows with the size of the grammar. Moreover, the overhead of method calls and returns can be a source of inefficiency. In this section we examine how to construct a table-driven LL(1) parser. Actually, the parser itself is standard across all grammars; we need only to provide an adequate parse table.

To make the transition from explicit code to table-driven processing, we use a stack to simulate the actions performed by MATCH and by the calls to the nonterminals' procedures. In addition to the methods typically provided by a stack, we assume that the top-of-stack contents can be obtained nondestructively (without popping the stack) via the method TOS( ).

In code form, the generic LL(1) parser is given in Figure 5.8. At each iteration of the loop at Step **5**, the parser performs one of the following actions.

- If the top-of-stack is a terminal symbol, then MATCH is called. This method, defined in Figure 5.5, ensures that the next token of the input stream matches the top-of-stack symbol. If successful, the call to MATCH advances the token input stream. For the table-driven parser, the matching top-of-stack symbol is popped at Step **9**.

- If the top-of-stack is a nonterminal symbol, then the appropriate production is determined at Step **10**. If a valid production is found, then APPLY is called to replace the top-of-stack symbol with the RHS of production *p*. These symbols are pushed such that the resulting top-of-stack is the first symbol on *p*'s RHS.

The parse is complete when the end-of-input symbol is matched at Step **8**.

Given a CFG that has passed the ISLL1 test in Figure 5.4, we next examine how to build its LL(1) parse table. The rows and columns of the parse table are labeled by the nonterminals and terminals of the CFG, respectively. The table—consulted at Step **10** in Figure 5.8—is indexed by the top-of-stack symbol (TOS( )) and by the next input token (*ts*.PEEK( )).

Each nonblank entry in a row is a production that has the row's nonterminal as its **left-hand side** (LHS) symbol. A production is typically represented by its rule number in the grammar. The table is used as follows.

1. The nonterminal symbol at the top-of-stack determines which row is chosen.

2. The next input token (*i.e.*, the **lookahead**) determines which column is chosen.

The resulting entry indicates which, if any, production of the CFG should be applied at this point in the parse.

For practical purposes, the nonterminals and terminals should be mapped to small integers to facilitate table lookup by (two-dimensional) array indexing. The

```
procedure LLPARSER(ts)
    call PUSH(S)
    accepted ← false
    while not accepted do                                          5
        if TOS( ) ∈ Σ                                              6
        then
            call MATCH(ts, TOS( ))                                 7
            if TOS( ) = $                                          8
            then  accepted ← true
            call POP( )                                            9
        else
            p ← LLtable[TOS( ), ts.PEEK( )]                        10
            if p = 0
            then  call ERROR(Syntax error—no production applicable)
            else  call APPLY(p)
end
procedure APPLY(p : A → X1 ... Xm)
    call POP( )
    for i = m downto 1 do
        call PUSH(Xi)
end
```

Figure 5.8: Generic LL(1) parser.

```
procedure FILLTABLE(LLtable)
    foreach A ∈ N do
        foreach a ∈ Σ do  LLtable[A][a] ← 0
    foreach A ∈ N do
        foreach p ∈ ProductionsFor(A) do
            foreach a ∈ Predict(p) do  LLtable[A][a] ← p
end
```

Figure 5.9: Construction of an LL(1) parse table.

	Lookahead					
Nonterminal	a	b	c	d	q	$
S	1	1	1		1	1
C			2	3		3
A	4	5	5		5	5
B		6	7	7	7	7
Q			9		8	9

Figure 5.10: LL(1) table. Error entries are not shown.

procedure for constructing the parse table is shown in Figure 5.9. Upon the procedure's completion, any entry marked 0 will represent a terminal symbol that does not predict any production for the associated nonterminal. Thus, if a 0 entry is accessed during parsing, the input string contains an error.

Using the grammar shown in Figure 5.2 and its associated Predict sets shown in Figure 5.3, we construct the LL(1) parse table shown in Figure 5.10. The table's contents are the rule numbers for productions as shown in Figure 5.2, with blanks rather than zeros to represent errors.

Finally, using the parse table shown in Figure 5.10, we trace the behavior of an LL(1) parser on the input string a b b d c $ in Figure 5.11.

5.5 Obtaining LL(1) Grammars

It can be difficult for inexperienced compiler writers to create LL(1) grammars. This is because LL(1) requires a unique prediction for each combination of nonterminal and lookahead symbol. It is easy to write productions that violate this requirement.

Fortunately, most LL(1) prediction conflicts can be grouped into two categories: *common prefixes* and *left-recursion*. Simple grammar transformations that eliminate common prefixes and left-recursion are known, and these transformations allow us to obtain LL(1) form for most CFGs.

5.5.1 Common Prefixes

In this category of conflicts, two productions for the same nonterminal share a **common prefix** if the productions' RHSs begin with the same string of grammar symbols. For the grammar shown in Figure 5.12, both Stmt productions are predicted by the if token. Even if we allow greater lookahead, the else that distinguishes the two productions can lie arbitrarily far ahead in the input, because Expr and StmtList can each generate a terminal string larger than any constant k. The grammar in Figure 5.12 is therefore not LL(k) for any k.

Prediction conflicts caused by common prefixes can be remedied by the simple factoring transformation shown in Figure 5.13. At Step **11** in this algorithm, a

Parse Stack	Action	Remaining Input
S		abbdc$
	Apply 1: S→AC$	
$CA		abbdc$
	Apply 4: A→aBCd	
$CdCBa		abbdc$
	Match	
$CdCB		bbdc$
	Apply 6: B→bB	
$CdCBb		bbdc$
	Match	
$CdCB		bdc$
	Apply 6: B→bB	
$CdCBb		bdc$
	Match	
$CdCB		dc$
	Apply 7: B→λ	
$CdC		dc$
	Apply 3: C→λ	
$Cd		dc$
	Match	
$C		c$
	Apply 2: C→c	
$c		c$
	Match	
$		$
	Accept	

Figure 5.11: Trace of an LL(1) parse. The stack is shown in the left column, with top-of-stack as the rightmost character. The input string is shown in the right column, processed from left-to-right.

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

Figure 5.12: A grammar with common prefixes.

```
procedure FACTOR( )
    foreach A ∈ N do
        α ← LongestCommonPrefix(ProductionsFor(A))
        while |α| > 0 do
            V ← NewNonTerminal()
            Productions ← Productions ∪ { A → αV }
            foreach p ∈ ProductionsFor(A) | RHS(p) = αβ_p do          11
                Productions ← Productions − { p }
                Productions ← Productions ∪ { V → β_p }
            α ← LongestCommonPrefix(ProductionsFor(A))
end
```

Figure 5.13: Factoring common prefixes.

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

Figure 5.14: Factored version of the grammar in Figure 5.12.

production is identified whose RHS shares a common prefix α with other productions; the remainder of the RHS is denoted β_p for production p. With the common prefix factored and placed into a new production for A, each production sharing α is stripped of this common prefix. Applying the algorithm in Figure 5.13 to the grammar in Figure 5.12 produces the grammar in Figure 5.14.

5.5.2 Left-Recursion

A production is **left-recursive** if its LHS symbol is also the first symbol of its RHS. In Figure 5.14, the production StmtList→StmtList ; Stmt is left-recursive. We extend this definition to nonterminals, so a nonterminal is left-recursive if it is the LHS symbol of a left-recursive production.

Grammars with left-recursive productions can never be LL(1). To see this, assume that some lookahead symbol t predicts the application of the left-recursive production A→Aβ. With recursive-descent parsing, the application of this production will cause procedure A to be invoked repeatedly, without advancing the input. With the state of the parse unchanged, this behavior will continue indefinitely. Similarly, with table-driven parsing, application of this production will repeatedly push

```
procedure ELIMINATELEFTRECURSION( )
    foreach A ∈ N do
        if ∃ r ∈ ProductionsFor(A) | RHS(r) = Aα
        then
            X ← NewNonTerminal()
            Y ← NewNonTerminal()
            foreach p ∈ ProductionsFor(A) do
                if p = r
                then  Productions ← Productions ∪ { A→ X Y }
                else  Productions ← Productions ∪ { X→RHS(p) }
            Productions ← Productions ∪ { Y→αY, Y→λ }
end
```

Figure 5.15: Eliminating left-recursion.

$A\beta$ on the stack without advancing the input.

The algorithm shown in Figure 5.15 removes left-recursion from a factored grammar. Consider the following left-recursive rules.

$$\begin{array}{lllll} 1 & A & \rightarrow & A & \alpha \\ 2 & & | & \beta & \end{array}$$

Each time Rule 1 is applied, an α is generated. The recursion ends when Rule 2 prepends a β to the string of α symbols. Using the regular-expression notation developed in Chapter Chapter:global:three, the grammar generates $\beta\alpha^\star$. The algorithm in Figure 5.15 obtains a grammar that also generates $\beta\alpha^\star$. However, the β is generated *first*. The α symbols are then generated via right-recursion. Applying this algorithm to the grammar in Figure 5.14 results in the grammar shown in Figure 5.16. Since X appears as the LHS of only one production, X's unique RHS can be automatically substituted for all uses of X. This allows Rules 4 and 5 to be replaced with StmtList→Stmt Y.

The algorithms presented in Figures 5.13 and 5.15 typically succeed in obtaining an LL(1) grammar. However, some grammars require greater thought to obtain an LL(1) version; some of these are included as exercises at the end of this chapter. All grammars that include the $ (end-of-input) symbol can be rewritten into a form in which all right-hand sides begin with a terminal symbol; this form is called **Greibach Normal Form** (GNF) (see Exercise 19). Once a grammar is in GNF, factoring of common prefixes is easy. Surprisingly, even this transformation does not guarantee that a grammar will be LL(1) (see Exercise 20). In fact, as we discuss in the next section, language constructs do exist that have no LL(1) grammar. Fortunately, such constructs are rare in practice and can be handled by modest extensions to the LL(1) parsing technique.

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

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

```
1  S     →  Stmt $
2  Stmt  →  if expr then Stmt else Stmt
3        |  if expr then Stmt
4        |  other
```

Figure 5.17: Grammar for if-then-else.

5.6 A Non-LL(1) Language

Almost all common programming language constructs can be specified by LL(1) grammars. One notable exception, however, is the if-then-else construct present in programming languages such as Pascal and C. The if-then-else language defined in Figure 5.16 has a token that *closes* an if—the endif. For languages that lack this delimiter, the if-then-else construct is subject to the so-called **dangling else** problem: A sequence of nested conditionals can contain more thens than elses, which leaves open the correspondence of thens to elses. Programming languages resolve this issue by mandating that each else is matched to its closest, otherwise unmatched then.

We next show that no LL(k) parser can handle languages that embed the if-then-else construct shown in Figure 5.17. This grammar has common prefixes that can be removed by the algorithm in Figure 5.13, but this grammar has a more serious problem. As demonstrated by Exercises 12 and 15, the grammar in Figure 5.17 is *ambiguous* and is therefore not suitable for LL(k) parsing. An **ambiguous grammar** can produce (at least) two distinct parses for some string in the grammar's language. Ambiguity is considered in greater detail in Chapter Chapter:global:six.

We do not intend to use the grammar of Figure 5.17 for LL(k) parsing. Instead, we study the *language* of this grammar to show that *no* LL(k) grammar exists for this language. In studies of this kind, it is convenient to elide unnecessary detail to expose a language's problematic aspects. In the language defined by the grammar of Figure 5.17, we can, in effect, regard the "if expr then Stmt" portion as an

1	S	$\rightarrow$	[S CL		1	S	$\rightarrow$	[S
2		|	λ		2		|	T
3	CL	$\rightarrow$	]		3	T	$\rightarrow$	[T]
4		|	λ		4		|	λ
	(a)					(b)		

Figure 5.18: Attempts to create an LL(1) grammar for DBL.

opening bracket and the "else Stmt" part as an *optional closing bracket*. Thus, the language of Figure 5.17 is structurally equivalent to the *dangling bracket language*, or DBL, defined by

$$\mathsf{DBL} = \{\, [^i\ ]^j \mid i \geq j \geq 0 \,\}.$$

We next show that DBL is not LL(k) for any k.

We can gain some insight into the problem by considering some grammars for DBL. Our first attempt is the grammar shown in Figure 5.18(a), in which CL generates an optional closing bracket. Superficially, the grammar appears to be LL(1), because it is free of left-recursion and common prefixes. However, the ambiguity present in the grammar of Figure 5.17 is retained this grammar. Any sentential form containing CL CL can generate the terminal] two ways, depending on which CL generates the] and which generates λ. Thus, the string [[ ] has two distinct parses.

To resolve the ambiguity, we create a grammar that follows the Pascal and C rules: Each] is matched with the *nearest unmatched* [. This approach results in the grammar shown in Figure 5.18(b). This grammar generates zero or more unmatched opening brackets followed by zero or more pairs of matching brackets. In fact, this grammar is parsable using most bottom-up techniques (such as SLR(1), which is discussed in Chapter Chapter:global:six). While this grammar is factored and is not left-recursive, it is not LL(1) according to the definitions given previously. The following analysis explains why this grammar is not LL(k) for any k.

$$\begin{aligned}
[\quad &\in \quad \mathsf{Predict}(\mathsf{S} \rightarrow [\ \mathsf{S}) \\
[\quad &\in \quad \mathsf{Predict}(\mathsf{S} \rightarrow \mathsf{T}) \\
[\,[\quad &\in \quad \mathsf{Predict}_2(\mathsf{S} \rightarrow [\ \mathsf{S}) \\
[\,[\quad &\in \quad \mathsf{Predict}_2(\mathsf{S} \rightarrow \mathsf{T}) \\
&\cdots \\
[^k \quad &\in \quad \mathsf{Predict}_k(\mathsf{S} \rightarrow [\ \mathsf{S}) \\
[^k \quad &\in \quad \mathsf{Predict}_k(\mathsf{S} \rightarrow \mathsf{T})
\end{aligned}$$

In particular, seeing only open brackets, LL parsers cannot decide whether to predict a matched or an unmatched open bracket. This is where bottom-up parsers have an

1	S	→	Stmt $
2	Stmt	→	if expr then Stmt V
3		|	other
4	V	→	else Stmt
5		|	λ

	Lookahead					
Nonterminal	if	expr	then	else	other	$
S	1				1	
Stmt	2				3	
V				4,5		5

Figure 5.19: Ambiguous grammar for if-then-else and its LL(1) table. The ambiguity is resolved by favoring Rule 4 over Rule 5 in the boxed entry.

advantage: They can delay applying a production until an entire RHS is matched. Top-down methods cannot delay—they must predict a production based on the first (or first k) symbols derivable from a RHS. To parse languages containing if-then-else constructs, the ability to postpone segments of the parse is crucial.

Our analysis thus concludes that LL(1) parser generators cannot automatically create LL(1) parsers from a grammar that embeds the if-then-else construct. This shortcoming is commonly handled by providing an ambiguous grammar along with some special-case rules for resolving any nonunique predictions that arise.

Factoring the grammar in Figure 5.17 yields the ambiguous grammar and (correspondingly nondeterministic) parse table shown in Figure 5.19. As expected, the else symbol predicts multiple productions—Rules 4 and 5. Since the else should match the closest then, we resolve the conflict in favor of Rule 4. Favoring Rule 5 would defer consumption of the else. Moreover, the parse table entry for nonterminal V and terminal else is Rule 4's only legitimate chance to appear in the parse table. If this rule is absent from the parse table, then the resulting LL(1) parser could never match *any* else. We therefore insist that rule V→ else Stmt be predicted for V when the lookahead is else. The parse table or recursive-descent code can be modified manually to achieve this effect. Some parser generators offer mechanisms for establishing priorities when conflicts arise.

5.7 Properties of LL(1) Parsers

We can establish the following useful properties for LL(1) parsers.

- A correct, leftmost parse is constructed.

  This follows from the fact that LL(1) parsers simulate a leftmost derivation. Moreover, the algorithm in Figure 5.4 finds a CFG to be LL(1) only if the

Predict sets of a nonterminal's productions are disjoint. Thus, the LL(1) parser traces the unique, leftmost derivation of an accepted string.

- All grammars in the LL(1) class are unambiguous.

  If a grammar is ambiguous, then some string has two or more distinct leftmost derivations. If we compare two such derivations, then there must be a nonterminal A for which at least two different productions could be applied to obtain the different derivations. In other words, with a lookahead token of x, a derivation could continue by applying $A \to \alpha$ or $A \to \beta$. It follows that $x \in \text{Predict}(A \to \alpha)$ and $x \in \text{Predict}(A \to \beta)$. Thus, the test at Step **4** in Figure 5.4 determines that such a grammar is not LL(1).

- All table-driven LL(1) parsers operate in linear time and space with respect to the length of the parsed input. (Exercise 16 examines whether recursive-descent parsers are equally efficient.)

  Consider the number of actions that can be taken by an LL(1) parser when the token x is presented as lookahead. Some number of productions will be applied before x either is matched or is found to be in error.

  - Suppose a grammar is λ-free. In this case, no production can be applied twice without advancing the input. Otherwise, the cycle involving the same production will continue to be applied indefinitely; this condition should have been reported as an error when the LL(1) parser was constructed.
  - If the grammar does include λ, then the number of nonterminals that could pop from the stack because of the application of λ-rules is proportional to the length of the input. Exercise 17 explores this point in more detail.

  Thus each input token induces a bounded number of parser actions. It follows that the parser operates in linear time.

  The LL(1) parser consumes space for the lookahead buffer—of constant size—and for the parse stack. The stack grows and contracts during parsing. However, the maximum stack used during any parse is proportional to the length of the parsed input, for either of the following reasons.

  - The stack grows only when a production is applied of the form $A \to \alpha$. As argued previously, no production could be applied twice without advancing the input and, correspondingly, decreasing the stack size. If we regard the number and size of a grammar's productions to be bounded by some constant, then each input token contributes to a constant increase in stack size.
  - If the parser's stack grew superlinearly, then the the parser would require more than linear time just to push entries on the stack.

	Column				
Row	1	2	3	4	5
1	L			P	
2		Q			R
3			U		
4	W	X			
5		Y		Z	

Figure 5.20: A sparse table.

5.8 Parse-Table Representation

Most entries in an LL(1) parse table are zero, indicating an error in the parser's input. LL(1) parse tables tend to be sparsely populated because the Predict sets for most productions are small relative to the size of the grammar's terminal vocabulary. For example, an LL(1) parser was constructed for a subset of Ada, using a grammar that contained 70 terminals and 138 nonterminals. Of the 9660 potential LL(1) parse-table entries, only 629 (6.5%) allowed the parse to continue.

Given such statistics, it makes sense to view the blank entries as a **default**; we then strive to represent the **nondefault** entries efficiently. Generally, consider a two-dimensional parse table with N rows, M columns, and E nondefault entries. The parse table constructed in Section 5.4 occupies space proportional to $N \times M$. Especially when $E \ll N \times M$, our goal is to represent the parse table using space proportional to E. Although modern workstations are equipped with ample storage to handle LL(1) tables for any practical LL(1) grammar, most computers operate more efficiently when storage accesses exhibit greater locality. A smaller parse table loads faster and makes better use of high-speed storage. Thus, it is worthwhile to consider sparse representations for LL(1) parse tables. However, any increase in space efficiency must not adversely affect the efficiency of accessing the parse table. In this section, we consider strategies for decreasing the size of a parse table T, such as the table shown in Figure 5.20.

5.8.1 Compaction

We begin by considering **compaction** methods that convert a table T into a representation devoid of default entries. Such methods operate as follows.

1. The nondefault entries of T are stored in compacted form.
2. A mapping is provided from the index pair (i, j) to the set $E \cup \{ \textit{default} \}$.
3. The LL(1) parser is modified. Wherever the parser accesses $T[i, j]$, the mapping is applied to (i, j) and the compacted form supplies the contents of $T[i, j]$.

Compact Entry Index	Compact Entry Contents	Table T Row	Table T Column
0	L	1	1
1	P	1	4
2	Q	2	2
3	R	2	5
4	U	3	3
5	W	4	1
6	X	4	2
7	Y	5	2
8	Z	5	4

Binary Search
(a)

Compact Entry Index	Compact Entry Contents	Table T Row	Table T Column	Hashes to
0	R	2	5	$10 \equiv 0$
1	L	1	1	1
2	Y	5	2	$10 \equiv 0$
3	Z	5	4	$20 \equiv 0$
4	P	1	4	4
5	Q	2	2	4
6	W	4	1	4
7				
8	X	4	2	8
9	U	3	3	9

Hash
(b)

Figure 5.21: Compact versions of the table in Figure 5.20. Only the boxed information is stored in a compact table.

Binary Search

The compacted form can be achieved by listing the nondefault entries in order of their appearance in T, scanning from left-to-right, top-to-bottom. The resulting compact table is shown in Figure 5.21(a). If row r of the compact table contains the nondefault entry $T[i, j]$, then row r contains also i and j, which are necessary for key comparison when the table is searched. We save space if $3 \times E < N \times M$, assuming each table entry takes one unit of storage. Because the data is sorted by row and column, the compact table can be accessed by **binary search**. Given E nondefault entries, each access takes $O(\log(E))$ time.

Hash Table

The compact table shown in Figure 5.21(b) uses $|E| + 1$ slots and stores $T[i, j]$ at a location determined by **hashing** i and j, using the hash function

$$h(i, j) = (i \times j) \bmod (|E| + 1).$$

To create the compact table, we process the nondefault entries of T in any order. The nondefault entry at $T[i, j]$ is stored in the compact table at $h(i, j)$ if that position is unoccupied. Otherwise, we search forward in the table, storing $T[i, j]$ at the next available slot. This method of handling collisions in the compact table is called **linear resolution**. Because the compact table contains $|E| + 1$ slots, one slot is always free after all nondefault entries are hashed. The vacant slot avoids an infinite loop when searching the compact table for a default entry.

Hash performance can be improved by allocating more slots in the compact table and by choosing a hash function that results in fewer collisions. Because the

nondefault entries of T are known in advance, both goals can be achieved by using **perfect hashing** [?]. With this technique, each nondefault entry $T[i, j]$ maps to one of $|E|$ slots using the key (i, j). A nondefault entry is detected when the perfect hash function returns a value greater than $|E|$.

5.8.2 Compression

Compaction reduces the storage requirements of a parse table by eliminating default entries. However, the indices of a nondefault entry must be stored in the compact table to facilitate nondefault entry lookup. As shown in Figure 5.21, a given row or column index can be repeated multiple times. We next examine a **compression** method that tries to eliminate such redundancy and take advantage of default entries.

The compression algorithm we study is called **double-offset indexing**. The algorithm, shown in Figure 5.22, operates as follows.

1. The algorithm initializes a vector V at Step **12**. Although the vector could hold $N \times M$ entries, the final size of the vector is expected to be closer to $|E|$. The entries of V are initialized to the parse table's default value.

2. Step **13** considers the rows of T in an arbitrary order.

3. When row i is considered, a shift value for the row is computed by the FINDSHIFT method. The shift value, retained in $R[i]$, records the amount by which an index into row i is shifted to find its entry in vector V. Method FITS checks that, when shifted, row i fits into V without any collision with the nondefault entries already established in V.

4. The size of V is reduced at Step **14** by removing all default values at V's high end.

To use the compressed tables, entry $T[i, j]$ is found by inspecting V at location $l = R[i] + j$. If the row recorded at $V.fromrow[l]$ is i, then the table entry at $V.entry[l]$ is the nondefault table entry from $T[i, j]$. Otherwise, $T[i, j]$ has the default value.

We illustrate the effectiveness of the algorithm in Figure 5.22 by applying it to the sparse table shown in Figure 5.20. Suppose the rows are considered in order $1, 2, 3, 4, 5$. The resulting structures, shown in Figure 5.23, can be explained as follows.

1. This row cannot be negatively shifted because it has an entry in column 1. Thus, $R[1]$ is 0 and $V[1 \ldots 5]$ represents row 1, with nondefault entries at index 1 and 4.

2. This row can merge into V without shifting, because its nondefault values (columns 2 and 5) can be accommodated at 2 and 5, respectively.

3. Simiarly, row 3 can be accommodated by V without any shifting.

```
procedure COMPRESS( )
    for i = 1 to N × M do                                              12
        V.entry[i] ← default
    foreach row ∈ { 1, 2, ..., N } do                                  13
        R[row] ← FINDSHIFT(row)
        for j = 1 to M do
            if T[row, j] ≠ default
            then
                place ← R[row] + j
                V.entry[place] ← T[row, j]
                V.fromrow[place] ← row
    call TRUNC( V )                                                    14
end
function FINDSHIFT(row) : Integer
    return ( min_{shift=−M+1}^{N×M−M} FITS(row, shift) )
end
function FITS(row, shift) : Boolean
    for j = 1 to M do
        if T[row, j] ≠ default and not ROOMINV(shift + j)              15
        then return (false)
    return (true)
end
function ROOMINV(where) : Boolean
    if where ≥ 1
    then
        if V.entry[where] = default
        then return (true)
    return (false)
end
procedure TRUNC( V )
    for i = N × M downto 1 do
        if V.entry[i] ≠ default
        then
            /⋆                  Retain V[1 . . . i]                  ⋆/
            return ()
end
```

Figure 5.22: Compression algorithm.

R

Row *i*	Shift *R*[*i*]
1	0
2	0
3	0
4	5
5	6

V

Index	Entry	From Row
1	L	1
2	Q	2
3	U	3
4	P	1
5	R	2
6	W	4
7	X	4
8	Y	5
9		
10	Z	5

Figure 5.23: Compression of the table in Figure 5.20. Only the boxed information is actually stored in the compressed structures.

4. When this row is considered, the first slot of V that can accommodate its leftmost column is slot 6. Thus, $R[4] = 5$ and row 4's nondefault entries are placed at 6 and 7.

5. Finally, columns 2 and 4 of row 5 can be accommodated at 8 and 10, respectively. Thus, $R[5] = 6$.

As suggested by the pseudocode at Step **13**, rows can be presented to FINDSHIFT in any order. However, the size of the resulting compressed table can depend on the order in which rows are considered. Exercises 22 and 23 explore this point further. In general, finding a row ordering that achieves maximum compression is an **NP-complete** problem. This means that the best-known algorithms for obtaining optimal compression would have to try all row permutations. However, compression heuristics work well in practice. When compression is applied to the Ada LL(1) parse table mentioned previously, the number of entries drops from 9660 to 660. This result is only 0.3% from the 629 nondefault entries in the original table.

5.9 Syntactic Error Recovery and Repair

A compiler should produce a useful set of diagnostic messages when presented with a faulty input. Thus, when a single error is detected, it is usually desirable to continue processing the input to detect additional errors. Generally, parsers can continue syntax analysis using one of the following approaches.

- With *error recovery*, the parser attempts to ignore the current error. The parser enters a configuration where it is able to continue processing the input.

- *Error repair* is more ambitious. The parser attempts to correct the syntactically faulty program by modifying the input to obtain an acceptable parse.

In this section, we explore each of these approaches in turn. We then examine error detection and recovery for LL(1) parsers.

5.9.1 Error Recover

With **error recovery**, we try to reset the parser so that the remaining input can be parsed. This process may involve modifying the parse stack and remaining input. Depending on the success of the recovery process, subsequent syntax analysis may be accurate. Unfortunately, it is more often the case that faulty error recovery causes errors to **cascade** throughout the remaining parse. For example, consider the C fragment `a=func c+d)`. If error recovery continues the parse by predicting a Statement after the `func`, then another syntax error is found at the parenthesis. A single syntax error has been amplified by error recovery by issuing two error messages.

The primary measure of quality in an error-recovery process is how few false or cascaded errors it induces. Normally, semantic analysis and code generation are disabled upon error recovery because there is no intention to execute the code of a syntactically faulty program.

A simple form of error recovery is often called **panic mode**. In this approach, the parser skips input tokens until it finds a frequently occurring delimiter (*e.g.*, a semicolon). The parser then continues by expecting those nonterminals that derive strings that can follow the delimiter.

5.9.2 Error Repair

With **error repair**, the parse attempts to repair the syntactically faulty program by modifying the parsed or (more commonly) the unparsed portion of the program. The compiler does not presume to know or to suggest an appropriate revision of the faulty program. The purpose of error repair is to analyze the offending input more carefully so that better diagnostics can be issued.

Error-recovery and error-repair algorithms can exploit the fact that LL(1) parsers have the **correct-prefix** property: For each state entered by such parsers, there is a string of tokens that could result in a successful parse. Consider the input string $\alpha\ \mathsf{x}\ \beta$, where token x causes an LL(1) parser to detect a syntax error. The correct-prefix property means that there is at least one string $\alpha\ \gamma \neq \alpha\ \mathsf{x}\ \beta$ that can be accepted by the parser.

What can a parser do to repair the faulty input? The following options are possible:

- Modification of α
- Insertion of text δ to obtain $\alpha\ \delta\ \mathsf{x}\ \beta$

1	S	→	V
2		|	W
3	V	→	v A b
4	W	→	w A c
5	A	→	λ

Figure 5.24: An LL(1) grammar.

- Deletion of x to obtain $\alpha\ \beta$

These options are not equally attractive. The correct-prefix property implies that α is at least a portion of a syntactically correct program. Thus, most error recovery methods do not modify α except in special situations. One notable case is **scope repair**, where nesting brackets may be inserted or deleted to match the corresponding brackets in x β.

Insertion of text must also be done carefully. In particular, error repair based on insertion must ensure that the repaired string will not continually grow so that parsing can never be completed. Some languages are **insert correctable**. For such languages, it is always possible to repair syntactic faults by insertion. Deletion is a drastic alternative to insertion, but it does have the advantage of making progress through the input.

5.9.3 Error Detection in LL(1) Parsers

The recursive-descent and table-driven LL(1) parsers constructed in this chapter are based on Predict sets. These sets are, in turn, based on First and Follow information that is computed globally for a grammar. In particular, recall that the production A→λ is predicted by the symbols in Follow(A).

Suppose that A occurs in the productions V→v A b and W→w A c, as shown in Figure 5.24. For this grammar, the production A→λ is predicted by the symbols in Follow(A) = { b, c }. Examining the grammar in greater detail, we see that the application of A→λ should be followed only by b if the derivation stems from V. However, if the derivation stems from W, then A should be followed only by c. As described in this chapter, LL(1) parsing cannot distinguish between contexts calling for application of A→λ. If the next input token is b or c, the production A→λ is applied, even though the next input token may not be acceptable. If the wrong symbol is present, the error is detected later, when matching the symbol after A in V→v A b or W→w A c. Exercise 25 considers how such errors can be caught sooner by **full LL(1) parsers**, which are more powerful than the **strong LL(1) parsers** defined in this chapter.

```
1   S   →   [ E ]
2       |   ( E )
3   E   →   a
```

```
procedure S(ts, termset)
    switch ()
        case ts.PEEK( ) ∈ { [ }
            call MATCH( [ )
            call E(ts, termset ∪ { ] } )                    16
            call MATCH( ] )
        case ts.PEEK( ) ∈ { ( }
            call MATCH( ( )
            call E(ts, termset ∪ { ) } )                    17
            call MATCH( ) )
end
procedure E(ts, termset)
    if ts.PEEK( ) = a
    then call MATCH(ts, a)
    else
        call ERROR(Expected an a)
        while ts.PEEK( ) ∉ termset do call ts.ADVANCE( )
end
```

Figure 5.25: A grammar and its Wirth-style, error-recovering parser.

5.9.4 Error Recovery in LL(1) Parsers

The LR(1) parsers described in Chapter Chapter:global:six are formally more powerful than the LL(1) parsers. However, the continued popularity of LL(1) parsers can be attributed, in part, to their superior error diagnosis and error recovery. Because of the predictive nature of an LL(1), leftmost parse, the parser can easily extend the parsed portion of a faulty program into a syntactically valid program. When an error is detected, the parser can produce messages informing the programmer of what tokens were *expected* so that the parse could have continued.

A simple and uniform approach to error recovery in LL(1) parsers is discussed by Wirth [Wir76]. When applied to recursive-descent parsers, the parsing procedures described in Section 5.3 are augmented with an extra parameter that receives a set of terminal symbols. Consider the parsing procedure A(*ts*, *termset*) associated with some nonterminal A. When A is called during operation of the recursive-descent parser, any symbol passed via *termset* can legitimately serve as the lookahead symbol when *this* instance of A returns. For example, consider the

grammar and Wirth-style parsing procedures shown in Figure 5.25. Error recovery is placed in E so that if an a is not found, the input is advanced until a symbol is found that *can* follow E. The set of symbols passed to E includes those symbols passed to S as well as a closing bracket (if called from Step **16**) or a closing parenthesis (if called from Step **17**). If E detects an error, the input is advanced until a symbol in *termset* is found. Because end-of-input can follow S, every *termset* includes $. In the worst case, the input program is advanced until $, at which point all pending parsing procedures can exit.

Summary

This concludes our study of LL parsers. Given an LL(1) grammar, it is easy to construct recursive-descent or table-driven LL(1) parsers. Grammars that are not LL(1) can often be converted to LL(1) form by eliminating left-recursion and by factoring common prefixes. Some programming language constructs are inherently non-LL(1). Intervention by the compiler writer can often resolve the conflicts that arise in such cases. Alternatively, more powerful parsing methods can be considered, such as those presented in Chapter Chapter:global:six.

Exercises

1. Show why the following grammar is or is not LL(1).

1	S	→	A B c
2	A	→	a
3		|	λ
4	B	→	b
5		|	λ

2. Show why the following grammar is or is not LL(1).

1	S	→	A b
2	A	→	a
3		|	B
4		|	λ
5	B	→	b
6		|	λ

3. Show why the following grammar is or is not LL(1).

1	S	→	A B B A
2	A	→	a
3		|	λ
4	B	→	b
5		|	λ

4. Show why the following grammar is or is not LL(1).

1	S	→	a S e
2		|	B
3	B	→	b B e
4		|	C
5	C	→	c C e
6		|	d

5. Construct the LL(1) parse table for the following grammar.

1	Expr	→	- Expr
2		|	(Expr)
3		|	Var ExprTail
4	ExprTail	→	- Expr
5		|	λ
6	Var	→	id VarTail
7	VarTail	→	(Expr)
8		|	λ

6. Trace the operation of an LL(1) parser for the grammar of Exercise 5 on the following input.

```
id - -id ( ( id ) )
```

7. Transform the following grammar into LL(1) form, using the techniques presented in Section 5.5.

```
1   DeclList       →   DeclList ; Decl
2                  |   Decl
3   Decl           →   IdList : Type
4   IdList         →   IdList , id
5                  |   id
6   Type           →   ScalarType
7                  |   array ( ScalarTypeList ) of Type
8   ScalarType     →   id
9                  |   Bound .. Bound
10  Bound          →   Sign intconstant
11                 |   id
12  Sign           →   +
13                 |   -
14                 |   λ
15  ScalarTypelist →   ScalarTypeList , ScalarType
16                 |   ScalarType
```

8. Run your solution to Exercise 7 through any LL(1) parser generator to verify that it is actually LL(1). How do you know that your solution generates the same language as the original grammar?

9. Show that every regular language can be defined by an LL(1) grammar.

10. A grammar is said to have *cycles* if it contains a nonterminal A such that $A \Rightarrow^+ A$. Show that an LL(1) grammar must not have cycles.

11. Construct an LL(2) parser for the following grammar.

```
1   Stmt   →   id ;
2              |   id ( IdList ) ;
3   IdList →   id
4              |   id , IdList
```

12. Show the two distinct parse trees that can be constructed for

```
if expr then if expr then other else other
```

using the grammar given in Figure 5.17. For each parse tree, explain the correspondence of then and else.

13. In Section 5.7, it is established that LL(1) parsers operate in linear time. That is, when parsing an input, the parser requires *on average* only a constant-bounded amount of time per input token.

    Is it ever the case that an LL(1) parser requires more than a constant-bounded amount of time to accept some particular symbol? In other words, can we bound by a constant the time interval between successive calls to the scanner to obtain the next token?

14. Design an algorithm that reads an LL(1) parse table and produces the corresponding recursive-descent parser.

15. An **ambiguous grammar** can produce two distinct parses for some string in the grammar's language. Explain why an ambiguous grammar is never LL(k) for any k, even if the grammar is free of common prefixes and left-recursion.

16. Section 5.7 argues that table-driven LL(1) parsers operate in linear time and space. Explain why this claim does or does not hold for recursive-descent LL(1) parsers.

17. Explain why the number of nonterminals that can pop from an LL(1) parse stack is not bounded by a grammar-specific constant.

18. Design an algorithm that computes $\mathsf{Predict}_k$ sets for a CFG.

19. As discussed in Section 5.5, a grammar is in **Greibach Normal Form** (GNF) if all productions are of the form $\mathsf{A} \rightarrow \mathsf{a}\alpha$, where a is a terminal symbol and α is a string of zero or more grammar (*i.e.*, terminal or nonterminal) symbols.

    Let G be a grammar that does not generate λ. Design an algorithm to transform G into GNF.

20. If we construct a GNF version of a grammar using the algorithm developed in Exercise 19, the resulting grammar is free of left-recursion. However, the resulting grammar can still have common prefixes that prevent it from being LL(1). If we apply the algorithm presented in Figure 5.13 of Section 5.5.1, the resulting grammar will be free of left-recursion and common prefixes. Show that the absence of common prefixes and left-recursion in an unambiguous grammar does not necessarily make a grammar LL(1).

21. Section 5.7 and Exercises 16 and 17 examine the efficiency of LL(1) parsers.

    (a) Analyze the efficiency of *operating* a table-driven LL(k) parser, assuming an LL(k) table has already been constructed. Your answer should be formulated in terms of the length of the parsed input.

    (b) Analyze the efficiency of *constructing* an LL(k) parse table. Your answer should be formulated in terms of the size of the grammar—its vocabularies and productions.

    (c) Analyze the efficiency of operating a recursive-descent LL(k) parser.

22. Apply the table compression algorithm in Figure 5.22 to the table shown in Figure 5.20, presenting rows in the order $1, 5, 2, 4, 3$. Compare the success of compression with the result presented in Figure 5.23.

23. Although table-compression is an NP-complete problem, explain why the following heuristic works well in practice.

    > Rows are considered in order of decreasing density of nondefault entries. (That is, rows with the greatest number of nondefault entries are considered first.)

    Apply this heuristic to the table shown in Figure 5.20 and describe the results.

24. A sparse array can be represented as a vector of rows, with each row represented as a *list* of nondefault column entries. Thus, the nondefault entry at $T[i, j]$ would appear as an element of list $R[i]$. The element would contain both its column identification (j) and the nondefault entry ($T[i, j]$).

    (a) Express the table shown in Figure 5.20 using this format.

    (b) Compare the effectiveness of this representation with those given in Section 5.8. Consider both the savings in space and any increase or decrease in access time.

25. Section 5.9.3 contains an example where the production $\mathsf{A} \rightarrow \lambda$ is applied using an invalid lookahead token. With Follow sets computed globally for a given grammar, the style of LL(1) parsing described in this chapter is known as **strong LL**(1). A **full LL**(1) parser applies a production only if the next input token is valid. Given an algorithm for constructing full LL(1) parse tables.

    Hint: If a grammar contains n occurrences of the nonterminal A, then consider *splitting* this nonterminal so that each occurrence is a unique symbol. Thus, A is split into $\mathsf{A}_1, \mathsf{A}_2, \ldots, \mathsf{A}_n$. Each new nonterminal has productions similar to A, but the context of each nonterminal can differ.

26. Consider the following grammar:

1	S	$\rightarrow$	V
2		|	W
3	V	$\rightarrow$	v A b
4	W	$\rightarrow$	w A c
5	A	$\rightarrow$	λ

    Is this grammar LL(1)? Is the grammar *full* LL(1), as defined in Exercise 25?

27. Section 5.9.4 describes an error recovery method that relies on dynamically constructed sets of Follow symbols. Compare these sets with the Follow information computed for full LL(1) in Exercise 25.

Bibliography

[Cic86] R.J. Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23:17–19, 1986.

[Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.