4. Identify the situations that would produce uncertainties in the execution of a pushdown automaton containing the transitions $(p, \lambda, y; q, z)$ and $(p, x, \lambda; r, w)$.

2.4 *LL*(*k*) PARSERS

It is time now to consider how parsing routines can be developed from pushdown automata. Traditionally this problem is encountered when a language is first described in terms of grammatical rewrite rules. Then, parsing routines are developed for the language using the theory of pushdown automata as a design tool. This is the context in which we frame our discussion.

The *LL* Parsing Process

One technique for translating context-free grammars into pushdown automata is to follow the process described in the proof of Theorem 2.2. This construction produces a pushdown automaton that analyzes its input string by first marking the bottom of the stack and pushing the grammar's start symbol on the stack. Then, it repeatedly executes the following three steps as applicable.

1. If the top of the stack contains a nonterminal from the grammar, replace that nonterminal according to one of the grammar's rewrite rules.
2. If the top of the stack contains a terminal, remove that terminal from the stack while reading the same terminal from the input. If the symbol on the input does not match the symbol on the stack, the input is declared to be an illegal string.
3. If the bottom-of-stack marker surfaces on the stack, remove it and declare the portion of the input string processed so far to be acceptable.

Recall that this process parses the input string by producing a leftmost derivation while reading the string from left to right. Consequently, a program segment obtained by translating the automaton directly into program statements will proceed in the same fashion. Parsers developed in this fashion are known as **LL parsers.** The first *L* denotes that the parser reads its input from Left to right; the second *L* denotes that the goal of the parser is to produce a Leftmost derivation.

Figure 2.25b shows a transition diagram constructed from the grammar in Figure 2.25a using the process presented in the proof of Theorem 2.2. (This is the context-free grammar we met in Figure 2.7 that generates strings of the form $x^n y^n$ for nonnegative integers n.) To produce a parsing routine from this grammar, we might convert the transitions of the machine directly into program statements to obtain the routine in Figure 2.26, where we have used the traditional while structure to simulate the activities available to the machine when in state q. (While the symbol on top of the stack is not the bottom-of-stack marker, the machine remains in state q.)

a. $S \to xSy$
$S \to \lambda$

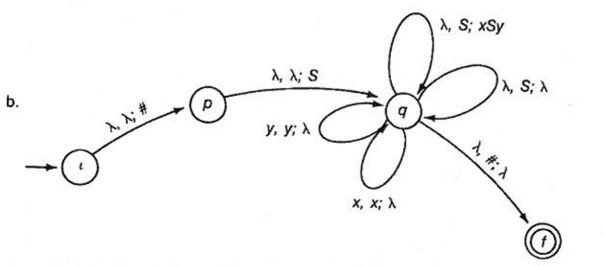
b.


Figure 2.25 A context-free grammar and an associated transition diagram for a pushdown automaton

```
State := ι;
push (#);
State := p;
push (S);
State := q;
while top-of-stack ≠ # do
    case top-of-stack do
        S: either pop (S) and push (xSy)
            or pop (S);
        x: pop (x), read an x from the input;
        y: pop (y), read a y from the input;
    end case
end while;
pop (#);
State := f
```

Figure 2.26 A "program" segment obtained by translating the diagram in Figure 2.25 into statements

Obviously, the segment in Figure 2.26 is not a finished product. For one thing, we have not allowed for errors caused by invalid inputs. For instance, if an x surfaces on the stack within the while loop, our routine assumes that the next symbol on the input is an x and executes the statement "read an x from the input string." In reality the next symbol may not be an x, so our routine should account for this possibility. Hence, we should expand the instruction "read an x from the input string" into the following pair of instructions:

```
read(symbol);
if symbol not x then exit to error routine;
```

Another minor problem with the routine in Figure 2.26 is that it may arrive at state f with an empty stack without having read the entire input string. For example, the string xyx is not in the language described by the original grammar, but our routine will never realize this. Instead, it will read only as far into the input as xy, where it will stop with the assumption that its input string was valid. This problem can be corrected by adding the statements

```
read(symbol);
if symbol not end-of-string marker then exit to error routine;
```

to the end of the routine.

There is, however, one problem in our routine that is more severe than the preceding ones: In some cases the directions present unresolved options. Indeed, if the current state is q and the symbol on top of the stack is S, the routine provides the choice of either replacing that S with xSy or merely removing the S from the stack. This problem is fundamentally different from the issues just discussed, in that it involves the selection of instructions rather than the mere clarification or refinement of an instruction's details.

Applying the Lookahead Principle

Fortunately, the nondeterminism in our routine can be resolved by employing the lookahead principle introduced in the previous section. If we find an x by peeking at the next symbol in the input, then we should replace S with the string xSy; otherwise we should replace it with the empty string. (Pushing xSy on the stack knowing that the next symbol in the input string is not an x would be admitting defeat. Once we push a terminal symbol on the stack, we must be able to match that symbol with an input symbol before it can be removed from the stack. If we pushed xSy on the stack while facing a symbol other than x on the input, the input symbol would not match the terminal x on top of the stack, and we would never be able to empty the stack and move to the accept state.)


```
State := ι;
push (#);
State := p;
push (S);
State := q;
read (Symbol);
while top-of-stack ≠ # do
    case top-of-stack of
        S: if Symbol ≠ x then pop (S)
               else pop (S), push (xSy);
        x: if Symbol not x then exit to error routine
               else pop (x), read (Symbol);
        y: if Symbol not y then exit to error routine
               else pop (y), read (Symbol);
    end case
end while;
pop (#)
if Symbol not end-of-string marker then exit to error routine;
State := f
```

Figure 2.27 A parsing routine based on the grammar of Figure 2.25

Following this lead, we can convert the nondeterministic diagram in Figure 2.25 into the deterministic program segment shown in Figure 2.27. Here we have used the variable symbol as a buffer in which to store the next symbol in the input. From this buffer the symbol can be interrogated when necessary to make decisions, but not processed until its time has come. In particular, note that the end-of-string marker, although detected, is not consumed by the routine. It is left in the buffer where it can be used as the first symbol in the next structure to be analyzed by the parsing system.

The problem encountered in the preceding example is a common phenomenon in *LL* parsers because it originated when the grammar proposed more than one way of rewriting the same nonterminal. Such multiple options are essential to grammars that must generate languages containing more than a single string. (A context-free grammar that provides only one way of rewriting each nonterminal is capable of generating only one string.) Thus, the underlying activity of *LL* parsers is that of predicting which of several rewrite rules should be used to process the remaining input symbols. Consequently, these parsers are called **predictive parsers.**

Many of the uncertainties faced by predictive parsers can be resolved by applying the lookahead principle. However, even in cases where the lookahead principle is the right technique its application may not be as straightforward as in our example. If we were to build a parser from the grammar in Figure 2.28, we would find that the decision regarding the

$S \rightarrow xSz$
$S \rightarrow xyTyz$
$T \rightarrow \lambda$

Figure 2.28 A context-free grammar that requires an *LL*(2) parser

rewriting of *S* cannot be resolved merely by peeking at the next input symbol. (Knowing that the next symbol is x does not tell us to apply $S \rightarrow xSy$ as opposed to $S \rightarrow xyTyz$.) Rather, the decision depends on the next two symbols. Thus, to develop a deterministic parsing routine we must provide buffer space for two input symbols.

As a result, there is a hierarchy of *LL* parsers whose distinguishing feature is the number of input symbols involved in their lookahead systems. These parsers are called **LL(k) parsers,** where k is an integer indicating the number of lookahead symbols employed by the parser. The example in Figure 2.27 is an *LL*(1) parser, whereas a parser based on the grammar in Figure 2.28 would be an *LL*(2) parser.

You may guess (correctly) that the burden of prediction placed on *LL*(*k*) parsers ultimately restricts the languages such parsers can handle. In truth, there are languages well within the bounds of pushdown parsers that cannot be recognized by any *LL*(*k*) parser, regardless of the size of k. The language $\{x^n : n \in \mathbb{N}\} \cup \{x^n y^n : n \in \mathbb{N}\}$, which we have already seen is deterministic context-free, is an example. Intuitively, any context-free grammar that generates this language must allow some nonterminal to be rewritten with either a string containing only *x*s or a string containing a balanced combination of *x*s and *y*s. This means there will be at least two rules for rewriting this nonterminal. In turn, any *LL*(*k*) parser will be faced with the problem of deciding which of these rules to apply when that nonterminal surfaces at the top of the stack. Unfortunately, regardless of the size of k, there are strings in the language in which the presence or absence of trailing *y*s cannot be detected without peeking beyond more than k *x*s. Thus, any particular *LL*(*k*) parser would be unable to handle the decisions required to parse this language.

The existence of a deterministic context-free language that cannot be parsed by any *LL*(*k*) parser suggests that there may be parsers based on the theory of pushdown automata that are more powerful than these predictive parsers—an hypothesis that leads us to the next section. However, with additional power comes additional complexity, and thus the simplicity of *LL*(*k*) parsers makes them a popular choice when they are capable of handling the language under investigation. For now, then, we forgo our quest for power and consider how *LL*(*k*) parsers can be simplified through the use of parse tables.

LL Parse Tables

A parse table for an *LL*(1) parser is a two-dimensional array. The rows are labeled by the nonterminals in the grammar on which the parser is based. The columns are labeled by the terminals in the grammar plus one additional column labeled EOS (representing the end-of-string marker). The $(m, n)^{th}$ entry in the table indicates what action should be performed when the nonterminal m appears on top of the stack and the lookahead symbol is n. If, in this setting, the nonterminal m should be replaced according to some rewrite rule, the right side of that rule appears as the $(m, n)^{th}$ entry. Otherwise, the entry contains an error indicator. For example, Figure 2.29 presents a parse table for the grammar of Figure 2.5.

Once a parse table has been constructed, the task of writing a program segment to parse the language is quite simple. All the segment must do is push the grammar's start symbol on the stack and then, until the stack becomes empty, either match terminals on top of the stack with those in the input or replace nonterminals on top of the stack as directed by the parse table. For example, the routine in Figure 2.30 is an *LL*(1) parser using the table in Figure 2.29.

	a	*b*	*z*	EOS
S	error	error	*zMNz*	error
M	*aMa*	error	*z*	error
N	error	*bNb*	*z*	error

Figure 2.29 An *LL*(1) parse table for the grammar in Figure 2.5

```
push (S);
read (Symbol);
while stack not empty do
    case top-of-stack of
        terminal: if top-of-stack = Symbol
            then pop stack and read (Symbol)
            else exit to error routine;
        nonterminal: if table [top-of-stack, Symbol] ≠ error
            then replace top-of-stack with table [top-of-stack, Symbol]
            else exit to error routine;
    end case
end while
if Symbol not end-of-string marker then exit to error routine
```

Figure 2.30 A generic *LL*(1) parsing routine

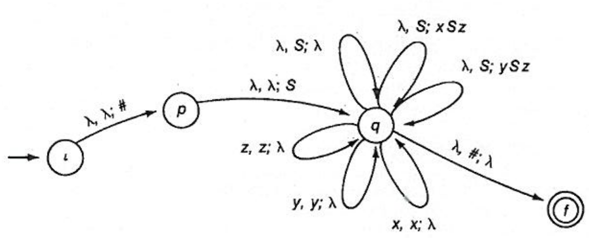
	x	y	EOS
S	xSy	λ	λ

Figure 2.31 An $LL(1)$ parse table for the grammar in Figure 2.7

Another advantage of using parse tables is that they allow the actual parsing algorithm to be standardized. The same algorithm can be used for any $LL(1)$ parser; to obtain a parser for another language, we merely substitute a new parse table in place of the old one. To emphasize this point, we close by observing that combining the program segment in Figure 2.30 with the parse table in Figure 2.31 produces a parser for the language generated by the grammar in Figure 2.7.

Exercises

1. Rewrite the program segment in Figure 2.30 using a repeat ⋯ until structure instead of a while structure.
2. Translate the transition diagram below directly into a program segment for parsing the language involved. What uncertainties must be resolved to obtain a deterministic routine?



3. Design an $LL(1)$ parse table for the following grammar.

$$S \rightarrow xSz$$
$$S \rightarrow ySz$$
$$S \rightarrow \lambda$$

4. How many symbols of lookahead would be required by an LL parser when parsing strings based on the following grammar? Design a corresponding parse table.

$$S \rightarrow xSy$$
$$S \rightarrow xy$$

2.5 *LR(k)* PARSERS

In the previous section we argued that the predictive nature of $LL(k)$ parsers restricts the class of languages that these parsers are able to handle. In this section we introduce a class of parsers that avoid many of the problems associated with their predictive counterparts. These parsers are known as **LR(k) parsers,** as they read their input from *Left* to right while constructing a *Rightmost* derivation of their input strings using a lookahead system involving k symbols.

The *LR* Parsing Process

Roughly speaking, an $LR(k)$ parser transfers symbols from its input to a stack until the uppermost stack symbols match the right side of some rewrite rule in the grammar on which the parser is based. At this point the parser can replace these symbols with the single nonterminal found on the left side of the rewrite rule before additional symbols are transferred from the input to the stack. In this manner the stack accumulates strings of terminals and nonterminals that are in turn replaced by nonterminals "higher" in the grammar. Ultimately, the entire contents of the stack collapses to the grammar's start symbol, indicating that the symbols read to that point form a string that can be derived by the grammar.

Based on this overall scheme, $LR(k)$ parsers are classified as **bottom-up parsers** since their activities correspond to constructing occurrences of nonterminals from their components until the grammar's start symbol is generated. In contrast, $LL(k)$ parsers are known as **top-down parsers** since they begin with the start symbol on the stack and repeatedly break the nonterminals on the stack into their components until a string of symbols matching the input string has been generated.

Let us back up now and develop the specifics of $LR(k)$ parsers. Recall that an $LL(k)$ parser is based on a pushdown automaton constructed from a context-free grammar; this construction is based on the process outlined in the proof of Theorem 2.2. In a similar manner, an $LR(k)$ parser is based on a pushdown automaton constructed from a context-free grammar, except that the automaton is constructed as described by the following five steps.