PROOF
Given a context-free grammar G we construct a pushdown automaton M as follows:

1. Designate the alphabet of M to be the terminal symbols of G, and the stack symbols of M to be the terminal and nonterminal symbols of G along with the special symbol #. (We may assume # is neither a terminal nor a nonterminal symbol in G.)
2. Designate the states of M to be ι, p, q, and f, ι being the initial state, and f being the only accept state.
3. Introduce the transition $(\iota, \lambda, \lambda; p, \#)$.
4. Introduce a transition $(p, \lambda, \lambda; q, S)$ where S is the start symbol in G.
5. Introduce a transition of the form $(q, \lambda, N; q, w)$ for each rewrite rule $N \rightarrow w$ in G. (Here we are using our new convention that allows a single transition to push more than one stack symbol. In particular, w may be a string of zero or more symbols including both terminals and nonterminals.)
6. Introduce a transition of the form $(q, x, x; q, \lambda)$ for each terminal x in G (i.e., for each symbol in the alphabet of M).
7. Introduce the transition $(q, \lambda, \#; f, \lambda)$.

A pushdown automaton constructed in this fashion will analyze an input string by first marking the bottom of the stack with the symbol #, then pushing the start symbol of the grammar on the stack and entering state q. From there until the symbol # returns to the top of the stack, the automaton will either pop a nonterminal from the top of the stack and replace it with the right side of an applicable rewrite rule, or it will pop a terminal from the top of the stack while reading the same terminal from the input. Once the symbol # returns to the top of the stack, the automaton will shift to its accept state f, indicating that the input consumed thus far is acceptable.

Note that the string of symbols making up the right side of a rewrite rule is pushed on the stack from right to left. Thus, the leftmost nonterminal in this string will be the first one to surface at the top of the stack; therefore, it will also be the first nonterminal on the stack to be replaced. Consequently, the automaton analyzes its input by performing a leftmost derivation according to the rules of the grammar on which it is based. But, as we have seen, the strings generated by a context-free grammar are exactly those that have a leftmost derivation. Thus, the automaton accepts exactly the same language as generated by the grammar.

■

The role of the various transitions constructed in the proof of Theorem 2.2 is probably best understood by means of an example. Let us consider the steps performed by the pushdown automaton described in the transition diagram of Figure 2.8, which was constructed from the context-free grammar in Figure 2.5. To evaluate the string *zazabzbz*, we start the machine from the configuration represented in Figure 2.9a. From this configuration the machine marks the bottom of the stack with the symbol # and shifts to state q while pushing the nonterminal S on the stack, to arrive at the configuration represented in Figure 2.9b. From this point on, the stack is used to hold a description of the structure that the machine hopes to find on the remaining part of its input stream. Thus, the S currently on the stack indicates that the machine hopes the remaining symbols on its input constitute a structure that can be generated from the nonterminal S.

However, since S is not a terminal the machine cannot expect to find the contents of its stack appearing on the input explicitly, and thus the nonterminal must be replaced before the machine tries to compare its stack directly to the input stream. This, in fact, is a general rule followed by the

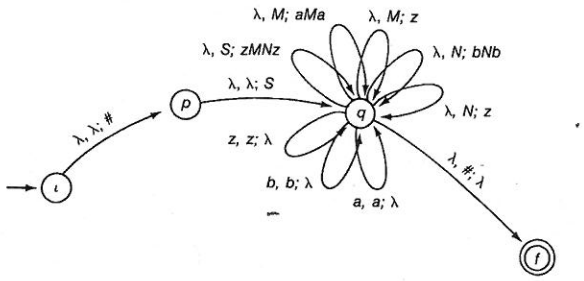

Figure 2.8 The transition diagram for a pushdown automaton constructed from the grammar of Figure 2.5 using the techniques presented in the proof of Theorem 2.2

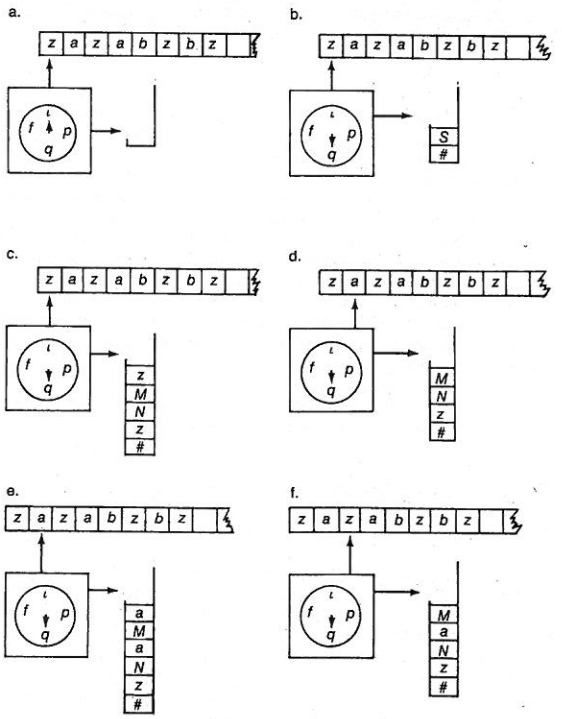

Figure 2.9 A pushdown automaton at work

machine: Any time the top of the stack contains a nonterminal, that nonterminal must be replaced with an equivalent yet more detailed description of the structure. This is the purpose of the transitions introduced by rule 5 in the proof of Theorem 2.2. Their execution replaces a nonterminal on top of the stack with a more detailed description of the structure according to some rewrite rule in the original grammar. Thus, the machine in our example proceeds by executing the transition $(q, \lambda, S; q, zMNz)$ to arrive at the configuration represented in Figure 2.9c.

Now the top of the machine's stack contains the terminal z, and thus the top of the stack can be compared to the input string via the transition $(q, z, z; q, \lambda)$. After this transition is executed the machine's stack contains MNz and the remaining symbols in the input string are *azabzbz*, as depicted in Figure 2.9d.

Again, the top of the stack contains a nonterminal that the machine must replace. However, in contrast to the previous replacement of the nonterminal S, there are two rewrite rules that can be applied to replace the nonterminal M. The pushdown automaton could execute the transition $(q, \lambda, M; q, z)$, which would cause the machine to fail on this attempt to accept the string, or it could execute $(q, \lambda, M; q, aMa)$, which in this case would be the correct choice. (This pushdown automaton is nondeterministic.) Let us assume the correct option since our goal is to observe how the machine could accept the input string in question. (Recall that a string is in the language of a nondeterministic machine if it is *possible* for the machine to accept the string.) After executing $(q, \lambda, M; q, aMa)$ the machine appears as shown in Figure 2.9e, with the terminal a on top of the stack. This terminal is then compared to the input string via the transition $(q, a, a; q, \lambda)$, leaving the stack containing $MaNz$ with the symbols *zabzbz* yet to be read from the input string, as shown in Figure 2.9f.

The remaining activities of the machine are summarized in Figure 2.10. Observe that reading the last symbol from the input uncovers the marker # on the stack. Then, the transition $(q, \lambda, \#; f, \lambda)$ transfers the machine to the accept state f where the input is declared accepted.

Let us now prepare ourselves for Theorem 2.3. Suppose we are given a pushdown automaton that accepts strings only with its stack empty. Then, from the automaton's point of view, its task is to try to move from its initial state to an accept state in such a way that its stack is in the same condition as when the computation started. How it accomplishes that goal depends on the transitions available. If, for example, ι is the machine's initial state and $(\iota, \lambda, \lambda; p, \#)$ is an available transition, then the automaton may try to accomplish its goal by first executing this transition. This would result in the machine finding itself in state p with the symbol # on its stack, and thus the goal of the machine would then be to move from state p to an accept state while removing the symbol # from its stack.

Contents of stack	Remaining input	Transition executed
λ	*zazabzbz*	(ι, λ, λ; *p*, #)
#	*zazabzbz*	(*p*, λ, λ; *q*, *S*)
S#	*zazabzbz*	(*q*, λ, *S*; *q*, *zMNz*)
zMNz#	*zazabzbz*	(*q*, *z*, *z*; *q*, λ)
MNz#	*azabzbz*	(*q*, λ, *M*; *q*, *aMa*)
aMaNz#	*azabzbz*	(*q*, *a*, *a*; *q*, λ)
MaNz#	*zabzbz*	(*q*, λ, *M*; *q*, *z*)
zaNz#	*zabzbz*	(*q*, *z*, *z*; *q*, λ)
aNz#	*abzbz*	(*q*, *a*, *a*; *q*, λ)
Nz#	*bzbz*	(*q*, λ, *N*; *q*, *bNb*)
bNbz#	*bzbz*	(*q*, *b*, *b*; *q*, λ)
Nbz#	*zbz*	(*q*, λ, *N*; *q*, *z*)
zbz#	*zbz*	(*q*, *z*, *z*; *q*, λ)
bz#	*bz*	(*q*, *b*, *b*; *q*, λ)
z#	*z*	(*q*, *z*, *z*; *q*, λ)
#	λ	(*q*, λ, #; *f*, λ)

Figure 2.10 The complete analysis of the string *zazabzbz* by the pushdown automaton described in Figure 2.8

In general, we represent such a goal by $<p, x, q>$, where p and q are states and x is a stack symbol of the machine. That is, $<p, x, q>$ represents the desire to move from state p to state q in such a way that the symbol x is removed from the top of the stack. A somewhat special case is denoted by $<p, \lambda, q>$, representing the goal of moving from state p to state q in a manner that leaves the stack essentially undisturbed. That is, upon reaching state q, the stack is to be the same as it was when in state p. An example of this is the original goal of moving from the initial state to an accept state as discussed above. Indeed, for each accept state f the automaton has a major goal represented by $<\iota, \lambda, f>$, where ι is the machine's initial state.

We are now in position to show that the languages accepted by pushdown automata are context-free.

THEOREM 2.3

For each pushdown automaton M, there is a context-free grammar G such that $L(M) = L(G)$.

PROOF

Given a pushdown automaton M, our task is to produce a context-free grammar G that generates the language $L(M)$. As a result of Theorem 2.1, we can assume that the pushdown automaton M accepts strings only with its stack empty. Having made this observation,

we construct G in such a way that its nonterminals represent the goals of M, as introduced above, and its rewrite rules represent refinements of large goals in terms of smaller ones.

More precisely, we specify that the nonterminals of G consist of the start symbol S together with all the goals of M, i.e., all syntactic structures of the form $<p, x, q>$ where p and q are states in M and x is either λ or a stack symbol in M. (Thus, even if M is a small automaton, G could have a large number of nonterminals.) The terminals of G are the symbols in the alphabet of M.

We are now prepared to introduce the first collection of rewrite rules of G. These rules are obtained as follows:

1. For each accept state f of M form the rewrite rule $S \rightarrow <\iota, \lambda, f>$, where ι is the initial state of M.

Rewrite rules obtained by #1 guarantee that any derivation using this grammar will start by replacing the grammar's start symbol with a major goal of the automaton.

Another collection of rewrite rules is obtained by

2. For each state p in M form the rewrite rule $<p, \lambda, p> \rightarrow \lambda$.

Rules obtained by #2 reflect the fact that a goal of moving from a state to itself without modifying the stack can be dropped from consideration.

Each of the remaining rewrite rules in G is constructed from a transition in M by either process #3 or #4 below.

3. For each transition $(p, x, y; q, z)$ in M (where y is not λ), generate a rewrite rule $<p, y, r> \rightarrow x<q, z, r>$ for each state r in M.

Rules generated by #3 reflect the fact that the goal of moving from state p to state r while removing y from the stack may be accomplished by first moving to state q while reading x from the input and exchanging z for y on the stack (using the transition $(p, x, y; q, z)$), and then trying to move from state q to state r while removing z from the stack.

4. For each transition of the form $(p, x, \lambda; q, z)$, generate all rewrite rules of the form $<p, w, r> \rightarrow x<q, z, k><k, w, r>$, where w is either a stack symbol or λ, while k and r (possibly equal) are states of M.

Rewrite rules constructed from #4 reflect the fact that the goal of moving from state p to state r while removing w from the stack may be accomplished by first moving to state q while reading x from the

may follow.) At this point execution would shift into M_2 (because of step 3 above), where the zs in the input string would lead to an accept state. After all, M_2 would proceed as though it were processing the latter portion of a string of the form $x^n y^{2n}$, but since its read instructions have been altered, it would actually read zs instead of ys.

To see that only strings of the form $x^n y^n z^n$ could be accepted, observe that to reach an accept state requires that M_1 process a string of the form $x^n y^n$ in order to transfer control to M_2, and then that M_2 find n zs, since M_2 proceeds as though it were processing the latter part of $x^n y^{2n}$.

We see, then, that our assumption that L can be accepted by a deterministic pushdown automaton leads to the conclusion that the language $\{x^n y^n z^n : n \in \mathbb{N}\}$ is context-free. But we know this is false. Consequently, our assumption must be in error: L cannot be accepted by a deterministic pushdown automaton.

■

In recognition of Theorem 2.6 we refer to the languages that are accepted by deterministic pushdown automata as the **deterministic context-free languages.** The fact that this class of languages does not include all the context-free languages indicates that our goal of building deterministic parsing routines based on pushdown automata will not be reached in the case of all the context-free languages, but only for the smaller class of deterministic context-free languages.

In fact, the picture is even bleaker. The deterministic pushdown automata we are considering accept strings without necessarily emptying their stacks, and as mentioned earlier, such a model is likely to lead to program segments that clutter a computer's memory with remnants of previous calculations. Unfortunately, this problem is more serious for deterministic pushdown automata than it is for ordinary pushdown automata, since we cannot modify every deterministic pushdown automaton so that it empties its stack before reaching an accept state.

To convince ourselves of this, consider the language L obtained by forming the union of $\{x^n : n \in \mathbb{N}\}$ and $\{x^n y^n : n \in \mathbb{N}\}$, which is accepted by the deterministic pushdown automaton represented by the skeletal diagram in Figure 2.22, and is therefore a deterministic context-free language. We claim that L cannot be accepted by a deterministic pushdown automaton that is required to empty its stack before reaching an accept state. Indeed, if M were such a machine and was given an input of the form $x^n y^n$, it would have to reach an accept state with an empty stack after reading each x. (Since the machine is deterministic, it must follow the same execution path

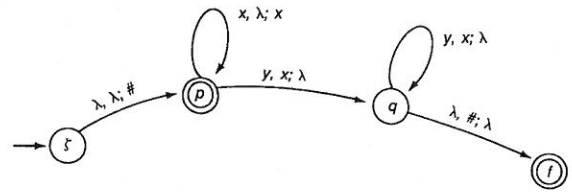

Figure 2.22 A skeletal diagram for a deterministic pushdown automaton M for which $L(M) = \{x^n : n \in \mathbb{N}^+\} \cup \{x^n y^n : n \in \mathbb{N}^+\}$

while processing the xs in $x^n y^n$ that it would follow when processing any string of xs, and since any string of xs is itself an acceptable string, the machine must empty its stack and move to an accept state after reading each x.) Now if we pick n to be greater than the number of states in M, we can conclude that at some point when processing the xs in the string $x^n y^n$ the machine must be in some state p with its stack empty at least twice. If m is the number of xs read between these visits to p, M must accept the string $x^{n+m} y^n$, which contradicts the definition of M.

In summary, we have found a hierarchy of languages associated with pushdown automata, as illustrated in Figure 2.23. The largest class is the collection of context-free languages that are accepted by general pushdown automata. Properly contained within this class are the deterministic context-free languages that are accepted by deterministic pushdown automata (those that are not required to empty their stacks). Then, properly contained within the class of deterministic context-free languages are the languages that can be accepted by deterministic pushdown automata that empty their stacks before accepting an input string.

The Lookahead Principle

At this point our hope of developing useable program segments for parsing a broad class of languages may begin to fade. It appears that any technique developed will be restricted to the smallest class of languages in Figure 2.23. Fortunately, the situation improves if we reconsider the manner in which the automata accept input strings. Recall from Section 2.1 that to accept a string a pushdown automaton must satisfy its accept criterion after reading the string but without moving its tape head any farther down the tape. This means that a pushdown automaton must reach an accept state without actually reading an end-of-string marker. The result is what we

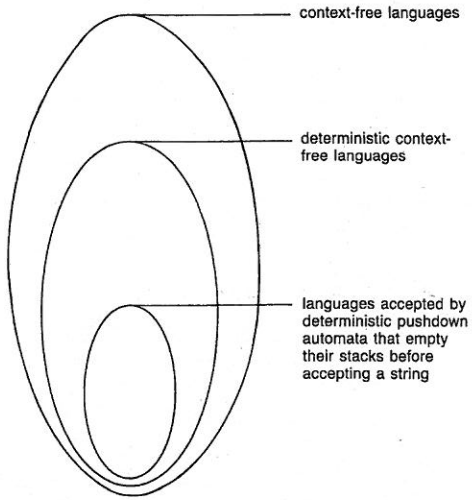

Figure 2.23 Classification of context-free languages

might consider a passive acceptance process in that the machine must "stumble" into an accept configuration without being told to do so by an end-of-string marker. In contrast, if we imagine a system by which the machine could peek at the next symbol on its tape (without actually advancing its tape head), it could detect an oncoming end-of-string marker and perform special "wrap-up" activities that it would not have executed otherwise.

We will refer to the technique of peeking at future symbols without actually reading them as the **lookahead principle.** Its application is exemplified by the while loop structure as displayed by the program segment in Figure 2.24. This segment essentially peeks at the next symbol and uses the information obtained to decide whether or not to process the symbol within the while structure. It is within the various case options that the routine finally decides whether to consume the symbol and prepare to process another one. Thus, the variable symbol is actually a holding place, or buffer, for the next input symbol. A symbol held there can be considered when


```
read (symbol);
while (symbol not end-of-string marker)
    case symbol of
        x: push (y) and read (symbol)
        y: exit to error routine
        z: push (#) and read (symbol)
    end case
end while
```

Figure 2.24 A typical ***while*** loop

making decisions but need not be consumed (or officially read) until the decision is made to process it. In particular, the end-of-string marker will not be consumed by this routine but will remain in the buffer after the while structure has terminated, where it will be available as input to the next routine. After all, in an actual compiler the end-of-string marker might well be the first symbol of the next structure to be analyzed.

We see, then, that applying the lookahead principle is a common programming technique. We are also about to see that by applying this principle when converting pushdown automata into program segments, we can construct programs that overcome the nondeterminism found in some pushdown automata. Thus, the theory of pushdown automata provides a foundation from which parsing routines for a wide range of context-free languages can be developed. How this is done is the subject of the next two sections.

Exercises

1. Apply Theorem 2.5 to prove that the language $\{x^m y^n x^m y^n x^m y^n : m, n \in \mathbb{N}^+\}$ is not context-free.
2. Give examples of each of the following:
   a. A language that is not context-free.
   b. A language that is context-free but not deterministic context-free.
   c. A language that is deterministic context-free but is not accepted by a deterministic pushdown automaton that is required to empty its stack.
   d. A language that is accepted by a deterministic pushdown automaton that is required to empty its stack but is not regular.
3. Draw the pertinent portion of a transition diagram for a deterministic pushdown automaton that from state p will move to state q while reading an x, popping a y, and pushing a z if the top of the stack contains a y, or will move to state q while reading an x, popping nothing, and pushing a z if the top of the stack is not a y.