to handle many of the syntax structures that occur in general programming languages.

In this chapter we generalize the concepts of finite automata and regular grammars to obtain techniques for parsing a broader range of languages known as the context-free languages. This we do by adding an internal memory system (in the form of a stack) to the automata being studied. This enhancement significantly increases the language processing powers of the automata and provides a context in which several efficient parsing algorithms are formulated; we will close this chapter with a discussion of some of the parsing techniques found in modern compilers.

To classify the languages recognized by the enhanced automata, we will again use the concept of a grammar. By allowing more complexity in the structure of a grammar's rewrite rules, we will be able to identify the grammars that generate the languages recognized by our enhanced machines. This grammatical characterization will prove advantageous in numerous settings; it is by means of such grammars that the syntax of most modern programming languages is described.

2.1 PUSHDOWN AUTOMATA

Recall from Chapter 1 that there is no finite automaton that can recognize the language $\{x^n y^n: n \in \mathbb{N}\}$. In fact, it was this language (with x being a left parenthesis and y being a right parenthesis) that was our primary example of the limitations of finite automata. We hypothesized that this problem occurs because finite automata have no way of remembering how many xs were found in the first part of the string, and so they are unable to check whether the same number of ys follow. Consequently, we now speculate that the problem could be resolved by adding some form of memory to the conceptual machine being considered.

Definition of Pushdown Automata

Following this lead, we introduce the class of machines known as **pushdown automata.** Such a machine is represented in Figure 2.1. As with finite automata, a pushdown automaton has an input stream and a control mechanism that can be in any one of a finite number of states. One of these states is designated as the initial state and at least one state is designated as an accept state. The major difference between pushdown automata and finite automata is that the former are endowed with a stack on which they can store information for later recall. (A stack is sometimes called a pushdown store or a pushdown stack and hence the name pushdown automata.)

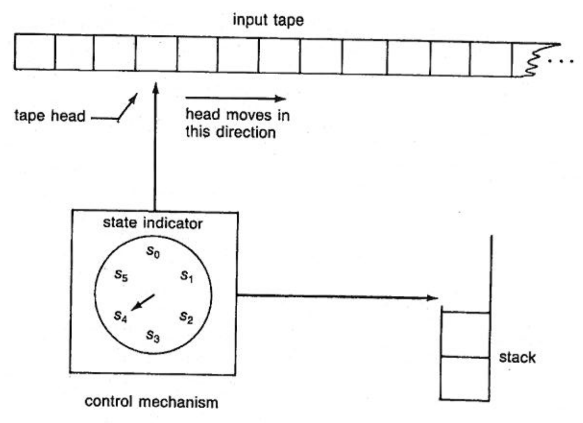


Figure 2.1 A pushdown automaton

The symbols that can be stored on this stack (known as the machine's **stack symbols**) constitute a finite set that may include some or all of the symbols in the machine's alphabet and perhaps some additional symbols for the machine to use as internal markers. For instance, a machine might wish to store special symbols on its stack to separate sections of the stack having different interpretations. More precisely, if a machine pushes a special symbol on the stack before performing any other calculations, then the presence of that symbol on top of the stack can be used as a "stack empty" indicator during later computations. We adopt the symbol # for this purpose in our examples.

Transitions executed by a pushdown automaton must be variations of the following basic sequence: read a symbol from the input, pop a symbol from the stack, push a symbol on the stack, and enter a new state. We represent this process with the notation $(p, x, s; q, y)$ where p, x, s, q, and y are the current state, the symbol in the alphabet that is read from the input, the symbol popped from the stack, the new state, and the symbol pushed on the stack, respectively. This notation is designed to indicate that the current state, input symbol, and top-of-stack symbol collectively help to determine the new state and the symbol to be pushed on the stack.

Variations of the basic transition process are obtained by allowing transitions to read, pop, or push the empty string. For example, a possible transition would be $(p, \lambda, \lambda; q, \lambda)$. That is, from state p the machine could decline to advance its tape head (which we think of as reading the empty string), decline to pop a symbol from its stack (pop the empty string), decline to push a symbol on the stack (push the empty string), and enter state q. Another example is the transition that merely transfers from state p to state q while popping the symbol s from the stack, as represented by $(p, \lambda, s; q, \lambda)$. Still other examples include such transitions as $(p, x, \lambda; q, z)$, $(p, \lambda, \lambda; q, z)$, etc.

To represent the collection of transitions available to a given pushdown automaton, it is convenient to use a transition diagram which resembles that of a finite automaton—states are represented as small circles and transitions as arcs between circles. However, in the case of pushdown automata, the labeling of arcs is more elaborate since there is more information to convey. An arc from state p to state q that represents the transition $(p, x, y; q, z)$ would be labeled $x, y; z$. For example, Figure 2.2 shows a transition diagram for a pushdown automaton in which the initial state is state 1 (as indicated by the pointer) and states 1 and 4 are accept states (indicated by double circles). From this diagram we can see that if the machine reads an x from the input while in state 2, it will push an x on the stack and return to state 2; if the machine reads a y from the input and is able to pop an x from the stack while in state 3, it will return to state 3; or if the symbol # is at the top of the stack when the machine is in state 3, the machine can pop this symbol and move to state 4.

As in the case of finite automata, pushdown automata can be implemented using a variety of technologies, and thus, we isolate the definitive properties of pushdown automata in the following formal definition.

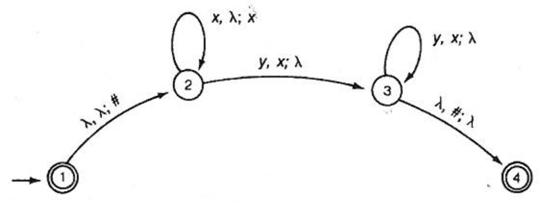


Figure 2.2 A transition diagram for a pushdown automaton

A pushdown automaton is a sextuple of the form $(S, \Sigma, \Gamma, T, \iota, F)$, where:

a. S is a finite collection of states.
b. Σ is the machine's alphabet.
c. Γ is the finite collection of stack symbols.
d. T is a finite collection of transitions.
e. ι (an element of S) is the initial state.
f. F (a subset of S) is the collection of accept states.

Pushdown Automata as Language Accepters

We can use pushdown automata to analyze strings in much the same way we used finite automata: We place the string to be analyzed on the machine's input tape with the machine's tape head over the leftmost cell of the tape. Then we start the machine from its initial state with an empty stack, and declare the string to be accepted if it is possible for the machine to reach an accept state after it has read the entire string. This does not mean the machine must find itself in an accept state immediately after it has read the last symbol from the input string, but it does mean the machine does not read any further down the tape. For instance, having read the last symbol from the input, a pushdown automaton may execute numerous transitions of the form $(p, \lambda, x; q, y)$ before it finally accepts the string.

We used "possible" in defining string acceptance by pushdown automata because the pushdown automata being considered here are nondeterministic; we have not made any restrictions as to the number of transitions that might be applicable at any given time. Hence, as in the case of nondeterministic finite automata, there may be several transition sequences that could be traversed from the machine's initial configuration, only one of which must lead to an accept state for us to declare the string accepted. (It is unfortunate that common terminology does not emphasize the nondeterministic nature of pushdown automata. Technically, they should be called nondeterministic pushdown automata.)

Again following the lead of finite automata, we refer to the collection of all strings accepted by a pushdown automaton M as the language accepted by the machine, denoted as $L(M)$. Once again we emphasize that the language $L(M)$ is not merely any collection of strings accepted by M, but *the* collection of *all* strings accepted by M.

An important class of machines is obtained by restricting the transitions available to pushdown automata to those of the form $(p, x, \lambda; q, \lambda)$. Steps of this form ignore the fact that the machine has a stack, and therefore the activities of the machine depend merely on the current state and the current input symbol. Thus, the class of machines constructed in this manner is the class of finite automata. Therefore, *the languages accepted by pushdown automata include the regular languages.*

Pushdown automata can also accept languages that finite automata cannot, one example being the language $\{x^n y^n : n \in \mathbb{N}\}$. In fact, Figure 2.2 is a transition diagram of such a machine. The first step is to mark the bottom of the stack with the symbol # and then push the *x*s on the stack as they are read from the input. Then the machine pops an x from the stack each time a y is read. Thus, when the symbol # reappears at the top of the stack, as many *y*s have been read as there were *x*s. Note that since the initial state is also an accept state, the machine is allowed to accept the string x^0y^0, which is λ.

Before we close this section, some additional comments are in order. Recall that the acceptance criterion given earlier allows a pushdown automaton to declare a string to be accepted without first emptying its stack. For example, a pushdown automaton based on the diagram in Figure 2.3 would accept the language $\{x^m y^n : m, n \in \mathbb{N}^+ \text{ and } m \geqslant n\}$, but those strings with more *x*s than *y*s would be accepted with *x*s remaining on the stack. (Note that the automaton could not accept strings with more *y*s than *x*s because it would not be able to read all the symbols from such a string.)

One might conjecture that the application of a theory based on such automata could easily lead to program modules that return control to other modules, while leaving remnants of their computations on the stack where these remnants might confuse future computations. For this reason, we often prefer to consider only pushdown automata that empty their stacks before reaching an accept state. We can see from Theorem 2.1 that this restriction does not reduce the power of the machines being considered.

THEOREM 2.1
For each pushdown automaton that accepts strings without emptying its stack, there is a pushdown automaton that accepts the same language but empties its stack before reaching an accept state.

PROOF
Suppose that $M = (S, \Sigma, \Gamma, T, \iota, F)$ is a pushdown automaton that accepts strings without necessarily emptying its stack. We could modify M as follows:

1. Remove the "initial" designation from the initial state of M. Then, introduce a new initial state and a transition that allows M to move from the new initial state to the old one while pushing a special symbol # (that was not previously in Γ) on the stack.

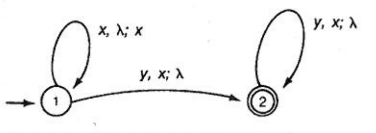


Figure 2.3 A pushdown automaton that can accept strings without an empty stack

2. Remove the accept status from each accept state of M. Then, introduce a state p along with transitions that allow the machine to move from each of the old accept states to p without reading, pushing, or popping any symbols.
3. For each x in Γ (not including #) introduce the transition $(p, \lambda, x; p, \lambda)$.
4. Introduce a new accept state q and the transition $(p, \lambda, \#; q, \lambda)$.

Note that the modified version of M merely marks the bottom of its stack before performing any computations, then simulates the computations of the original machine until that original machine would have declared the input to be accepted. At this point the modified machine shifts to state p, empties its stack, and then shifts to its accept state q while removing the bottom-of-stack marker. Thus, both the original and modified machines accept the same strings, except that the modified version reaches its accept state only when its stack is empty.

■

Figure 2.4 shows the result of applying the technique in the preceding proof to the diagram in Figure 2.3. A pushdown automaton based on this new diagram will accept the same strings as the original, but cannot accept a string unless its stack is empty.

Finally, it is important to remind ourselves again that the pushdown automata being considered here are nondeterministic. The modification process described in the proof of Theorem 2.1 could introduce numerous points of nondeterminism via the transitions that lead from the old accept states of the machine to the new state p.

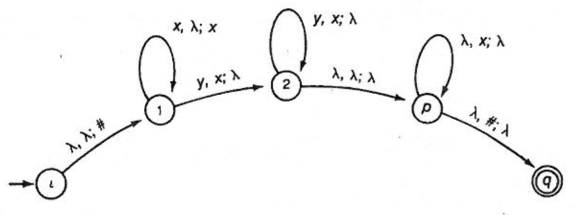
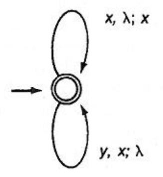


Figure 2.4 The diagram of Figure 2.3 after being modified to empty its stack before accepting a string

Exercises

1. Design a pushdown automaton M such that $L(M) = \{x^n y^m x^n : m, n \in \mathbb{N}\}$.
2. What language is accepted by the pushdown automaton whose transition diagram is shown below?



3. Modify the transition diagram in Exercise 2 so that the associated pushdown automaton will accept the same set of strings but will do so with its stack empty.
4. Show how two pushdown automata M_1 and M_2 can be combined to form a single pushdown automaton that would accept the language $L(M_1) \cup L(M_2)$.

2.2 CONTEXT-FREE GRAMMARS

Now that we have extended the machines under consideration from finite automata to pushdown automata, our investigation turns to the question

of what languages these extended machines can recognize. We begin to look for an answer by returning to the concept of grammars.

Definition of Context-Free Grammars

To characterize the languages recognized by pushdown automata, we introduce the concept of a **context-free grammar.** In contrast to regular grammars, these grammars have no restrictions on the form of the right side of their rewrite rules, although the left side of each rule is still required to be a single nonterminal. The grammar in Figure 2.5 is a context-free grammar but not a regular grammar.

The term "context-free" reflects the fact that since the left side of each rewrite rule can contain only a single nonterminal, the rule can be applied regardless of the context in which that nonterminal is found. In contrast, consider a rewrite rule whose left side contains more than a nonterminal, such as $xNy \rightarrow xzy$. Such a rule says that the nonterminal N can be replaced by the terminal z only when it is surrounded by the terminals x and y. The ability to remove N by applying the rule would therefore depend on the context rather than being context-free.

Context-free grammars generate strings via derivations, just as regular grammars. However, in the case of context-free grammars, questions can arise as to which nonterminal to replace at a particular step in the derivation. For example, when generating a string with the grammar of Figure 2.5, the first step yields the string $zMNz$, providing the option of replacing either the nonterminal M or N in the next step. Consequently, to generate the string $zazabzbz$, one might produce the derivation

$$S \Rightarrow zMNz \Rightarrow zaMaNz \Rightarrow zazaNz \Rightarrow zazabNbz \Rightarrow zazabzbz$$

by following the rule of thumb of always applying a rewrite rule to the leftmost nonterminal in the current string (this is called a **leftmost derivation**). One might also produce the derivation

$$S \Rightarrow zMNz \Rightarrow zMbNbz \Rightarrow zMbzbz \Rightarrow zaMabzbz \Rightarrow zazabzbz$$

by always applying a rewrite rule to the rightmost nonterminal, which

$S \rightarrow zMNz$
$M \rightarrow aMa$
$M \rightarrow z$
$N \rightarrow bNb$
$N \rightarrow z$

Figure 2.5 A context-free grammar that generates strings of the form $za^n za^n b^m zb^m z$, where $m, n \in \mathbb{N}$

would result in a **rightmost derivation.** One could even follow other patterns and obtain other derivations of the same string.

The point is that the order in which the rewrite rules are applied has no effect when determining which strings can be generated from a given context-free grammar. This becomes clear by recognizing that *if a string can be generated by any derivation, it can be generated by a leftmost derivation.* To see this we first consider the parse tree associated with a derivation.

A **parse tree** is nothing more than a tree whose nodes represent terminals and nonterminals from the grammar, with the root node being the grammar's start symbol and the children of each nonterminal node being the symbols that replace that nonterminal in the derivation. (No terminal symbol can be an interior node of the tree, and no nonterminal symbol can be a leaf.) A parse tree for the string *zazabzbz* using the grammar of Figure 2.5 and either of the previous derivations is shown in Figure 2.6.

Now, to see that any string generated from a context-free grammar can be generated by a leftmost derivation, we observe that derivations that differ merely in the order in which rewrite rules are applied correspond to the same parse tree. The order in which the rules are applied simply reflects the order in which the branches of the parse tree are constructed. A leftmost derivation corresponds to constructing the parse tree in a left-branch-first fashion, whereas a rightmost derivation corresponds to a right-branch-first construction. However, the order in which the branches are constructed does not affect the structure of the final tree, as each branch is independent

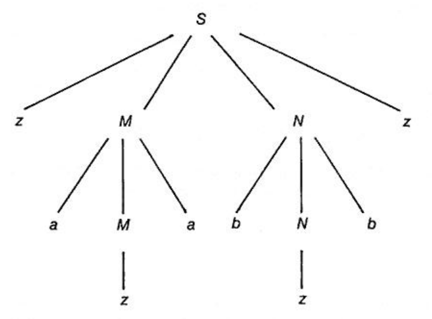


Figure 2.6 The parse tree for the string *zazabzbz* using the grammar of Figure 2.5

$S-xSy$
$S-\lambda$

Figure 2.7 A context-free grammar that generates strings of the form x^ny^n, where $n \in \mathbb{N}$

of the others. Thus, given a derivation that is not leftmost, one could construct the associated parse tree and then construct a leftmost derivation of the same string by a systematic "leftmost evaluation" of the tree.

Finally, we note that the flexibility allowed in context-free grammars provides for the construction of a grammar that generates the language $\{x^ny^n: n \in \mathbb{N}\}$ as shown in Figure 2.7. This example combined with the fact that any regular grammar is also a context-free grammar allows us to conclude that the context-free grammars generate a larger collection of languages than the regular grammars. We call the languages generated by context-free grammars the **context-free languages.**

Context-Free Grammars and Pushdown Automata

Before considering the relationship between context-free grammars and pushdown automata, we must clarify a point in notation. It will be convenient in the following discussions to consider single transitions that push more than one symbol on the stack such as $(p, a, s; q, xyz)$. In this case the symbols z, y, and x (in that order) are to be pushed on the stack. Hence, after executing the transition, x will be at the top of the stack (with y below it, and z on the bottom). Note that allowing transitions of this form is merely a matter of notational convenience and does not add any capabilities to the machine. Indeed, the multiple push transition $(p, a, s; q, xyz)$ could be simulated by the sequence of traditional transitions $(p, a, s; q_1, z)$, $(q_1, \lambda, \lambda; q_2, y)$, and $(q_2, \lambda, \lambda; q, x)$, where q_1 and q_2 are additional states that cannot be reached by any other sequence of transitions.

Now we must show that *the languages generated by context-free grammars are exactly the languages accepted by pushdown automata.* This we do in two stages. First, we show that for any context-free grammar G, there is a pushdown automaton M such that $L(M) = L(G)$ (Theorem 2.2). Then, we show that for any pushdown automaton M, there is a context-free grammar G such that $L(G) = L(M)$ (Theorem 2.3).

THEOREM 2.2
For each context-free grammar G, there is a pushdown automaton M such that $L(G) = L(M)$.