Chapter 18

Pushdown Automata, Context Free Grammars and Languages

18.1 Pushdown automata

We turn next to a class of automata which are more powerful than the finite automata in the sense that they accept a larger class of languages. These are the pushdown automata (pda's).

A pda is essentially a finite automaton with an auxiliary tape on which it may read, write, and erase symbols. This tape is organized as a *stack* or *pushdown store* similar in principle to the spring-loaded devices for holding plates seen in cafeterias. Both work on the basis of "last in, first out;" that is, the most recently added item is the first one to be removed. Items below the topmost ones cannot be reached without first removing items above them on the stack

Pda's, like finite automata, read their input tapes from left to right and have a finite number of internal states. There is a designated initial state, and a set of final, or accepting, states. The transitions of a pda, however, allow the top symbol of the stack to be read and removed, added to, or left unchanged We can represent these transitions schematically as $(q_i, a, A) \rightarrow (q_j, \gamma)$, where q_i and q_j are states, a is a symbol of the input alphabet, A is a symbol of the stack alphabet (which need not be the same as the input alphabet), and γ is a string of stack symbols Such an instruction

is interpreted as follows: when in state q_i, reading a on the input tape, and reading A at the top of the stack, go to state q_j and *replace A by the string γ*. If γ were, for example, the string BC, the A would be removed and BC added to the stack (in the order B first, C next) so that C would now become the top symbol on the stack. In case γ is e, the empty string, the net effect is to remove ("pop") A from the stack. The symbol next below A, if any, would then become the top symbol. If γ were, for example, AB, the effect would be to add ("push") a B on top of the A. If γ were A, the transition would leave the stack unchanged.

We also allow e to appear in the position of A in the above schema. In this case the transition does not depend on the contents of the stack since e can always be read at the top of the stack whatever it may actually contain. Note that the e here does *not* indicate that the stack must by empty. If $A = e$ and $\gamma = B$, for example, the transition would push B onto whatever was already on the stack.

The stack is assumed to be empty at the beginning of a computation with the pda in its initial state and the reading head positioned over the left-most symbol of the input. An input tape is accepted if the computation leads to a situation in which all three of the following are simultaneously true:

(i) the entire input has been read

(ii) the pda is in a final state

(iii) the stack is empty

One could define acceptance by empty stack or final state or, as we have done, by both, and the resulting classes of automata turn out to be equivalent. This choice is convenient for our purposes. The following is an example of a pda which accepts the language $\{a^n b^n \mid n \geq 0\}$:

(18–1)

States:	$K = \{q_0, q_1\}$
Input alphabet:	$\Sigma = \{a, b\}$
Stack alphabet:	$\Gamma = \{A\}$
Initial state:	q_0
Final states:	$F = \{q_0, q_1\}$
Transitions:	$\Delta = \left\{ \begin{array}{l} (q_0, a, e) \to (q_0, A) \\ (q_0, b, A) \to (q_1, e) \\ (q_1, b, A) \to (q_1, e) \end{array} \right\}$

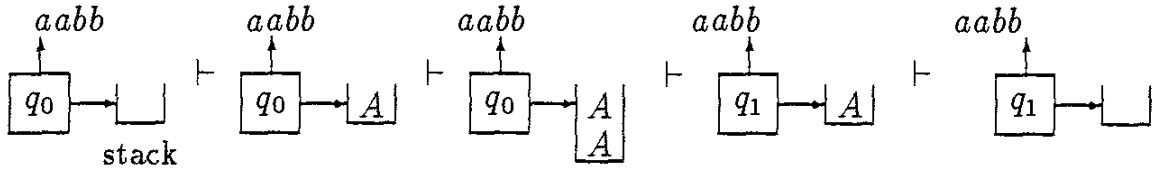

Figure 18–1

Fig 18-1 shows how this pda would accept input *aabb*.
Since the entire input is read and the pda halts in a final state with an empty stack, this input is accepted.

The reader should also be able to verify the following statements about the behavior of this pda:

(i) *ba* is rejected: the pda blocks in state q_0 and fails to read the entire input

(ii) *aaabb* is rejected: the pda halts in state q_1 with A on the stack

(iii) *aabbb* is rejected: the pda fails to read the last b since there is no A on the stack

(iv) e is accepted: the computation begins and ends in q_0 with an empty stack

This machine works by using its stack as a counter for keeping track of the number of a's in the initial part of the input string Once a b is found, it switches to state q_1 and begins popping an A from the stack for each b encountered. Only if the number of b's equals the number of a's will the stack be empty at the end (and the entire input string read).
Problem: Why is it necessary to go to a new state when the first b is encountered, i.e., why not stay in state q_0?
Here is an example of a pda which accepts the language $\{xx^R \mid x \in \{a,b\}^*\}$

(18–2) $K = \{q_0, q_1\}$, $\Sigma = \{a, b\}$, $\Gamma = \{A, B\}$,
Initial state $= q_0$, $F = \{q_0, q_1\}$

$$\Delta = \left\{ \begin{array}{ll} (q_0, a, e) \to (q_0, A) & (q_0, b, B) \to (q_1, e) \\ (q_0, b, e) \to (q_0, B) & (q_1, a, A) \to (q_1, e) \\ (q_0, a, A) \to (q_1, e) & (q_1, b, B) \to (q_1, e)\} \end{array} \right\}$$

This machine works by putting an image of the left half of its input string on the stack (in the form of capital letters), and then, after non-deterministically "guessing" that the middle of the string has been reached, comparing each input symbol in the right half against the top symbol of the stack. If the symbols correspond, the stack symbol is removed; if not, the machine blocks. Since symbols come off the stack in the reverse of the order in which they went on, the stack will be emptied just in case the right half of the input is the reversal or "mirror image" of the left half.

This pda is non-deterministic since there is more than one move available to the automaton in certain situations. If, for example, it has just read an a in state q_0 (and therefore put an A on the stack) and is now reading another a, it could execute either the first or the third instruction above. The former corresponds to the "pushing" mode in which the image of the left half is being placed onto the stack; the latter corresponds to a decision that the middle of the string has just been encountered and that it is time to switch from pushing to popping mode. As with the finite automata, we say that a non-deterministic pda accepts if there exists *at least one* computational path on which the input is accepted; an input is rejected if there is no such accepting path. By this definition, the above machine accepts the language $\{xx^R \mid x \in \{a,b\}^*\}$.

The pda in (18-1), by contrast, is said to be deterministic, in that it has at most one move available to it for any situation. Unlike the finite automata, we do not insist that deterministic pda's have one move available for *every* situation; rather, that in no situation is more than one move allowed. One can determine from inspection of the instructions of a pda whether it is deterministic or properly non-deterministic. A pda is deterministic iff there are no two distinct instructions $(q_i, a, A) \rightarrow (q_j, \gamma)$ and $(q_i, a, A') \rightarrow (q_k, \delta)$ such that $A = A'$ or $A = e$ or $A' = e$. Thus, as with the finite automata, the deterministic machines are a proper subset of the non-deterministic.

The question then naturally arises as to whether deterministic and non-deterministic pda's are equivalent. The answer is that they are not, although it is not a simple matter to give a proof of this fact. We will simply note that this result agrees with our intuition that no deterministic pda could be devised to accept $\{xx^R \mid x \in \{a,b\}^*\}$, there being no way in general for a pda reading strictly left to right to tell with certainty when the center of the input string has been reached. In contrast, the language $\{xcx^R \mid x \in \{a,b\}^*\}$ in which the center of the string is marked by the c is easily acceptable by a deterministic pda.

It is often a matter of some practical interest to be able to tell whether a non-deterministic pda language is also accepted by some deterministic pda. This is so because many programming languages—languages used for writing instructions for computers to execute—belong to the class of non-deterministic pda languages. When programs are compiled, i e., translated into sequences of 1's and 0's for execution by the computer, the compilation process is carried out by what is in effect a pda, and if this pda can be made deterministic, then the process can be made more efficient by avoiding back-tracking or the pursuit of alternative paths. Most programming languages in current use are in fact deterministic pda languages (or nearly so), but it is an unfortunate fact that there is no way to tell in general whether any arbitrarily given non-deterministic pda language is also a deterministic pda language.

We end this section by giving formal definitions of pda's and the related notions of situation, acceptance, and so on. Note that in the following we have generalized the notion of a transition of a pda to allow the possibility that a *string* of input symbols and a *string* of stack symbols can be read on a single move. This does not affect the power of the automata and has the advantage of bringing the definitions into a form parallel to those for non-deterministic finite automata. A non-deterministic finite automaton will thus appear formally as a non-deterministic pda which never makes use of its stack.

DEFINITION 18.1 *A* non-deterministic pushdown automaton *is a sextuple* $\langle K, \Sigma, \Gamma, \Delta, s, F\rangle$, *where* K *is a finite set of states,* Σ *is a finite set (the input alphabet),* Γ *is a finite set (the stack alphabet),* $s \in K$ *is the initial state,* $F \subseteq K$ *is the set of final states, and* Δ, *the set of transitions, is a finite subset of* $K \times \Sigma^* \times \Gamma^* \times K \times \Gamma^*$. ■

DEFINITION 18.2 *A* situation *of a pda is a quadruple* (x, q, y, z) *where* $q \in K$, $x, y \in \Sigma^*$, *and* $z \in \Gamma^*$. ■

The intended interpretation is that the pda is in state q with x to the left of the reading head, y to the right of the reading head with the left-most symbol of y currently being scanned, and z *is* the contents of the stack.

DEFINITION 18 3 *Given a non-deterministic pda* M, *we say that situation* (x, q, y, z) produces (x', q', y', z') in-one-move *iff* $x' = x\alpha$, $y = \alpha y'$, $z = \gamma w$, $z' = \delta w$, *and* $(q, \alpha, \gamma) \to (q', \delta) \in \Delta$. ■

DEFINITION 18.4 Produces *is the reflexive, transitive closure of the* produces-in-one-move *relation.*

These are denoted, as usual, by $\vdash_M^*$ *and* $\vdash_M$, *respectively.* ■

DEFINITION 18.5 *Given a pda* M, *a string* $x \in \Sigma^*$ *is* accepted *iff* $(e, s, x, e) \vdash_M^* (x, q, e, e)$ *for some* $q \in F$. *The* language accepted by M *is the set of all strings accepted.* ■

DEFINITION 18.6 *A pda is* deterministic *iff for no pair of distinct transitions,* $(q_i, x_i, \gamma_i) \rightarrow (q_j, \delta_j)$ *and* $(q_k, x_k, \gamma_k) \rightarrow (q_l, \delta_l)$ *is it the case that* $q_i = q_k$, *and* x_i *is a substring of* x_k or *vice-versa, and* γ_i *is a substring of* γ_k or *vice-versa.* ■

18.2 Context free grammars and languages

Non-deterministic pda's accept exactly the languages generated by context free (Type 2) grammars. Recall that in a context free grammar every rule is of the form $A \rightarrow \psi$, where A is a non-terminal symbol and ψ is any string, possibly empty, from the union of the terminal and non-terminal alphabets. It follows from this definition that every right-linear grammar is also a context free grammar, and therefore that the regular languages are contained in the context free languages. This containment is proper since, as we have seen, $\{a^n b^n \mid n \geq 0\}$ is not a regular language, but it can be generated by the simple context free grammar containing only the two rules $S \rightarrow aSb$ and $S \rightarrow e$.

The proof of the equivalence of context free languages and non-deterministic pda languages is too long and complex to give here. To give something of the flavor of this proof, we will show an algorithm for constructing from any given context free grammar an equivalent non-deterministic pda (but we will not prove formally that the constructed pda is actually equivalent). For the construction in the reverse direction—from non-deterministic pda to equivalent context free grammar—we refer the reader to Hopcroft and Ullman (1979) or Lewis and Papadimitriou (1981), which also contain references to the original sources.

Given a context free grammar $G = \langle V_N, V_T, S, R \rangle$, we construct an equivalent non-deterministic pda M as follows. The states of M are q_0 and q_1, with q_0 being the start state and q_1 being the only final state. The input alphabet is V_T and the stack alphabet $V_N \cup V_T$. The transitions of M are constructed out of the rules of G in the following way:

(i) M contains the instruction $(q_0, e, e) \rightarrow (q_1, S)$

(ii) For each rule of the grammar $A \rightarrow \psi$, M contains an instruction $(q_1, e, A) \rightarrow (q_1, \psi)$.

(iii) For each symbol $a \in V_T$, M contains an instruction $(q_1, a, a) \rightarrow (q_1, e)$.

As an example, let us take G to be as follows:

(18–3) $V_N = \{S\}$; $V_T = \{a, b\}$; $R = \{S \rightarrow aSb, S \rightarrow e\}$

(This is the grammar we referred to above which generates $\{a^n b^n \mid n \geq 0\}$.)

According to the construction procedure just given, M will contain the following:

(18–4) $K = \{q_0, q_1\}$; $\Sigma = \{a, b\}$; $\Gamma = \{S, a, b\}$; $s = q_0$; $F = \{q_1\}$;

$$\Delta = \left\{ \begin{array}{l} (q_0, e, e) \rightarrow (q_1, S) \\ (q_1, e, S) \rightarrow (q_1, aSb) \\ (q_1, e, S) \rightarrow (q_1, e) \\ (q_1, a, a) \rightarrow (q_1, e) \\ (q_1, b, b) \rightarrow (q_1, e) \end{array} \right\}$$

M accepts the input string $aabb$ by the following computation: $(e, q_0, aabb, e)$ $\vdash$ $(e, q_1, aabb, S) \vdash (e, q_1, aabb, aSb) \vdash (a, q_1, abb, Sb) \vdash (a, q_1, abb, aSbb) \vdash$ $(aa, q_1, bb, Sbb) \vdash (aa, q_1, bb, bb) \vdash (aab, q_1, b, b) \vdash (aabb, q_1, e, e)$

M works by loading S onto its stack and then simulating a derivation there by manipulations which correspond to the rewriting rules of G. When a terminal symbol appears at the top of the stack, it is popped off if it matches the symbol being read on the input; otherwise the computation blocks. When a non-terminal appears at the top of the stack it is rewritten in a way licensed by the rules of G. Thus, M carries out what is in effect a left-most derivation (one in which the left-most non-terminal symbol is rewritten at each step) according to the grammar. If the derived terminal string matches the input, the stack will be emptied, the entire input read, and the string accepted.

Note that pda's constructed in this way will in general be non-deterministic since there may be in the grammar more than one rule rewriting some non-terminal A.

18.3 Pumping Theorem for cfl's

There is a Pumping Theorem for the cfl's which is similar in form to the Pumping Theorem for fal's. It is useful primarily in showing that a particular language is not context free.

The theorem makes use of the fact that a derivation by a cfg can be naturally associated with a parse tree (see Section 16 4), and the fact that the maximum width of any such parse tree is constrained by its height. Let us see what this means in more detail.

Given a cfg G, there is some maximum number of symbols on the right hand side of any rule. Suppose, for example, that no rule has a right hand side longer than 4 symbols. This means that in one rule application, the width of the tree (the number of symbols in its yield) can have increased by at most 3. If each of the 4 symbols just introduced should happen to be expanded into 4 symbols, then by these steps the single node could have grown into 16 symbols, *but no more than 16 symbols*. If we define the height of a tree to be the length of the longest path in it extending continuously downward from the root, then what we have said is that (for a given grammar) the maximum width of a parse tree is bounded by its height. More specifically, if n is the maximum length of a right side of the rules of G, then the maximum width of a parse tree generated by G of height h is just n^h. (It may in fact be considerably less than this, depending on the exact nature of G, but we are interested here only in setting an upper bound) To phrase this another way, if a parse tree for some grammar G has width greater than n^m, where n is the maximum length of the right side of rules of G, we can be sure that the height of the tree is greater than m.

Let us now suppose that G has m non-terminal symbols in its alphabet. If we find a parse tree generated by G of width greater than n^m, then there must be some continuously descending path in the tree of length greater than m, and thus some non-terminal symbol must appear at least twice along this path (there being only m symbols to choose from). Let us represent this situation by the diagram in Fig. 18-2.

The repeated non-terminal is called A. S dominates a terminal string w, and so each non-terminal in the tree must also dominate some terminal string. Let x be the terminal string dominated by the lower A, and vxy the terminal string dominated by the upper A. (x must be a substring of this string since, by hypothesis, the lower A is dominated by the upper A.) Let u and z be terminal strings dominated by S to the left and right, respectively,

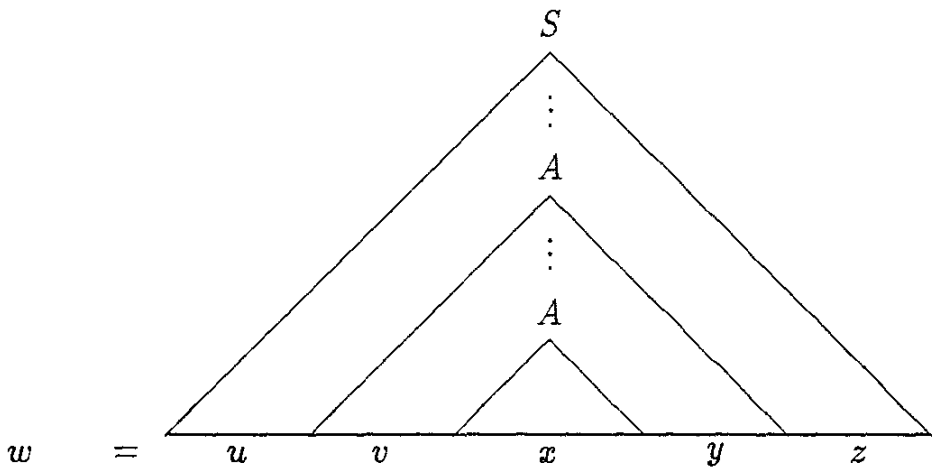

Figure 18–2.

of vxy. This general situation must obtain whenever there is a repeated non-terminal along some path in a parse tree.

But note that the lower subtree rooted by A could have stood in the place in the tree where the upper A-rooted tree stands. The rules of G are, after all, context free, so if it is possible to rewrite A in one position ultimately to yield x, the same is possible in any position in which A appears. Thus, the tree in Fig. 18-3 must also be generated by G.

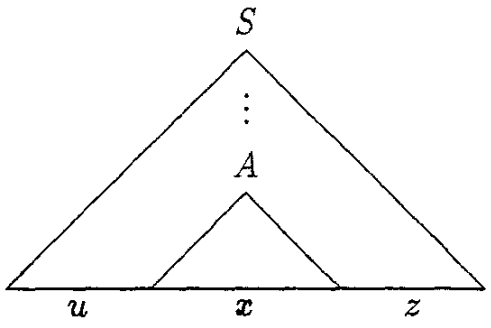

Figure 18–3.

Further, the lower A in Fig. 18-2 could have been rewritten as the upper

one was, to produce the tree shown in Fig. 18-4.

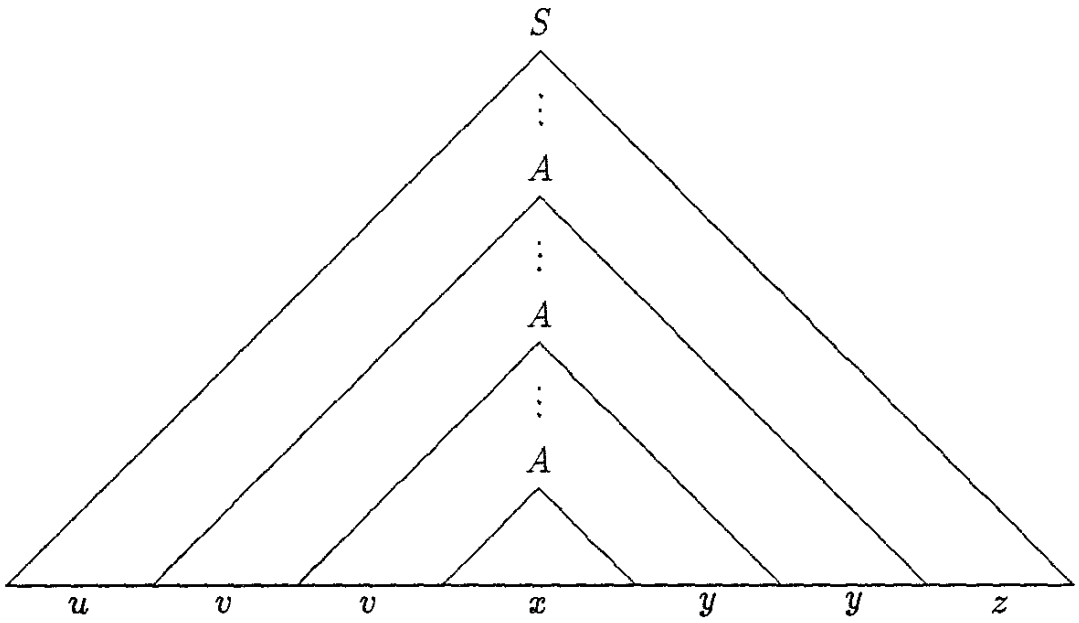

Figure 18–4.

Since we know that the derivation can be terminated by rewriting the lowest A, ultimately to give x, it follows that the string $uvvxyyz$ must also be generated by G. The process just illustrated could, of course, be carried out any finite number of times; thus, uv^ixy^iz is generated for all $i \geq 0$. We are now ready to state the Pumping Theorem for context free languages.

THEOREM 18.1 *If L is an infinite context free language, then there is some constant K such that any string w in L longer than K can be factored into substrings $w = uvxyz$ such that v and y are not both empty and $uv^ixy^iz \in L$ for all $i \geq 0$.* ■

Note that if L is an infinite language then it is guaranteed to contain strings longer than any given constant K. What is K? It is a number which depends on the grammar for L and which is big enough to ensure that any strings longer than K have derivation trees with a repeated non-terminal along some path. There is always such a K for any cfg since G contains a finite number of non-terminal symbols, and there is some maximum to the degree of branching allowed on the right sides of rules.

What about the stipulation that v and y are not both empty? If they were, we could get from A to A on a branch of a derivation tree without generating any terminal symbols either to the left or right. This situation could arise, since rules of the form $A \to B$, $B \to C$, $C \to A$ are allowed in a cfg. However, not all the non-terminals in the grammar could appear only in rules of this form or else the grammar could not generate an infinite language. That is, there must be *some* non-terminal symbol A which can be repeated along some path, i.e., $A \overset{*}{\underset{G}{\Rightarrow}} vAy$, such that not both v and y are empty, and further, such a non-terminal must appear in the derivation of any string longer than K.

Note that like the Pumping Theorem for fal's, this theorem is a conditional but not a biconditional. Given a language L, it is not particularly informative to find strings u, v, x, y, z such that $uv^ixy^iz \in L$ for all $i \geq 0$. Rather, we use the theorem in its contrapositive form: if there do not exist strings u, v, x, y, z such that..., then we can conclude that L is not an infinite cfl. Let us see how this can be done in the case of the language $L = \{a^n b^n c^n \mid n \geq 0\}$.

Suppose L were context free Since it is infinite, the Pumping Theorem must apply, and there would be some constant K such that any string in L longer than K—let us choose $a^K b^K c^K$, for example—would be factorable into $uvxyz$ such that v and y are not both empty and the v and y are pumpable. We show that no such factorization can exist.

First, v cannot consist of both a's and b's, because when it is pumped, it would produce strings containing b's before a's, which cannot be in L. Similarly, v cannot consist of both b's and c's, and the same argument applies to the other pumpable term, y. Therefore, the only possibilities remaining are for v to consist of just a's or just b's or just c's. Then no matter how we choose y, the result of pumping v and y simultaneously gives strings not in L. Suppose, for example, that v consists of a's and y of b's. Then on pumping v and y the a's and b's increase but not the c's, and we get strings not in L. The other cases are similar. We conclude that there is no choice of u, v, x, y, z meeting the conditions of the Pumping Theorem for this language; therefore, L is not context free.

18.4 Closure properties of context free languages

Given the class of context free languages (identical to the non-deterministic pda languages), we want to investigate whether this class is closed under

operations such as union, intersection, complementation, etc. We will see that unlike the fal's the cfl's are not so conveniently and tidily closed under all these operations

Union: Given two cfg's $G_1 = \langle V_{N_1}, V_{T_1}, S_1, R_1 \rangle$ and $G_2 = \langle V_{N_2}, V_{T_2}, S_2, R_2 \rangle$, we form grammar G in the following way If the non-terminals of G_1 and G_2 are not disjoint sets, we make them so (by appending primes to every symbol of G_2, say). The start symbol of G we take to be S, and G contains, in addition to R_1 and R_2, the rules $S \rightarrow S_1$ and $S \rightarrow S_2$ G is context free, and it generates $L(G_1) \cup L(G_2)$ since the start symbol may be either rewritten as S_1, whereupon G behaves like G_1, or as S_2, whereupon G behaves like G_2. A string $x \in (V_{T_1} \cup V_{T_2})^*$ is generated by G just in case it is generated by G_1 or by G_2 (or both) Since this method of construction is general (and, as the reader will have noted, quite similar to that used in showing that the fal's are closed under union; see Section 17.2), we conclude that the cfl's are closed under union

Concatenation (or Set Product): The method of construction is similar to that for union except that instead of the two rules mentioned there we add to G the single rule $S \rightarrow S_1 S_2$. Thus, G will generate all strings of the form xy such that $x \in L(G_1)$ and $y \in L(G_2)$. Further, G will generate only such strings, and, again, since the method of construction is general, we conclude that the cfl's are closed under concatenation.

Kleene Star: Given $G = \langle V_N, V_T, S, R \rangle$, we construct G^* as follows: The start symbol of G^* is S', and G contains, in addition to all the rules in R, the rules $S' \rightarrow e$ and $S' \rightarrow S'S$. G^* generates all strings in $(L(G))^*$ since by application of the rules rewriting S', G^* produces strings S^n for all $n \geq 0$. Each such S can be rewritten to produce a string in $L(G)$, and e is produced by the rule $S' \rightarrow e$. Further, all strings in $(L(G))^*$ can be generated in this way. Thus, the cfl's are closed under Kleene star.

Problem: Why was it necessary to introduce the new start symbol S'? Why not just add the rules $S \rightarrow SS$ and $S \rightarrow e$?

Intersection: The cfl's are *not* closed under intersection To see this, we note that the languages $\{a^i b^i c^j \mid i, j \geq 0\}$ and $\{a^k b^l c^l \mid k, l \geq 0\}$ are both context free. The former is generated by a grammar containing the rules:

(18–5)
$$\begin{aligned} S &\rightarrow BC \\ B &\rightarrow aBb \\ B &\rightarrow e \\ C &\rightarrow cC \\ C &\rightarrow e \end{aligned}$$

and the grammar for the latter is similar The intersection of these two languages, however, is $\{a^n b^n c^n \mid n \geq 0\}$, which we proved above by the Pumping Theorem not to be a context free language

Recall in this connection what it means to say that a set is not closed under a certain operation. We have shown that the intersection of two cfl's is sometimes not a cfl It is not claimed that the result is *never* a cfl. Indeed, this could not be so, since the regular languages are necessarily cfl's, and since the regular languages are closed under intersection, the result is regular, hence, context free.

Complementation: The cfl's are not closed under complementation. Given two cfl's L_1 and L_2 over some alphabet Σ, if their complements L_1' and L_2' (i e , $\Sigma^* - L_1$ and $\Sigma^* - L_2$, respectively) were context free, then so would be their union, $L_1' \cup L_2'$. The complement of this, in turn, $(L_1' \cup L_2')'$, would also be context free, but this is equal by DeMorgan's Laws to $L_1 \cap L_2$, which is not necessarily context free. Hence, the complement of a cfl is not necessarily a cfl

Intersection with a Regular Language: Although the intersection of two arbitrary cfl's L_1 and L_2 is not in general a cfl, it happens that if one of the languages is restricted to being regular, then the intersection is always a cfl. A demonstration of this fact is somewhat involved and depends on constructing a non-deterministic pda accepting $L_1 \cap L_2$ out of a pda accepting L_1 and a finite automaton accepting L_2. (The non-closure of cfl's under intersection implies that it is not in general possible to coalesce two non-deterministic pda's in this way.)

The closure of the cfl's under intersection with a regular language can be a convenience in showing certain languages not to be context free. For example, the language $L = \{x \in \{a, b, c\}^* \mid x$ contains equal numbers of a's, b's, and c's$\}$, although not context free, resists application of the Pumping Theorem However, if L is first intersected with the regular language $a^*b^*c^*$, the result is $\{a^n b^n c^n \mid n \geq 0\}$, which we have shown not to be context free. Now if L were a cfl, its intersection with a regular language would also be a cfl; hence, L is not context free.

18.5 Decidability questions for context free languages

Context-free languages differ from fal's also in respect to which questions can be answered by algorithm. For the fal's we saw that there were algorithms

for answering questions such as membership, emptiness, etc. We shall see that some of these questions have algorithmic solutions for the cfl's and some do not.

Membership: Given an arbitrary cfg G and an arbitrary string x, is x generated by G? One might propose an algorithm for answering this question of the following sort: Start producing derivations by G in some systematic fashion, discarding any whose last lines are longer than x. This will be some finite number of derivations. If x has not been generated by this point, it is not going to be

As matters stand, this algorithm might not be successful for two reasons: first, the grammar may contain rules of the form $A \rightarrow e$, which allows derivations to become shorter. Thus, we cannot be sure that we can stop examining derivations when their last lines reach the length of x A derivation might produce longer strings which then shrink to produce x Second, because rules such as $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$, etc. might be present in the grammar, derivations might continue indefinitely without their final strings getting any longer This subverts our claim that we need to examine only a finite number of derivations to see if any generate x. The proposed algorithm would work, however, if we could somehow contrive to remove all rules of both types from a cfg while leaving the generative power of the grammar unchanged. We will now show that this can in fact be done.

If there is a rule $A \rightarrow e$ in the grammar, this rule can be dispensed with if we add more rules to the grammar in the following way. Whenever A appears on the right side of a rule we add another rule identical to it except that the A on the right is deleted. For example, given rules $A \rightarrow e$ and $B \rightarrow cAbBa$, we would add the rule $B \rightarrow cbBa$. Now what would have been accomplished by application of the first two rules in sequence can be accomplished by the last rule alone. The original rule is of course preserved since there may be other rules expanding A. We continue in this way for every rule containing an A on the right side, and repeat the process for every non-terminal which can be rewritten as e. (Note that if such a non-terminal appeared more than once on a right side, e.g., $B \rightarrow aAbAc$, we would add rules $B \rightarrow abAc$, $B \rightarrow aAbc$, and $B \rightarrow abc$.) This process must eventually come to an end since there are finitely many rules to begin with, there are a finite number of non-terminals, and a finite number of rules are added at each step. When we are done, we may remove all rules of the form $A \rightarrow e$ from the grammar since they are superfluous. The one exception is the rule $S \rightarrow e$, if it is present, which must be retained in order to generate the empty string as a member of

the language The presence of this rule will not interfere with the workings of our algorithm, however, since if there is a derivation of some non-empty string x which involves one or more applications of the rule $S \rightarrow e$, there will also be, after carrying out the procedure just outlined, a derivation of x which does not involve any applications of this rule Thus, we can produce an equivalent cfg in which derivations are essentially non-shrinking.

What about rules of the form $A \rightarrow B$? These can be removed in the following way. Pick a rule of the form $A \rightarrow w$, where w is something other than a single non-terminal symbol. (If there are no such rules, the grammar generates no terminal strings and the membership question is settled at once.) Now for each non-terminal C distinct from A, determine whether $C \overset{*}{\underset{G}{\Rightarrow}} A$, i.e., whether C can be rewritten in some finite number of steps to give A This can be done by examining all sequences $C, B_1, B_2, \ldots, B_n, A$ (where B_1, B_2, are single non-terminals) of length no more than the number of non-terminals in the grammar to see whether they are allowed by the grammar. The restriction on length of derivations is possible because if there is such a derivation with repeated symbols $C \Rightarrow \ldots \Rightarrow B_i \Rightarrow \ldots \Rightarrow B_i \Rightarrow B_j \Rightarrow \ldots \Rightarrow A$ then there is also a shorter one with the section between repetitions removed: $C \Rightarrow \ldots \Rightarrow B_i \Rightarrow B_j \Rightarrow \ldots \Rightarrow A$. Thus, the number of non-terminals fixes an upper bound on the length of such derivations, and we can effectively determine whether $C \overset{*}{\underset{G}{\Rightarrow}} A$. If so, then we add the rule $C \rightarrow w$ to the grammar, and thus the derivation $C \overset{*}{\underset{G}{\Rightarrow}} A \underset{G}{\Rightarrow} w$ can be replaced by the derivation $C \underset{G}{\Rightarrow} w$ directly. We continue this process for all rules of the form $A \rightarrow w$ $(w \notin V_N)$ and all non-terminals distinct from A. When we have finished, all rules of the form $A \rightarrow B$ can be removed from the grammar without affecting the terminal strings generated.

Once all these steps have been carried out, the proposed algorithm for answering the membership question can be executed and is guaranteed to lead to an answer in a finite amount of time. No claim is made that this is particularly *efficient* way to answer the membership question, but we are not concerned with relative amounts of computational labor here—only with showing that the question can be answered in some finite amount of time by mechanical means.

Emptiness: Does an arbitrarily given cfg G generate any strings at all? There is an algorithm for answering this, the emptiness question, and it depends on the following observation. If G generates any terminal strings, it generates some terminal string with a parse tree which has no non-terminals repeated along any path. Refer again to Fig. 18-2 which we used in proving

the Pumping Theorem. If a parse tree for some terminal string has a repeated non-terminal A along some path, the subtree rooted by the upper A could be replaced by the subtree rooted by the lower A, and the result is also a parse tree for a terminal string generated by the grammar (cf. Fig. 18-3). Clearly all repeated occurrences of non-terminals could be removed in this way. Thus, in order to see whether G generates any terminal strings, all we have to do is examine the finite number of parse trees which contain no repeated elements along any path. The exact number we need to look at will depend on the number of non-terminals in the grammar and the degree of branching allowed by the rules of G, but it will be finite. If no terminal string has appeared as the yield of a tree by this point, none is ever going to appear. This answers the emptiness question.

Undecidable questions: Many problems concerning context free languages have no algorithmic solution. We cannot provide demonstrations of these facts here since they require results from Turing machine theory, which we have not yet examined. We will simply list some of the more important undecidable questions:

a. Given an arbitrary context free grammar G, there is no algorithm for determining:

(i) whether $L(G) = V_T^*$, i.e., whether G generates all strings over the terminal alphabet

(ii) whether the complement of $L(G)$, i.e., $V_T^* - L(G)$, is empty, infinite, regular, or context free.

b. Given two arbitrary context free grammars G_1 and G_2, there is no algorithm for determining:

(i) whether $L(G_1) \subseteq L(G_2)$

(ii) whether $L(G_1) = L(G_2)$

(iii) whether $L(G_1) \cap L(G_2) = \emptyset$

(iv) whether $L(G_1) \cap L(G_2)$ is infinite, regular, or context free.

Recall that the lack of a single algorithm for deciding every one of an infinite class of cases does not preclude the possibility that for *certain* context free grammars these questions might be answerable. In fact, for all context free grammars which happen to be regular, there are algorithmic solutions to all of the above questions, as we saw in Section 17.3.1.

18.6 Are natural languages context free?

In Section 17.3.2 we showed that English could not be a regular language. Could it or any other natural language be context free? This question has attracted considerable attention in the years since Chomsky first outlined the hierarchical categorization of formal languages (1963). The prevailing view has been that natural languages are not context free. Attempts to demonstrate this have usually centered on finding instances of so-called "cross-serial" dependencies of arbitrarily large size in some particular language. In a cross-serial dependency, items are linked in left-to-right order as shown in Fig 18-5.

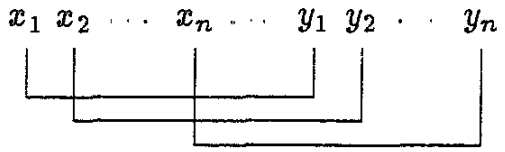

Figure 18–5.

(Compare this to the nested dependencies illustrated in Fig 17-17.)

A language such as $\{xx \mid x \in \{a,b\}^*\}$ exhibits cross-serial dependencies (for strings of length $2n$ in the i^{th} and $(n+i)^{th}$ symbols must match) and is not context free.

Pullum and Gazdar (1982) review the various attempts to establish that natural languages are not context free and find all of them either formally or empirically flawed. In the latter category are instances in which claims that certain forms are ungrammatical are unjustified and more probably involve semantic or pragmatic anomaly. A common formal mistake was to assume that because one has found a subset of a language which exhibits cross-serial dependency the language as a whole is thereby shown not to be context free. Note that the non-context free language $\{xx \mid x \in \{a,b\}^*\}$ is a subset of the context free (indeed, regular) language $\{a,b\}^*$.

Recently, however, evidence has appeared for the non-context freeness of Swiss German which seems unassailable on either formal or empirical grounds (Shieber, 1985). Swiss German, like its much studied cousin Dutch, allows cross-serial order in dependent clauses. Sentences like the following are grammatical:

(18–6) Jan säit das mer em Hans es huus hälfed aastriiche
John said that we Hans–Dat the house–Acc helped paint
"John said that we helped Hans paint the house "

(18–7) Jan säit das mer d'chind em Hans es huus lönd hälfe aastriiche
John said that we the children – Acc Hans – Dat the house – Acc let help paint
"John said that we let the children help Hans paint the house."

The NP's and the V's of which the NP's are objects occur in cross-serial order: in (18-7) *d'chind* ("the children") is the object of *lönd* ("let"), *em Hans* is the object of *hälfe* ("help"), and *es huus* ("the house") is the object of *aastriiche* ("paint"). Furthermore, the verbs mark their objects for case: *Hälfe* requires dative case, while *lönd* and *aastriiche* require accusative Sentences in which the case marking does not follow this restriction are uniformly rejected by native speakers as ungrammatical. (Since case marking is unlikely to be accounted for semantically or pragmatically, this avoids the empirical trap mentioned above) It also appears that there are no limits other than performance constraints on the length of such constructions in grammatical sentences of Swiss German. Shieber then intersects Swiss German with the regular language:

(18–8) R = Jan säit das mer (d'chind)*(em Hans)* es huus haend wele (laa)* (hälfe)* aastriiche
John said that we (the children)* (Hans)* the house have wanted to (let)*(help)* paint

(Here the *haend wele* ("have wanted to") is present in order to put all the succeeding verbs in their infinitive forms. Schieber shows that this insertion does not affect grammaticality judgments.)

The result of intersecting R and Swiss German is all sentences of the following form:

(18–9) L = Jan säit das mer (d'chind)^n(em Hans)^m es huus haend wele (laa)^n (hälfe)^m aastriiche

where the number of nouns in the accusative case matches the number of

verbs requiring this case and similarly for the dative case, and all accusative case nouns (except the constant "es huus") precede all dative case nouns and all accusative-case marking verbs precede all dative-case marking verbs. The strings of this sublanguage of Swiss German are of the form $wa^nb^mxc^nd^my$, which can be shown to be non-context free by the Pumping Theorem. Since the context free languages are closed under intersection with a regular language, this demonstrates fairly convincingly that Swiss German is not context free.

Note that the formal difficulty mentioned above has been avoided. The sublanguage of Swiss German shown to be non-context free is not merely a subset of the original but a subset obtained by intersection with a regular language The latter operation preserves context freeness while the operation of simply selecting a subset in general does not.

Attempts to arrive at the corresponding results for Dutch could not succeed because Dutch does not have verbs with differing case-marking properties which can occur in arbitrarily long cross-serial dependent clauses.

Exercises

1. Construct context free grammars generating each of the following languages.

   (a) $L_1 = a^nb^ma^n(n, m \geq 1)$

   (b) $L_2 = a^nb^na^mb^m(n, m \geq 1)$

   (c) $L_3 = \{x \mid x \in \{a, b\}^*$ and x contains twice as many b's as a's $\}$

   (d) $L_4 = \{xx^R \mid x \in \{a, b\}^*\}$

   (e) $L_5 = \{x \in \{a, b\}^* \mid x = x^R\}$

2. Show that for every context free grammar there is an equivalent grammar in which all productions are of the form $A \rightarrow BC$ or $A \rightarrow a$ (A, B, C in V_N, a in V_T). Such a grammar is said to be in *Chomsky Normal Form* (Chomsky, 1959).

3. Show by means of the Pumping Theorem that the following languages are not context free.

   (a) $\{a^{n^2} | (n \geq 1)\}$

   (b) $\{a^n | n$ is prime (i.e., divisible only by 1 and by itself)$\}$

4. Given a language L we define the *reversal* of L, denoted L^R, as $\{x^R \mid x \in L\}$, where x^R is the reversal of x. Show that the context free languages are closed under reversal.

5. Construct a deterministic pda which accepts the language $(ab)^n(cd)^n$ for all $n \geq 1$. (The parentheses are not part of the language nor do they indicate optionality; they are used in the expression above only for grouping.)

6. Construct a deterministic pda which accepts every string of the form xc, where x is a string of a's and b's of length 0 or more in which the total number of a's is exactly equal to the total number of b's.

7. Construct a non-deterministic pda which accepts every string which is of the form $a^n b a^n$ or of the form $a^{2n} b a^n$ for all $n \geq 1$

8. Is the union of the languages of two determinsitc pda's necessarily the language of some deterministic pda? Justify your answer.

9. Show by means of the Pumping Theorem for context free languages that $a^i b^j c^{\max(i,j)}$ is not context free, where $\max(i,j)$ is the larger of i and j.