COMS W3261
Computer Science Theory
Chapter 7 Notes

Alexander Roth

2014–10–04

Properties of Context-Free Languages

Our first task is to simplify context-free grammars; these simplifications make it easier to prove facts about CFL's, since we can claim that if a language is a CFL, then it has a grammar in some special form.

Normal Forms for Context-Free Grammars

The goal of this section is to show that every CFL (without ϵ) is generated by a CFG in which all productions are of the form $A \to BC$ or $A \to \alpha$, where A, B, and C are variables, and a is a terminal. This form is called *Chomsky Normal Form.* To get there, we need to make a number of preliminary simplifications, which are themselves useful in various ways:

1. We must eliminate *useless symbols*, those variables or terminals that do not appear in any derivation of a terminal string from the start symbol.
2. We must eliminate *ϵ-productions*, those of the form $A \to \epsilon$ for some variable A.
3. We must eliminate *unit productions*, those of the form $A \to B$ for variables A and B.

Eliminating Useless Symbols

We say a symbol X is *useful* for a grammar $G = (V, T, P, S)$ if there is some derivation of the form $S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} w$, where w is in T^*. Note that X may be in either V or T, and the sentential form $\alpha X \beta$ might be the first or last in the derivation. If X is not useful, we say it is *useless.*

Our approach to eliminating useless symbols begins by identifying the two things a symbol has to be able to do to be useful:

1. We say X is *generating* if $X \overset{*}{\Rightarrow} w$ for some terminal string w. Note that every terminal is generating, since w can be that terminal itself, which is derived by zero steps.

2. We say X is *reachable* if there is a derivation $S \overset{*}{\Rightarrow} \alpha X \beta$ for some α and β.

If we eliminate the symbols that are not generating first, and then eliminate from the remaining grammar those symbols that are not reachable, we shall have only the useful symbols left.

Theorem 1. Let $G = (V, T, P, S)$ be a CFG, and assume that $L(G) \neq \emptyset$; i.e., G generates at least one string. Let $G_1 = (V_1, T_1, P_1, S)$ be the grammar we obtain by the following steps:

1. First eliminate nongenerating symbols and all productions involving one or more of those symbols. Let $G_2 = (V_2, T_2, P_2, S)$ be this new grammar. Note that S must be generating, since we assume $L(G)$ has at least one string, so S has not been eliminated.

2. Second, eliminate all symbols that are not reachable in the grammar G_2.

Then G_1 has no useless symbols, and $L(G_1) = L(G)$.

Computing the Generating and Reachable Symbols

Let $G = (V, T, P, S)$ be a grammar. To compute the generating symbols of G, we perform the following induction.

BASIS: Every symbol of T is obviously generating; it generates itself.

INDUCTION: Suppose there is a production $A \to \alpha$, and every symbol of α is already known to be generating. Then A is generating. Note that this rule includes the case where $\alpha = \epsilon$; all variables that have ϵ as production body are surely generating.

Theorem 2. The algorithm above finds all and only the generating symbols of G.

Now, let us consider the inductive algorithm whereby we find the set of reachable symbols for the grammar $G = (V, T, P, S)$. Again, we can show that by trying our best to discover reachable symbols, any symbol we do not add to the reachable set is really not reachable.

BASIS: S is surely reachable.

INDUCTION: Suppose we have discovered that some variable A is reachable. Then for all productions with A in the head, all the symbols of the bodies of those productions are also reachable.

Theorem 3. The algorithm above finds all and only the reachable symbols of G.

Eliminating ϵ-Productions

Now, we shall show that ϵ-productions, while a convenience in many grammar-design problems, are not essential. Of course without a production that has an ϵ body, it is impossible to generate the empty string as a member of the language. Thus, what we actually prove is that if language L has a CFG, then $L - \{\epsilon\}$ has a CFG without ϵ-productions. If ϵ is not in L, then L itself is $L - \{\epsilon\}$, so L has a CFG without ϵ-productions.

We must first discover which variables are "nullable". A variable A is nullable if $A \overset{*}{\Rightarrow} \epsilon$. If A is nullable, then whenever A appears in a production body, say $B \to CAD$, A might derive ϵ. We make two versions of the production, one without A in the body and one with A. However, if we use the version with A present, then we cannot allow A to derive ϵ.

Let $G = (V, T, P, S)$ be a CFG. WE find all the nullable symbols of G by the following iterative algorithm.

BASIS: If $A \to \epsilon$ is a production of G, then A is nullable.

INDUCTION: If there is a production $B \to C_1 C_2 \cdots C_k$, where each C_i is nullable, then B is nullable. Note that each C_i must be a variable to be nullable, so we only have to consider productions with all-variable bodies.

Theorem 4. In any grammar G, the only nullable symbols are the variables found by the algorithm above.

Now, we give the construction of a grammar without ϵ-productions. Let $G = (V, T, P, S)$ be a CFG. Determine all the nullable symbols of G. We construct a new grammar $G_1 = (V, T, P_1, S)$, whose set of productions P_1 is determined as follows.

For each production $A \to X_1 X_2 \cdots X_k$ of P, where $k \geq 1$, suppose that M of the k X_i's are nullable symbols. The new grammar G_1 will have 2^m versions of this production, where the nullable X_i's in all possible combinations are present or absent. There is one exception: if $m = k$, i.e. all symbols are nullable, then we do not include the case where all X_i's are absent.

Since this construction eliminates ϵ-productions, we shall claim that for every CFG G, there is a grammar G_1 with no ϵ- productions, such that

$$L(G_1) = L(G) = \{\epsilon\}$$

Theorem 5. If the grammar G_1 is constructed from G by the above construction for eliminating ϵ-productions, then $L(G_1) = L(G) - \{\epsilon\}$.

Eliminating Unit Productions

A *unit production* is a production of the form $A \to B$, where both A and B are variables. These productions can be useful. For example, consider the following unambiguous grammar for simple arithmetic expressions:

$$\begin{aligned} I &\to a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F &\to I \mid (E) \\ T &\to F \mid T * F \\ E &\to T \mid E + T \end{aligned}$$

We could expand the T in the production $E \to T$ in both possible ways, replacing it by the two productions $E \to F \mid T * F$. That change still doesn't eliminate unit productions, because we have introduced unit production $E \to F$ that as not previously part of the grammar. Further expanding $E \to F$ by the two productions for F gives us $E \to I \mid (E) \mid T * F$. We still have a unit production; it is $E \to I$. Finally, we are left with:

$$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E) \mid T * F \mid E + T \mid$$

Now the unit production for E is gone. Note that $E \to a$ is *not* a unit production, since the lone symbol in the body is a terminal, rather than a variable as is required for unit productions.

This technique often works. However, it can fail if there is a cycle of unit productions, such as $A \to B$, $B \to C$, and $C \to A$. The technique that is guaranteed to work involves first finding all those pairs of variables A and B such that $A \overset{*}{\Rightarrow} B$ using a sequence of unit productions only. Note that it is possible for $A \overset{*}{\Rightarrow} B$ to be true even though no unit productions are involved.

Once we have determined all such pairs, we can replace any sequence of derivation steps in which $A \Rightarrow B_1 \Rightarrow B_2 \to \cdots \Rightarrow B_n \Rightarrow \alpha$ by a production that uses the nonunit production $B_n \to \alpha$ directly from A; that is, $A \to \alpha$. To begin, here is the inductive construction of the pairs (A, B) such that $A \overset{*}{\Rightarrow} B$ using only unit productions. Call such a pair a *unit pair.*

BASIS: (A, A) is a unit pair for any variable A. That is $A \overset{*}{\Rightarrow} A$ by zero steps.

INDUCTION: Suppose we have determined that (A, B) is a unit pair, and $B \to C$ is a production, where C is a variable. Then (A, C) is a unit pair.

Theorem 6. The algorithm above finds exactly the unit pairs for a CFG G.

To eliminate unit productions, we proceed as follows. Given a CFG $G = (V, T, P, S)$, construct CFG $G_1 = (V, T, P_1, S)$:

1. Find all the unit pairs of G.

2. For each unit pair (A, B), add to P_1 all the productions $A \to \alpha$, where $B \to \alpha$ is a nonunit production in P. Note that $A = B$ is possible; in that way, P_1 contains all the nonunit productions in P.

Theorem 7. If grammar G_1 is constructed from grammar G by the algorithm described above for eliminating unit productions, $L(G_1) = L(G)$.

We want to convert any CFG G into an equivalent CFG that has no useless symbols, ϵ-productions, or unit productions. Some care must be taken in the order of application of the constructions. A safe order is:

1. Eliminate ϵ-productions.
2. Eliminate unit productions.
3. Eliminate useless symbols.

We must order the three steps above as shown or the result might still have some of the features we are seeking to eliminate.

Theorem 8. If G is a CFG generating a language that contains at least one string other that ϵ, then there is another CFG G_1 such that $L(G_1) = L(G) - \{\epsilon\}$, and G_1 has no ϵ-productions, unit productions, or useless symbols.

Chomsky Normal Form

Every nonempty CFL without ϵ has a grammar G in which all productions are in one of two simple forms, either:

1. $A \to BC$, where A, B, and C, are each variables, or
2. $A \to a$, where A is a variable and a is a terminal.

Further, G has no useless symbols. Such a grammar is said to be a *Chomsky Normal Form*, or CNF.

To put a grammar in CNF, start with one that satisfies the restrictions of Theorem 8; that is, the grammar has no ϵ-productions, unit productions, or useless symbols. Every production from such a grammar is either of the form $A \to a$, which is already in a form allowed by CNF, or it has a body of length 2 or more. Our tasks are:

a) Arrange that all bodies of length 2 or more consist only of variables.

b) Break bodies of length 3 or more into a cascade of productions, each with a body consisting of two variables.

The construction of (a) is as follows. For every terminal a that appears in a body of length 2 or more, create a new variable say A. This variable has only one production, $A \to a$. Now, we use A in place of a everywhere a appears in a body of length 2 or more.At this point, every production has a body that is either a single terminal or at least two variables and no terminals.

For step (b), we must break those productions $A \to B_1 B_2 \cdots B_k$, for $k \geq 3$, into a group of productions with two variables in each body. We introduce $k-2$ new variables, $C_1, C_2, \ldots, C_{k-2}$. The original production is replaced by the $k-1$ productions

$$A \to B_1 C_1, C_1 \to B_2 C_2, \ldots, C_{k-3} \to B_{k-2} C_{k-2} \to B_{k-1} B_k$$

Theorem 9. If G is a CFG whose language contains at least one string other than ϵ, then there is a grammar G_1 in Chomsky Normal Form, such that $L(G_1) = L(G) - \{\epsilon\}$.

The Pumping Lemma for Context-Free Languages

The "pumping lemma for context-free languages" says that in any sufficiently long string in a CFL, it is possible to find at most two short, nearby substrings, that we can "pump" in tandem. That is, we may repeat both of the strings i times, for any integer i, and the resulting string will still be in the language.

The Size of Parse Trees

One of the users of CNF is to turn parse trees into binary trees. These trees have some convenient properties, one of which we shall exploit.

Theorem 10. Suppose we have a parse tree according to a Chomsky-Normal-Form grammar $G = (V, T, P, S)$, and suppose that the yield of the tree is a terminal string w. If the length of the longest path is n, then $|w| \leq 2^{n-1}$.

Statement of the Pumping Lemma

The pumping lemma for CFL's is quite similar to the pumping lemma for regular languages, but we break each string z in the CFL L into five parts, and we pump the second and fourth, in tandem.

Theorem 11. (The pumping lemma for context-free languages) Let L be a CFL. Then there exists a constant n such that if z is any string in L such that $|z|$ is at least n, then we can write $z = uvwxy$, subject to the following conditions:

1. $|vwx| \leq n$. That is, the middle portion is not too long.
2. $vx \neq \epsilon$. Since v and x are the pieces to be "pumped," this condition says that at least one of the strings we pump must not be empty.
3. For all $i \geq 0$, uv^iwx^iy is in L. That is, the two strings v and x may be "pumped" any number of times, including 0, and the resulting string will still be a member of L.

Applications of the Pumping Lemma for CFL's

Notice that, like the earlier pumping lemma for regular languages, we use the CFL pumping lemma as an "adversary game," as follows.

1. We pick a language L that we want to show is not a CFL.
2. Our "adversary" gets to pick n, which we do not know, and we therefore must plan for any possible n.

3. We get to pick z, and we may use n as a parameter when we do so.

4. Our adversary gets to break z into $uvwxy$, subject only to the constraints that $|vwx| \leq n$ and $vx \neq \epsilon$.

5. We "win" the game, if we can, by picking i and showing that uv^iwx^iy is not in L.

Closure Properties of Context-Free Languages

First, we introduce an operation called substitution, in which we replace each symbol in the strings of one language by an entire language. This operation, a generalization of the homomorphism, is useful in proving some other closure properties of CFL's, such as the regular-expression operations: union, concatenation, and closure. We show that CFL's are closed under homomorphisms and inverse homomorphisms. Unlike the regular languages, CFL's are not closed under intersection or difference. However, the intersection or difference of a CFL and a regular language is always a CFL.

Substitutions

Let Σ be an alphabet, and suppose that for every symbol a in Σ, we choose a language L_a. These chosen languages can be over any alphabets, not necessarily Σ and not necessarily the same. This choice of languages defines a function s (*a substitution*) on Σ, and we shall refer to L_a as $s(a)$ for each symbol a.

If $w = a_1a_2 \cdots a_n$ is a string in Σ^*, then $s(w)$ is the language of all strings $x_1x_2 \cdots x_n$ such that strings x_i is in the language $s(a_i)$, for $i = 1, 2, \ldots, n$. Put another way, $s(w)$ is the concatenation of the languages $s(a_1)s(a_2) \cdots s(a_n)$. We can further extend the definition of s to apply to languages: $s(L)$ is the union of $s(w)$ for all strings w in L.

Theorem 12. If L is a context-free language over alphabet Σ, and s is a substitution on Σ such that $s(a)$ is a CFL for each a in Σ, then $s(L)$ is a CFL.

Applications of the Substitution Theorem

Theorem 13. The context-free languages are closed under the following operations:

1. Union.

2. Concatenation.

3. Closure (*), and positive closure ($^+$).

4. Homomorphism.

Reversal

The CFL's are also closed under reversal. We cannot use the substitution theorem, but there is a simple construction using grammars.

Theorem 14. If L is a CFL, then so is L^R.

Intersection With a Regular Language

The CFL's are not closed under intersection. On the other hand, there is a weaker claim we can make about intersection. The context-free languages are closed under the operation of "intersection with a regular language."

Theorem 15. If L is a CFL and R is a regular language, then $L \cap R$ is a CFL.

Since we know that the CFL's are not closed under intersection, but are closed under intersection with a regular language, we also know about the set-difference and complementation operations on CFL's.

Theorem 16. The following are true about CFL's L, L_1, and L_2, and a regular language R.

1. $L - R$ is a context-free language.
2. $\overline{L}$ is not necessarily a context-free language.
3. $L_1 - L_2$ is not necessarily context-free.

Inverse Homomorphisms

Theorem 17. Let L be a CFL and h a homomorphism. Then $h^{-1}(L)$ is a CFL.

Decision Properties of CFL's

Complexity of Converting Among CFG's and PDA's

Before proceeding to the algorithms for deciding questions about CFL's, let us consider the complexity of converting from one representation to another. The running time of the conversion is a component of the cost of the decision algorithm whenever the language is given in a form other than the one for which the algorithm is designed.

In what follows, we shall let n be the length of the entire representation of a PDA or CFG. Using this parameter as the representation of the size of the grammar or automaton is "coarse," in the sense that some algorithms have a running time that could be described more precisely in terms of more specific parameters, such as the number of variables of a grammar or the sum of the lengths of the stack strings that appear in the transition function of a PDA.

However, the total-length measure is sufficient to distinguish the most important issues: is an algorithm linear in the length (i.e., does it take little more time than it takes to read its input), is it exponential in the length (i.e., you can perform the conversion only for rather small examples), or is it some nonlinear polynomial (i.e., you can run the algorithm, even for large examples, but the time is often quite significant).

There are several conversions we have seen so far that are linear in the size of the input. Since they take linear time, the representation that they produce as output is not only produced quickly, but it is of size comparable to the input size. These conversions are:

1. Converting a CFG to a PDA.
2. Converting a PDA that accepts by final state to a PDA that accepts by empty stack.
3. Converting a PDA that accepts by empty stack to a PDA that accepts by final state.

Theorem 18. There is an $O(n^3)$ algorithm that takes a PDA P whose representation has length n and produces a CFG of length at most $O(n^3)$. This CFG generates the same language as P accepts by empty stack. Optionally, we can cause G to generate the language that P accepts by final state.

Running Time of Conversion to Chomsky Normal Form

As decision algorithms may depend on first putting a CFG into Chomsky Normal Form, we should also look at the running time of the various algorithms that we used to convert an arbitrary grammar to a CNF grammar. Most of the steps preserve, up to a constant factor, the length of the grammar's description; that is, starting with a grammar of length n they produce another grammar of length $O(n)$. The good news is summarized in the following list of observations:

1. Using the proper algorithm, detecting the reachable and generating symbols of a grammar can be done in $O(n)$ time. Eliminating the resulting useless symbols takes $O(n)$ time and does not increase the size of the grammar.
2. Constructing the unit pairs and eliminating unit productions takes $O(n^2)$ time and resulting grammar has length $O(n^2)$.
3. The replacement of terminals by variables in production bodies takes $O(n)$ time and results in a grammar whose length is $O(n)$.
4. The breaking of production bodies of length 3 or more into bodies of length 2 takes $O(n)$ time and results in a grammar of length $O(n)$.

Theorem 19. Give a grammar G of length n, we can find an equivalent Chomsky- Normal-Form grammar for G in time $O(n^2)$; the resulting grammar has length $O(n^2)$

Testing Emptiness of CFL's

Because of the importance of this test, we shall consider in detail how much time it takes to find all the generating symbols of a grammar G. Suppose the length of G is n. Then there could be on the order of n variables, and each pass of the inductive discovery of generating variables could take $O(n)$ time to examine all the productions of G. If only one new generating variable is discovered on each pass, then there could be $O(n)$ passes. Thus, a naive implementation of the generating-symbols test is $O(n^2)$

However, there is a more careful algorithm that sets up a data structure in advance to make our discovery of generating symbols take $O(n)$ time only.

Testing Memberyship in a CFL

We can also decide membership of a string w in a CFL L. There are several inefficient ways to make the test; they take time that is exponential in $|w|$, assuming a grammar or PDA for the language L is given and its size is treated as a constant, independent of w.

There is a much more efficient technique based on the idea of "dynamic programming," which may also be known to you as a "table-filling algorithm" or "tabulation." This algorithm, known as the *CYK Algorithm*, starts with a CNF grammar $G = (V, T, P, S)$ for a language L. The input to the algorithm is a string $w = a_1 a_2 \cdots a_n$ in T^*. In $O(n^3)$ time, the algorithm constructs a table that tells whether w is in L.

In the CYK algorithm we construct a triangular table. The horizontal axis corresponds to the positions of the string $w = a_1 a_2 \cdots a_n$. The table entry X_{ij} is the set of variables A such that $A \overset{*}{\Rightarrow} a_i a_{i+1} \cdots a_j$. Note in particular, that we are interested in whether S is in the set X_{1n}, because that is the same as saying $S \overset{*}{\Rightarrow} w$, i.e., w is in L.

To fill the table, we work row-by-row upwards. Notice that each row corresponds to one length of substrings; the bottom row is for strings of length 1, the second-from-bottom row for strings of length 2, and so on, until the top-row corresponds to the one substring of length n, which is w itself. Since there are $n(n+1)/2$ table entries, the whole table-construction process takes $O(n^3)$ time. Here is the algorithm for computing the X_{ij}'s:

BASIS: We compute the first row as follows. Since the string beginning and ending at position i is just the terminal a_i, and the grammar is in CNF, the only way to derive the string a_i is to use a production of the form $A \rightarrow a_i$. Thus, X_{ii} is the set of variables A such that $A \rightarrow a_i$, is a production of G.

INDUCTION: Suppose we want to compute X_{ij}, which is in row $j-i+1$, and we have computed all the X's in the rows below. That is, we know about all proper prefixes and proper suffixes of that string. As $j - i > 0$ may be assumed, we know that any derivation of $A \overset{*}{\Rightarrow} a_i a_{i+1} \cdots a_j$ must start out with some step $A \Rightarrow BC$. Then, B derives some prefix of $a_i a_{i+1} \cdots a_j$, say $B \overset{*}{\Rightarrow} a_i a_{i+1} \cdots a_k$, for some $k < j$. Also, C must then derive the

remainder of $a_i a_{i+1} \cdots a_j$, that is, $C \overset{*}{\Rightarrow} a_{k+1} a_{k+2} \cdots a_j$.
We conclude that in order for A to be in X_{ij}, we must find variables B and C, and integer k such that:

1. $i \leq k < j$.
2. B is in X_{ik}.
3. C is in $X_{k+1,j}$.
4. $A \to BC$ is a production of G.

Finding such variables A requires us to compare at most n pairs of previously computed sets: $(X_{ii}, X_{i+1,j})$, $(X_{i,i+1}, X_{i+2,j})$, and so on until $(X_{i,j-1}, X_{jj})$.

Theorem 20. The algorithm described above correctly computes X_{ij} for all i and j; thus w is in $L(G)$ if and only if S is in X_{1n}. Moreover, the running time of the algorithm is $O(n^3)$.

Preview of Undecidable CFL Problems

The following are undecidable:

1. Is a given CFG G ambiguous?
2. Is a given CFL inherently ambiguous?
3. Is the intersection of two CFL's empty?
4. Are two CFL's the same?
5. Is a given CFL equal to Σ^*, where Σ is the alphabet of this language?

Summary of Chapter 7

Eliminating Using Symbols A variable can be eliminated from a CFG unless it derives some string of terminals and also appears in at least one string derived from the start symbol. To correctly eliminate such useless symbols, we must first test whether a variables derives a terminal string, and eliminate those that do not, along with all their productions. Only then do we eliminate variables that are not derivable from the start symbol.

Eliminating ϵ- and Unit-productions Given a CFG, we can find another CFG that generates the same language, except for string ϵ, yet has no ϵ-productions (those with body ϵ) or unit productions (those with a single variable as the body).

Chomsky Normal Form Given a CFG that derives at least one nonempty string, we can find another CFG that generates the same language, except for ϵ, and is in Chomsky Normal Form: there are no useless symbols, and every production body consists of either two variables or one terminal.

The Pumping Lemma In any CFL, it is possible to find, in any sufficiently long string of the language, a short substring such that two ends of that substring can be "pumped" in tandem; i.e., each can be repeated any desired number of times. The strings being pumped are not both ϵ This lemma and a more powerful version called Ogden's lemma, allow us to prove many languages not to be context-free.

Operations That Preserve Context-Free Languages The CFL's are closed under substitution, union, concatenation, closure (star), reversal, and inverse homomorphisms. CFL's are not closed under intersection or complementation, but the intersection of a CFL and a regular language is always a CFL.

Testing Emptiness of a CFL Given a CFG, there is an algorithm to tell whether it generates any strings at all. A careful implementation allows this test to be conducted in time that is proportional to the size of the grammar itself.

Testing Membership in a CFL The Cocke-Younger-Kasami algorithm tells whether a given string is in a given context-free language. For a fixed CFL, this test takes time $O(n^3)$, if n is the length of the string being tested.