ECS 120 Lesson 11 – Chomsky Normal Form

Oliver Kreylos

Monday, April 23rd, 2001

Today we are going to look at a special way to write down context-free grammars that will make reasoning about them easier. This special form was introduced by Noam Chomsky himself and is called the *Chomsky Normal Form* (CNF). We will show that for every context-free grammar G, there is an equivalent grammar G' that is in Chomsky Normal Form. The constructive proof for this claim will provide an algorithm to transform G into G'.

1 Definition of Chomsky Normal Form

A context-free grammar $G = (V, \Sigma, R, S)$ is said to be in Chomsky Normal Form (CNF), if and only if every rule in R is of one of the following forms:

1. $A \to a$, for some $A \in V$ and some $a \in \Sigma$
2. $A \to BC$, for some $A \in V$ and $B, C \in V \setminus \{S\}$
3. $S \to \epsilon$

In other words: Every rule either replaces a variable by a single character or by a pair of variables except the start symbol, and the only rule that can have the empty word as its right-hand side must have the start symbol as its left-hand side.

From the above definition it follows, that every parse tree for a grammar in CNF must be a binary tree, and the parse tree for any non-empty word cannot have any leaves labeled with ϵ in it. The use for the Chomsky Normal Form is to make many of the proofs about context-free languages we will encounter later much easier by allowing us to assume that every context-free

grammar we want to reason about is in Chomsky Normal Form. We will first see the usefulness of CNF in the proof for the Context-Free Pumping Lemma.

2 Transforming a Grammar to CNF

In order to construct the grammar G' in CNF that is equivalent to a given grammar G, we first have to identify how exactly G can violate the rules for a CNF. Since the CNF only restricts the rules in G, we have to look only at R. Here are the "bad" cases of rules:

1. $A \to uSv$, where $A \in V$ and $u, v \in (V \cup \Sigma)^*$. The start symbol must not appear on the right-hand side of any rule. We call rules of this type *start symbol rules*.

2. $A \to \epsilon$, where $A \in V \setminus \{S\}$. The only symbol that can be replaced by the empty word is the start symbol. We call rules of this type *ϵ-rules*.

3. $A \to B$, where $A, B \in V$. The only rules involving variables on the right-hand side must have exactly two of them. We call rules of this type *unit rules*.

4. $A \to w$, where $A \in V$, $w \in (V \cup \Sigma)^*$ and w contains at least one character and at least one variable. The only rules where characters appear on the right-hand side must have exactly one character as the right-hand side. We call rules of this type *mixed rules*.

5. $A \to w$, where $A \in V$ and $w \in (V \cup \Sigma)^*$ with $|w| > 2$. Rules must either have one symbol (one character) or two symbols (two variables) as the right-hand side. We call rules of this type *long rules*.

To transform a grammar to CNF, we will have to take care of these five cases of violations. We will fix all these cases in the order presented. The running example for the following sections will be the grammar $G = \big(\{S, A, B\}, \{\mathtt{a}, \mathtt{b}\}, R, S\big)$, where R contains the rules

$$\begin{aligned} S &\to ASA \mid \mathtt{a}B \\ A &\to B \mid S \\ B &\to \mathtt{b} \mid \epsilon \end{aligned}$$

2.1 Start Symbol Rules

To remove the start symbol from the right-hand side of all rules in R, we employ the same trick we used in finite automata, to ensure that no arrows enter the start state: We add a new symbol S_0, make it the start symbol in the new grammar G_1, and add the single rule $S_0 \to S$ to R to get the rules for G_1. Since S_0 does not appear in any rules (it is a new symbol), the new grammar has no start symbol rules.

The language of the new grammar is the same as the language of the old grammar: If $w \in L(G)$ is a word in the language of G, there exists a derivation $S \stackrel{*}{\Rightarrow} w$. We know that $S_0 \to S$ is a rule in G_1; therefore, $S_0 \stackrel{*}{\Rightarrow} w$ is a derivation for w in G_1, meaning $w \in L(G_1)$. Conversely, if $w \in L(G_1)$, there is a derivation $S_0 \stackrel{*}{\Rightarrow} w$. Since the only rule having S_0 as left-hand side in G_1 is the rule $S_0 \to S$, the first step in the derivation of w must have been $S_0 \to S$. Therefore, we can split the derivation into $S_0 \to S, S \stackrel{*}{\Rightarrow} w$, meaning that $w \in L(G)$.

The new grammar for the example grammar G is given by $G_1 = \big(\{S_0, S, A, B\}, \{\mathtt{a}, \mathtt{b}\}, R_1, S_0\big)$, where R_1 contains the rules

$$\begin{array}{rcl} S_0 & \to & S \\ S & \to & ASA \mid \mathtt{a}B \\ A & \to & B \mid S \\ B & \to & \mathtt{b} \mid \epsilon \end{array}$$

2.2 ϵ-Rules

The algorithm to remove ϵ-rules works in two steps. First, we identify all variables that can yield the empty string, either directly or indirectly. We call these variables *nullable*. Second, we remove all direct rules $A \to \epsilon$ from the grammar, and fix up the grammar by removing all occurrences of nullable variables from the right-hand sides of all rules. For example, if there is a rule $A \to \epsilon$, this makes A a nullable variable. Now, if there is another rule $B \to ACA$, each of the occurrences of A in that rule could reduce to the empty word. In other words, the final result of applying that rule could be any one of C, AC, CA or ACA. If we remove the rule $A \to \epsilon$, we have to explicitly add all of those cases to the rules for the symbol B.

2.2.1 Identifying Nullable Variables

We define the set of nullable variables recursively:

Base Case If there is a rule $A \to \epsilon \in R_1$, A is a nullable variable.

Inductive Case If there is a rule $A \to B_1 \dots B_n \in R_1$, where all variables $B_1, \dots, B_n \in V$ are nullable, A is a nullable variable.

2.2.2 Removing ϵ-Rules

Let G_1 be a grammar, and let $N \subset V$ be the set of nullable variables. We build a new set of rules R_2 by classifying each rule in R_1 into one of the following cases:

1. If the rule is $A \to \epsilon$ for some $A \in V$, drop it.
2. If the rule is $A \to w$ for some $A \in V$ and some $w \in (V \cup \Sigma)^* \setminus \{\epsilon\}$, where w does not contain any nullable variables, add it to R_2 unmodified.
3. Otherwise, the rule is $A \to w$, where w contains some nullable variables. We break up w in the following way: $w = w_0 N_1 w_1 N_2 w_2 \dots w_{n-1} N_n w_n$, where the N_i are occurrences of nullable variables, and the $w_i \in (V \cup \Sigma)^*$ do not contain any nullable variables. We then add all rules to R_2 that can be generated from $A \to w$ by removing any combination of occurrences of the N_i from w (there will be 2^n of such new rules).

Finally, if the start symbol S of G_1 is a nullable variable (meaning that $\epsilon \in L(G_1)$), then we add the rule $S \to \epsilon$ to R_2.

For the example grammar G_1, the set of nullable variables is $N = \{A, B\}$. The new grammar is given by $G_2 = \big(\{S_0, S, A, B\}, \{\mathtt{a}, \mathtt{b}\}, R_2, S_0\big)$, where R_2 contains the rules

$$
\begin{aligned}
S_0 &\to S \\
S &\to ASA \mid \mathtt{a}B \mid AS \mid SA \mid S \mid \mathtt{a} \\
A &\to B \mid S \\
B &\to \mathtt{b}
\end{aligned}
$$

2.3 Unit Rules

The algorithm to remove unit rules is similar to the one for removing ϵ-rules. First, we identify a set of *unit pairs*. These are pairs of symbols (A, B), where A yields B: $A \overset{*}{\Rightarrow} B$. We then remove all unit rules by copying right-hand sides: If there is a rule $A \to B$, (A, B) is a unit pair. Then, if there is a rule $B \to w$, we can derive w from A by $A \to B$, $B \to w$. To remove the unit rule and still generate an equivalent grammar, we have to add the right-hand side w to the rules for A directly: $A \to w$. Since there can be cyclical unit rules, e. g., $A \to B$, $B \to C$, $C \to A$, applying this algorithm ad-hoc is tedious, confusing and error-prone. The algorithm using unit pairs is just tedious, which is preferrable.

2.3.1 Identifying Unit Pairs

We define the set of unit pairs recursively. The base case for this is not quite intuitive: One would think that the base case should be all pairs (A, B) where there exists a rule $A \to B$. But we decide to include all pairs (A, A) into the set of unit pairs instead; after all, from the definition of derivation follows that $A \overset{*}{\Rightarrow} A$. Adding these "dummy" unit pairs makes the second part of the algorithm a lot easier. Here is the recursive definition:

Base Case For every variable $A \in V$, (A, A) is a unit pair.

Inductive Case If $(A, B) \in V^2$ is a unit pair, and $B \to C \in R_2$ is a rule for some variable $C \in V$, then (A, C) is a unit pair. From (A, B) being a unit pair, we know that $A \overset{*}{\Rightarrow} B$; adding the fact that $B \to C$ yields $A \overset{*}{\Rightarrow} C$, making (A, C) a unit pair.

2.3.2 Removing Unit Rules

Let G_2 be a grammar, and let $U \subset V^2$ be its set of unit pairs. We build a new set of rules R_3 in the following way: For every unit pair $(A, B) \in U$, and every non-unit rule $B \to w$, add the rule $A \to w$ to R_3. This will copy all the original non-unit rules, since (A, A) is a unit pair for every variable A (the reason why we included them as the base case); and it will also copy all indirect rules of the form $A \to w$, where $A \overset{*}{\Rightarrow} B$ and $B \to w$. Therefore, the new grammar will be equivalent to the old grammar, but it will not have any unit rules.

For the example grammar G_2, the set of unit pairs is $U = \big\{(S_0, S_0), (S, S), (A, A), (B, B), (S_0, S), (A, B)\big\}$. The new grammar is given by $G_3 = \big(\{S_0, S, A, B\}, \{\mathtt{a}, \mathtt{b}\}, R_3, S_0\big)$, where R_3 contains the rules

$$\begin{aligned} S_0 &\to ASA \mid \mathtt{a}B \mid AS \mid SA \mid \mathtt{a} \\ S &\to ASA \mid \mathtt{a}B \mid AS \mid SA \mid \mathtt{a} \\ A &\to \mathtt{b} \mid ASA \mid \mathtt{a}B \mid AS \mid SA \mid \mathtt{a} \\ B &\to \mathtt{b} \end{aligned}$$

2.4 Mixed Rules

The strategy to remove mixed rules is a lot simpler than the last two. Let $A \to w \in R_3$ be a mixed rule. Then we can write w as $w = v_0 c_1 v_1 \dots v_{n-1} c_n v_n$, where the $c_i \in \Sigma$ are occcurrences of characters, and the $v_i \in V^*$ are strings of only variables. Then we will add a new symbol C_i to V_4 for every character c_i, and we will add the rules $C_i \to c_i$ to R_4. Finally, we define $w' := v_0 C_1 v_1 \dots v_{n-1} C_n v_n \in V^*$ and add the rule $A \to w'$ to R_4. This will get rid of the mixed rule (now the right-hand side consists of variables only), and it will not change the language generated by the grammar: If the rule $A \to w$ is part of the derivation for some word, we can replace that single rule by applying the rule $A \to w'$ first, and then replacing all C_i by c_i using their respective rules.

For the example grammar G_3, we only have to add one new symbol U to V. There are multiple mixed rules, but they all share the common right-hand side $\mathtt{a}B$. Thus, adding the single rule $U \to \mathtt{a}$ is sufficient. The new grammar is given by $G_4 = \big(\{S_0, S, A, B, U\}, \{\mathtt{a}, \mathtt{b}\}, R_4, S_0\big)$, where R_4 contains the rules

$$\begin{aligned} S_0 &\to ASA \mid UB \mid AS \mid SA \mid \mathtt{a} \\ S &\to ASA \mid UB \mid AS \mid SA \mid \mathtt{a} \\ A &\to \mathtt{b} \mid ASA \mid UB \mid AS \mid SA \mid \mathtt{a} \\ B &\to \mathtt{b} \\ U &\to \mathtt{a} \end{aligned}$$

2.5 Long Rules

The final step in creating a CNF, removing all long rules, is also easy. Let $A \to B_1 \dots B_n$ be a long rule, i. e., $n > 2$. We already removed all mixed rules,

thus we know that all the B_i are variables. We will break up every single long rule into several "short" rules, by introducing new "helper variables" and splitting the right-hand side from left to right: We add new symbols $A_1, \ldots, A_{n-2}$ to the set of variables, and add the following rules to R_5: $A \to B_1 A_1, A_1 \to B_2 A_2, \ldots, A_{n-2} \to B_{n-1} B_n$.

After this step, all rules in R_5 are either of the form $A \to BC$ or $A \to a$ (except the potential special rule $S \to \epsilon$), and the new grammar still generates the same language. If the rule $A \to B_1 \ldots B_n$ was part of the derivation for some word w,we can derive the same word in the new grammar by using the rules $A \to B_1 A_1, \ldots, A_{n-1} \to B_{n-1} B_n$ in that order. The final string generated by them will be $B_1 \ldots B_n$ as well.

For the example grammar G_4, we only have to add one new symbol A_1 to the set of variables. All long rules in G_4 share the same right-hand side ASA. We therefore add A_1 to V_5, add the rule $A_1 \to SA$ to R_5, and replace each rule $C \to ASA$ (for any variable $C \in V$) by $C \to AA_1$. The new grammar is given by $G_5 = \big(\{S_0, S, A, B, U, A_1\}, \{\mathtt{a}, \mathtt{b}\}, R_5, S_0\big)$, where R_5 contains the rules

$$
\begin{aligned}
S_0 &\to AA_1 \mid UB \mid AS \mid SA \mid \mathtt{a} \\
S &\to AA_1 \mid UB \mid AS \mid SA \mid \mathtt{a} \\
A &\to \mathtt{b} \mid AA_1 \mid UB \mid AS \mid SA \mid \mathtt{a} \\
B &\to \mathtt{b} \\
U &\to \mathtt{a} \\
A_1 &\to SA
\end{aligned}
$$

3 Practicality of CNF

As can be seen from the example grammar G_5, its set of rules is quite a bit larger than the set of rules of the original grammar G. Also, if G was designed from a recursive language definition, its rules will closely model the language's structure, as we have already seen. The CNF will not exhibit this intuitive relation to its language. This means that the conversion to CNF is usually not done when designing a grammar, and it is also not done when it is anticipated that the grammar will be changed at some point in time, for example due to changes in the language. The CNF is mainly used for theoretical reasoning about properties of context-free languages: With the

construction discussed above, we know that every context-free language is described by some grammar in CNF. When having to do a proof about a context-free language, we can thus assume it is given by a CNF grammar, and can exploit its special structure. We will see an example for this in the upcoming proof for the Context-Free Pumping Lemma.

An important question is the size of the resulting CNF grammar. The good news is, that four of the five steps performed do not blow up the size of the grammar beyond reasonable limits. Removing start rules, unit rules, mixed rules and long rules increase the size of the grammar by only a linear factor, e. g., the size might increase by a factor of five. The bad news is, that the removal of ϵ-rules might make the grammar prohibitively large. We recall that for any right-hand side that contains n nullable variables, we have to add 2^n new rules with any combination of nullable variables removed. If n is big, 2^n is huge, which can make conversion to CNF impractical. For theoretical reasoning this is of little concern; but if one wants to apply this algorithm in practice, another way has to be found. One solution exploits the fact that the five steps can be applied in different orders: For example, removing mixed and long rules first, and then start rules and ϵ-rules and unit rules, also yields a grammar in CNF[1]. But since long rules are removed first, every right-hand side when removing ϵ-rules is at most two symbols long; therefore, we have to add at most four new rules. This trick makes the size of the generated grammar manageable.

[1]Not all orders of the steps do! For example, removing ϵ-rules might introduce new unit rules. Thus, the latter have to be removed after ϵ-rules.