CS 273, Lecture 15
Chomsky Normal form

6 March 2008

1 Chomsky Normal Form

Some algorithms require context-free grammars to be in a special format, called a "normal form." One example is ***Chomsky Normal Form*** (CNF). Chomsky Normal Form requires that each rule in the grammar look like:

- $\mathsf{T} \to \mathsf{BC}$, where T, B, C are all variables and neither B nor C is the start symbol,
- $\mathsf{T} \to \mathsf{a}$, where T is a variable and a is a terminal, or
- $\mathsf{S} \to \epsilon$, where S is the start symbol.

Why should we care for CNF? Well, its an effective grammar, in the sense that every variable that being expanded (being a node in a parse tree), is guaranteed to generate a letter in the final string. As such, a word w of length n, must be generated by a parse tree that has $O(n)$ nodes. This is of course not necessarily true with general grammars that might have huge trees, with little strings generated by them.

1.1 Outline of conversion algorithm

All context-free grammars can be converted to CNF. Here is an outline of the procedure:

(i) Create a new start symbol S_0, with new rule $\mathsf{S}_0 \to \mathsf{S}$ mapping it to old start symbol (i.e., S).

(ii) Remove "nullable" variables (i.e., variables that can generate the empty string).

(iii) Remove "unit pairs" (i.e., variables that can generate each other).

(iv) Restructure rules with long righthand sides.

Let's look at an example grammar with start symbol S.

(G0)
$$\begin{array}{|ll|}\hline \Rightarrow & \mathsf{S} \to \mathsf{TST} \mid \mathsf{aB} \\ & \mathsf{T} \to \mathsf{B} \mid \mathsf{S} \\ & \mathsf{B} \to \mathsf{b} \mid \epsilon \\ \hline \end{array}$$

After adding the new start symbol S_0, we get the following grammar.

(G1)

$$\begin{array}{rl} \Rightarrow & \mathsf{S}_0 \to \mathsf{S} \\ & \mathsf{S} \to \mathsf{TST} \mid \mathsf{aB} \\ & \mathsf{T} \to \mathsf{B} \mid \mathsf{S} \\ & \mathsf{B} \to \mathtt{b} \mid \epsilon \end{array}$$

1.2 Removing nullable variables

A ***nullable*** variable is a variable from which we can derive the empty string from it (maybe by a long sequence of derivations). That is, a variable X in a grammar such that $X \stackrel{*}{\Rightarrow} \epsilon$. In the grammar (G1), T and B are both nullable. In CNF, however, only the start symbol is allowed to be nullable. We look at each rule R with a nullable variable on its righthand side. We add all possible variations of R in which the nullable variable(s) have been deleted; that is, we replace a nullable variable by the empty string. Of course, we never add back a rule with empty righthand side.

Then, we remove all rules mapping a variable to ϵ (except for the starting symbol).

For example, in the above grammar, we have $\mathsf{S} \to \mathsf{TST}$. Since T is nullable, we need to add $\mathsf{S} \to \mathsf{ST}$ and $\mathsf{S} \to \mathsf{TS}$ and $\mathsf{S} \to \mathsf{S}$ (which is of course a silly rule, so we will not waste our time putting it in). We also have $\mathsf{S} \to \mathsf{aB}$. Since B is nullable, we need to add $\mathsf{S} \to \mathsf{a}$. The output grammar is then:

(G2)

$$\begin{array}{rl} \Rightarrow & \mathsf{S}_0 \to \mathsf{S} \\ & \mathsf{S} \to \mathsf{TST} \mid \mathsf{aB} \mid \mathsf{a} \mid \mathsf{ST} \mid \mathsf{TS} \\ & \mathsf{T} \to \mathsf{B} \mid \mathsf{S} \\ & \mathsf{B} \to \mathtt{b} \end{array}$$

1.3 Removing unit pairs

A ***unit rule*** has on its righthand side only a single variable. We need to remove all the unit rules in the grammar (because CNF does not allow such rules).

To do this, define a ***unit pair*** to be a pair of variables (T, B) such that $\mathsf{T} \stackrel{*}{\Rightarrow} \mathsf{B}$. In our grammar, the unit pairs are $(\mathsf{S}_0, \mathsf{S})$, (T, B), (T, S). If the two variables are the same, as in (S, S), we can simply remove the offending unit rule (which we already avoided, by removing $\mathsf{S} \to \mathsf{S}$).

Otherwise, for each unit pair (T, Y) in the grammar, we find all the rules $\mathsf{Y} \to \alpha$ and add the rule $\mathsf{T} \to \alpha$ (here α can be an arbitrary string of variables and terminals). This way, the unit rule is no longer necessary.

In our example, this means copying the productions for S up to S_0, copying the productions for S down to T, and copying the production $\mathsf{B} \to \mathtt{b}$ to $\mathsf{T} \to \mathtt{b}$.

(G3)

$$\begin{array}{rl} \Rightarrow & \mathsf{S}_0 \to \mathsf{TST} \mid \mathsf{aB} \mid \mathsf{a} \mid \mathsf{ST} \mid \mathsf{TS} \\ & \mathsf{S} \to \mathsf{TST} \mid \mathsf{aB} \mid \mathsf{a} \mid \mathsf{ST} \mid \mathsf{TS} \\ & \mathsf{T} \to \mathtt{b} \mid \mathsf{TST} \mid \mathsf{aB} \mid \mathsf{a} \mid \mathsf{ST} \mid \mathsf{TS} \\ & \mathsf{B} \to \mathtt{b} \end{array}$$

1.4 Final restructuring

Now, we can directly patch any places where our grammar rules have the wrong form for CNF. First, if the rule has at least two symbols on its righthand side but some of them are terminals, we introduce new variables which expand into these terminals. For our example, the offending rules are $S_0 \to aB$, $S \to aB$, and $T \to aB$. We can fix these by replacing the a's with a new variable U, and adding a rule $U \to a$.

(G4)

$$
\begin{array}{rl}
\Rightarrow & S_0 \to TST \mid UB \mid a \mid ST \mid TS \\
& S \to TST \mid UB \mid a \mid ST \mid TS \\
& T \to b \mid TST \mid UB \mid a \mid ST \mid TS \\
& B \to b \\
& U \to a
\end{array}
$$

Then, if any rules have more than two variables on their righthand side, we fix that with more new variables. For the grammar (G4), the offending rules are $S_0 \to TST$, $S \to TST$, and $T \to TST$. We can rewrite these using a new variable Z and a rule $Z \to ST$. This gives us the CNF grammar

(G5)

$$
\begin{array}{rl}
\Rightarrow & S_0 \to TZ \mid UB \mid a \mid ST \mid TS \\
& S \to TZ \mid UB \mid a \mid ST \mid TS \\
& T \to b \mid TZ \mid UB \mid a \mid ST \mid TS \\
& B \to b \\
& U \to a \\
& Z \to ST
\end{array}
$$

Theorem 1.1 *Any context-free grammar can be converted into Chomsky normal form.*

In the following, we will need the following easy observation.

Observation 1.2 *Consider a grammar G which is CNF, and a variable X of G which is not the start variable. Then, any string derived from X must be of length at least one.*

2 CNF have compact parsing trees

In this section, we prove that CNF give very compact parsing trees for strings in the language of the grammar. The proof the following claim (stating this fact) is an example of grammar-based induction.

Claim 2.1 *if G is a context-free grammar in Chomsky normal form, and w is a string of length $n \geq 1$, then any leftmost derivation of w from any variable X contains exactly $2n - 1$ steps.*

This claim is actually true for all derivations, not just leftmost ones. But the proof is easier to follow for leftmost derivations. We remind the reader that a ***leftmost derivation*** if at every step the leftmost remaining variable is the one replaced.

Proof: Proof by induction on the length of w. Suppose that the start symbol of G is S.

Base of induction: If $|w| = 1$, then w contains only a single character c. So any derivation of w from a variable X must contain exactly one step, using the rule $\mathsf{X} \to c$.

Induction: Suppose the claim is true for all strings of length $< k$, and let w be a string of length k and suppose we have a leftmost derivation for w starting with some variable X.

Consider the first step in this derivation. If the first step uses a rule of the form $\mathsf{X} \to c$, then w contains only a single character and we are back in the base case. So, let us consider the only other possibility: the first step uses a rule of the form $\mathsf{X} \to \mathsf{TB}$.

Then we can split up the derivation as follows: the first step expands X into TB. The next few steps expand T into some (non-empty) string x. The final few steps expand B into some string y. Suppose that $|x| = i$ and $|y| = j$. Since $w = xy$, we have that $i + j = k$.

Because the grammar is in Chomsky normal form, neither T nor B is the (glorious) start symbol. So neither T nor B can expand into the empty string. So x and y both contain some characters; that is, $i > 0$ and $j > 0$. This means that $0 < |x| = i < i + j = k$ and $0 < |y| = j < i + j = k$. Namely, $|x| < k$ and $|y| < k$. So we can apply our inductive hypothesis to x and y.

That is, since $|x| = i$, the derivation $\mathsf{T} \stackrel{*}{\Rightarrow} x$ must take exactly $2i - 1$ steps. Similarly, the derivation $\mathsf{B} \stackrel{*}{\Rightarrow} y$ must take exactly $2j - 1$ steps. But then the whole derivation of w from X takes $1 + (2i - 1) + (2j - 1) = 2(i + j) - 1 = 2k - 1$ steps. ■

Some things to notice about this proof:

- We use "strong" induction. That is, we assume the statement is true for all values $< k$. We need this because the inductive step divides the string into two parts whose lengths we have limited control over.

- We remove the **first** step in the derivation. Not, for example, the final step. In general, results about grammars require removing the first step of a derivation or the top node in a parse tree.

- It almost never works right to start with a derivation or tree of size k and try to expand it into a derivation or parse tree of size $k + 1$. Please avoid this common mistake.

  As an example of why inherent hopelessness of arguing in this wrong direction, consider the following grammar:

(G6)

$$\begin{array}{|lll|}\hline \Rightarrow & \mathsf{S} \to \mathsf{T} \mid \mathsf{B} \\ & \mathsf{T} \to \mathtt{aa}\mathsf{T} \mid \epsilon \\ & \mathsf{B} \to \mathtt{bb}B \mid \mathtt{b} \\ \hline \end{array}$$

  All the even-length strings are generated from T and contain all $\mathtt{a}$'s. We claim that all the odd-length strings are generated from B and contain all $\mathtt{b}$'s. As such, there is no way to prove this claim by induction, by taking the parse tree for an even-length string and graft on a couple nodes to make a parse tree for the next longer length string.

3 Some notes on normal forms

Context-free grammars can be put into a variety of different normal forms. Another well-known normal form is Greibach Normal Form, in which every rule has to have the form $A \to \mathtt{a}w$, where $\mathtt{a}$ is a terminal and w is a string of zero or more variables. In this normal form, every string of length n has a derivation with exactly n steps. The algorithm for converting to Greibach normal form is complex.

Chomsky and Greibach normal form were named after two researchers who worked on automata theory back in the Dawn of Time (i.e. the 1960's): Noam Chomsky and Sheila Greibach. Noam Chomsky is better known for his political writings and his work in linguistics. Early work on formal linguistics was closely connected to work in automata theory, though the research communities have since diverged.

Also notice that the precise definition of Chomsky normal form varies, e.g. some authors don't allow any ϵ rules, so that the normal form only applies to languages without the empty string.

4 Algorithm issues

For small grammars, such as the one above, you can quickly find the nullable variables and unit pairs by inspection. To implement a conversion program, especially for use on large grammars, we need an explicit algorithm for finding them. It's also helpful to have algorithms that eliminate "useless" rules and variables.

4.1 Finding nullable variables and unit pairs

Nullable variables can be found using an iterative tracing algorithm. We build a table that shows, for each variable, whether we have established that it is nullable. First we mark as nullable any variable A for which there is a rule $A \to \epsilon$. Then repeatedly examine all the grammar rules. For each rule $B \to w$, mark B as nullable if everything in w is already marked as nullable. Halt when you've done a complete pass through the set of rules without any new nullable markings.

A very similar algorithm can be used to find all the unit pairs. Start with all pairs (A, A). Then, repeatedly examing all the grammar rules. For each rule $B \to C$ where (C, D) is marked as a unit pair, mark (B, D) as a unit pair. Again, halt when you've been through the whole grammar and no new pairs have been added.

4.2 Removing useless parts of the grammar

Similar tracing algorithms can also be used to remove variables that are useless, i.e. cannot contribute to a derivation of a terminal string from the start symbol. Useless variables tend not to occur in hand-built grammars, but they can be frequent in grammars produced by algorithms such as the conversion of a PDA to a context-free grammar.

A variable can be useless in two ways

- It can't generate a string of terminals.

- It can't be reached from the start symbol.

An automatic cleanup algorithm removes the non-generating variables first (along with all grammar rules that use them). Then it removes the non-reachable variables (again, along with all rules involving them). Exercise for the reader: what can go wrong if we do these two steps in the other order?

To find the generating symbols, we start at the terminals and trace backwards. A variable B is generating if there is a rule $B \to w$ in which everything in w has been marked as generating.

To find the reachable symbols, we start at the start symbol and trace forwards along the rules. That is, if $B \to w$ is a rule and B is already marked as reachable, then everything in w must be reachable.