Takeaway Notes: Normal Forms and Pumping Lemma for Context-Free Languages

Contents

1 Introduction

In this set of takeaway notes, we look at two ideas, one new and one familiar: *normal forms* and the *pumping lemma* for CFLs. Normal forms, in particular Chomsky normal form (CNF), helps "standardize" a grammar (without loss of equivalency to

the language it generates) in order to reason about it more easily. The pumping lemma for CFLs performs the same function as the pumping lemma for regular languages: it helps us prove whether a language is context-free.

2 Formal Definitions

2.1 Chomsky Normal Form

The intuition behind Chomsky normal form is straightforward: place a grammar in a particular "format" such that it is easier to work with and to prove things about. For a normal form to work (CNF isn't the only one), we must ensure that a normalized grammar still generates the same language as the original grammar. If we can show this, we also assure that every context-free language has a grammar that is in Chomsky normal form.

A grammar (V, T, P, S) is in Chomsky normal form if *all* of its production rules are in one of these formats:

- $A \to BC$, where $A, B, C \in V$
- $A \to a$, where $A \in V$ and $a \in T$

That's it. Note that such a grammar cannot generate ϵ. However it can generate anything else. Thus, for an arbitrary context-free language L, we say that $L - \{\epsilon\}$ can be expressed as a CNF-compliant grammar.

The definition is straightforward enough; the trick is in *converting* any given grammar into CNF. That requires a few additional concepts.

2.2 Converting to CNF

To make a CFL's grammar "Chomsky-ready," we must first perform three steps:

1. Eliminate ϵ *productions*: productions of the form $A \to \epsilon$
2. Eliminate *unit productions*: productions of the form $A \to B$
3. Eliminate *useless symbols*: symbols that cannot be derived from the start symbol

Once a grammar has eliminated these, we may still have production rules that do not meet the two conditions for CNF. The final step is then the conversion of those rules to one of the two forms.

The aforementioned steps are recommended in the given order to minimize repeat passes in the grammar, because the first two steps might generate more useless symbols that would not have been caught had useless symbols been removed first.

2.2.1 ϵ-Productions

ϵ-productions ($A \to \epsilon$) are easy to spot but a little trickier to eliminate. The main idea is that since the variable symbol involved can potentially go to nothing, then *everywhere that it appears* we can substitute nothing. This creates new production rules that are effectively the same as if we had "pre-performed" the ϵ-production in advance.

2.2.2 Unit Productions

Unit productions are perhaps the trickiest ones to remove, but the idea is similar to eliminating ϵ-productions: "pre-perform" these productions to produce new productions that skip the unit substitutions. The general approach for this (particularly because it's possible to have a cycle of unit productions) is to identify *unit pairs* in a CFG—all pairs of symbols (A, B) such that A can generate B completely via unit productions—and to include rules that go from the first symbol of the pair to the non-unit productions that come from the second.

2.2.3 Useless Symbols

A symbol is *useful* if it is both *generating* and *reachable*. Thus, it is *useless* if it isn't one of those.

- A symbol $X \in V \cup T$ is *generating* if $X \overset{*}{\to} w$ for some terminal string w.
- A symbol $X \in V \cup T$ is *reachable* if $S \overset{*}{\to} \alpha X \beta$ for some $\alpha, \beta \in V \cup T$

Productions with useless symbols are simply taken away from the grammar. To minimize repeated "passes" over the grammar, productions with non-generating symbols should be taken away first, then productions with non-reachable ones. For example, with the grammar:

$$S \to AB \mid aA \to b$$

If we try to remove non-reachable symbols first, then on that pass we remove nothing because all symbols are reachable. Removing non-generating symbols will then eliminate $S \to AB$ but than then makes A unreachable, requiring a repeat back to the non-reachable cleanup step.

2.2.4 Conforming to the Two Formats

Once the three cleanup steps are done, we may still have production rules consisting of more than two variables, more than one terminal, or a mix of terminals and variables. This is where the *actual* conversion to CNF takes place. In general, this is done by inventing new variable symbols to represent substrings of the existing productions, thus "clustering" the productions into just pairs of variables or single terminals.

2.2.5 Proofs About Each Step

The validity of CNF and the process of converting to CNF must be proven at every step. Thus, behind the scenes of each procedure and algorithm mentioned here is a proof that said procedure or algorithm does not change the language of the grammar that we are manipulating. These are elided in the takeaway notes but are part of any full text on the theory of computation.

2.3 The Pumping Lemma for CFLs

CFLs have a pumping lemma that is analogous to the pumping lemma for regular languages, and it serves a similar purpose: it allows us to prove, via contradiction, that a given language is not context-free: given L is a CFL, there exists a constant n such that if $z \in L$ and $|z| \geq n$, then z is of the form $uvwxy$ under the following conditions:

- $|vwx| \leq n$
- $vx \neq \epsilon$ (i.e., at least one of v or x may not be empty)
- $\forall\ i \geq 0, uv^iwx^iy \in L$

2.3.1 Intuition Behind the Lemma's Proof

The proof of the lemma assumes that we have a CNF grammar for L, which is why we needed to establish CNF first before going into the lemma. Because we know that any context-free grammar can be rewritten in CNF, this assumption is valid and we can proceed. We will only discuss the core insight behind the proof here; details are elided and left to full texts on the theory of computation.

As with the core insight behind the regular language pumping lemma, we rely on the pigeonhole principle. Recall that for the regular language version, we observe

that a finite-state automaton with n states that accepts a string whose length is greater than n must have repeated at least one state on the way to accepting that string. If it repeats such a state, then it must be able to repeat that state over and over again, as long as the string repeats the same sequence of characters that got it to loop back to that repeated state.

For the CFL pumping lemma, we instead pigeonhole *the number of variable symbols* in the CNF grammar. Let's call this number of variables m; i.e., $m = |V|$ according to our grammar tuple notation. We now pigeonhole m in this way: because we have assumed that the grammar is in Chomsky normal form, then its parse trees are binary at most. Thus, to yield a string of length 2^m, the parse tree must have at least $m + 1$ levels. And if there are $m + 1$ levels, then at least one of the m variable symbols must have been repeated. If we choose the same set of productions that got us "back" to the repeated variable, we would repeat the same two strings on the left and right of that variable's production rule (remember that in CNF, the only rule we accept that has variables is of the form $A \to BC$). We can then choose that sequence of productions over and over again—and that's where we get our "pump."

Thus, the n of the CFL pumping lemma is 2^m where $m = |V|$. With than n, we *must* be repeating a variable in our parse tree, and because we are repeating that variable, we can repeat it indefinitely as long as we choose the same set of productions for its left and right yields. If *no* pair of these repeatable strings (that keeps a string in the language) can be found for some string $z \in L$ where $|z| \geq n$, then the parse tree cannot exist for z and thus the language cannot be context-free.

2.3.2 Sample Use of the Lemma

Let's show how the CFL pumping lemma can be used to show that $L = 0^k 1^k 2^k$ is not context-free. First, we pick $z = 0^n 1^n 2^n$ where n is the n of the pumping lemma.

By the pumping lemma, z must be of the form $uvwxy$ where the conditions of the lemma are met. Thus, we can assert that $|vwx| \leq n$.

Due to that maximum length, there are only two cases for vwx:

- vwx is completely contained within a single symbol (0, 1, or 2)
- vwx crosses two symbols (01 or 12)

In the case where vwx is completely contained within a symbol, either v or x must be of the form a^i where $a \in \{0, 1, 2\}$. In that case, pumping either v or x will produce a string that is not in L.

In the case where vwx straddles two symbols (crossing either $0^n 1^n$ or $1^n 2^n$), either v or x is of the form a^i where a is either in $\{0, 1\}$ or $\{1, 2\}$. In that case, pumping

either v or x will produce a string that is not in L. (there is also a trivial subcase here, where v or x is of the form $a^i b^j$ where a and b are the symbols being straddled—note again that pumping that string will break it out of $L = 0^k 1^k 2^k$)

3 Related Ideas

We use the CFL pumping lemma as the jumping-off point to go to the next highest category of languages, where like the jump from regular to context-free, we require a new computational model and now study the characteristics of languages in that level. This will lead us to the most advanced machine we can define—which therefore makes it the very model that we now (currently?) have for all of computation.

4 Important Skills

For these takeaways, we want to be able to do the following:

- Perform the necessary cleanup steps that prepare any context-free grammar for conversion into Chomsky normal form
- Use the CFL pumping lemma to show that a language is not context-free

5 Big Picture Points

Things you shouldn't forget even years after taking this course:

- All context-free grammars can be converted into Chomsky normal form (CNF), which is a context-free grammar whose parse trees are all binary.
- Thanks to CNF, we can derive a pumping lemma for context-free languages, whose key insight is that for strings of a certain length, their parse trees must have repeated variables along a given path, and if that variable has been repeated, then we should be able to repeat those productions indefinitely.