CS 4510 Automata and Complexity Jan 29th 2024

Lecture 5: Context-Free Grammars

Lecturer: Abrahim Ladha *Scribe(s): Rishabh Singhal*

1 Background

Automata So far we have only looked at automata. These are usually tasked with **Decision or Recognition**. It's a fairly mechanical model, a decision procedure. You look at the input scanning left to right and do something. This corresponds well to your previous intuition on programming. Let D be an automata.

- Given $w \in \Sigma^*$, is $w \in L(D)$? This is not that hard, you simply run the automata on the input.
- Characterizing all of $L(D)$? This can be much harder for an automata. If I give you a DFA or NFA and ask you to describe exactly the strings it accepts and rejects, this is not as easy.

Grammars In contrast, a grammar is tasked with **Production or Generation**. A grammar will non-deterministically produce only the correct strings, like a flower blooming. It does not go left to right, but from inside out. It doesn't start with an input to look at, it starts with nothing. Defined with the rules we give it, it will produce a string according to those rules.

- Given $w \in \Sigma^*$, is $w \in L(G)$? This is surprisingly non-trivial
- Characterizing all of $L(G)$? This is surprisingly easier.

2 Formal Definition of Context-Free Grammar

We represent a context-free grammar (CFG) as a four tuple (V, Σ, R, S) such that:

V Non-Terminals or Variables. These are always capitalized like $\{S, A, B, ...\}$

Σ Terminals or our alphabet. These are always lower-case like $\{a, b, c, ...\}$

R Productions or Rules. Each are of them will be of form $V \to (V \cup \Sigma)^*$. The left-hand side of the production will always be a single non-terminal and the right-hand side will be a string of terminals and non-terminals.

$S \in V$ is our designated start non-terminal.

2.1 Computation

How does a grammar produce a string? For $A \in V, w \in (V \cup \Sigma)^*$, with production of the form $A \to w$, we apply a production as a substring replacement of a "working string" like $xAz \implies xwz$, for $x, z \in (V \cup \Sigma)^*$. When we write $w_i \implies w_{i+1}$ we mean that w_i "yields" w_{i+1} after application of one production. If $S \implies w_1 \implies w_2 \implies w_3 ... \implies w$ with $w \in \Sigma^*$ we say that $w \in L(G)$ and may write $S \overset{*}{\implies} w$. We stop applying productions only when there are no more non-terminals in the working string.

For a context-free grammar G, we characterize the set of strings in $L(G)$ as those and only those produced non-deterministically starting from S. Observe that a production halts when there are no more non-terminals in the working string. We say that a language L is context-free if there exists a context-free grammar G such that $L = L(G)$. We call these CFLs. Note that a CFG takes no input, and only produces exactly and only the correct strings. This is why we say a grammar produces a string, but an automata decides or accepts a string.

3 Examples

Like a state diagram, you can give all parts of the CFG by just giving the set of productions. It implicitly gives the terminals and non-terminals, and we always denote S as the start non-terminal.

3.1 $\{a^n b^n \mid n \in \mathbb{N}\}$

We write $\{S \to aSb, S \to \varepsilon\}$ or just $\{S \to aSb \mid \varepsilon\}$. If we have two or more productions with the same beginning non-terminal, we may use "|" as a shorthand to "or" those productions together. This grammar still has two distinct productions. Let us say we want to produce a^3b^3 the process we follow is

$$S \implies aSb \implies a(aSb)b \implies aaSbb \implies aaaSbbb \implies aaabbb \implies a^3b^3$$

We repeatedly apply the first production, and terminate when we have no more non-terminals in our working string. This occurs when we apply the second rule, $S \to \varepsilon$. Notice that it has to produce exactly the strings of the form $a^n b^n$. Also notice how the nondeterminism decides what string is produced is determined by the order of the sequence of rules you apply. This was our canonical example of a non-regular language, the first one we used for pumping. This should convince you atleast, that the languages produced by context-free grammars, $\mathscr{L}(CFG)$, is not equal to the regular languages. Later we will show it is a strict super set.

3.2 $\{ww^R \mid w \in \Sigma^*\}$

Our productions are similar. $\{S \to aSa \mid bSb \mid \varepsilon\}$. This generates even length palindromes. As we apply productions, the left and right of our primary recursive production effectively act like write-only two stacks, mirrors of each other. This generates the string which is a palindrome and these strings are also even in length. We can conserve the same idea, to generate palindrome of odd length.

3.3 $\{w\Sigma w^R \mid w \in \Sigma^*\}$

We write this as $\{S \to aSa \mid bSb \mid a \mid b\}$. We may combine ideas from the previous two examples to show the set of all palindromes is a context-free language, with the grammar $\{S \to aSa \mid bSb \mid a \mid b \mid \varepsilon\}$. We pumped a third language, $\{ww \mid w \in \Sigma^*\}$. As some foreshadowing, this language is not regular, but it is also not context free.

3.4 Σ^*

There exist many equivalent grammars for this language. These may include

- $S \to aSa \mid bSb \mid aSb \mid bSa \mid a \mid b \mid \varepsilon$
- $S \to aaS \mid abS \mid baS \mid bbS \mid a \mid b \mid \varepsilon$
- $S \to aS \mid bS \mid \varepsilon$

3.5 1^*

This one is easy, $\{S \to 1S \mid \varepsilon\}$

3.6 $\emptyset$

If a grammar produces no strings, not even ε, it is either trivial, or some how does not have a halting condition. There are a few you could come up with, but a non-trivial grammar for this would be $\{S \to A, A \to S\}$. No production of this terminates with a string of only terminals, so it produces no strings. Similarly $\{S \to 1S\}$ produces no strings. Its only production has no termination condition, and a word is only produced after it has no more non-terminals.

3.7 Dyck Language

Consider the grammar $\{S \to (S) \mid SS \mid \varepsilon\}$. This language is the set of balanced, or matching paranthesis. It has a special name, called the Dyck language.

If you wanted to prove the Dyck langauge was non-regular, you could pump it with a choice of string $s = (^p)^p$. We can prove it is not regular by closure. Assume to the contrary $L(G)$ was regular. Then by closure, so must be $L(G) \cap (^*)^*$. The left side enforces that the number of opens equals the number of closes, and the right hand side enforces that all the opens come before all the closes. The intersection is the logical and of these, so we see this intersection must be equal to $\{(^n)^n \mid n \in \mathbb{N}\}$, our canonical non-regular language, a contradiction. Therefore, the Dyck language is not regular.

3.8 Arithmetic Expressions

Consider the following grammar:

$$\begin{aligned} S &\to S + T \mid T \\ T &\to T \times F \mid F \\ F &\to (S) \quad \mid a \end{aligned}$$

with $V = \{S, T, F\}, \Sigma = \{(,), \times, +, a\}$. Lets do an example of a long production to show this grammar generates $(a + a) \times a$

$$\begin{aligned} S \implies T \implies T \times F \implies F \times F \implies (S) \times F \implies \\ (S) \times a \implies (S + T) \times a \implies (T + T) \times a \implies (F + T) \times a \implies \\ (F + F) \times a \implies (F + a) \times a \implies (a + a) \times a \end{aligned}$$

3.9 Propositional Calculus

Many programming languages are parsed via context-free grammars. We give a CFG for the well formed formulas of the propositional calculus. Let our alphabet be $\Sigma = \{(,), \vee, \wedge, \neg, 1\}$ where 1^n is a representation of the nth propositional variable. Our productions are then $S \to (S \vee S) \mid (S \wedge S) \mid \neg(S) \mid N$ and $N \to 1N \mid 1$. Recall that the logical and, or, and negation are complete for propositional logic so if we wanted to express $p \implies q$, instead we may do $(\neg 1 \vee 11)$. We also have more paranthesis than necessary, but notice that they are cheap, and can only help with ambiguity. This way we do not need a PEMDAS style operator precedence rule to parse a proposition.

The previous two examples show how useful CFGs may be for programming languages. If you can define a CFG for your programming language, you can then check if a program has a syntax error if you can check that grammar doesn't produce it. The previous example will never produce a string like $)11(1)\neg\neg(\wedge 1)$.

3.10 $\{w\#x \mid x$ contains w^R as a substring $\}$

If x contains w^R as a substring, then $x = \Sigma^* w^R \Sigma^*$, so $w\#x = w\#(\Sigma^* w^R \Sigma^*) = (w(\#\Sigma^*)w^R)\Sigma^*$. We first will nondeterministically produce and match w with w^R, then we will produce the rest of x. $S \to XY, Y \to aY \mid bY \mid \varepsilon, X \to aXa \mid bXb \mid \#Y$. This is a cool grammar, as it shows the power of nondeterminism. You may have had to create some previous non-trivial deterministic algorithms in order to find the longest palindromic substring or something. You were looking for a needle in a haystack. Here through the power of nondeterminism, we can come at the problem from a different direction. First place the needle, then nondeterministically build all possible haystacks around it.

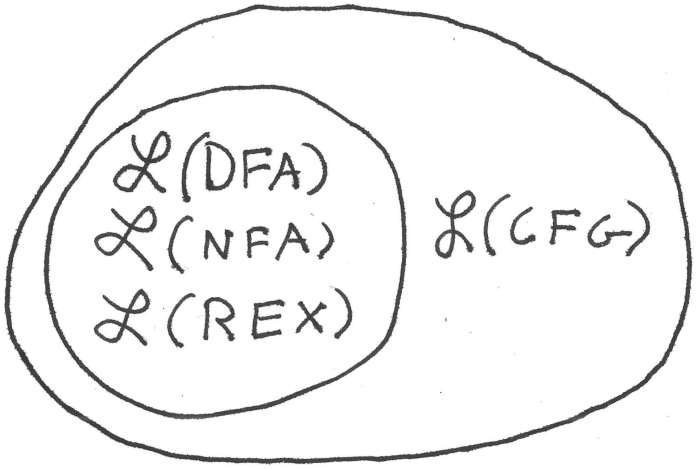
3.11 One last example

Consider $\{a^n b a^m b^{n+m} \mid n, m \in \mathbb{N}\}$. First notice that for some n, m that $a^n b a^m b^{n+m} = a^n b a^m b^n b^m$. We have matching blocks of the same size, but we can't pair them up as written. We notice that letters of the same kind obviously commute, so we see $a^n b a^m b^n b^m =$

$a^n b a^m b^m b^n = a^n (b a^m b^m) b^n$. This gives us the intuition on how we would build our grammar as $\{S \to aSb \mid bR, R \to aRb \mid \varepsilon\}$. Just to work out some productions, they may look like

$$S \overset{*}{\Longrightarrow} a^n S b^n \overset{*}{\Longrightarrow} a^n b R b^n \overset{*}{\Longrightarrow} a^n b a^m R b^m b^n \overset{*}{\Longrightarrow} a^n b a^m b^m b^n = a^n b a^m b^{m+n}$$

4 Relationship with Regular Languages



Every regular language is context-free, but not every context-free language is regular. We can prove the containment in two ways.

4.1 By Closure

We prove that every regular language is also context free. Let $\mathscr{L}(CFG)$ be the languages produced by context-free grammars. We prove $\mathscr{L}(REX) \subsetneq \mathscr{L}(CFG)$ by induction. Note that the containment is strict because we know that $\{a^n b^n \mid n \in \mathbb{N}\}$ cannot be regular by the pumping lemma, but is context-free.

First we prove the base case. We give context-free grammars for $\emptyset, \varepsilon, a, b$

$\emptyset$ $S \to S$

ε $S \to \varepsilon$

a $S \to a$

b $S \to b$

Let G_1, G_2 be two CFGs to produce L_1 and L_2 with start non-terminals S_1, S_2 respectively. We prove that the context-free grammars are closed under union, concatenation, and star.

$L_1 \cup L_2$ Copy all productions, add new start state S, and a new production $S \to S_1 \mid S_2$

$L_1 L_2$ Similarly, with new production $S \to S_1 S_2$

L_1^* Similarly add new productions $S \to S_1 S \mid \varepsilon$

Through a similar process to converting a regular expression into an NFA, you may apply this proof to convert a regular expression into a context-free grammar, thus concluding the proof that every regular language is also context-free. Later we will show CFLs are not closed under intersection or complement. This may be intuitive, if you observe the behavior of a CFG. It only knows how to grow correct strings. Given a grammar which produces only the right strings, it gives no idea on how to create a grammar to only produce the wrong ones.

4.2 Regular Grammars

We say a grammar is right-regular if it only has productions of the form $A \to aB$ or $A \to a$ or $A \to \varepsilon$, where A, B are any non-terminals, and a is any terminal. Certainly every right-regular grammar is also context-free, we claim that the right-regular grammars decide exactly the regular languages. The proof of this characterization is not complicated, but tedious[1]. Instead we will highlight just the part of given a DFA, how one might construct a right-regular grammar. For a DFA of the form $(Q, \Sigma, q_0, \delta, F)$ we construct a grammar (V, Σ, R, S).

- For $Q = \{q_0, ..., q_k\}$ we have non-terminals $V = \{Q_0, ..., Q_k\}$
- The set of terminals for our grammar is identical to the alphabet for our DFA: $\Sigma = \Sigma$
- For q_0 the start state of our DFA, we designate our start non-terminal as Q_0
- For every transition of the form $\delta(q_i, a) = q_j$, we add production $Q_i \to aQ_j$
- For every $q_f \in Q$, we add production $Q_f \to \varepsilon$

Convince yourself of its correctness. Since every regular grammar is a context-free grammar, and there are context-free non-regular languages, this should convince you that we are working with a strictly more powerful computational model, $\mathscr{L}(DFA) \subsetneq \mathscr{L}(CFG)$.

[1] I have a more detailed proof here https://ladha.me/files/sectionX/regulargrammars.pdf