Lecture 5: Context-Free Languages

Ryan Bernstein

April 14, 2016

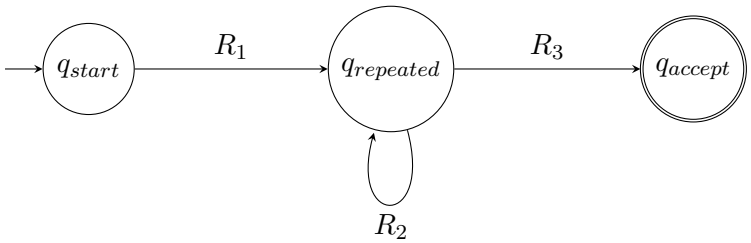
1 Introductory Remarks

- Again, in case you haven't heard about it, Assignment 1 is up and due next Tuesday.

1.1 Recapitulation

On Tuesday, we looked at what made a language non-regular. We started by exploring the limits of regular languages. Notably, the computational context of a finite automaton — one of which exists to decide *every* regular language — contains only a single state or set of states from the finite set Q.

This means that there are a finite number of possible computational contexts. While finite automata *can* decide infinite languages, sufficiently long strings require them to repeat some sequence of one or more states over and over again. Since the states themselves are the only means we have of tracking our computation, every repetition of this sequence is *exactly identical*. We drew a GNFA with a repeated state like this:



We further observed that since we have paths from q_{repeat} 1) to q_{accept} and 2) back to q_{repeat}, we can execute this loop as many or as few times as we see fit. This is the basis for the Pumping Lemma, which gives us a strategy with which to prove the nonregularity of some language.

In more abstract terms, we characterized regular languages by their lack of the capacity to count infinitely. This is the thread that we'll follow today to transition into our next class of languages.

1.2 Lecture 4.5: The Pumping Lemma Revisited

2 Counting with Pushdown Automata

We've said that regular languages are limited by their inability to count infinitely. Since the entire context of an NFA or DFA is limited to a single state (or set of states) from some finite set, we have a limited number of possible machine contexts to choose from, and therefore, the Pumping Lemma tells us that sufficiently long strings must see the same context twice.

As an aside, the word "context" may seem somewhat awkward, but we're using it to attempt to avoid overloading the term *state*. All of these machines have some finite set Q of control states. All we're denoting with the term "context" is the state (or set of states, in an NFA) that the machine is in at any given point during computation. This becomes especially important now, as we add to the number of possible contexts.

We address the issue of infinite counting with a new type of automaton, called a *pushdown automaton*. Like finite automata, pushdown automata have some finite set of states. We now allow for the possibility of an infinite number of possible contexts by adding to the machine an unbounded stack.

This stack works exactly like the stacks you looked at in classes like CS 163. It obeys *last in, first out* semantics. On any transition in our machine, we now gain the ability to either 1) *push* some new element onto the stack, or 2) *pop* the last element off of the top.

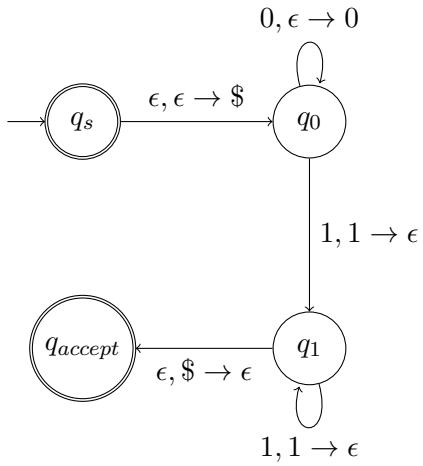
When writing stacks in other classes, you probably added a function called `isEmpty()` or similar to determine when you reached the bottom of your stack. Here, we have no such function. Pushing and popping are the *only* means that we have of interacting with the stack. To accommodate this, pushdown automata often begin with an ϵ-transition that pushes some sentinel character onto the stack. If at any point we pop that same sentinel value back off of the stack, we know that our stack is empty.

2.1 Drawing Pushdown Automata

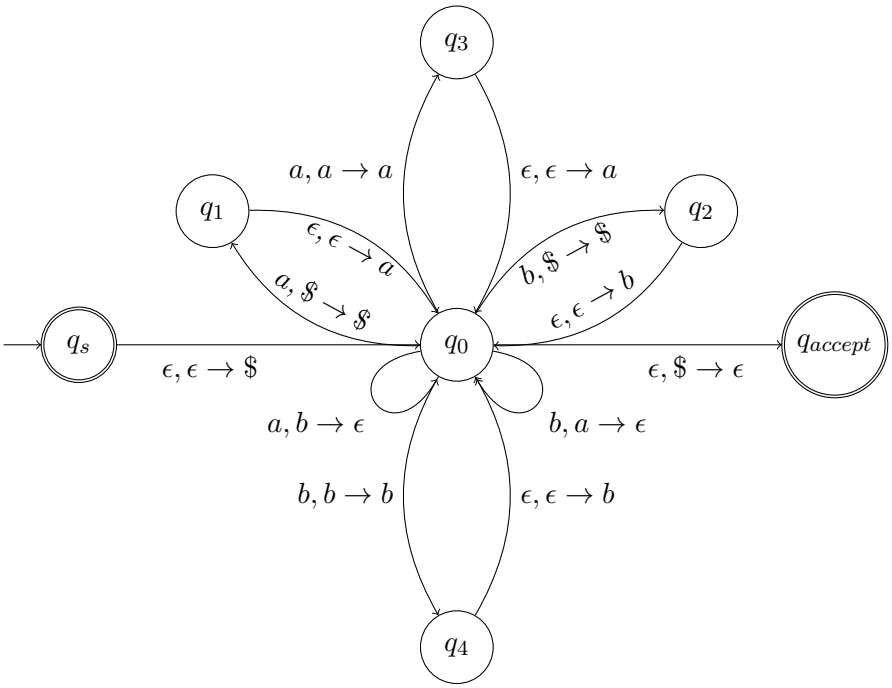
A pushdown automaton looks a lot like an NFA. We still draw a series of states linked by transitions. Like an NFA, a PDA need not fully specify a transition for every state/input pair. PDAs can also contain ϵ-transitions.

Now, though, we need to define the way each transition interacts with the stack as well as the input character that it consumes. To do this, we add to each transition an annotation of the form $a \to b$ where a is the character we pop from the top of the stack and b is the character that we push onto it. Either or both of these characters may be ϵ, which is ignored.

Example 1 Draw a PDA to decide the language $A = \{0^n 1^n \mid n \geq 0\}$



Example 2 Draw a PDA that decides the language $B = \{s \in \{a, b\}^* \mid s \text{ contains the same number of } a\text{s and } b\text{s}\}$.



This example is a little bit trickier. We can sum up our strategy as follows:

- Whenever we see a character and pop the empty delimiter from the stack, replace the delimiter and then push whichever character we just saw
- Whenever we see a character and pop the same character from the stack, replace the character that we just saw and then push another copy
- Whenever we read a character from the input and then pop the *other* character from the stack, just loop back to q_0
- When we run out of input and our stack is empty, transition to the accept state.

This depends on the idea that at any point, our stack is:

1. Empty,
2. composed entirely of as, or
3. composed entirely of bs

Seeing the same character makes the stack grow taller; seeing the opposite character makes it shrink.

2.2 Nondeterminism in Pushdown Automata

When we were building finite automata, we made a point of emphasizing that the addition of nondeterminism didn't allow us to create machines that decided new languages — in other words, that DFAs and NFAs were equivalent in power.

This is *not* the case with pushdown automata. Nondeterministic PDAs are capable of deciding languages that deterministic machines cannot. Since only nondeterministic pushdown automata are capable of deciding the full class of context-free languages, we'll be focusing exclusively on these.

The previous example is a perfect example of why nondeterminism is significant here. When our stack is empty, the addition of ϵ-transitions in particular allows us to transition from q_0 to either q_1 or q_2 and then come back while only having removed a single character from the input. This allows us to operate on the stack multiple times, replacing the \$ delimited that we saw and then pushing whatever character we just read from the input.

2.3 Formal Definition of a Nondeterministic Pushdown Automaton

A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_{start}, F)$, where:

- Q is a finite set of states
- $Sigma$ is the input alphabet
- Γ is the stack alphabet
- δ is a transition function. $\delta : (Q \times \Sigma_{epsilon} \times \Gamma_{\epsilon}) \to \mathbb{P}((Q \times \Gamma_{\epsilon}))$
- q_{start} is the start state. $q_{start} \in Q$.
- F is a set of accepting states. $F \subseteq Q$

3 Context-Free Languages

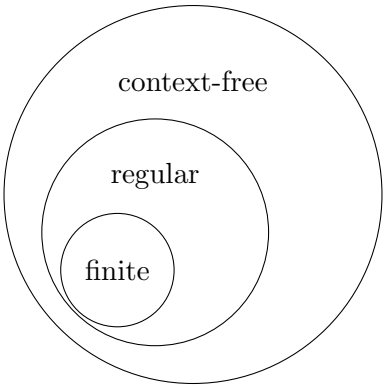
As with regular languages and DFAs before, we now have our first (not very useful) definition of a context-free language:

> A language is *context-free* if and only if there is some nondeterministic pushdown automaton that decides it.

We have another, much less formal definition based on the way that we conceived of the pushdown automaton: if regular languages are those that can't count infinitely, context-free languages are those that can count zero or one things infinitely. When looking at some language and trying to determine its class, "Do I have to count?" is often an excellent first question.

3.1 The Chomsky Hierarchy Revisited: Every Regular Language is Context-Free

At the end of the last lecture, we drew the following diagram of (part of) the Chomsky Hierarchy of Languages:



Notably, we've drawn context-free languages as a circle that fully encompasses the regular language. In other words, we've implied here that regular languages are a subset of context-free languages, and that every regular language is also context-free.

We'll go through a slightly more rigorous proof of this later, but for now, this is actually pretty obvious. We define a regular language as any language decided by an NFA. We define a context-free language as any language decided by a PDA. Since we "created" pushdown automata by adding a stack to an NFA, we can simply trace this backwards and make the following informal observation:

> A language is regular if it is decided by a pushdown automaton that ignores its stack.

We can therefore say that our diagram is accurate, and that regular languages are a proper subset of context-free languages.

4 Context-Free Grammars

Pushdown automata can be useful in deciding certain types of context-free languages, but as we've seen, managing the stack in a state diagram gets complex rather quickly. As with regular expressions and regular languages, though, we have another method of describing context-free languages that is often easier. We describe context-free languages using *context-free grammars*, which you may remember from CS 250.

Context-free grammars are composed of variables, terminals, and rules that transform the former into the latter. Variables are usually written using uppercase letters, while terminals are written in lowercase. Each context-free grammar describes some context-free language. We say that a string s is in the language of some grammar G if s can be generated from G's start variable using its rules appropriately.

4.1 Examples of Context-Free Grammars

Example 1 Write a CFG that describes the language $A = \{0^n 1^n \mid n \geq 0\}$.

This one is pretty simple. We simply keep our variable S in the center of our string, adding 0s on the left and 1s on the right simultaneously. We also allow S to produce the empty string ϵ so that we have a way

to terminate.

$$S \to 0S1 \mid \epsilon$$

Example 2 Write a CFG that describes the language $B = \{s \in \{a, b\}^* \mid s \text{ contains the same number of } a\text{s and } b\text{s}\}$.

B is again a bit trickier, but nowhere near as complex as the equivalent pushdown automaton. We can do this by ensuring that we only ever produce matched pairs of as and bs. By placing another start variable before, between, and after this pair, we can allow these pairs to occur an unlimited number of times in an arbitrary order.

$$S \to SaSbS \mid SbSaS \mid \epsilon$$

Example 3 Write a CFG that describes the language $C = \{a^i b^j \mid i \neq j\}$.

We can think of this language as ordered, but unbalanced strings of as and bs. We can generate such an unbalanced string in two steps:

1. Create a minimally unbalanced string
2. Allow that string to grow in either:
   (a) A strictly balanced fashion, or
   (b) In the direction of its existing imbalance

This gives us a grammar something like the following:

$$\begin{aligned} S &\to aA \mid Bb \mid \epsilon \\ A &\to aA \mid C \\ B &\to Bb \mid C \\ C &\to aCb \mid \epsilon \end{aligned}$$

4.2 Formal Definition of a Context-Free Grammar

A context-free grammar is a 4-tuple (V, Σ, R, S), where:

- V is a finite set of variables, also known as nonterminals
- Σ is the alphabet of terminal symbols used in strings generated by this grammar
- R is a set of rules that transform variables to strings of variables and terminals
- S is the start variable from which every string in the language of this grammar can be generated. $S \in V$

5 More on Context-Free Languages

The example about arithmetic expressions illustrates the power of context-free languages and their usefulness in the real world. Most programming languages are formally specified using some context-free grammar. Many languages, including Java and C, make these grammars available online.

We've now introduced two methods of representing context-free languages. What's the difference? Since these methods are equivalent in power, we're able to prove that a language is context-free using whichever one is easiest for the task at hand. In my experience, this is usually the grammar.

This is because drawing out a pushdown automaton requires us to manually manage our stack. Our transition space has now expanded from $Q \times \Sigma_\epsilon$ to $Q \times \Sigma_\epsilon \times \Gamma_\epsilon$, and including all of the required transitions gets messy rather quickly, even if we don't need to fully-specify our machine.

With a grammar, on the other hand, we don't need to construct our string strictly from left to right. This makes it much easier to construct strings in pieces, similar to the way that we build regular expressions for regular languages. These pieces can also grow from the inside out, as we saw with languages such as $\{0^n 1^n \mid n \geq 0\}$. This ability to build strings from the inside out is especially useful when specifying grammars for languages that require balanced or matched tokens, such as HTML/XML tags or nested parentheses and braces.