Lecture 28

Determinism and Pushdown Automata

“...This was the first time in Knuth’s experience that automata theory had taught him how to solve a real programming problem better than he could solve it before. He showed his results to the third author (V. R. Pratt), and Pratt modified Knuth’s data structure so that the running-time was independent of the alphabet size. When Pratt described the resulting algorithm to Morris, the latter recognized it as his own, and was pleasantly surprised to learn of the $O(m + n)$ time bound, which he and Pratt described in a memorandum [22]. Knuth was chagrined to learn that Morris had already discovered the algorithm, without knowing Cook’s theorem; but the theory of finite-state machines had been of use to Morris too, in his initial conceptualization of the algorithm, so it was still legitimate to conclude that automata theory had actually been helpful in this practical problem.”


In the previous lecture we defined what it means for a pushdown automaton to be deterministic. Specifically, we had the following definition:

28.1 Definition

A **deterministic pushdown automaton** is one whose transition rule, $\delta$, satisfies both of the following:

1. $\delta(q, \sigma, \gamma)$ has at most one element for all $\sigma$ and $\gamma$. 

Copyright © Dean Kelley
2. For any $q \in Q$ and $\gamma \in \Gamma$, when $\delta(q, \lambda, \gamma)$ is defined then $\delta(q, a, \gamma)$ is not defined for all $a \in \Sigma$, and vice versa\(^1\).

We focused on the language $\mathcal{P}$ of all palindromes over $\{a, b\}$ which consists of all strings of the forms $ww^R$, $waw^R$, or $wbw^R$ where $w \in \{a, b\}^*$. The pushdown automaton that we had for $\mathcal{P}$ had nondeterminism present as it crosses the center of strings. It looked like this

![Pushdown automaton for the general language of all palindromes over \{a, b\}](image)

Figure 1: Pushdown automaton for the general language of all palindromes over $\{a, b\}$

Its transitions are given by the following:

\[
\begin{align*}
\delta(q_0, \sigma, \gamma) &= (q_0, \sigma \gamma) & (1) \\
\delta(q_0, \sigma, \gamma) &= (q_1, \gamma) & (2) \\
\delta(q_0, \sigma, \sigma) &= (q_1, \lambda) & (3) \\
\delta(q_1, \sigma, \sigma) &= (q_1, \lambda) & (4) \\
\delta(q_1, \lambda, Z) &= (q_0, Z) & (5)
\end{align*}
\]

Transitions represented by (2) and (3) are where the nondeterminism comes about in this pushdown automaton.

The question we’re going to deal with is, “Is there a deterministic pushdown automaton for $\mathcal{P}$?”

If you want to answer “yes” to this question, you have to find a deterministic pushdown automaton that accepts $\mathcal{P}$. Go ahead and try. You’ll soon find that maybe we should consider the alternative answer. But how do you show that the answer is “no?” Basically you have to show that no deterministic pushdown automaton can accept $\mathcal{P}$, and this requires some brainwork!

1 Showing that no deterministic pushdown automaton accepts $\mathcal{P}$

We’ll start by assuming that there is a deterministic pushdown automaton which accepts $\mathcal{P}$ and then we’ll work towards some kind of contradiction.

Before we even do that, though, we’ll pick up an important result based on something from Lecture 8.

\(^1\)For any $q$, viewing $\delta(q, \sigma, \lambda)$ as specifying 0 or more elements of $Q \times \Gamma^* \cdot \{\lambda, Z\}$, we could say that not both of $\delta(q, \lambda, \gamma)$ and $\delta(q, a, \gamma)$ are nonempty for every $a$. 
1.1 Distinguishing strings with $\mathcal{P}$

If you remember way back to Lecture 8, we defined indistinguishability as follows:

28.2 Definition

Let $L$ be any language over an alphabet $\Sigma$. Two strings $u$ and $v$ over $\Sigma$ are indistinguishable by $L$ if, for all strings $z$ over $\Sigma$, either $uz$ and $vz$ are both in $L$ or both are not in $L$. (Sometimes $u$ and $v$ are said to be ‘$L$-indistinguishable’)

Rephrasing this definition we have:

28.3 Definition

Let $L$ be any language over an alphabet $\Sigma$. Two strings $u$ and $v$ over $\Sigma$ are $L$-distinguishable if there is any string in $\Sigma^*$ for which either $uz \in L$ and $vz \notin L$ or else $uz \notin L$ and $vz \in L$.

It turns out that $\mathcal{P}$ can distinguish every pair of distinct strings over $\Sigma = \{a, b\}$!

To see this, pick any pair of distinct strings over $\Sigma$, $x$ and $y$. Distinct means that $x \neq y$, of course, but we could have that $|x| = |y|$, or not.

If $|x| = |y|$ then we can use $x^R$ to fill the role of $z$ in the definition because $xx^R \in \mathcal{P}$ and $yx^R \notin \mathcal{P}$.

On the other hand, if $|x| \neq |y|$ then one of them is shorter than the other. Suppose that $|x| < |y|$. Then there are two cases. Either $x$ is a prefix of $y$ or else it’s not.

If $x$ is a prefix of $y$ then we have that $y = xu$ for some nonempty string $u$. Let $\sigma$ be whichever of $a$ or $b$ works so that $u\sigma$ is not a palindrome and use $\sigma x^R$ in the role of $z$ in Definition 28.3. Then $x\sigma x^R$ is a palindrome (an odd-length palindrome with $\sigma$ in the center) but $y\sigma x^R = xu\sigma x^R$ which we’ve guaranteed cannot be a palindrome.

If $x$ isn’t a prefix of $y$, then we can use $x^R$ in the role of $z$ in the definition to distinguish $x$ and $y$.

We’ve just proved this handy little theorem:

28.1 Theorem

Every pair of distinct strings, $x$ and $y$, over $\Sigma = \{a, b\}$ are $\mathcal{P}$-distinguishable.

1.2 Tweaking deterministic pushdown automata

Our next step is to get a little control over how pushdown automata fiddle with their stack. When we have a transition that looks like $\delta(q, a, X) = (p, \alpha)$, the symbol $X$ on the stack is replaced by the symbols of the string $\alpha$. There are actually two actions taking place in such a transition. The $X$ is removed from the stack and then the $\alpha$ is placed on the stack. Basically, this transition combines a push and a pop.

We will separate the push and the pop parts of the transition $\delta(q, a, X) = (p, \alpha)$ like this.
First, add a new state corresponding to $\alpha$, say $p_\alpha$.
Then replace the original transition by two new ones:

\[
\delta(q, a, X) = (p_\alpha, \lambda) \\
\delta(p_\alpha, \lambda, \gamma) = (p, \alpha\gamma) \quad \text{for all } \gamma \in \Gamma
\]

In this arrangement, we first pop the $X$ from the stack and change to a state which remembers that we’re going to put $\alpha$ on the stack. Then, the second set of transitions, actually performs the push of $\alpha$. Since we don’t know what lies under $X$ on the stack, we have to cover all possibilities. That’s why we have to add pushing transitions from $p_\alpha$ for all possible stack symbols.

The effect of this change is that when an individual transition doesn’t change the stack’s size, we are guaranteed that it didn’t add or remove anything. We weren’t assured of this before because a transition such as

\[
\delta(q, a, X) = (p, Y)
\]

actually changed the top symbol without changing the stack size at all! In our new, tweaked, pushdown automaton, this transition requires two steps; one to pop the $X$ and another to add the $Y$.

1.3 Why $\mathcal{P}$ isn’t deterministic

Suppose that there is a deterministic pushdown automaton which accepts $\mathcal{P}$, call it $M$. Tweak $M$ as we did above so that the only time it changes its stack size is when it adds or removes something.

Let $x$ be any string over our input alphabet $\{a, b\}$. If we think of all strings $xy$ over $\{a, b\}$ (i.e., all strings having $x$ as a prefix), and think about what $M$ might do with them, we can define $\hat{y}$ to be the string for which $M$’s stack has minimum height after $x\hat{y}$ has been processed. ($M$ may or may not have accepted $x\hat{y}$, it doesn’t matter...all we care about is the height of the stack here).

With a little thought, you realize that there are infinitely many $x\hat{y}$ pairs (because we have infinitely many choices for $x$...countably infinitely many).

On the other hand, since $Q$ and $\Gamma \cup \{Z\}$ are finite, there are only finitely many state-stack symbol pairs possible in the final configuration of any computation on any input string.
Note that every string over \{a,b\} is associated with some state/stack pair in the final configuration of \(M\) on the string. Consequently, focusing on specific strings...the \(x\hat{y}\) strings... each of the \(x\hat{y}\) is associated with some state/stack pair in \(M\)'s final configuration on \(x\hat{y}\).

But there are finitely many state/stack pairs and infinitely many \(x\hat{y}\) strings! Hmmm....

We quickly realize that there must be at least 2 distinct strings which leave \(M\) with the same state/stack pair. Suppose that \(r = x\hat{y}\) and \(s = u\hat{v}\) are two such strings. \(M\)'s processing on \(r\) ends with the same state and stack top symbol as its processing on \(s\).

So, suppose that \(z\) is any string. What does \(M\) do with \(rz\) and \(sz\)? What if \(z = rR\)? Our \(M\) is supposed to accept \(\mathcal{P}\), so it will accept \(rz\). It will also accept \(sz\). It has to, because as far as \(M\) is concerned, \(r\) and \(s\) look alike and adding \(z\) to each of them still makes them look alike!

This alone isn’t enough to insure that we’ve got a contradiction because we might have an \(r\) and \(s\) for which \(z = rR\) will still make \(sz\) be a palindrome (e.g., \(r = aba\) and \(s = ab\)).

Going back to Theorem 28.1, we know that we are guaranteed that for any \(r\) and \(s\), there is some string \(z\) which makes one of \(rz\) or \(sz\) be in \(\mathcal{P}\), and the other not in \(\mathcal{P}\). So, for our specific \(r\) and \(s\), it may or may not be that \(z = rR\), but that doesn’t matter because we’re guaranteed that there’s some \(z\) that does the trick. That is, there is some \(z\) for which one of \(rz\) or \(sz\) is in \(\mathcal{P}\) and the other isn’t. But our dumb \(M\) thinks they’re either both in or both out of \(\mathcal{P}\)! This is the contradiction we want.

We’ve proved the following:

\textbf{28.2 Theorem}

There are languages accepted by pushdown automata which are not accepted by deterministic pushdown automata\(^2\).

\(^2\)Equivalently, “there are languages accepted by pushdown automata which are not deterministic context free languages.” But this statement feels funny because we really haven’t connected the languages accepted by pushdown automata to context free languages in any kind of rigorous manner. We’re going to fix that in the next two lectures, so hold on!!
To say that this is an important fact is sort of like saying that it’s an important fact that combustion takes place when you toss a lit match into a can of gasoline. Determinism is important when we write programs. In particular when we write programs which translate some programming language into machine instructions (but also when we do things like adjust the dollar amount in your checking account).

We encountered a similar sentiment when we discussed ambiguity in context free grammars and inherent ambiguity in context free languages. You can’t have\(^3\) a compiler for an inherently ambiguous programming language!

The language of palindromes, \(P\), was our poster child for nondeterminism. A popular poster child for ambiguity is \(L_a = \{a^n b^m c^k \mid n = m \text{ or } m = k\}\). For \(P\), identifying the point at which you’re done processing the prefix and now begin processing the suffix is where the nondeterminism arises.

For \(L_a\), ambiguity arises in any grammar because the strings which look like \(a^n b^n c^n\) have both allowable patterns but can be generated either as strings in which there are equally many \(a\)s as \(b\)s, or as those in which there are equally many \(b\)s as \(c\)s. It turns out that this leads to the same kind of nondeterministic needs as we have in \(P\)!

For \(P\) you have to guess the center, for \(L_a\) you have to guess whether to keep track of \(a\)s and \(b\)s or \(b\)s and \(c\)s.

In both cases, the only possible pushdown automata wind up being nondeterministic. Ambiguity and determinism are related by the following fact:

28.1 Fact

If \(L\) is accepted by a deterministic pushdown automaton, then there is an unambiguous context free grammar which generates \(L\).

If you form the contrapositive of Fact 28.1 you have:

28.2 Fact

If a context free language \(L\) is inherently ambiguous, then \(L\) is a nondeterministic context free language.

\(^3\)Or, at least, conveniently make