SE 3310b

Theoretical Foundations of Software Engineering

Pushdown Automata and Pumping Lemma for Context-free Languges

Aleksander Essex

Western Engineering

Pushdown Automata

Introduction

Recall our language:

$$L = \{0^n 1^n : n \geqslant 0\}$$

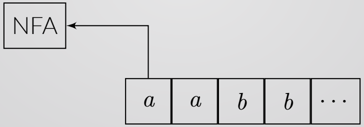
We used the pumping lemma to prove L is not regular. Then we gave the context-free grammar that generates this language:

$$S \rightarrow 0S1 \mid \epsilon$$

Recall grammars are like a recipe for *generating* a language. On the other hand, automata are devices for *recognizing* a language. In this lecture we're going to look at a class of automata for recognizing context-free languages, i.e., **push-down automata**.

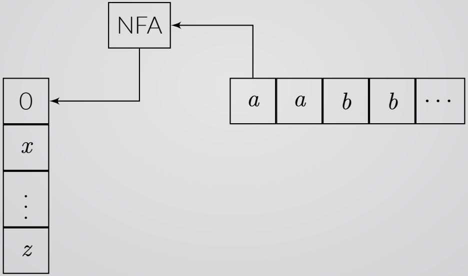
Pushdown Automata

Consider an NFA reading input (i.e., $aabb\ldots$ in this example). It starts at the left (i.e., at the a) and moves to the right, one symbol at a time:



Pushdown Automata

A **pushdown automaton** is essentially a non-deterministic finite automaton with the added ability to read and write from a stack:



Each time it reads an input symbol, it can also read the top of the stack, and do a stack operation.

Pushdown Automaton Definition

Definition 1 (Pushdown Automaton)**.**

A **pushdown automaton** is a 6-tuple:

1. Q: Finite set of states
2. Σ: Input alphabet
3. Γ: Stack alphabet
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$: Transition function
5. $q_0 \in Q$: Start state
6. $F \subseteq Q$: Set of accept states

Pushdown Automaton Notation

From the definition we see that a transition involves the current state, the current input symbol, and the current stack symbol (on the top of the stack), and yields a next state, and a next stack symbol.

$$a, b \to c$$

which can be read as: ”reading a on input and b on top of the stack, replace b with c.”

Pushdown Automaton Notation

- $a = \varepsilon$: We make the transition without reading an input symbol (like in an NFA)
- $b = \varepsilon$: We make the transition without reading or popping from the stack
- $c = \varepsilon$: We make the transition without pushing anything to the stack.

Western Engineering

Pushdown Automaton: The Stack

Using this notation, we can define a set of stack operations:

- **Push** - $x, \varepsilon \rightarrow y$: If you read x on input, push y
- **Pop** - $x, y \rightarrow \varepsilon$: If you read x on input and y on the stack, push ϵ.
- **Replace** - $x, y \rightarrow z$: If you read x on input and y on the stack, push z (i.e., replace y with z)
- **Do nothing** - $x, \varepsilon \rightarrow \varepsilon$: If you read x on input, do nothing to the stack

Bottom of the stack: Technically the stack is a string and is initialized as ε. Depending on application, you may wish to push an explicit ”bottom-of-the-stack” when the automaton starts.

Western Engineering

PDA Example 1

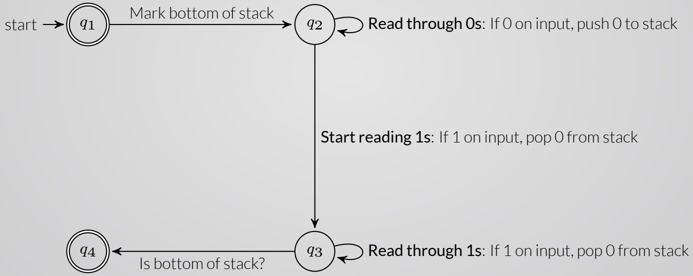
Recall our language $L = \{0^n 1^n : n \geqslant 0\}$. Let's draw a PDA that recognizes this language. Here's what we'll need to do:

1. Mark the bottom of the stack so we'll know if/when we encounter it later
2. Start pushing all the 0's onto the stack
3. As soon as you see a 1, start popping items off the stack: pop one item for each 1 you see
4. If the stack becomes empty upon seeing the last 1, then you saw equal 0's and 1's. Therefore accept.

Western Engineering

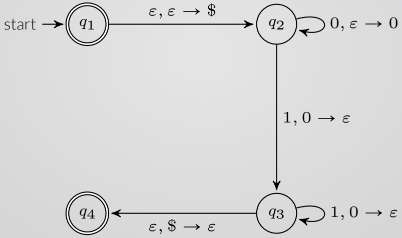
PDA Example 1

Putting this together, we have:


start
q_1
Mark bottom of stack
q_2
Read through 0s: If 0 on input, push 0 to stack
Start reading 1s: If 1 on input, pop 0 from stack
q_3
Read through 1s: If 1 on input, pop 0 from stack
q_4
Is bottom of stack?

PDA Example 1

Or, rewritten in our transition notation:



Example 1: Sanity check

- **More 0s**: If we have more 0's than 1's, then there will still be 0's on the stack when the computation stops, and thus we won't be able to transition to the accept state q_4.
- **More 1s**: If we have more 1's than 0's, then as soon as we run out of 0's, we'll see the \$ symbol and transition to q_4. However, since 1's are remaining in the input (and there are no defined transitions out of q_4) the computation "poofs"/dies.
- **Out of order 0's and 1's**: Same as before; as per the transitions there is nowhere else for the computation to go, so it also "poofs"/dies.

Equivalence of PDAs and CFGs

Theorem 2.

A language is context-free iff there exists some PDA that recognizes it.

To prove this, we follow the same strategy when we proved the equivalence of DFAs and NFAs, i.e.,

- Prove for every CFG generating language L, there exists some PDA recognizing L (see: Lemma 2.21 in Sipser)
- Prove for every PDF recognizing L, there exists some CFG generating L (see: Lemma 2.27 in Sipser)

Western Engineering

Pumping Lemma for Context-free Languages

Non Context-free Languages

Consider the language $L = \{a^n b^n : n \geqslant 0\}$. We've proven this language non-regular using the pumping lemma for regular languages, and the result of our study thus far as shown CFGs and PDFs are capable of some form of ”counting.” But just how many things can a CFG/PDF keep track of?

Non Context-free Languages

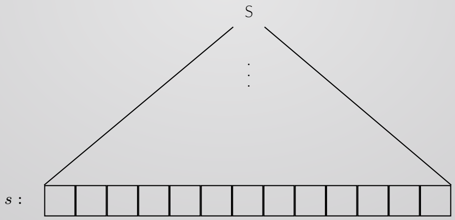
Now consider another language: $L' = \{a^n b^n c^n : n \geqslant 0\}$. Now we need to keep track of *three* things.

Is there a CFG or PDF that can generate (resp. recognize) L'? It seems like *no*. Recall our solution to L was to push all the *a*'s on to the stack, and pop them off to match each of the *b*'s. But if we try to do this for L', when we're done comparing a's to b's, we're left with an empty stack. So how can you compare *c*'s if there's nothing left in memory to compare to?

Western Engineering

Parse Trees

Let L be a language, and let $s \in L$ be a string. If L is context-free, there exists some context-free grammar to generate s. We can derive s beginning with a start-variable S, and substituting variables/terminals according to the grammar until only terminals remain:



Parse Trees

Recall in the *pumping lemma for regular languages*, if a string was ”long enough,” then by the pigeonhole principle it must revisit a state within the automaton forming a loop.

Let’s extend this idea to the context-free grammar setting: if a string is ”long enough” then by the pigeonhole principle it must revisit a variable somewhere during the derivation.

Parse Trees: Example

Consider the following grammar:

$$\begin{array}{rcr} S & \rightarrow & aTb \\ T & \rightarrow & aTb \mid \varepsilon \end{array}$$

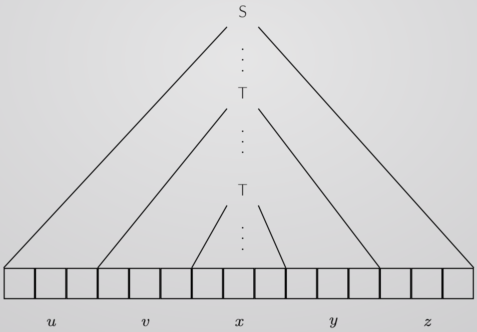
Suppose we wanted the generate the string $aabb$. Under the following derivation:

$$S \rightarrow aTb \rightarrow aaTbb \rightarrow aa\varepsilon bb \rightarrow aabb$$

the variable T was repeated. That is, T is substituted for something that led to another T. And, like in the pumping lemma for regular languages, we can "pump" this loop and the result should still be in our language.

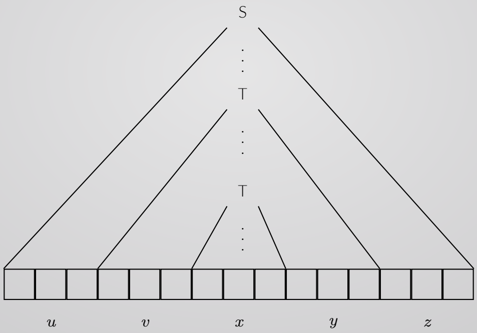
Parse Trees

This diagrams depicts a loop, i.e, a variable T that follows a derivation leading back to a T. The resulting string s can segmented in to five parts: $uvxyz$. Notice the outcome of the $T \ldots T$ loop is strictly contained in substrings v and y.



Pumping a Derivation

Ok so what happens if we substitute one of the Ts with another $T \ldots T$ derivation (i.e., pump once)?



Pumping a Derivation

If $s = uvxyz$, and v and y are the result of the $T \dots T$ derivation, then if we repeat (pump) the $T \dots T$ segment, i.e., to something of the form: $T \dots T \dots T$, then the v and y sections will be repeated: $s' = uvvxyyz$. If L is context-free, then s' must be a member of L. So too must $s'' = uv \dots vxy \dots yz$.

Similarly, we should be able to remove the $T \dots T$ segment, and hence the v and y substrings, and still have the result be part of the language.

So if $s = uvxyz \in L$, and L is context-free, then is must also be the case that $s' = uv^ixy^iz \in L$ for $i \geqslant 0$.

Pumping Lemma for Context-free Languages

Definition 3 (Pumping Lemma for Context-free Languages)**.**

If L is a context-free language, then for all strings $s \in L$ where $|s| \geqslant p$ (the pumping length), then s can be divited into five parts: $s = uvxyz$ where:

1. $uv^ixy^iz \in L$ for $i \geqslant 0$
2. $|vy| > 0$ (i.e., vy is non-empty)
3. $|vxy| \leqslant p$ (i.e., vxy is no greater than the pumping length)

Western Engineering

Example 1: Counting 3 Things

Use the pumping lemma for context-free languages to prove the following language is non context-free:

$$L = \{a^n b^n c^n : n \geqslant 0\}$$

Example 2: String Copy

Use the pumping lemma for context-free languages to prove the following language is non context-free:

$$L = \{ww : w \in \{0,1\}^*\}$$

Languages Classes

We've looked at both counting and string-copy languages. Here's a table showing the capabilities/limitations of each language class:

Language Class	Counting Languages	String-copy Languages
Regular	Can't count unbounded	Can't copy unbounded
Context-free	Two things: $a^n b^n$	Can't copy forwards (can do ww^R)
Context-sensitive	Up to four things: $a^n b^n c^n d^n$	Single copy: ww
Recursively enumerable	Unrestricted: $a^n b^n \ldots$	Unrestricted: $ww \ldots w$