3. Pushdown (string) automata
(and context-free grammars, a bit)

A pushdown automaton is a certain kind of "infinite-state machine". Its distinctive properties relative to other kinds of infinite-state machines come from the fact that its unbounded memory takes the form of a *stack*.

We can define pushdown automata in a way that closely parallels FSAs. (One difference is that I'm allowing ε-transitions here — but nothing much changes if we allow these in FSAs too.)

(1) A pushdown automaton (PDA) is a five-tuple $(\Gamma, \Sigma, I, F, \Delta)$ where:
- Γ is the *stack alphabet*;
- Σ is the *(surface) alphabet*;
- $I \subseteq \Gamma^*$ is the set of initial stack contents;
- $F \subseteq \Gamma^*$ is the set of ending stack contents; and
- $\Delta \subseteq \Gamma^* \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*$ is a *finite* set of transitions.

It's easiest to see how PDAs work via some examples.

1 A "counting" language

Here's the definition of a PDA:

(2) $\Gamma = \{\mathrm{X}, \mathrm{Y}, \mathrm{A}\}$
$\Sigma = \{\mathtt{a}, \mathtt{b}\}$
$I = \{\mathrm{X}\}$
$F = \{\mathrm{Y}\}$
$\Delta = \{(\mathrm{X}, \mathtt{a}, \mathrm{AX}), (\mathrm{X}, \varepsilon, \mathrm{Y}), (\mathrm{AY}, \mathtt{b}, \mathrm{Y})\}$

The transition $(\mathrm{X}, \mathtt{a}, \mathrm{AX})$ is like a schema with unboundedly many specific instantiations of the form $(\cdots\mathrm{X}, \mathtt{a}, \cdots\mathrm{AX})$, i.e. "pop X, emit a, push A, push X".

This PDA generates $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$ by classifying prefixes into unboundedly many categories, identified by the contents of the stack.

To show that this PDA generates aaabbb, for example, we need to find a corresponding sequence of transitions from the initial stack-contents $\mathrm{X} \in I$ to the ending stack-contents $\mathrm{Y} \in F$.

(3)

	Transition	String	Stack Contents
Step 0	—	ε	X
Step 1	$(\mathrm{X}, \mathtt{a}, \mathrm{AX})$	a	AX
Step 2	$(\mathrm{X}, \mathtt{a}, \mathrm{AX})$	aa	AAX
Step 3	$(\mathrm{X}, \mathtt{a}, \mathrm{AX})$	aaa	AAAX
Step 4	$(\mathrm{X}, \varepsilon, \mathrm{Y})$	aaa	AAAY
Step 5	$(\mathrm{AY}, \mathtt{b}, \mathrm{Y})$	aaab	AAY
Step 6	$(\mathrm{AY}, \mathtt{b}, \mathrm{Y})$	aaabb	AY
Step 7	$(\mathrm{AY}, \mathtt{b}, \mathrm{Y})$	aaabbb	Y

2 A nesting-dependencies language

(4) $\Gamma = \{\text{X}, \text{Y}, \text{F}, \text{T}\}$
$\Sigma = \{\texttt{flip}, \texttt{flop}, \texttt{tick}, \texttt{tock}\}$
$I = \{\text{X}\}$
$F = \{\text{Y}\}$
$\Delta = \{(\text{X}, \texttt{flip}, \text{FX}), (\text{X}, \texttt{tick}, \text{TX}), (\text{X}, \varepsilon, \text{Y}), (\text{FY}, \texttt{flop}, \text{Y}), (\text{TY}, \texttt{tick}, \text{Y})\}$

(5)

	Transition	String	Stack Contents
Step 0	—	ε	X
Step 1	(X, `flip`, FX)	`flip`	FX
Step 2	(X, `tick`, TX)	`flip tick`	FTX
Step 3	(X, `flip`, FX)	`flip tick flip`	FTFX
Step 4	(X, `flip`, FX)	`flip tick flip flip`	FTFFX
Step 5	(X, ε, Y)	`flip tick flip flip`	FTFFY
Step 6	(FY, `flop`, Y)	`flip tick flip flip flop`	FTFY
Step 7	(FY, `flop`, Y)	`flip tick flip flip flop flop`	FTY
Step 8	(TY, `tock`, Y)	`flip tick flip flip flop flop tock`	FY
Step 9	(FY, `flop`, Y)	`flip tick flip flip flop flop tock flop`	Y

We've seen this pattern generated in two different ways now:

- Categorizing **initial** portions of a string ("growing left-to-right") using **unbounded stack** memory.
- Categorizing **medial** portions of a string ("growing inside-out") using **finite** memory.

3 Relationship to context-free grammars

Any context-free phrase-structure grammar can be mechanically converted into a pushdown automaton that generates the same set of strings.[1] (And vice-versa, although the other direction is a bit trickier.)

To make things very slightly more manageable, let's assume that:

- the right-hand side of any context-free rule is either (i) a single terminal symbol, or (ii) one or more nonterminal symbols; and
- the CFG to convert has a unique start symbol, S.

Then the conversion works like this:

(6) a. The stack alphabet Γ of the PDA is the set of nonterminal symbols of the CFG.
b. The surface alphabet Σ of the PDA is the set of terminal symbols of the CFG.
c. $I = \{\varepsilon\}$, i.e. the stack starts empty
d. $F = \{\text{S}\}$
e. For each rule $A \to x$, we include in Δ a transition (ε, x, A).
f. For each rule $A \to B_1 \dots B_n$, we include in Δ a transition $(B_1 \dots B_n, \varepsilon, A)$.

Suppose for example we have the very simple CFG in (7).

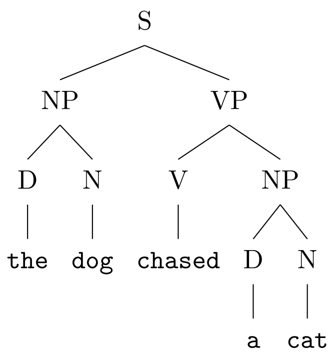
[1] Actually, there are various ways to do this. The one illustrated here corresponds to "bottom-up" or "shift-reduce" parsing/recognition; alternatives include top-down and left-corner.

(7)

S → NP VP	D → `the`
NP → D N	D → `a`
VP → V NP	N → `dog`
	N → `cat`
	V → `chased`

Then the derivation indicated in (8) corresponds to the sequence of PDA transitions in (9).

(8)



(9)

	Transition	String	Stack Contents
Step 0	—	ε	ε
Step 1	$(\varepsilon, \texttt{the}, \mathrm{D})$	`the`	D
Step 2	$(\varepsilon, \texttt{dog}, \mathrm{N})$	`the dog`	D N
Step 3	$(\mathrm{D\ N}, \varepsilon, \mathrm{NP})$	`the dog`	NP
Step 4	$(\varepsilon, \texttt{chased}, \mathrm{V})$	`the dog chased`	NP V
Step 5	$(\varepsilon, \texttt{a}, \mathrm{D})$	`the dog chased a`	NP V D
Step 6	$(\varepsilon, \texttt{cat}, \mathrm{N})$	`the dog chased a cat`	NP V D N
Step 7	$(\mathrm{D\ N}, \varepsilon, \mathrm{NP})$	`the dog chased a cat`	NP V NP
Step 8	$(\mathrm{V\ NP}, \varepsilon, \mathrm{VP})$	`the dog chased a cat`	NP VP
Step 9	$(\mathrm{NP\ VP}, \varepsilon, \mathrm{S})$	`the dog chased a cat`	S

A useful exercise is to apply this CFG-to-PDA conversion to the following CFG, and compare the workings of the resulting PDA to the example run in (5).

S → FLIP S FLOP	FLIP → `flip`
S → TICK S TOCK	FLOP → `flop`
S → FLIP FLOP	TICK → `tick`
S → TICK TOCK	TOCK → `tock`