Context-free grammars & pushdown automata

How to build a PDA that accepts precisely the strings generated by a CFG,

<u>Idea</u>: CFG $G = (\underbrace{\Sigma}_{\text{letters}}, \underbrace{V}_{\text{vars}}, \overbrace{S}^{\text{start var}}, \overbrace{P}^{\text{productions}})$

10/15

variables go on the stack
letters that would be gen. get matched w/ input string.

<u>Assumption</u>: My grammar is in Chomsky Normal Form: productions are either:

$$A \rightarrow x$$

or

$$A \rightarrow BC$$

How will the PDA work?

First, push the start var on the stack.

Repeatedly, either

1) Match top var of stack w/ first remaining letter in the string, discarding both

(w a prod. $A \rightarrow x$)

or 2) Replace top var in stack w/ 2 vars corresp to a prod $A \to BC$.

If we end up w/ no letters in string and no vars on stack, then go to a final state and declare success.

Formally: Our ~~PDA~~ $M = (Q, \Sigma, \Gamma, q_0, F, z, \delta)$

has $Q = \{q_0, q, q_f\}$

$\Sigma = \Sigma$ ← from our CFG G

$\Gamma = V \cup \{z\}$ ← assumed to not be in V

$q_0 = q_0$

$F = \{q_f\}$

$z = z$

δ includes:

$$(q_0, \lambda, z) \longrightarrow (q, Sz)$$

For each production of the form $A \to x$,

$$(q, x, A) \longrightarrow (q, \lambda).$$

For each production of the form $A \to BC$

$$(q, \lambda, A) \to (q, BC).$$

$$(q, \lambda, z) \to (q_f, z)$$

irrelevant

Example:

$$\left.\begin{array}{l} S \to AB \mid x \\ A \to BC \mid y \\ B \to BS \mid w \\ C \to x \end{array}\right\}$$

grammar

$\Sigma = \{w, x, y\}$

$V = \{A, B, C, S\}$

Two Derivations: (both leftmost)

$$S \to x$$

$$S \to AB \to BCB \to wCB \to wxB$$

$$\to wxBS \to wxw\cancel{S} \to wxwx$$

PDA:

$$(q_0, \lambda, z) \longrightarrow (q, Sz)$$
$$(q, x, S) \longrightarrow (q, \lambda)$$
$$(q, y, A) \longrightarrow (q, \lambda)$$
$$(q, w, B) \longrightarrow (q, \lambda)$$
$$(q, x, C) \longrightarrow (q, \lambda)$$
$$(q, \lambda, S) \longrightarrow (q, AB)$$
$$(q, \lambda, A) \longrightarrow (q, BC)$$
$$(q, \lambda, B) \longrightarrow (q, BS)$$
$$(q, \lambda, z) \longrightarrow (q_f, z)$$

Match our derivations with processing the PDA on the out put string:

State	Remaining string	Stack
q_0	x	z
q	x	Sz
q	λ	z
q_f	λ	z ← final.

State	Remaining String	Stack	derivation
q_0	wxwx	z	
q	wxwx	Sz	S
q	wxwx	ABz	AB
q	wxwx	BCBz	BCB
q	xwx	CBz	wCB
q	wx	Bz	wxB
q	wx	BSz	wxBS
q	x	Sz	wxwS
q	λ	z	wxwx
q_f	λ	z	$\leftarrow$ final

I hope this convinces you the PDA will accept every string produces by the grammar.

Why does this only accept strings produced by the grammar?

1) The first move in the PDA is always to

$$(q, \underset{\uparrow \text{ input}}{w}, Sz)$$

General idea of induction proofs:
find "loop-invariant":

We have a derivation of our grammar that produces:

$$\cancel{xV\ldots, \quad x \in \Sigma^*}, \qquad (*)$$

$$xA \qquad x \in \Sigma^*, \ A \in V^*$$

where we have Az on the stack and the ~~remaing~~ remaining input is y, where $w = xy$.

After every move in our PDA, statement $(*)$ is true, (b/c it was true before the move, and our moves $(q \ldots) \to (q \ldots)$ keep this true. (I could take 5 minutes to check carefully).

The only way to get to our final state is if we have z on the stack, which corresponds to a derivation

$$S \to \cdots\cdots \to x, \quad \text{where } w = xy, \ y \text{ rem. input}.$$

We can only accept at the final state if $y = \lambda$ (by rules of PDAs), so $w = x$.

Take 5 minutes to check carefully:
Induction ~~how~~ on # times through loop.

<u>Base case:</u> 0 times - we start with w as input, $S z$ on stack (once we've gotten to state q). This matches the derivation S ~~⇒~~ (where we haven't done anything)

<u>Inductive case:</u> Before doing the loop, we had some derivation

$$S \to \cdots \to xA,$$

where $w = xy$, y rem. input, A on stack.

If we do a PDA move of the from

$$(q, \ell, B) \longrightarrow (q, \lambda),$$

then we had B as the 1st letter in A, ℓ as first letter in y, so we have the derivation

$$S \to \cdots \to \overset{xBA'}{\overset{\|}{x\cancel{A}}} \to x\ell A'$$

since $B \rightarrow \ell$ is a production in our grammar, and the remaining input loses an ℓ, and stack loses a B, and the loop-inv. is still true