Lecture 12

Pushdown automata

In this lecture, we will define a computational model that captures the context-free languages. This model is called *pushdown automata*. A pushdown automaton is a machine which is similar to an NFA, but which has access to (unbounded) memory in a limited form. Specifically, a pushdown automaton (PDA) will have access to a *stack*, which is a block of memory of unbounded length but which can only be accessed by "pushing" or "popping" a single element from the top of the stack. We also note that pushdown automata, as defined in this course, will always be non-deterministic; one could define deterministic PDAs as well, but that turns out to give rise to a different (strictly weaker) computational model, unlike what happened with DFAs and NFAs.

Each pushdown automaton has a language it recognizes. It turns out that a language A has a pushdown automaton recognizing it if and only if it is context-free; in this way, the PDAs give a new characterization of the context-free languages, which can be more intuitive in some situations.

12.1 The definition of pushdown automata

A pushdown automaton (PDA) is a 6-tuple

$$P = (Q, \Sigma, \Gamma, \delta, q_0, F)$$

where

1. Q is a non-empty finite set, whose elements are called *states*,
2. Σ is a non-empty finite set, called the *input alphabet*,
3. Γ is a non-empty finite set, called the *stack alphabet*,
4. δ is a transition function with signature
$$\delta \colon Q \times (\Sigma \cup (\{\downarrow, \uparrow\} \times \Gamma) \cup \{\epsilon\}) \to \mathcal{P}(Q),$$
5. $q_0 \in Q$ is a *start state*, and
6. $F \subseteq Q$ is a set of *accept states*.

We also require that the three sets Σ, $\{\downarrow, \uparrow\} \times \Gamma$, and $\{\epsilon\}$ be disjoint.

As you can see, the definition of a PDA is somewhat similar to the definition of an NFA (note that the output of δ is a set of states rather than a single state, just as it was in the definition of an

NFA). The main difference is the existence of a *stack*, which is a separate block of memory that the machine can access. This stack has its own alphabet, Γ; in each cell of the stack, a symbol from Γ may be written.

When a PDA P is run on a string w, the stack of P starts out empty. Then, when P takes a transition that is labeled with $(\downarrow, c_1)$ for some $c_1 \in \Gamma$, the symbol c_1 gets *pushed* onto the top of the stack; from that point on, the stack will have c_1 at the top, until another symbol c_2 is pushed, at which point the stack will have the symbol c_2 at the top (and c_1 will be underneath it). Later on in the run, symbols may get "popped" from the stack when a transition such as $(\uparrow, c_2)$ is taken. This will remove c_2 from the top of the stack, and leave c_1 there instead.

Note that if a symbol $c \in \Gamma$ is at the top of the stack, we are not allowed to take a transition $(\uparrow, c')$ for $c' \neq c$; such a transition attempts to pop c' from the stack, but since the top of the stack has c, the transition is forbidden and a different transition must be taken instead (recall that the pushdown automaton is non-deterministic, so there may be many possible transitions it may take from a given state).

This explains how P behaves when it takes transitions labeled $(\uparrow, c)$ or $(\downarrow, c)$ for $c \in \Gamma$. P may also take ϵ-transitions between its states, which work the same way as in NFAs. Finally, P may take transitions that are labeled by symbols from Σ. Such transitions read a bit of the input, just as in NFAs. Arcs in a PDA may be labeled by any one of these different types of labels: the ϵ label; a label $c \in \Sigma$ corresponding to reading a bit of the input; or a label $(\downarrow, c')$ or $(\uparrow, c')$ for $c' \in \Gamma$, corresponding to pushing or popping the symbol c' on the stack. If there is a valid sequence of transitions which starts from the start state of P, ends at an accept state of P, and reads exactly the input string w, then we say P accepts w. We will now formally define this.

12.2 Defining acceptance for a PDA

In order to formally define what it means for a PDA P to accept a string w, we will first define the concept of a *valid stack string*, which is a sequence of symbols from $\{\uparrow, \downarrow\} \times \Gamma$ that corresponds to a legal sequence of push/pop operations made to a stack that starts out empty. For example, the sequence

$$(\downarrow, 0), (\downarrow, 1), (\uparrow, 1), (\downarrow, 0), (\uparrow, 0)$$

is a valid stack string, because it corresponds to pushing 0, then pushing 1, then popping 1, then pushing 0, then popping 0; the contents of the stack after each of these operations will be $\epsilon, 0, 01, 0, 00, 0$ (where the top of the stack is the rightmost symbol in each string). On the other hand, the sequence

$$(\downarrow, 0), (\downarrow, 1), (\uparrow, 0)$$

is not a valid stack string, because we are trying to pop 0 when the top of the stack has a 1. That is, in this sequence of operations, 0 is pushed onto the stack, then 1 is pushed, and then we try to pop 0, but this is not possible because the top of the stack has a 1. Hence this is not a valid stack string. Similarly,

$$(\downarrow, 0), (\uparrow, 0), (\uparrow, 0)$$

is also not valid, because we are trying to pop a 0 after the stack is already empty.

How should we formally define the set of valid stack strings? It turns out that one way to do so is to use context-free grammars. Consider the context-free grammar which has the rule

$$S \to (\downarrow c)S(\uparrow c)S$$

for each $c \in \Gamma$, as well as the rule $S \to \epsilon$. This context-free grammar G will generate all the valid stack strings that end up with an empty stack. Since we want to allow the stack to be non-empty at the end, we will take the language of all prefixes of this language; that is, we define the set of valid stack strings to be $\text{Prefix}(L(G))$ for this context-free grammar G. Note that since the set of all prefixes of a context-free language is context-free, the set of valid stack strings happens to be a context-free language over the alphabet $\{\uparrow, \downarrow\} \times \Gamma$.

Now that we have defined the set of valid stack strings, we can define what it means for a PDA to accept a string w.

Definition 12.1. *Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a pushdown automaton, and let $w \in \Sigma^*$ be a string. We say that P* accepts *w if there exist*

- *a natural number $m \in \mathbb{N}$,*
- *a sequence of states $p_0, p_1, \ldots, p_m \in Q$, and*
- *a sequence of symbols $c_1, c_2, \ldots, c_m \in \Sigma \cup (\{\uparrow, \downarrow\} \times \Gamma) \cup \{\epsilon\}$*

such that the following properties hold:

1. *$p_0 = q_0$ and $p_m \in F$,*
2. *$p_i \in \delta(p_{i-1}, c_i)$ for all $i \in \{1, 2, \ldots, m\}$,*
3. *the sequence of symbols from $c_1, c_2, \ldots, c_m$ which are from Σ form exactly the string w when concatenated together in order, and*
4. *the sequence of symbols from $c_1, c_2, \ldots, c_m$ which are from $\{\uparrow, \downarrow\} \times \Gamma$ form a valid stack string when concatenated together in order.*

In other words, we say that P accepts w if there is a sequence of transitions we can take from the start state to an accept state which, together, read exactly w from the input and which push and pop from the stack in a legal way. Note the non-determinism of the definition: the sequence of states and symbols c_i does not need to be unique, and as long as any such sequence is valid and leads to an accept state, we accept the string.

If P does not accept w, we say that P rejects w. The language recognized by P, denoted by $L(P)$, is the set of all strings in Σ^* that are accepted by P.

12.3 Transition diagrams and examples of PDAs

We draw PDAs using transition diagrams, similar to how we drew DFAs and NFAs. In a transition diagram for a PDA, we will have one node (circle) per state, the accept states will be circled twice, and transitions between states will be denoted using arrows. The start state q_0 will have an arrow coming in from nowhere, and in all other cases, an arrow between two states p_1 and p_2 will be labeled by $c \in \Sigma \cup (\{\uparrow, \downarrow\} \times \Gamma) \cup \{\epsilon\}$ and will denote the fact that p_2 is in $\delta(p_1, c)$; that is, it will denote the fact that transitioning from p_1 to p_2 while performing the action specified by c is legal (where the actions specified by c might be pushing or popping from the stack, reading an input symbol, or doing nothing (denoted by ϵ)). We give an example of a transition diagram in Figure 12.1.

In this PDA, the state q_0 is the start state, and q_3 is the only accept state. The arcs are labeled by 0, 1, $\downarrow 0$, $\uparrow 0$, and ϵ, from which we can infer that the input alphabet of this PDA is $\Sigma = \{0, 1\}$ and the stack alphabet is $\Gamma = \{0\}$. From q_0, the PDA can read a 0 from the input, followed by

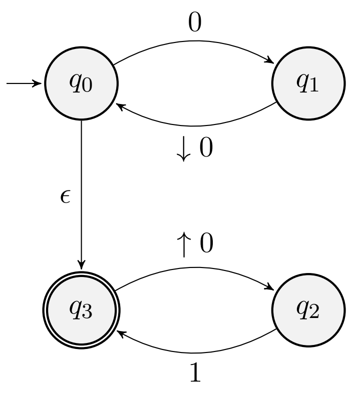

Figure 12.1: An example of a PDA.

pushing 0 onto the stack; alternatively, the PDA can ϵ-transition at any time from q_0 to q_3, at which point it can pop a 0 from the stack followed by reading a 1 from the input.

What does this PDA do? With some thought, it should be clear that this PDA recognizes the language $\{0^n1^m : n, m \in \mathbb{N}, n \geq m\}$. This language is not regular, so we see already that PDAs are more powerful than NFAs. The reason is that the stack can allow the PDA to count things: in this case, the PDA pushes a 0 onto the stack whenever it reads a 0 from the input, allowing it to keep track of the number of 0s; it then pops a 0 from the stack each time it sees a 1, which ensures that the number of 1s it sees will not exceed the number of 0s seen. (Remember, in order for a PDA P to accept a string w, it must reach an accept state after reading all the symbols of w; if P did not finish reading all the symbols of w, it must keep transitioning until it does so.)

Note that in the above PDA, if we go from q_0 to q_1 while reading 0 from the input, we must follow this up by returning from q_1 to q_0 and pushing a 0 onto the stack. In a sense, the state q_1 is only there to ensure that we do the actions "read 0 from the input" and "push 0 onto the stack" together. Similarly, q_2 just ensures that we pop 0 from the stack and read 1 from the input together. To simplify the transition diagrams, we will allow arcs to be labeled by a *sequence* of symbols from $\Sigma \cup (\{\uparrow, \downarrow\} \times \Gamma) \cup \{\epsilon\}$ instead of just one at a time. This will allow us to draw the above PDA using just two states, as in Figure 12.2.

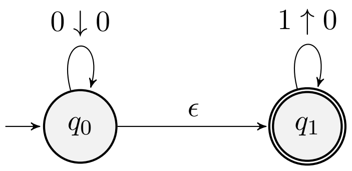

Figure 12.2: An example of a PDA with more concise notation for the transitions.

It will often be convenient to allow a PDA to check if its stack is empty. The way we defined it, this is not really possible: a PDA can only try to pop a specified symbol from the stack. However, we can simulate this behavior by adding a special symbol to the stack alphabet, which we will denote $\diamond$. We can always modify any PDA so that it pushes the special stack symbol $\diamond$ to the stack at the very beginning. The PDA can then check if the stack is "empty" by trying to pop $\diamond$ from it (and it can push $\diamond$ back onto the stack afterwards, to ensure that the bottom of the stack always contains $\diamond$).

Additionally, note that the way we defined PDAs allows them to accept a string even when the stack is not empty at the end of the sequence of transitions. We can always modify a PDA so that it accepts only when the stack is empty: the idea would be that each of the previous accept states will have an ϵ transition to a special state p, the state p will have a loop that pops any symbol in $\Gamma \setminus \{\diamond\}$ from the stack, and finally, we will have a new accepting state p' and we transition from p to p' when popping $\diamond$ from the stack. Modified this way, the PDA will now have only one accept state p', and whenever p' is reached, the stack will be empty; however, the PDA still accepts the exact same strings it did before. We give an example of this in Figure 12.3, where we illustrate this modification for the PDA in Figure 12.2.

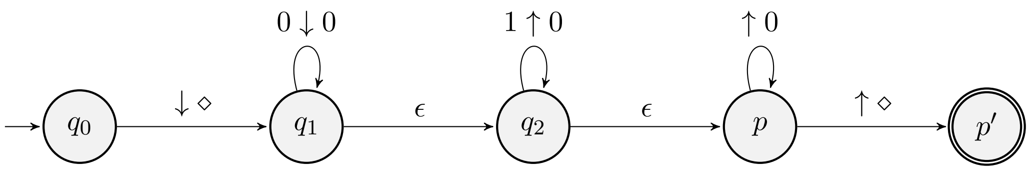

Figure 12.3: A PDA modified to empty the stack before accepting.

If we want to accept the language $\{0^n1^n : n \in \mathbb{N}\}$ instead of the language $\{0^n1^m : n, m \in \mathbb{N}, n \geq m\}$, we could do so by skipping the part where we allow the PDA to remove extra 0s in its stack without reading 1s from the input; this gives the PDA in Figure 12.4.

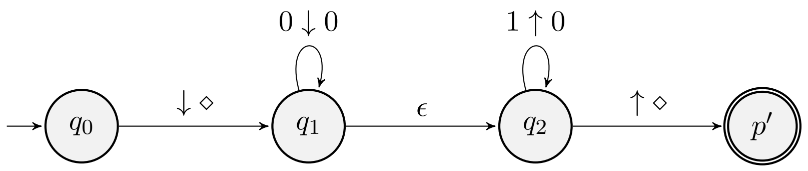

Figure 12.4: A PDA for the language $\{0^n1^n : n \in \mathbb{N}\}$.

12.4 Equivalence of PDAs and CFGs

We now show that a language is context-free if and only if there is a PDA recognizing it. Remember that we defined context-free languages in terms of CFGs that generate them; this means that what we really need to show is that for any CFG G, then there is a PDA P such that $L(P) = L(G)$, and also, for any PDA P, there is a CFG G such that $L(G) = L(P)$. We prove the two directions separately.

Theorem 12.2. *Let $G = (V, \Sigma, R, S)$ be a CFG. Then there is a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ such that $L(P) = L(G)$.*

Proof. The idea is to construct the PDA P so that it uses its stack to expand out grammar rules of G. In other words, at the beginning, P will push the start variable S of G onto its stack; subsequently, P will be allowed to pop a variable in V from its stack, and immediately push back in a sequence of symbols that occurs on the right hand side of a substitution rule for that variable. That is, for each rule $X \to w$ of the grammar G (where $w \in (V \cup \Sigma)^*$), we allow the PDA P to pop X from its stack and push back the symbols of w in reverse order. Additionally, if the top of the stack contains a symbol from Σ, we allow P to pop that symbol if it also reads the same symbol from the input.

This way, P can non-deterministically choose a sequence of grammar rules of G to follow, substituting the left-most variable each time; if the left-most symbol is not a variable, P must read the same symbol from the input in order to proceed. At the end, P will accept if the stack and the input are both empty.

To explicitly implement this, we will take $\Gamma = \Sigma \cup V \cup \{\diamond\}$, where $\diamond$ is a symbol not in Σ or V. We will have four primary states q_0, q_1, q_2, and q_3, plus some helper states that we will specify later. From the initial state q_0, our only transition will be to push $\diamond$ and go to q_1. From q_1, the only transition will be to push S (the start variable of G) and go to q_2. q_3 will be the only accept state, and will have no outgoing transitions; it will have a single incoming transition, which comes in from q_2 and pops $\diamond$ from the stack.

The main action will take place in q_2. We will have several loops that start and end at q_2, though these loops will sometimes use helper states (for example, going from q_2 to p_1, from p_1 to p_2, and then from p_2 back to q_2; such helper states will allow these loops to perform many actions in sequence).

There will be two types of loops on q_2. For the first type, we will have one loop on q_2 for each symbol $c \in \Sigma$; this loop will read c from the input and pop c from the stack. (Recall that if c is not the next character of the input or if c is not at the top of the stack, such a loop cannot be taken.) For the second type, we will have one loop for each rule $X \to w$ in R (where $w \in (V \cup \Sigma)^*$). This loop will pop X from the stack, and push the symbols of w onto the stack in reverse order, so that the first symbol of w ends up being at the top of the stack. An illustration of this PDA is given in Figure 12.5.

We now claim that this PDA accepts a string $w \in \Sigma^*$ if and only if G generates w. First, if G generates w, then there is a leftmost derivation of w in G; an accepting path for w in P will be to transition from q_0 to q_1 to q_2 while pushing $\diamond$ and S onto the stack, and then follow the loops on q_2 that substitute the leftmost variable according to the leftmost derivation of w; when the top of the stack is a symbol in Σ instead of a variable, we follow the loops on q_2 that pop that symbol and read it from the input w. Eventually, once all the substitution rules of the leftmost derivation of w have been followed, the stack will contain only $\diamond$ and the input will be fully read; at that point, we can transition from q_2 to q_3, arriving at q_3 with empty stack and empty input. Since q_3 is an accepting state, this is an accepting path for w.

Conversely, any accepting path for a string w in P must start by pushing $\diamond$ and S and transitioning to q_2, and it must end by popping $\diamond$ and transitioning to q_3. This means that such a path must read w from the input while clearing the symbol S from the stack, using only the loops on q_2 to do so. The sequence of loops on q_2 that are followed will give us a leftmost derivation of w in G (since if any string other than w is generated, that string cannot be read using the loops of q_2 while clearing the stack). This means that if P accepts w, there is a derivation of w in G. We conclude that $L(P) = L(G)$, as desired. □

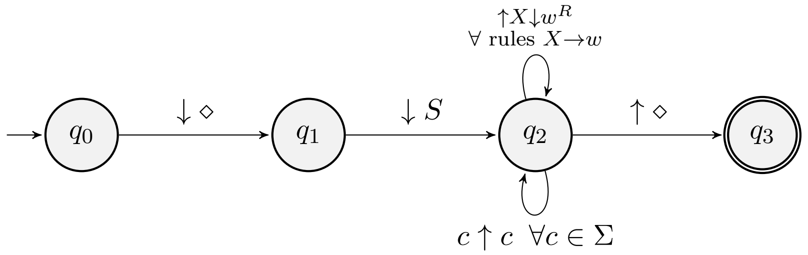
$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[r, "\downarrow \diamond"] & q_1 \arrow[r, "\downarrow S"] & q_2 \arrow[loop above, "\substack{\uparrow X \downarrow w^R \\ \forall \text{ rules } X\to w}"] \arrow[loop below, "c \uparrow c \ \ \forall c \in \Sigma"'] \arrow[r, "\uparrow \diamond"] & q_3
\end{tikzcd}$$

Figure 12.5: A sketch of the PDA P recognizing the language of the CFG G.

Let us now prove the reverse direction: that every PDA can be converted into a CFG.

Theorem 12.3. *Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA. Then there is a CFG $G = (V, \Sigma, R, S)$ such that $L(G) = L(P)$.*

The proof of this theorem is a bit long, but the idea is not too scary. The idea is to first ensure that P only accepts when its stack is empty (which is a modification we've previously seen how to do), and then to use one variable for each pair of states p and q of P, and this variable $X_{p,q}$ will generate all the strings that can cause P to go from p to q when the stack is empty at both the beginning and end. The trick is that for any path of P that pushes a symbol c to the stack, the path must later pop c from the stack (or else we don't care about the path because it can't be an accepting path). We can then split the path into shorter paths, all of which will have empty stack at the beginning and end (one of them will be the path that starts right after pushing c and ends right before popping c). This lets us almost ignore the stack entirely (since it always starts and ends empty).

Proof. As we've seen, we can modify any PDA so that it always clears its stack before accepting. For this reason, we may assume that P has this property (if not, we could first apply this transformation to P and then proceed with the rest of this proof). This means that P only accepts if it ends up at an accept state while the stack and the input stream are both empty.

For each pair of states $p, q \in Q$, we will have a variable $X_{p,q} \in V$. Our goal will be that this variable generates all the strings $w \in \Sigma^*$ that can cause P to go from state p to state q while leaving the stack unchanged. That is, a string w will be generated by $X_{p,q}$ if there is a sequence of transitions in P such that if we start at p with empty stack, we will reach q with empty stack while reading exactly the string w from the input (we allow P to push and pop from the stack along the way, so long as the stack is empty at the beginning and at the end).

We will also have an additional start variable S, together with the rules $S \to X_{q_0,p}$ for each $p \in F$. This corresponds to saying that a string is generated by S if and only if the state P can go from q_0 to some accept state p with the stack being empty at the start and end, which is exactly what we want.

It remains to describe the rules for the states $X_{p,q}$. First, we add the rules $X_{p,p} \to \epsilon$ to the grammar G for each $p \in Q$. For states $p, q \in Q$ such that there is an ϵ-transition from p to q in P, we also include the rule $X_{p,q} \to \epsilon$ in our grammar G. For each $p, p', q \in Q$, we include the rule $X_{p,q} \to X_{p,p'} X_{p',q}$. For any $p, q \in Q$ and any $c \in \Sigma$ such that there is a transition from p to q that reads c from the input, we include the rule $X_{p,q} \to c$. Finally, for any $p, p', q, q' \in Q$ and any $c \in \Gamma$ such that there is a transition rule from p to p' that pushes c and a transition rule from q' to q that pops c, we include the rule $X_{p,q} \to X_{p',q'}$.

We now argue that the variables $X_{p,q}$ in the grammar G generate exactly the strings $w \in \Sigma^*$ for which P can go from p to q while reading w with the stack being empty at the beginning and end. There are two directions: showing that each w generated by $X_{p,q}$ can cause P to go from p to q in this way, and showing that each string w that can cause P to go from p to q in this way can be generated by $X_{p,q}$.

We tackle the former direction first. We want to show that for any string w and any $p, q \in Q$ such that there is a sequence of transitions in P which go from p to q while reading w (with the stack empty at the beginning and end), the variable $X_{p,q}$ can generate w in G. We prove this by induction on the length of the sequence of transitions from p to q in P. If this sequence is empty, we must have $p = q$ and $w = \epsilon$, in which case $X_{p,q}$ can generate w using the rule $X_{p,p} \to \epsilon$. Otherwise, consider the first transition from p in the path to q. Suppose this transition goes to state p'. Since the stack starts out empty, this transition cannot pop from the stack.

If it is an ϵ transition, then w can cause P to go from p' to q, and this is a shorter sequence of transitions; by our induction hypothesis, we have $X_{p',q} \Rightarrow_G^* w$; we can then use the rules $X_{p,q} \to X_{p,p'}X_{p',q}$ and $X_{p,p'} \to \epsilon$ to generate w from $X_{p,q}$. If the transition from p to p' reads $c \in \Sigma$ from the input, then we have $w = cx$ for some string $x \in \Sigma^*$, and P has a sequence of transitions from p' to q which reads x and is shorter than the transition from p to q of w; by our induction hypothesis, we must have $X_{p',q} \Rightarrow_G^* x$, and together with the rules $X_{p,q} \to X_{p,p'}X_{p',q}$ and $X_{p,p'} \to c$ we can generate w from $X_{p,q}$.

Finally, if the transition from p to p' pushes $c \in \Gamma$ onto the stack, then because the stack ends up empty when we reach q, the symbol c must be popped at some point along the way, say in a transition from q' to q''. Let x be the string of symbols read from the input along the path from p' to q', and let y be the string of symbols read from the input along the path from q'' to q. Then $w = xy$, and since these paths are shorter and maintain empty stacks at the beginning and end, the induction hypothesis gives us $X_{p',q'} \Rightarrow_G^* x$ and $X_{q'',q} \Rightarrow_G^* y$. We can then use the rule $X_{p,q} \to X_{p,q''}X_{q'',q}$ together with the rule $X_{p,q''} \to X_{p',q'}$ to generate w from $X_{p,q}$. In all cases, we managed to generate w from $X_{p,q}$, completing the induction argument.

In the reverse direction, we have to show that if a string $w \in \Sigma^*$ is generated by a variable $X_{p,q}$ in G for some $p, q \in Q$, then there is a sequence of transitions in P that starts at p, reads w, ends at q, and has empty stack at the beginning and end. We can once again do this by induction on the length of the derivation of w from $X_{p,q}$. That is, let w be a string that has a derivation from $X_{p,q}$, and suppose that all strings with shorter derivations indeed correspond to such paths in P. We need to show that w also corresponds to such a path.

If the first rule of the derivation of w is $X_{p,q} \to \epsilon$, then $w = \epsilon$ and either $p = q$ or else there is an ϵ-transition from p to q; in both cases, the path in P from p to q is evident. If the first rule of the derivation of w is $X_{p,q} \to c$ for some $c \in \Sigma$, then $w = c$ and there is a transition from p to q in P that reads c from the input, so the path in P is again clear. If the first rule of the derivation is $X_{p,q} \to X_{p,p'}X_{p',q}$ for some $p' \in Q$, then we must have $X_{p,p'} \Rightarrow_G^* x$ and $X_{p',q} \Rightarrow_G^* y$ with $xy = w$, and these derivations must be shorter that of w; by the induction hypothesis, this means there is a path in P that reads x and goes from p to p' while clearing the stack, and another path in P that reads y and goes from p' to q while clearing the stack; concatenating these paths together gives a path from p to q that reads w and clears the stack.

Finally, if the first rule of the derivation of w is $X_{p,q} \to X_{p',q'}$, then there must be some $c \in \Gamma$ such that there is a transition from p to p' in P that pushes c and a transition from q' to q in P that pops c. Also, we must have $X_{p',q'} \Rightarrow_G^* w$, and this derivation must be shorter than the derivation of w from $X_{p,q}$. By our induction hypothesis, there must be a path in P that goes from p' to q' while reading w and clearing the stack; by appending the transition from p to p' (which pushes c) before this path and appending the transition from q' to q (which pops c) after this path, we get a path from p to q in P which reads w and clears the stack.

This concludes our induction proof, and hence we conclude that all variables $X_{p,q}$ in G generate exactly the strings w that can cause P to go from p to q while clearing the stack. Since the start variable S of G has rules $S \to X_{q_0,p}$ for $p \in F$, the start variable must generate exactly the strings that can cause P to go from q_0 to some accept state while clearing the stack, which is exactly the set of strings P accepts; hence $L(G) = L(P)$, as desired. $\square$