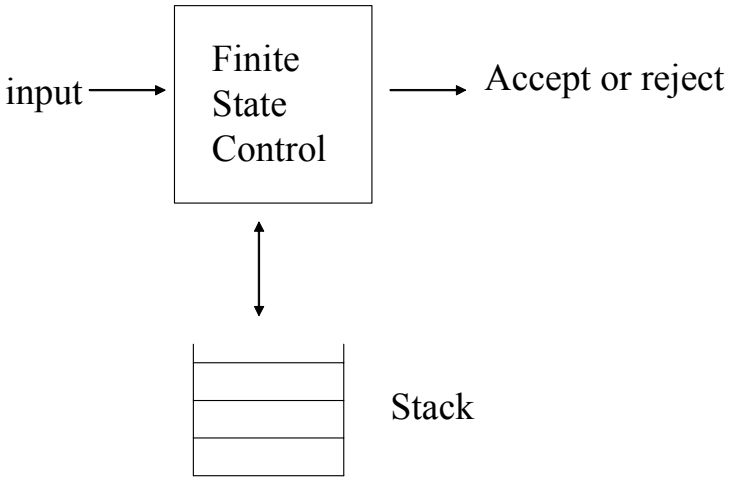
Pushdown Automata
CS351

Just as a DFA was equivalent to a regular expression, we have a similar analogy for the context-free grammar. A **pushdown automata (PDA)** is equivalent in power to context-free grammars. We can give either a context-free grammar generating a context-free language, or we can give a pushdown automaton. Certain languages are more easily described in forms of one or the other.

A pushdown automata is quite similar to a nondeterministic finite automata, but we add the extra component of a stack. The stack is of infinite size, and this is what will allow us to recognize some of the non-regular languages.

We can visualize the pushdown automata as the following schematic:



The finite state control reads inputs, one symbol at a time. The pushdown automata makes transitions to new states based on the input symbol, current state, and the top of the stack. In the process we may manipulate the stack. Ultimately, the control accepts or rejects the input.

In one transition the PDA may do the following:

1. Consume the input symbol. If ε is the input symbol, then no input is consumed.
2. Go to a new state, which may be the same as the previous state.
3. Replace the symbol at the top of the stack by any string.
    a. If this string is ε then this is a pop of the stack
    b. The string might be the same as the current stack top (does nothing)
    c. Replace with a new string (pop and push)
    d. Replace with multiple symbols (multiple pushes)

As an informal example, consider the language $L = \{0^n1^n \mid n \geq 0\}$. A finite automaton is unable to recognize this language because it cannot store an arbitrarily large number of values in its finite memory. A PDA is able to recognize this language because it can use its stack to store the number of 0's it has seen. The unlimited nature of the stack allows the DPA to store numbers of unbounded size. Here is an informal description of the PDA to describe this language:

1. As each 0 is read, push it onto the stack
2. As soon as 1's are read, pop a 0 off the stack
3. If reading the input is finished exactly when the stack is empty, accept the input else reject the input

Note that this description of the PDA is deterministic. However, in general the PDA is nondeterministic. This feature is crucial because, unlike finite automata, nondeterminism adds power to the capability that a PDA would have if they were only allowed to be deterministic. For example consider the following language, which requires a nondeterministic PDA to describe:

Consider the language $L = \{ ww^R \mid w \text{ is in } (0+1)^* \}$

This language corresponds to even-length palindromes. Informally we can design a PDA to accept this language as follows:

1. Start in state q_0 that represents the state where we haven't yet seen the reversed part of the string. While in state q_0 we read each input symbol and push them on the stack.
2. At any time, assume we have seen the middle; i.e. "fork" off a new branch that assumes we have seen the end of w. We signify this choice by spontaneously going to state q_1. This behaves just like a nondeterministic finite automaton; we'll continue in both the forked-branch and the original branch. One of these branches may die, but as long as one of them reaches a final state we accept the input.
3. In state q_1 compare input symbols with the top of the stack. If match, pop the stack and proceed. If no match, then the branch of the automaton dies.
4. If we empty the stack then we have seen ww^R and can proceed to a final accepting state.

Formal Definition of the Pushdown Automaton

Formally a PDA is defined as seven components:

$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$.

Q = finite set of states, like the finite automaton
Σ = finite set of input symbols, the alphabet
Γ = finite stack alphabet, components we are allowed to push on the stack
δ = transition function, which takes the triple: $\delta(q, a, X)$ where

q = state in Q
a = input symbol in Σ
X = stack symbol in Γ

The output of δ is the finite **set** of pairs (p, γ) where p is a new state and γ is a new string of stack symbols that replaces X at the top of the stack. If $\gamma = \varepsilon$ then we pop the stack, if $\gamma = X$ the stack is unchanged, and if $\gamma = YZ$ then X is replaced by Z and Y is pushed on the stack. Note the new stack top is to the left end.

q_0 = start state
Z_0 = start symbol. Initially, the PDA's stack consists of one instance of this start symbol and nothing else. We can use it to indicate the bottom of the stack.

F = Set of final accepting states.

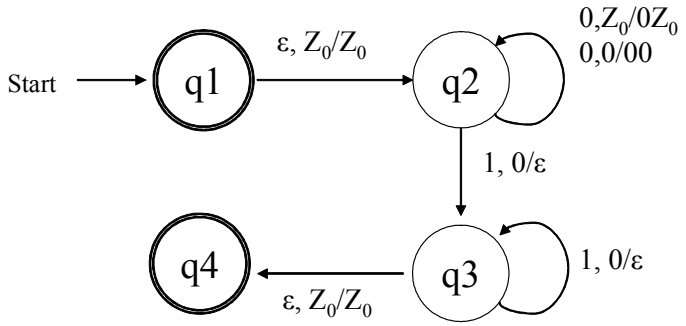
Here is a formal description of the PDA that recognizes $L = \{0^n1^n \mid n \geq 0\}$.

$Q = \{ q_1, q_2, q_3, q_4 \}$
$\Sigma = \{0, 1\}$
$\Gamma = \{0, Z_0\}$
F = {q1, q4}
And δ is described by the following table:

Input:	**0**	**0**	**1**	**1**	**ε**	**ε**
Stack:	**0**	**Z_0**	**0**	**Z_0**	**0**	**Z_0**
→q_1						$\{(q_2, Z_0)\}$
q_2	$\{(q_2, 00)\}$	$\{(q_2, 0Z_0)\}$	$\{(q_3, \varepsilon)\}$			
q_3			$\{(q_3, \varepsilon)\}$			$\{(q_4, Z_0)\}$
→q_4						

Be careful of the notation; when we go to $(q_2, 00)$ then this is pushing an extra 0 onto the stack (which initially contained a zero). When we go to state (q_2, Z_0) we are keeping Z_0 on the top of the stack.

We can also use a graphical notation to describe this PDA;



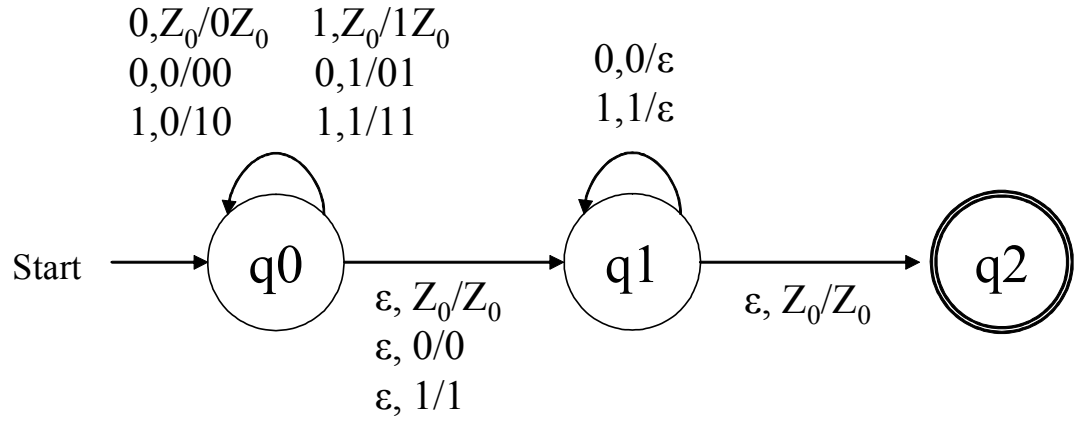
This format uses dark circles to indicate final states just like a normal automaton. Arrows indicate transitions to new states. The edges use the format:

Input-Symbol, Top-of-Stack / String-to-replace-top-of-stack

This means that 0,0/00 is taken when the input is 0 and the top of stack is 0, replacing the top of the stack with 00 (i.e. we push a single 0). 1,0/ε is taken when the input is 1 and the top of stack is 0, replacing the top of stack with ε (i.e. we pop the 0 off).

The only thing this graphical format doesn't tell us is which stack symbol is the start symbol. But this is going to be Z_0 unless indicated otherwise.

Here is the graphical description of the PDA that accepts the language
$L = \{ ww^R \mid w \text{ is in } (0+1)^* \}$



This works by staying in state q_0 when we are reading w. Each time we push the input symbol on the stack. Every time we "guess" that we have reached the end of w and are beginning w^R by going to q_1 on an epsilon-transition. In state q_1 we pop off each 0 or 1 we encounter that matches the input. Any other input will "die" for this branch of the PDA. If we ever reach the bottom of the stack, we go to an accepting state.

Instantaneous Descriptions of a PDA (ID's)

For a FA, the only thing of interest about the FA is its state. For a PDA, we want to know its state and the entire content of its stack. Often the stack is one of the most useful pieces of information, since it is not bounded in size.

We can represent the instantaneous description (ID) of a PDA by the following triple (q,w,γ):

1. q is the state
2. w is the remaining input
3. γ is the stack contents

By convention the top of the stack is shown at the left end of γ and the bottom at the right end.

Moves of a PDA

To describe the process of taking a transition, we can adopt a notation similar to δ and δ^ like we used for DFA's. In this case we use the "turnstile" symbol $\vdash$ which is used as:

$$(q, aw, X\beta) \vdash (p, w, \alpha\beta)$$

In other words, we took a transition such that we went from state q to p, we consumed input symbol a, and we replaced the top of the stack X with some new string α.

We can extend the move symbol to taking many moves:

$\vdash^*$ represents zero or more moves of the PDA.

As an example, consider the PDA that accepted $L = \{ ww^R \mid w \text{ is in } (0+1)^* \}$.
We can describe the moves of the PDA for the input 0110:

$(q_0, 0110, Z_0) \vdash (q_0, 110, 0Z_0) \vdash (q_0, 10, 10Z_0) \vdash (q_1, 10, 10Z_0) \vdash (q_1, 0, 0Z_0) \vdash (q_1, \varepsilon, Z_0) \vdash (q_2, \varepsilon, Z_0)$.

We could have taken other moves rather than the ones above, but they would have resulted in a "dead" branch rather than an accepting state.

Language of a PDA

The PDA consumes input and accepts input when it ends in a final state. We can describe this as:

$$L(P) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha) \} \text{ where } q \in F.$$

That is, from the starting state, we can make moves that end up in a final state with any stack values. The stack values are irrelevant as long as we end up in a final state.

It turns out we can also describe a language of a PDA by ending up with an empty stack with no further input:

$$N(P) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon) \} \text{ where } q \text{ is any state.}$$

That is, we arrive at a state such that P can consume and at the same time empty its stack.

It turns out that we can show the classes of languages that are L(P) for some PDA P is equivalent to the class of languages that are N(P) for some PDA P. This class is also exactly the context-free languages. See the text for the proof.

Equivalence with Context-Free Grammars

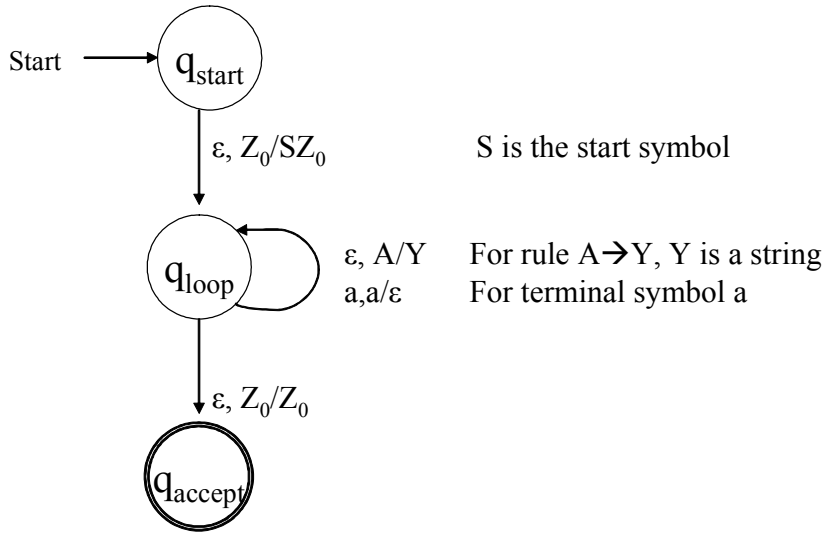
A context-free grammar and pushdown automata are equivalent in power. To prove this, we must show that we can take any CFG and express it as a PDA. Then we must take a PDA and show we can construct an equivalent CFG. We'll show the CFG→PDA process, but only give an overview of the process of going from the PDA→CFG (more details in the textbook).

Theorem: Given a CFG grammar G, then some pushdown automata P recognizes L(G).

Proof idea: The PDA P will work by accepting its input w, if G generates that input, by determining whether there is a derivation for w. Each sentential form of the derivation yields variables and terminals. We will design P to determine whether some series of substitutions using the rules of G leads from the start variable to w.

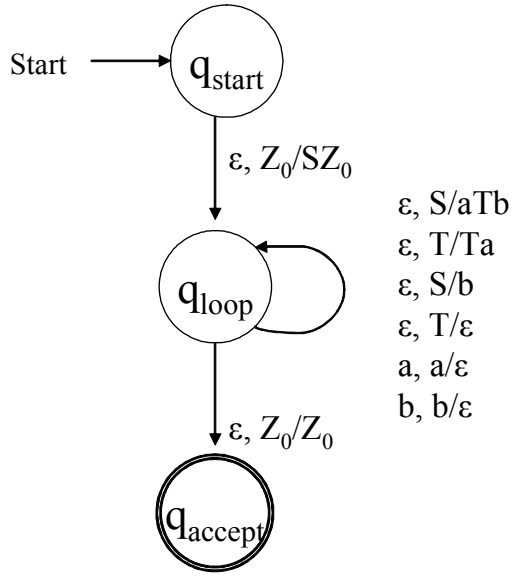
One tricky part arises in dealing with the sentential forms (intermediate strings) of the derivation. These strings consist of variables and terminals. However, the PDA can access only the top symbol on the stack and that might be a terminal instead of a variable. To get around this problem, we match terminal symbols on the stack before the first variable immediately with symbols in the input string.

Here is the state diagram for a PDA we can construct for an arbitrary CFG:



Here is an example for the following grammar:

S→ aTb | b
T → Ta | ε



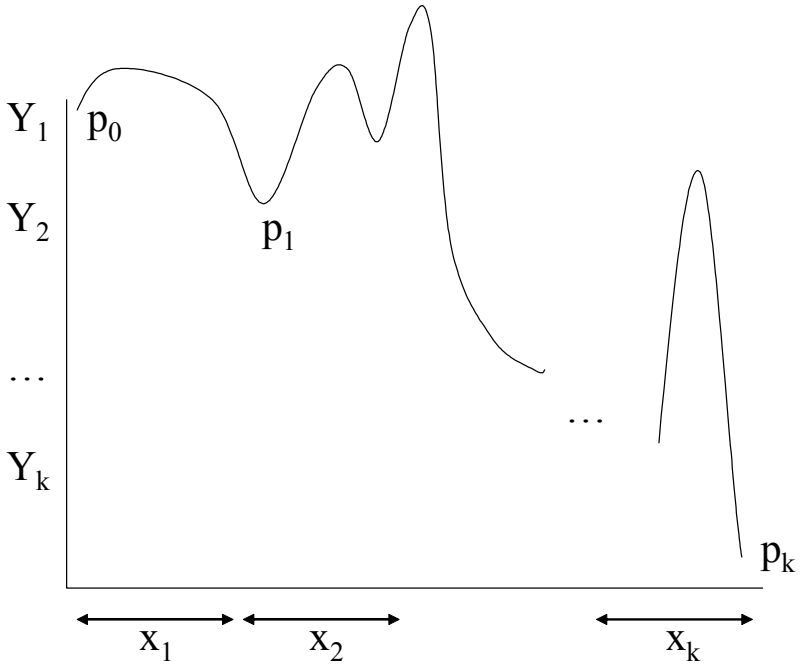
For example, given the string "aab" derived by S ⇒ aTb ⇒ aTab ⇒ aab
We have the corresponding moves:

(q_s, aab, Z_0) ├ (q_{loop}, aab, SZ_0) ├ (q_{loop}, aab, $aTbZ_0$) ├ (q_{loop}, ab, TbZ_0) ├ (q_{loop}, ab, $TabZ_0$) ├ (q_{loop}, ab, abZ_0) ├ (q_{loop}, b, bZ_0) ├ (q_{loop}, ε, Z_0) ├ (q_{accept}, ε, Z_0)

Theorem: Given a PDA P, then some CFG grammar G recognizes L(P).

This direction is necessary to show that both CFG's and PDA's are equivalent. We will only give a high level overview of how the proof is constructed; it is somewhat more complex then the direction from CFG to PDA because it is harder to "program" a grammar than it is to program a PDA.

The idea behind the proof is to use the version of the PDA that accepts the input when the stack is empty. This means that the PDA may push values on the stack, but eventually on the path to accepting the input, it must pop off all of the stack symbols. For example, if the stack contains $Y_1Y_2\ldots Y_k$ then there must be some states p_0, p_1, … on the path to the final state where the stack contains no elements such that we remove elements from the stack:



Here, input x_1 is read while Y_1 is popped, and a similar effect for x_2. Note that the net effect is the pop and it might require many intermediate moves.

Using this idea of the popping to an empty stack, we can construct an equivalent grammar to a PDA that uses variables for a change in state from p to q when Y has been replaced by ε on the stack.

This event is represented by the variable using the composite symbols [pXq]. This is a single variable in the grammar (e.g., A→aA we might have [pXq] → a[pXq]).

The start symbol of G can then be described as:

$S \rightarrow [q_0Z_0p]$.

We then need to construct all variables to fill out the grammar, where variable [qZp] corresponds to $(q, wx, ZY) \vdash^* (p, x, Y)$ and also $(q, w, Z) \vdash^* (p, \varepsilon, \varepsilon)$.

Here are a few examples:

PDA goes from q to p on input a, pops Z:	$[qZp] \rightarrow a$
PDA goes from q to p on input a, replaces top of stack with new value:	$[qZp] \rightarrow a[rYp]$
PDA goes from q to p on input a, replaces top of stack with two values:	$[qZp] \rightarrow a[rYr_1][r_1Y_2p]$

In the previous two examples we actually need to check all states $r_1 \ldots r_k$.

Deterministic Pushdown Automata

A DPDA is simply a pushdown automata without nondeterminism. That is, we are not allowed to have epsilon transitions or multiple identical inputs leading to different states. We will only be in one state at a time.

The deterministic pushdown automata is not as powerful as the nondeterministic pushdown automata. This machine accepts a class of languages somewhere between regular languages and context-free languages. For this reason, the DPDA is often skipped as a topic; however, in practice, the DPDA can be useful since determinism is much easier to implement. Parses in programs such as YACC are actually implemented using a DPDA.

This means the question really becomes what do we want to express in something like a programming language, and how might a DPDA limit what we can express.

A DPDA can accept all regular languages and all unambiguous context-free languages. Acceptance by empty stack is also difficult for a DPDA. These conditions are typically satisfactory for a programming language, but not satisfactory for something like representing English sentences.