UNIT-VI

PUSHDOWN AUTOMATA

The context free languages have a type of automaton that defined them. This automaton, called a "pushdown automaton", is an extension of the nondeterministic finite automaton with ε – transitions, which is one of the ways to define the regular languages.

The pushdown automaton is essentially an ε – NFA with the addition of a stack. The stack can be read, pushed and popped only at the top, just like the "stack" data structure.

We define two different versions of the pushdown automaton: one that accepts by entering an accepting state, like finite automata do and another version that accepts by emptying its stack, regardless of the state it is in. we show that these two variations accept exactly the context free languages i.e. grammars can be converted to equivalent pushdown automata and vice-versa.

We also consider briefly the subclass of pushdown automata that is deterministic. These accept all the regular languages, but only a proper subset of the CFL's.

DEFINITION OF PUSHDOWN AUTOMATA

Informal Introduction

- The pushdown automaton is in essence a nondeterministic finite automaton with ε – transitions permitted and one additional capability: a stack on which it can store a string of "stack symbols".
- The presence of a stack means that unlike finite automaton, the pushdown automaton can remember an infinite amount of information.
- However, unlike a general purpose computer, which also has the ability to remember arbitrarily large amounts of information, the pushdown automaton can only access the information on its stack in a first in first out away.
- As a result, there are languages that could be recognized by some computer program, but are not recognizable by any pushdown automaton.
- In fact pushdown automata recognize all and only the context free languages. While there are many languages that are context free, including some we have seen that are not regular languages, there are also some simple-to-describe languages that are not context free.
- An example of non context free language is $\{0^n1^n2^n \mid n \geq 1\}$, the set of strings consisting of equal groups of 0's 1's and 2's

MODEL

- A pushdown automaton is essentially a finite automaton with a stack data structure is represented in the figure 6.1:

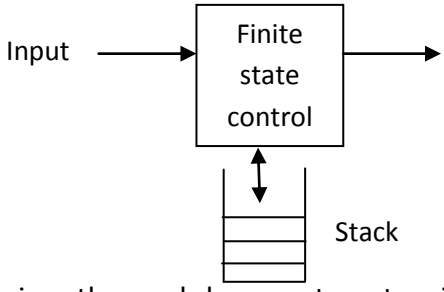


Figure 6.1: A pushdown automata with stack

❖ We can view the pushdown automaton informally as the device suggested in figure 6.1. finite state control reads inputs, one symbol at a time. The pushdown automaton is allowed to observe the symbol at the top of the stack and to base its transition on its current state, input symbol, and the symbol at the top of stack.

Example 6.1:

❖ Let us consider the language, L_{wwr} = { ww^R | w is in (0 + 1)* }, often referred to as "w-w-reversed" is the even-length palindromes over alphabet {0, 1}. It is the CFL generated by the grammar given below (with productions P → 0 and P → 1 omitted):

P → ε
P → 0
P → 1
P → 0P0
P → 1P1

❖ We can design an informal pushdown automaton accepting L_{wwr}, as follows.

- Start in a state q_0 that represents a "guess" that we have not yet seen the middle i.e. we have not seen the end of the string w that is to be followed by its own reverse. While in state q_0, we read symbols and store them on the stack by pushing a copy of each input symbol on to the stack, in turn.
- At any time, we may guess that we have seen the middle i.e. the end of w. At this time, w will be on the stack, with the right end of w at the top and the left end at the bottom. We signify this choice by spontaneously going to state q_1. Since the automaton is nondeterministic, we actually make both guesses: we guess we have seen the end of w, but we also stay in state q_0 and continue to read inputs and store them on the stack.
- Once in state q_1 we compare input symbols with the symbol at the top of the stack. If they match, we consume the input symbol, pop the stack, and proceed. If they do not match, we have guessed wrong; our guessed w was not followed by w^R. This branch dies, although other branches of the nondeterministic automaton may survive and eventually lead to acceptance.
- If we empty the stack, then we have indeed seen some input w followed by w^R. We accept the input that was read up to this point.

FORMAL DEFINITION OF PUSHDOWN AUTOMATA

- Formal definition for *pushdown automata* (PDA) involves seven components. We write the specification of a PDA P as follows:

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

- The components have the following meanings:
  - Q: a finite set of *states*, like the states of a finite automata
  - Σ: a finite set of *input symbols*, also analogous to the corresponding component of a finite automaton.
  - Γ: a *finite stack alphabet*. It is the set of symbols that we are allowed to push onto the stack.
  - δ: the *transition function*. As for a finite automaton, δ governs the behavior of the automaton. Formally, δ takes as argument a triple δ(q, a, X), where:
    1. q is the state in Q
    2. a is either an input symbol in Σ or a = ε, the empty string, which is assumed not to be an input symbol.
    3. X is a stack symbol, which is a member of Γ.

    The output of δ is a finite set of pairs (p, γ), where p is the new state and γ is the string of stack symbols that replaces X at the top of stack.

    For instance if γ = ε, then stack is popped. If γ = X, then the stack is unchanged, and if γ = YZ, then X is replaced by Z and Y is pushed onto the stack.
  - q_0: the *start state*. The PDA is in this state before making any transitions.
  - Z_0: the *start symbol*. Initially, the PDA's stack consists of one instance of this symbol, and nothing else.
  - F: the set of *accepting states*, or *final states*.

Example 6.2:

- Let us design the PDA P to accept the language L_{wwr} of example 6.1. First there are a few details not present in that example that we need to understand in order to manage the stack properly. We shall use a stack symbol Z_0 to mark the bottom of the stack.
- We need to have this symbol present so that, after we pop w off the stack and realize that we have seen ww^R on the input, we still have something on the stack to permit us to make a transition to the accepting state, q_2.
- Thus our PDA for L_{wwr} can be described as

$$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

- Where δ is defined by the following rules:

- $\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$ and $\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$. One of these rules applies initially, when we are in state q_0 and we see the start symbol Z_0 at the top of the stack. We read the first input and push it on to the stack, leaving Z_0 below to mark the bottom.
- $\delta(q_0, 0, 0) = \{(q_0, 00)\}$, $\delta(q_0, 0, 1) = \{(q_0, 01)\}$, $\delta(q_0, 1, 0) = \{(q_0, 10)\}$, and $\delta(q_0, 1, 1) = \{(q_0, 11)\}$. These four, similar rules allow us to stay in state q_0 and read inputs, pushing each onto the top of stack and leaving the previous top stack symbol alone.
- $\delta(q_0, \varepsilon, Z_0) = \{(q_1, Z_0)\}$, $\delta(q_0, \varepsilon, 0) = \{(q_1, 0)\}$, $\delta(q_0, \varepsilon, 1) = \{(q_1, 1)\}$. These three rules allow P to go from state q_0 to state q_1 spontaneously (on ε input), leaving intact whatever symbol is at the top of the stack.
- $\delta(q_1, 0, 0) = \{(q_1, \varepsilon)\}$, and $\delta(q_1, 1, 1) = \{(q_1, \varepsilon)\}$. Now in state q_1 we can match input symbols against the top symbols on the stack, and pop when the symbols match.
- $\delta(q_1, \varepsilon, Z_0) = \{(q_2, Z_0)\}$. Finally if we expose the bottom-of-stack marker Z_0 and we are in state q_1, then we have found an input of the form ww^R. We go to state q_2 and accept.

GRAPHICAL NOTATION FOR PDA

❖ The transition diagram for PDA contains:

- The nodes correspond to the states of the PDA.
- An arrow labeled Start indicates the start state, and doubly circled states are accepting, as for finite automata.
- The arcs correspond to transitions of the PDA in the following sense.
  1. An arc labeled **a, X/α** from state q to state p means that δ(q, a, X) contains the pair (p, α), perhaps among other pairs.
  2. That is, the arc label tells what input is used, and also gives the old and new tops of the stack.

***Note*:** The only thing that the diagram does not tell us is which stack symbol is the start symbol. Conventionally, it is Z_0, unless we indicate otherwise.

❖ The PDA of example 6.1 is represented by the diagram shown in figure 6.2:

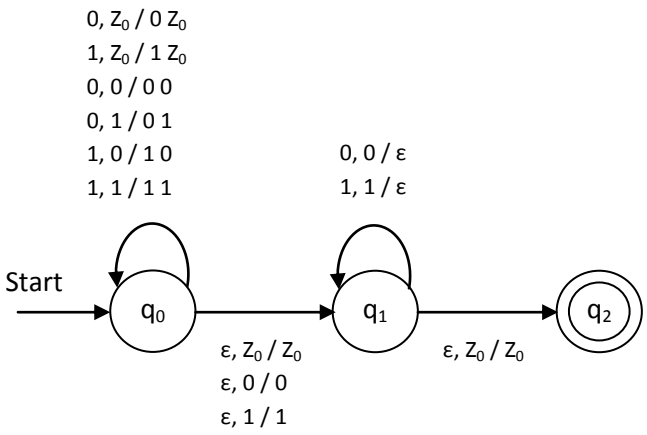


Figure 6.2: Representing a PDA as a generalized transition

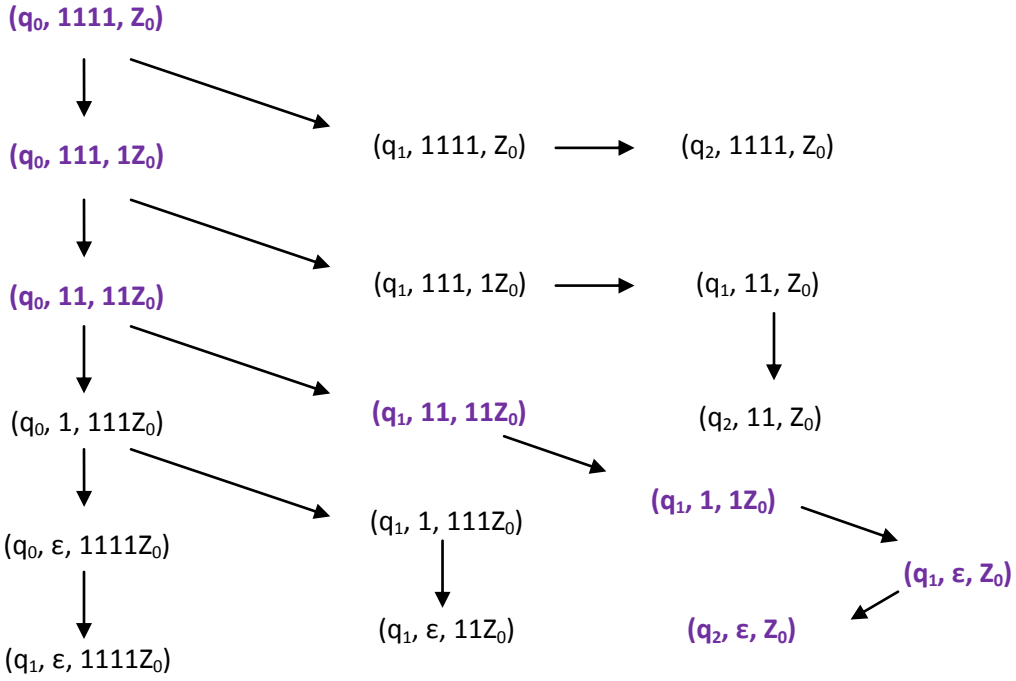
- Unlike Finite Automata, the Pushdown Automata involves both the state and the contents of the stack.
- Thus we shall represent the configuration of a PDA by a triple (q, w, γ), where
  - q is the state
  - w is the remaining input and
  - γ is the stack contents
- Conventionally we show the top of the stack at the left end of γ and the bottom at the right end. Such a triple is called as *Instantaneous Description* or ID, of the PDA.
- For PDA's we need a notation that describes changes in the state, the input and stack. Thus we adopt the "turnstyle" notation for connecting pairs of ID's that represent one or many moves of a PDA.
- Let P = (Q, Σ, Γ, δ, q_0, Z_0, F) be a PDA. Define $\overset{\vdash}{}_P$ or ⊢ when P is understood as follows.
- Suppose δ(q, a, X) contains (p, α). Then for all strings w in Σ* and β in Γ*:

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

- This move reflects the idea that by consuming *a* (which may be ε) from the input and replacing X on top of stack by α, we can go from state q to state p.
- Note that what remains on the input, w, and what is below the top of stack, β, do not influence the action of the PDA; they merely carried along, perhaps to influence events later.

Example 6.3:

- Let us consider the action of the PDA of example 6.2 on the input 1111. Since q_0 is the start state and z_0 is start symbol, the initial ID is (q_0, 1111, Z_0).on this input, the PDA has the opportunity to guess wrongly several times. The entire sequence of ID's that the PDA can reach from the initial ID's that the PDA can reach from the initial ID (q_0, 1111, Z_0) is shown in figure given below: (arrows represents the ⊢ relation)



- From the initial ID, there are two choices of move. The first guesses that the middle has not been seen and leads to ID (q_0, 111, $1Z_0$). In effect, a 1 has been removed from the input and pushed on to the stack.
- The second choice from the initial ID guesses that the middle has been reached. Without consuming input, the PDA goes to state q_1, leading to the ID (q_1, 1111, Z_0). Since the PDA may accept if it is in state q_1 and sees Z_0 on top of its stack, the PDA goes from there to ID (q2, 1111, Z_0).
- That ID is not exactly an accepting ID, since the input has not been completely consumed. Had the input been ε rather than 1111, the same sequence of moves would have led to ID (q_2, ε, Z_0), which would show that ε is accepted.
- The Pushdown Automata may also guess that it has seen the middle after reading one 1, i.e. when it is in the ID (q_0, 111, $1Z_0$). That guess also leads to failure, since the entire input cannot be consumed.
- The **correct guess**, that middle is reached after reading two 1's gives us the sequence of ID's (q_0, 1111, Z_0) ⊢ (q_0, 111, $1Z_0$) ⊢ (q_0, 11, $11Z_0$) ⊢ (q_0, 11, $11Z_0$) ⊢ (q_0, 1,1 Z_0) ⊢ (q_1, ε, Z_0) ⊢ (q_2, ε, Z_0).
- There are three important principles about ID's and their transitions that we shall need in order to reason about PDA's:
  - If a sequence of ID's (*computation*) is legal for a PDA P, then the computation formed by adding the same additional input string to the end of the input (second component) in each ID is also legal.
  - If a computation is legal for a PDA P, then the computation formed by adding the same additional stack symbols below the stack in each ID is also legal.
  - If a computation is legal for a PDA P, and some tail of the input is not consumed, then we can remove this tail from the input in each ID, and the resulting computation will still be legal.
  - Intuitively, data that P never looks at cannot affect ifs computation.

Exercises:

- Suppose the PDA P = ({q_0, q_1}, {0, 1}, {X, Y, Z}, δ, q_0, Z, {q_1}) has the following transition functions:
  - δ (q_0, 0, Z) = {(q_1, Z)}
  - δ (q_0, 1, Z) = {(q_0, XZ)}
  - δ (q_0, 0, X) = {(q_0, ε)}
  - δ (q_0, 1, X) = {(q_0, XX)}
  - δ (q_1, 0, Z) = {(q_1, YZ)}
  - δ (q_1, 1, Z) = {(q_0, Z)}
  - δ (q_1, 0, Y) = {(q_1, YY)}
  - δ (q_1, 1, Y) = {(q_1, ε)}

Starting from the initial ID show all the reachable ID's when the input w is:

1. 000011
2. 011001
3. 0110110

THE LANGUAGES OF PDA

- We have assumed that a PDA accepts it input by consuming it and entering an accepting state. We call this approach *"acceptance by final state"*.
- There is a second approach to define the language of PDA that has important applications. We may also define for any PDA the language *"accepted by empty stack"*, i.e. the set of strings that cause the PDA to empty the stack, starting from the initial ID.
- These two methods are equivalent, in the sense that a language L has a PDA that accepts it by final state if and only if L has a PDA that accepts it by empty stack.
- However for a given PDA P, the languages that P accepts by final state and empty stack are usually different.
- We shall now show how to convert a PDA accepting L by final state into another PDA that accepts L by empty stack and vice-versa.

ACCEPTANCE BY FINAL STATE

- Let P = (Q, Σ, Γ, δ, q0, Z_0, F) be a PDA. Then L(P), the *language accepted by P by final state*, is

$$\{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \alpha)\}$$

for some state q in F and any stack string α. That is starting in the initial ID with w waiting on input, P consumes w from the input and enters an accepting state.

- The contents of the stack at that time are irrelevant.

Example:

- We have claimed that the PDA of example 6.2 accepts the language L_{wwr}, the language of strings in {0, 1}* that have the form ww^R. Let us see why that statement is true.
- *The proof is an if-and-only-if statement*: the PDA P of example 6.2 accepts string x by final state if and only if x is of the form ww^R.
- (If) This part is easy; we have only to show the accepting computation of P. If $x = ww^R$, then observe that

$(q_0, ww^R, Z_0) \overset{*}{\vdash} (q_0, w^R, w^RZ_0) \vdash (q_1, w^R, w^RZ_0) \overset{*}{\vdash} (q_1, \varepsilon, Z_0) \vdash (q_2, \varepsilon, Z_0)$

- That is one option the PDA has is to read w from its input and store it on its stack, in reverse. Next it goes spontaneously to state q_1 and matches w^R on the input with the same string on its stack, and finally goes spontaneously to state q_2.

- (only-if) This part is harder. First, observe that the only way to enter accepting state q_2 is to be in state q_1 and have Z_0 at the top of the stack. Also, any accepting computation of P will start in state q_0, make one transition to q_1, and never return to q_0.
- Thus it is sufficient to find the conditions on x such that $(q_0, x, Z_0) \overset{*}{\vdash}_P (q_1, \varepsilon, Z_0)$; these will be exactly the strings x that P accepts by final state.

ACCEPTANCE BY EMPTY STACK

- For each PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, we also define

$$N(P) = \{ w \mid (q_0, w, Z_0) \overset{*}{\vdash}_P (q, \varepsilon, \varepsilon)\}$$

for any state q. That is, N(P) is the set of inputs w that P can consume and at the same time empty its stack. N(P) stands for null stack, a synonym for "empty stack".

Example:

- The PDA P of example 6.2 never empties its stack, so $N(P) = \Phi$.
- However a small modification will allow P to accept L_{wwr} by empty stack as well as by final state. Instead of the transition $\delta(q_1, \varepsilon, Z_0) = \{(q_2, Z_0)\}$, use $\delta(q_1, \varepsilon, Z_0) = \{(q_2, \varepsilon)\}$. Now P pops the last symbols off its stack as it accepts, and $L(P) = N(P) = L_{wwr}$.
- Since the set of accepting states is irrelevant, we shall sometimes leave off the last (seventh) component from the specification of a PDA P, if all we care about is the language that P accepts by empty stack. Thus, we would write P as a six – tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0)$

EQUIVALENCE OF ACCEPTANCE BY FINAL STATE AND EMPTY STACK

FROM EMPTY STACK TO FINAL STATE

- We shall show that classes of languages that are L(P) for some PDA P is the same as the class of languages that are N(P) for some PDA P.
- Our first construction shows how to take a PDA P_N that accepts a language L by empty stack and construct a PDA P_F that accepts L by final state.

Theorem:

- If $L = N(P_N)$ for some PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$ then there is a PDA P_F such that $L = L(P_F)$

Proof:

- The idea behind the proof is in figure 6.3. We use a new symbol X_0, which must not be a symbol of Γ, X_0 is both the start symbol of P_F and a marker on the bottom of the stack that lets us know when P_N has reached an empty stack.
- That is if P_F sees X_0 on top of stack, then it knows that P_N would empty its stack on the same input.
- We also need a new start state p_0, whose sole function is to push Z_0, the start symbol of P_N on to the top of the stack and enter state q_0, the start state of P_N.

- Then P_F simulates P_N, until stack of P_N is empty, which P_F detects because it sees X_0 on the top of the stack.
- Finally we need another new state, p_f, which is the accepting state of P_F this PDA transfers to state p_f whenever it discovers that P_N would have emptied its stack.
- The specification of P_F is as follows:

$$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$$

- where δ_F is defined by:
  - $\delta_F(p_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$. In its start state, P_F makes a spontaneous transition to the start state of P_N, pushing its start symbol Z_0 onto the stack.
  - For all states q in Q, inputs a in Σ or a = ε, and stack symbols Y in Γ, $\delta_F(q, a, Y)$ contains all the pairs in $\delta_N(q, a, Y)$.
  - In addition to rule (b), $\delta_F(q, \varepsilon, X_0)$ contains (p_f, ε) for every state q in Q.

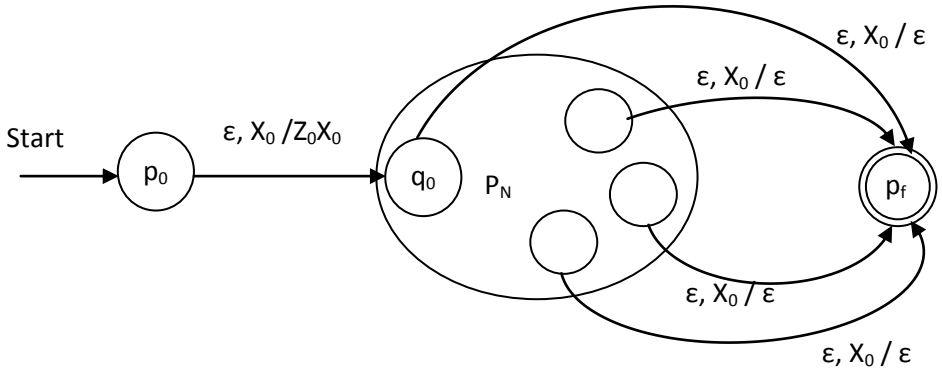


Figure 6.3: P_F simulates P_N and accepts if P_N empties the stack

- We must show that w is in $L(P_F)$ if and only if w is in $N(P_N)$.
- (If) we are given that $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \varepsilon)$ for some state q.
- (only if) the converse requires only that we observe the additional transitions of rules (a) and (c) give us very limited ways to accept w by final state. We must use rule (c) at the last step, and we can only use that rule if the stack of P_F contains only X_0. X_0's ever appear on the stack except at the bottommost position. Furthermore rule (a) is only used at the first step, and it must be used at the first step.

FROM FINAL STATE TO EMPTY STACK

- Now let us go in the appropriate direction: take a PDA P_F that accepts a language L by final state and construct another PDA P_N that accepts L by empty stack.
- The construction is simple and suggested in the figure 6.4. From each accepting state of P_F add a transition on ε to a new state p.
- When in state p, P_N pops its stack and does not consume any input.

- Thus whenever P_F enters an accepting state after consuming input w, P_N will empty its stack after consuming w.

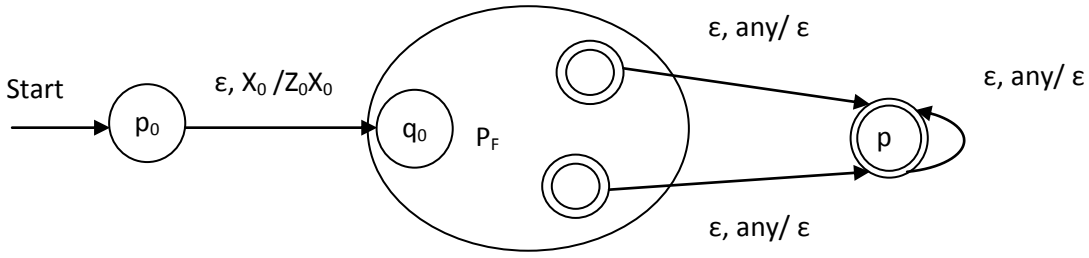


Figure 6.4: P_N simulates P_F and empties its stack when and only when P_N enters an accepting state

- To avoid simulating a situation where P_F empties its stack without accepting, P_N must also use a marker X_0 on the bottom of its stack. The marker is P_N's start symbol, P_N must start in a new state p_0, whose sole function is to push the start symbol of P_F on the stack and go to the start state of P_F.
- The construction is sketched in Fig. 6.4, and we give it formally in the next theorem.

  Theorem:
- Let L be $L(P_F)$ for some PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q0, Z_0, F)$. Then there is a PDA P_N such that $L = N(P_N)$.

  Proof:
- The construction is suggested in figure 6.4. Let

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

- Where δ_N is defined by:
  - $\delta_N(p_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$. We start by pushing the start symbol of P_F on to the stack and going to the start state of P_F.
  - For all states q in Q, input symbols a in Σ or a = ε, and Y in Γ, $\delta_N(q, a, Y)$ contains every pair that is in $\delta_F(q, a, Y)$. That is, P_N simulates P_F.
  - For all accepting states q in F and stack symbols Y in Γ or $Y = X_0$, $\delta_N(q, \varepsilon, Y)$ contains (p, ε). By this rule, whenever P_F accepts, P_N can start emptying its stack without consuming any more input.
  - For all stack symbols Y in Γ or $Y = X_0$, $\delta_N(p, \varepsilon, Y) = \{(p, \varepsilon)\}$. Once in state p, which only occurs when P_F has accepted, P_N pops every symbol on its stack, until the stack is empty. No further input is consumed.
- Now we must prove that w is in $N(P_N)$ if and only if w is in $L(P_F)$.
- The "if" part is a direct simulation and the "only-if" part requires that we examine the limited number of things that the constructed PDA P_N can do.
- (If) suppose $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \alpha)$ for some accepting state q and stack string α.

- (Only if) the only way P_N can empty its stack is by entering state p, since X_0 is sitting at the bottom of the stack and X_0 is not a symbol on which P_F has any moves.
- The only way P_N can enter state p is if the simulated P_F enters an accepting state. The first move of P_N is surely the move given in rule (a). Thus every accepting computation of P_N looks like

$$(p_0, w, X_0) \vdash_{P_N} (q_0, w, Z_0X_0) \overset{*}{\vdash}_{P_N} (q, \varepsilon, \alpha X_0) \overset{*}{\vdash}_{P_N} (p, \varepsilon, \varepsilon)$$

- Where q is an accepting state of P_F. Moreover between ID's (q_0, w, Z_0X_0) and $(q, \varepsilon, \alpha X_0)$, all the moves are moves of P_F.
- In particular X_0 was never the top stack symbol prior to reaching ID $(q, \varepsilon, \alpha X_0)$ (i.e. although α could be ε, in which case P_F can emptied its stack at the same time it accepts). Thus we conclude that the same computation can occur in P_F, without the X_0 on the stack; that is $(q_0, w, Z_0) \overset{*}{\vdash}_{P_N} (q, \varepsilon, \alpha)$.
- Now we can see that P_F accepts w by final state, so w is in $L(P_F)$.

EQUIVALENCE OF PDA'S AND CFG'S

- Now we shall demonstrate that the languages defined by PDA's are exactly the context-free languages.
- The plan of attack is suggested by **figure 6.5**.The goal is to prove that the following three classes of languages:
  - The context – free languages i.e. the languages defined by CFG's.
  - The languages that are accepted by final state by some PDA.
  - The languages that are accepted by empty stack by some PDA.

  are all the same class. We have already shown that (b) and (c) are the same. It turns out to be easiest next to show that (a) and (3) are the same, thus implying the equivalence of all three.

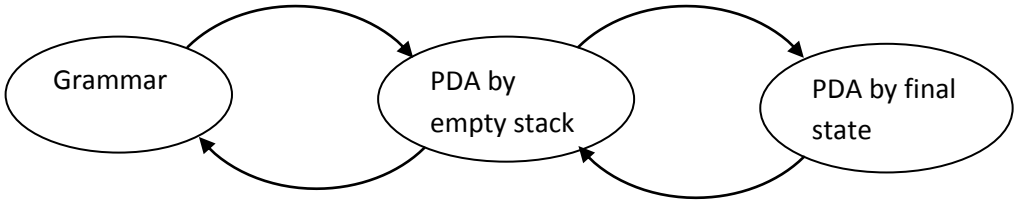


Figure 6.5: Organization of constructions showing equivalence of three ways of defining the CFL's

FROM GRAMMARS TO PUSHDOWN AUTOMATA

- Given CFG G we construct a PDA that simulates the left most derivations of G. Any left sentential form that is not a terminal string can be written as **xAα**, where **A** is the left most variable, **x** is whatever terminals appear to its left, and **α** is the string of terminals and variables that appear to the right of **A**.
- We call **Aα** the tail of this left sentential form. If a left sentential form consists of only terminals, then its tail is **ε**.
- The idea behind the construction of PDA from a grammar is to have a PDA simulate the sequence of left sentential forms that the grammar uses to generate a given terminal string w.
- The tail of each sentential form **xAα** appears on the stack, with **A** at the top. At that time, **x** will be represented by our having consumed **x** from the input, leaving whatever of w follows its prefix **x**. i.e. if w = xy, then y will remain on the input.
- Suppose the PDA is in an ID (q, y, Aα), representing left sentential form **xAα**. It guesses the production to use to expand **A**, say **A → β**. The move of the PDA is to replace **A** on the top of the stack by β, entering ID (q, y, βα). Note that there is only one state, q, for this PDA.
- Now (q, y, βα) may not be a representation of the next left sentential form, because β may have a prefix of terminals. In fact, β may have no variables at all and α may have a prefix of terminals.
- Whatever terminals appear at the beginning of βα need to be removed, to expose the next variable at the top of the stack. The terminals are compared against the next input symbols, to make sure our guesses at the left most derivation of input string w are correct; if not this branch of the PDA dies.
- If succeeded in this way to guess a leftmost derivation of w, then we shall eventually reach the left sentential form w. At that point all the symbols on the stack have either been expanded (if they are variables) or matched against the input (if they are terminals). The stack is empty and we accept by empty stack.
- The above informal construction can be made precise as follows. Let G = (V, Σ, Q, S) be a CFG. Construct a PDA P that accepts L(G) by empty stack as follows:

$$P = (\{q\}, \Sigma, V \cup \Sigma, \delta, q, S)$$

- Where transition function δ is defined by:
  - For each variable A.

    $\delta(q, \varepsilon, A) = \{(q, \beta) \mid A \rightarrow \beta \text{ is a production of P}\}$

  - For each terminal a, $\delta(q, a, a) = \{(q, \varepsilon)\}$.

Example:

- Let us consider the expression grammar given below to a PDA.

$$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$E \rightarrow I \mid E * E \mid E + E \mid (E)$$

- The set of terminals for the PDA is {a, b, 0, 1, (,), +, *}. These eight symbols and the symbols I and E form the stack alphabet.
- The transition function for the PDA is:
  - δ (q, ε, I) = {(q, a), (q, b), (q, Ia), (q, Ib), (q, I0), (q, I1)}.
  - δ (q, ε, E) = {(q, I), (q, E*E), (q, E+E), (q, (E))}.
  - δ (q, a, a) = {(q, ε)}; δ (q, b, b) = {(q, ε)}; δ (q, a, a) = {(q, ε)}; δ (q, 0, 0) = {(q, ε)}; δ (q, 1, 1) = {(q, ε)}; δ (q, (, () = {(q, ε)}; δ (q,),)) = {(q, ε)}; δ (q, *, *) = {(q, ε)}; δ (q, +, +) = {(q, ε)};
- Note that (a) and (b) come from rule (a), while the eight transitions of (c) come from rule (b). Also δ is empty except as defined by (a) through (c).

FROM PDA'S TO GRAMMARS:

- Now we complete the proofs of equivalence by showing that for every PDA P, we can find a CFG G whose language is the same language that P accepts by empty stack.
- The idea behind the proof is to recognize that the fundamental event in the history of a PDA's processing of a given input is the net popping of one symbol off the stack, while consuming some input.
- A PDA may change state as it pops stack symbols, so we should also note the state that it enters when it finally pops a level off its stack.
- The figure 6.6 below suggests how we pop a sequence of symbols $Y_1, Y_2 \ldots Y_k$ off the stack. Some input x_1 read while Y_1 is popped.

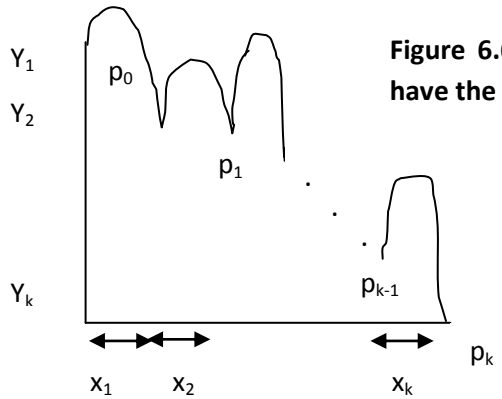


Figure 6.6: A PDA makes a sequence of moves that have the net effect of popping a symbol off the stack

- We should emphasize that the "pop" is the net effect of (possibly) many moves. For example, the first move may change Y_1 to some other symbol Z. The next move may replace Z by UV; later moves have the effect of popping U, and then other moves pop V. The net

effect is that Y_1 has been replaced by nothing i.e. it has been popped, and all the input symbols consumed so far constitute x_1.

- We also shown in the figure 6.6 the net change of state. We suppose that the PDA starts out in state p_0, with Y_1 at the top of the stack. After all the moves whose net effect is to pop Y_1, the PDA is in state p_1.
- It then proceeds to (net) pop Y_2, while reading input string x_2 and winding up, perhaps after many moves, in state p_2 with Y_2 off the stack.
- The computation proceeds until each of the symbols on the stack is removed.
- Our construction of an equivalent grammar uses variables each of which represents an event consisting of:
    1. The net popping of some symbol X from the stack, and
    2. A change in state from some p at the beginning to q when X has finally been replaced by ε on the stack.
- We represent such a variable by the composite symbol [pXq]. Remember that this sequence of characters is our way of describing one variable; it not five grammar symbols.
- The formal construction is given by the theorem given below:

Theorem 6(a):

- Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ be a PDA. Then there is a context free grammar G such that L(G) = N(P).

Proof:

- We shall construct G = (V, Σ, R, S), where the set of variables V consists of:
    1. The special symbol S, which is the start symbol and
    2. All symbols of the form [pXq], where p and q are states in Q, and X is a stack symbol, in Γ.
- The productions of G are as follows:
    - For all states p, G has the production $S \rightarrow [q_0 Z_0 p]$. It is intended to generate all those strings w that cause P to pop Z_0 from its stack while going from state q_0 to state p.

      $$\text{i.e. } (q_0, w, Z_0) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$$

    - Let δ(q, a, X) contain the pair $(r, Y_1Y_2....Y_k)$, where:
        1. a is either the symbol in Σ or a = ε.
        2. k can be any number, including 0, in which case the pair is (r, ε).

           Then for all lists of states $r_1, r_2, \ldots, r_k$, G has the production

           $$[qXr_k] \rightarrow a[rY_1r_1][r_1Y_2r_2] \ldots [r_{k-1}Y_kr_k]$$

    - This production says that one way to pop X and go from state q to state r_k is to read a (which may be ε), then use some input to pop Y_1 off the stack while going from state r to state r_1, then read some more input that pop Y_2 off the stack and goes from state r_1 to r_2 and so on.

Example:

- Let us convert the PDA $P_N = (\{q\}, \{0, 1\}, \{Z, A, B\}, \delta_N, q, Z)$ to a grammar. Recall that P_N accepts all strings such that, the number of 0's is equal to the number of 1's. Since p has only one state and three stack symbols, the construction is simple. There are only four variables in the grammar G.
  - S, the start symbol, which is in every grammar constructed by the method of theorem 6(a), and
  - [qZq]
  - [qAq]
  - [qBq]
- The last three are the only triples that can be assembled from the states and stack symbols of P_N.
- The productions of grammar G are as follows:
  - The only production for S is S → [qZq]. However if there were n states of the PDA, then there would be n productions of this type, since the last state could be any of the n states. The first state would have to be the start state and the stack symbol would have to be the start symbol, as in our productions above.
  - The production [qZq] → 0[qZq][qAq] results from the fact that $\delta_N(q, 0, Z)$ contains (q, ZA). Again for this simple example there is only one production. However if there were n states then this one rule would produce n^2 productions since the two middle states of the body could also be any one state. That is if p and r were any two states of the PDA, then production [qZp] → 0[qZr][rAp] would be produced. In a similar way we get the productions.
  - [qZq] → 1[qZq][qBq].
  - [qAq] → 0[qAq][qAq].
  - [qBq] → 1[qBq][qBq].
  - From the fact that $\delta_N(q, 0, B)$ contains (q, ε), we have production [qBq] → 0. Notice that in this case, the list of stack symbols by which B is replaced is empty, so the only symbols in the body are the input symbol that caused the move. Similarly
  - [qAq] → 1.
  - [qAq] → ε.
- We may for convenience, replace the triple [qZq] by some less complex symbol, say X and similarly [qAq] and [qBq] by A and B, respectively. If we do then the complete grammar consists of the productions:

$$S \rightarrow X$$
$$X \rightarrow 0XA$$
$$X \rightarrow 1XB$$
$$A \rightarrow 0AA$$

B → 1BB
A → 1
B → 0
X → ε

- In fact, if we notice that X and S derive exactly the same strings, we may identify them as one and write the complete grammar as

G = ({S, A, B}, {0, 1}, {S → 0SA | 1SB | ε, A → 0AA | 1, B → 1BB | 0}, S)

INTRODUCTION TO DCFL

- In formal language theory, deterministic context-free languages (DCFL) are a proper subset of context-free languages. They are the context-free languages that can be accepted by a deterministic pushdown automaton.
- The notion of the DCFL is closely related to the deterministic pushdown automaton (DPDA). It is where the language power of a pushdown automaton is reduced if we make it deterministic; the pushdown automaton becomes unable to choose between different state transition alternatives and as a consequence cannot recognize all context-free languages.
- Unambiguous grammars do not always generate a DCFL. For example, the language of even-length palindromes on the alphabet of 0 and 1 has the unambiguous context-free grammar S → 0S0 | 1S1 | ε. The problem of whether a given context-free language is deterministic is undecidable.
- A pushdown automaton A=(Q,Σ, Γ, δ, q_0,Z_0, F) is deterministic if:
  - Whenever (q, a, X) is nonempty for some a Є Σ, then (q, ε, X) is empty, and
  - For each q ∈ Q, a ∈ Σ ∪ {ε} and X ∈ Γ, δ(q, a, X) contains at most one element.
- A language L is a deterministic context-free language (DCFL) if it is accepted by a deterministic pushdown automaton (DPDA).

DETERMINISTIC PDA

- While PDA's are by definition allowed to be nondeterministic the deterministic subcase is quite important. In particular, parser generally behave like deterministic PDA's, so the class of languages that can be accepted by these automata is interesting for the insights it gives us into what constructs are suitable for use in programming language.

DEFINITION OF DETERMINISTIC PDA

- Intuitively, a PDA is deterministic if there is never a choice of move in any situation. These choices are of two kinds.
- If δ(q, a, X) contains more than one pair, then surely the PDA is nondeterministic because we can choose among these pairs when deciding on the next move.

- However even if δ(q, a, X) is always a singleton, we could still have a choice between using a real input symbol, or making a move on ε.
- Thus we define a PDA P = (Q, Σ, Γ, δ, q_0, Z_0, F) to be deterministic (a deterministic PDA or DPDA), if and only if the following conditions are met:
  - δ(q, a, X) has at most one member for any q in Q, a in Σ or a = ε, and X in Γ.
  - If δ(q, a, X) is non empty, for some a in Σ, then δ(q, ε, X) must be empty.

Example:

- Consider the language L = {0^n1^n | n ≥ 1}. It turns out that this language can be recognized by a deterministic PDA.
- The strategy of the PDA is to store 0's on its stack, until it sees a 1. It then goes to another state, in which it pops the 0's each time it reads a 1.
- If it finds the bottom before reading the entire input, it dies: its input cannot be of the form 0^m1^m.
- If it succeeds in popping its stack down to the initial symbol, which marks the bottom of the stack, when the entire input has been read, then it accepts its input.
- The DPDA for language L is shown as a transition diagram in **figure 6.7**:

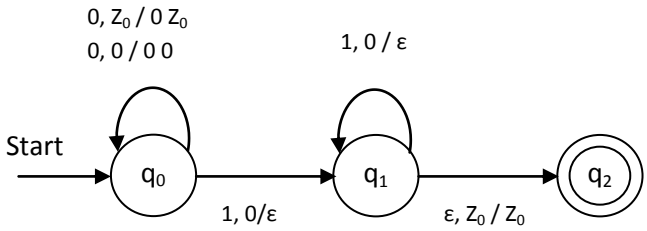


Figure 6.7: A Deterministic PDA accepting {0^n1^n| n > 1}

REGULAR LANGUAGES AND DETERMINISTIC PDA'S

- The DPDA's accept a class of languages that is between the regular languages and the CFL's.

Theorem:

- If L is a regular language, then L = L(P) for some DPDA P.

Proof:

- Essentially a DPDA can simulate the deterministic finite automaton. The PDA keeps some stack symbol Z_0 on its stack, because a PDA has to have the stack, but really the PDA ignores its stack and just uses its state.
- Formally, let A = (Q, Σ, δ_A, q_0, F) be a DFA. Construct DPDA P = (Q, Σ, {Z_0}, δ_P, q_0, Z_0, F) by defining $\delta_P(q, a, Z_0) = \{(p, Z_0)\}$ for all states p and q in Q, such that $\delta_A(q, a) = p$.
- If we want the DPDA to accept by empty stack, then we find that our language recognizing capability is rather limited.

- Say that a language L has the prefix property if there are no two different strings x and y in L such that x is a prefix of y.

Theorem:

- A language L is N(P) for some DPDA P if and only if L has the prefix property and L is L(P') for some DPDA P'.

DPDA's and context free languages

- We have already seen that a DPDA can accept languages like L_{wcwr} that are not regular. To see this language is not regular, suppose it were, and use the pumping lemma.
- If n is the constant of the pumping lemma, then consider the strings $w = 0^n c 0^n$, which is in L_{wcwr}.
- However when we pump this string, it is the first group of 0's whose length must change so we get in L_{wcwr} strings that have the "center" marker not in the center.
- Since these strings are not in L_{wcwr}, we have contradiction and conclude that L_{wcwr} is not regular.
- On the other hand, there are CFL's like L_{wwr} that cannot be L(P) for any DPDA P. A formal proof is complex, but the intuition is transparent.
- If P is DPDA accepting L_{wwr}, then given a sequence of 0's it must store them on the stack, or do something equivalent to count an arbitrary number of 0's. For instance if could store one X for two 0's it sees, and use the state to remember whether the number was even or odd.
- Suppose that P has seen n 0's and then sees 110^n. if must verify that there were n 0's after the 11, and to do so it must pop its stack.
- Now P has seen 0^n110^n. If it sees an identical string next, it must accept, because the complete input is of the form ww^R, with $w = 0^n110^n$. However, if it sees 0^m110^m for some **$m \neq n$,** P must not accept. Since its stack is empty.
- It cannot remember what arbitrary integer n was and must not able to recognize L_{wwr} correctly.
- Our conclusion is that
    - The language accepted by DPDA's by final state properly included the regular languages, but are properly included in the CFL's.