CONTEXT FREE LANGUAGES (CONT’D)

**PUSHDOWN AUTOMATA (PDA)**

**In this section, we introduce a new type of computational model called *pushdown automata (PDA).* These automata are like nondeterministic finite automata but have an extra component called a *stack.***

**The stack provides additional memory beyond the finite amount available in the control. The stack allows pushdown automata to recognize some nonregular languages.**

**Pushdown automata (PDA) are equivalent in power to context-free grammars. This equivalence is useful because it gives us two options for proving that a language is context free.**

**Then, a language is context free if we can give either a context-free grammar generating it or a push­down automaton recognizing it.**

**Certain languages are more easily described in terms of generators (grammars), whereas others are more easily described in terms of recog­nizers (push-down automata).**

**The following figure is a schematic representation of a finite automaton.**

**The control represents the states and transition function, the tape contains the input string, and the arrow represents the input head, pointing at the next input symbol**

**State**

**Control**

**a**

**a**

**b**

**b**

**FIGURE 2.4: Schematic of a Finite Automaton**

**Input**

**With the addition of a stack component we obtain a schematic representation of a pushdown automaton, as shown in the following picture.**

**FIGURE 2.3: Schematic of a Pushdown Automaton**

**State**

**Control**

**a**

**a**

**b**

**b**

**Input**

**x**

**y**

**z**

**…**

**A pushdown automaton (PDA), can write symbols on the stack and read them back later.**

**Writing a symbol "pushes down" all the other symbols on the stack.**

**At any time, the symbol on the top of the stack can be *read and removed*. The remaining symbols then move back up.**

**Writing a symbol on the stack is often re­ferred to *pushing* the symbol, and removing a symbol is referred to as *popping* it.**

**Note that all access to the stack, for both reading and writing, may be done only at the top. In other words, a stack is a *"last in, first out"* storage device. If certain information is written on the stack and additional information is written afterward, the earlier information becomes inaccessible until the later informa­tion is removed.**

**(Plates on a cafeteria serving counter illustrate a stack. The stack of plates rests on a spring so that when a new plate is placed on top of the stack, the plates below it move down. The stack on a pushdown automaton is like a stack of plates, with each plate having a symbol written on it).**

**A stack is valuable because it can hold an unlimited amount of information.**

**Recall that a finite automaton is unable to recognize the language** **{0n1n| n ≥ 0} because it cannot store very large numbers in its finite memory.**

**A PDA is able to recognize this language because it can use its stack to store the number of 0s it has seen (n) and then remove all (n) of them when it sees the (n) 1’s.**

**Thus, the unlimited nature of a stack allows the PDA to store numbers of unbounded size. The following *informal description* shows how the automaton for this language works.**

**Read symbols from the input. As each 0 is read, push it onto the stack. As soon as 1s are seen, pop a 0 off the stack for each 1 read. If reading the input is finished exactly when the stack becomes empty of 0s, *accept* the input.**

**If the stack becomes empty while 1s remain or if the 1s are finished while the stack still contains 0s or if any 0s appear in the input following 1s, *reject* the input.**

**As mentioned earlier, pushdown automata may be nondeterministic.**

**This fea­ture is crucial because, in contrast with the finite automata situation, nondeterminism adds power to the capability that pushdown automata would have if they were allowed only to be deterministic.**

**Some languages, such as {0n1n| n ≥ 0}, do not require nondeterminism, but others do. We give a language requiring nondeterminism later on.**

###### FORMAL DEFINITION OF A PUSHDOWN AUTOMATON

**The formal definition of a pushdown automaton is similar to that of a finite au­tomaton, except for the stack. The stack is a device containing symbols drawn from some alphabet.**

**The machine may use different alphabets for its input and its stack, so now we specify both an input alphabet  and a stack alphabet Γ.**

**At the heart of any formal definition of an automaton is the transition function, for that describes its behavior.**

**Recall that  =  ∪{} and Γ = Γ ∪{}.**

**The domain of the transition function is *Q* x  x Γ.**

**That is, the current state, next input symbol read, and top symbol of the stack determine the next move of a pushdown automaton.**

**Either symbol may be ** causing the machine to move without reading a symbol from the input or without reading a symbol from the stack.**

**For the range of the transition function we need to consider what to allow the automaton to do when it is in a particular situation.**

**It may enter some new state and possibly write a symbol on the top of the stack.**

**The function “*”* can indicate this action by returning a member of *Q* together with a member of *Γ*, that is, a member of *Q x Γ*.**

**Because we allow nondeterminism in this model, a situation may have several legal next moves.**

**The transition function incorporates nonde­terminism in the usual way, by returning a set of members of *Q x Γ,* that is, a member of *Ρ(Q x Γ),* (here *P( )* is the “power set of”).**

**Putting it all together, our transition function ** takes the form**

***: Q x  x Γ → P(Q x Γ).***

**DEFINITION 2.8**

**A *pushdown automaton* is a 7-tuple (Q, , Γ, , q0, $, F), where Q, , Γ and Fare all finite sets, and**

**1. Q is a finite set of *states*, like the states of a finite automaton**

**2.  is the finite *input alphabet*, analogous to the corresponding component of a FA**

**3. Γ is the *stack alphabet*. This component, which has no FA analog, is the finite set of symbols that we are allowed to push onto the stack.**

**4: **: *Q* x ** x Γ** → *P*(Qx Γ**) is the *transition function*. As for a FA, ** governs the behavior of the automaton. Formally, ** takes as argument a triple ** (q, a, X), where**

1. **q is a state in Q**
2. ***a* is either an input symbol in ** or a = , the empty string, which is not an input symbol.**
3. **X is a stack symbol, that is, a member of Γ.**

**5. *q0 ∈ Q* is the *start state*. The PDA is in this state before making any transitions.**

**6. $ *∈* Γis the stack’s *start symbol*. Initially, the PDA’s stack consists of one instance of this symbol, and nothing else.**

**7. F *⊆ Q* is the set of *accepting states* or *final states*.**

**A pushdown automaton *M* = (Q, , Γ, , q0, $, F) computes as follows.**

**It ac­cepts input *w* if *w* can be written as *w = w1w2 ...*  *wm*, where each *wi ∈ * and sequences of states *r0, r1,* ..., *rm* ∈ *Q* and strings *s0, s1, ... , sm ∈ Γ\** exist that satisfy the next three conditions.**

**The strings *si* represents the sequence of stack contents that *M* has on the accepting branch of the computation.**

**1. *r0 = qo* and *s0 = .* This condition signifies that *M* starts out properly, in the start state and with an empty stack (that is, the *bottom of the stack has the empty symbol*).**

**2. For *i = 0,... , m-1,* we have *(ri+1*, *b)* ∈*( ri*, *wi+1, a),* where *si* = *at*and *si+1 = bt* for some *a, b ∈ Γ* and *t ∈ Γ\*.***

**This condition states that *M* moves properly according to the state, stack, and next input symbol.**

***3. rm* ∈ *F.* This condition states that an accept state occurs at the input end.**

**EXAMPLES OF PUSHDOWN AUTOMATA (PDA)**

Note: In our class we may assume that the $ sign that marks the "bottom" of the stack (or "top" of the stack if the stack is empty) is *already there.* In some of the examples below they include steps to "write the $ sign in the empty stack.

**EXAMPLE 2.9**

**The following is the formal description of the PDA that recognizes the language {0n1n | *n* ≥ 0}.**

**Let M1, be (Q, , Γ, , q1, $, F), where**

**Q = {q1, q2, q3, q4},**

**= {0, 1},**

**Γ = {0, $}**

**F = {q1, q4}**

**And is given by the following table, wherein blank entries signify **.**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input: →** | **0** | | | **1** | | | **** | | |
| **Stack: →** | **0** | **$** | **** | **0** | **$** | **** | **0** | **$** | **** |
| ***q*1** |  |  |  |  |  |  |  |  | **{( *q*2, $)}** |
| ***q*2** |  |  | **{( *q*2, 0)}** | **{( *q*3,  )}** |  |  |  |  |  |
| ***q*3** |  |  |  | **{( *q*3,  )}** |  |  |  | **{( *q*4,  )}** |  |
| ***q*4** |  |  |  |  |  |  |  |  |  |

**We can also use a state diagram to describe a PDA, as shown in the following three figures.**

**Such diagrams are similar to the state diagrams used to describe finite automata, modified to show how the PDA uses its stack when going from state to state.**

**Notation:**

**We write “*a, b →* c" to signify that when the machine is reading an *a* from the input it replaces the symbol *b* on the top of the stack with a *c*.**

**Any of *a, b,* and c may be *.***

**If *a* is *,* the machine may make this transition without reading any symbol from the input.**

**If *b* is *,* the machine may make this transition without reading and popping any symbol from the stack.**

**If *c* is *,* the machine does not write any symbol on the stack when going along this transition.**

q1

q4

**, 🡪 $ (marking the stack with the $ sign).**

**, $ 🡪 deleting the $ sign written in the stack while transferring to the accepting state.**

**0, $ 🡪 0$**

**0, 0 🡪 00**

**Stacking 0's by "push" operations**

**1, 0 🡪  popping the first 0 at the top of the stack while transferring to the next state.**

**1, 0 🡪  popping 0's from the stack**

**Figure 2.6: PDA that recognizes {0n1n | n ≥ 0}**

**Let’s explain the symbols in the Figure above:**

** 🡪 $ : the machine is reading a  from the input and it sees a  at the top of the stack, it replaces the symbol  on top of the stack with a $.**

**$ 🡪 0$: the machine is reading a 0 from the input. It places the first 0 on top of the stack moving the $ sign down the stack.**

**$ 🡪 0: the machine is reading a following 0's from the input. It places these 0's on top of the stack moving the remaining symbols down the stack.**

**1, 0 🡪  : the machine is reading a 1 from the input. It pops the 0's on top of the stack, that is, it pops the stack by replacing the 0's with 's.**

**$ 🡪  : the machine is reading a  from the input it replaces the symbol $ on top of the stack with a .**

**The formal definition of a PDA contains no explicit mechanism to allow the PDA to test for an empty stack.**

**As we can see, this PDA is able to get this effect by initially placing a special symbol $ on the stack. Then if it ever sees the $ again, it knows that the stack effectively is empty.**

**In this class we will assume that the $ sign is already there marking the bottom (top) of the stack.**

**Subsequently, when we refer to testing for an empty stack in an informal description of a PDA, we implement the procedure in an equivalent way.**

**Similarly, PDAs cannot test explicitly for having reached the end of the input string.**

**This PDA is able to achieve that effect because the accept state takes effect *only when the machine is at the end of the input, that is, there no more input.***

**Thus, from now on, we assume that PDAs can test for the end of the input, and we know that we can implement it in the same manner.**

**EXAMPLE 2.10**

**This example illustrates a pushdown automaton that recognizes the language {aibjck | i, j, k, ≥ 0 and i = j or i = k}**

**Informally the PDA for this language works by first reading and pushing the a's. When the a's are done the machine has all of them on the stack so that it can match them with either the b's or the c's.**

**This maneuver is a bit tricky because the machine doesn't know in advance whether to match the a's with the b's or the c's.**

**Nondeterminism comes in handy here.**

**Using its nondeterminism, the PDA can guess whether to match the a's with the "b's or with the c's, as shown in the figure below.**

**Think of the machine as hav­ing two branches of its nondeterminism, one for each possible guess.**

**If either of them match, that branch accepts and the entire machine accepts. In fact, we could show, though we do not do so, that nondeterminism is *essential* for recognizing this language with a PDA.**

**q7**

**q4**

** 🡪 $**

**, $ 🡪 $**

**b, a 🡪 **

**FIGURE 2.7: PDA that recognizes {aibjck | i, j, k ≥ 0 and i = j or i = k}**

**(In this class you don't need q1 and its transition. Start at q2).**

**, a 🡪 a**

**, a 🡪 a**

**, a 🡪 a**

**, $ 🡪 $**

**c, $🡪 $**

**c, a 🡪 **

**b, a 🡪 a**

**a, $ 🡪 a$**

**a, a 🡪 aa**

**EXAMPLE 2.11**

**In this example we give a PDA M3 recognizing the language**

**{wwR | w ∈{0, 1}\* }**

**Recall that *wR* means *w* written backwards. The informal description of the PDA follows.**

**Begin by pushing the symbols that are read onto the stack.**

**At each point non-deterministically guess that the middle of the string has been reached and then change into popping off the stack for each symbol read, *checking to see that they are the same*.**

**If they were always the same symbol and the stack empties at the same time as the input is finished, accept; otherwise reject. The following is the diagram of this machine.**

q1

q4

**,  🡪 $ This step is to write the $ sign into the stack**

**, $ 🡪 $**

**0, $🡪 0$**

**, $ 🡪$**

**, 0 🡪0**

**, 1 🡪1**

**0, 0 🡪 **

**FIGURE 2.8: PDA that recognizes {wwR | w ∈ {0,1}\*}**

**1, $🡪 1$**

**1, 1 🡪 **

**EXAMPLE 2.11-A (6.1) A second solution to problem 2.11**

**(Taken from the book by Hopcroft, Motwani & Ullman: “Introduction to Automata Theory, Languages and Computation)**

**Let us consider the language**

**Lwwr = {wwR | w ∈{0, 1}\*}**

**This language, often referred to as “w-w-reversed” is the even length palindromes over the alphabet {0, 1}\*. It is a CFL, generated by the grammar P → 0P0 | 1P1 | .**

**We can design an informal PDA accepting Lwwr, as follows.**

**1. Start in a state q0 that represent 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 *q0*, we read symbols and store them on the stack, by pushing a copy of each symbol onto the stack, in turn.**

**2. 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 q1. Since the automaton is nondeterministic, we actually make both guesses: we guess we have seen the end on *w*, but we also stay in state q0 and continue to read inputs and store them on the stack.**

**3. Once in state q1, 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 *wR*. This branch dies, although other branches of the nondeterministic automaton may survive and eventually lead to acceptance.**

**4. If we empty the stack, then we have indeed seen some input *w* followed by *wR*. We accept the input that was read up to this point.**

**EXAMPLE 2.11-B (6.2)**

**Let us now design a PDA P to accept the language Lwwr just described. 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 $ 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 *wwR* on the input, we still have something on the stack to permit us to make a transition to the accepting state, q2. Thus, our PDA for L can be described as**

**P = ({q0, q1, q2}, {0, 1}, {0, 1, $}, , q0, $, {q2})**

**Where is defined by the following rules:**

**1.(q0, 0, $) = {(q0, 0$)} and (q0, 1, $) = {(q0, 1$)}. One of these rules applies initially, when we are at state q0 and we see the start symbol $ at the top of the stack. We read the first input, and push it onto the stack, leaving $ below to mark the bottom.**

**2. (q0, 0, 0) = {(q0, 00)}, (q0, 0, 1) = {(q0, 01)},**

**(q0, 1, 0) = {(q0, 10)}, (q0, 1, 1) = {(q0, 11)}.**

**These four, similar rules allow us to stay in state q0 and read inputs, pushing each onto the top of the stack and leaving the previous top stack symbol alone.**

**3. (q0, , $) = {(q1, $)}, (q0, , 0) = {(q1, 0)} and , (q0, , 1) = {(q1, 1)}}. These three rules allow P to go from state q0 to state q1 spontaneously (on input ), leaving intact whatever symbol is at the top of the stack.**

**4. (q1, 0, 0) = {(q1, )} and (q1, 1, 1) = {(q1, )}. Now in state q1 we can match input symbols against the top symbols on the stack, and pop when the symbols match.**

**5. (q1, , $) = {(q2, $)}. Finally, if we expose the bottom-of-stack marker $ and we are in state q1, then we have found an input of the form *wwR*. We go to state q2 and accept.**

**EXAMPLE 2.11-C (6.3)**

**The list of  facts, as in the previous example, is not too easy to follow. Sometimes, a diagram, generalizing the transition diagram of a finite automaton, will make aspects of the behavior of a given PDA clearer. We shall therefore introduce and subsequently use a *transition diagram* for PDA’s in which:**

**a) The nodes correspond to the states of the PDA.**

**b) An arrow labeled Start indicates the start state, and doubly circled states are accepting, as for finite automata.**

**c) The arcs correspond to transitions of the PDA in the following sense. An arc labeled a, X/ from state *q* to state *p* means that (q, a, X) contains the pair (p, ), perhaps among other pairs. That is, the arc label tells what input is used, and also gives the old and new tops of the stack.**

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

**The diagram below shows the previous PDA represented as a generalized transition diagram.**

**q2**

**, $ / $**

**0, 0 / **

**PDA that recognizes {wwR | w in (0+1)\*}**

**1, 1 / **

**, 0 / 0**

**, 1 / 1**

**, $ / $**

**0, $ / 0$**

**0, 0 / 00**

**1, 1 / 11**

**1, $ / 1$**

**0, 1 / 01**

**1, 0 / 10**

**EXAMPLE 2.11-C (6.3)**

**(I) Let’s consider the action of the PDA above on input 1111.**

**The start state is q0 and $ is the top of the stack symbol, thus, initial ID (instantaneous description) is (q0, 1111, $). On this input, the PDA has an opportunity to guess wrongly several times. The entire sequence of IDs that the PDA can reach from the initial ID (q0, 1111, $) is shown step by step below.**

**(1) From the initial ID (q0, 1111, $), there are moves as follows:**

**First: The PDA assumes that it’s already at the middle of the input and then is at q0 looking at a  input and at a $ at the bottom of the stack. It transfers to q1 and continues.**

**(q0, 1111, $) → [,$/$] → (q1, 1111, $) → [,$/$] → (q2, 1111, $) reject, the input is not ! continue iterating!**

**or**

**Second: The PDA assumes that it is one step from the middle sees a 1 coming in and a $ on top of the stack. It then pushes the 1 into the stack and stays at q0.**

**(q0, 1111, $) → [1,$/1$] → (q0, 111, 1$) → [,1/1]→ (q1, 111, 1$) → [1,1/] → (q1, 11, $) → [,$/$] → (q2, 11, $) reject, the input is not ! Continue iterating!**

**Or**

**Third: The PDA assumes that it is two steps from the middle sees a 1 coming in and a $ on top of the stack. It then pushes the 1 into the stack and stays at q0. Then it sees another 1 coming in and pushes it onto the stack and stays at q0. Now it transfers to q1 and starts popping the stack. After that it transfers to q2 with no input anymore. It accepts.**

**(q0, 1111, $) → [1,$/1$] → (q0, 111, 1$) → [,1/11]→ (q0, 11, 11$) → [,1/1] → (q1, 11, 11$) → [1,1/] → (q1, 1, 1$) → [1,1/] → (q1, , $) → [,$/$]** **→ (q2, **, $) accept! STOP. THERE IS NO NEED TO CONTINUE ITERATING!**

**(II) Let’s NOW consider the action of the PDA above on input 111.**

**As before, the start state is q0 and $ is the top of the stack symbol, thus, initial ID (instantaneous description) is (q0, 111, $). The entire sequence of IDs that the PDA can reach from the initial ID (q0, 111, $) is shown step by step below.**

**(1) From the initial ID (q0, 111, $), there are moves as follows:**

**First: The PDA assumes that it’s already at the middle of the input and then is at q0 looking at a  input and at a $ at the bottom of the stack. It transfers to q1 and continues.**

**(q0, 111, $) → [,$/$] → (q1, 111, $) → [,$/$] → (q2, 111, $) reject, the input is not ! Continue iterating!**

**Second: The PDA assumes that it is one step from the middle sees a 1 coming in and a $ on top of the stack. It then pushes the 1 into the stack and stays at q0.**

**(q0, 111, $) → [1,$/1$] → (q0, 11, 1$) → [,1/1] → (q1, 11, 1$) → [1,1/] → (q1, 1, $) → [,$/$] → (q2, 1, $) reject, the input is not ! Continue iterating!**

**Third: The PDA PDA assumes that it is two steps from the middle sees a 1 coming in and a $ on top of the stack. It then pushes the 1 into the stack and stays at q0. The process continues as shown below**

**(q0, 111, $) → [1,$/1$] → (q0, 11, 1$)→ [1,1/11] → (q0, 1, 11$) → [,1/1] → (q1, 1, 11$) → [1,1/] → (q1, , 1$) → reject, the process is stack at q1! Continue iterating!  
Fourth: From the state (q0, , 111$), there is only one last choice as follows:**

**(q0, 111, $) → [1,$/1$] → (q0, 11, 1$) → [1,1/11] → (q0, 1, 11$) → [1,1/11] → (q0, , 111$) → [,1/1] → (q1, , 111$) → reject, the process is stack at q1! End of the iterations!**

**♦**

**There are three important principles about ID’s and their transitions that we shall need in order to reason about PDA’s:**

**1. If a sequence of ID’s (*computations*) is legal for a PDA, 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.**

**2. 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.**

**3. 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 resultant computation will still be legal.**

**In summary:**

**Theorem: If P = (Q, , Γ, , $, F) is a PDA and (q, x, ) −\*→(p, y, ), then for any *w* ∈ \* and  ∈ Γ\*, it is also true that**

**(q, x*w*, ) −\*→(p, y*w*, ).**

**Note that if  = , then we have a formal statement of principle (1) above, and if w = , then we have the second principle.**

**Proof:**

**The proof is actually a very simple induction on the number of steps in the sequence of ID’s that take (q, x*w*, ) to (p, y*w*, ).**

**Each of the moves in the sequence (q, x, ) −\*→(p, y, ) is justified by the transitions of P without using w and/or  in any way. Therefore, each move is still justified when these strings are sitting on the input and the stack.**

**PDA EQUIVALENCE WITH CONTEXT-FREE GRAMMARS**

**In this section we show that context-free grammars and pushdown automata are equivalent in power.**

**Both are capable of describing the class of context-free lan­guages.**

**We show how to convert any context-free grammar into a pushdown automaton that recognizes the same language and vice versa.**

**Recalling that we defined a context-free language to be any language that can be described with a context-free grammar, our objective is the following theorem.**

**THEOREM 2.12**

**A language is context free *if and only if* some pushdown automaton recognizes it.**

**As usual for "if and only if" theorems, we have two directions to prove. In this theorem, both directions are interesting. First, we do the easier forward direc­tion.**

**We will perform the proof by using two lemmas:**

**LEMMA 2.13**

**If a language is context free, then some pushdown automaton recognizes it.**

**and**

**LEMMA 2.15**

**If a pushdown automaton recognizes some language, then it is context free.**

**♦**

**LEMMA 2.13**

**If a language is context free, then some pushdown automaton recognizes it.**

**PROOF IDEA**

**Let *A* be a CFL. From the definition we know that *A* has a CFG, *G,* generating it. We show how to convert *G* into an equivalent PDA, which we call P.**

**The PDA *P* that we now describe will work by accepting its input *w,* if *G* gen­erates that input, by determining whether there is a derivation for *w.***

**Recall that a derivation is simply the sequence of substitutions made as a grammar generates a string. Each step of the derivation yields an *intermediate string* of variables and terminals.**

**We design *P* to determine whether some series of substitutions using the rules of *G* can lead from the start variable to *w.***

**One of the difficulties in testing whether there is a derivation for *w* is in fig­uring out which substitutions to make.**

**The PDA's nondeterminism allows it to guess the sequence of correct substitutions.**

**At each step of the derivation one of the rules for a particular variable is selected nondeterministically and used to substitute for that variable.**

**The PDA *P* begins by writing the start variable on its stack. It goes through a series of intermediate strings, making one substitution after another. Eventually it may arrive at a string that contains only terminal symbols, meaning that it has derived a string using the grammar. Then *P* accepts if this string is identical to the string it has received as input.**

**Implementing this strategy on a PDA requires one additional idea.**

**We need to see how the PDA stores the intermediate strings as it goes from one to another. Simply using the stack for storing each intermediate string is tempting. However, that doesn't quite work because the PDA needs to find the variables in the inter­mediate string and make substitutions.**

**The PDA can access only the top symbol on the stack and that may be a terminal symbol instead of a variable.**

**The way around this problem is to keep *only part* of the intermediate string on the stack: the symbols starting with the first variable in the intermediate string.**

**Any ter­minal symbols appearing before the first variable are matched immediately with symbols in the input string. The following figure shows the PDA *P.***

control

0

1

0

1

0

0

A

1

A

1

$

0 1 A 1 A 0

**FIGURE 2.9: P representing the intermediate string 01A1A0**

**The following is an informal description of P.**

**1. Place the marker symbol $ and the start variable on the stack,**

**2. Repeat the following steps forever.**

**a. If the top of stack is a variable symbol *A,* nondeterministically select one of the rules for *A* and substitute *A* by the string on the right-hand side of the rule.**

**b. If the top of stack is a terminal symbol *a,* read the next symbol from the input and compare it to *a.* If they match, repeat. If they do not match, reject on this branch of the nondeterminism.**

**c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.**

**PROOF**

**We now give the formal details of the construction of the pushdown automaton P = (Q, Γ, q1, $, F).**

**To make the construction clearer we use shorthand notation for the transition function.**

**This notation provides a way to write an entire string on the stack in one step of the machine.**

**We can simulate this action by introducing additional states to write the string one symbol at a time, as implemented in the following formal construction.**

**Let *q* and *r* be states of the PDA, and let *a* be in  and *s* be in Γ*.***

**Say that we want the PDA to go from *q* to r when it reads *a* and pops *s.*  Furthermore we want it to push the entire string *u= u1...ul* on the stack at the same time.**

**We can implement this action by introducing new states**

**q*1*, ... *, ql-1***

**and setting the transition function**

*** (q,a,s)* to contain *(q1, ul),***

*** (q1, , ) = {(q2, ul-1)},***

*** (q2, , ) = {(q3, ul-2)},***

***.***

***.***

***.***

*** (ql-1, , ) = {(r, u1)}.***

**We use the notation (r, *u) ∈*** ***(q,* a, *s)* to mean that when *q* is the state of the automaton, *a* is the next input symbol, and *s* is the symbol on the top of the stack, the PDA may read the *a* and pop the *s,* then push the string *u* onto the stack and go on to the state r.**

**The following figure shows this implementation pictorially.**

q

r

**a, s xyz**

**e, e y**

**a, s z**

**FIGURE 2.10: Implementing the shorthand (r, xyz) in (q, a, s)**

**e, e y**

**The states of P are**

**Q = {qstart, qloop, qaccept} ∪ E,**

**where E is the set of states we need for implementing the shorthand just described. The start state is qstart. The only accept state is qaccept.**

**The transition function is defined as follows.**

**We begin by initializing the stack to contain the symbols $ and S, implementing step 1 in the informal description:**

**(qstart, , ) = {(qloop, S$)}.**

**Then we put in transitions for the main loop of step 2.**

**First, we handle case (a) wherein the top of the stack contains a variable. Let (qloop, , A) = {(qloop, w) where A→ w is a rule in R}.**

**Second, we handle case (b) wherein the top of the stack contains a terminal. Let(qloop, a, a) = {(qloop, )}.**

**Finally, we handle case (c) wherein the empty stack marker $ is on the top of the stack. Let (qloop, e, $) = {(qaccept, e)}.**

**The state diagram is shown in the following figure.**

qaccept

**e, e s$**

**FIGURE 2.11: State diagram of P**

**e, $ e**

**a, a e for terminal a**

**e, A w for rule A w**

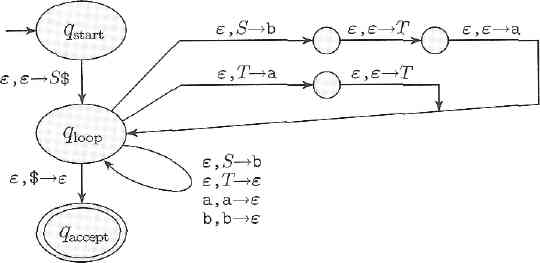
**EXAMPLE 2.14**

**We use the procedure developed in Lemma 2.13 to construct a PDA *P1* from the following CFG *G.***

***S* → aTb | b**

***T* → Ta | e**

**The transition function is shown in the following diagram.**

****

**FIGURE 2.12: State diagram of *P1***

**Now we prove the reverse direction of Theorem 2.12. For the forward direc­tion we gave a procedure for converting a CFG into a PDA. The main idea was to design the automaton so that it simulates the grammar.**

**Now we want to give a procedure for going the other way: converting a PDA into a CFG.**

**We design the grammar to simulate the automaton. This task is a bit tricky because "program­ming" an automaton is easier than "programming" a grammar,**

**LEMMA 2.15**

**If a pushdown automaton recognizes some language, then it is context free.**

**PROOF IDEA**

**We have a PDA *P,* and we want to make a CFG *G* that generates all the strings that *P* accepts. In other words, *G* should generate a string if that string causes the PDA to go from its start state to an accept state.**

**To achieve this outcome we design a grammar that does somewhat more. For each pair of states *p* and *q* in *P* the grammar will have a variable *Apq.* This variable generates all the strings that can take *P* from *p* with an empty stack to *q* with an empty stack.**

**Observe that such strings can also take *P* from *p* to *q,* regardless of the stack contents at *p,* leaving the stack at *q* in the same condition as it was at *p.***

**First, we simplify our task by modifying *P* slightly to give it the following three features.**

**1. It has a single accept state, qaccept.**

**2. It empties its stack before accepting.**

**3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack *(*a  *pop* move), but does not do both at the same time.**

**Giving *P* features 1 and 2 is easy.**

**To give it feature 3, we replace each transi­tion that simultaneously pops and pushes with a two transition sequence that goes through a new state, and we replace each transition that neither pops nor pushes with a two transition sequence that pushes then pops an arbitrary stack symbol.**

**To design *G* so that *Apq* generates all strings that take *P* from *p* to *q,* starting and ending with an empty stack, we must understand how *P* operates on these strings.**

**For any such string *x,* P's first move on *x* must be a push, because every move is either a push or a pop and *P* can't pop an empty stack. Similarly the last move on *x* must be a pop, because the stack ends up empty.**

**Two possibilities occur during P's computation on *x.* Either the symbol popped at the end is the symbol that was pushed at the beginning, or not. If so, the stack is empty only at the beginning and end of P's computation on *x.* If not, the initially pushed symbol must get popped at some point before the end of *x* and thus the stack becomes empty at this point.**

**We simulate the former possi­bility with the rule *Apq* → *aArsb* where *a* is the input symbol read at the first move, *b* is the symbol read at the last move, *r* is the state following *p,* and *s* the state preceding *q.***

**We simulate the latter possibility with the rule *Apq* →*AprArq,* where r is the state when the stack becomes empty.**

**PROOF**

**Say that *P* = *(Q, Γ, q0, {qaccept})* and construct *G.* The variables of *G* are *{Apq | p, q* ∈ *Q}.* The start variable is *Aqo,qaccept.* Now we describe G's rules.**

**• For each *p,q,r,s* ∈ *Q, t* ∈ *Γ*, and *a, b* ∈ e *if (p,a,e)* contains (*r,* *t)* and * (s, b, t)* contains *(q, e)* put the rule *Apq → aArsb* in *G.***

**• For each p, *q, r* ∈ *Q* put the rule *Apq* → *AprArq* in *G.***

**• Finally, for each *p*∈ *Q* put the rule *App* → *e* in *G.***

**You may gain some intuition for this construction from the following figures.**

p

r

q

**FIGURE 2.13: PDA computation corresponding to the rule Apq 🡪 AprArq**

Stack

height

input string

generated

by Apr

generated

by Arq

generated

by Apq

p

r

q

**FIGURE 2.14: PDA computation corresponding to the rule Apq 🡪 a Ars b**

Stack

height

input string

generated

by Ars

generated

by Apq

s

**Now we prove that this construction works by demonstrating that *Apq* gener­ates *x* if and only if *x* can bring *P* from *p* with empty stack to *q* with empty stack.**

**We consider each direction of the if and only if as a separate claim.**

**CLAIM 2.16:**

**If *Apq* generates *x,* then *x* can bring *P* from *p* with empty stack to *q* with empty stack.**

**We prove this claim by induction on the number of steps in the derivation of *x* from *Apq.***

***Basis:* The derivation has 1 step.**

**A derivation with a single step must use a rule whose right-hand side contains no variables. The only rules in *G* where no variables occur on the right-hand side are *App → e.* Clearly, input *e* takes *P* from *p* with empty stack to *p* with empty stack so the basis is proved.**

***Induction step:* Assume true for derivations of length at most *k,* where *k*  ≥ 1, and prove true for derivations of length *k +* 1.**

**Suppose that *Apq =\**=> *x* with *k* + 1 steps. The first step in this derivation is either *Apq* => *aArsb* or *Apq* => *AprArq.* We handle these two cases separately.**

**In the first case, consider the portion *y* of *x* that *Ars* generates, so *x =* *ayb.* Because *Ars =\**=> *y* with *k* steps, the induction hypothesis tells us that *P* can go from *r* on empty stack to *s* on empty stack.**

**Because *Apq → aArsb* is a rule of *G, (p,a,* e) contains *(r, t)* and *(s,* b, *t)* contains *(q, e).* Hence, if *P* starts at *p* with an empty stack, after reading *a* it can go to state *r* and push *t* on the stack. Then reading string *y* can bring it to *s* and leave *t* on the stack. Then after reading *b* it can go to state *q* and pop *t* off the stack. Therefore *x* can bring it from *p* with empty stack to *q* with empty stack.**

**In the second case, consider the portions *y* and *z* of *x* that *Apr* and Arq respec­tively generate, so *x = yz.* Because *Apr =\*=> y* in at most *k* steps and *Arq* =\*=> z in at most *k* steps, the induction hypothesis tells us that *y* can bring *P* from *p* to *r,* and s can bring P from r to *q,* with empty stacks at the beginning and end. Hence *x* can bring it from *p* with empty stack to *q* with empty stack. This completes the induction step.**

**CLAIM 2.17**

**If *x* can bring *P* from *p* with empty stack to *q* with empty stack, *Apq* generates *x.***

**We prove this claim by induction on the number of steps in the computation of *P* that goes from *p* to *q* with empty stacks on input *x.***

***Basis:* The computation has 0 steps.**

**If a computation has 0 steps, it starts and ends at the same state, say, *p.* So we must show that *App =\**=> *x.* In 0 steps, *P* only has time to read the empty string, so *x = e.* By construction, *G* has the rule *App →* e, so the basis is proved.**

***Induction step:* Assume true for computations of length at most *k,* where *k* ≥ 0, and prove true for computations of length *k* + 1.**

**Suppose that *P* has a computation wherein *x* brings *p* to *q* with empty stacks in *k* + 1 steps. Either the stack is empty only at the beginning and end of this com­putation, or it becomes empty elsewhere, too.**

**In the first case, the symbol that is pushed at the first move must be the same as the symbol that is popped at the last move. Call this symbol *t.* Let *a* be the input read in the first move, *b* be the input read in the last move, *r* be the state after the first move, and *s* be the state before the last move. Then *(p, a, e)* contains (r, *t)* and * (s, b, t)* contains *(q, e),* and so rule *Apq → aArsb* is in *G.***

**Let *y* be the portion of *x* without *a* and *b,* so *x = ayb.* Input *y* can bring *P* from *r* to *s* without touching the symbol *t* that is on the stack and so *P* can go from *r* with an empty stack to *s* with an empty stack on input *y.***

**We have removed the first and last steps of the *k* + 1 steps in the original computation on *x* so the computation on *y* has *(k* + 1) - 2 = *k* - 1 steps. Thus the induction hypothesis tells us that *Ars* =\*=> *y.* Hence *Apq =\**=> *x.***

**In the second case, let *r* be a state where the stack becomes empty other than at the beginning or end of the computation on *x.* Then the portions of the com­putation from *p* to *r* and from r to *q* each contain at most *k* steps. Say that *y* is the input read during the first portion and *z* is the input read during the second portion. The induction hypothesis tells us that *Apr =\*=> y* and *Arq* =\*=> *z.* Because rule *Apq*  *→ AprArq* is in *G, Apq =\**=> x, and the proof is complete.**

**That completes the proof of Lemma 2.15 and of Theorem 2.12.**

**We have just proved that pushdown automata recognize the class of context free languages. This proof allows us to establish a relationship between the reg­ular languages and the context-free languages.**

**Because every regular language is recognized by a finite automaton and every finite automaton is automatically a pushdown automaton that simply ignores its stack, we now know that every reg­ular language is also a context-free language.**

**COROLLARY 2.18:**

Context Free

languages

Regular

languages

**Every regular language is context free.**