Exploiting non determinism:

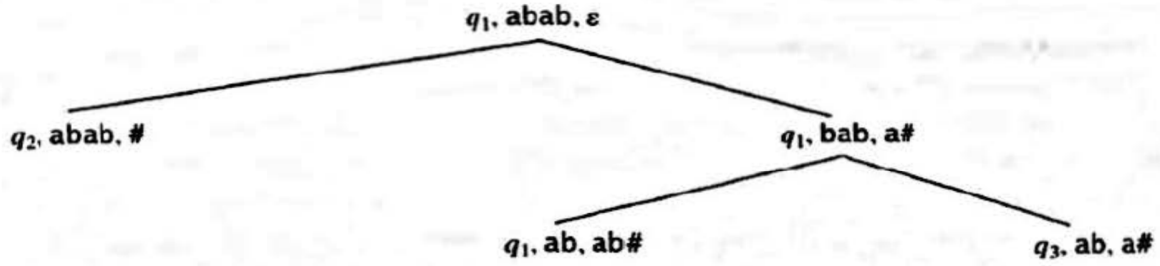
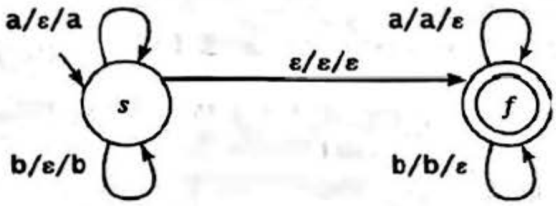


FIGURE 12.1 Viewing nondeterminism as search through a space of computation paths.

EXAMPLE 12.5 Even Length Palindromes

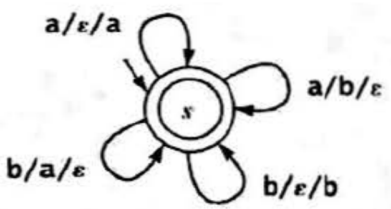
Consider again PalEven = $\{ww^R : w \in \{a, b\}^*\}$, the language of even-length palindromes of a's and b's. The following nondeterministic PDA M accepts PalEven:



M is nondeterministic because it cannot know when it has reached the middle of its input. Before each character is read, it has two choices: It can guess that it has not yet gotten to the middle. In that case, it stays in state s, where it pushes each symbol it reads. Or it can guess that it has reached the middle. In that case, it takes the ε-transition to state f, where it pops one symbol for each symbol that it reads.

EXAMPLE 12.6 Equal Numbers of a's and b's

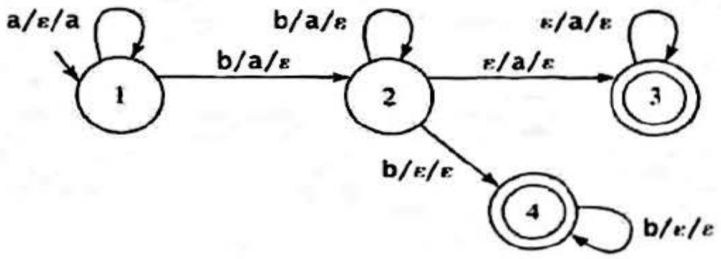
Let $L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$. Now we don't know the order in which the a's and b's will occur. They can be interleaved. So for example, any PDA to accept L must accept aabbba. The only way to count the number of characters that have not yet found their mates is to use the stack. So the stack will sometimes count a's and sometimes count b's. It will count whatever it has seen more of. The following simple PDA accepts L:



This machine is highly nondeterministic. Whenever it sees an a in the input, it can either push it (which is the right thing to do if it should be counting a's) or attempt to pop a b (which is the right thing to do if it should be counting b's). All the computations that make the wrong guess will fail to accept since they will not succeed in clearing the stack. But if $\#_a(w) = \#_b(w)$, there will be one computation that will accept.

EXAMPLE 12.7 The a Region and the b Region are Different

Let $L = \{a^m b^n : m \neq n; m, n > 0\}$. We want to build a PDA M to accept L. It is hard to build a machine that looks for something negative, like $\neq$. But we can break L into two sublanguages: $\{a^m b^n : 0 < m < n\}$ and $\{a^m b^n : 0 < n < m\}$. Either there are more a's or more b's. M must accept any string that is in either of those sublanguages. So M is:

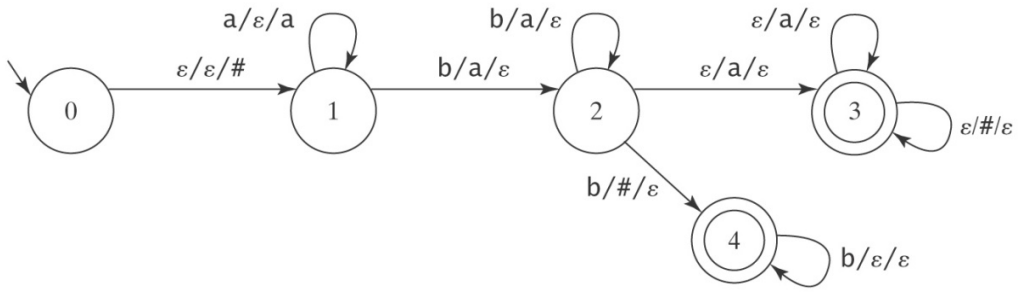


As long as M sees a's, it stays in state 1 and pushes each a onto the stack. When it sees the first b, it goes to state 2. It will accept nothing but b's from that point on. So far, its behavior has been deterministic. But, from state 2, it must make choices. Each time it sees another b and there is an a on the stack, it should consume the b and pop the a and stay in state 2. But, in order to accept, it must eventually either read at least one b that does not have a matching a or pop an a that does not have a matching b. It should do the former (and go to state 4) if there is a b in the input stream when the stack is empty. But we have no way to specify that a move can be taken only if the stack is empty. It should do the latter (and go to state 3) if there is an a on the stack but the input stream is empty. But we have no way to specify that the input stream is empty.

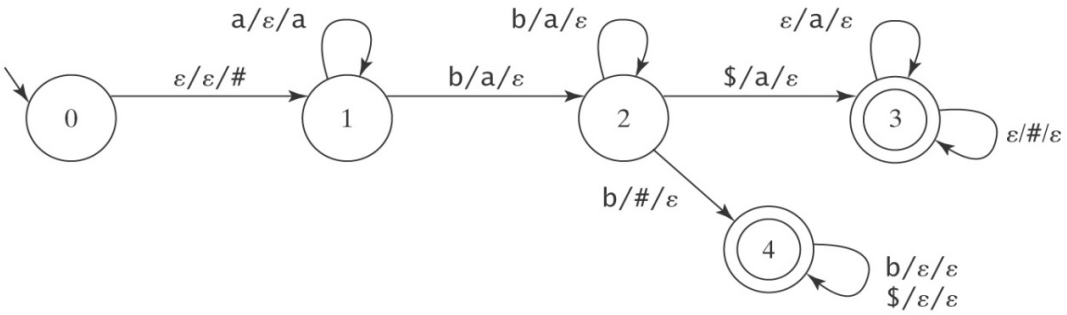
As a result, in most of its moves in state 2, M will have a choice of three paths to take. All but the correct one will die out without accepting. But a good deal of computational effort will be wasted first.

Techniques for reducing non determinism:

a) Using a bottom of stack marker
- Need to detect bottom of stack, so push # onto the stack before we start.



b) Using an end-of string Marker
- Need to detect end of input. Add to *L* a termination character (e.g., $)



Non-determinism and halting:

Recall that a computation C of a PDA $M = (K, \Sigma, \Gamma, \Delta, s, A)$ on a string w is an accepting computation iff:

$$C = (s, w, \varepsilon) \vdash_M^* (q, \varepsilon, \varepsilon), \text{ for some } q \in A.$$

We'll say that a computation C of M ***halts*** iff at least one of the following conditions holds:

- C is an accepting computation. or
- C ends in a configuration from which there is no transition in Δ that can be taken.

We'll say that M ***halts*** on w iff every computation of M on w halts. If M halts on w and does not accept, then we say that M rejects w.

Unfortunately, the facts about context-free languages and PDAs are different from the facts about regular languages and FSMs. Now we must face the following:

1. There are context-free languages for which no deterministic PDA exists. We'll prove this as Theorem 13.13.
2. It is possible that a PDA may
   - not halt, or
   - not ever finish reading its input.

So, let M be a PDA that accepts some language L. Then, on input w, if $w \in L$ then M will halt and accept. But if $w \notin L$, while M will not accept w, it is possible that it will not reject it either. To see how this could happen, let $\Sigma = \{a\}$ and consider the PDA M, shown in Figure 12.4. $L(M) = \{a\}$. The computation $(1, a, \varepsilon) \vdash (2, a, a) \vdash (3, \varepsilon, \varepsilon)$ will cause M to accept a. But consider any other input except a. Observe that:

- M will never halt. There is no accepting configuration, but there is always at least one computational path that has not yet halted. For example, on input aa, one such path is:
  (1, aa, ε) |- (2, aa, a) |- (1, aa, aa) |- (2, aa, aaa) |-
  (1, aa, aaaa) |- (2, aa, aaaaa) |-...
- M will never finish reading its input unless its input is ε. On input aa, for example, there is no computation that will read the second a.

3. There exists no algorithm to minimize a PDA. In fact, it is undecidable whether a PDA is already minimal.

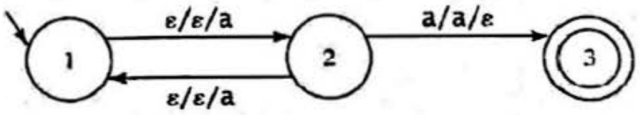


FIGURE 12.4 A PDA that may neither accept nor reject.

Problem 2 is especially critical. This same problem also arose with NDFSMs. But there we had a choice of two solutions:

- Use *ndfsmtodfsm* to convert the NDFSM to an equivalent deterministic one. A DFSM halts on input w in $|w|$ steps.
- Simulate the NDFSM using *ndfsmsimulate*, which ran all computational paths in parallel and handled ε-transitions in a way that guaranteed that the simulation of an NDFSM M on input w halted in $|w|$ steps.

Neither of those approaches works for PDAs. There may not be an equivalent deterministic PDA. And it is not possible to simulate all paths in parallel on a single PDA because each path would need its own stack. So what can we do? Solutions to these problems fall into two classes:

- Formal ones that do not restrict the class of languages that are being considered. Unfortunately, these approaches generally do restrict the *form* of the grammars and PDAs that can be used. For example, they may require that grammars be in Chomsky or Greibach normal form. As a result, parse trees may not make much sense.
- Practical ones that work only on a subclass of the context-free languages. But the subset is large enough to be useful and the techniques can use grammars in their natural forms.

Alternative equivalent Definition for PDA

We could have defined a PDA somewhat differently. We list here a few reasonable alternative definitions. In all of them a PDA M is a s ктuple $(K, \Sigma, \Gamma, \Delta, s, A)$:

- We allow M to pop and to push any string in Γ^*. In some definitions, M may pop only a single symbol but it may push any number of them. In some definitions, M may pop and push only a single symbol.
- In our definition, M accepts its input w only if, when it finishes reading w, it is in an accepting state and its stack is empty. There are two alternatives to this:
  - Accept if, when the input has been consumed, M lands in an accepting state, regardless of the contents of the stack.
  - Accept if, when the input has been consumed, the stack is empty, regardless of the state M is in.

EXAMPLE 12.14 Accepting by Final State Alone

Define a PDA $M = (K, \Sigma, \Gamma, \Delta, s, A)$ in exactly the way we have except that it will accept iff it lands in an accepting state, regardless of the contents of the stack. In other words, if $(s, w, \varepsilon) \vdash_M^* (q, \varepsilon, \gamma)$ and $q \in A$, then M accepts.

To show that this model is equivalent to ours, we must show two things: For each of our machines, there exists an equivalent one of these, and, for each of these, there exists an equivalent one of ours. We'll do the first part to show how such a construction can be done. We leave the second as an exercise.

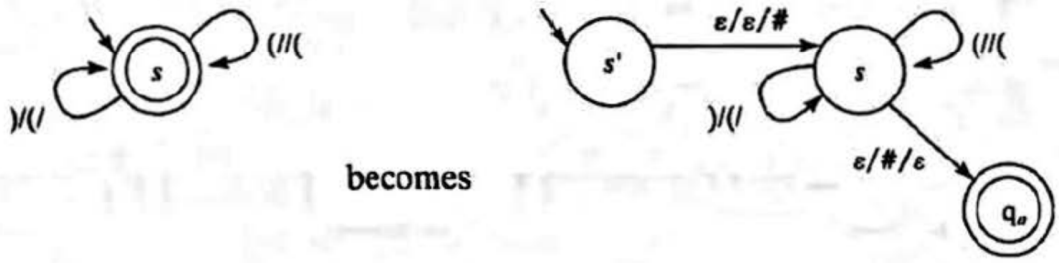
Given a PDA M that accepts by accepting state and empty stack, construct a new PDA M' that accepts by accepting state alone, where $L(M') = L(M)$. M' will have a single accepting state q_a. The only way for M' to get to q_a will be to land in an accepting state of M when the stack is logically empty. But there is no way to check that the stack is empty. So M' will begin by pushing a bottom-of-stack marker #, onto the stack. Whenever # is the top symbol on the stack, the stack is logically empty.

So the construction proceeds as follows:

1. Initially, let $M' = M$.
2. Create a new start state s'. Add the transition $((s', \varepsilon, \varepsilon), (s, \#))$.
3. Create a new accepting state q_a.
4. For each accepting state a in M do:
   Add the transition $((a, \varepsilon, \#), (q_a, \varepsilon))$.
5. Make q_a the only accepting state in M'.

It is easy to see that M' lands in its only accepting state (q_a) iff M lands in some accepting state with an empty stack. Thus M' and M accept the same strings.

As an example, we apply this algorithm to the PDA we built for the balanced parentheses language Bal:



Notice, by the way, that while M is deterministic, M' is not.