Lecture 14

Stack machines

We will now define a new computational model, the *stack machine model*, and observe that it is equivalent in power to the deterministic Turing machine model. There are two principal motives behind the introduction of the stack machine model at this point in the course:

1. Stack machines are easier to work with than Turing machines, at least at lower levels of detail.

   Whereas formally specifying Turing machines for even the most basic computations is a tedious task prone to errors, the analogous task for stack machines is simpler in comparison. Through the equivalence of the two models, one might therefore be more easily convinced that Turing machines abstract the capabilities of ordinary computers.

2. The equivalence of stack machines and Turing machines provides us with a nice example in support of the Church–Turing thesis.

   The stack machine model is natural and intuitive, particularly within the context of this course and the other models we have studied, and you can certainly imagine building one—or even just emulating one yourself using sheets of paper to implement stacks. That is, they represent mechanical processes. As the Church–Turing thesis predicts, anything you can compute with this model can also be computed with a deterministic Turing machine.

The stack machine model resembles the pushdown automata model, but unlike that model the stack machine model permits the use of *multiple stacks*. This makes all the difference in the world. In fact, as we will see, just two stacks endow a stack machine with universal computational power. Also unlike the pushdown automata model, we will only consider a deterministic variant of the stack machine model (although one can easily define nondeterministic stack machines).

14.1 Definition of stack machines

In this section we will formally define the deterministic stack machine model, but before proceeding to the definition it will be helpful to first mention a few specific points about the model.

1. As was already suggested, this model may be viewed as a deterministic, multi-stack version of a PDA. We may have any number r of stacks, so long as the number is fixed. Formally we do this by defining r-stack deterministic stack machines for every positive integer r.
2. For simplicity we assume that every stack has the same stack alphabet Δ.[1] This alphabet must include a special bottom-of-the-stack symbol $\diamond$, and it must also include the input alphabet Σ (which itself may not include the symbol $\diamond$). We require that the input alphabet is contained in the stack alphabet because the input is assumed to be stored (left-most symbol on top) in the first stack when the computation begins.
3. The state set Q of a stack machine must include a start state q_0 as well as the *halting states* q_{acc} and q_{rej} (which cannot be equal, naturally).
4. Each non-halting state of a stack machine, meaning any element of Q except q_{acc} and q_{rej}, always has exactly one of the r stacks associated with it, and must be either a *push state* or a *pop state*. When the stack machine transitions from a push state to another state, a symbol is always pushed onto the stack associated with that state, and similarly when a stack machine transitions from a pop state to another state, a symbol is popped off of the stack associated with that state (assuming it is non-empty).

Definition 14.1. An r-stack deterministic stack machine (r-DSM, for short) is an 7-tuple

$$M = (Q, \Sigma, \Delta, \delta, q_0, q_{acc}, q_{rej}), \tag{14.1}$$

where

1. Q is a finite and nonempty set of *states*,
2. Σ is an *input alphabet*, which may not include the bottom-of-the-stack symbol $\diamond$,
3. Δ is a *stack alphabet*, which must satisfy $\Sigma \cup \{\diamond\} \subseteq \Delta$,

[1]It would be straightforward to generalize the model to allow for each stack to have a different stack alphabet—we are just opting for a simpler definition.

4. δ is a *transition function* of the form

$$\delta : (Q\backslash\{q_{acc}, q_{rej}\}) \to \big(\{1,\ldots,r\} \times \Delta \times Q\big) \cup \big(\{1,\ldots,r\} \times Q^{\Delta}\big), \tag{14.2}$$

   and

5. $q_0, q_{acc}, q_{rej} \in Q$ are the *initial state*, *accept state*, and *reject state*, respectively, which must satisfy $q_{acc} \neq q_{rej}$.

When we refer to a DSM without specifying the number of stacks r, we simply mean an r-DSM for some choice of a fixed positive integer r. Sometimes we may not care specifically how many stacks a given DSM has, and we leave the number unspecified for simplicity.

In the specification of the transition function δ, the notation Q^{Δ} refers to the set of all functions from Δ to Q. The interpretation of the transition function δ is as follows:

1. If p is a non-halting state and $\delta(p) \in \{1,\ldots,r\} \times \Delta \times Q$, then the state p is a push state; if it is the case that

$$\delta(p) = (k, a, q), \tag{14.3}$$

   then when the machine is in state p, it pushes the symbol a onto stack number k and transitions to state q.

2. If p is a non-halting state and $\delta(p) \in \{1,\ldots,r\} \times Q^{\Delta}$, then the state p is a pop state; if it is the case that

$$\delta(p) = (k, f) \tag{14.4}$$

   for $f : \Delta \to Q$, then when the machine is in state p, it pops whatever symbol $a \in \Delta$ is on the top of stack number k and transitions to the state $f(a)$. Notice specifically that this allows for conditional branching: the state $f(a)$ that the machine transitions to may depend on the symbol a that is popped.

   If it so happens that stack number k is empty in this situation, the machine simply transitions to the state q_{rej}. That is, popping an empty stack immediately causes the machine to reject.

The computation of a stack machine M on an input $x \in \Sigma^*$ begins in the initial state q_0 with the input string $x \in \Sigma^*$ is stored in stack 1. Specifically, the top symbol of stack 1 is the first symbol of x, the second to top symbol of stack 1 contains the second symbol of x, and so on. At the bottom of stack 1, underneath all of the input symbols, is the bottom-of-the-stack symbol $\diamond$. All of the other stacks initially contain just the bottom-of-the-stack symbol $\diamond$.

The computation then continues in the natural way so long as the machine is in a non-halting state. If either of the states q_{acc} or q_{rej} is reached, the computation halts, and the input is accepted or rejected accordingly. Of course, just like a Turing machine, there is also a possibility for computations to carry on indefinitely, and in such a situation we will refer to the machine *running forever*.

State diagrams for DSMs

Deterministic stack machines may be represented by state diagrams in a way that is similar to, but in some ways different from, the other models we have discussed. As usual, states are represented by nodes in a directed graph, directed edges (with labels) represent transitions, and the accept and reject states are labeled as such. You will be able to immediately recognize that a state diagram represents a DSM in these notes from the fact that the nodes are square-shaped (with slightly rounded corners) rather than circle or oval shaped.

In the state diagram of a DSM, the nodes themselves, rather than the transitions, indicate which operation (push or pop) is performed as the machine transitions from a given state, and to which stack the operation refers. Each push state must have a single transition leading from it, with the label indicating which symbol is pushed, with the transition pointing to the next state. Each pop state must have one directed edge leading from it for each possible stack symbol, indicating to which state the computation is to transition (depending on the symbol popped).

We will also commonly assign names like X, Y, and Z to different stacks, rather than calling them *stack 1*, *stack 2*, and so on, as this makes for more natural, algorithmically focused descriptions of stack machines, where we view the stacks as being akin to variables in a computer program.

Figure 14.1 gives an example of a state diagram of a 3-DSM. In this diagram, stack 1 (which stores the input when the computation begins) is named X and the other two stacks are named Y and Z. This DSM accepts every string in the language

$$\{w\#w : w \in \{0,1\}^*\} \tag{14.5}$$

and rejects every string in the complement of this language (over the alphabet $\{0,1,\#\}$).

Two additional comments on state diagrams for DSMs are in order. First, aside from the accept and reject states, we tend not to include the names of individual states in state diagrams. This is because the names we choose for the states are irrelevant to the functioning of a given machine, and omitting them makes for less cluttered diagrams. In rare cases in which it is important to include the name of a state in a state diagram, we will just write the state name above or beside its corresponding node.

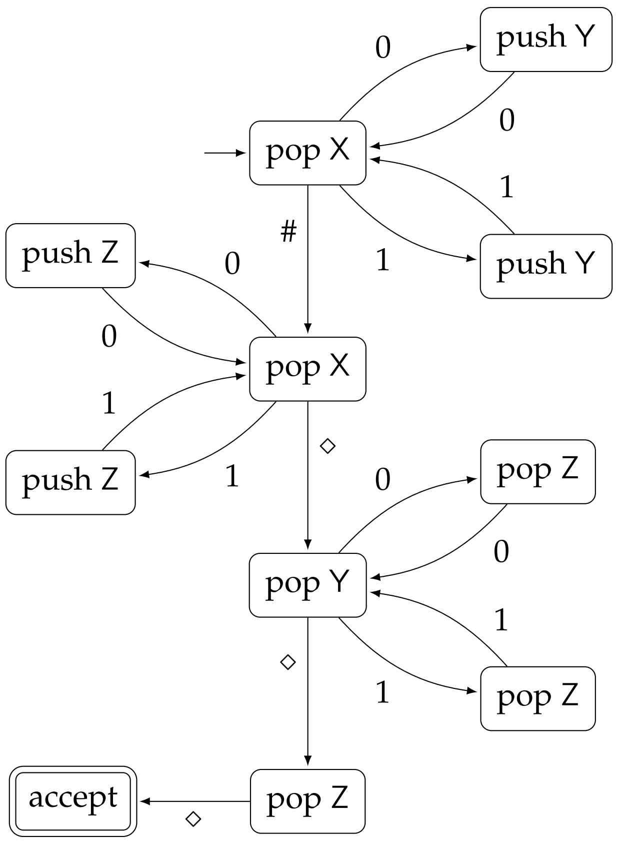

Figure 14.1: A 3-DSM for the language $\{w\#w : w \in \{0,1\}^*\}$.

Second, although every deterministic stack machine is assumed to have a reject state, we often do not bother to include it in state diagrams. Whenever there is a state with which a pop operation is associated, and one or more of the possible stack symbols does not appear on any transition leading out of this state, it is assumed that the "missing" transitions lead to the reject state.

For example, in Figure 14.1, there is no transition labeled $\diamond$ leading out of the initial state, so it is implicit that if $\diamond$ is popped off of X from this state, the machine enters the reject state.

Subroutines

Just like we often do with ordinary programming languages, we can define *subroutines* for stack machines. This can sometimes offer a major simplification to the descriptions of stack machines.

For example, consider the DSM whose state diagram is shown in Figure 14.2.

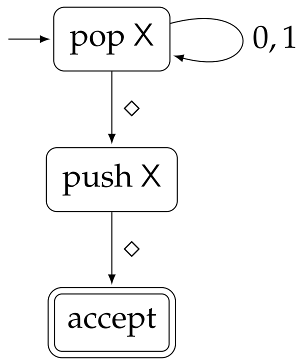

Figure 14.2: An example of a state diagram describing a 1-DSM, whose sole stack is named X. This particular machine is not very interesting from a language-recognition viewpoint—it accepts every string—but it performs the useful task of erasing the contents of a stack. Here the stack alphabet is assumed to be $\Delta = \{0, 1, \diamond\}$, but the idea is easily extended to other stack alphabets.

Before discussing this machine, let us agree that whenever we say that a particular stack *stores* a string x, we mean that the bottom-of-the-stack marker $\diamond$ appears on the bottom of the stack, and the symbols of x appear above this bottom-of-the-stack marker on the stack, with the leftmost symbol of x on the top of the stack. We will never use this terminology in the situation that the symbol $\diamond$ itself appears in x. Using this terminology, the behavior of the DSM illustrated in Figure 14.2 is that if its computation begins with X storing an arbitrary string $x \in \{0, 1\}^*$, then the computation always results in acceptance, with X storing ε. In other words, the DSM erases the string stored by X and halts.

The simple process performed by this DSM might be useful as a subroutine inside of some more complicated DSM, and of course a simple modification allows us to choose any stack in place of X that gets erased. Rather than replicating the description of the DSM from Figure 14.4 inside of this more complicated hypothetical DSM, we can simply use the sort of shorthand suggested by Figure 14.3.

More explicitly, the diagram on the left-hand side of Figure 14.3 suggests a small part of a hypothetical DSM, where the DSM from Figure 14.2 appears inside of the dashed box. Note that we have not included the accept state in the dashed box because, rather than accepting, we wish for control to be passed to the state labeled "another state" as the erasing process completes. We also do not have that the "pop X" state is the initial state any longer, because rather than starting at this state, we have that control passes to this state from the state labeled "some state." There could, in fact, be multiple transitions from multiple states leading to the "pop X" state in the hypothetical DSM being considered.

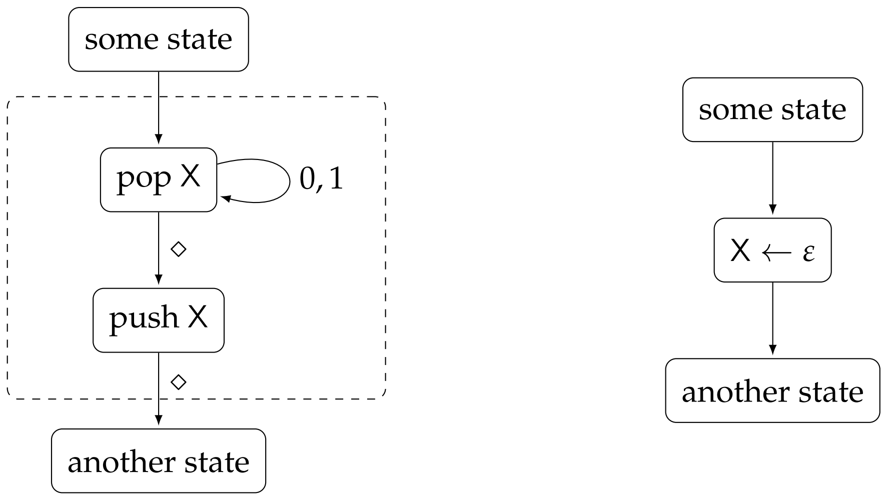

Figure 14.3: The diagrams on the left and right describe equivalent portions of a larger DSM; the contents of the dotted rectangle in the left-hand diagram is viewed as a *subroutine* that is represented by a single rectangle labeled "X ← ε" in the right-hand diagram.

In the diagram on the right-hand side of Figure 14.3 we have replaced the dashed box with a single rectangle labeled "X ← ε." This is just a label that we have chosen, but of course it is a fitting label in this case. The rectangle labeled "X ← ε" looks like a state, and we can think of it as being like a state with which a more complicated operation than push or pop is associated, but the reality is that it is just a short-hand for the contents of the dashed box on the left-hand side diagram.

The same general pattern can be replicated for just about any choice of a DSM. That is, if we have a DSM that we would like to use as a subroutine, we can always do this as follows:

1. Let the original start state of the DSM be the state to which some transition points.
2. Remove the accept state, modifying transitions to this removed accept state so that they point to some other state elsewhere in the larger DSM to which control is to pass once the subroutine is complete.

Naturally, one must be careful when defining and using subroutines like this, as computations could easily become corrupted if subroutines modify stacks that are being used for other purposes elsewhere in a computation. The same thing can, of course, be said concerning subroutines in ordinary computer programs.

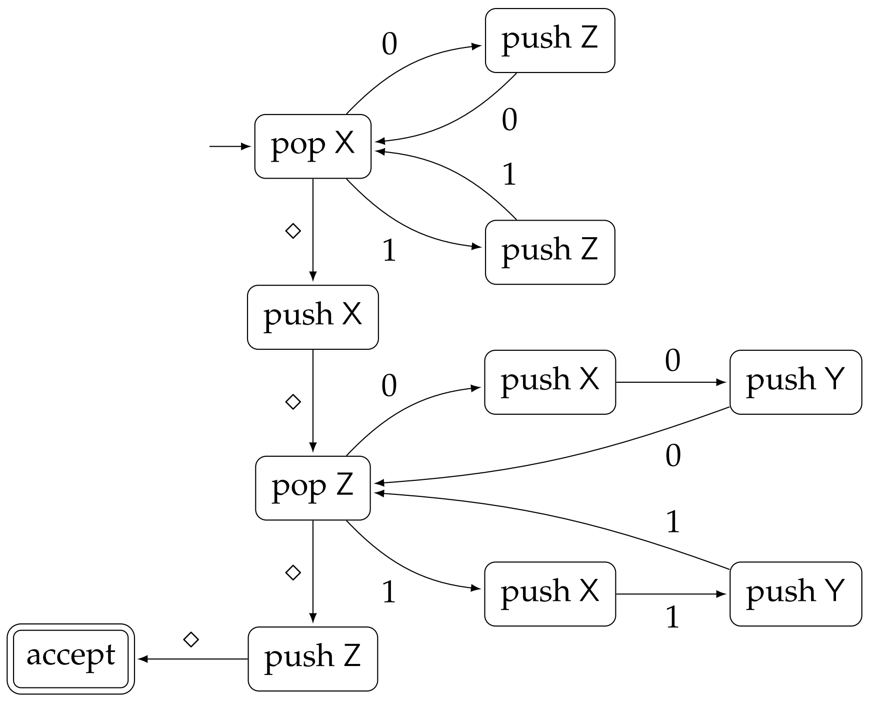

Figure 14.4: An example of a state diagram describing a 3-DSM (with stacks named X, Y, and Z). This machine performs the task of copying the contents of one stack to another: X is copied to Y. The stack Z is used as workspace to perform this operation.

Another example of a subroutine is illustrated in Figure 14.4. This stack machine copies the contents of one stack to another, using a third stack as an auxiliary (or workspace) stack to accomplish this task. Specifically, under the assumption that a stack X stores a string $x \in \{0,1\}^*$ and stacks Y and Z store the empty string, the illustrated 3-DSM will always lead to acceptance—and when it does accept, the stacks X and Y will both store the string x, while Z will revert to its initial configuration in which it stores the empty string. In summary, if initially $(\mathsf{X},\mathsf{Y},\mathsf{Z})$ stores $(x,\varepsilon,\varepsilon)$, then upon certain acceptance $(\mathsf{X},\mathsf{Y},\mathsf{Z})$ will store (x,x,ε).

If we wish to use this DSM as a subroutine in a more complicated DSM, we could again represent the entire DSM (minus the accept state) by a single rectangle, just like we did in Figure 14.3. A fitting label in this case is "$\mathsf{Y} \leftarrow \mathsf{X}$."

One more example of a DSM that is useful as a subroutine is pictured in Figure 14.5. Notice that in this state diagram we have made use of the two previous subroutines to make the figure simpler. After each new subroutine is defined, we are naturally free to use it to describe new DSMs. The DSM in the figure *reverses* the string stored by X. It uses a workspace stack Y to accomplish this task—but in

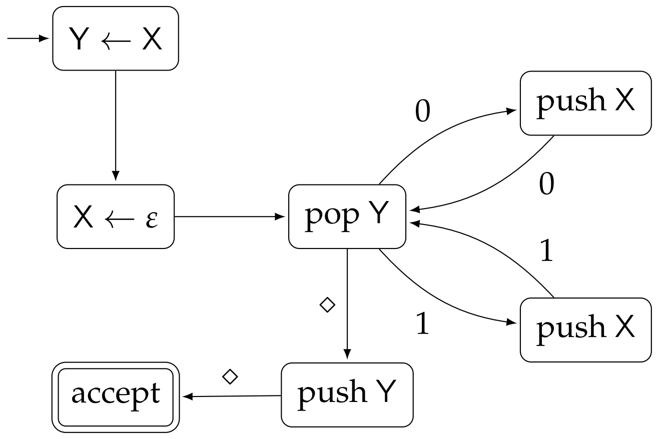

Figure 14.5: A DSM that reverses the string stored by a stack X.

fact it also uses a workspace stack Z, which is hidden inside the subroutine labeled "Y ← X." In summary, it transforms (X, Y, Z) from $(x, \varepsilon, \varepsilon)$ to $(x^{\mathrm{R}}, \varepsilon, \varepsilon)$.

Note, by the way, that we do not in general need to list all of the workspace stacks used by stack machines—we have only done this here to make their use clear. So long as workspace stacks are used correctly, they can safely be ignored.

14.2 Equivalence of DTMs and DSMs

We will now argue that deterministic Turing machines and deterministic stack machines are equivalent computational models. This will require that we establish two separate facts:

1. Given a DTM M, there exists a DSM K that simulates M.
2. Given a DSM M, there exists a DTM K that simulates M.

Just like in the previous lecture, this does not necessarily mean that one step of the original machine corresponds to a single step of the simulator: the simulator might require many steps to simulate one step of the original machine. A consequence of both facts listed above is that, for every input string w, K accepts w whenever M accepts w, K rejects w whenever M rejects w, and K runs forever on w whenever M runs forever on w.

The two simulations are described in the subsections that follow. These descriptions are not intended to be formal proofs, but they should provide enough information to convince you that the two models are indeed equivalent.

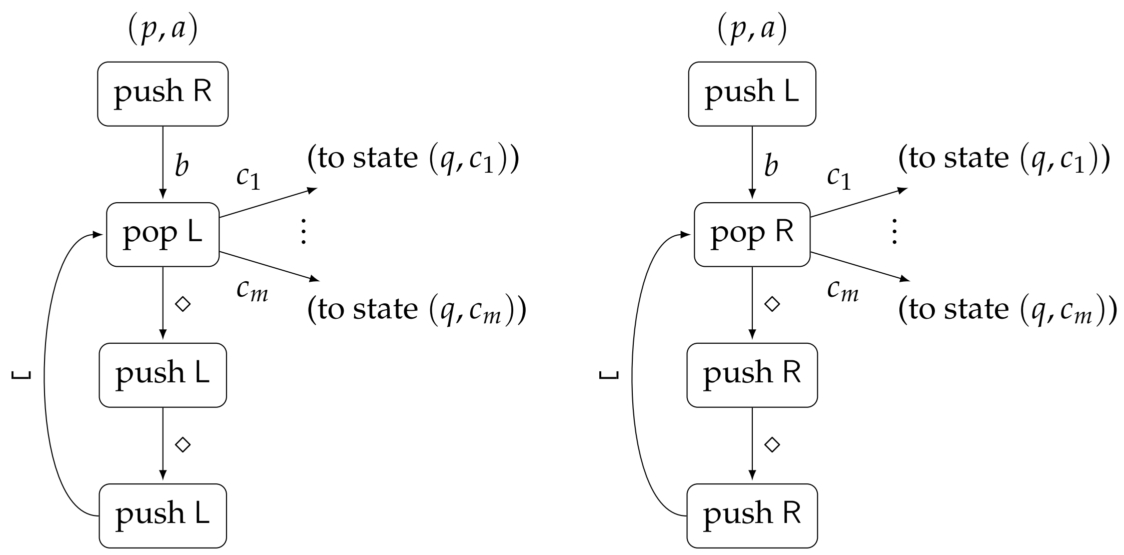

Figure 14.6: For each state/symbol pair $(p, a) \in (Q \backslash \{q_{\text{acc}}, q_{\text{rej}}\}) \times \Gamma$ of M, there are two possibilities: if $\delta(p, a) = (q, b, \leftarrow)$, then K includes the states and transitions in the left-hand diagram, and if $\delta(p, a) = (q, b, \rightarrow)$, then K includes the states and transitions in the right-hand diagram. (The diagrams are symmetric under swapping L and R.)

Simulating a DTM with a DSM

First we will discuss how a DSM can simulate a DTM. To simulate a given DTM M, we will define a DSM K having two stacks, called L and R (for "left" and "right," respectively). The stack L will represent the contents of the tape of M to the left of the tape head (in reverse order, so that the topmost symbol of L is the symbol immediately to the left of the tape head of M) while R will represent the contents of the tape of M to the right of the tape head. The symbol in the tape square of M that is being scanned by its tape head will be stored in the internal state of K, so this symbol does not need to be stored on either stack. Our main task will be to define K so that it pushes and pops symbols to and from L and R in a way that mimics the behavior of M.

To be more precise, suppose that $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ is the DTM to be simulated. The DSM K will require a collection of states for every state/symbol pair $(p, a) \in Q \times \Gamma$. Figure 14.6 illustrates these collections of states in the case that p is a non-halting state. If it is the case that $\delta(p, a) = (q, b, \leftarrow)$, then the states and transitions on the left-hand side of Figure 14.6 mimic the actions of M in this way:

1. The symbol b gets written to the tape of M and the tape head moves left, so K

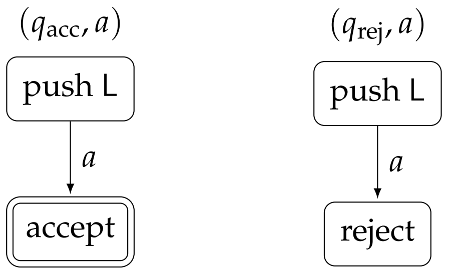

Figure 14.7: For each state/symbol pair $(p, a) \in \{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma$ of M, the DSM K simply transitions to its accept or reject state accordingly. The symbol a stored in the finite state memory of K is pushed onto the stack L as this transition is followed. (The choice to push this symbol onto L rather than R is more or less arbitrary, and this operation is not important if one is only interested in whether M accepts, rejects, or runs forever; this operation only has relevance when the contents of the tape of M when it halts are of interest.)

pushes the symbol b onto R to record the fact that the symbol b is now to the right of the tape head of M.

2. The symbol that was one square to the left of the tape head of M becomes the symbol that M scans because the tape head moved left, so K pops a symbol off of L in order to learn what this symbol is and stores it in its finite state memory. In case K pops the bottom-of-the-stack marker, it pushes this symbol back, pushes a blank, and tries again; this has the effect of inserting extra blank symbols as M moves to previously unvisited tape squares.
3. As K pops the top symbol off of L, as described in the previous item, it transitions to the new state (q, c), for whatever symbol c it popped. This sets up K to simulate the next step of M.

The situation is analogous in the case $\delta(p, a) = (q, b, \rightarrow)$, with left and right (and the stacks L and R) swapped.

For each pair (p, a) where $p \in \{q_{\text{acc}}, q_{\text{rej}}\}$, there is no next-step of M to simulate, so K simply transitions to its accept or reject state accordingly, as illustrated in Figure 14.7. Note that if we only care about whether M accepts or rejects, as opposed to what is left on its tape in case it halts, we could alternatively eliminate all states of the form (q_{acc}, a) and (q_{rej}, a), and replace transitions to these eliminated states with transitions to the accept or reject state of K.

The start state of K is the state $(q_0, \sqcup)$ and it is to be understood that stack R is stack number 1 (and therefore contains the input along with the bottom-of-the-stack marker) while L is stack number 2. The initial state of K therefore represents the initial state of M, where the tape head scans a blank symbol and the input is written in the tape squares to the right of this blank tape square.

Simulating a DSM with a DTM

Now we will explain how a DSM can be simulated by a DTM. The idea behind this simulation is fairly straightforward: the DTM will use its tape to store the contents of all of the stacks of the DSM it is simulating, and it will update this information appropriately so as to mimic the DSM. This will require many steps in general, as the DTM will have to scan back and forth on its tape to manipulate the information representing the stacks of the original DSM.

In greater detail, suppose that $M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ is an r-DSM. The DTM K that we will define to simulate M will have the Cartesian product alphabet:

$$\Gamma = \big(\Delta \cup \{\#, \sqcup\}\big)^r. \tag{14.6}$$

That is, we are thinking about K as having a tape with multiple tracks, just like in the previous lecture. We assume that # is a special symbol that is not contained in the stack alphabet Δ of M, and that the blank symbol $\sqcup$ is also not contained in Δ. Figure 14.8 provides an illustration of how the tape will simulate k stacks. As before, it is to be understood that the true blank symbol of K is the symbol $(\sqcup, \ldots, \sqcup)$, and that an input string $a_1 \cdots a_n \in \Sigma^*$ of M is to be identified with the string of tape symbols

$$(a_1, \sqcup, \ldots, \sqcup) \cdots (a_n, \sqcup, \ldots, \sqcup) \in \Gamma^*. \tag{14.7}$$

The purpose of the symbol # is to mark a position on the tape of K; the contents of the stacks of M will always be to the left of these # symbols. The first thing that K does, before any steps of M are simulated, is to scan the tape (from left to right) to find the end of the input string. In the first tape square after the input string, it places the bottom-of-the-stack marker $\diamond$ in every track, and in the next square to the right it places the # symbol in every track. Once these # symbols are written to the tape, they will remain there for the duration of the simulation. The DTM then moves its tape head to the left, so that it is positioned over the # symbols, and begins simulating steps of the DSM M.

The DTM K will store the current state of M in its internal memory, and one way to think about this is to imagine that K has a collection of states for every state $q \in Q$ of M (which is similar to the simulation in the previous subsection, except there we had a collection of states for every state/symbol pair rather than just for each state). The DTM K is defined so that this state will initially be set to q_0 (the start state of M) when it begins the simulation.

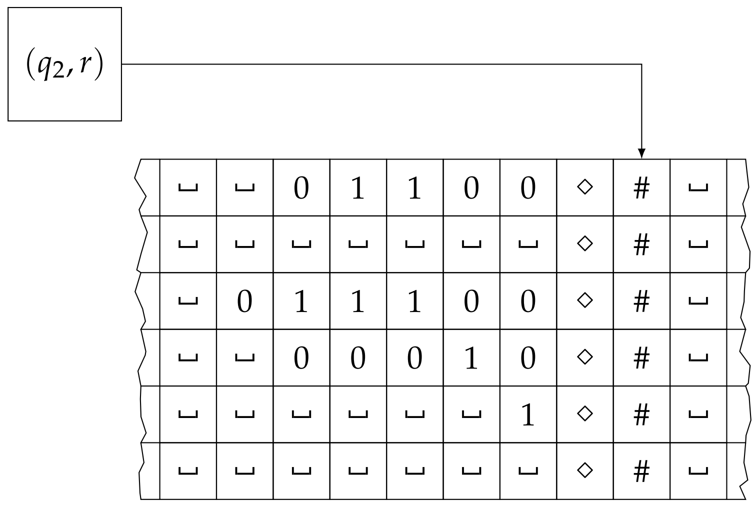

Figure 14.8: An example of a DTM whose tape has 6 tracks, each representing a stack. This figure is consistent with the stack alphabet of the DSM that is being simulated being $\Delta = \{0, 1, \diamond\}$; the situation pictured is that the DSM stacks 0 through 5 store the strings 01100, ε, 011100, 00010, 1, and ε, respectively.

There are two possibilities for each non-halting state $q \in Q$ of M: it is either a push state or a pop state. In either case, there is a stack index k that is associated with this state. The behavior of K is as follows for these two possibilities:

1. If q is a push state, then there must be a symbol $a \in \Gamma$ that is to be pushed onto stack k. The DTM K scans left until it finds a blank symbol on track k, overwrites this blank with the symbol a, and changes the state of M stored in its internal memory exactly as M does.
2. If q is a pop state, then K needs to find out what symbol is on the top of stack k. It scans left to find a blank symbol on track k, moves right to find the symbol on the top of stack k, changes the state of M stored in its internal memory accordingly, and overwrites this symbol with a blank. Naturally, in the situation where M attempts to pop an empty stack, K will detect this (as there will be no non-blank symbols to the left of the # symbols), and it immediately transitions to its reject state.

In both cases, after the push or pop operation was simulated, K scans its tape head back to the right to find the # symbols, so that it can simulate another step of M.

Finally, if K stores a halting state of M when it would otherwise begin simulating a step of M, it accepts or rejects accordingly. In a situation in which the contents of the tape of K after the simulation are important, such as when M computes a function rather than simply accepting or rejecting, one may of course define K so that it first removes the # symbols and $\diamond$ symbols from its tape prior to accepting or rejecting.