Lecture 14

Turing machines and their equivalence to stack machines

In this lecture we will discuss the *Turing machine* model of computation. This model is named after Alan Turing (1912–1954), who proposed it in 1936. It is difficult to overstate the importance of Alan Turing’s work to this course; the subject of theoretical computer science effectively started with Turing’s work, and for this reason he is sometimes referred to as the father of theoretical computer science.

The intention of the Turing machine model is to provide a simple mathematical abstraction of general computations. This is also the intention of the stack machine model, and we will soon see that the two models are in fact equivalent—so it is simply a matter of preference to base the theory of computability on one model or the other, or even on a different model that is equivalent to Turing machines and stack machines.\(^1\) It should be made clear, however, that Turing machines do, in fact, reign supreme: in most courses similar to CS 360 taught around the world, there is no mention at all of stack machines, and it is just the Turing machine that is studied in the context of computability theory.

The idea that Turing machine computations are representative of a fully general computational model is called the *Church–Turing thesis*. Here is one statement of this thesis (although it is the idea rather than the exact choice of words that is important):

**Church–Turing thesis:** Any function that can be computed by a mechanical process can be computed by a Turing machine.

Note that this is not a mathematical statement that can be proved or disproved. If you wanted to try to prove a statement along these lines, the first thing you

\(^1\) One well-known example is λ-calculus, which was proposed by Alonzo Church a short time before Turing proposed the Turing machine. A sketch of a proof of the equivalence of Turing machines and λ-calculus appeared in Turing’s 1936 paper.
would most likely do is to look for a mathematical definition of what it means for a function to be “computed by a mechanical process,” and this is precisely what the Turing machine model was intended to provide.

While people have actually built machines that behave like Turing machines, this is mostly a recreational activity. The Turing machine model was never intended to serve as a practical device for performing computations, but rather was intended to provide a rigorous mathematical foundation for reasoning about computation, and it has served this purpose very well since its introduction.

14.1 Turing machine definitions

We will begin with an informal description of the Turing machine model before stating the formal definition. There are three components of a Turing machine:

1. The finite state control. This component is in one of a finite number of states at each instant, and is connected to the tape head component.

2. The tape head. This component scans one of the tape squares of the tape at each instant, and is connected to the finite state control. It can read and write symbols from/to the tape, and it can move left and right along the tape.

3. The tape. This component consists of an infinite number of tape squares, each of which can store one of a finite number of tape symbols at each instant. The tape is infinite both to the left and to the right.

Figure 14.1 illustrates these three components and the way they are connected.

The idea is that the action of a Turing machine at each instant is determined by the state of the finite state control together with just the one symbol that is
stored in the tape square that the tape head is currently reading. Thus, the action is determined by a finite number of possible alternatives: one action for each state/symbol pair. Depending on the state and the symbol being scanned, the action that the machine is to perform may involve changing the state of the finite state control, changing the symbol on the tape square being scanned, and moving the tape head to the left or right. Once this action is performed, the machine will again have some state for its finite state control and will be reading some symbol on its tape, and the process continues. Just like the stack machine model, one may consider both deterministic and nondeterministic variants of the Turing machine model.

When a Turing machine begins a computation, an input string is written on its tape, and every other tape square is initialized to a special blank symbol (which may not be included in the input alphabet). Naturally, we need an actual symbol to represent the blank symbol in these notes, and we will use the symbol \( \omega \) for this purpose. More generally, we will allow the possible symbols written on the tape to include other non-input symbols in addition to the blank symbol, as it is sometimes convenient to allow this possibility.

Similar to stack machines, we require that Turing machines have two special states: an accept state \( q_{\text{acc}} \) and a reject state \( q_{\text{rej}} \). If the machine enters one of these two states, the computation immediately stops and accepts or rejects accordingly. When we discuss language recognition, our focus is on whether or not a given Turing machine eventually reaches one of the states \( q_{\text{acc}} \) or \( q_{\text{rej}} \), but we can also use the Turing machine model to discuss function computations by taking into account the contents of the tape after the accept state (let us say) has been reached.

**Formal definition of DTMs**

With the informal description of Turing machines from above in mind, we will now proceed to the formal definition.

**Definition 14.1.** A deterministic Turing machine (or DTM, for short) is a 7-tuple

\[
M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}),
\]

where \( Q \) is a finite and nonempty set of states; \( \Sigma \) is an alphabet called the input alphabet, which may not include the blank symbol \( \omega \); \( \Gamma \) is an alphabet called the tape alphabet, which must satisfy \( \Sigma \cup \{\omega\} \subseteq \Gamma \); \( \delta \) is a transition function having the form

\[
\delta : Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \rightarrow\};
\]

\( q_0 \in Q \) is the initial state; and \( q_{\text{acc}}, q_{\text{rej}} \in Q \) are the accept and reject states, which satisfy \( q_{\text{acc}} \neq q_{\text{rej}} \).
The interpretation of the transition function is as follows. Suppose the DTM is currently in a state \( p \in Q \), the symbol stored in the tape square being scanned by the tape head is \( a \in \Gamma \), and it is the case that \( \delta(p, a) = (q, b, D) \) for \( D \in \{ \leftarrow, \rightarrow \} \). The action performed by the DTM is then to (i) change state to \( q \), (ii) overwrite the contents of the tape square being scanned by the tape head with \( b \), and (iii) move the tape head in direction \( D \) (either left or right). In the case that the state is \( q_{\text{acc}} \) or \( q_{\text{rej}} \), the transition function does not specify an action, because we assume that the DTM halts once it reaches one of these two states.

**Turing machine computations**

If we have a DTM \( M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \), and we wish to consider its operation on some input string \( w \in \Sigma^* \), we assume that it is started with its components initialized as illustrated in Figure 14.2. That is, the input string is written on the tape, one symbol per square, with each other tape square containing the blank symbol, and with the tape head scanning the tape square immediately to the left of the first input symbol. (In the case that the input string is \( \varepsilon \), all of the tape squares start out storing blanks.)

Once the initial arrangement of the DTM is set up, the DTM begins taking steps, as determined by the transition function \( \delta \) in the manner suggested above. So long as the DTM does not enter one of the two states \( q_{\text{acc}} \) or \( q_{\text{rej}} \), the computation continues. If the DTM eventually enters the state \( q_{\text{acc}} \), it accepts the input string, and if it eventually enters the state \( q_{\text{rej}} \), it rejects the input string. Thus, there are three possible alternatives for a DTM \( M \) on a given input string \( w \):

1. \( M \) accepts \( w \).
2. \( M \) rejects \( w \).
3. \( M \) runs forever on input \( w \).

In some cases one can design a particular DTM so that the third alternative does not occur, but in general it might.
Representing configurations of DTMs

In order to speak more precisely about Turing machines and state a formal definition concerning their behavior, we will require a bit more terminology. When we speak of a configuration of a DTM, we are speaking of a description of all of the Turing machine’s components at some instant. This includes

1. the state of the finite state control,
2. the contents of the entire tape, and
3. the tape head position on the tape.

Rather than drawing pictures depicting the different parts of Turing machines, like in Figure 14.2, we use the following compact notation to represent configurations. If we have a DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$, and we wish to refer to a configuration of this DTM, we express it in the form

$$u(q, a)v$$

for some state $q \in Q$, a tape symbol $a \in \Gamma$, and (possibly empty) strings of tape symbols $u$ and $v$ such that

$$u \in \Gamma^* \setminus \{\omega\} \Gamma^* \text{ and } v \in \Gamma^* \setminus \Gamma^* \setminus \{\omega\}$$

(i.e., $u$ and $v$ are strings of tape symbols such that $u$ does not start with a blank and $v$ does not end with a blank). What the expression (14.3) means is that the string $uav$ is written on the tape in consecutive squares, with all other tape squares containing the blank symbol; the state of $M$ is $q$; and the tape head of $M$ is positioned over the symbol $a$ that occurs between $u$ and $v$.

For example, the configuration of the DTM in Figure 14.1 is expressed as

$$0\$1(q_4, 0)0\#$$

while the configuration of the DTM in Figure 14.2 is

$$(q_0, \omega)w$$

(for $w = a_1 \cdots a_n$).

When working with descriptions of configurations, it is convenient to define a few functions as follows. We define $\alpha : \Gamma^* \to \Gamma^* \setminus \{\omega\} \Gamma^*$ and $\beta : \Gamma^* \to \Gamma^* \setminus \Gamma^* \setminus \{\omega\}$ recursively as

$$\alpha(w) = w \quad \text{ (for } w \in \Gamma^* \setminus \{\omega\} \Gamma^*)$$
$$\alpha(\omega w) = \alpha(w) \quad \text{ (for } w \in \Gamma^*)$$

(14.7)
and
\[ \beta(w) = w \quad (\text{for } w \in \Gamma^* \setminus \{\omega\}) \]
\[ \beta(w\omega) = \beta(w) \quad (\text{for } w \in \Gamma^*), \]
and we define
\[ \gamma : \Gamma^*(Q \times \Gamma)^* \rightarrow (\Gamma^* \setminus \{\omega\}\Gamma^*) \times (Q \times \Gamma) \times (\Gamma^* \setminus \{\omega\}) \]
(14.9)
as
\[ \gamma(u \cdot q \cdot a \cdot v) = \alpha(u)(q, a) \beta(v) \]
(14.10)
for all \( u, v \in \Gamma^* \), \( q \in Q \), and \( a \in \Gamma \). This is not as complicated as it might appear: the function \( \gamma \) just throws away all blank symbols on the left-most end of \( u \) and the right-most end of \( v \), so that a proper expression of a configuration remains.

### A Yields Relation for DTMs

Now we will define a yields relation, in a similar way to what we did for context-free grammars and stack machines. This will in turn allow us to formally define acceptance and rejection for DTMs.

**Definition 14.2.** Let \( M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej}) \) be a DTM. We define a yields relation \( \vdash_M \) on pairs of expressions representing configurations as follows:

1. For every choice of \( p \in Q \setminus \{q_{acc}, q_{rej}\}, q \in Q \), and \( a, b \in \Gamma \) satisfying
\[ \delta(p, a) = (q, b, \rightarrow), \]
the yields relation includes these pairs for all \( u \in \Gamma^* \setminus \{\omega\}^*, v \in \Gamma^* \setminus \{\omega\}, \)
and \( c \in \Gamma \):
\[ u(p, a) \vdash_M \gamma(u(b, c)v) \]
\[ u(p, a) \vdash_M \gamma(ub(q, \omega)) \]
(14.12)

2. For every choice of \( p \in Q \setminus \{q_{acc}, q_{rej}\}, q \in Q \), and \( a, b \in \Gamma \) satisfying
\[ \delta(p, a) = (q, b, \leftarrow), \]
the yields relation includes these pairs for all \( u \in \Gamma^* \setminus \{\omega\}^*, v \in \Gamma^* \setminus \{\omega\}, \)
and \( c \in \Gamma \):
\[ uc(p, a) v \vdash_M \gamma(u(q, c)b v) \]
\[ (p, a) v \vdash_M \gamma((q, \omega)b v) \]
(14.14)
In addition, we let $\vdash^* M$ denote the reflexive, transitive closure of $\vdash M$. That is, we have

$$u(p,a)v \vdash^* M y(q,b)z$$

if and only if there exists an integer $m \geq 1$, strings $w_1, \ldots, w_m, x_1, \ldots, x_m \in \Gamma^*$, symbols $c_1, \ldots, c_m \in \Gamma$, and states $r_1, \ldots, r_m \in Q$ such that $u(p,a)v = w_1(r_1, c_1)x_1$, $y(q,b)v = w_m(r_m, c_m)x_m$, and

$$w_k(r_k, c_k)x_k \vdash_M w_{k+1}(r_{k+1}, c_{k+1})x_{k+1}$$

for all $k \in \{1, \ldots, m - 1\}$.

A somewhat more intuitive explanation of this definition is as follows. Whenever we have

$$u(p,a)v \vdash M y(q,b)z$$

it means that by running $M$ for one step we move from the configuration represented by $u(p,a)v$ to the configuration represented by $y(q,b)z$; and whenever we have

$$u(p,a)v \vdash^* M y(q,b)z$$

it means that by running $M$ for some number of steps, possibly zero steps, we will move from the configuration represented by $u(p,a)v$ to the configuration represented by $y(q,b)z$.

**Acceptance and rejection for DTMs**

Finally, we can write down a definition for acceptance and rejection by a DTM, using the relation $\vdash^* M$ we just defined.

**Definition 14.3.** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be a DTM and let $w \in \Sigma^*$ be a string. If there exist strings $u, v \in \Gamma^*$ and a symbol $a \in \Gamma$ such that

$$(q_0, \omega)w \vdash^*_M u(q_{\text{acc}}, a)v,$$

then $M$ accepts $w$. If there exist strings $u, v \in \Gamma^*$ and a symbol $a \in \Gamma$ such that

$$(q_0, \omega)w \vdash^*_M u(q_{\text{rej}}, a)v,$$

then $M$ rejects $w$. If neither of these conditions hold, then $M$ runs forever on input $w$.

In words, if a DTM is set in its initial configuration, for some input string $w$, and starts running, it accepts $w$ if it eventually enters its accept state, it rejects $w$ if it eventually enters its reject state, and it runs forever if neither of these possibilities holds.
Similar to what we have done for other computational models, we write \( L(M) \) to denote the language of all strings that are accepted by a DTM \( M \). As for stack machines, the language \( L(M) \) doesn’t really tell the whole story, because a string \( w \notin L(M) \) might either be rejected or it may cause \( M \) to run forever, but the notation is useful nevertheless.

A simple example of a Turing machine

Let us now see an example of a DTM, which we will describe using a state diagram. In the DTM case, we represent the property that the transition function satisfies \( \delta(p, a) = (q, b, \rightarrow) \) with a transition of the form

\[
\begin{array}{c}
\text{p} \\
\end{array} \quad a, b \rightarrow \quad \begin{array}{c}
\text{q} \\
\end{array}
\]

and similarly we represent the property that \( \delta(p, a) = (q, b, \leftarrow) \) with a transition of the form

\[
\begin{array}{c}
\text{p} \\
\end{array} \quad \leftarrow a, b \quad \begin{array}{c}
\text{q} \\
\end{array}
\]

Figure 14.3: A DTM \( M \) for the language \( \{0^n1^n : n \in \mathbb{N}\} \).
The state diagram for the example is given in Figure 14.3. The DTM $M$ described by this diagram is for the language

$$A = \{0^n1^n : n \in \mathbb{N}\}. \quad (14.21)$$

To be more precise, $M$ accepts every string in $A$ and rejects every string in $\overline{A}$.

The specific way that the DTM $M$ works can be summarized as follows. The DTM $M$ starts out with its tape head scanning the blank symbol immediately to the left of its input. It moves the tape head right, and if it sees a 1 it rejects: the input string must not be of the form $0^n1^n$ if this happens. On the other hand, if it sees another blank symbol, it accepts: the input must be the empty string, which corresponds to the $n = 0$ case in the description of $A$. Otherwise, it must have seen the symbol 0, and in this case the 0 is erased (meaning that it replaces it with the blank symbol), the tape head repeatedly moves right until a blank is found, and then it moves one square back to the left. If a 1 is not found at this location the DTM rejects: there weren’t enough 1s at the right end of the string. Otherwise, if a 1 is found, it is erased, and the tape head moves all the way back to the left, where we essentially recurse on a slightly shorter string.

Of course, the summary just suggested doesn’t tell you precisely how the DTM works—but if you didn’t already have the state diagram from Figure 14.3, the summary would probably be enough to give you a good idea for how to come up with the state diagram (or perhaps a slightly different one operating in a similar way).

In fact, an even higher-level summary would probably be enough. For instance, we could describe the functioning of the DTM $M$ as follows:

1. Accept if the input is the empty string.
2. Check that the left-most non-blank symbol on the tape is a 0 and that the right-most non-blank symbol is a 1. Reject if this is not the case, and otherwise erase these symbols and goto 1.

There will, of course, be several specific ways to implement this algorithm with a DTM, with the DTM $M$ from Figure 14.3 being one of them.

Because state diagrams for DTMs tend to be complicated (and often completely incomprehensible) for all but the simplest of DTMs, it is very common that DTMs are described in a high level way, as in the last description above.
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 $M$.

Here we have used the term *simulate*, as we quite frequently will throughout the remainder of the course. It refers to the situation in which one machine (the simulator) mimics another machine (which we’ll call the original machine, for lack of a better name). Note that 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.

Simulating a DTM with a DSM

First we will discuss how a DSM can simulate a DTM. To simulate a given a 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 $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ 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 14.4 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 14.4 mimic the actions of $M$ in this way:
For each state/symbol pair \((p, a) \in (Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}) \times \Gamma\) 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.

1. The symbol \(b\) gets written to the tape of \(M\) and the tape head moves left, so \(K\) 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 14.5. 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 \((q_{\text{acc}}, a)\) and \((q_{\text{rej}}, a)\), 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 \((q_0, \omega)\) and it is to be understood that stack \(R\) is stack 0 (and therefore contains the input along with the bottom-of-the-stack marker) while \(L\) is stack 1. 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 a large tape alphabet:

\[
\Gamma = (\Delta \cup \{\#, \omega\})^r.
\]

Here, we assume that \# is a special symbol that is not contained in the stack alphabet \(\Delta\) of \(M\), and that the blank symbol \(\omega\) is also not contained in \(\Delta\). A helpful way to think about a DTM whose tape alphabet is a Cartesian product like this is to imagine that its tape is divided into *tracks*, as Figure 14.6 suggests. Note that we
Figure 14.6: 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, \( \varepsilon \), 011100, 00010, 1, and \( \varepsilon \), respectively.

are not modifying the definition of a DTM at all when we speak of separate tracks on a tape like this—it’s just a way of thinking about the tape alphabet \( \Gamma \). It is to be understood that the true blank symbol of \( K \) is the symbol \( (\underline{\_} , . . . , \underline{\_}) \), and that an input string \( a_1 \cdots a_n \in \Sigma^* \) of \( M \) is to be identified with the string of tape symbols

\[
(a_1, \underline{\_}, \ldots, \underline{\_}) \cdots (a_n, \underline{\_}, \ldots, \underline{\_}) \in \Gamma^*.
\]

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.
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.