第三章

3.1

a. $q_1 0, \sqcup q_2 \sqcup, \sqcup\sqcup q_{\text{accept}}$

b. $q_1 00, \sqcup q_2 0, \sqcup x q_3 \sqcup, \sqcup q_5 x \sqcup, q_5 \sqcup x \sqcup, \sqcup q_2 x \sqcup, \sqcup x q_2 \sqcup, \sqcup x \sqcup q_{\text{accept}}$

c. $q_1 000, \sqcup q_2 00, \sqcup x q_3 0, \sqcup x 0 q_4 \sqcup, \sqcup x 0 \sqcup q_{\text{reject}}$

d. $q_1 000000, \sqcup q_2 00000, \sqcup x q_3 0000 \sqcup, \sqcup x 0 q_4 000, \sqcup x 0 x q_3 00, \sqcup x 0 x 0 q_4 0,$
$\sqcup x 0 x 0 x q_3 \sqcup, \sqcup x 0 x 0 q_5 x \sqcup, \sqcup x 0 x q_5 0 x \sqcup, \sqcup x 0 q_5 x 0 x \sqcup, \sqcup x q_5 0 x 0 x \sqcup, \sqcup q_5 x 0 x 0 x \sqcup,$
$q_5 \sqcup x 0 x 0 x \sqcup, \sqcup q_2 x 0 x 0 x \sqcup, \sqcup x q_2 0 x 0 x \sqcup, \sqcup x x q_3 x 0 x \sqcup, \sqcup x x x q_3 0 x \sqcup, \sqcup x x x 0 q_4 x \sqcup,$
$\sqcup x x x 0 x q_4 \sqcup, \sqcup x x x 0 x \sqcup q_{\text{reject}}$

3.2

a. $q_1 11, \sqcup q_3 1; \sqcup 1 q_3 \sqcup, \sqcup 1 \sqcup q_{\text{reject}}$

b. $q_1 1\#1, \sqcup q_3 \#1, \sqcup \# q_5 1, \sqcup \# 1 q_5 \sqcup, \sqcup \# q_7 1 \sqcup, \sqcup q_7 \# 1 \sqcup, q_7 \sqcup \# 1 \sqcup, \sqcup q_9 \# 1 \sqcup, \sqcup \# q_{11} 1 \sqcup,$
$\sqcup q_{12} \# x \sqcup, q_{12} \sqcup \# x \sqcup, \sqcup q_{13} \# x \sqcup, \sqcup \# q_{14} x \sqcup, \sqcup \# x q_{14} \sqcup, \sqcup \# x \sqcup q_{\text{accept}}$

c. $q_1 1\#\#1, \sqcup q_3 \#\#1, \sqcup \# q_5 \#1, \sqcup \#\# q_{\text{reject}} 1$

d. $q_1 10\#11, \sqcup q_3 0\#11, \sqcup 0 q_3 \#11, \sqcup 0 \# q_5 11, \sqcup 0 \# 1 q_5 1, \sqcup 0 \# 11 q_5 \sqcup, \sqcup 0 \# 1 q_7 1 \sqcup,$
$\sqcup 0 \# q_7 11 \sqcup, \sqcup 0 q_7 \# 11 \sqcup, \sqcup q_7 0 \# 11 \sqcup, q_7 \sqcup 0 \# 11 \sqcup, \sqcup q_9 0 \# 11 \sqcup, \sqcup 0 q_9 \# 11 \sqcup, \sqcup 0 \# q_{11} 11 \sqcup,$
$\sqcup 0 q_{12} \# x 1 \sqcup, \sqcup q_{12} 0 \# x 1 \sqcup, q_{12} \sqcup 0 \# x 1 \sqcup, \sqcup q_{13} 0 \# x 1 \sqcup, \sqcup x q_8 \# x 1 \sqcup, \sqcup x \# q_{10} x 1 \sqcup,$
$\sqcup x \# x q_{10} 1 \sqcup, \sqcup x \# x 1 q_{\text{reject}} \sqcup$

e. $q_1 10\#10, \sqcup q_3 0\#10, \sqcup 0 q_3 \#10, \sqcup 0 \# q_5 10, \sqcup 0 \# 1 q_5 0, \sqcup 0 \# 10 q_5 \sqcup, \sqcup 0 \# 1 q_7 0 \sqcup,$
$\sqcup 0 \# q_7 10 \sqcup, \sqcup 0 q_7 \# 10 \sqcup, \sqcup q_7 0 \# 10 \sqcup, q_7 \sqcup 0 \# 10 \sqcup, \sqcup q_9 0 \# 10 \sqcup, \sqcup 0 q_9 \# 10 \sqcup, \sqcup 0 \# q_{11} 10 \sqcup,$
$\sqcup 0 q_{12} \# x 0 \sqcup, \sqcup q_{12} 0 \# x 0 \sqcup, q_{12} \sqcup 0 \# x 0 \sqcup, \sqcup q_{13} 0 \# x 0 \sqcup, \sqcup x q_8 \# x 0 \sqcup, \sqcup x \# q_{10} x 0 \sqcup,$
$\sqcup x \# x q_{10} 0 \sqcup, \sqcup x \# q_{12} x x \sqcup, \sqcup x q_{12} \# x x \sqcup, \sqcup q_{12} x \# x x \sqcup, q_{12} \sqcup x \# x x \sqcup, \sqcup q_{13} x \# x x \sqcup,$
$\sqcup x q_{13} \# x x \sqcup, \sqcup x \# q_{14} x x \sqcup, \sqcup x \# x q_{14} x \sqcup, \sqcup x \# x x q_{14} \sqcup, \sqcup x \# x x \sqcup q_{\text{accept}}$

3.3 If a language L is decidable, it can be decided by a deterministic TM and that is automatically a nondeterministic TM.

If a language L is decided by a nondeterministic TM N, we construct a deterministic TM D_2 that decides L. TM D_2 uses the same algorithm as in the TM D described in the proof of Theorem 3.10, with an additional step 5: *Reject* if all branches of nondeterminism of N are exhausted.

We argue that D_2 is a decider for L. If N accepts, D_2 will eventually find an accepting branch and accept, too. If N rejects, all of its branches halt and reject. Therefore, by the theorem on trees given in the exercise, its entire computation tree is finite, and D will halt and reject when this entire tree has been explored.

3.4 An enumerator is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{print}}, q_{\text{accept}})$, where Q, Σ, Γ are all finite sets and

i) Q is the set of states,

ii) Γ is the work tape alphabet,

iii) Σ is the output tape alphabet,

iv) $\delta\colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\} \times \Sigma_\varepsilon$ is the transition function,

v) $q_0 \in Q$ is the start state,

vi) $q_{\text{print}} \in Q$ is the print state, and

vii) $q_{\text{accept}} \in Q$ is the reject state, where $q_{\text{print}} \neq q_{\text{reject}}$.

The computation of an enumerator E is defined as in an ordinary TM, except for the following points. It has two tapes, a work tape and a print tape, both initially blank. At each step, the machine may write a symbol from Σ on the output tape, or nothing, as determined by δ. If $\delta(q, a) = (r, b, \mathrm{L}, c)$, it means that in state q, reading a, enumerator E enters state r, writes b on the work tape, moves the work tape head left (or right, if L had been R), writes c on the output tape, and moves the output tape head to the right if $c \neq \varepsilon$.

Whenever state q_{print} is entered, the output tape is reset to blank and the head returns to the left-hand end. The machine halts when q_{accept} is entered. $L(E) = \{w \in \Sigma^* | \ w$ appears on the work tape if q_{print} is entered$\}$.

3.5

a. Yes. Γ is the tape alphabet. A Turing Machine can write any characters in Γ on its tape, and $\sqcup \in \Gamma$ according to the definition.

b. No. Σ never contains $\sqcup$ but Γ always contains $\sqcup$. So, they cannot be equal.

c. Yes. If the Turing Machine tries to move its head off the left-hand end of the tape, it remains on the same tape cell.

d. No. Any Turing Machine must contain two distinct states q_{accept} and q_{reject}. So, a Turing Machine contains at least two states.

3.6

In Stage 2 of this algorithm: "Run M on s_i", if M loops on a certain input s_i, E would not check any inputs after s_i. If that were to occur, E might fail to enumerate $L(M)$ as required.

3.7

The variables $x_1, \ldots, x_k$ have infinitely may possible settings. A Turing machine would required infinite time to try them all. But, we require that every stage in the Turing machine description be completed in a finite number of steps.

3.8

a. "On input string w:

1. Scan the tape and mark the first 0 which has not been marked. If there is no unmarked 0, go to stage 4. Otherwise, move the head back to the front of the tape.
2. Scan the tape and mark the first 1 which has not been marked. If there is no unmarked 1, *reject*.
3. Move the head back to the front of the tape and repeat stage 1.
4. Move the head back to the front of the tape. Scan the tape to see if any unmarked 1s remain. If there are none, *accept*. Otherwise, *reject*."

b. "On input string w:

1. Scan the tape and mark the first 0 which has not been marked. If there is no unmarked 0, go to stage 4.

2. Move on and mark the next unmarked 0. If there is not any on the tape, *reject*. Otherwise, move the head back to the front of the tape.
3. Scan the tape and mark the first 1 which has not been marked. If there is no unmarked 1, *reject*.
4. Move the head back to the front of the tape and repeat stage 1.
5. Move the head back to the front of the tape. Scan the tape to see if there are any unmarked 1s. If there is not, *accept*. Otherwise, *reject*."

c. "On input string w:

1. Scan the tape and mark the first 0 which has not been marked. If there is no unmarked 0, go to stage 4.
2. Move on and mark the next unmarked 0. If there is not any on the tape, *accept*. Otherwise, move the head back to the front of the tape.
3. Scan the tape and mark the first 1 which has not been marked. If there is no unmarked 1, *accept*.
4. Move the head back to the front of the tape and repeat stage 1.
5. Move the head back to the front of the tape. Scan the tape to see if there are any unmarked 1s. If there is not, *reject*. Otherwise, *accept*."

3.9 **a.** By Example 2.20, no PDA recognizes $B = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. The following 2-PDA recognizes B. Push all the a's that appear in the front of the input tape to stack 1. Then push all the b's that follow the a's in the input stream into stack 2. If it sees any a's at this stage, *reject*. When it sees the first c, pop stack 1 and stack 2 at the same time. After this, reject the input if it sees any a's or b's in the input stream. Pop stack 1 and stack 2 for each input character c it reads. If the input ends when both stacks are empty, *accept*. Otherwise, *reject*.

b. We show that a 2-PDA can simulate a TM. Furthermore, a 3-tape NTM can simulate a 3-PDA, and an ordinary deterministic, 1-tape TM can simulate a 3-tape NTM. Therefore a 2-PDA can simulate a 3-PDA, and so they are equivalent in power.

The TM by 2-PDA simulation is done as follows. We split the tape of a TM into two stacks. Stack 1 stores the characters on the left of the head, with the bottom of stack storing the leftmost character of the tape in the TM. Stack 2 stores the characters on the right of the head, with the bottom of the stack storing the rightmost character on the tape in the TM. In other words, if a TM configuration is aq_ib, the corresponding 2-PDA is in state q_i, with stack 1 storing the string a and stack 2 storing the string b^R, both from bottom to top.

For each transition $\delta(q_i, c_i) = (q_j, c_j, \mathrm{L})$ in the TM, the corresponding PDA transition pops c_i off stack 2, pushes c_j into stack 2, pops stack 1 and pushes the character into stack 2, and goes from state q_i to q_j. For any transition $\delta(q_i, c_i) = (q_j, c_j, \mathrm{R})$ in the TM, the corresponding PDA transition pops c_i off stack 2 and takes away c_i, push c_j into stack 1, and goes from state q_i to q_j.

3.10 We first simulate an ordinary TM by a write-twice TM. The write-twice TM simulates a single step of the original TM by copying the entire tape over to a fresh section of the tape to the right of the currently used section. The copying procedure operates character by character, marking a character as it is copied. This procedure alters each tape square twice, once to write the character for the first time and another time to mark that it is copied. The position of the original TMs tape head is marked on the tape. When copying the cells around the the marked position, the tape contents is updated according to the rules of the TM.

To carry out the simulation with a write-once TM, operate as before, except that each cell of the tape is represented by two cells. The first of these contains the original TM's tape contents and the second is for the mark used in the copying procedure. The input is not presented to the machine in the format with two cells per symbol, so the very first time the tape is copied, the copying marks are put directly over the input symbols.

3.11 A Turing machine with doubly infinite tape can easily simulate an ordinary Turing Machine. It needs to mark the left-hand end of the input so that it can prevent the head from moving off of that end.

To simulate the doubly infinite tape TM by an ordinary TM, we show how to simulate it with a 2-tape TM, which was already shown to be equivalent in power to an ordinary TM. The first tape of the 2-tape TM is written with the input string and the second tape is blank. We cut the tape of the doubly infinite tape TM into two parts, at the starting cell of the input string. The portion with the input string and all the blank spaces to its right appears on the first tape of the 2-tape TM. The portion to the left of the input string appears on the second tape, in reverse order.

3.12 We simulate an ordinary TM with a reset TM has only the RESET and R operations. When the ordinary TM moves its head right, the reset TM does the same. When the ordinary TM moves its head left, the reset TM cannot, so it gets the same effect by marking the current head location on the tape, then resetting and copying the entire tape one cell to the right, except for the mark, which is kept on the same tape cell. Then it resets again, and scans right until it find the mark.

3.13 We simulate this TM variant by an NFA. The NFA only needs to keep track of the TM's current state and what the TMs written on the current tape cell. Remembering what it has written on tape cells to the left of the current head position is unnecessary, because the TM is unable to return to these cells an read them. Using an NFA in the actual construction is convenient because it allows ϵ moves which are useful for simulating the "stay put" TM transitions.
More formally, the NFA has $|Q \times \Gamma| + 3$ states. The first $|Q \times \Gamma|$ states correspond to the state of the TM variant and the character on the tape that the head of the TM points to. The extra states are the start state q_{start}, a special accept state q_{accept} and a special reject state q_{reject}. The transition function δ' for the NFA is constructed according δ. First we set $\delta'(q_{\text{start}}, p) = \{q_{0p}\}$, where q_0 is the start state of the TM variant. Next, we set $\delta'(q_{\text{accept}}, i) = \{q_{\text{accept}}\}$ for any i.
If $\delta(p, a) = (q_{\text{accept}}, b, w)$, where $w = \text{R}$ or S, we set $\delta'(q_{pa}, \epsilon) = \{q_{\text{accept}}\}$. If $\delta(p, a) = (q_{\text{reject}}, b, w)$, where $w = \text{R}$ or S, we set $\delta'(q_{pa}, \epsilon) = \{q_{\text{reject}}\}$.

3.14

First, show that we can simulate a queue automaton with a Turing Machine.

Assume that the input string is on TM tape. Move to the end of the string and insert a new character, say #, to denote the beginning of the queue. Then shift this delimiter and each character of the input tape to the right one symbol. At the beginning of the string, insert another delimiter $. The $ symbol will denote which symbol is currently being read in the queue automaton simulation, and the queue itself will be maintained at the end of the string between the $ and the end-of-tape character.

Now, run as a subroutine of the TM the queue automaton. Whenever the queue automaton would pop an element on the queue, run a subroutine instead for the TM that would move to the leftmost edge of the queue portion of the tape, read in that symbol and then shift each element of the queue to the left one space. Likewise, for push, move to the end of the queue portion of the tape, move the end-of-tape character one symbol farther, and then insert the target symbol at this new space.

To show that a queue automaton can simulate a Turing Machine, start with the input on a separate read-only tape as per the format, and an empty queue. To initialize the queue, push a delimiter $ and then each character of the input string in succession. Then push another delimiter # to make the end of the input string. $ will denote the current character being read on the tape encoded circularly in the queue, and the # will denote the end of the string.

The queue automaton, having already read in all of the input strings, will then perform epsilon transitions to simulate the TM states. To read in a character, it will be popped

from the queue and the appropriate write will be pushed at the end. However, merely reading the queue successively will simulate a TM that only reads to the right. To move to the left, the queue automaton will cycle (popping and pushing the same symbol) until is reaches the $ symbol that will always be popped before the current symbol is read, and then pushed after the written value is pushed. To maintain the $ symbol then, we must cycle through the entire queue each time we read: for if we read to the left in the TM, we will simulate it by cycling through our entire tape until we reach the $ symbol pointing to the tape symbol just to the left. However, even reading to the right, we must maintain the $ symbol as the delimiter just to the left of the symbol we are currently reading on the tape. By cycling through the queue as such, and altering the order in which the $ symbol is pushed in relation to when the TM symbol to be written is pushed, we can simulate all tape operations.

3.15

a. For any two decidable languages L_1 and L_2, let M_1 and M_2 be the TMs that decide them. We construct a TM M' that decides the union of L_1 and L_2:

"On input w:

1. Run M_1 on w, if it accepts, *accept*.
2. Run M_2 on w, if it accepts, *accept*. Otherwise, *reject*."

M' accepts w if either M_1 or M_2 accepts it. If both reject, M' rejects.

b. For any two decidable languages L_1 and L_2, let M_1 and M_2 be the TMs that decide them. We construct a NTM M' that decides the concatenation of L_1 and L_2:

"On input w:

1. For each way to cut w into two parts $w = w_1w_2$:
2. Run M_1 on w_1.
3. Run M_2 on w_2.
4. If both accept, *accept*. Otherwise, continue with the next w_1, w_2.
5. All cuts have been tried without success, so *reject*."

We try every possible cut of w. If we ever come across a cut such that the first part is accepted by M_1 and the second part is accepted by M_2, w is in the concatenation of L_1 and L_2. So M' accept w. Otherwise, w does not belong to the concatenation of the languages and is rejected.

c. For any decidable language L, let M be the TM that decides it. We construct a NTM M' that decides the star of L:

"On input w:

1. For each way to cut w into parts so that $w = w_1w_2 \dots w_n$:
2. Run M on w_i for $i = 1, 2, \dots, n$. If M accepts each of these string w_i, *accept*.
3. All cuts have been tried without success, so *reject*."

If there is a way to cut w into different substrings such that every substring is accepted by M, w belongs to the star of L and thus M' accepts w. Otherwise, w is rejected. Since there are finitely many possible cuts of w, M' will halt after finitely many steps.

d. For any decidable language L, let M be the TM that decides it. We construct a TM M' that decides the complement of L:

"On input w:

1. Run M on w. If M accepts, *reject*; if M rejects, *accept*."

Since M' does the opposite of whatever M does, it decides the complement of L.

e. For any two decidable languages L_1 and L_2, let M_1 and M_2 be the TMs that decide them. We construct a TM M' that decides the intersection of L_1 and L_2:

"On input w:

1. Run M_1 on w, if it rejects, *reject*.
2. Run M_2 on w, if it accepts, *accept*. Otherwise, *reject*."

M' accepts w if both M_1 and M_2 accept it. If either of them rejects, M' rejects w, too.

3.16

a. For any two Turing-recognizable languages L_1 and L_2, let M_1 and M_2 be the TMs that recognize them. We construct a TM M' that recognizes the union of L_1 and L_2:

"On input w:

1. Run M_1 and M_2 alternatively on w step by step. If either accept, *accept*. If both halt and reject, then *reject*."

If any of M_1 and M_2 accept w, M' will accept w since the accepting TM will come to its accepting state after a finite number of steps. Note that if both M_1 and M_2 reject and either of them does so by looping, then M will loop.

b. For any two Turing-recognizable languages L_1 and L_2, let M_1 and M_2 be the TMs that recognize them. We construct a NTM M' that recognizes the concatenation of L_1 and L_2:

"On input w:

1. Nondeterministically cut w into two parts $w = w_1w_2$.
2. Run M_1 on w_1. If it halts and rejects, *reject*. If it accepts, go to stage 3.
3. Run M_2 on w_2. If it accepts, *accept*. If it halts and rejects, *reject*."

If there is a way to cut w into two substrings such M_1 accepts the first part and M_2 accepts the second part, w belongs to the concatenation of L_1 and L_2 and M' will accept w after a finite number of steps.

c. For any Turing-recognizable language L, let M be the TM that recognizes it. We construct a NTM M' that recognizes the star of L:

"On input w:

1. Nondeterministically cut w into parts so that $w = w_1 w_2 \cdots w_n$.
2. Run M on w_i for all i. If M accepts all of them, *accept*. If it halts and rejects any of them, *reject*."

If there is a way to cut w into substrings such M accepts the all the substrings, w belongs to the star of L and M' will accept w after a finite number of steps.

d. For any two Turing-recognizable languages L_1 and L_2, let M_1 and M_2 be the TMs that recognize them. We construct a TM M' that recognizes the intersection of L_1 and L_2:

"On input w:

1. Run M_1 on w. If it halts and rejects, *reject*. If it accepts, go to stage 3.
2. Run M_2 on w. If it halts and rejects, *reject*. If it accepts, *accept*."

If both of M_1 and M_2 accept w, w belongs to the intersection of L_1 and L_2 and M' will accept w after a finite number of steps.

3．17

When I first read this problem I thought that it would require some kind of esoteric existence proof...but it turns out that there's actually a simple construction for the desired language C. Now we know by definition that there is a Turing machine that recognizes the language B (of Turing machine descriptions). Equivalently, we can use this to define an enumerator for B. We can define an enumerator for the language C using the following construction:

1. Let <M_1>, <M_2>, ... be an enumeration of B.

2. Save the value '0' as the size of the last TM description output.

3. For each <M_i> in B, if the length of <M_i> is larger than the last TM description output, then output <M_i> as is. Otherwise, add useless states to M_i until it's description is larger than the last TM output; and then output the result. Save the size of what was output.

There will be a one to one correspondence between the TMs enumerated for B and C, such that the corresponding TMs will always be equivalent. The enumerator for C also has the nice property that the TMs are output in increasing order of size. But this means exactly that the language C is decidable.

3.18

If A is decidable, the enumerator operates by generating the strings in lexicographic order and testing each in turn for membership in A using the decider. Those strings which are found to be in A are printed.

If A is enumerable in lexicographic order, we consider two cases. If A is finite, it is decidable because all finite languages are decidable. If A is infinite, a decider for A operates as follows. On receiving input w, the decider enumerates all strings in A in order until some string lexicographically after w appears. That must occur eventually because A is infinite. If w has appeared in the enumeration already then *accept*, but if it hasn't appeared yet, it never will, so *reject*.

Note: Breaking into two cases is necessary to handle the possibility that the enumerator may loop without producing additional output when it is enumerating a finite language. As a result, we end up showing that the language is decidable but we do not (and cannot) algorithmically construct the decider for the language from the enumerator for the language. This subtle point is the reason for the star on the problem.

3-19

(a) Prove that every infinite Turing-recognizable language has an infinite Turing-decidable subset.

Solution: Enumerators come to our rescue on this problem. Let A be an infinite recognizable language. It must have some enumerator E. We define the following enumerator E':

E'.

1. Set $n = -1$.
2. Start simulating E. Each time it prints a string y:
   - If $|y| > n$, then print y and set $n = |y|$.
   - Otherwise, do nothing about y.

The variable n stores the length of the most recently printed string by E'. In this way, E' will only print a new string if it is longer than everything it has previously printed, so its output is in lexicographic order. E' will clearly only print strings that are in A, so $L(E') \subseteq A$. Finally, no matter what value n currently has, E will eventually output a string longer than n characters (since A is infinite). So E' will print an infinite number of strings.

We have shown an infinite language $L(E') \subseteq A$. Since this language has a lexicographic enumerator, it is decidable. ■

3.20

We give a DFA A that is equivalent to TM M. Design A in two stages. First, we arrange the states and transitions of A to simulate M on the read-only input, portion of the tape. Second, we assign the accept states of A to correspond to the action of M on the read/write portion of the tape to the right of the input. In the first stage, we construct A to store a function

$$F_s : (Q \cup \{\text{first}\}) \longrightarrow (Q \cup \{\texttt{acc}, \texttt{rej}\})$$

in its finite control where Q is the set of M's states and s is the string that it has read so far. Note that only finitely many such functions exist, so we assign one state for each possibility.

The function F_s contains information about the way M would compute on a string s. In particular, $F_s(\text{first})$ is the state that M enters when it is about to move off of the right end of s for the first time, when M is started in its starting state at the left end of s. If M accepts or rejects (either explicitly or by looping) before ever moving off of the right end of s, then $F_s(\text{first}) = \texttt{acc}$ or $\texttt{rej}$ accordingly. Furthermore, for $F_s(q)$ is the state M enters when it is about to move off of the right end of s for the first time, when M is started in state q at the right end of s. As before, if M accepts or rejects before ever moving off of the right end of s, then $F_s(q) = \texttt{acc}$ or $\texttt{rej}$.

Observe that, for any string s and symbol a in Σ, we can determine F_{sa} from a, F_s, and M's transition function. Furthermore, F_ε is predetermined because $F_\varepsilon(\text{first}) = q_0$, the start state of M, and $F_\varepsilon(q) = q$ (because TMs that attempt to move off the leftmost end of the tape just stay in the left-hand cell). Hence we can obtain A's start state and transition function. That completes the first stage of A's design.

To complete the second stage, we assign the accept states of M. For any two strings s and t, if $F_s = F_t$, then M must have the same output on s and t—either both s and t are in $L(M)$ or both are out. Hence A's state at the end of s is enough to determine whether M accepts s. More precisely, we declare a state of A to be accepting if it was assigned to a function F that equals F_s for any string s that M accepts.

3.21

1) 由 $c_{max} \geq |c_1|$知，当$|x| \leq 1$，则欲判定不等式明显成立。
2) 当$|x| > 1$ 时，由

$c_1x^n + c_2x^{n-1} + \ldots + c_nx + c_{n+1} = 0$

$\Rightarrow$ $c_1x = -(c_2 + \ldots + c_nx^{2-n} + c_{n+1}x^{1-n})$

$\Rightarrow$ $|c_1|\,|x| = |c_2 + \ldots + c_nx^{2-n} + c_{n+1}x^{1-n}|$

$< |c_2| + \ldots + |c_n||x|^{2-n} + |c_{n+1}|\,|x|^{1-n}$

$\leq |c_2| + \ldots\ldots |c_n| + |c_{n+1}||x_0|$ （$|x| >= 1$,当 $a <= 0$ 时，$|x|^a < 1$）

$\leq n\, c_{max}$

$< (n + 1)\, c_{max}$

$\Rightarrow$ $|x| < (n + 1)\, c_{max} / |c_1|$.

由(1)和(2)可知结论成立。

3.22

The language A is one of the two languages, $\{0\}$ or $\{1\}$. In either case the language is finite, and hence decidable. If we are not able to determine which of these two languages is A, we won't be able to describe the decider for A, but we can give two TMs, one of which is A's decider.