Computer Science 351

More About Turing Machines

Instructor: Wayne Eberly

Department of Computer Science
University of Calgary

Lecture #12

Goals for Today

Goals for Today:

- Introduce ***Turing machines that compute functions.***
- Introduce a suggested ***design process*** for Turing machines.

Turing Machines That Compute Functions

Let Σ_1 and Σ_2 be alphabets such that $\sqcup \notin \Sigma_1$ and $\sqcup \notin \Sigma_2$.

Definition: A ***Turing machine that computes a (partial or total) function*** $f : \Sigma_1^\star \to \Sigma_2^\star$ is modified version of a Turing machine, M, that satisfies the following properties.

1. M has a finite ***tape alphabet*** such that $\Sigma_1 \subseteq \Gamma$, $\Sigma_2 \subseteq \Gamma$, and $\sqcup \in \Gamma$.
2. Instead of an accept state q_{accept} and a reject state q_{reject}, M has a single ***halt*** state q_{halt}. (All other states are "non-halting" states.)
3. For every string $\omega \in \Sigma_1^\star$ such that $f(\omega)$ is defined, if M is executed on input ω then this execution halts, and $f(\omega)$ is written at the leftmost cells of the tape (with infinitely many copies of $\sqcup$ to the right) — and the tape head is at the leftmost cell of the tape when the computation ends.

Turing Machines That Compute Functions

4. For every string $\omega \in \Sigma_1^\star$ such that $f(\omega)$ is *not* defined, the execution of M on input ω never halts.

Definition: A partial or total function $f : \Sigma_1^\star \to \Sigma_2^\star$ is ***Turing-computable*** (or just ***computable***) if there is a Turing machine that computes f.

Turing Machine Design

- "Turing machine design" is not a significant part of this course, by itself — but it is necessary to design Turing machines for several (generally, reasonably simple) computational problems in order to prove claims or solve more interesting problems.
- A recommended ***design process*** for Turing machines is now given.
- ***Ongoing Example:*** This will be used to design a Turing machine that decides the language

$$L = \{\omega \in \Sigma^\star \mid \omega \text{ has the same number of } \mathtt{a}\text{'s as } \mathtt{b}\text{'s}\}$$

for $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

Turing Machine Design

Summary of Design Process:

1. Solve the problem at a ***high level*** first — proving correctness of the solution (if this is not obvious).
2. Gradually add implementation details — checking that no errors have been introduced (so that the more detailed solution is still correct) until the solution is detailed enough to be implemented using a Turing machine.

This is based on an approach suggested in Sipser's text, *Introduction to the Theory of Computation*.

Design: Start with High-Level Description

One should often begin by choosing an ***algorithm*** that will — somehow — solve the given problem.

- This version of the solution — which can be given as pseudocode or simple English — is sometimes called a ***high-level description*** of a Turing machine.
- One can establish ***correctness***, at this level, by describing the effects of the operations that have been carried out. This can sometimes resemble the process of "proving correctness of an algorithm" described in a course like CPSC 331 or CPSC 413.

A high-level description of an algorithm — or Turing machine — that decides membership in the given language, L, is as follows.

Application to the Example Problem

Consider the following ***algorithm***:

On input $\omega \in \Sigma^\star$:

1. Consider all symbols in ω to be ***unmarked***.
2. `while` (there is at least one unmarked symbol) {
3. `if` (the leftmost unmarked symbol is "`a`") {
4. Mark this "leftmost unmarked symbol".
5. `if` (there is also an unmarked copy of "`b`") {
6. Mark the leftmost copy of "`b`"
   } `else` {
7. ***reject*** ω
   }

Application to the Example Problem

```
    } else {
8.    Mark the leftmost symbol (which must be "b")
9.    if (there is also an unmarked copy of "a") {
10.     Mark the leftmost copy of "a"
      } else {
11.     reject ω
      }
    }
  }
12. accept ω
```

Application to the Example Problem

Claim: If this algorithm is executed on an input string $\omega \in \Sigma^\star$ then then execution of the algorithm halts. Furthermore, ω is ***accepted*** if $\omega \in L$ and ω is ***rejected*** if $\omega \notin L$.

Proof: Consider an input string $\omega \in \Sigma^\star$ Let n be the number of `a`'s in ω and let m be the number of `b`'s in ω.

- It is easily proved by induction on h that, for every number h such that $0 \leq h \leq \texttt{min}(n, m)$, there are at least h executions of the body of the `while` loop (at lines 2–11). Furthermore, after the h^{th} execution of the body of the loop[1], the string obtained from ω includes
  - exactly h marked copies of `a`,
  - exactly h marked copies of `b`,
  - exactly $n - h$ unmarked copies of `a`, and
  - exactly $m - h$ unmarked copies of `b`

[1] — or before the *first* execution of the body of the loop, if $h = 0$ —

Application to the Example Problem

Either $\omega \in L$, so that $n = m$, or $\omega \notin L$ — so that either $n < m$ or $n > m$.

- If $\omega \in L$, so that $n = m$, then — at the beginning of the n executions of the body of the loop, the test at line 2 is reached and failed — so that the step at line 12 is reached and ω is accepted, as desired.
- If $\omega \notin L$, and $n < m$, then, after n executions of the body of the loop, there are no unmarked copies of "a" left — but there is at least one unmarked copy of "b". Thus the test at line 2 is passed, the test at line 3 is checked and failed. After an execution of the step at line 8 the test at line 9 is checked and failed — so that the step at line 11 is executed and ω is rejected, as desired.

Application to the Example Problem

- In the only other case $\omega \notin L$ and $n > m$. In this case, after m executions of the body of the loop, there are no unmarked copies of "b" left — but there is at least one unmarked copy of "a". Thus the test at line 2 is passed, and the test at line 3 is checked and passed as well. After an execution of the step at line 4 the test at line 5 is checked and failed — so that the step at line 7 is executed and ω is rejected, as desired.

Thus the execution halts in every case — with ω accepted if $\omega \in L$ and ω rejected if $\omega \notin L$ — as needed to establish the claim. □

Design: Continue to Implementation Level

An ***implementation description*** includes additional details about

- the way the Turing machine uses its finite control to remember information,
- the way the Turing machine moves its tape head, and
- the way it uses its tape

(in order to implement the algorithm) are described. This information is generally given in simple written English (or simple pseudocode).

There will often be *many* choices that can be made about how to implement the high-level algorithm that has been described.

Application to the Example Problem

- Recall that the input string is initially stored at the leftmost cells of the tape (with infinitely many copies of "␣" to the right) and the tape head is located at the leftmost cell of the tape.
- ***First Idea:*** Include a new symbol, "x" to the tape alphabet, Γ — and that a symbols is "marked" by replacing it with a copy of "x" on the tape.
- Suppose the tape head is *always* at *the leftmost cell* of tape when an execution of the test at line 2 begins — and the Turing machine begins to carry out this test by sweeping to the *right* on the tape.

Application to the Example Problem

- The Turing machine should sweep right over copies of "x" without changing them — until either "a", "b", or "␣" is seen.
  (a) *Case:* A copy of "a" is seen first (after zero or more copies of "x": In this case the test at line 2 has passed — and the test at line 3 has, effectively, been carried out and passed too, so that the execution should continue with an implementation of the step at line 4.
  (b) *Case:* A copy of "b" is seen first (after zero or more copies of "x": In this case the test at line 2 has passed — and the test at line 3 has, effectively, been carried out and failed, so that the execution should continue with an implementation of the step at line 8.
  (c) *Case:* A copy of "␣" is seen first (after zero or more copies of "x": In this case the test at line 2 has failed failed — and the execution should continue with the step at line 12. Since ω should be ***accepted*** at this point the Turing machine can simply move to its accept state.

Application to the Example Problem

- In cases (a) and (b), the next step (at line 4 or 8) can be carried out by writing "x" on the tape and moving right — completing the move where the leftmost unmarked symbol was discovered.
- To carry out the next test (at line 5 or 9) the tape head should continue to sweep *right* until either "blank" or an unmarked letter, that is *not* the same as the one discovered to end the step at line 2, is found.
  - If "blank" was seen first, then the test (at line 5 or 9) failed and either the step at line 7 or the step at line 11 should be carried out — and ω should be ***rejected*** — so it suffices for the Turing machine to move to its reject state, to finish.
  - Otherwise the test (at line 5 or 9) passed and either the step at line 6 or the step at line 10 should be carried out. The Turing machine can begin to do this by writing "x" over the unmarked symbol that has just been discovered.

Application to the Example Problem

- ***Problem:*** The tape head should now be moved back to the leftmost cell of the tape — so that an implementation of the test at line 2 can be started, once again.
- However, there is no way to *detect* the leftmost cell of the tape using only a finite amount of information (remembered using the Turing machine's finite control) and the symbol visible on the tape.

Application to the Example Problem

- ***Second Idea:*** Introduce *two* new tape symbols — which we can call "X" and "x".
- If $\omega = \lambda$ then ω will accepted right away. Otherwise "X" will be written on top of the non-blank symbol on the leftmost cell (then a symbol is marked for the *first* time) — so that the leftmost cell of the tape will always store "X" after that.
- No other copy of "X" will be written again — because "x" will be written on top of unmarked symbols (when they are being marked) after that.

Application to the Example Problem

- The algorithm must be revised, because the *first* execution of the body of the loop is now different from all of the later executions of this loop body (and will require its own code, before the main loop).
- However, it is now easy to *complete* the sweep left, to the leftmost cell, at the end of an execution of the loop body: This should stop as soon as "X" is visible on the tape.
- A supplemental document includes complete pseudocode for the "implementation-level" algorithm that can now be given — and proved to be correct (using the correctness of the "high-level" algorithm that it is based on).

Formal Description

The ***formal definition*** of a Turing machine includes a specification of all of the components of a Turing machine, including its finite control, tape alphabet and transition function.

- If the implementation-level description was sufficiently detailed then it should be *too* difficult to give this (although it still might take time).

Application to the Example Problem

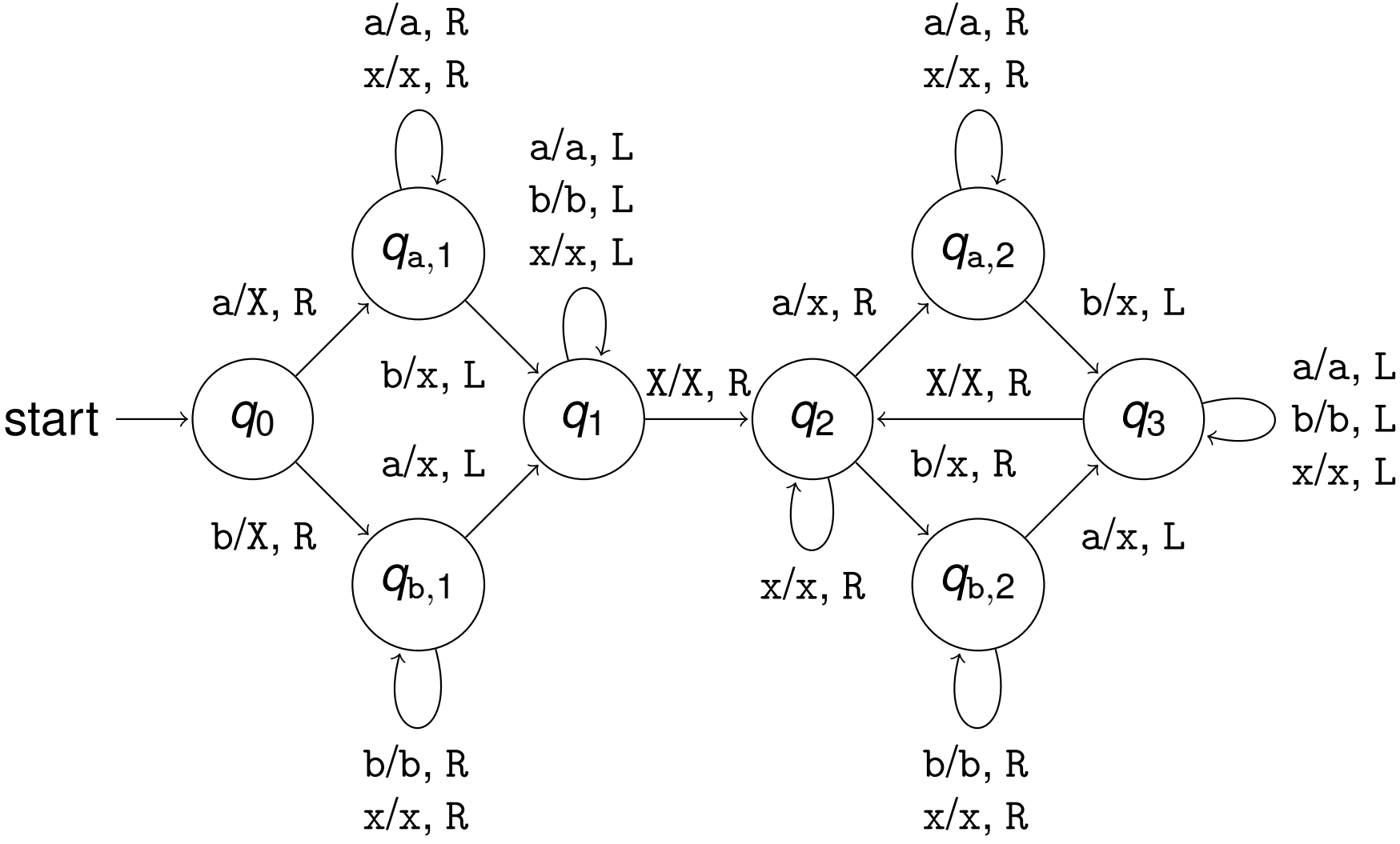
The material given, so far, can be used to produce a Turing machine

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$$

where

- $Q = \{q_0, q_{\text{a},1}, q_{\text{b},1}, q_1, q_{\text{a},2}, q_{\text{b},2}, q_2, q_3, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{\mathtt{a}, \mathtt{b}\}$,
- $\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{x}, \mathtt{X}, \sqcup\}$, and
- an *incomplete* transition diagram is as shown on the following slide: Missing transitions for "$\sqcup$" out of q_0 and q_2 go to q_{accept} (moving right), while missing transitions for "$\sqcup$" out of $q_{\text{a},1}$, $q_{\text{b},1}$, $q_{\text{a},2}$ and $q_{\text{b},2}$ go to q_{reject} (moving right).

Application to the Example Problem



Application to the Example Problem

- The supplemental document, mentioned above, also includes additional details about how this formal description was obtained from the implementation description — and also why the correctness of the implementation description implies that the Turing machine M, with this formal description, decides the above language L.