Notes on Models of Computation
Chapter 9

H. Conrad Cunningham

06 April 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

Note: These notes were written primarily to accompany my use of Chapter 1 of the Linz textbook *An Introduction to Formal Languages and Automata* [[1].

10 Turing Machines

A finite accepter (nfa, dfa)

- has no local storage
- accepts a regular language

A pushdown accepter (npda, dpda)

- has a stack for local storage
- accepts a language from a larger family
  - an npda accepts a context-free language
  - a dpda accepts a deterministic context-free language

The family of regular languages is a subset of the deterministic context-free languages, which is a subset of the context-free languages.

But, as we saw in Chapter 8, not all languages of interest are context-free. To accept languages like $\{a^n b^n c^n : n \geq 0\}$ and $\{ww : w \in \{a, b\}^*\}$, we need an automaton with a more flexible internal storage mechanism.

What kind of internal storage is needed to allow the machine to accept languages such as these? multiple stacks? a queue? some other mechanism?

More ambitiously, what is the most powerful automaton we can define? What are the limits of mechanical computation?

This chapter introduces the *Turing machine* to explore these theoretical questions. The Turing machine is a fundamental concept in the theoretical study of computation.

The Turing machine

- has a *tape*, a one-dimensional array of readable and writable cells that is unbounded in both directions
- accepts a language from the family of *recursively enumerable languages*, a larger family of languages than context-free

Although Turing machines are simple mechanisms, the *Turing thesis* (also known as the Church-Turing thesis) maintains that *any computation that can be carried out on present-day computers an be done on a Turing machine.*

Note: Much of the work on computability was published in the 1930's, before the advent of electronic computers a decade later. It included work by Austrian (and later American) logician Kurt Goedel on primitive recursive function theory, American mathematician Alonso Church on lambda calculus (a foundation of functional programming), British mathematician Alan Turing (also later a PhD student of Church's) on Turing machines, and American mathematician Emil Post on Post machines.

10.1 The Standard Turing Machine

10.1.1 What is a Turing Machine?

10.1.1.1 Schematic Drawing of Turing Machine Linz Figure 9.1 shows a schematic drawing of a standard Turing machine.

This deviates from the general scheme given in Chapter 1 in that the input file, internal storage, and output mechanism are all represented by a single mechanism, the tape. The input is on the tape at initiation and the output is on that tape at termination.

On each move, the tape's *read-write head* reads a symbol from the current tape cell, writes a symbol back to that cell, and moves one cell to the left or right.

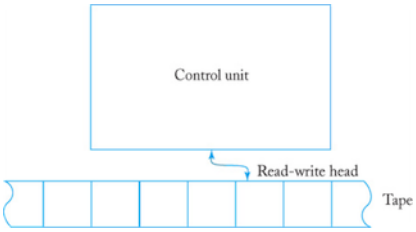


Figure 1: **Linz Fig. 9.1: Standard Turing Machine**

10.1.1.2 Definition of Turing Machine Turing machines were first defined by British mathematician Alan Turing in 1937, while he was a graduate student at Cambridge University.

Linz Definition 9.1 (Turing Machine): A *Turing machine* M is defined by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$$

where

1. Q is the set of internal states
2. Σ is the input alphabet
3. Γ is a finite set of symbols called the *tape alphabet*
4. δ is the transition function
5. $\Box \in \Gamma$ is a special symbol called the *blank*
6. $q_0 \in Q$ is the initial state
7. $F \subseteq Q$ is the set of final states

We also require

8. $\Sigma \subseteq \Gamma - \{\Box\}$

and define

9. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$.

Requirement 8 means that the blank symbol $\square$ cannot be either an input or an output of a Turing machine. It is the default content for any cell that has no meaningful content.

From requirement 9, we see that the arguments of the transition function δ are:

- the current state of the control unit
- the current tape symbol

The result of the transition function δ gives:

- the new state of the control unit
- the symbol that replaces the current symbol on the tape
- a move symbol L or R, denoting a move of the read-write head to the *left* or the *right* on the tape

In general, δ is a partial function. That is, not all configurations have a next move defined.

10.1.1.3 Linz Example 9.1 Consider a Turing machine with a move defined as follows:

$$\delta(q_0, a) = (q_1, d, R)$$

Linz Figure 9.2 shows the situation (a) before the move and (b) after the move.

Figure 2: **Linz Fig. 9.2: One Move of a Turing Machine**

10.1.1.4 A Simple Computer A Turing machine is a simple computer. It has

- a processing unit that has a finite memory
- a tape that provides unlimited secondary storage capacity
- a limited set of instructions

The Turing machine can

- sense the symbol under the tape's read-write head
- use the result to decide what to do next
- write a symbol back to the tape
- change the state of the control
- move the read-write head one position to the left or right on the tape

The transition function δ determines the behavior of the machine, i.e., it is the machine's *program*.

The Turing macine starts in initial state q_0 and then goes through a sequence of moves defined by δ. A cell on the tape may be read and written many times.

Eventually the Turing machine may enter a configuration for which δ is undefined. When it enters such a state, the machine *halts*. Hence, this state is called a *halt state*.

Typically, no transitions are defined on any final state.

10.1.1.5 Linz Example 9.2 Consider the Turing machine defined by

$$Q = \{q_0, q_1\},$$
$$\Sigma = \{a, b\},$$
$$\Gamma = \{a, b, \Box\},$$
$$F = \{q_1\}$$

where δ is defined as follows:

1. $\delta(q_0, a) = (q_0, b, R)$,
2. $\delta(q_0, b) = (q_0, b, R)$,
3. $\delta(q_0, \Box) = (q_1, \Box, L)$.

Linz Figure 9 .3 shows a sequence of moves for this Turing machine:

- It begins in state q_0 with the input positioned over an a.
- When an a is read, transition rule 1 fires, replaces a by b on the tape, moves right, and stays in state q_0.
- When a b is read, transition rule 2 fires, leaves b on the tape, moves right, and stays in state q_0.
- It continues moving right, replacing each a by a b and leaving each b unchanged.
- When a blank ($\Box$) is read, transition rule 3 fires, leaves the blank on the tape, moves left, and enters final state q_1.

Figure 3: **Linz Fig. 9.3: A Sequence of Moves of a Turing Machine**

10.1.1.6 Transition Graph for Turing Machine As with finite and pushdown automata, we can use transition graphs to represent Turing machines. We label the edges of the graph with a triple giving (1) the current tape symbol, (2) the symbol that replaces it, and (3) the direction in which the read-write head moves.

Linz Figure 9.4 shows a transition graph for the Turing machine given in Linz Example 9.2.

Figure 4: **Linz Fig. 9.4: Transition Graph for Example 9.2**

10.1.1.7 Linz Example 9.3 (Infinite Loop) Consider the Turing machine defined in Linz Figure 9.5.

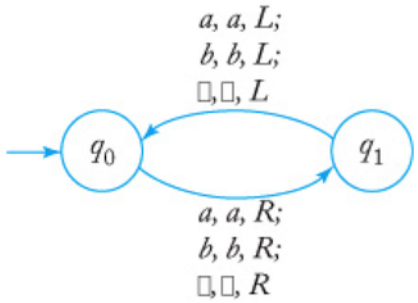


Figure 5: **Linz Fig. 9.5: Infinite Loop**

Suppose the tape initially contains $ab\ldots$ with the read-write head positioned over the a and in state q_0. Then the Turing machine executes the following sequence of moves:

1. The machine reads symbol a, leaves it unchanged, moves right (now over symbol b), and enters state q_1.
2. The machine reads b, leaves it unchanged, moves back left (now over a again), and enters state q_0 again.
3. The machine then repeats steps 1-3.

Clearly, regardless of the tape configuration, this machine does not halt. It goes into an *infinite loop*.

10.1.1.8 Standard Turing Machine Because we can define a Turing machine in several different ways, it is useful to summarize the main features of our model.

A *standard Turing machine*:

1. has a tape that is unbounded in both directions, allowing any number of left and right moves
2. is deterministic in that δ defines at most one move for each configuration
3. has no special input or output files. At the initial time, the tape has some specified content, some of which is considered input. Whenever the machine halts, some or all of the contents of the tape is considered output.

These definitions are chosen for convenience in this chapter. Chapter 10 (which we do not cover in this course) examines alternative versions of the Turing machine concept.

10.1.1.9 Instantaneous Description of Turing Machine As with pushdown automata, we use *instantaneous descriptions* to examine the configurations in a sequence of moves. The notation (using strings)

$$x_1 q x_2$$

or (using individual symbols)

$$a_1 a_2 \cdots a_{k-1} q a_k a_{k+1} \cdots a_n$$

gives the instantaneous description of a Turing machine in state q with the tape as shown in Linz Figure 9.5.

By convention, the read-write head is positioned over the symbol to the right of the state (i.e., a_k above).

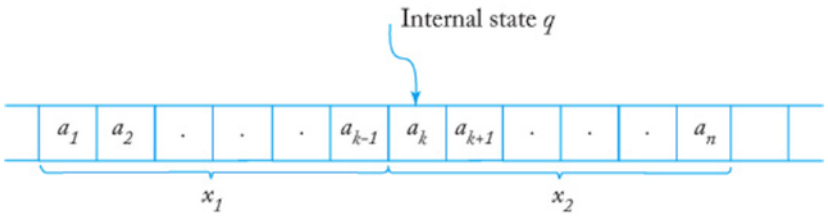


Figure 6: **Linz Fig. 9.6: Tape Configuration** $a_1 a_2 \cdots a_{k-1} q a_k a_{k+1} \cdots a_n$

A tape cell contains $\Box$ if not otherwise defined to have a value.

Example: The diagrams in Linz Figure 9.3 (above) show the instantaneous descriptions $q_0 aa$, $bq_0 a$, $bbq_0 \Box$, and $bq_1 b$.

As with pushdown automata, we use $\vdash$ to denote a *move*.

Thus, for transition rule

$$\delta(q_1, c) = (q_2, e, R)$$

we can have the move

$$abq_1 cd \vdash abeq_2 d.$$

As usual we denote the *transitive closure of move* (i.e., arbitrary number of moves) using:

$$\vdash^*$$

We also use subscripts to distinguish among machines:

$$\vdash_M$$

10.1.1.10 Computation of Turing Machine Now let's summarize the above discussion with the following definitions.

Linz Definition 9.2 (Computation): Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ be a Turing machine. Then any string $a_1 \cdots a_{k-1} q_1 a_k a_{k+1} \cdots a_n$ with $a_i \in \Gamma$ and $q_1 \in Q$, is an *instantaneous description* of M.

A *move*

$$a_1 \cdots a_{k-1} q_1 a_k a_{k+1} \cdots a_n \vdash a_1 \cdots a_{k-1} b q_2 a_{k+1} \cdots a_n$$

is possible if and only if

$$\delta(q_1, a_k) = (q_2, b, R).$$

A *move*

$$a_1 \cdots a_{k-1} q_1 a_k a_{k+1} \cdots a_n \vdash a_1 \cdots q_2 a_{k-1} b a_{k+1} \cdots a_n$$

is possible if and only if

$$\delta(q_1, a_k) = (q_2, b, L).$$

M halts starting from some initial configuration $x_1 q_i x_2$ if

$$x_1 q_i x_2 \ \vdash^* \ y_1 q_j a y_2$$

for any q_j and a, for which $\delta(q_j, a)$ is undefined.

The sequence of configurations leading to a halt state is a *computation*.

If a Turing machine does not halt, we use the following *special notation* to describe its computation:

$$x_1 q x_2 \vdash^* \infty$$

10.1.2 Turing Machines as Language Acceptors

Can a Turing machine accept a string w?

Yes, using the following setup:

- Write w on the tape initially.
- Fill all the unused cells on the tape with blanks $\Box$.
- Start the Turing machine with read-write head over leftmost symbol of w.
- If the machine halts in a final state, then it *accepts* string w.

Linz Definition 9.3 (Language Accepted by Turing Machine): Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ be a Turing machine. Then the *language accepted by* M is

$$L(M) = \{w \in \Sigma^+ : q_0 w \vdash^* x_1 q_f x_2, q_f \in F, x_1, x_2 \in \Gamma^*\}.$$

Note: The finite string w must be written to the tape with blanks on both sides. No blanks can are embedded within the input string w itself.

Question: What if $w \notin L(M)$?

The Turing machine might:

1. halt in nonfinal state
2. never halt

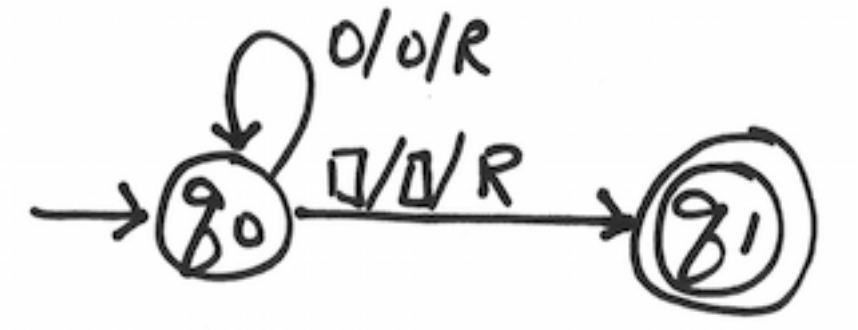
Any string for which the machine does not halt is, by definition, not in $L(M)$.

10.1.2.1 Linz Example 9.6 For $\Sigma = \{0, 1\}$, design a Turing machine that accepts the language denoted by the regular expression 00^*.

We use two internal states $Q = \{q_0, q_1\}$, one final state $F = \{q_1\}$, and transition function:

$\delta(q_0, 0) = (q_0, 0, R)$,
$\delta(q_0, \Box) = (q_1, \Box, R)$

The transition graph shown below implements this machine.



- While a 0 appears under the read-write head, the head moves to the right.
- If a blank is read, the machine halts in final state q_1.
- If a 1 is read, the machine halts in the nonfinal state q_0 because $\delta(q_0, 1)$ is undefined.

The Turing machine also halts in a final state if started in state q_0 on a blank. We could interpret this as acceptance of λ, but for technical reasons the empty string is not included in Linz Definition 9.3.

10.1.2.2 Linz Example 9.7 For $\Sigma = \{a, b\}$, design a Turing machine that accepts

$L = \{a^n b^n : n \geq 1\}$.

We can design a machine that incorporates the following algorithm:

While both a's and b's remain
  replace leftmost a by x
  replace leftmost b by y
If no a's or b's remain
  accept
else
  reject

Filling in the details, we get the following Turing machine for which:

$Q = \{q_0, q_1, q_2, q_3, q_4\}$
$F = \{q_4\}$
$\Sigma = \{a, b\}$
$\Gamma = \{a, b, x, y, \Box\}$

The transitions can be broken into several sets.

The first set

1. $\delta(q_0, a) = (q_1, x, R)$
2. $\delta(q_1, a) = (q_1, a, R)$
3. $\delta(q_1, y) = (q_1, y, R)$
4. $\delta(q_1, b) = (q_2, y, L)$

replaces the leftmost a with an x, then causes the read-write head to travel right to the first b, replacing it with a y. The machine then enters a state q_2, indicating that an a has been successfully paired with a b.

The second set

5. $\delta(q_2, y) = (q_2, y, L)$
6. $\delta(q_2, a) = (q_2, a, L)$
7. $\delta(q_2, x) = (q_0, x, R)$

reverses the direction of movement until an x is encountered, repositions the read-write head over the leftmost a, and returns control to the initial state.

The machine is now back in the initial state q_0, ready to process the next a-b pair.

After one pass through this part of the computation, the machine has executed the partial computation:

$$q_0 aa \cdots abb \cdots b \vdash^* xq_0 a \cdots ayb \cdots b$$

So, it has matched a single a with a single b.

The machine continues this process until no a is found on leftward movement.

If all a's have been replaced, then state q_0 detects a y instead of an a and changes to state q_3. This state must verify that all b's have been processed as well.

8. $\delta(q_0, y) = (q_3, y, R)$
9. $\delta(q_3, y) = (q_3, y, R)$
10. $\delta(q_3, \Box) = (q_4, \Box, R)$

The input $aabb$ makes the moves shown below. (The bold number in parenthesis gives the rule applied in that step.)

		q_0aabb	– start at left end
(1)	$\vdash$	xq_1abb	– process 1st a-b pair
(2)	$\vdash$	xaq_1bb	– moving to right
(4)	$\vdash$	xq_1ayb	
(6)	$\vdash$	q_2xayb	– move back to left
(7)	$\vdash$	xq_0ayb	
(1)	$\vdash$	xxq_1yb	– process 2nd a-b pair
(3)	$\vdash$	$xxyq_1b$	– moving to right
(4)	$\vdash$	xxq_2yy	
(5)	$\vdash$	xq_2xyy	– move back to left
(7)	$\vdash$	xxq_0yy	
(8)	$\vdash$	$xxyq_3y$	– no a's
(9)	$\vdash$	$xxyyq_3\Box$	– check for extra b's
(10)	$\vdash$	$xxyy\Box q_4\Box$	– done, move to final

The Turing machine halts in final state q_4, thus accepting the string $aabb$.

If the input is not in the language, the Turing machine will halt in a nonfinal state.

For example, consider:

- a^nb^m for $n > m$?
  - halts in nonfinal state q_1 when $\Box$ found
- a^nb^m for $0 < n < m$?
  - halts in nonfinal state q_3 when b found
- aba?
  - halts in nonfinal state q_3 when a found
- b?
  - halts in nonfinal state q_0 when b found

10.1.3 Turing Machines as Transducers

Turing machines are more than just language accepters. They provide a simple abstract model for computers in general. Computers transform data. Hence, Turing machines are *transducers* (as we defined them in Chapter 1). For a computation, the

- *input* consists of all the nonblank symbols on the tape initially
- *output* consists of is whatever is on the tape when the machine halts in a final state

Thus, we can view a Turing machine transducer M as an implementation of a function f defined by

$$\hat{w} = f(w)$$

provided that

$$q_0 w \vdash_M^* q_f \hat{w},$$

for some final state q_f.

Linz Definition 9.4 (Turing Computable): A function f with domain D is said to be *Turing-computable*, or just *computable*, if there exists some Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ such that

$$q_0 w \vdash_M^* q_f f(w),\ q_f \in F,$$

for all $w \in D$.

Note: A transducer Turing machine must start on the leftmost symbol of the input and stop on the leftmost symbol of the output.

10.1.3.1 Linz Example 9.9

Compute $x + y$ for positive integers x and y.

We use *unary notation* to represent the positive integers, i.e., a positive integer is represented by a sequence of 1's whose length is equal to the value of the integer. For example:

$$1111 \ = \ 4$$

The computation is

$$q_0 w(x)0w(y) \vdash^* q_f w(x+y)0$$

where 0 separates the two numbers at initiation and after the result at termination.

Key idea: Move the separating 0 to the right end.

To achieve this, we construct $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ with

$Q = \{q_0, q_1, q_2, q_3, q_4\}$
$F = \{q_4\}$

$\delta(q_0, 1) = (q_0, 1, R)$
$\delta(q_0, 0) = (q_1, 1, R)$
$\delta(q_1, 1) = (q_1, 1, R)$
$\delta(q_1, \Box) = (q_2, \Box, L)$
$\delta(q_2, 1) = (q_3, 0, L)$
$\delta(q_3, 1) = (q_3, 1, L)$
$\delta(q_3, \Box) = (q_4, \Box, R)$

The sequence of instantaneous descriptions for adding 111 to 11 is shown below.

$q_0111011$	$\vdash 1q_011011 \vdash 11q_01011 \vdash 111q_0011$
	$\vdash 1111q_1111 \vdash 11111q_11 \vdash 111111q_1\Box$
	$\vdash 11111q_21 \vdash 1111q_310 \vdash 111q_3110$
	$\vdash 11q_31110 \vdash 1q_311110 \vdash q_3111110$
	$\vdash q_3\Box111110 \vdash q_4111110$

10.1.3.2 Linz Example 9.10 Construct a Turing machine that copies strings of 1's. More precisely, find a machine that performs the computation

$$q_0w \vdash^* q_fww,$$

for any $w \in \{1\}^+$.

To solve the problem, we implement the following procedure:

1. Replace every 1 by an x.
2. Find the rightmost x and replace it with 1.
3. Travel to the right end of the current nonblank region and create a 1 there.
4. Repeat steps 2 and 3 until there are no more x's.

A Turing machine transition function for this procedure is as follows:

$\delta(q_0, 1) = (q_0, x, R)$
$\delta(q_0, \Box) = (q_1\Box, L)$
$\delta(q_1, x) = (q_2, 1, R)$
$\delta(q_2, 1) = (q_2, 1, R$
$\delta(q_2, \Box) = (q_1, 1, L)$
$\delta(q_1, 1) = (q_1, 1, L)$
$\delta(q_1, \Box) = (q_3, \Box, R)$

where q_3 is the only final state.

Linz Figure 9.7 shows a transition graph for this Turing machine.

This is not easy to follow, so let us trace the program with the string 11. The computation performed is as shown below.

q_011	$\vdash xq_01 \vdash xxq_0\Box \vdash xq_1x$
	$\vdash x1q_2\Box \vdash xq_111 \vdash q_1x11$

$\vdash\ 1q_211\ \vdash\ 11q_21\ \vdash\ 111q_2\Box$
$\vdash\ 11q_111\ \vdash\ 1q_1111$
$\vdash\ q_11111\ \vdash\ q_1\Box1111\ \vdash\ q_31111$

10.1.3.3 Linz Example 9.11 Suppose x and y are positive integers represented in unary notation.

Construct a Turing machine that halts in a final state q_y if $x \geq y$ and in a nonfinal state q_n if $x < y$.

That is, the machine must perform the computation:

$q_0w(x)0w(y) \vdash^* q_yw(x)0w(y)$, if $x \geq y$
$q_0w(x)0w(y) \vdash^* q_nw(x)0w(y)$, if $x < y$

We can adapt the approach from Linz Example 9.7. Instead of matching a's and b's, we match each 1 on the left of the dividing 0 with the 1 on the right. At the end of the matching, we will have on the tape either

$xx \cdots 110xx \cdots x\Box$

or

$xx \cdots xx0xx \cdots x11\Box$,

depending on whether $x > y$ or $y > x$.

A transition graph for machine is shown below.

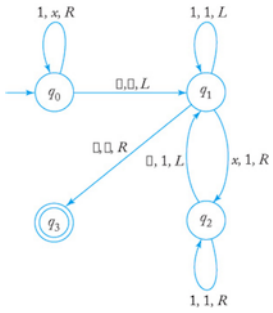
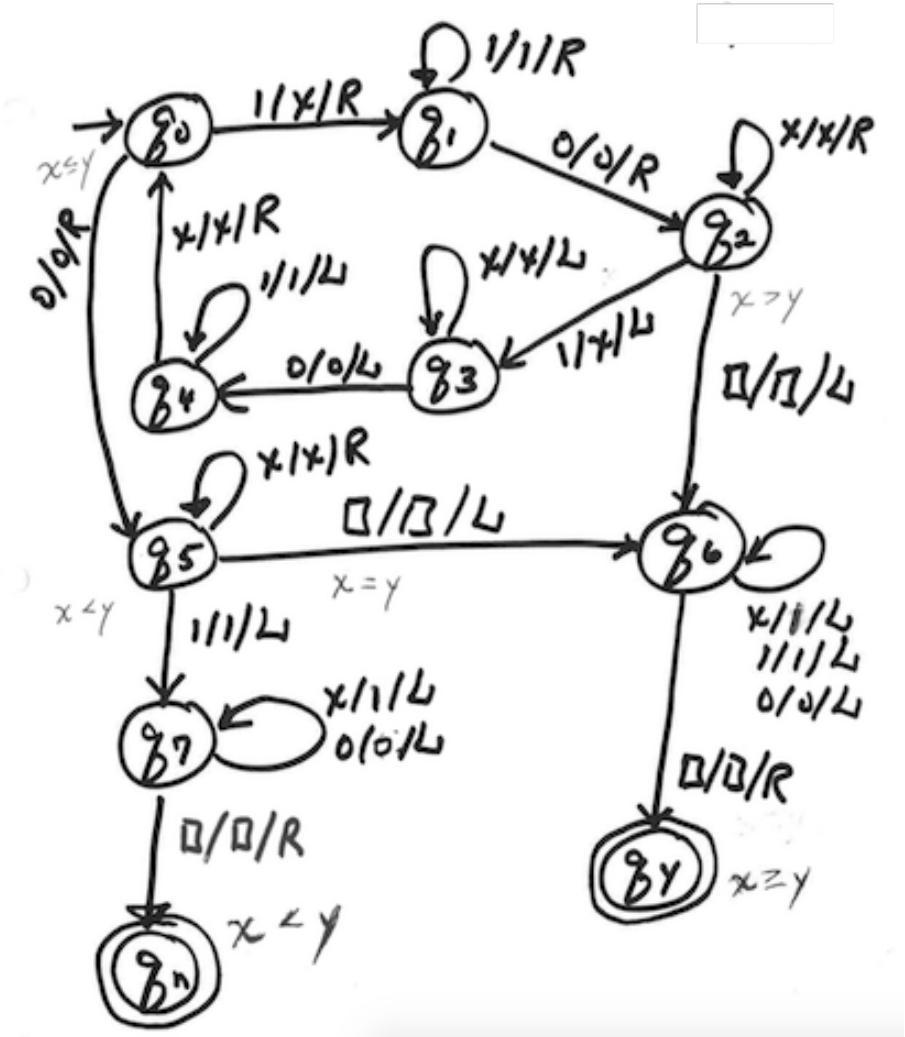


Figure 7: **Linz Fig. 9.7: Transition Graph for Example 9.10**



10.2 Combining Turing Machines for Complicated Tasks

10.2.1 Introduction

How can we compose simpler operations on Turing machines to form more complex operations?

Techniques discussed in this section include use of:

- *Top-down stepwise refinement*, i.e., starting with a high-level description and refining it incrementally until we obtain a description in the actual language
- Block diagrams or pseudocode to state high-level descriptions

10.2.2 Using Block Diagrams

In the *block diagram* technique, we define high-level computations in boxes without internal details on how computation is done. The details are filled in on a subsequent refinement.

To explore the use of block diagrams in the design of complex computations, consider Linz Example 9.12, which builds on Linz Examples 9.9 and 9.11 (above).

10.2.2.1 Linz Example 9.12 Design a Turing machine that computes the following function:

$f(x, y) = x + y$, if $x \geq y$,
$f(x, y) = 0$, if $x < y$.

For simplicity, we assume x and y are positive integers in unary representation and the value zero is represented by 0, with the rest of the tape blank.

Linz Figure 9.8 shows a high-level block diagram of this computation. This computation consists of a network of three simpler machines:

- a Comparer C to determine whether or not $x \geq y$
- an Adder A that computes $x + y$
- an Eraser E that changes every 1 to a blank

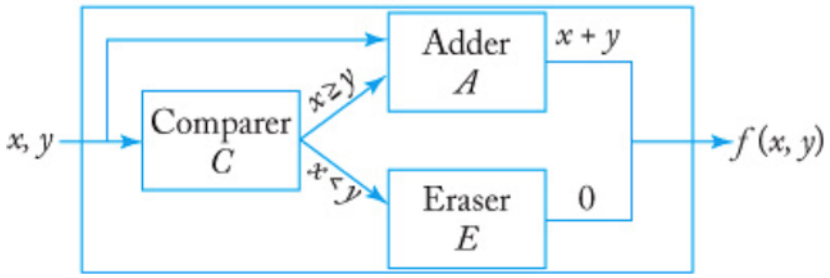


Figure 8: **Linz Fig. 9.8: Block Diagram**

We use such high-level diagrams in subsequent discussions of large computations. How can we justify that practice?

We can implement:

- the Comparer program C as suggested in Linz Example 9.11, using a Turing machine having states indexed with C
- the Adder program A as suggested in Linz Example 9.9, with states indexed with A
- the Eraser program E by constructing a Turing machine having states indexed with E

Comparer C carries out the computations

$$q_{C,0}w(x)0w(y) \vdash^* q_{A,0}w(x)0w(y), \text{ if } x \geq y,$$

and

$$q_{C,0}w(x)0w(y) \vdash^* q_{E,0}w(x)0w(y), \text{ if } x < y.$$

If $q_{A,0}$ and $q_{E,0}$ are the initial states of computations A and E, respectively, then C starts either A or E.

Adder A carries out the computation

$$q_{A,0}w(x)0w(y) \vdash^* q_{A,f}w(x+y)0.$$

And, Eraser E carries out the computation

$$q_{E,0}w(x)0w(y) \vdash^* q_{E,f}0.$$

The outer diagram in Linz Figure 9.8 thus represents a single Turing machine that combines the actions of machines C, A, and E as shown.

10.2.3 Using Pseudocode

In the *pseudocode* technique, we outline a computation using high-level descriptive phrases understandable to people. We refine and translate it to lower-level implementations later.

10.2.3.1 Macroinstructions A simple kind of pseudocode is the macroinstruction. A *macroinstruction* is a single statement shorthand for a sequence of lower-level statements.

We first define the macroinstructions in terms of the lower-level language. Then we compose macroinstructions into a larger program, assuming the relevant substitutions will be done.

10.2.3.2 Linz Example 9.13 For this example, consider the macroinstruction

if a then q_j else q_k.

This means:

- If the Turing machine reads an a, then it, regardless of its current state, transitions into state q_j without changing the tape content or moving the read-write head.
- If the symbol read is not an a, then it transitions into state q_k without changing anything.

We can implement this macroinstruction with several steps of a Turing machine:

$\delta(q_i, a) = (q_{j0}, a, R)$ for all $q_i \in Q$
$\delta(q_{j0}, c) = (q_j, c, L)$ for all $c \in \Gamma$

$\delta(q_i, b) = (q_{k0}, b, R)$ for all $q_i \in Q$ and all $b \in \Gamma - \{a\}$
$\delta(q_{k0}, c) = (q_k, c, L)$ for all $c \in \Gamma$

States q_{j0} and q_{k0} just back up Turing machine tape position one place.

Macroinstructions are expanded at each occurrence.

10.2.3.3 Subprograms While each occurrence of a macroinstruction is expanded into actual code, a *subprogram* is a single piece of code that is invoked repeatedly.

As in higher-level language programs, we must be able to call a subprogram and then, after execution, return to the calling point and resume execution without any unwanted effects.

How can we do this with Turing machines?

We must be able to:

- preserve information about the calling program's configuration (state, read-write head position, tape contents), so that it can be restored on return from the subprogram
- pass information from the calling program to the called subprogram and vice versa

We can do this by partitioning the tape into several regions. Linz Figure 9.9 illustrates this technique for a program A (a Turing machine) that calls a subprogram B (another Turing machine).

1. A executes in its own workspace.
2. Before transferring control to B, A writes information about its configuration and inputs for B into some separate region T.
3. After transfer, B finds its input in T.
4. B executes in its own separate workspace.
5. When B completes, it writes relevant results into T.
6. B transfers control back to A, which resumes and gets the needed results from T.

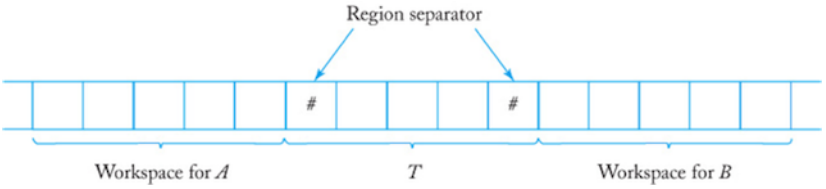


Figure 9: **Linz Fig. 9.9: Tape Regions for Subprograms**

Note: This is similar to what happens in an actual computer for a subprogram (function, procedure) call. The region T is normally a segment pushed onto the program's runtime stack or dynamically allocated from the heap memory.

10.2.3.4 Linz Example 9.14 Design a Turing machine that multiplies x and y, positive integers represented in unary notation.

Assume the initial and final tape configurations are as shown in Linz Figure 9.10.

We can multiply x by y by adding y to itself x times as described in the algorithm below.

Repeat until x contains no more 1's\
  Find a 1 in x and replace it with another symbol a\
  Replace the leftmost 0 by $0y$\
Replace all a's with 1's

Figure 10: **Linz Fig. 9.10: Multiplication**

Although the above description of the pseudocode approach is imprecise, the idea is sufficiently simple that it is clear we can implement it.

We have not proved that the block diagram, macroinstruction, or subprogram approaches will work in all cases. But the discussion in this section shows that it is plausible to use Turing machines to express complex computations.

10.3 Turing's Thesis

The *Turing thesis* is an hypothesis that *any computation that can be carried out by mechanical means can be performed by some Turing machine.*

This is a broad assertion. It is not something we can prove!

The Turing thesis is actually a definition of mechanical computation: *a computation is mechanical if and only if it can be performed by some Turing machine.*

Some arguments for accepting the Turing thesis as the definition of mechanical computation include:

1. Anything that can be computed by any existing digital computer can also be computed by a Turing machine.
2. There are no known problems that are solvable by what we intuitively consider an algorithm for which a Turing machine program cannot be written.
3. No alternative model for mechanical computation is more powerful than the Turing machine model.

The Turing thesis is to computing science as, for example, classical Newtonian mechanics is to physics. Newton's "laws" of motion cannot be proved, but they could possibly be invalidated by observation. The "laws" are plausible models that have enabled humans to explain much of the physical world for several centuries.

Similarly, we accept the Turing thesis as a basic "law" of computing science. The conclusions we draw from it agree with what we know about real computers.

The Turing thesis enables us to formalize the concept of algorithm.

Linz Definition 9.5 (Algorithm): An *algorithm* for a function $f : D \to R$ is a Turing machine M, which given as input any $d \in D$ on its tape, eventually halts with the correct answer $f(d) \in R$ on its tape. Specifically, we can require that

$$q_0 d \vdash_M^* q_f f(d), q_f \in F,$$

for all $d \in D$.

To prove that "there exists an algorithm", we can construct a Turing machine that computes the result.

However, this is difficult in practice for such a low-level machine.

An alternative is, first, to appeal to the Turing thesis, arguing that anything that we can compute with a digital computer we can compute with a Turing machine. Thus a program in suitable high-level language or precise pseudocode can compute the result. If unsure, then we can validate this by actually implementing the computation on a computer.

Note: A higher-level language is *Turing-complete* if it can express any algorithm that can be expressed with a Turing machine. If we can write a Turing machine simulator in that language, we consider the language Turing complete.

10.4 References

[1] Peter Linz. 2011. *Formal languages and automata* (Fifth ed.). Jones & Bartlett, Burlington, Massachusetts, USA.