Formal Languages

Turing's Thesis

Variations of the Turing Machine

Equivalence of Classes of Automata

- Two automata are equivalent if they accept the same language.
- Consider two classes of automata *C1* and *C2*. If for every automaton *M1* in *C1* there is an automaton *M2* in *C2* such that *L* (*M1*)= *L* (*M2*).

- To demonstrate the equivalence in connection with TM, we use the technique of simulation.

- Let *M* be an automaton. We say that another automaton *M*′ can simulate a computation of *M* if *M*′ can mimic the computation of *M*.

Same Power of two classes means:

To demonstrate the equivalence of two classes of automata, for every machine in one class, there is a machine in the second class capable of simulating it and vice versa

For any machine M_1 of first class

there is a machine M_2 of second class

such that: $L(M_1) = L(M_2)$

And vice-versa

Variations of the Standard TM Model

Turing machines with:

- Stay-Option
- Semi-Infinite Tape
- Off-Line
- Multitape
- Multidimensional
- Nondeterministic

Turing Machine with Stay Option:

- Standard Turing machine, the read-write head must move either to the right or to the left.
- Sometimes it is convenient to provide a third option, to have the read-write head stay in place after rewriting the cell content.

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$$

For each transition

$$\delta(q_i, a) = (q_j, b, L \text{ or } R),$$

we put into $\widehat{\delta}$

$$\widehat{\delta}(\widehat{q}_i, a) = (\widehat{q}_i, b, L \text{ or } R)$$

Turing Machine with Stay Option:

Now for the transition with stay option……

$$\delta\,(q_i, a) = (q_j, b, S),$$

we put into $\widehat{\delta}$ the corresponding transitions

$$\widehat{\delta}\,(\widehat{q}_i, a) = (\widehat{q}_{js}, b, R)\,,$$

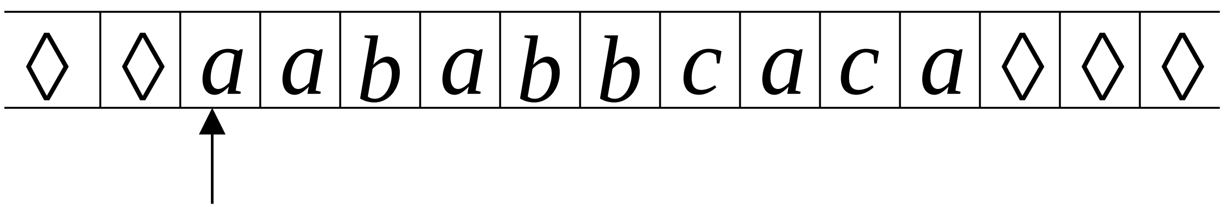
and

$$\widehat{\delta}\,(\widehat{q}_{js}, c) = (\widehat{q}_j, c, L)$$

for all $c \in \Gamma$.

Turing Machines with Stay-Option

The head can stay in the same position

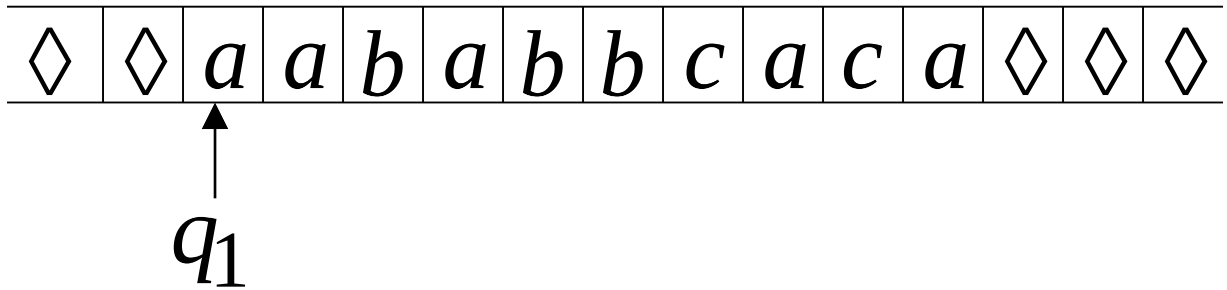


Left, Right, Stay

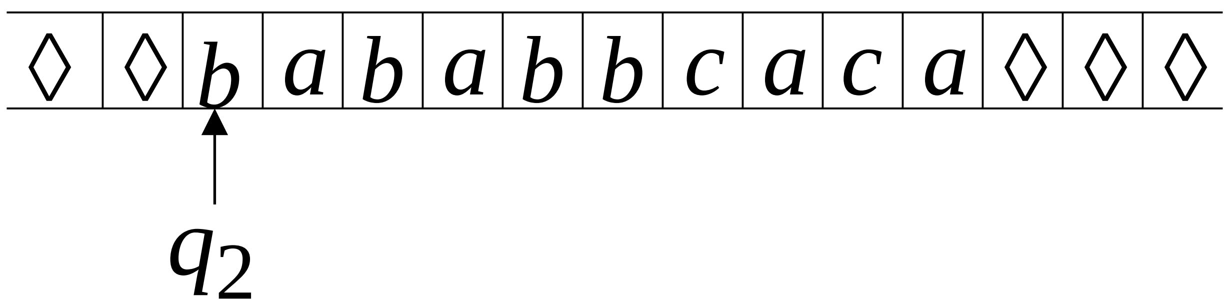
L,R,S: moves

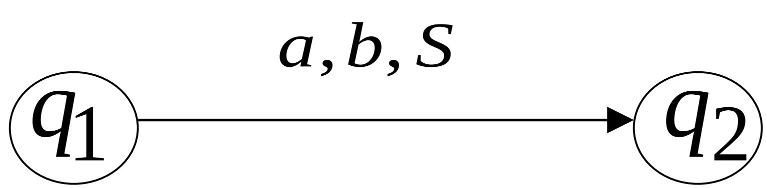
Example:

Time 1



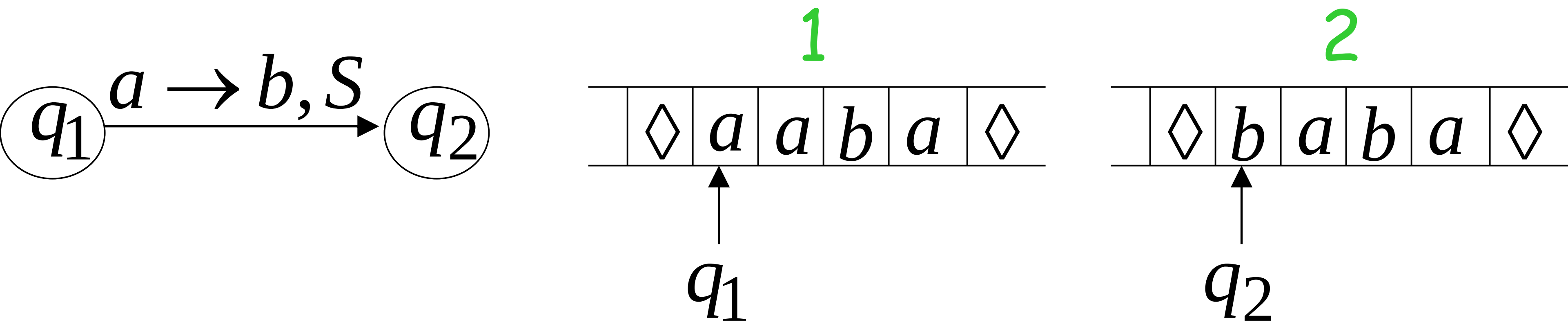
Time 2



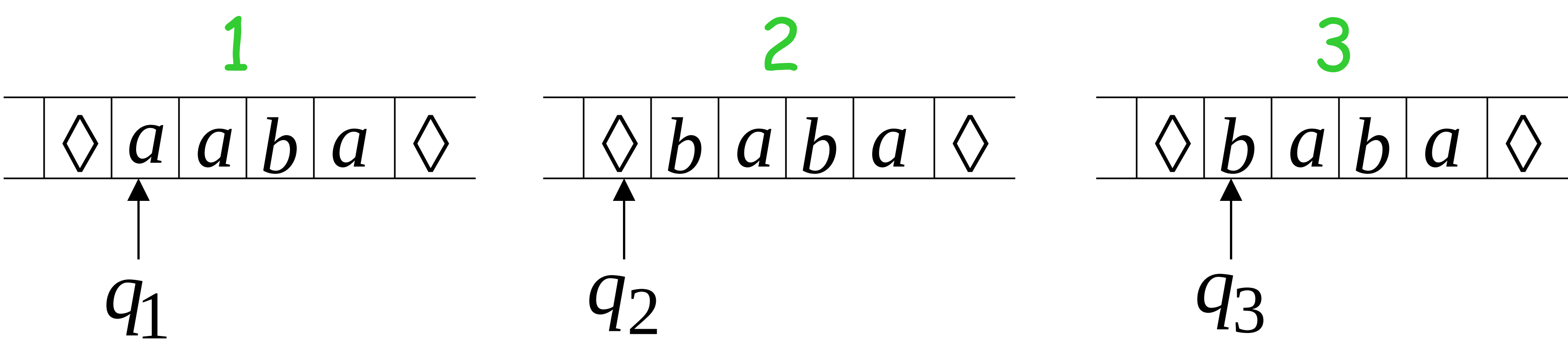


Example

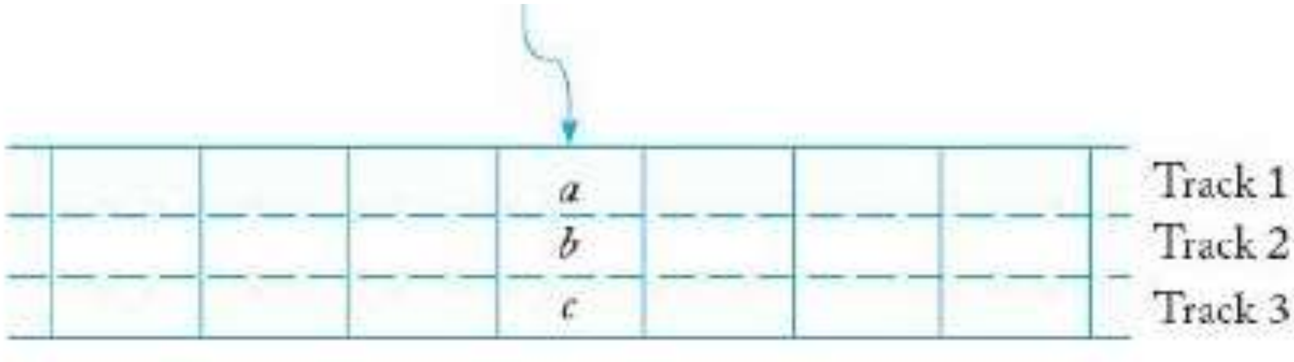
Stay-Option Machine:



Simulation in Standard Machine:

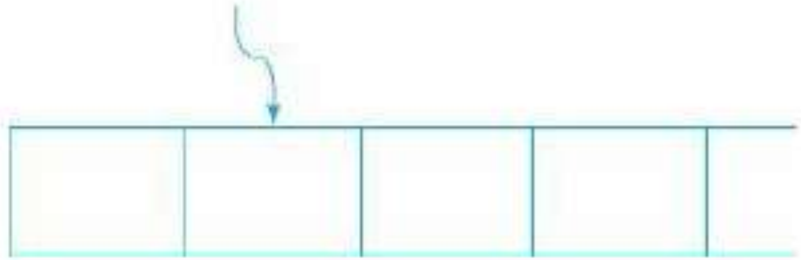


Multiple Tracks in TM



- we have divided each cell of the tape into three parts, called **tracks**, each containing one member of the triplet.
- Based on this visualization, such an automaton is sometimes called a Turing machine with **multiple tracks**.

Standard TM

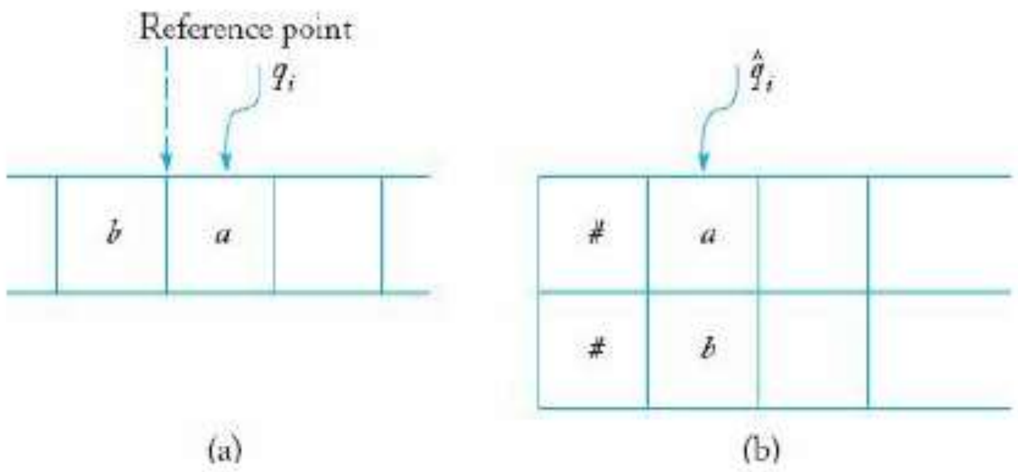


Semi Infinite tape TM

- To simulate a standard Turing machine *M* by a machine with a semi-infinite tape, we use the arrangement.
- The simulating machine has a tape with two tracks.
- On the upper one, we keep the information to the right of some reference point on *M*'s tape

- The reference point can be the position of the read-write head at the start of the computation.
- The lower track contains the left part of M's tape in reverse order.
- Assume that the machine to be simulated and the simulating machine are in the respective configurations shown in Figure 10.4 and that the move to be simulated is generated by $\delta(q_i, a) = (q_j, c, L)$.

Figure 10.4



(a) Machine to be simulated.

(b) Simulating machine.

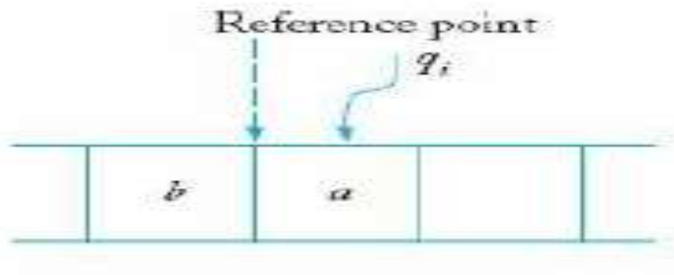
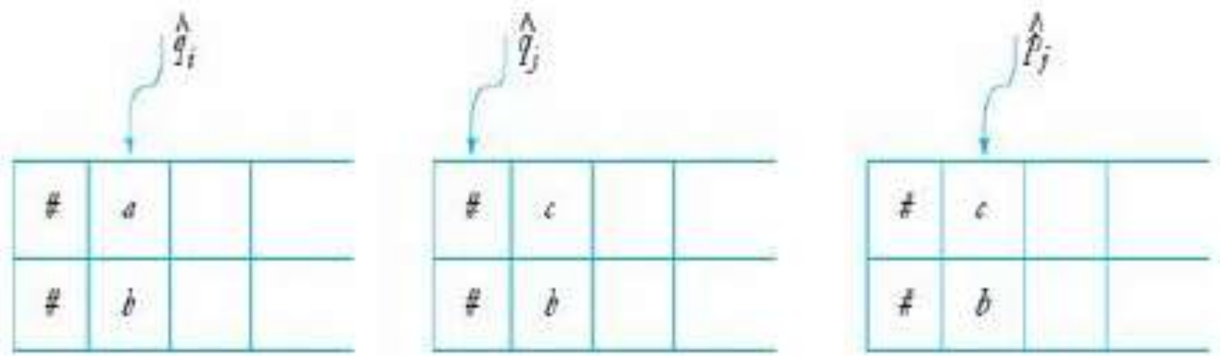


Figure 10.5



Sequence of configurations in simulating $\delta(q_i, a) = (q_j, c, L)$.

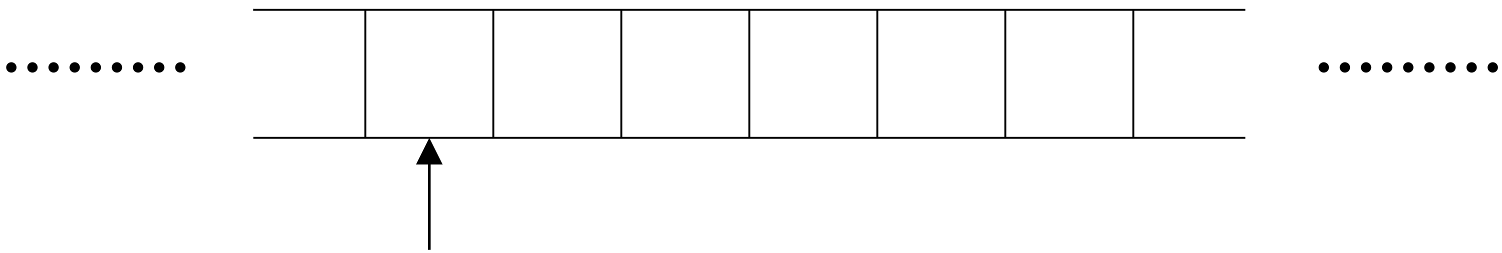
$$\widehat{\delta}\left(\widehat{q}_i, (a, b)\right) = \left(\widehat{q}_j, (c, b), L\right)$$

$$\widehat{\delta}\left(\widehat{q}_i, (\#, \#)\right) = \left(\widehat{p}_j, (\#, \#), R\right)$$ where $\widehat{P}_j \in Q_L$, and $\widehat{q}_i \in Q_U$

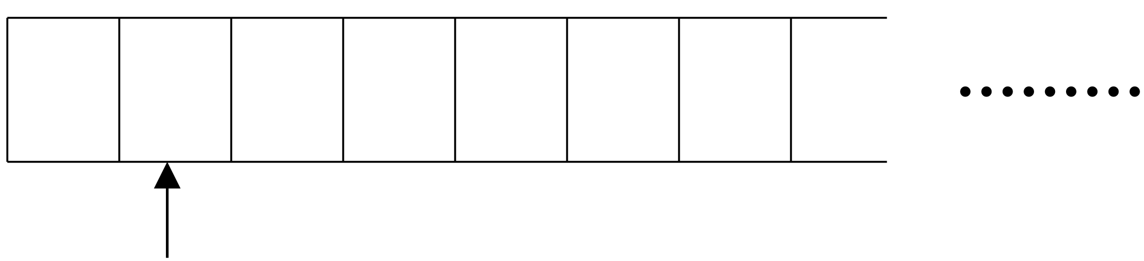
and Pj will work on the lower track.

Semi-infinite tape machines simulate Standard Turing machines:

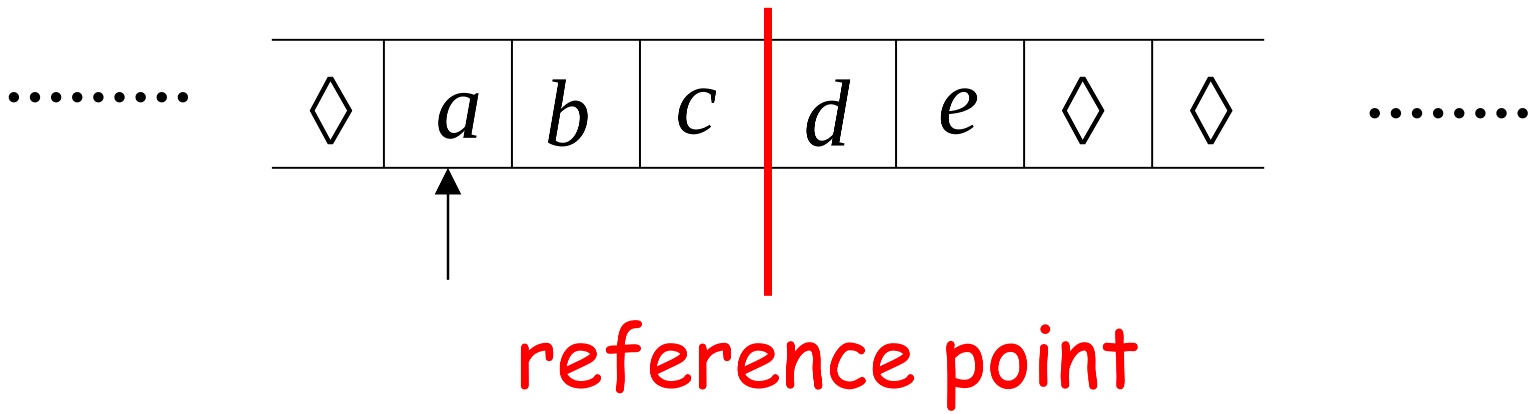
Standard machine



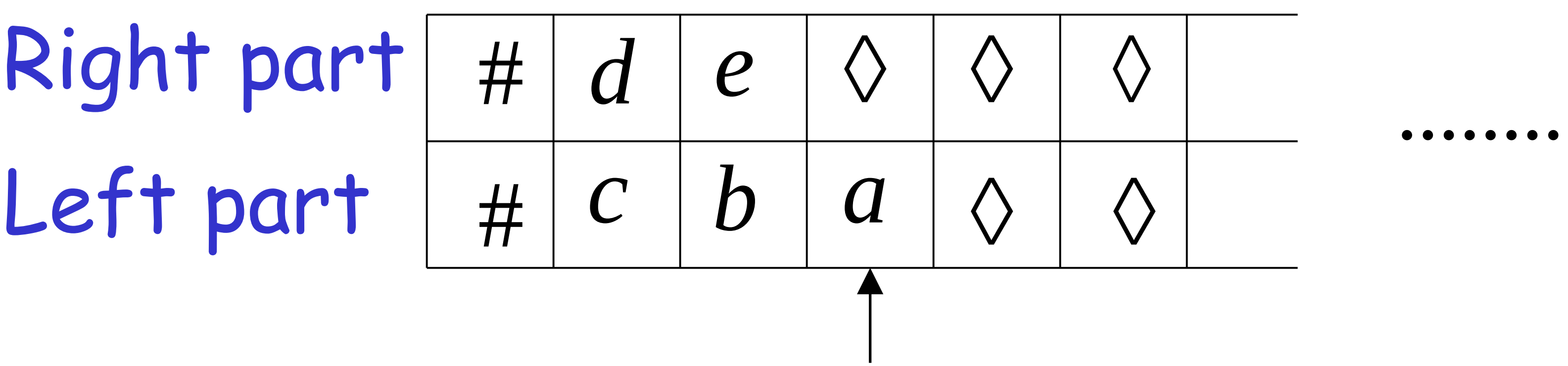
Semi-infinite tape machine



Standard machine

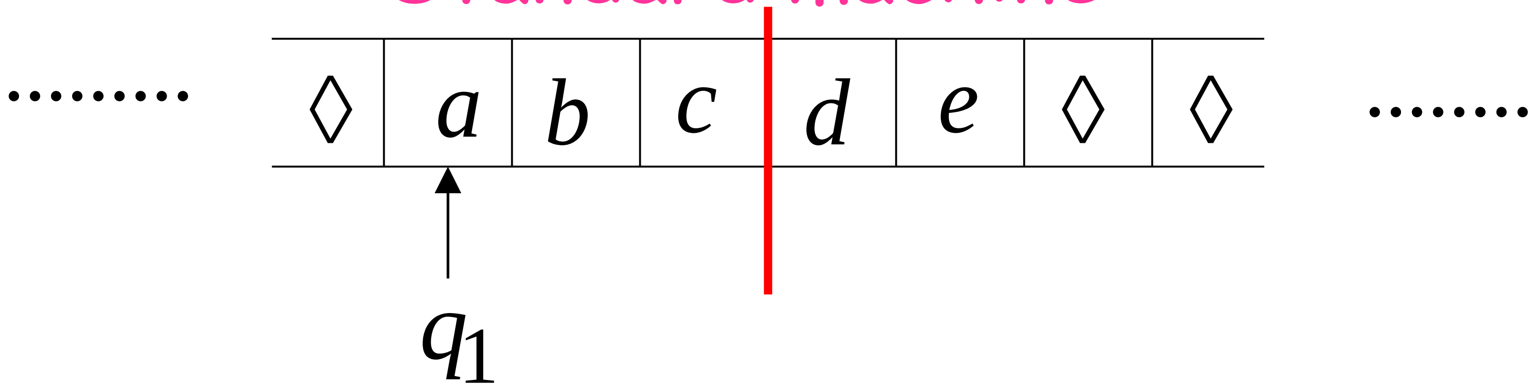


Semi-infinite tape machine with two tracks

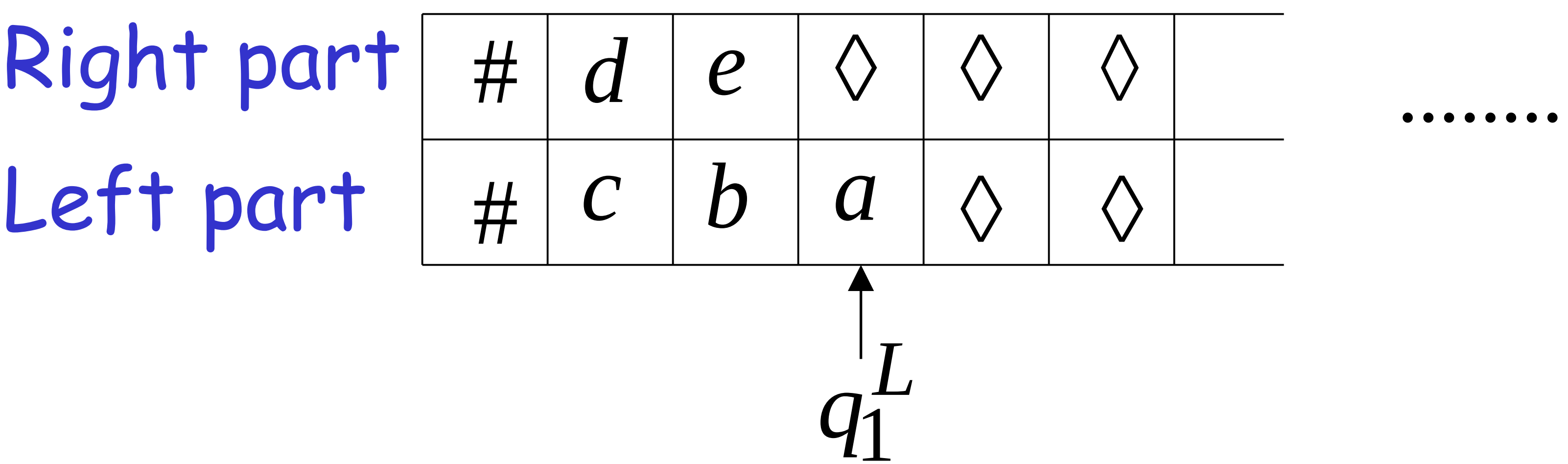


Time 1

Standard machine

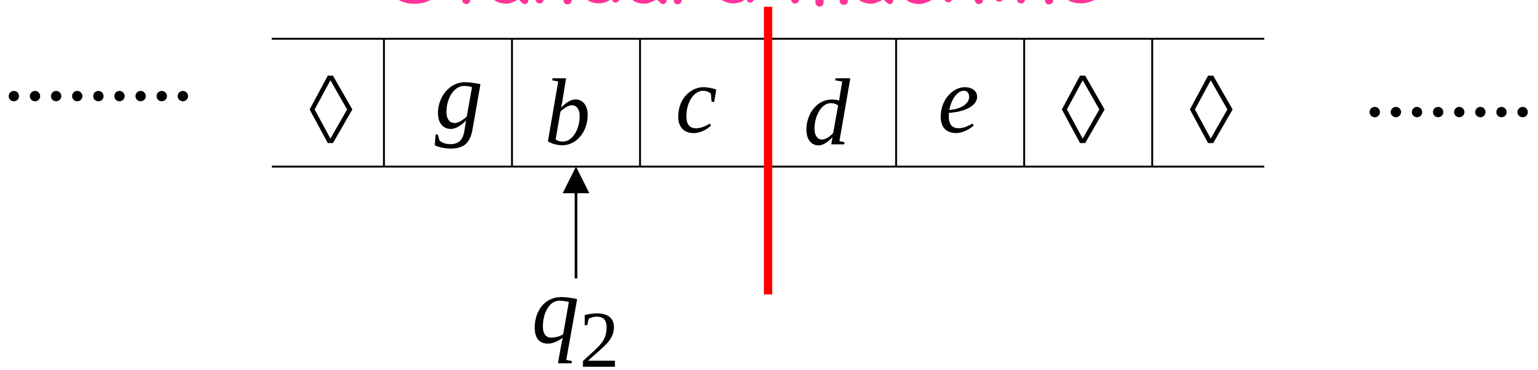


Semi-infinite tape machine



Time 2

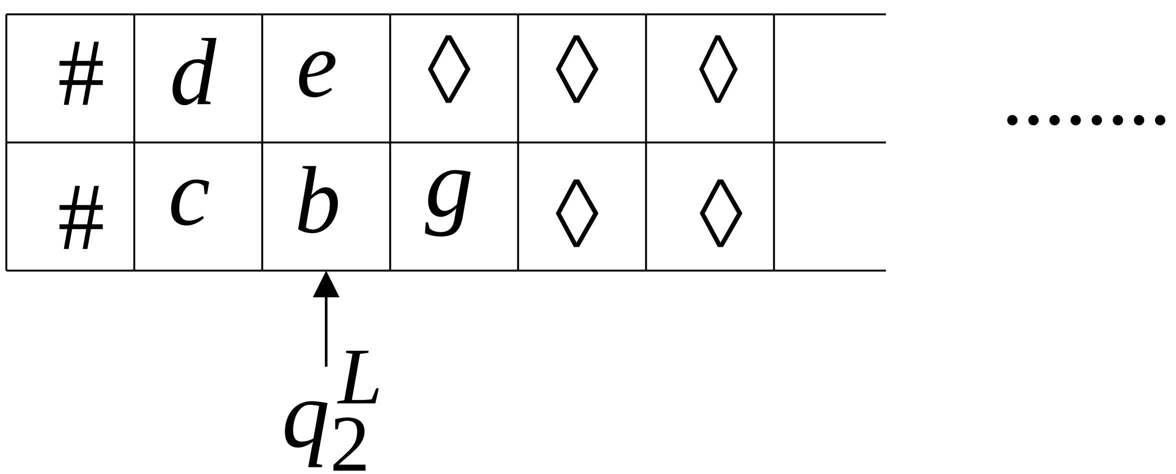
Standard machine



Semi-infinite tape machine

Right part

Left part



At the border:

Semi-infinite tape machine

Right part

$$q_1^R \xrightarrow{(\#,\#)\rightarrow(\#,\#),R} q_1^L$$

Left part

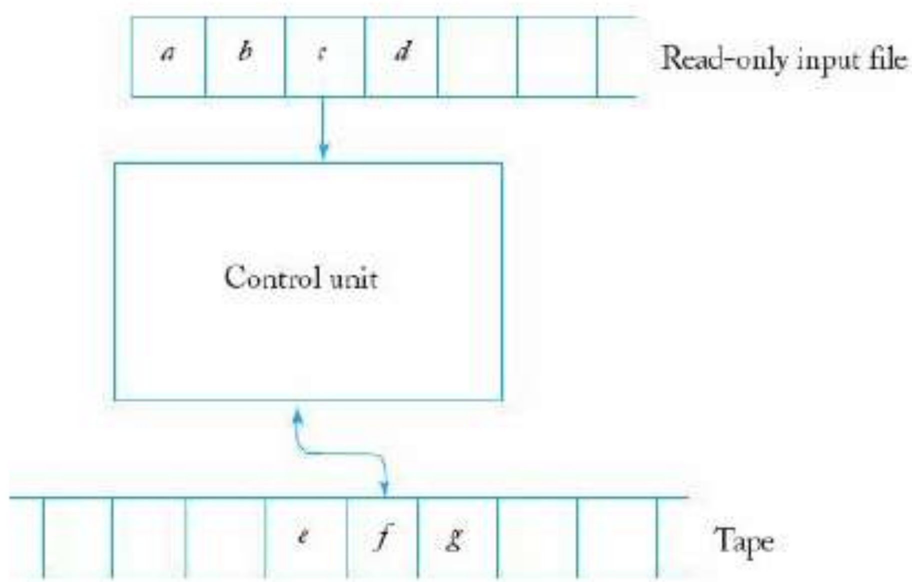
$$q_1^L \xrightarrow{(\#,\#)\rightarrow(\#,\#),R} q_1^R$$

The Off-Line Turing Machine

- If we put the input file back into the picture, then it is known as an **off-line Turing machine** .
- In such a machine, each move is governed by the internal state, what is currently read from the input file, and what is seen by the read-write head.

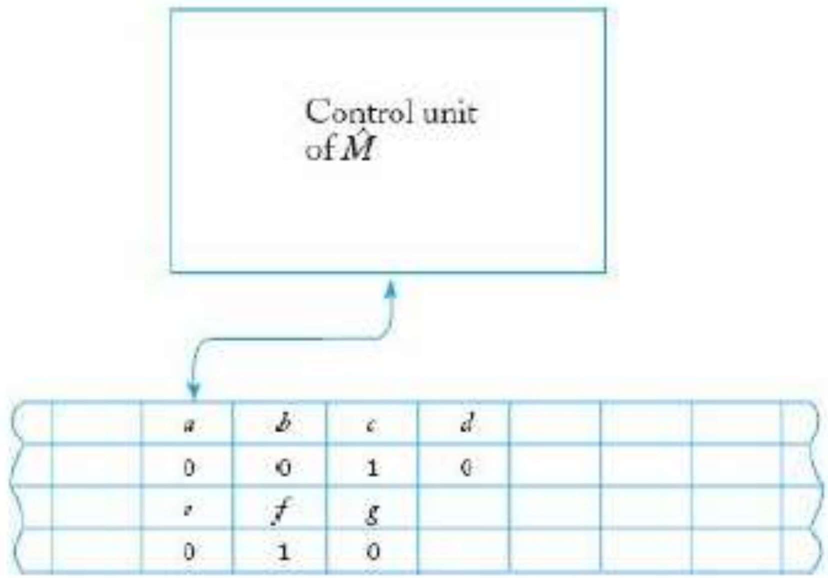
A schematic representation of an off-line machine is shown in Figure 10.6.

Figure 10.6



- A standard machine can simulate the computation of an off-line machine by using the four-track arrangement shown in Figure 10.7.

Figure 10.7



- The first track has the input, the second marks the position at which the input is read, the third represents the tape of M, and the fourth shows the position of M's read-write head.

Turing Machines with More Complex Storage

A) Multitape Turing

$$\delta : Q \times \Gamma^n \rightarrow Q \times \Gamma^n \times \{L, R\}^n$$

- A multitape Turing machine is a Turing machine with several tapes, each with its own independently controlled read-write head (Figure 10.8 and Figure 10.9).

Figure10.8

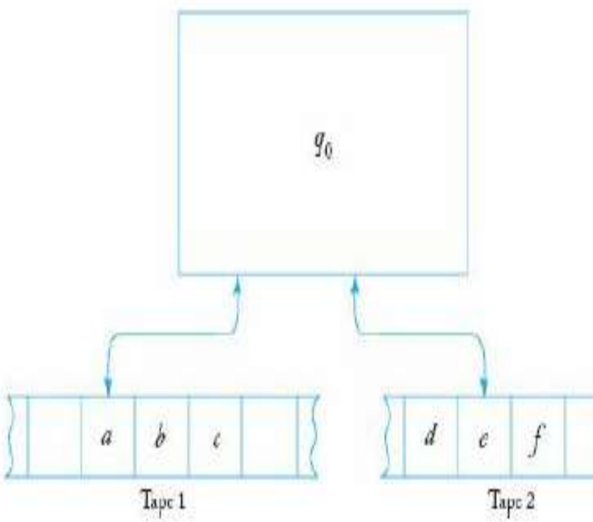
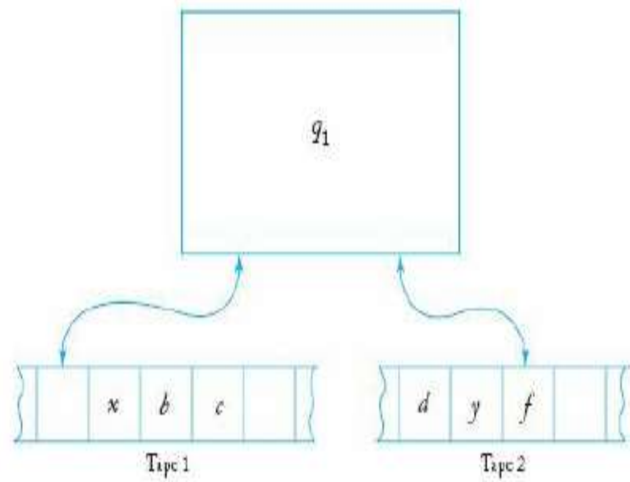
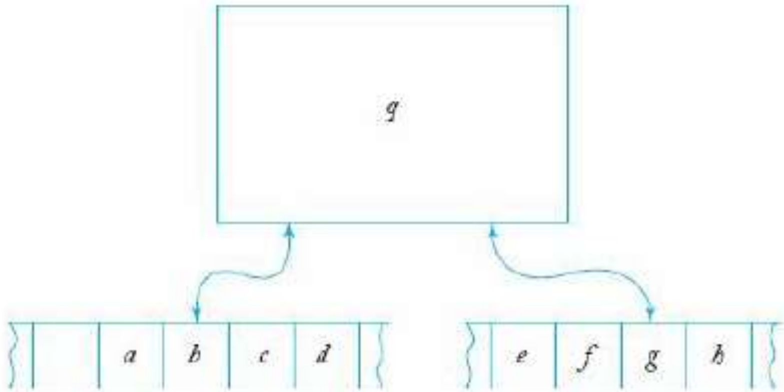


Figure10.9

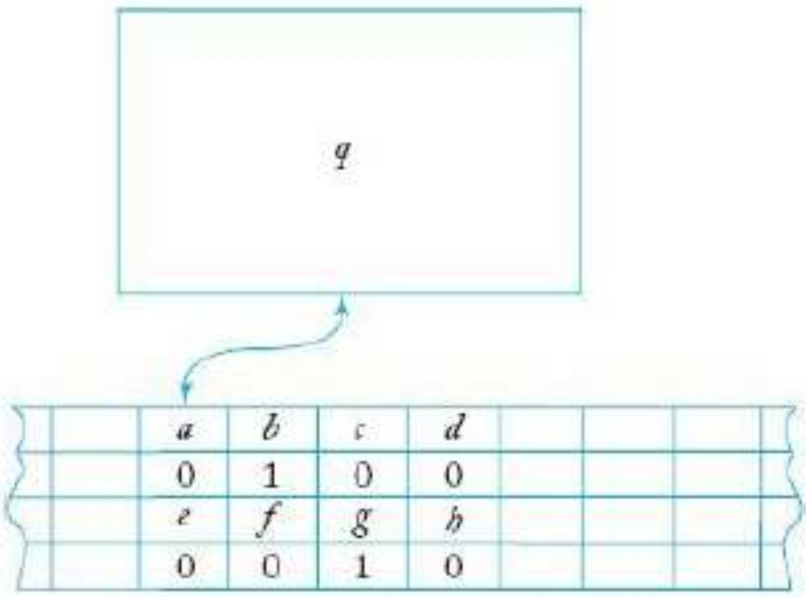


$$\delta\,(q_0, a, e) = (q_1, x, y, L, R)$$

The representation of a multitape machine by a single-tape machine is similar to that used in the simulation of an off-line machine.



Multitape TM

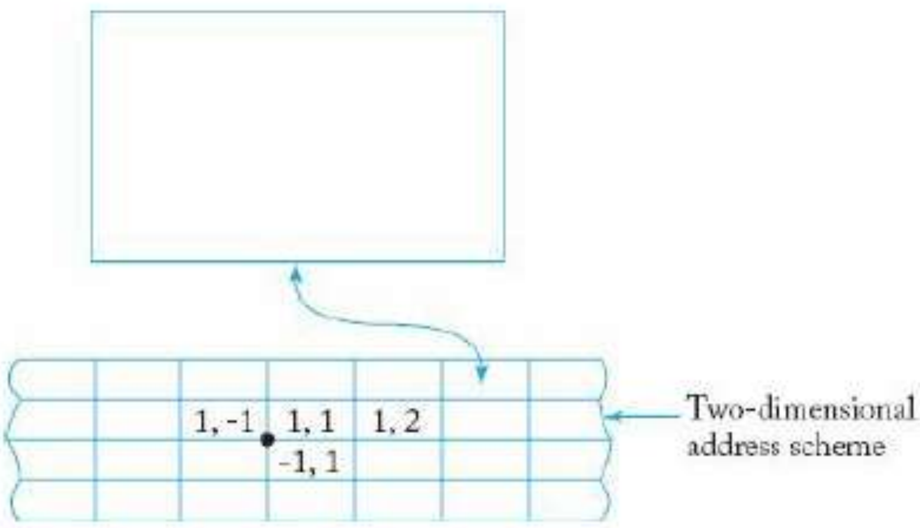


Simulated in single tape

Multidimensional Turing Machines

- A multidimensional Turing machine is one in which the tape can be viewed as extending infinitely in more than one dimension.
- A diagram of a two-dimensional Turing machine is shown in Figure 10.12

Figure 10.12



The formal definition of a two-dimensional Turing machine involves a transition function δ of the form

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, U, D\},$$

where U and D specify movement of the read-write head up and down, respectively.

- To simulate this machine on a standard Turing machine, we can use the two-track model depicted i n Figure 10.13 and the configuration in which cell (1, 2) contains a and cell (10, – 3) contains *b* is shown as

Figure 10.13

a				*b*					
1	#	2	#	1	0	#	–	3	#

The two-track tape of the simulating machine will use one track to <u>store cell contents</u> and the other one to <u>keep the associated address</u>.

Nondeterministic Turing Machines

A nondeterministic Turing machine is an automaton as given by a function

$$\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{L, R\}}.$$

If a Turing machine has transitions specified by

$$\delta\,(q_0, a) = \{(q_1, b, R), (q_2, c, L)\},$$

it is nondeterministic. The moves

$$q_0 aaa \vdash b q_1 aa$$

and

$$q_0 aaa \vdash q_2 \square caa$$

are both possible.

- A nondeterministic Turing machine M is said to accept a language L if, for all $w \in L$, at least one of the possible configurations accepts w.

- There may be branches that lead to nonaccepting configurations, while some may put the machine into an infinite loop.

- A nondeterministic Turing machine M is said to *decide* a language L if, for all $w \in \Sigma^*$, there is a path that leads either to acceptance or rejection.

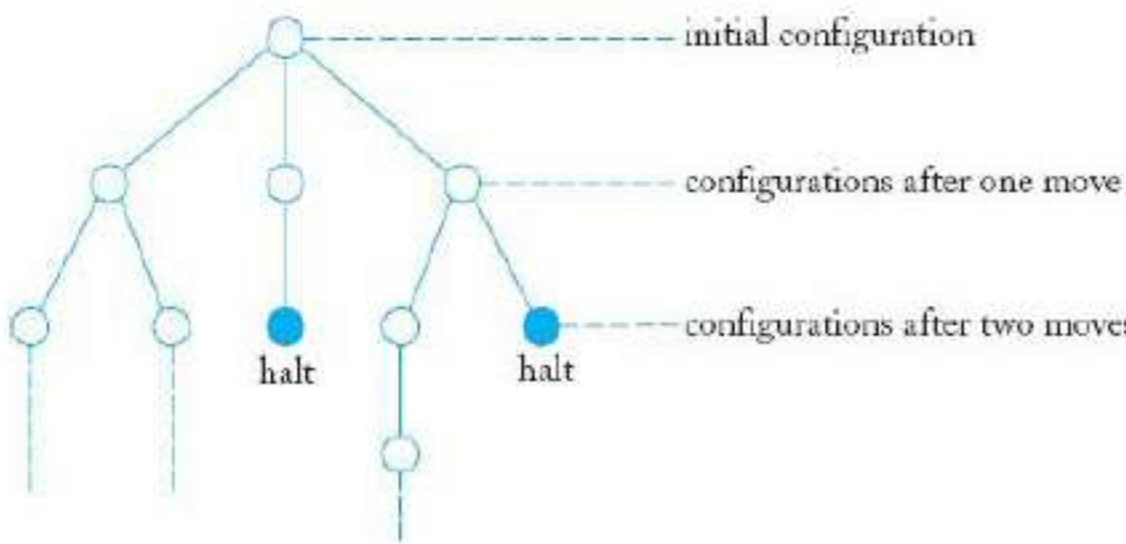
- One way to visualize the simulation is to use a standard Turing machine, keeping all possible instantaneous descriptions of the nondeterministic machine on its tape, separated by some convention.

- The symbols × are used to delimit the area of interest, while + separates individual instantaneous descriptions.

Figure 10.14

Figure 10.15

Decision tree



Definition 10.3

A nondeterministic Turing machine M is said to accept a language L if, for all $w \in L$, at least one of the possible configurations accepts w. There may be branches that lead to nonaccepting configurations, while some may put the machine into an infinite loop. But these are irrelevant for acceptance.

A nondeterministic Turing machine M is said to *decide* a language L if, for all $w \in \Sigma^*$, there is a path that leads either to acceptance or rejection.

A Universal Turing Machine

A universal Turing machine M_u is an automaton that, given as input the description of any Turing machine M and a string w, can simulate the computation of M on w.

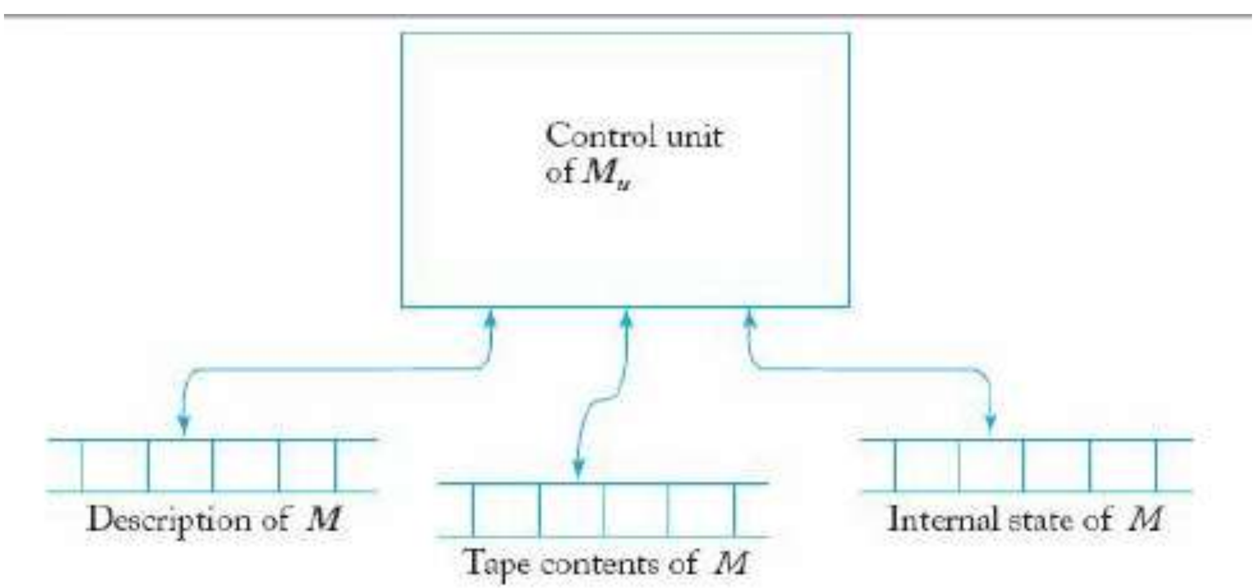
- Assume that $Q = \{q_1, q_2, \ldots, q_n\}$, with q_1 the initial state, q_2 the single final state, and $\Gamma = \{a_1, a_2, \ldots a_m\}$, where a_1 represents the blank.
- We then select an encoding in which q_1 is represented by 1, q_2 is represented by 11, and so on.
- Similarly, a_1 is encoded as 1, a_2 as 11, etc. The symbol 0 will be used as a separator between the 1's.
- For L - encoded as 1 and Right – encoded as 11

For example, δ (q_1, a_2) = (q_2, a_3, L) might appear as

…10110110111010….

It follows from this that any Turing machine has a finite encoding as a string on $\{0,1\}^+$

A universal Turing machine M_u then has an input alphabet that includes {0, 1} and the structure of a multitape machine, as shown in Figure 10.16.



- For any input M and *w*, tape 1 will keep an encoded definition of *M*.
- Tape 2 will contain the tape contents of *M*, and tape 3 the internal state of *M*.
- *Mu* looks first at the contents of tapes 2 and 3 to determine the configuration of M.
- It then consults tape 1 to see what *M* w would do in this configuration.
- Finally, tapes 2 and 3 will be modified to reflect the result of the move.

A limitation of Turing Machines:

$$\text{Turing Machines are } \underbrace{\text{“hardwired”}}_{\substack{\text{they execute} \\ \text{only one program}}}$$

Real Computers are re-programmable

Solution: Universal Turing Machine

Attributes:

- Reprogrammable machine
- Simulates any other Turing Machine

Universal Turing Machine

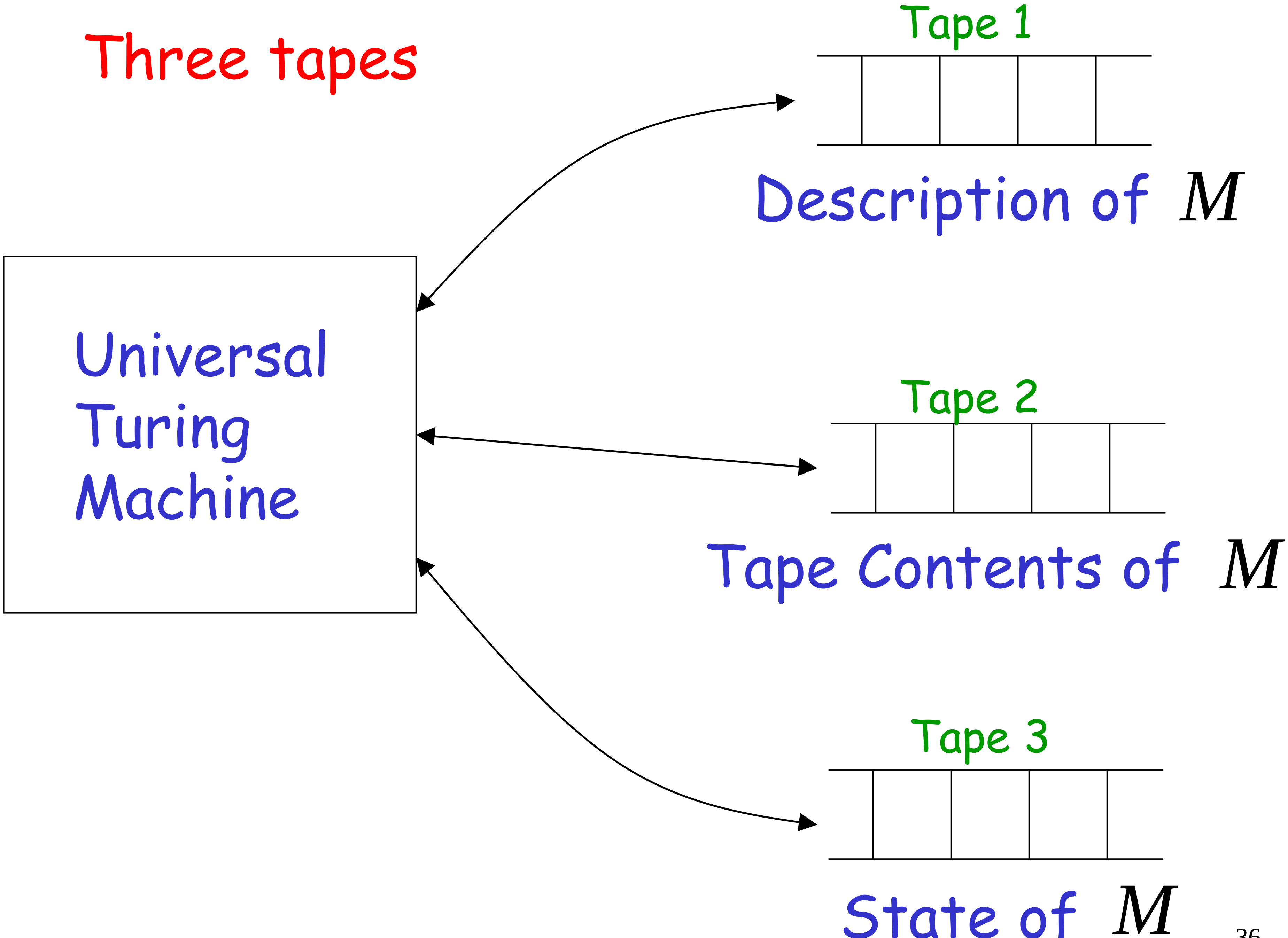
simulates any other Turing Machine M

Input of Universal Turing Machine:

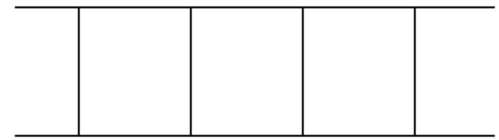
Description of transitions of M

Initial tape contents of M

Three tapes


Universal Turing Machine
Tape 1
Description of M
Tape 2
Tape Contents of M
Tape 3
State of M

Tape 1



Description of M

We describe Turing machine M
as a string of symbols:

We encode M as a string of symbols

Alphabet Encoding

Symbols:	a	b	c	d	$\cdots$
	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	
Encoding:	1	11	111	1111	

State Encoding

States:	q_1	q_2	q_3	q_4	$\cdots$
	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	
Encoding:	1	11	111	1111	

Head Move Encoding

Move:	L	R
	$\downarrow$	$\downarrow$
Encoding:	1	11

Transition Encoding

Transition: $\delta(q_1, a) = (q_2, b, L)$

Encoding: $1\,0\,1\,0\,11\,0\,11\,0\,1$

separator

Machine Encoding

Transitions:

$$\delta(q_1, a) = (q_2, b, L) \qquad \delta(q_2, b) = (q_3, c, R)$$

Encoding:

$$10101101101\ 00\ 1101101110111011$$

separator

Tape 1 contents of Universal Turing Machine:

encoding of the simulated machine M
as a binary string of 0's and 1's

A Turing Machine is described
with a binary string of 0's and 1's

Therefore:

The set of Turing machines forms a language:

each string of the language is
the binary encoding of a Turing Machine

Linear Bounded Automata

- Linear bounded automaton, like a standard Turing machine, has an unbounded tape.
- But how much of the tape can be used is a function of the input.
- In particular, we restrict the usable part of the tape to exactly the cells taken by the input.
- To enforce this, we can envision the input as bracketed by two special symbols, the **left-end marker** [and the **right-end marker**].
- For an input w, the initial configuration of the Turing machine is given by the instantaneous description *q*0 [*w*].
- The end markers cannot be rewritten, and the read-write head cannot move to the left of [or to the right of].

- A linear bounded automaton is a nondeterministic Turing machine M = (Q, Σ, Γ,δ, q_0, ,F), subject to the restriction that Σ must contain two special symbols [and], such that δ (q_i,[) can contain only elements of the form (q_j, [,R), and δ (q_i,]) can contain only elements of the form (q_j,],L).

Definition:

A string w is accepted by a linear bounded automaton if there is a possible sequence of moves

$$q_0 [w] \overset{*}{\vdash} [x_1 q_f x_2]$$

for some $q_f \in F$, $x_1, x_2 \in \Gamma^*$. The language accepted by the lba is the set of all such accepted strings.

Formal Languages

Recursively Enumerable Languages

Recursive Languages

Definition:

A language is **recursively enumerable** if some Turing machine accepts it

Let L be a recursively enumerable language

and M the Turing Machine that accepts it

For string w :

if $w \in L$ then M halts in a final state

if $w \notin L$ then M halts in a non-final state

or loops forever

Definition:

A language is **recursive**
if some Turing machine accepts it
and halts on any input string

In other words:

A language is recursive if there is
a membership algorithm for it

Let L be a recursive language

and M the Turing Machine that accepts it

For string w :

if $w \in L$ then M halts in a final state

if $w \notin L$ then M halts in a non-final state

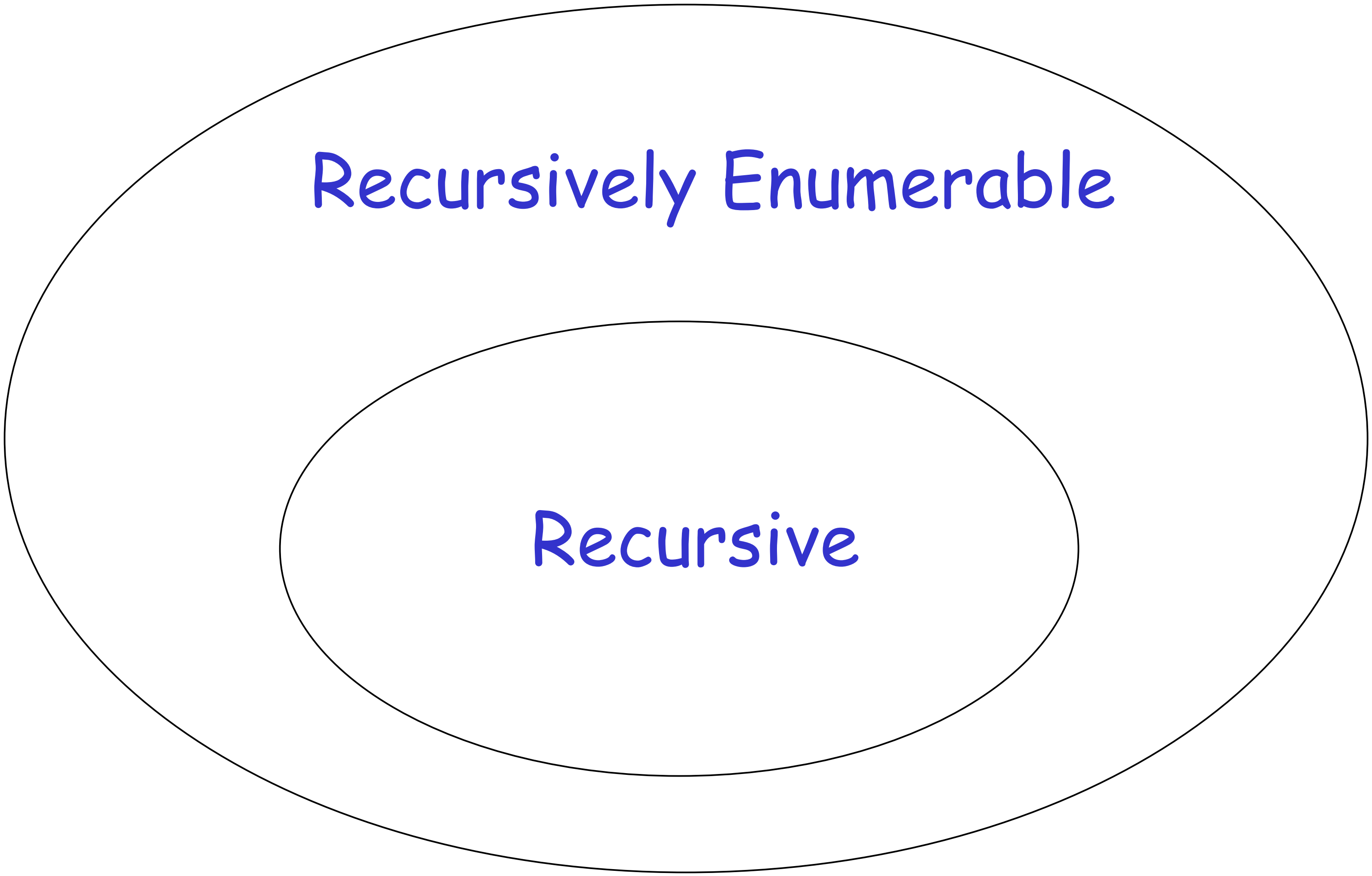
Recursive Language:

- A language 'L' is said to be recursive if there exist a TM which will accept all the strings in 'L' and reject all the strings not in 'L'.
- The TM will halt every time and give an answer (accepted or rejected) for each and every string input.

Recursively Enumerable Language:

- A language 'L' is said to be recursively enumerable language if there exist a TM which will accept and therefore halt for all input strings which are in L.
- But may or may not halt for the strings which are not in 'L'.

Non Recursively Enumerable



A language L is said to be **recursively enumerable** if there exists a Turing machine that accepts it.

This definition implies only that there exists a Turing machine M, such that, for every $w \in L$,

$$q_0 w \overset{*}{\vdash}_M x_1 q_f x_2,$$

with q_f a final state. The definition says nothing about what happens for w not in L; it may be that the machine halts in a nonfinal state or that it never halts and goes into an infinite loop.

A language L on Σ is said to be **recursive** if there exists a Turing machine M that accepts L and that halts on every w in Σ^+. In other words, a language is recursive if and only if there exists a membership algorithm for it.

Unrestricted Grammars

A grammar $G = (V, T, S, P)$ is called **unrestricted** if all the productions are of the form

$$u \rightarrow v,$$

where u is in $(V \cup T)^+$ and v is in $(V \cup T)^*$.

- In an unrestricted grammar, essentially no conditions are imposed on the productions.
- Any number of variables and terminals can be on the left or right, and these can occur in any order.
- There is only one restriction: λ is not allowed as the left side of a production.

Unrestricted Grammars contd…

- Any language generated by an unrestricted grammar is recursively enumerable.

- For every recursively enumerable language *L*, there exists an unrestricted grammar *G*, such that $L = L(G)$.

Context-Sensitive Grammars and Languages

A grammar $G = (V, T, S, P)$ is said to be **context-sensitive** if all productions are of the form

$$x \to y,$$

where $x, y \in (V \cup T)^+$ and

$$|x| \leq |y|.$$

Context-Sensitive Languages

A language L is said to be context-sensitive if there exists a context-sensitive grammar G, such that $L = L(G)$ or $L = L(G) \cup \{\lambda\}$.

Why λ included in language but not in grammar.....

- According to the definition of Context Sensitive Grammar it implies that $x \rightarrow \lambda$ is not allowed.
- So that a context-sensitive grammar can never generate a language containing the empty string.
- Yet, every context-free language without λ can be generated by a special case of a context sensitive grammar, say by one in Chomsky or Greibach normal form.
- By including the empty string in the definition of a context-sensitive language (but not in the grammar), we can claim that the family of context-free languages is a subset of the family of context-sensitive languages.

Context-Sensitive Languages and Linear Bounded Automata

A language L is said to be context-sensitive if there exists a context-sensitive grammar G, such that $L = L(G)$ or

$L = L(G) \cup \{\lambda\}$.

- If a language L is accepted by some linear bounded automaton M, then there exists a context-sensitive grammar that generates L.

Relation Between Recursive and Context-Sensitive Languages

As we know that every context-sensitive language is accepted by some Turing machine and is therefore recursively enumerable.

Example for an unrestricted grammar:

$$S \to aBc$$

$$aB \to cA$$

$$Ac \to d$$

Why not context-sensitive?

$$S \to aBc$$

$$aB \to cA$$

$$Ac \to d$$

$$|x| \leq |y| .$$

So…not context sensitive

The language $L = \{a^n b^n c^n : n \geq 1\}$ is a context-sensitive language. We show this by exhibiting a context-sensitive grammar for the language. One such grammar is

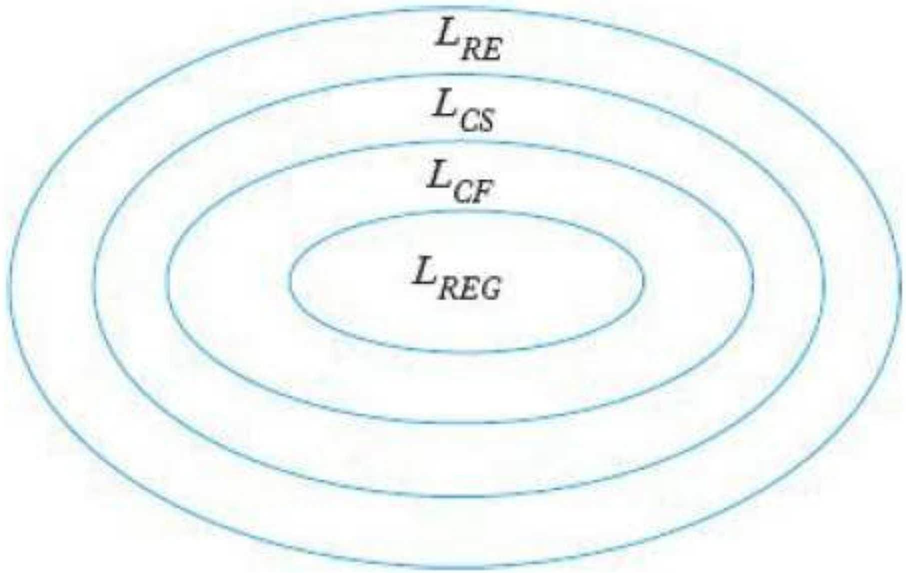
$$
\begin{aligned}
S &\to abc|aAbc, \\
Ab &\to bA, \\
Ac &\to Bbcc, \\
bB &\to Bb, \\
aB &\to aa|aaA.
\end{aligned}
$$

We can see how this works by looking at a derivation of $a^3b^3c^3$.

$$
\begin{aligned}
S &\Rightarrow aAbc \Rightarrow abAc \Rightarrow abBbcc \\
&\Rightarrow aBbbcc \Rightarrow aaAbbcc \Rightarrow aabAbcc \\
&\Rightarrow aabbAcc \Rightarrow aabbBbccc \\
&\Rightarrow aabBbbccc \Rightarrow aaBbbbccc \\
&\Rightarrow aaabbbccc.
\end{aligned}
$$

The Chomsky Hierarchy

- Type 0 languages are those generated by unrestricted grammars, that is, the recursively enumerable languages.
- Type 1 consists of the context-sensitive languages
- Type 2 consists of the context-free languages
- Type 3 consists of the regular languages.



original Chomsky hierarchy

Chomsky hierarchy

Hierarchy of grammars according to Chomsky is explained below as per the grammar types –

Type 0. Unrestricted grammars

Turing Machine (TM)

Type 1. Context-sensitive grammars

Linear Bounded Automaton (LBA)

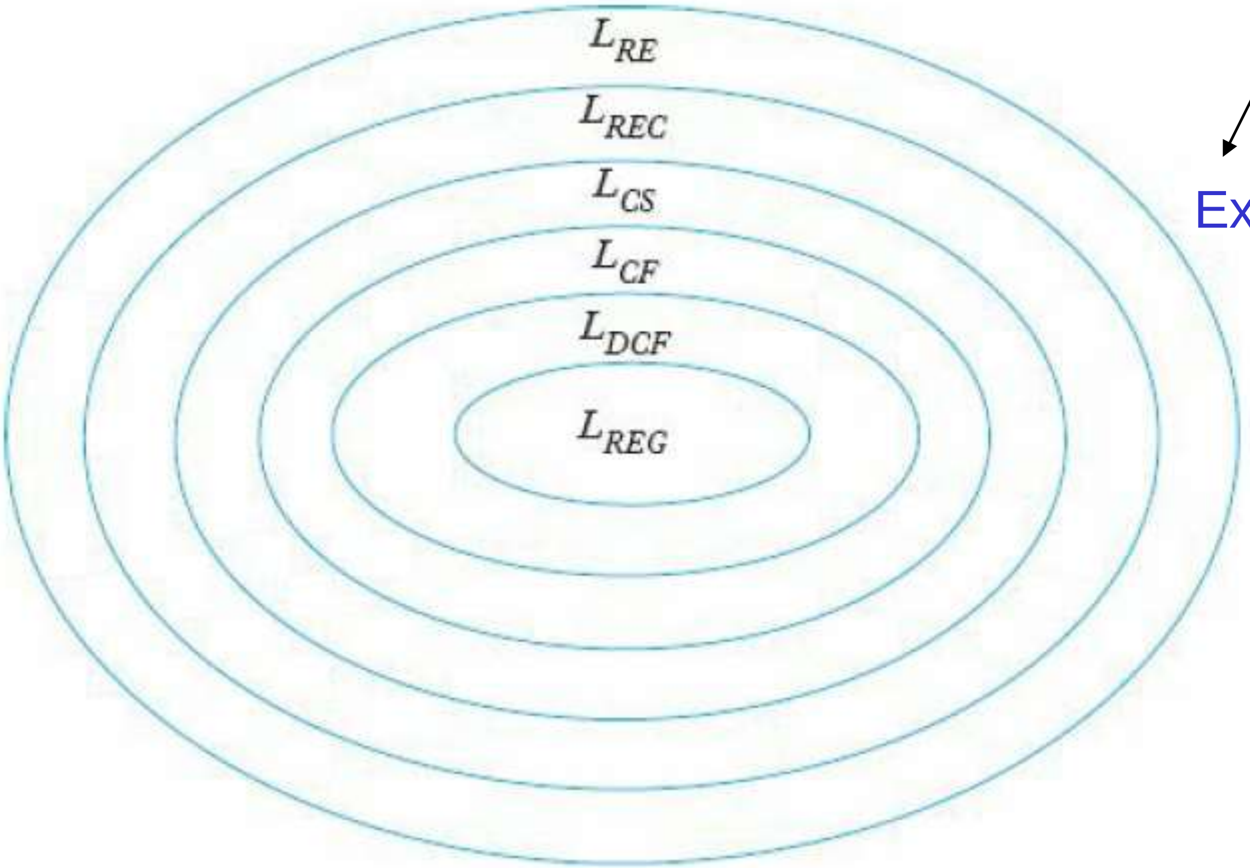
Type 2. Context-free grammars

Pushdown Automaton (PDA)

Type 3. Regular grammars

Finite Automaton (FA)

Including the families of deterministic context-free languages(L_{DCF}) and recursive languages (L_{REC}), we arrive at the extended hierarchy shown in



Extended hierarchy

A special subclass of context-free languages are the deterministic context-free languages which are defined as the set of languages accepted by a deterministic pushdown automaton

Computability and Decidability:

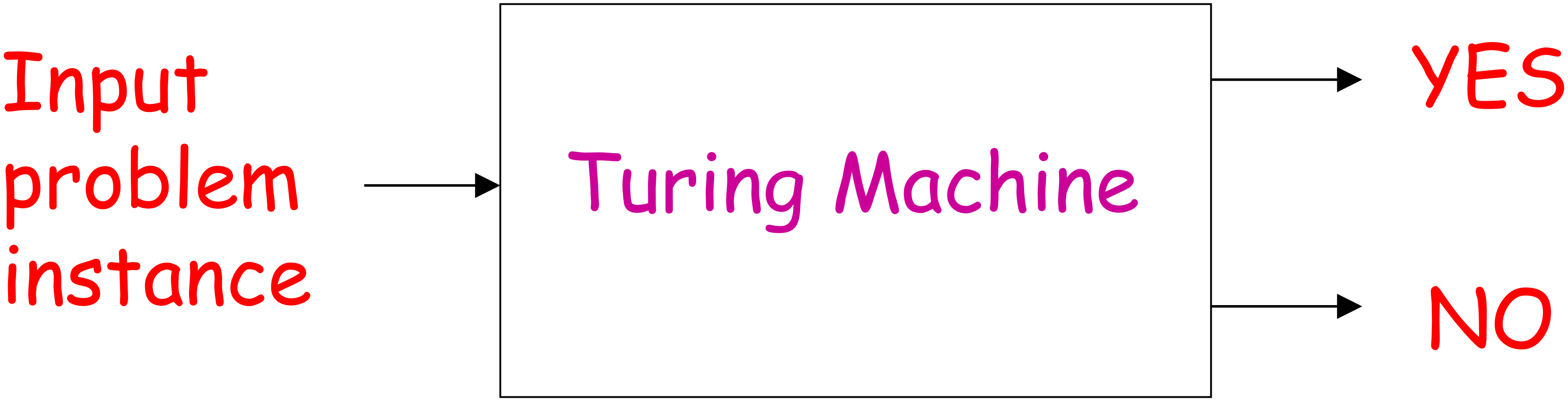
- A function *f* on a certain domain is said to be computable if there exists a Turing machine that computes the value of *f* for all arguments in its domain.
- A function is uncomputable if no such Turing machine exists.
- There may be a Turing machine that can compute *f* on part of its domain, but we call the function computable only if there is a Turing machine that computes the function on the whole of its domain.
- The result of a computation is a simple “yes” or “no.” In this case, we talk about a problem being **decidable** or **undecidable**.
- By a *problem* we will understand a set of related statements, each of which must be either true or false.
- For example, we consider the statement “For a context-free grammar *G*, the language *L* (*G*) is ambiguous.”
- For some *G* this is true, for others it is false, but clearly we must have one or the other.
- The problem is to decide whether the statement is true for any *G* we are given.

A problem is decidable if some Turing machine decides (solves) the problem

Decidable problems:

- Does Machine M have three states ?
- Is string w a binary number?
- Does DFA M accept any input?

The Turing machine that decides (solves) a problem answers YES or NO for each instance of the problem

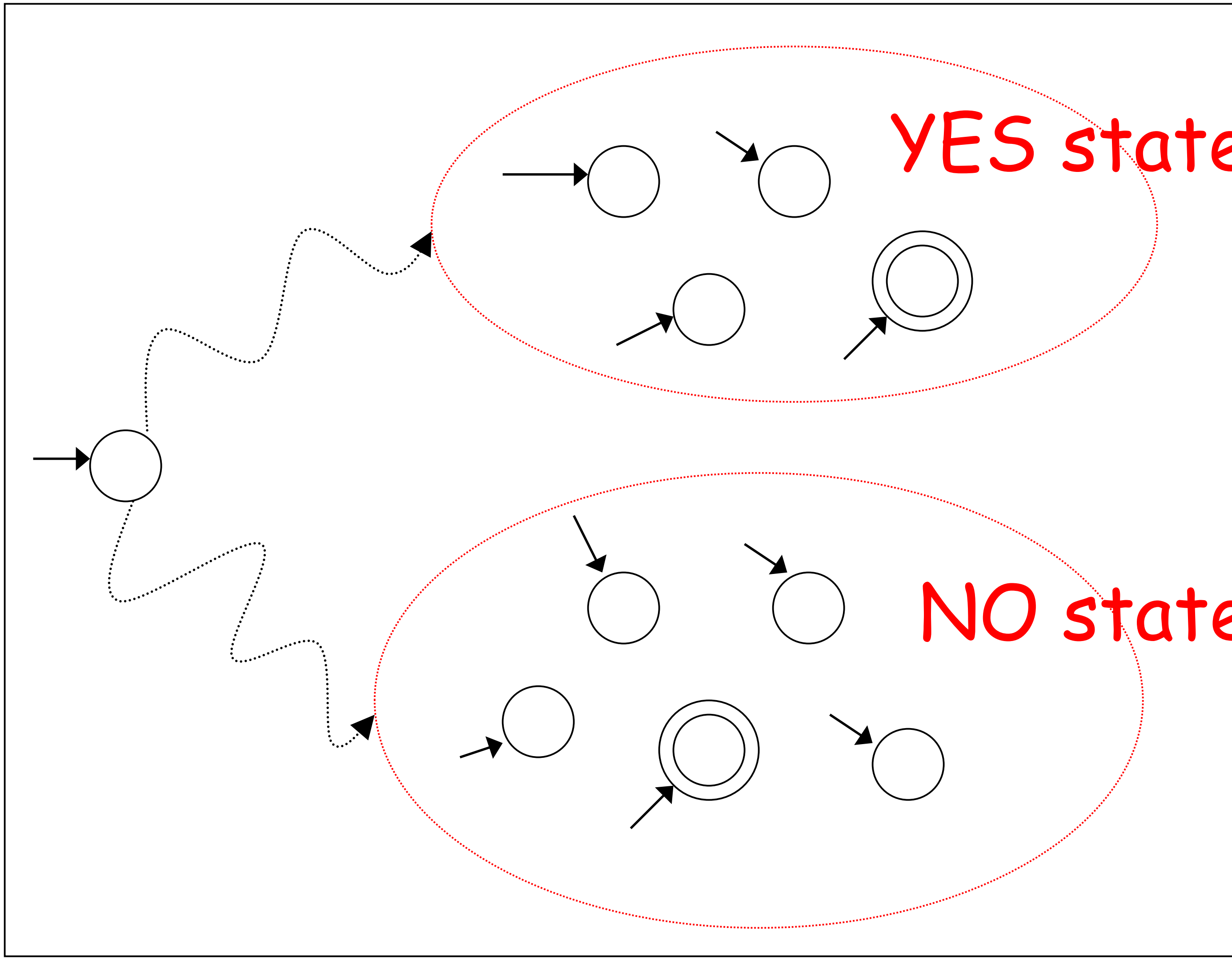


The machine that decides (solves) a problem:

- If the answer is YES
  then halts in a yes state
- If the answer is NO
  then halts in a no state

These states do not have to be final states

Turing Machine that decides a problem



YES and NO states are halting states

Undecidability:

The Post Correspondence Problem is a well-known undecidable problem, meaning there's no algorithm that can determine, for all possible instances of the problem, whether a solution exists or not.

- In many instances it is cumbersome to work with the halting problem directly, and it is convenient to establish some intermediate results that bridge the gap between the halting problem and other problems.
- These intermediate results follow from the undecidability of the halting problem, is the **Post correspondence problem**.

The Post correspondence problem can be stated as follows. Given two sequences of n strings on some alphabet Σ, say

$$A = w_1, w_2, \ldots w_n$$

and

$$B = v_1, v_2, \ldots, v_n,$$

we say that there exists a Post correspondence solution (PC-solution) for pair (A,B) if there is a nonempty sequence of integers $i, j, \ldots, k$, such that

$$w_i w_j \ldots w_k = v_i vj \ldots v_k.$$

The Post correspondence problem is to devise an algorithm that will tell us, for any (A, B), whether or not there exists a PC-solution.

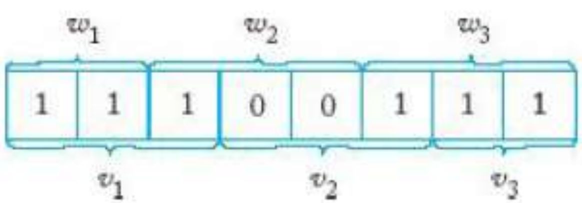
Example 12.5

Let $\Sigma = \{0,1\}$ and take A and B as

$$w_1 = 11, w_2 = 100, w_3 = 111,$$
$$v_1 = 111, v_2 = 001, v_3 = 11.$$

For this case, there exists a PC-solution as Figure 12.7 shows.

Figure 12.7



If we take

$$w_1 = 00, w_2 = 001, w_3 = 1000,$$
$$v_1 = 0, v_2 = 11, v_3 = 011,$$

there cannot be any PC-solution simply because any string composed of elements of A will be longer than the corresponding string from B.