PART III

Turing Machines and Effective Computability

PART III Chapter 1

Turing Machines

Turing machines

- the most powerful automata (> FAs and PDAs)
- invented by Turing in 1936
- can compute any function normally considered computable
- Turing-Church Theses:
  - Anything (function, problem, set etc.) that is (though to be) computable is computable by a Turing machine (i.e., Turing-computable).
- Other equivalent formalisms:
  - post systems (string rewriting system)
  - PSG (phrase structure grammars) : on strings
  - μ-recursive function : on numbers
  - λ-calculus, combinatory logic: on λ-term
  - C, BASIC, PASCAL, JAVA languages,… : on strings

Informal description of a Turing machine

1. Finite automata (DFAs, NFAs, etc.):
   - limited input tape: one-way, read-only
   - no working-memory
   - finite-control store (program)
2. PDAs:
   - limited input tape: one-way, read-only
   - one additional stack as working memory
   - finite-control store (program)
3. Turing machines (TMs):
   - a semi-infinite tape storing input and supplying additional working storage.
   - finite control store (program)
   - can read/write and two-way(move left and right) depending on the program state and input symbol scanned.

Turing machines and LBAs

4. Linear-bounded automata (LBA): special TMs

- **the input tape is of the same size as the input length (i.e., no additional memory supplied except those used to store the input)**
- **can read/write and move left/right depending on the program state and input symbol scanned.**

- **Primitive instructions of a TM (like +,-,*, etc in C or BASIC):**

1. L, R // moving the tape head left or right

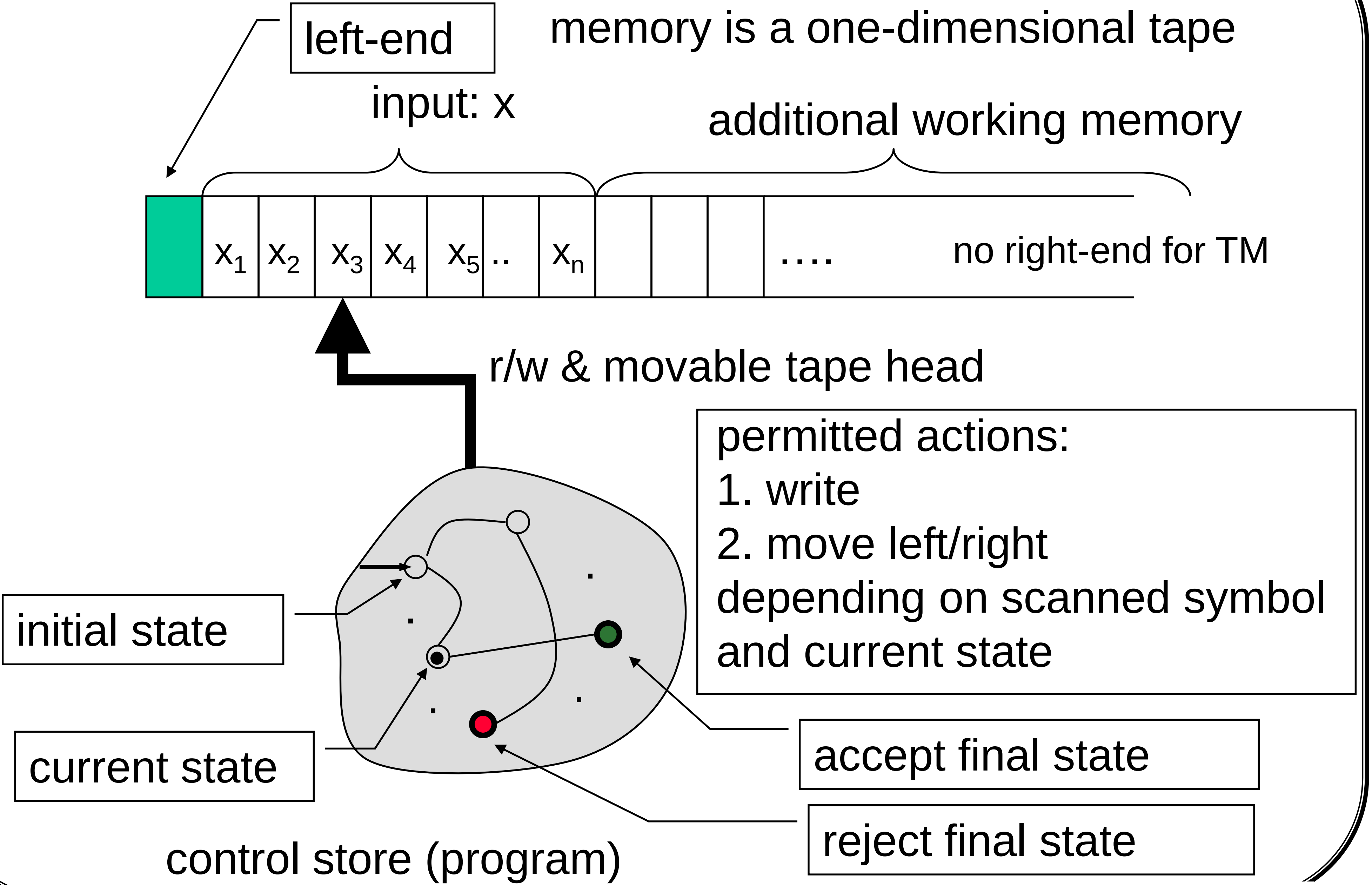
2. a $\in \Gamma$, // write the symbol a $\in \Gamma$ on the current scanned position

depending on the precondition:

1. current state and

2. current scanned symbol of the tape head

The model of a Turing machine



left-end

memory is a one-dimensional tape

input: x

additional working memory

no right-end for TM

r/w & movable tape head

permitted actions:
1. write
2. move left/right
depending on scanned symbol and current state

initial state

current state

accept final state

reject final state

control store (program)

The structure of a TM instruction:

- **An instruction of a TM is a tuple:**

$$(q,\quad a,\quad p,\quad d) \in Q \text{ x } \Gamma \text{ x } Q \text{ x } (\Gamma \text{ U } \{L,R\})$$

where

- **q is the current state**
- **a is the symbol scanned by the tape head**
- **(q,a) define a precondition that the machine may encounter**
- **(p,d) specify the actions to be done by the TM once the machine is in a condition matching the precondition (i.e., the symbol scanned by the tape head is ‘a’ and the machine is at state q)**
- **p is the next state that the TM will enter**
- **d is the action to be performed:**
  - ✪**d = b $\in \Gamma$ means “write the symbol b to the tape cell currently scanned by the tape head”.**
  - ✪**d = R (or L) means “move the tape head one tape cell in the right (or left, respectively) direction.**

- **A Deterministic TM program δ is simply a set of TM instructions (or more formally a function: δ: Q x Γ --> Qx (Γ U{L,R}))**

Formal Definition of a standard TM (STM)

- **A deterministic 1-tape Turing machine (STM) is a 9-tuple**

  M = $(Q, \Sigma, \Gamma, [, \square, \delta, s, t, r)$ where

  - **Q : is a finite set of (program) states with a role like labels in traditional programs**
  - **Γ : tape alphabet**
  - **$\Sigma \subset \Gamma$: input alphabet**
  - **$[\in \Gamma - \Sigma$: The left end-of-tape mark**
  - **$\square \in \Gamma - \Sigma$ is the blank tape symbol**
  - **$s \in Q$: initial state**
  - **$t \in Q$: the accept state**
  - **$r \neq t \in Q$: the reject state and**
  - **δ: Q - {t,r} --> Qx(Γ U {L,R}) is a *total* transition function with the restriction:** if $\delta(p, [\,) = (q, d)$ then d = R. i.e., the STM cannot write any symbol at left-end and never move off the tape to the left.

Configurations and acceptances

- Issue: h/w to define configurations like those defined at FAs and PDAs ?
- At any time t_0 the TM M's tape contains a semi-infinite string of the form

$$\mathrm{Tape}(t_0) = [\; y_1 y_2 \ldots y_m \,\square\,\square\,\square\,\square \ldots.. \quad (y_m \neq \square)$$

- Let $\square^{\omega}$ denotes the semi-infinite string:

$$\square\,\square\,\square\,\square\,\square \ldots..$$

Note: Although the tape is an infinite string, it has a finite canonical representation: $y\square^{\omega}$ where $y = [\; y_1 \ldots y_m$ (with $y_m \neq \square$)

A configuration of the TM M is a global state giving a snapshot of all relevant info about M's computation at some instance in time.

Formal definition of a configuration

Def: a cfg of a STM M is an element of

$$CF_M =_{def} Q \text{ x } \{[y \Box^{\omega} \mid y \in (\Gamma-\{[\})^*\} \text{ x } N \qquad // \ N = \{0,1,2,\ldots\} \ //$$

When the machine M is at cfg (p, z, n) , it means M is

1. at state p

2. Tape head is pointing to position n and

3. the input tape content is z.

Obviously cfg gives us sufficient information to continue the execution of the machine.

Def: 1. [Initial configuration:] Given an input x and a STM M,

the initial configuration of M on input x is the triple:

$$(s, [x \Box^{\omega}, 0)$$

2. If cfg1 = (p, y, n), then cfg1 is an accept configuration if p = t (the accept configuration), and cfg1 is an reject cfg if p = r (the reject cfg). cfg1 is a halting cfg if it is an accept or reject cfg.

One-step and multi-step TM computations

- one-step Turing computation ($|\text{--}_M$) is defined as follows:
- $|\text{--}_M \subset CF_M^2$ s.t.

0. $(p,z,n)\ |\text{--}_M\ (q,s^n_b(z), n)$ if $\delta(p,z_n) = (q, b)$ where $b \in \Gamma$
1. $(p,z,n)\ |\text{--}_M\ (q,z, n-1)$ if $\delta(p,z_n) = (q, L)$
2. $(p,z,n)\ |\text{--}_M\ (q,z, n+1)$ if $\delta(p,z_n) = (q, R)$

  - where $s^n_b(z)$ is the resulting string with the n-th symbol of z replaced by 'b'.
  - ex: $s^4_b([\ \text{baa}\underline{\text{a}}\text{cabc}\ldots) = [\ \text{baa}\underline{\text{b}}\text{cabc}\ldots$

- $|\text{--}_M$ is defined to be the set of all pairs of configurations each satisfying one of the above three rules.

Notes: 1. if $C=(p,z,n)\ |\text{--}_M\ (q,y,m)$ then $n \geq 0$ and $m \geq 0$ (why?)

2. $|\text{--}_M$ is a function [from nonhalting cfgs to cfgs] (i.e., if $C\ |\text{--}_M\ D$ & $C\ |\text{--}_M\ E$ then $D=E$).

3. define $|\text{--}^n_M$ and $|\text{--}^*_M$ (ref. and tran. closure of $|\text{--}_M$) as usual.

accepting and rejecting of TM on inputs

- **$x \in \Sigma$ is said to be accepted by a STM M if**

  $$icfg_M(x) =_{def} (s, [x\square^{\omega}, 0) \vdash^*_M (t,y,n) \quad \text{for some y and n}$$

  - **I.e, there is a finite computation**

    $$(s, [x\square^{\omega}, 0) = C_0 \vdash_M C_1 \vdash_M \ldots. \vdash_M C_k = (t,y,n)$$

  starting from the initial configuration and ending at an accept configuration.

- **x is said to be rejected by a STM M if**

  $$(s, [x\square^{\omega}, 0) \vdash^*_M (r,y,n) \quad \text{for some y and n}$$

  - **I.e, there is a finite computation**
  - $$(s, [x\square^{\omega}, 0) = C_0 \vdash_M C_1 \vdash_M \ldots. \vdash_M C_k = (t,y,n)$$
  - **starting from the initial configuration and ending at a reject configuration.**

Notes: 1. It is impossible that x is both accepted and rejected by a STM. (why ?)

2. It is possible that x is neither accepted nor rejected. (why ?)

Languages accepted by a STM

Def:

1. M is said to *halt* on input x if either M accepts x or rejects x.
2. M is said to *loop* on x if it does not halt on x.
3. A TM is said to be *total* if it halts on all inputs.
4. The language accepted by a TM M,

   $L(M) =_{def} \{x \text{ in } \Sigma^* \mid x \text{ is accepted by } M, \text{ i.e., } (s, [x\square^{\omega}, 0) \mid\!\!-\!\!-^*_M (t, -,-) \}$

5. If L = L(M) for some STM M

   ==> L is said to be *recursively enumerable (r.e.)*

6. If L = L(M) for some total *STM M*

   ==> L is said to be *recursive*

7. If $\sim L =_{def} \Sigma^* - L = L(M)$ for some STM M (or total STM M)

   ==> L is said to be *Co-r.e. (or Co-recursive, respectively)*

Some examples

Ex1: Find a STM to accept $L_1 = \{ w \# w \mid w \in \{a,b\}^* \}$

note: L_1 is not CFL.

The STM has tape alphabet $\Gamma = \{a, b, \#, -, □, [\}$ and behaves as follows:
on input $z = w \# w \in \{a,b,\#\}^*$

```
1.  if z is not of the form {a,b}* # {a,b}* => goto reject
2.  move left until '[' is encountered and  in that case move right
3. while I/P = '-' move right;
4. if I/P = 'a' then
     4.1  write '-'; move right until # is encountered; Move right;
     4.2  while I/P = '-' move right
     4.3  case (I/P) of {  'a' : (write '-'; goto 2);     o/w: goto reject  }
5. if I/p = 'b' then … // like 4.1~ 4.3
6. If I/P = '#' then      // All symbols left to # have been compared
     6.1 move right
     6.2  while I/P = '-" move right
     6.3  case (I/P) of {'□' : goto Accept;     o/w: go to Reject  }
```

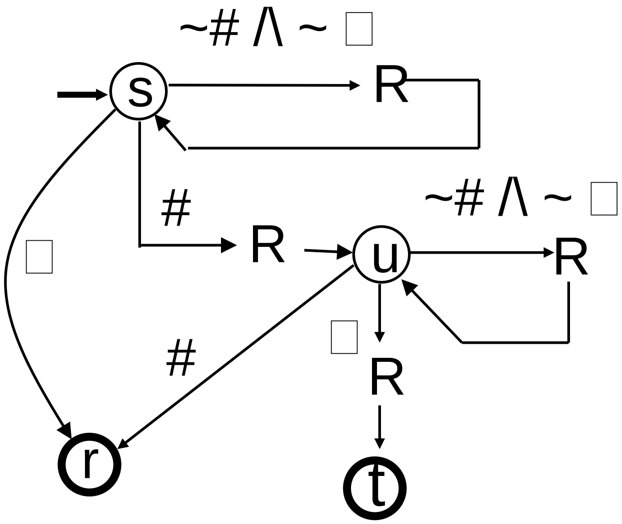
More detail of the STM

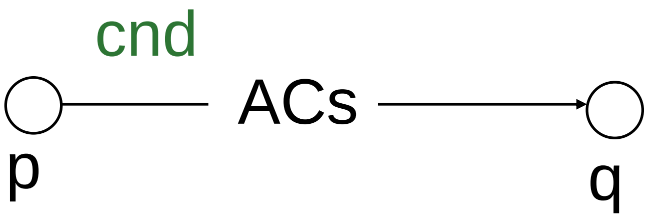
Step 1 can be accomplished as follows:

```
1.1  while (~# /\ ~ □)   R; // or equivalently, while (a \/ b\/[) R
     if □ => reject   // no # found on the input
     if # => R;
1.2  While ( ~# /\ ~ □ ) R;
     if □ => goto accept [or goto 2 if regarded as a subroutine]
     if #   => goto Reject;     // more than one #s found
```

Step 1 requires only two states:

Graphical representation of a TM





means:

if (state = p) /\ (cnd true for i/p)

then 1. perform ACs and 2. go to q

ACs can be primitive ones: R, L, a,…

or another subroutine TM M_1.

Ex: the arc from s to s implies the existence of 4 instructions:

(s, a, s, R), (s,b,s,R), (s, [,s,R), and (s,-, s,R)

Tabular form of a STM

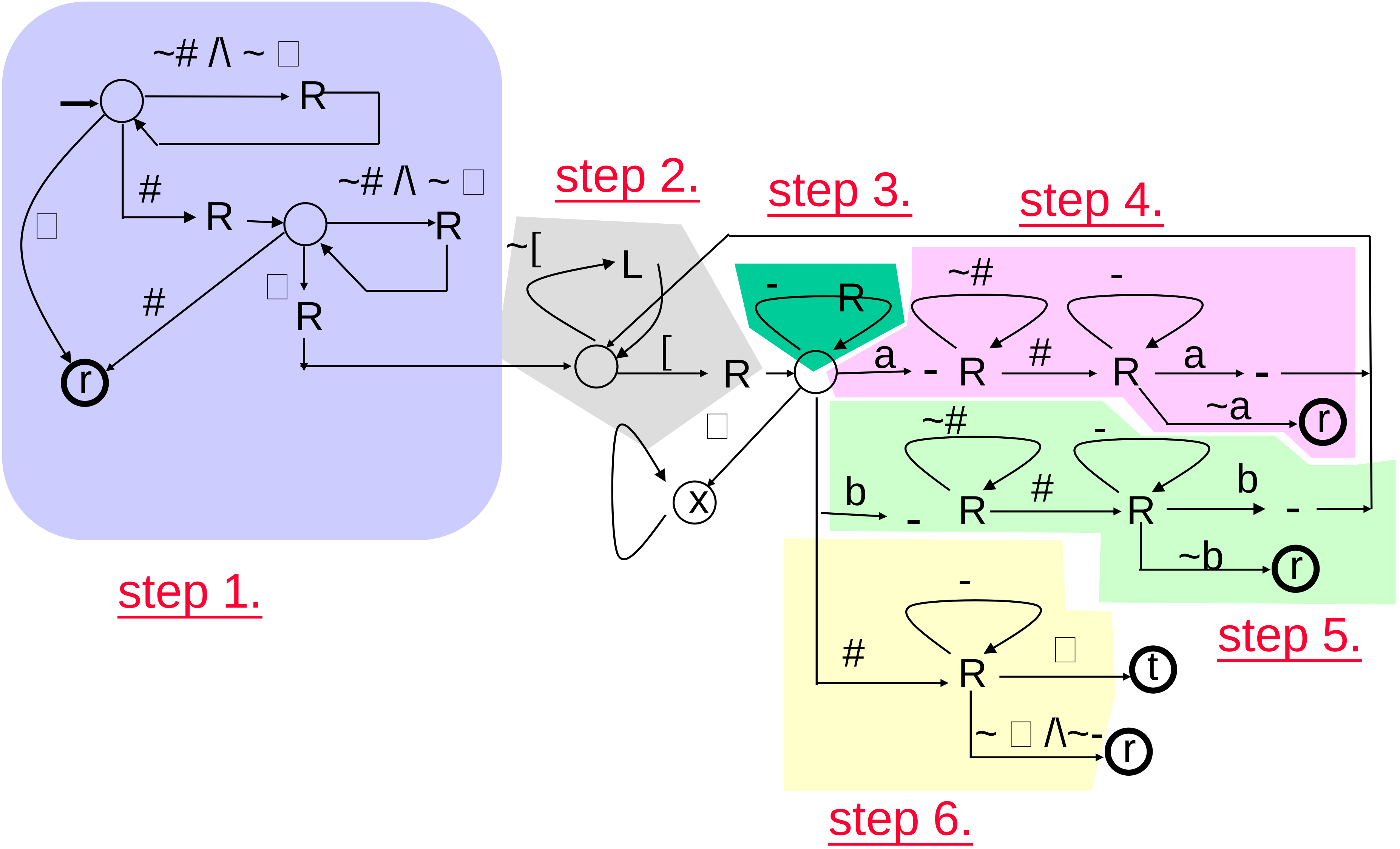
- Translation of the graphical form to tabular form of a STM

δ Γ Q	[	a	b	#	-	□
>s	s,R	s,R	s,R	u,R	x	r,x
u	x	u,R	u,R	r,x	x	t, □
tF	halt	halt	halt	halt	halt	halt
rF	halt	halt	halt	halt	halt	halt

X means don't care

The rows for t & r indeed need not be listed!!

The complete STM accepting L_1



R.e. and recursive languages

Recall the following definitions:

1. M is said to *halt* on input x if either M accepts x or rejects x.

2. M is said to *loop* on x if it does not halt on x.

3. A TM is said to be *total* if it halts on all inputs.

4. The language accepted by a TM M,

L(M) $=_{\text{def}}$ {x ∈Σ* | x is accepted by M, i.e., (s, [x □$^{\omega}$,0) |--$^{*}_{M}$ (t, -,-) }

5. If L = L(M) for some STM M

==> L is said to be *recursively enumerable (r.e.)*

6. If L = L(M) for some total *STM M*

==> L is said to be *recursive*

7. If ~ L$=_{\text{def}}$ Σ* - L = L(M) for some STM M (or total STM M)

==> L is said to be *Co-r.e. (or Co-recursive, respectively)*

Recursive languages are closed under complement

Theorem 1: **Recursive languages are closed under complement.** **(i.e., If L is recursive, then ~L = Σ* - L is recursive.)**

pf: Suppose L is recursive. Then L = L(M) for some total TM M.

Now let M* be the machine M with accept and reject states switched.

Now for any input x,

▯ $x \notin {\sim}L \Rightarrow x \in L(M) \Rightarrow icfg_M(x) \mid\!\text{-}_M^{*} (t,-,-) \Rightarrow$

▯ $icfg_{M^*}(x) \mid\!\text{-}_{M^*}^{*} (r^*,-,-) \Rightarrow x \notin L(M^*)$.

▯ $x \in {\sim}L \Rightarrow x \notin L(M) \Rightarrow icfg_M(x) \mid\!\text{-}_M^{*} (r,-,-) \Rightarrow$

▯ $icfg_{M^*}(x) \mid\!\text{-}_{M^*}^{*} (t^*,-,-) \Rightarrow x \in L(M^*)$.

Hence ~L = L(M*) and is recursive.

Note.The same argument cannot be applied to r.e. languages. (why?)

Exercise: Are recursive sets closed under union, intersection, concatenation and/or Kleene's operation ?

Some more termonology

Decidability and semidecidability

Solvability and semisolvabilty

- P : a statement about strings (or a property of strings)
- A: a set of strings
- Q : a (decision) Problem.

We say that

1. P is decidable <==> { x | P(x) is true } is recursive
2. A is recursive <==> “x $\in$ A” is decidable.
3. P is semidecidable <==> { x | P(x) is true } is r.e.
4. A is r.e. <==> “x $\in$ A” is semidecidable.
5. Q is solvable <=> Rep(Q) $=_{def}$ {“P” | P is a positive instance of Q } is recursive.
6. Q is semisolvale <==> Rep(Q) is r.e..

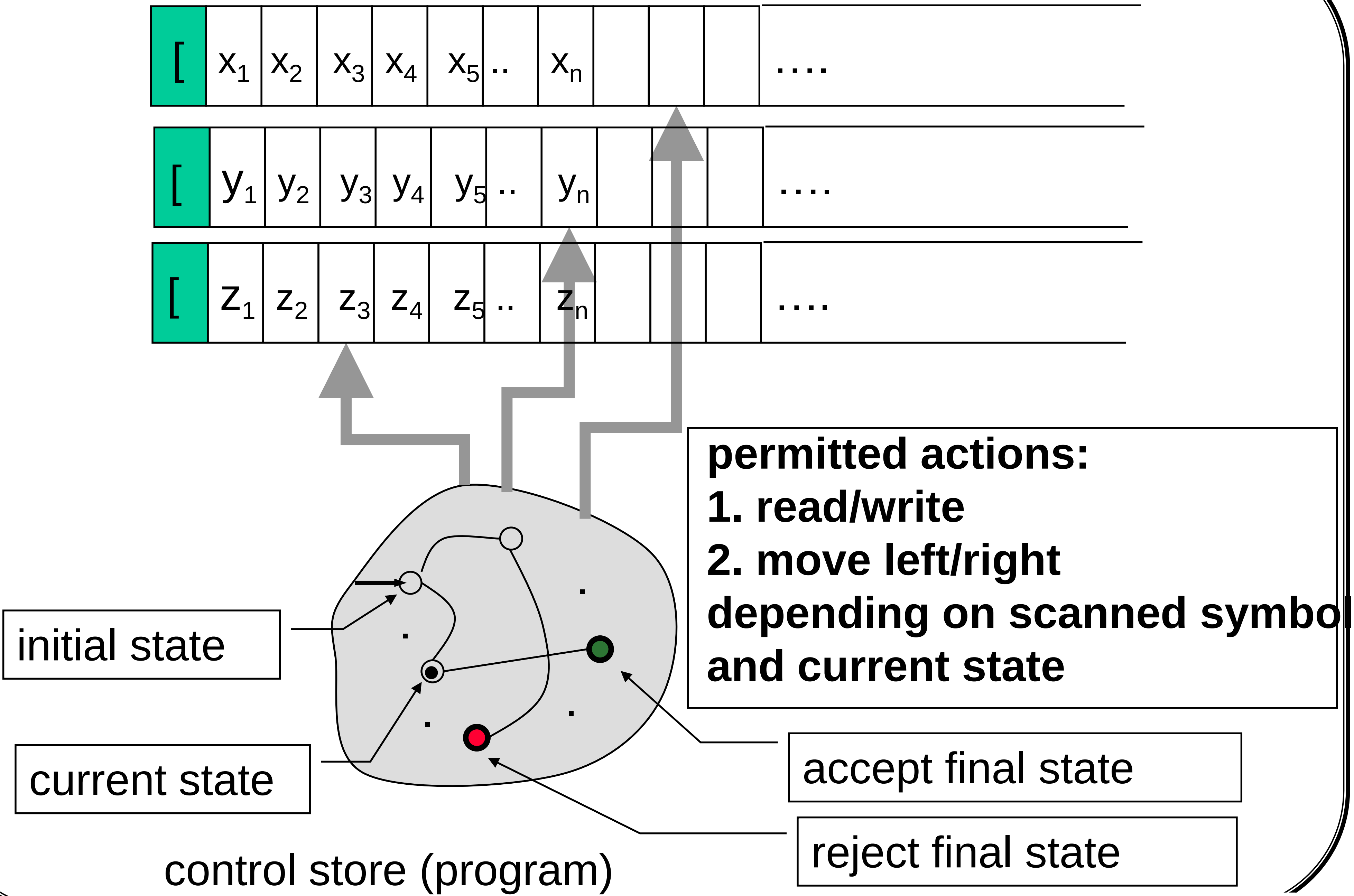
multi-tape TM

- **A k-tape ($k\geq 1$) Turing machine is a 9-tuple**

  $M = (Q,\Sigma,\Gamma, [, □□, \delta, s, t,r)$ where

  - **Q : is a finite set of (program) states like labels in traditional programs**
  - **Γ : tape alphabet**
  - **$\Sigma \subset \Gamma$: input alphabet**
  - **$[\in \Gamma - \Sigma$: The left end-of-tape mark**
  - **□□ $\in \Gamma - \Sigma$ is the blank tape symbol**
  - **$s \in Q$: initial state**
  - **$t \in Q$: the accept state**
  - **$r \neq t \in Q$: the reject state and**
  - **δ: $(Q - \{t,r\}) \times \Gamma^k \dashrightarrow Q \times (\Gamma \cup \{L,R\})^k$ is a *total* transition function with the restriction: if $\delta(p, x_1,\ldots,x_k) = (q, y_1,\ldots,y_k)$ then if $xj = [\Longrightarrow y_j = [$ or R. i.e., the TM cannot overwrite other symbol at left-end and never move off the tape to the left.**

3-tape Turing machine



permitted actions:
1. read/write
2. move left/right
depending on scanned symbols
and current state

initial state

current state

accept final state

reject final state

control store (program)

Equivalence of STMs and Multi-tape TMs

- M = (Q,Σ,Γ, [, □, δ, s, t,r): a k-tape TMs

==> M can be simulated by a k-track STM:

M' = (Q',Σ,Γ', [,□,δ', s, t',r') where

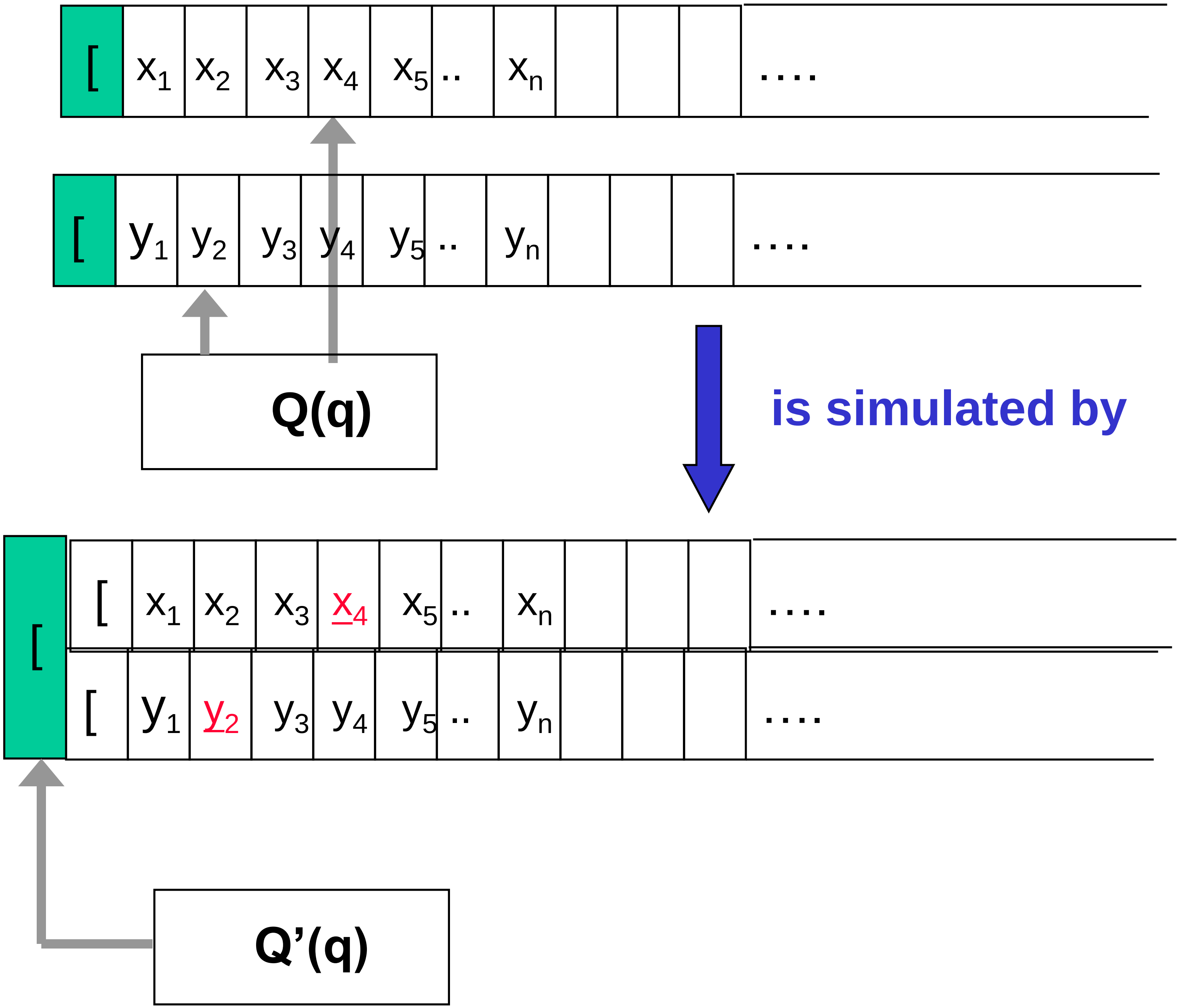
□ $\Gamma' = \Gamma \cup (\Gamma \cup \underline{\Gamma})^k$ where $\underline{\Gamma} = \{ \underline{a} \mid a \in \Gamma\}$.

- M' = init • M'' where the task of Init is to convert initial input tape content : $[x_1x_2...x_n$ □$□^\omega$ into

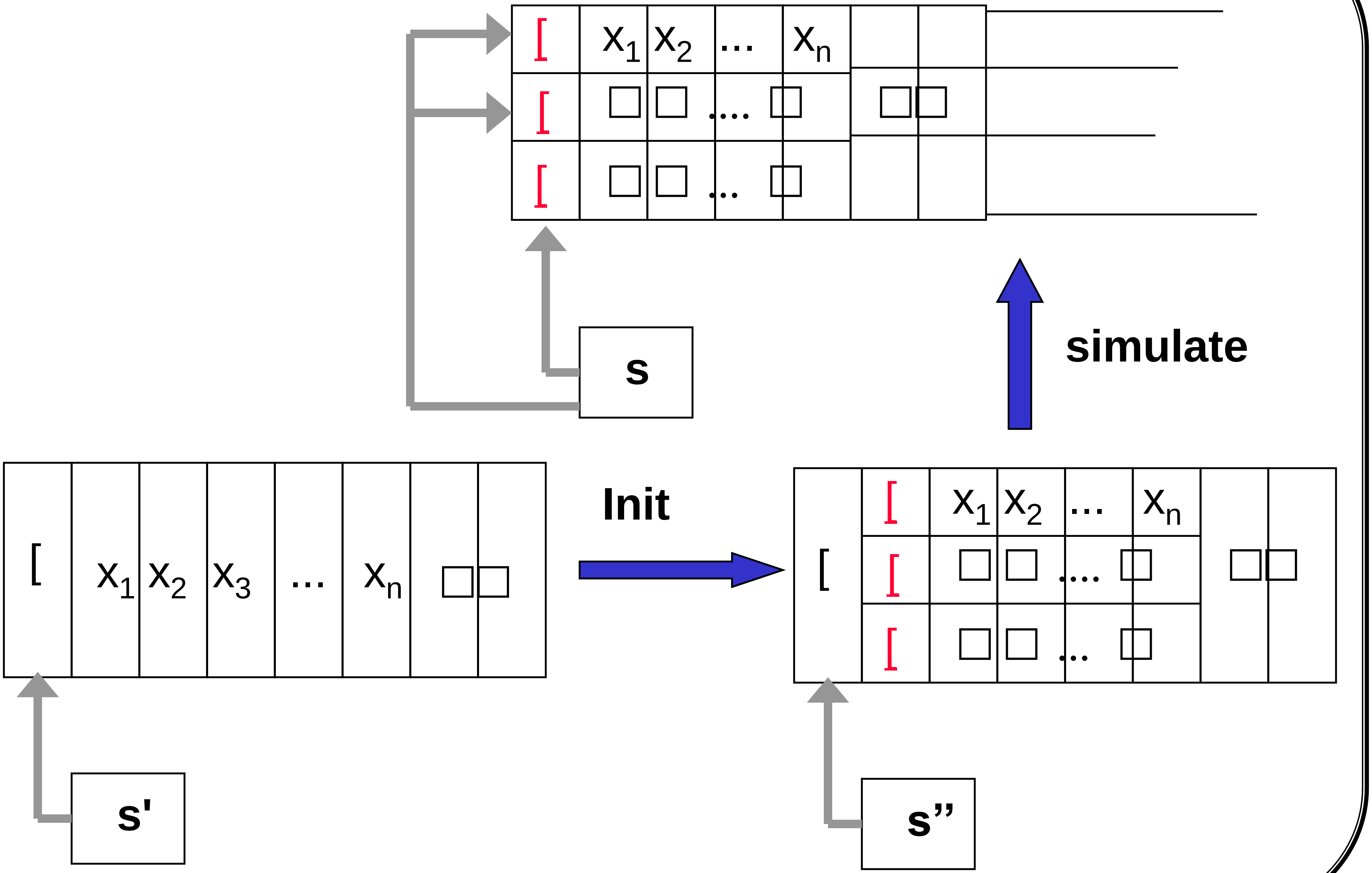
[	[	x_1	x_2	…	x_n	□	□
	[	□	□	....	□		
	[	□	□	...	□		

and then go to the initial state s'' of M'' to start simulation of M.

- Each state q of M is simulated by a submachine M_q of M'' as follows:


[x_1 x_2 x_3 x_4 x_5 .. x_n ....
[y_1 y_2 y_3 y_4 y_5 .. y_n ....
Q(q)
[
[x_1 x_2 x_3 $\underline{x}_4$ x_5 .. x_n ....
[y_1 $\underline{y}_2$ y_3 y_4 y_5 .. y_n ....
Q'(q)

is simulated by


[
x1
x2
...
xn
□□
s
simulate
Init
[
x1
x2
x3
...
xn
□□
s'
[
x1
x2
...
xn
□□
s''

How does M'' simulate M ?

- let $(q, x^1, p_1, y^1),\ldots,(q,x^m,p_m,y^m)$ be the set of all instructions (starting from state q) having the form $\delta(q,x^i) = (p_i,y^i)$. where $x^i,y^i = (x^i_1,x^i_2,\ldots,x^i_k), (y^i_1,y^i_2,\ldots,y^i_k)$. Then M_q behaves as follows:

0. [terminate?] if q = t then accept; if q = r then reject.

1. [determine what symbols are scanned by tape heads]

    for j = 1 to k do { // determine symbol scanned by jth head

    move right until the symbol at the jth track is underlined,

    remember which symbol is underlined (say $\underline{a}_j$) in the control store and then move to left end.}

2. [perform action:$\delta(q, a_1,\ldots,a_k) = (p, b_1,\ldots,b_k)$ for each tape head]

    for j= 1 to k do{ // perform b_j at the jth tape

case1. $b_j = b\in\Gamma$==>MR until $\underline{a}_j$; replace symbol $\underline{a}_j$ at jth track by $\underline{b}_j$

case2. b_j = R ==>MR unitl $\underline{a}_j$, replace $\underline{a}_j$ by a_j and underline its right neighbor symbol.

case3: b_j = L. Similar to case 2. Finally move to left end. }

3. [go to next state] go to start state of M_p to simulate M at state p.

Running time analysis

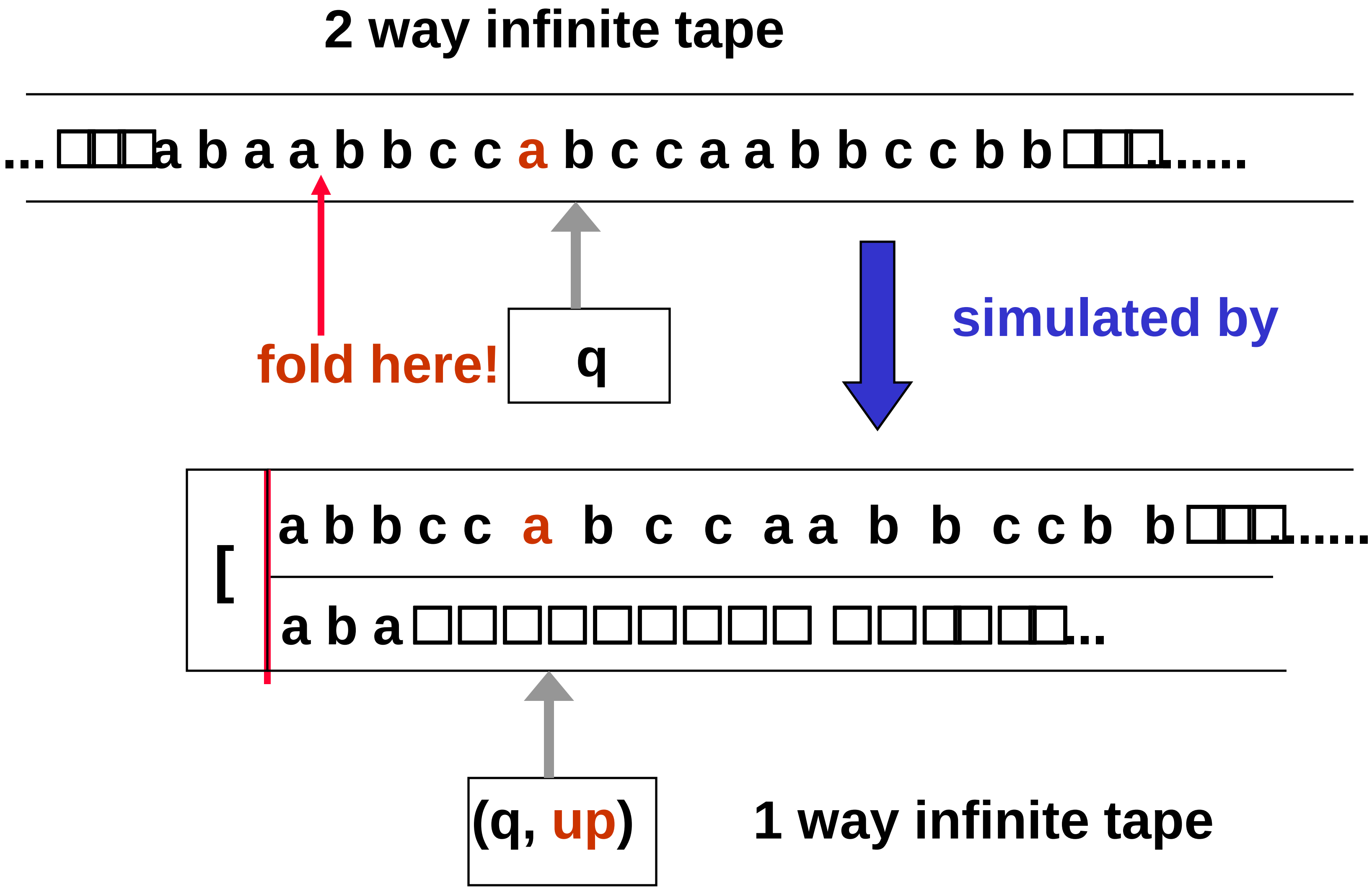
- How many steps of M’’ are needed to simulate one step of execution of M ?
- Sol:
- Assume the running time of M on input x of length n is f(n).
  - step 1 requires time : O(k x 2 f(n))
  - Step 2 requires time: O(k x 2 f(n))
  - Step 3 requires O(1) time
  - => Each step requires time O(4k x f(n)).
  - and total time required to simulate M = f(n) x O(4k f(n))
  - = O $(f(n)^2)$.

Turing machine with 2 way infinite tape

- **A 2way single tape Turing machine is a 8-tuple**

  M = (Q,Σ,Γ, [, □, δ, s, t,r) where

  - **Q : is a finite set of (program) states like labels in traditional programs**
  - **Γ : tape alphabet**
  - **$\Sigma \subset \Gamma$: input alphabet**
  - **$[\in \Gamma - \Sigma$: The left end-of-tape mark (no longer needed!!)**
  - **$\Box \in \Gamma - \Sigma$ is the blank tape symbol**
  - **$s \in Q$: initial state**
  - **$t \in Q$: the accept state**
  - **$r \neq t \in Q$: the reject state and**
  - **δ: (Q - {t,r})x Γ --> Qx(Γ∪{L,R}) is a *total* transition function.**


2 way infinite tape
... □□□a b a a b b c c a b c c a a b b c c b b □□□......
fold here!
q
simulated by
[
a b b c c a b c c a a b b c c b b □□□......
a b a □□□□□□□□□ □□□□□□...
(q, up)
1 way infinite tape

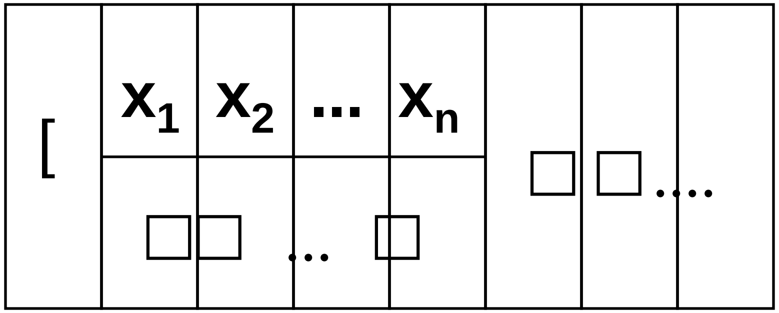
Equivalence of STMs and 2way TMs

- $M = (Q,\Sigma,\Gamma,\square,\delta, s, t,r)$: a 2way TM

==> M can be simulated by a 2-track STM:

$M' = (Q',\Sigma,\Gamma', [, \square \square,\delta', s, t',r')$ where

- $Q' = Q \cup (Q\times\{u,d\}) \cup \{...\}$,
- $\Gamma' = \Gamma \cup \Gamma^2 \cup \{ [\}$,
- M' = init • M'' where the task of Init is to convert initial input tape content : □φ $x_1x_2...x_n$,□φ into



and then go to the initial state s'' of M'' to start simulation of M.

- Each instruction of M is simulated by one or two instructions of M'' as follows:

How to simulate 2way tape TM using 1way tape TM

Let $\delta(q,x) = (p, y)$ be an instruction of M then:

case 1: $y \in \Gamma$

- ==> $\delta''((q,u), (x,z)) =((p,u),(y,z))$ and
- $\delta''((q,d),(z,x)) = ((p,d), (z,y))$ for all $z \in \Gamma$

case2 : y = R.

- ==> $\delta''((q,u), (x,z)) =((p,u),R)$ and $\delta''((q,d),(z,x)) = ((p,d), L)$
- for all $z \in \Gamma$.

case 3: y = L.

- ==> $\delta''((q,u), (x,z)) =((p,u),L)$ and $\delta'((q,d),(z,x)) = ((p,d), R)$
- for all $z \in \Gamma$.

additional conditions [left end==> change direction] :

- : $\delta''((q,u), [) = ((q,d),R)$, $\delta''((q,d),[)=((q,u),R)$ for all $q \notin \{t,r\}$.

Properties of r.e. languages

- **Theorem: If both L and ~L are r.e., then L (and ~L) is recursive.**

Pf: Suppose L=L(M_1) and ~L = L(M_2) for two STM M_1 and M_2.

Now construct a new 2 -tape TM M as follows:

```
on input: x
1. copy x from tape 1 to tape 2.    // COPY
2.  Repeat { execute 1 step of $M_1$ on tape 1;
                      execute 1 step of $M_2$ on tape 2 }
     until either $M_1$ accept or $M_2$ accept.
3. If $M_1$ accept then [M] accept
   If $M_2$ accept then [M] reject.  // 2+3 = $M_1$ || $M_2$ defined later
```

So if x $\in$ L => M_1 accept x => M accept

if x $\notin$L => M_2 accept ==> M reject.

Hence M is total and L =L(M) and L is recursive.

Interleaved execution of two TMs

- $M_1 = (Q_1,\Sigma,\Gamma, [,\Box, \delta_1, s_1, t_1,r_1)$; $M_2 = (Q_2,\Sigma,\Gamma, [,\Box, \delta_2, s_2, t_2,r_2)$ where

  $\delta_1: Q_1 \text{ x } \Gamma \text{ --> } Q_1 \text{ x } (\Gamma \text{ U}\{L,R\})$; $\delta_2: Q_2 \text{ x } \Gamma \text{ --> } Q_2 \text{ x } \Gamma \text{ U}(\{L,R\})$;

==> $M =_{def} M_1 \,||\, M_2 = (Q_1\text{x}Q_2\text{x}\{1,2\}, \Sigma,\Gamma, [,\Box, \delta,s,T,R)$where

1. $\delta: Q_1\text{x}Q_2\text{x}\{1,2\}\text{x } \Gamma^{\,2} \text{ --> } (Q_1\text{x}Q_2\text{x}\{1,2\}) \text{ x } (\Gamma \text{ U}\{L,R\})^2$ is given by
   - □ Let $\delta_1(q_1,a) = (p_1,a')$ and $\delta_2(q_2,b) = (p_2,b')$ then
   - □ $\delta((q_1,q_2,1),(a,b)) = ((p_1,q_2,2), a',b)$ and
   - □ $\delta((q_1,q_2,2),(a,b)) = ((q_1,p_2,1), a,b')$
2. M has initial state $s = (s_1,s_2,1)$.
3. M has halting states T from M_1: $\{(t_1,q_2,1) \mid q_2 \in Q_2\} \text{ U } \{(r_1,q_2,1) \mid q_2 \in Q_2\}$ and halting states R from M_2: $\{(q_1,t_2,2) \mid q_1 \in Q_1\} \text{ U } \{(q_1,r_2,2) \mid q_1 \in Q_1\}$.
4. By suitably designating halting states of M as accept or reject states, we can construct machine accepting languages that are boolean combination of $L(M_1)$ and $L(M_2)$. Ex: $T = \{(t_1,q_2,1) \mid q_2 \in Q_2\}$ and $R = \{(q_1,t_2,2) \mid q_1 \in Q_1\}$ in previous example.

A programming Language for TMs and Universal TM

- Proposed Computation models
  - TMs (DTM, NDTM, RATM, multi-tape, 2way, multi Dimensional, multi-head, and their combinations,...)
  - Grammars
  - u-recursive function,
  - λ-calculus
  - Counter Machine
  - 2STACK machine
  - Post system,...
- All can be shown to have the same computation power
- Church-Turing Thesis:
  - A language or function is computable iff it is Turing-computable (i.e., can be computed by a total TM).
  - An algorithm is one that can be carried out by a total TM.

Universal Turing machine

- TMs considered so far are dedicated in the sense that each of their control store is hard-wired to solve one particular problem
    - e.g., TMs for +, x, copy,...
- Problem: Is there any TM that can compute what all TMs can compute ?

Yes!! we call it **universal TM (UTM),** which is nothing special but a general-purpose TM.

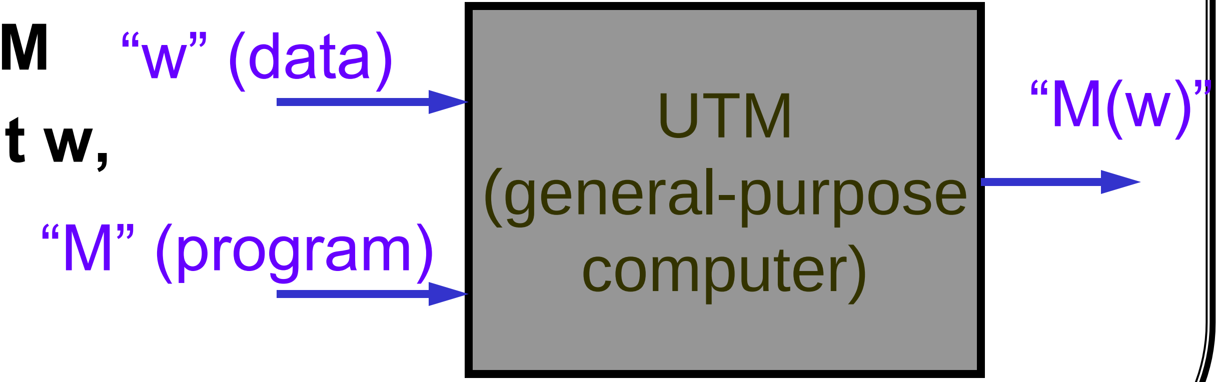
- UTM is a TM simulator,i.e.,

given a spec “M” of a TM M

and a desc “w”of an input w,

UTM can simulate the

execution of M on w.

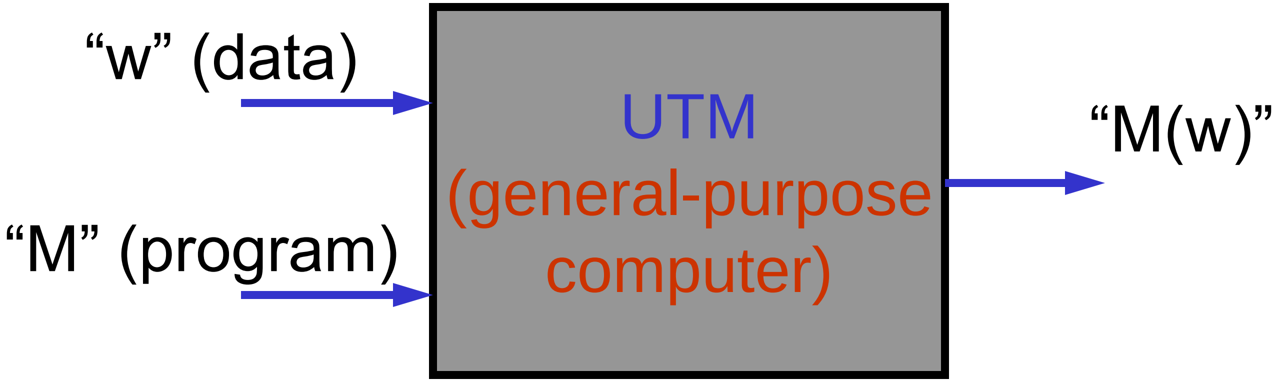


“w” (data)

“M” (program)

UTM (general-purpose computer)

“M(w)”


"w" (data)
"M" (program)
UTM
(general-purpose
computer)
"M(w)"

TL : a programming language for TMs

- M = (Q,Σ,Γ, [, □□, δ, s, t,r) : a STM.
- TM M can be described by a string (i.e., a TL-program) as follows:
- Tape symbols of M are encoded by strings from a{0,1}*
  - blank(□) ==> a0 left-end [==> a1
  - R ==> a00 L ==> a01
  - others => a10,a11,a000,a001,….
- State symbols of M are encoded by strings from q{0,1}*
  - start state s ==> q0;
  - accept state t ==> q1, reject state r ==> q00;
  - other states => q01,q10,q11, q111,…
- For b ∊ ΓU{R,L} U Q, we use e(b) ∊ a{0,1}* U q{0,1}* to denote the encoding of b.

An example

- M = (Q,Σ,Γ, [, □ □, δ, s, t,r) where
  - Q = {s, g, r,t }, Γ = { [, a, □ □} and
  - δ = { (s, a, g, □), (s, □,t, □), (s, [, s, R),
  - (g, a, s, a), (g, U, s, R), (g, [, r, R) }

==>

- Suppose state and tape symbols are represented in TL as follows:
- s => q0 ; t => q1; r => q00; g => q01
- □ => a0; [=> a1; R => a00; L => a01;
- a => a10
- Hence a string: ‘[aa□a’ ∈ Γ* can be encoded in TL as
- e([aa□a) = “[aa□a” $=_{def}$ a1a10a10a0a10

Describe a TM by TL

- Let δ = { α_j | α_j = (p_j, a_j, q_j, b_j); j = 1 ..n} be the set of all instructions. ==>
- M can be encoded in TL as a string
  - e(M) = “M”∈ {q,a,0,1,[, (,), ‘ }*
  - $=_{def}$ e(α_1),e(α_2),e(α_3),....,e(α_n)
  - where for j = 1 to n,
  - e(α_j) $=_{def}$ ‘(‘ e(p_j) ‘,’ e(a_j) ‘,’ e(q_j) ‘,’ e(b_j) ‘)’
  - ex: for the previous example: we have
  - δ = { (s, a, g, □), (s, □□,t, □□), (s, [, s, R),
  - (g, a, s, a), (g, U, s, R), (g, [, r, R) } hence
  - e(M) =“M” = ‘(q0,a10,q01,a0),(q0,a0,q1,a0),...
  - ...,(q01,a1,q00,a00)’

TL and UTM U

- **Let Σ_0= {q,a,0,1,(,), ，} ==> the set of TL-programs,**

  TL $=_{def}$ { x | x = e(M) for some STM M }

  is the set of all string representations of STMs.

and $\Gamma_0 = \Sigma_0$ U {□□,[} is the tape alphabet of UTM U.

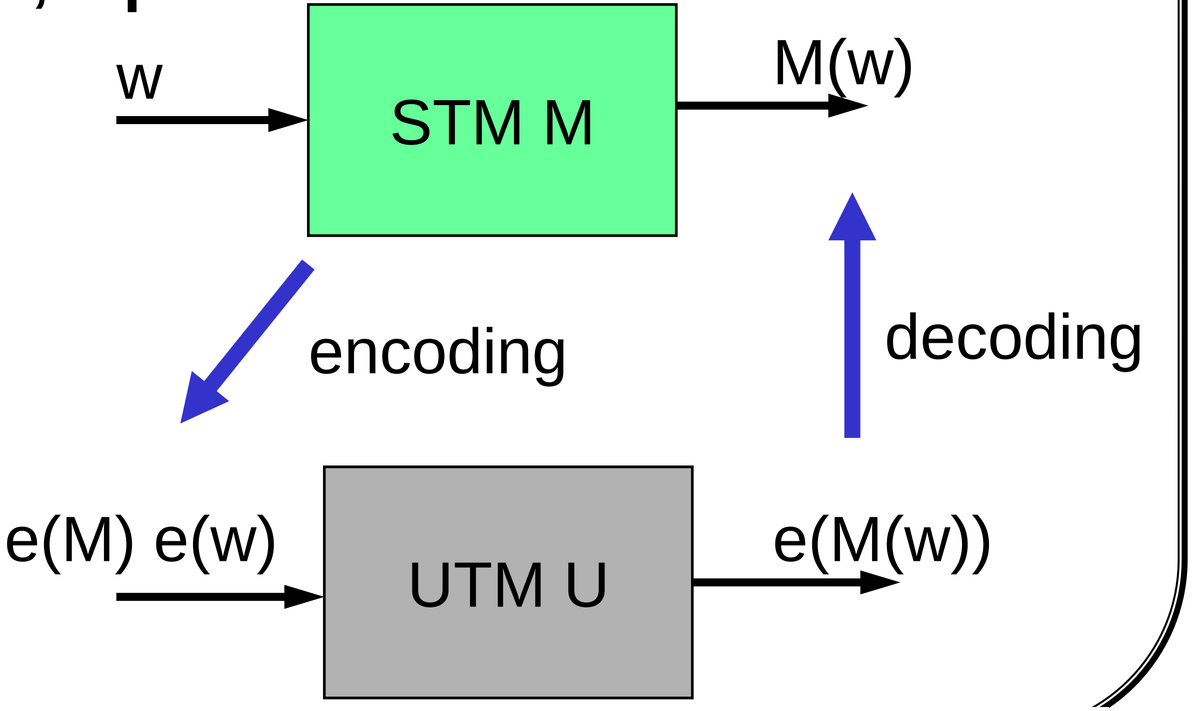
Note: Not only encoding TMs, TL can also encode data.

- **Relationship between TM, input and UTM:**

Note: if such U exists,

then we need not

implement other TMs



The design of UTM U

- We use U(e(M)e(w)) to denote the result of UTM U on executing input e(M)e(w).
- U will be shown to have the property: for all machine M and input w,

  M halts on input w with result M(w) iff

  U halts on input e(M)e(w) with result e(M(w))

  i.e., e(M(w)) = U(e(M)e(w)).
- U can be designed as a 3-tape TM.
  - 1st tape : first store input “M” “w”; and then used as the [only working] tape of M and finally store the output.
  - 2nd tape: store the program “M” (instruction table)
  - 3rd tape: store the current state of M (program counter)

- **U behaves as follows:**

1. [Initially:]

- **1. copy “M” from 1st tape to 2nd tape.**
- **2. shift “w” at the 1st tape down to the left-end**
- **3. place ‘q0’ at the 3rd tape (PC).**

2. [simulation loop:] // between each simulation step, 2,3rd tape heads point to left-end; and 1st tape head points to the a pos of the encoded version of the symbol which original M would be scanning.

Each step of M is simulated by U as follows:

2.1 [halt or not] If PC =e(t) or e(r) ==> acept or reject, respectively.

2.2 [Instruction fetch] U scans its 2nd tape until it finds a tuple ($q\alpha$,$a\beta$,$q\gamma$,$a\lambda$) s.t. (1) $q\alpha$ matches PC and (2) $a\beta$ matches 1st tape’s encoded scanned symbol

2.3 [Instruction execution] :

- 1. change PC to qγ.
- 2. perform action suggested by aλ.
- if aλ = e(b) with b$\in\Gamma$ ==> write aλ at the 1st tape head pos.
- if aλ = e(L) ==> U move 1st tape head to the previous a position.
- if aλ = e(R) ==> U move 1st tape head to the next a position or append a0^J to the 1st tape in case such an 'a' cannot be found.

***Theorem:** M(w) accepts, rejects or does not halt iff U("M""w") accepts, rejects or does not halt, respectively.*

The Halting Problem

- L : any of your favorite programming languages (C, C++, Java, BASIC, TuringLang, etc.)
- Problem: Can you write an L-program HALT(P,X), which takes another L-program P(-) and string X as input, and

HALT(P,X) return true if P(X) halt and

return false if P(X) does not halt.

- The problem of ,given an (L-)program P and a data X,determining if P will halt on input X is called the halting problem [for L].
- Use simple diagonalization principle, we can show that the halting problem is undecidable (or unsolvable)[i.e., *the program HALT(P,X) does not exist!!]*

Halt(P,X) does not exist

- **Ideas leading to the proof:**

Problem1 : What about the truth value of the sentence:

L: L is false

Problem 2 : Let S = $\{X \mid X \notin X\}$. Then does S belong to S or not ?

The analysis: $S \in S \Rightarrow S \notin S$; $S \notin S \Rightarrow S \in S$.

Problem 3 : 矛盾說: **1.** 我的矛無盾不穿 **2.** 我的盾可抵擋所有茅

結論: **1. 2.** 不可同時為真。

Problem 4 : 萬能上帝: 萬能上帝無所不能 **=>** 可創造一個不服從他的子民

=> 萬能上帝無法使所有子民服從**=>** 萬能上帝不是萬能 **.**

結論:萬能上帝不存在。

Conclusion:

□ **1. S is not a set!!**

□ **2. If a language is too powerful, it may produce expressions that is meaningless or can not be realized.**

- **Question: If HALT(P,X) can be programmed, will it incur any absurd result like the case of S?**

Ans: yes!!

The proof

- Assume HALT(P,X) does exist (i.e, can be programmed).
- Now construct a new program Diag() with HALT() as a subroutine as follows:

```
Diag(P)
$L_1$:  if HALT(P,P) then goto $L_1$ ;
$L_2$:  end.
```

- Properties of Diag():
  - 1. Diag(P) halts iff HALT(P,P) returns false.
  - 2. Hence if P is a program ==>
  - Diag(P) halts iff HALT(P,P) returns false iff P does not halt on P (i.e., P(P) does not halt).
- The absurd result: Will Diag() halt on input ‘Diag’ ?

Diag(Diag) halts <=> Diag does not halt on input Diag. (by (2))

a contradiction!! Hence both Diag and HALT are not programmable

Analysis of diag(p) and HALT(p,p)

1. Let $p_1,p_2,\ldots$ be the set of all programs accepting one string input.
2. cell(m,n) = 1/0 means m(n) halts/does not halt.
3. The diagonal row corresponds to the predicate “p(p) halts”.
4. if the diagonal row could be decided by the program HALT(p,p) then the diag() program would exist (= p_m for some m).
5. Property of diag(p_j) : $p_j(p_j)$ halts iff diag(p_j) does not halt.
4. There is a logical contradiction in (diag, diag) as to it is 0 or 1.
6. Hence neither diag() nor HALT() exist.

	p_1	p_2	p_3	…	p_k	diag	…	…
p_1	0							
p_2		1						
p_3			1					
…				….				
p_k					1			
diag	1	0	0	…	0	x ~x	0	…
…							1	
…								…

The Halting problem (for Turing machine)

- H $=_{def}$ {"M" "w" |STM M halts on input w }
  $\subseteq$ Σ_0^* = {a,q, 0,1,(,), , }* .
- Notes:

1. By Turing-Church Thesis, any computable function or language can be realized by a [standard] Turing machine; hence *H represents all instances of program-data pairs (P,D) s.t. program P halts on input D*.
2. Hence to say HALT(P,X) does not exist is equivalent to say that there is no (total) Turing machine that can decide the language H (i.e., H is not recursive.)

Theorem: H is r.e. but not recursive.

Pf: (1) H is r.e. since H can be accepted by modifying the universal TM U so that it accepts no matter it would accept or reject in the original UTM.

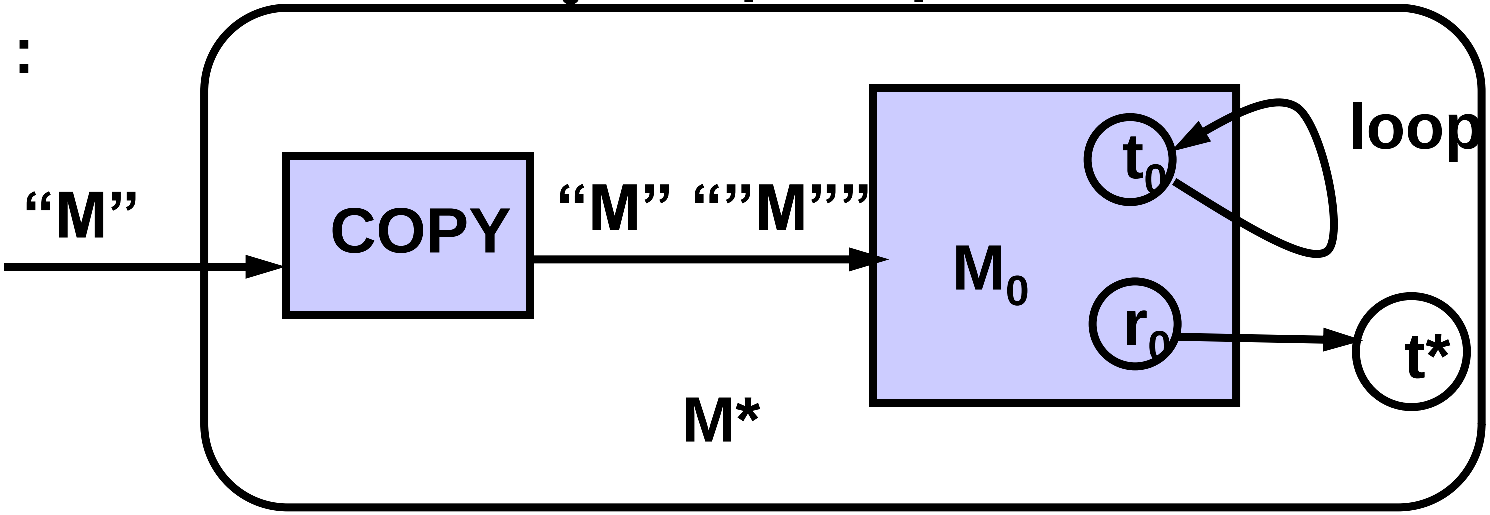
(2) H is not recursive: the proof is analog to previous proof.

Assume H is recursive => H = L(M) for some total TM M_0.

H is not recursive

Now design a new TM M* with Σ_0 as input alphabet as follows:

- On input “M” :



1. append a description ““M”” (i.e., e(“M”)) of the TL program “M” to the input program ”M”.

2. call and execute M_0(<u>“M” e(”M”)</u>) but // use alphabet Σ_0

 2.1 if M_0 accept (t_0) => M* loop and does not halt.

 2.2 if M_0 reject (r_0) => goto accept state t* of M* and halt.

Notes:1. ”M” $\in \Sigma_0^* = \{a,q, 0,1,(,), \ , \}^*$ and e(”M”) is a further encoding of “M” over Σ_0 . e.g., a q 0 1 ( ) → a10 a11 a000 a001 a010 a011 …

2. Although not all "M" is meaningful to M, we need care about only those Ms to which "M" is meaningful.

H is not recursive

Properties of M*:

0. M* and M_0 use input alphabet Σ_0.

1. M_0 is to HALT what M* is to Diag.

2. M* (“M”) halts iff M_0(“M” “”M””) reject iff [M does not halt on input ”M” or M cannot process "M"].

Absurd result: Will M*(“M*”) halt ?

By (2), M*(“M*”) halts iff M* does not halt on input “M*”,

a contradiction!!

Hence M* and M_0 does not exist.

Corollary: The class of recursive languages is a proper subclass of the class of r.e. languages.

pf: H is r.e. but not recursive.

More undecidable problems about TMs

- Theorem: The following problems about TMs are undecidable.

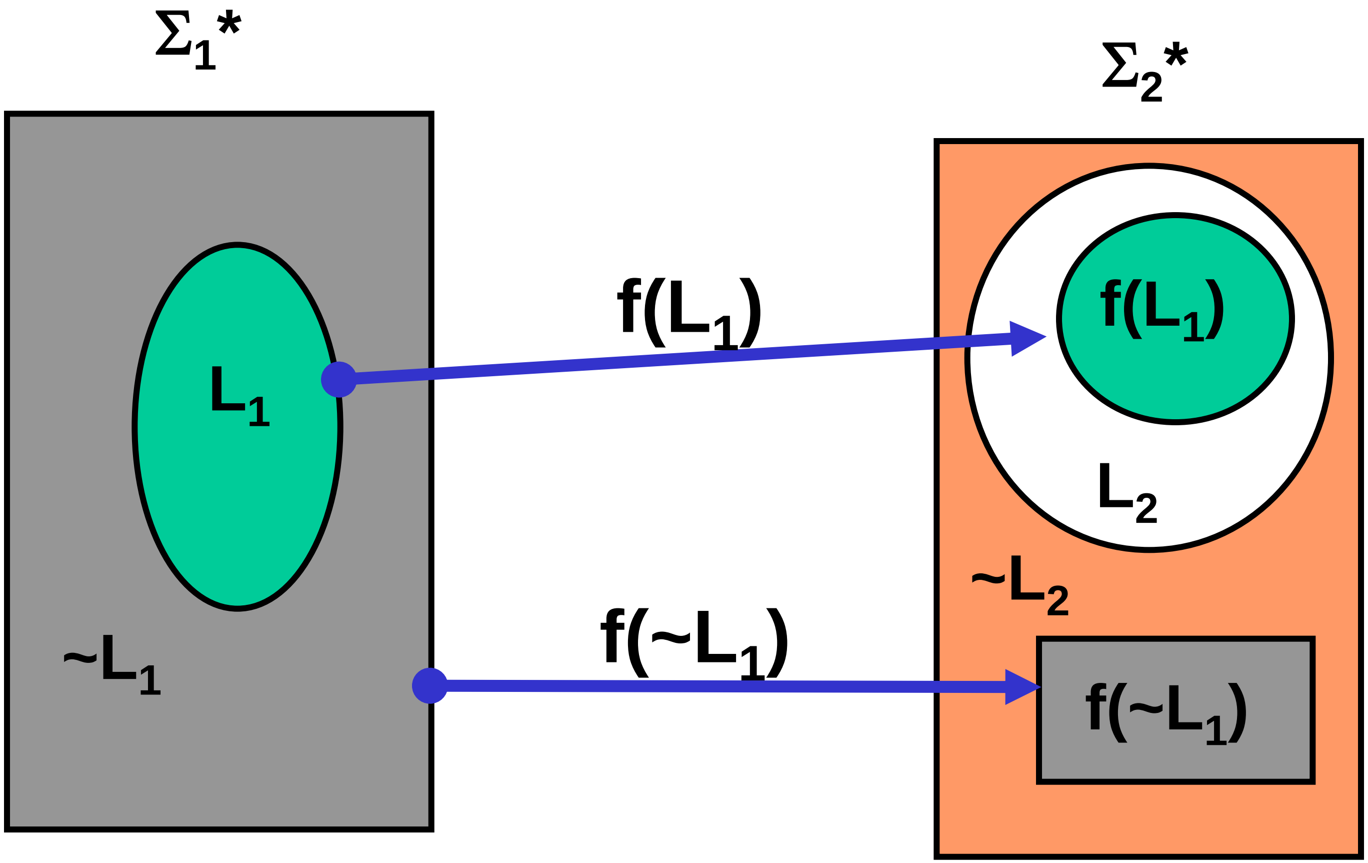
1. [General Halting Problem; GHP] Given a TM M and input w, does M halt on input w.
2. [Empty input Halting Problem; EHP] Given a TM M, does M halt on the empty input string ε?
3. Given a TM M, does M halt on any input ?
4. Given a TM M, does M halt on all inputs ?
5. Given two TMs M1 and M2, do they halt on the same inputs ?
6. Given a TM M, Is the set {x | M halts on x} recursive ?
7. [Halting problem for some fixed TM] Is there a TM M, the halting problem for which is undecidable (i.e., {x | M halt on x} is not recursive ?

pf: (1) has been shown; for (7) the UTM U is just one of such machines. (2) ~(6) can be proved by the technique of problem reduction and (1).

Problem Reduction

- L_1, L_2 : two languages over Σ_1 and Σ_2 respectively.
- A reduction from L_1 to L_2 is a computable function f : Σ_1^* --> Σ_2^* s.t. $f(L_1) \subseteq L_2$ and $f(\sim L_2) \subseteq \sim L_2$.

I.e., for all string $x \in \Sigma_1^*$, $x \in L_1$ iff $f(x) \in L_2$.



Problem reduction.

- We use $L_1 \angle_f L_2$ to mean f is a reduction from L_1 to L_2.
  We use $L_1 \angle L_2$ to mean there is a reduction from L_1 to L_2.
  If $L_1 \angle L_2$ we say L_1 is reducible to L_2.
- $L_1 \angle L_2$ means L_1 is simpler than L_2 in the sense that we can modify any program deciding L_2 to decide L_1. Hence

1. if L_2 is decidable then so is L_1 and
2. If L_1 is undecidable, then so is L_2.

Theorem: If $L_1 \angle L_2$ and L_2 is decidable(or semidecidable, respectively), then so is L_1.

Pf: Let L_2 = L(M) and T is the TM computing the reduction f from L_1 to L_2. Now let M* be the machine: on input x

1. call T, save the result f(x) on input tape
2. Call M // let M* accept (or reject) if M accept (or reject).

==> M* can decide (or semidecide) L_1. QED

Proof of other undecidable problems

- pf of [EHP]:Let f be the function s.t.

f(x) = “W_wM” if x = “M” ”w”; o/w let f(x) = “N”.

where W_w is a TM which always writes the string w on the input tape , go back to left-end and then exits. and N is a specific constant machine which never halts.

Lemma: f is a reduction from H to EHP.

pf: 1. f is computable. (OK!)

2. for any input x if x$\in$ H => x = “M””w” for some TM M and input w and M halts on input w

=> W_w M halts on empty input => f(x) = “W_wM” $\in$ EHP.

3. for any input x if x $\notin$ H =>f(x) = “N” or x = “M””w” for someTM M and input w and M(w) does not halt

=> N or W_w M does not halt on empty input => f(x) = “N” or “W_wM” $\notin$ EHP.

Corollary: H $\angle$ EHP and EHP is undecidable.

Example for why f is computable

- input: x = “M” “abcd”

=> f(x) = “R aR bR cR dL LLL M”

=> = “$(n_0,[,n_1,R)$,

$(n_1,?,n_2,a)$, (n_2,a,n_3,R)

$(n_3,?,n_4,b)$, (n_4,b,n_5,R)

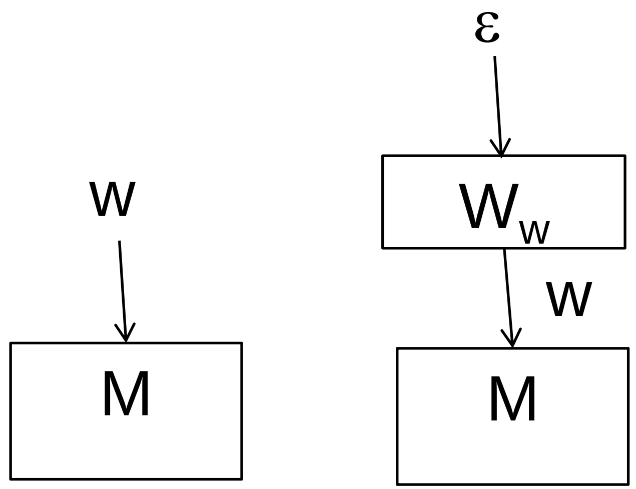
…

…

$(n_7,?,n_8,d)$,(n_8,d,n_9,L),

$(n_9,?,[, n_9,L)$,

$(n_9, [, n_{10},R)$, $(n_{10}, ?, q_0, L)$” + “M“

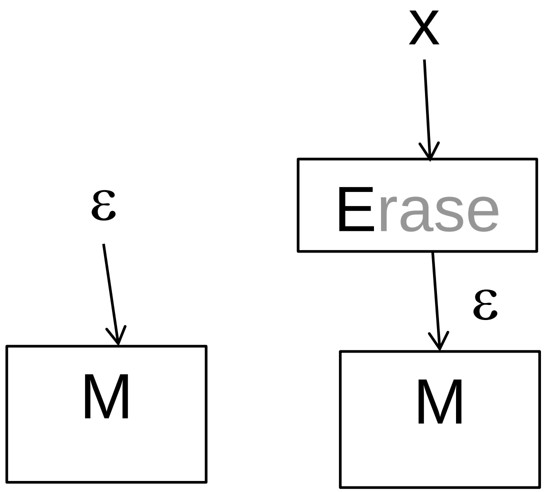


f("M""w") = "W_w M"

M(w) halts iff $W_w M(\varepsilon)$ halts

- **EHP ∠ (3)={“M” | M halt on some input},**
- **EHP ∠ (4) ={“M” | M halt on all inputs}.**
- **Let f be the function s.t.for any TM M,**

f(“M”) = “ERASE M”, where ERASE is the machine which would erase all its input, and go back to the left end.



f("M") = "E M"

- M(ε) halts iff EM(x) halts for all/some x.

Lemma: f is a reduction from EHP to (3) and(4).

pf: 1. f is computable.

2. for any TM M, “M” ∈ EHP <=> M halts on empty input

<=> ERASE M halts on some/all inputs

<=> “ERASE M” ∈ (3),(4). QED

Corollary: (3) and(4) are undecidable.

(5): (4) is reducible to (5). hint: M halts on all inputs iff M and T halt on same inputs, where T is any TM that always halts.

Proof of more undecidable problems

6.1. Given a TM M, Is the set $H(M) =_{def} \{x \mid M \text{ halts on } x\}$ recursive ?

6.2. Given a TM M, Is the set $\{x \mid M \text{ halts on } x\}$ context free ?

6.3. Given a TM M, Is the set $\{x \mid M \text{ halts on } x\}$ regular ?

pf: we show that ~EHP (non-halting problem on empty input) $\angle$ (6.1, 6.2 and 6.3). note: ~EHP is not r.e. and hence not recursive.

For any input machine M, construct a new 2-tape machine M' as follows: on input y

1. move y to 2nd tape.
2. run M as a subroutine with empty input on 1st tape.
3. if M halts, then run (1-tape UTM) U with 2nd tape as input.

analysis:

M halts on empty input => M' behaves the same as U => H(M') = H(U) =H is not recursive (and neither context free nor regular).

M loops on empty input => M' loops on all inputs => H(M') = {} is regular (and context-free and recursive as well).

Obviously M' can be computed given M, Hence ~EHP $\angle$ (6.1, 6.2 and 6.3).

Note: This means all 3 problems are even not semidecidable.

More undecidable problems

- A *semi-Thue system* is a pair $T = (\Sigma, P)$ where
  - Σ is an alphabet and
  - P is a finite set of rules of the form:
  - $\alpha \rightarrow \beta$ where $\alpha \in \Sigma^+$ and $\beta \in \Sigma^*$.
- The derivation relation induced by a semi-Thue system T:

$$\rightarrow_T =_{def} \{ (x\alpha y, x\beta y) \mid x, y \in \Sigma^*, \alpha \rightarrow \beta \in P \}$$

let $\rightarrow^*_T$ be the ref. and trans. closure of $\rightarrow_T$.

- The *word-problem* for semi-Thue system:

Given a semi-Thue system T and two words x and y, determine if $x \rightarrow^*_T y$?

Theorem: the word problem for Semi-Thue system (WST) is undecidable. I.e., WST $=_{def}$ { “(T,x,y)” | $x \rightarrow^*_T y$ } is undecidable.

The undecidability of the word problem for ST system

- **We reduce the halting problem H to WST.**
- **Let M be a STM. we construct a semi-Thue system T(M) with alphabet Σ_M = Q U Γ U $\{],q_f,q_g \}$. The rule set P_M of T(M) is defined as follows : Where a $\in \Gamma$, b $\in \Gamma$U $\{] \}$, p,q $\in$ Q,**
  - **$pa \rightarrow aq$ if $\delta(p,a) = (q, R)$,**
  - **$p] \rightarrow \Box q]$ if $\delta(p,\Box) = (q,R)$**
  - **$bpa \rightarrow qba$ if $\delta(p,a) = (q,L)$**
  - **$p \rightarrow q_f$ if $p \in \{t, r\}$ // halt =>enter q_f, ready to eliminate all tape symbols left to current position.**
  - **$q_f[\rightarrow [q_f$**
  - **$xq_f \rightarrow q_f$ where x $\in \Gamma \setminus \{ [\}$**
  - **$[q_f \rightarrow [q_g$, // ready to eliminate all remaining tape symbols**
  - **$[q_g x \rightarrow [q_g$ where x $\in \Gamma \setminus \{] \}$.**

WST is undecidable.

- Lemma: (s, [x, 0) $|\!-^*_M$ (h,y,n) iff s[x] $\rightarrow^*_{T(M)}$ [q_g].

Sol: note: h means t or r. Let (s, [x, 0) = C_0 $|\!-_M$ C_1 |- … $|\!-_M$ C_m = (h,y,n) be the derivation.

consider the mapping: f((p, [z, i)) = [$z_{1..i-1}$ p $z_{i..|z|}$],

we have C $|\!-_M$ D ⇔ f(C) $\rightarrow_{T(M)}$ f(D) for all configuration C,D (**).

This can be proved by the definition of T(M).

Hence f(C_0) = s[x] $\rightarrow^*_{T(M)}$ f(C_m) = [$y_{1..n-1}$ h $y_{n..|y|}$]

$\rightarrow^*$ [$y_{1..n-1}$ q_f $y_{n..|y|}$] $\rightarrow^*$ [q_f $y_{n..|y|}$]

$\rightarrow^*$ [q_g $y_{n..|y|}$] $\rightarrow^*$ [q_g]

Conversely, if s[x] $\rightarrow^*$ [q_g], there must exist intermediate cfgs

s.t. s[x] $\rightarrow^*$ [yhz] $\rightarrow^*$ [yq_fz] $\rightarrow^*$ [q_gz] $\rightarrow^*$ [q_g].

Hence (s, [x, 0) $|\!-^*_M$ (h,yz,|y|) and M halts on input x.

Word problem for special SemiThue systems.

Corollary: H is reduciable to WST.

Sol: for any TM M and input x, M halts on x iff (s, [x, 0) $|-^*_M$ (h,y ,n) for some y,n iff s[x] $\rightarrow_{T(M)}$ $[q_g]$.

I.e., "(M,x)" ∈ H iff "(T(M), s[x],$[q_g]$)" ∈ WST, for all TM M and input x. Hence H is reducible to WST.

Theorem: WST is undecidable.

[word problem for semi-Thue system T]: Given a semi-Thue system T, the word problem for T is the problem of, given two input strings x, y, determining if x $\rightarrow^*_T$ y.

Define WST(T) = { (x,y) | x $\rightarrow^*_T$ y }

Theorem: Let M be any TM. If the halting problem for M is undecidable, then WST(T(M)) is undecidable.

Pf: since H(M) (Halting Problem for M) is reducible to WST(T(M)).

Corollary: There are semi-Thue systems whose word problem are undeciable. In particular, WST(T(UTM)) is undeciable.

The PCP Problem

- [Post correspondence system]

Let C = $[(x_1,y_1),\ldots,(x_k,y_k)]$: a finite list of pairs of strings over Σ

A sequence j1,j2,...jn of numbers in [1,k] (n > 0) is called a solution index (and $x_{j1}\ x_{j2}\ \ldots\ x_{jn}$ a solution) of the system C iff $x_{j1}\ x_{j2}\ \ldots\ x_{jn} = y_{j1}\ y_{j2}\ \ldots\ y_{jn}$.

Ex: Let Σ = {0,1} and C= [(1,101),(10,00),(011,11)]. Does C has any solution?

Ans: yes. the sequence 1 3 2 3 is a solution index

since x1 x3 x2 x3 = 1 011 10 011 = 101 11 00 11 = y1 y3 y2 y3.

The PCP Problem

- [Post correspondence problem:] Given any correspondence system C, determine if C has a solution ?

I.e. PCP $=_{def}$ {“C” | C = $[(x_1,y_1),\ldots,(x_k,y_k)]$, $k > 0$, is a list of pairs of strings and C has a solution. }

Theorem: PCP is undecidable.

pf: Since word problem of some particular semi-Thue systems WST(T) is reducible to PCP.

Undecidability of the PCP Problem

- Let T = (Σ , P) be a semi-Thue system with alphabet {0,1} whose word problem is undecidable.
- For each pair of string x, y ∈ Σ*, we construct a PCS C(x,y) as follows:
  - C(x,y) has alphabet Σ = {0,1, *, $\underline{0}$, $\underline{1}$, $\underline{*}$, [,]}
  - if z = $a_1a_2\ldots a_k$ is a string over {0,1,*}, then let $\underline{z}$ denote $\underline{a_1a_2\ldots a_k}$.
  - wlog, let 0→0, 1→1 ∈ P.
  - C(x,y) = { $(\alpha,\underline{\beta}),(\underline{\alpha},\beta)$ | α →β ∈ P} U
  - { ([x*, [), ($\underline{*}$, *), (*,$\underline{*}$), (], $\underline{*}$y]) }

$\underline{0}$	0	$\underline{1}$	1	$\underline{*}$	*	$\underline{\alpha}$	α	…	[x*	]	
0	$\underline{0}$	1	$\underline{1}$	*	$\underline{*}$	β	$\underline{\beta}$	…	[	$\underline{*}$y]	

Example

Ex: Let $T = \{ 11 \rightarrow 1, 00 \rightarrow 0, 10 \rightarrow 01 \}$

Problem $x = 1100 \rightarrow^*_T y = 01$?

Then

$C(x,y) = \{ (11, \underline{1}), (\underline{11}, 1), (\underline{00},0), (00, \underline{0}), (\underline{10},01)\ (10,\underline{01})$

$(\underline{1},1), (1,\underline{1}),(0,\underline{0}),(\underline{0},0), (*,\underline{*}), (\underline{*},*) \} \quad \cup$

$([1100*, [), (], \underline{*}01] \}$

Derivation vs solution

The derivation : 1100 → 100 → 10 → 01 → 01
can be used to get a solution and vice versa.

[1100*	**$\underline{1}$ $\underline{0}$ $\underline{0}$ $\underline{*}$**	**1 0 ***	**$\underline{01}$ $\underline{*}$**	**01**	**]**						
[	**11 0 0 ***	**$\underline{1}$ $\underline{00}$ $\underline{*}$**	**10 ***	**$\underline{01}$**	**$\underline{*}$01]**						

Def: u,v : two strings. We say u matches v if there are common index sequence j1,j2,…,jm(m > 0) s.t. $u = y_{j1}y_{j2}\ldots y_{jm}$ and $v = x_{j1}x_{j2}\ldots x_{jm}$

Facts :Let x and y be any bit strings, then

1. $x \rightarrow_T y$ implies x^* matches $\underline{y^*}$ and $\underline{x^*}$ matches y^*,
2. x^* matches $\underline{y^*}$ (or $\underline{x^*}$ matches y^*) implies $x \rightarrow^*_T y$.

Theorem: $x \rightarrow^*_T y$ iff C(x,y) has a solution

Corollary: PCP is undecidable.

Theorem: $x \rightarrow^*_T y$ iff C(x,y) has a solution

- Only-if part: a direct result of the following lemma:
- Lemma1: If $x = x_0 \rightarrow x_1 \rightarrow x_2 \ldots \rightarrow x_{2k} = y$ is a derivation of y from x, then

$$\alpha = [x^* \, \underline{x_1{*}} \, x_2 \, {*} \, \underline{x_3{*}} \ldots \underline{x_{2k-1}} \, {*} \, x_{2k}]$$

is a solution of C(x,y).

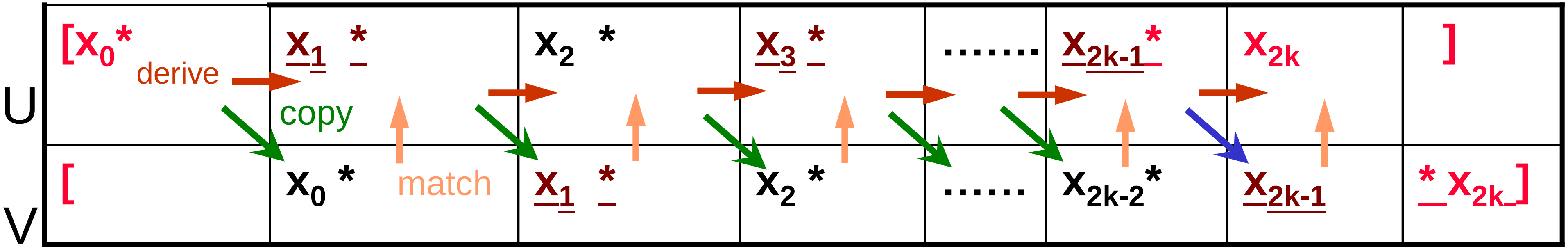
pf: The arrange of α as a sequence of strings from the 1st (and 2nd) components of C(x,y) is given as follows:

$[x_0{*}$	$\underline{x_1}$ $\underline{*}$ copy	x_2 $*$	$\underline{x_3}$ $\underline{*}$	…….	$\underline{x_{2k-1}{*}}$	x_{2k}	$]$
$[$	x_0 $*$	$\underline{x_1}$ $\underline{*}$	x_2 $*$	……	$x_{2k-2}{*}$	$\underline{x_{2k-1}}$	$\underline{*}\,x_{2k}]$

- Note since $x_j \rightarrow x_{j+1}$, by previous facts, $(\underline{x_{j+1}{*}}\,, x_j{*})$ and $(x_{j+1}{*}, \underline{x_j{*}})$ match (corresponding to the same index).
- It is thus easy to verify that both sequences correspond to the same solution index, and α hence is a solution.

Theorem: x $\rightarrow^*_T$ y iff C(x,y) has a solution

- if-part: Let a solution U of C(x,y) be arranged as follows:



U	$[x_0*$	$\underline{x_1}\ \underline{*}$	$x_2\ *$	$\underline{x_3}\ \underline{*}$	…….	$\underline{x_{2k-1}}\underline{*}$	x_{2k}	]
V	[	$x_0\ *$	$\underline{x_1}\ \underline{*}$	$x_2\ *$	……	$x_{2k-2}*$	$\underline{x_{2k-1}}$	$\underline{*}\ x_{2k}]$

derive

copy

match

- Then (U,V) must begin with:

  □ ([x* , [ ) and must end with ( ] , $\underline{*}$y])

- => the solution must be of the form: [x* w $\underline{*}$y]

if w contains] => can be rewritten as [x*z$\underline{*}$y] v$\underline{*}$y] ==> U = [x*z$\underline{*}$y] = V is also a solu.

⇒ To equal [x* , V must begin with [x*

⇒ To match x* , U must proceeds with [x* $\underline{x_1}\underline{*}$ ➔ x=x_0 → x_1 (since x match $\underline{x_1}$)

To equal [x* $\underline{x_1}\underline{*}$,V must proceeds with [x* $\underline{x_1}\underline{*}$

To match [x* $\underline{x_1}\underline{*}$,U must proceeds with [x* $\underline{x_1}\underline{*}$ x_2* ➔ x_1 → X_2

⇒ … ➔ x=x_0 →x_1 → … →x_{k-1} with U = [x*… x_{2k-2}* x_{2k-1}* and V = [x*… x_{2k-2}*

To equal U = [x*… x_{2k-2}* x_{2k-1}* ,V proceeds with [x*… x_{2k-2}* x_{2k-1}

To match V = [x*… x_{2k-2}* x_{2k-1},U must proceeds with U = [x*… x_{2k-2}* x_{2k-1} *x_{2k}

⇒ x→*y finally to close the game V proceeds with [x*… x_{2k-2}* x_{2k-1}*x2k] and U proceeds with [x*… x_{2k-2}* x_{2k-1}*x_{2k}]

Decidable and undecidable problems about CFLs

- The empty CFG problem:

  I/P: a CFG G = (N, Σ, P, S)

  O/P: “yes” if L(G) = {}; “no” if L(G) is not empty.
- There exists efficient algorithm to solve this problem.

```
  Alg empty-CFG(G)
1. Mark all symbols in Σ.
2. Repeat until there in no new (nonterminal) symbols marked
  for each rule A -> $X_1$ $X_2$ … $X_m$ in P do
    if ALL $X_i$’s are marked then mark A
3. If S is not marked then return(“yes”) else return(“no”).
```

- The alg can be implemented to run in time $O(n^2)$.
- Similar problems existing efficient algorithms:

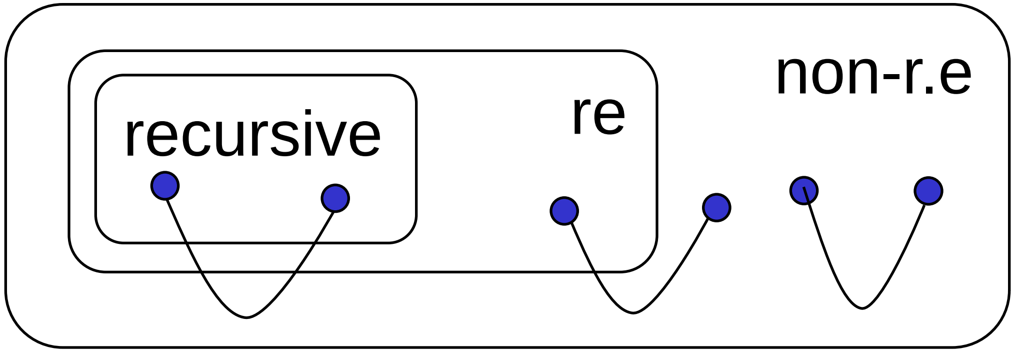
  □ 1. L(G) is infinite   2. L(G) is finite

  □ 3. L(G) contains ε (or any specific string)

  □ 4. Membership (if a given input string x in L(G))

Undecidable problems about CFL

- But how about the problem:
  - Whether L(G) = Σ*, the universal language ?
- Relations between L, ~L and their recursiveness
  - If L is recursive, ~L is recursive.
  - If L and ~L are r.e., then both are recursive.
  - If L is r.e. but not recursive, then ~L is not r.e.
  - If L is not r.e., then ~L is not recursive(but may be r.e.).



Undecidable problems for CFGs

- **Theorem** : there is no algorithm that can determine whether the languages of two CFGs are not disjoint (overlap).

(i.e., the set NDCFG = { “(G1,G2)” | G1 and G2 are CFGs and L(G1) $\cap$L(G2) is not empty } is undecidable (but it is r.e => its complementation is not r.e.).

Pf: Reduce PCP to NDCFG.

Let C = (Σ_c, { (x_1,y_1), …, (x_n,y_n)) be a PCS.

Let G_z = (N_z, Σ_z, S_z, P_z), where z = x or y,

□ N_z = {S_z} $\Sigma_z = \Sigma_c$ U {1,2,…, n}

□ P_z = { $S_z \rightarrow z_i\ S_z\ i$, $S_z \rightarrow z_i\ i$ | i = 1..n }

Lemma1: $S_z \rightarrow^*$ w iff there is seq $j_k…j_1$ in [1,n]* with

w = $z_{j1}\ z_{j2} \dots z_{jk}\ j_k\ j_{k-1} \dots j_1$.

Lemma2: C has a solution iff $L(G_x) \cap L(G_y) \neq \varnothing$

pf: C has a solution w’ iff w’ = $x_{j1}\ x_{j2} \dots x_{jk} = y_{j1}\ y_{j2} \dots y_{jk}$ for some $j_1\ j_2 \dots j_k$

iff $S_x \rightarrow^*$ w’ $j_k\ j_{k-1} \dots j_1$. and $S_y \rightarrow^*$ w’ $j_k\ j_{k-1} \dots j_1$ iff $L(G_x) \cap L(G_y) \neq \varnothing$

corollary: NDCFG is undecidable.

pf: since PCP is reducible to NDCFG.

Example

Ex: Let $\Sigma = \{a,b\}$ and $C = [(a,aba),(ab,bb),(baa,aa)]$.

$\Rightarrow G_x : S_x \rightarrow a1 \mid ab2 \mid baa3$

$\Rightarrow \qquad \mid a\, S_x\, 1 \mid ab\, S_x\, 2 \mid baa\, S_x\, 3$

$\Rightarrow G_y : S_y \rightarrow aba\ 1 \mid bb\ 2 \mid aa\ 3$

$\Rightarrow \qquad \mid aba\, S_y\, 1 \mid bb\, S_y\, 2 \mid aa\, S_y\, 3$

$L(G_x)$ and $L(G_y)$ has common member

$\Rightarrow$ a baa ab baa 3231 (Gx).

$\Rightarrow$ aba aa bb aa 3231 (Gy)

Whether L(G) = Σ* is undecidable for CFGs

- The set UCFG = { "G" | G is a CFG and L(G) = Σ* } is undecidable.

Pf: 1. For the previous grammar G_x and G_y, it can be shown that ~L(G_x) and ~L(G_z) are both context-free languages.

Hence the set A $=_{def}$ ~(L(G_x) ∩ L(G_y)) = ~L(G_x) U ~L(G_y) is context-free. Now let G_C be the CFG for A.

By previous lemma : C has no solution iff L(G_x) ∩ L(G_y) = ∅ iff ~A = ∅ iff A = Σ* iff G_C ∈ UCFG.

Hence any program deciding UCFG could be used to decide ~PCP,

but we know ~PCP is undecidable (indeed not r.e.), UCFG thus is undecidable (not r.e.).

~L(Gz) is context-free

- $\alpha \notin L(G_z)$ iff

  1. α is not of the form $\Sigma_z^+ \{1,\ldots,n\}^+$ (I.e., $\alpha \in \sim \Sigma_z^+ \{1,\ldots,n\}^+$) or

  2. α is one of the form: where $k > 0$,

  2.1 $z_{jk} \ldots z_{j1}\ j_1\ j_2 \ldots j_k \{1,\ldots,n\}^+$ or

  2.2 $\Sigma_c^+ z_{jk} \ldots z_{j1}\ j_1\ j_2 \ldots j_k$ or

  2.3 $\sim(\Sigma_c^* z_{jk})\ z_{k-1} \ldots z_{j1}\ j_1\ j_2 \ldots j_{k-1}\ j_k \{1,\ldots,n\}^*$

2.1 : G_1 : $S_1 \rightarrow S_z A$; $A \rightarrow 1 \mid 2 \ldots \mid n \mid 1A \mid 2A \mid \ldots \mid nA$

2.2 : G_2 : $S_2 \rightarrow B S_z$; $B \rightarrow a \mid b \ldots \mid Ba \mid Bb \mid \ldots$

2.3 : G_3 : $S_3 \rightarrow N_k S_z k A' \mid N_k k A'$ for all $k = 1.. n$, where

- N_k is the start symbol of the linear grammar for the reg expr $\sim(\Sigma_c^* z_{jk})$,
- $A' \rightarrow A \mid \varepsilon$

Ambiguity of CFGs is undecidable

- **The set AMBCFG = {“G” | G is an ambiguous CFG } is undecidable.**

Pf: reduce PCP to AMBCFG.

Let G_x, G_y be the two grammars as given previously.

let G be the CFG with

- **N = {S, S_x, S_y},**
- **S_G = S**
- **P = P_x U P_y U {S → S_x, S→S_y }**

Lemma: C has a solution iff $L(G_x)$ and $L(G_y)$ are not disjoint iff G is ambiguous.

pf: 1. C has a solution w

=> $w = x_{j1}\, x_{j2} \ldots x_{jk} = y_{j1}\, y_{j2} \ldots y_{jk}$ for some $J = j_k\, j_{k-1} \ldots j_1$

=> $S \rightarrow S_x \rightarrow^* wJ$ and $S \rightarrow S_y \rightarrow^* wJ$

=> G is ambiguous.

2. G ambiguous

=> there exist two distinct derivations Δ_1 and Δ_2 for a certain string α=wJ => Δ_1 and Δ_2 must have distinct 1st steps (since G_x and G_y are deterministic)

=> C has a solution w with solution index J.

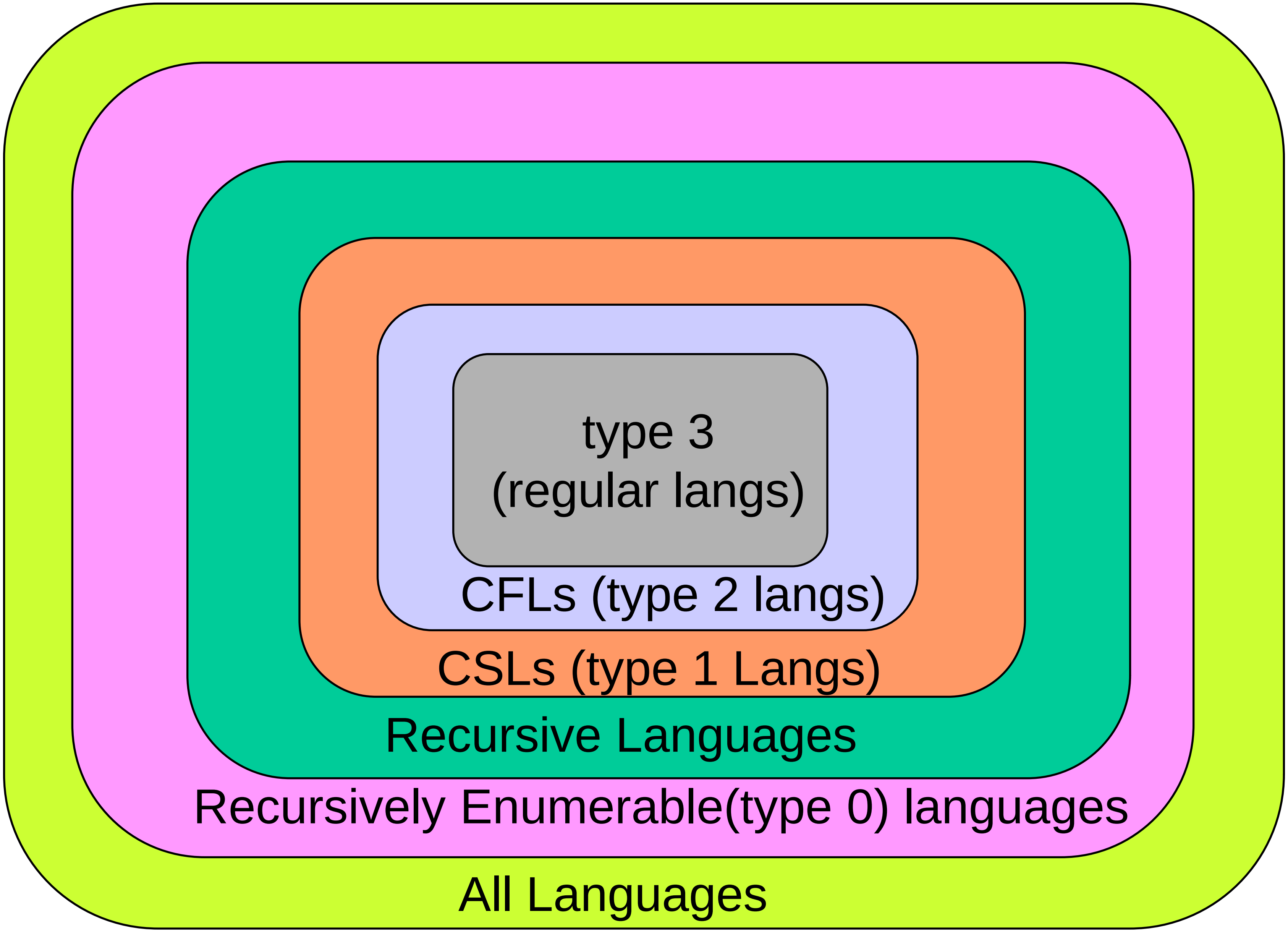
Corollary: AMBCFG is undecidable.

The Chomsky Hierarchy

- Relationship of Languages, Grammars and machines

Language	recognition model	generation model
Regular languages (type 3) languages	Finite automata (DFA, NFA)	regular expressions linear grammars
CFL (type 2, Context Free) languages	Pushdown automata	CFG ; type 2 (context free) grammars
CSL (type 1, Context sensitive) Languages	LBA (Linear Bound Automata)	CSG (Context sensitive, type 1 Grammars)
Recursive Languages	Total Turing machines	-
R.E. (Recursively enumerative, type 0) Languages	Turing machines	GPSG(type 0, general phrase-structure, unrestricted) grammar

The Chomsky Hierarchy



Phrase-structure grammar

Def.: A phrase-structure grammar G is a tuple G=(N, Σ, S, P) where

- **N, Σ, and S are the same as for CFG, and**
- **P, a finite subset of $(NU\Sigma)^* N (NU\Sigma)^* \times (NU\Sigma)^*$, is a set of production rules of the form:**
- **$\alpha \rightarrow \beta$ where**
- **$\alpha \in (NU\Sigma)^* N (NU\Sigma)^*$ is a string over $(NU\Sigma)^*$ containing at least on nonterminal.**
- **$\beta \in (NU\Sigma)^*$ is a string over $(NU\Sigma)^*$.**

Def: G is of type

- **2 => $\alpha \in N$.**
- **3 (right linear)=> $A \rightarrow a B$ or $A \rightarrow a$ ($a \neq \varepsilon$) or $S \rightarrow \varepsilon$.**
- **1 => $S \rightarrow \varepsilon$ or $|\alpha| \leq |\beta|$.**

Derivations

- Derivation $\rightarrow_G \subseteq (N\cup\Sigma)^* \times (N\cup\Sigma)^*$ is the least set of pairs such that :

$$\forall\ x,y \in (\Sigma\ \cup N)^*,\ \alpha \rightarrow \beta \in P,\quad x\alpha y \rightarrow_G x\beta y.$$

- Let $\rightarrow^*_G$ be the ref. and tran. closure of $\rightarrow_G$.
- L(G) : the languages generated by grammar G is the set:

$$L(G) =_{def} \{x \in \Sigma^* \mid S \rightarrow^*_G x \}$$

Example

- **Design CSG to generate the language $L=\{0^n1^n2^n \mid n \geq 0\}$, which is known to be not context free.**

Sol:

Consider the CSG G_1 with the following productions:

$S \rightarrow \varepsilon$, $\quad S \rightarrow 0SA2 \quad 2A \rightarrow A2$,

$0A \rightarrow 01 \quad 1A \rightarrow 11$

For G_1 we have

$S \rightarrow 0SAB \rightarrow \ldots \rightarrow 0^k(A2)^k \rightarrow^* 0^kA^k2^k \rightarrow 0^k1^k2^k \therefore L \subseteq L(G1)$.

Also note that

- □ **if $S \rightarrow^* \alpha \Longrightarrow \#0(\alpha) = \#(A|1)(\alpha) = \#(2)(\alpha)$.**
- □ **if $S \rightarrow^* \alpha \in \{0,1,2\}^* \Rightarrow \alpha_k = 0 \Longrightarrow \alpha_j = 0$ for all $j < k$.**
- □ **$\alpha_k = 1 \Rightarrow \alpha_j = 1$ or 0 for all $j < k$.**
- □ **Hence α must be of the form $0^*1^*2^* \Rightarrow \alpha \in L$. QED**

- **Lemma 1 : if $S \rightarrow^{*} \alpha \in \Sigma^{*}$, then it must be the case that**

  $S \rightarrow^{*} 0^{*}\ S\ (A + 2)^{*} \rightarrow 0^{*}\ (A+2)^{*} \rightarrow^{*} 0^{*}1\ (A+2)^{*} \rightarrow^{*} 0^{*}1^{*}2^{*}$.