MA/CSSE 474
Theory of Computation

TM Macro Language

Your Questions?

- Previous class days' material
- Reading Assignments
- HW 14 problems
- Exam 3
- Anything else

TMs are complicated

- … and *very* low-level!
- We need higher-level "abbreviations".
  - Macros

A Macro language for Turing Machines

(1) Define some basic machines

You need to learn this simple language. I will use it and I expect you to use it on HW and tests (for exams I'll give you a handout with the details).

- Symbol writing machines

For each $x \in \Gamma$, define M_x, written as just x, to be a machine that writes x. Read-write head ends up in original position.

- Head moving machines

R: for each $x \in \Gamma$, $\delta(s, x) = (h, x, \rightarrow)$
L: for each $x \in \Gamma$, $\delta(s, x) = (h, x, \leftarrow)$

- Machines that simply halt:

h, which simply halts (don't care whether it accepts).
n, which halts and rejects.
y, which halts and accepts.

Checking Inputs and Combining Machines

Machines to:

- Check the tape and branch based on what character we see, and
- Combine the basic machines to form larger ones.

To do this, we need two forms:

- $M_1 \; M_2$

- $M_1 \xrightarrow{\textit{<condition>}} M_2$

Turing Machines Macros Cont'd

Example:

$$\begin{tikzcd} >M_1 \arrow[r, "a"] \arrow[d, "b"'] & M_2 \\ M_3 \end{tikzcd}$$

- Start in the start state of M_1.
- Compute until M_1 reaches one of its halt states, which are not halt states in the combined machine.
- Examine the tape and take the appropriate transition.
- Start in the start state of the next machine, etc.
- Halt if any component reaches a halt state and has no place to go.
- If any component fails to halt, then the entire machine may fail to halt.

More macros

$M_1 \xrightarrow{a} M_2$, $M_1 \xrightarrow{b} M_2$ becomes $M_1 \xrightarrow{a,\ b} M_2$

$M_1 \xrightarrow{\text{all elems of } \Gamma} M_2$ becomes $M_1 \longrightarrow M_2$ or M_1M_2

Variables

$M_1 \xrightarrow[\text{except a}]{\text{all elems of } \Gamma} M_2$ becomes $M_1 \xrightarrow{x \leftarrow \neg a} M_2$
and x takes on the value of
the current square

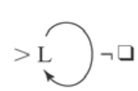
$M_1 \xrightarrow{a,\ b} M_2$ becomes $M_1 \xrightarrow{x \leftarrow a,\ b} M_2$
and x takes on the value of
the current square

$M_1 \xrightarrow{x = y} M_2$
if $x = y$ then take the transition

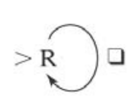
e.g., $> \xrightarrow{x \leftarrow \neg Æ} Rx$ if the current square is not blank, go right and copy it.

Blank/Non-blank Search Machines

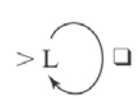
Find the first blank square to the right of the current square. $R_{Æ}$



Find the first blank square to the left of the current square. $L_{Æ}$



Find the first nonblank square to the right of the current square. $R_{\neg Æ}$



Find the first nonblank square to the left of the current square $L_{\neg Æ}$

More Search Machines

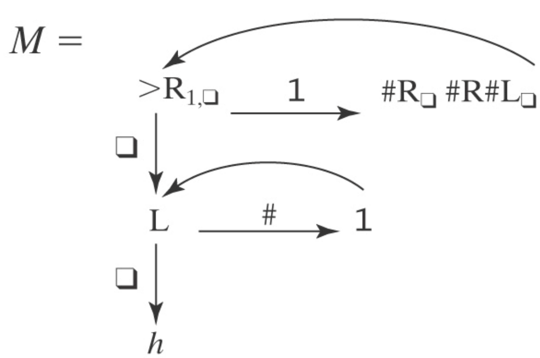
L_a	Find the first occurrence of a to the left of the current square.
$R_{a,b}$	Find the first occurrence of a or b to the right of the current square.
$\begin{array}{l} L_{a,b} \xrightarrow{a} M_1 \\ \downarrow b \\ M_2 \end{array}$	Find the first occurrence of a or b to the left of the current square, then go to M_1 if the detected character is a; go to M_2 if the detected character is b.
$L_{x\leftarrow a,b}$	Find the first occurrence of a or b to the left of the current square and set *x* to the value found.
$L_{x\leftarrow a,b}Rx$	Find the first occurrence of a or b to the left of the current square, set *x* to the value found, move one square to the right, and write *x* (a or b).

An Example

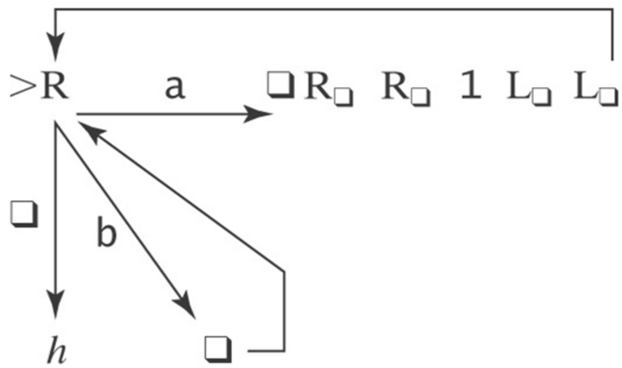
Input: Æ *w* $w \in \{1\}^*$

Output: Æ w^3

Example: Æ 111Æ Æ Æ Æ Æ Æ Æ Æ Æ Æ Æ Æ Æ Æ



What does this machine do?



Exercise

Initial input on the tape is an integer written in binary, most significant bit first (110 represents 6).

Design a TM that replaces the binary representation of *n* by the binary representation of *n*+1.

Two Flavors of TMs

1. Recognize a language
2. Compute a function

Turing Machines as Language Recognizers

Let $M = (K, \Sigma, \Gamma, \delta, s, \{y, n\})$.

- M ***accepts*** a string w iff $(s, \underline{\square}w) \vdash_M^* (y, w')$ for some string w' (that includes an underlined character).
- M ***rejects*** a string w iff $(s, \underline{\square}w) \vdash_M^* (n, w')$ for some string w'.

M ***decides*** a language $L \subseteq \Sigma^*$ iff:
For any string $w \in \Sigma^*$ it is true that:
if $w \in L$ then M accepts w, and
if $w \notin L$ then M rejects w.

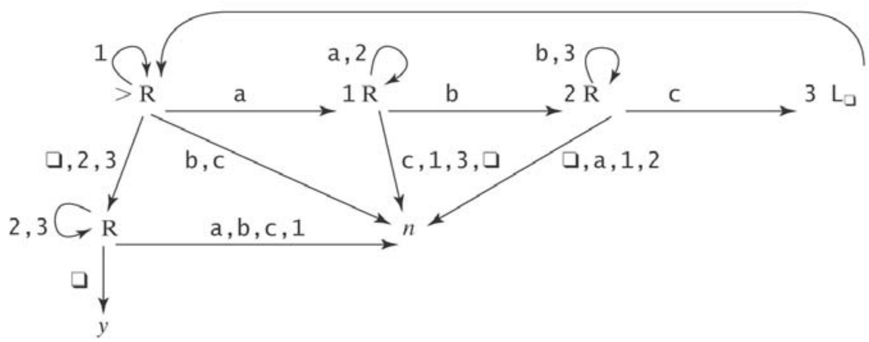
A language L is ***decidable*** iff there is a Turing machine M that decides it. In this case, we will say that L is in ***D***.

A Deciding Example

$A^nB^nC^n = \{a^n b^n c^n : n \geq 0\}$

Example: Æ aabbccÆ Æ Æ Æ Æ Æ Æ Æ Æ

Example: Æ aaccbÆ Æ Æ Æ Æ Æ Æ Æ Æ



Semideciding a Language

Let Σ_M be the input alphabet to a TM M. Let $L \subseteq \Sigma_M^*$.

M ***semidecides*** *L* iff, for any string $w \in \Sigma_M^*$:

- $w \in L \rightarrow M$ accepts w
- $w \notin L \rightarrow M$ does not accept w.

M may either:
reject or
fail to halt.

A language *L* is ***semidecidable*** iff there is a Turing machine that semidecides it. We define the set ***SD*** to be the set of all semidecidable languages.

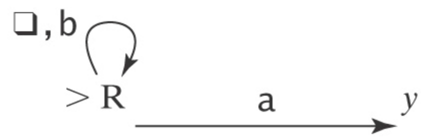
Example of Semideciding

Let L = b*a(a ∪ b)*

We can build M to semidecide L:

1. Loop
   1.1 Move one square to the right. If the character under the read head is an a, halt and accept.

In our macro language, M is:

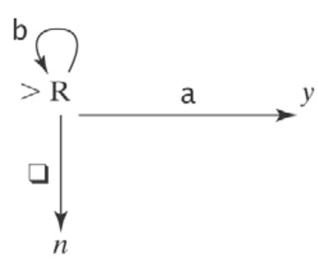


Example of Deciding the same Language

L = b*a(a ∪ b)*. We can also decide L:

Loop:
   1.1 Move one square to the right.
   1.2 If the character under the read/write head is an a, halt and accept.
   1.3 If it is ❑ , halt and reject.

In our macro language, M is:



TM that Computes a Function

Let $M = (K, \Sigma, \Gamma, \delta, s, \{h\})$.

Define $M(w) = z$ iff $(s, \underline{\text{Æ}}\,w) \mid\!\!-_M{}^{*}\ (h, \underline{\text{Æ}}\,z)$.

Let $\Sigma' \subseteq \Sigma$ be M's output alphabet.
Let f be any function from Σ^* to Σ'^*.

M ***computes*** *f* iff, for all $w \in \Sigma^*$:

- If w is an input on which f is defined: $M(w) = f(w)$.
- Otherwise $M(w)$ does not halt.

Notice that the TM's function computes with strings (Σ^* to Σ'^*), not directly with numbers.

A function f is ***recursive*** or ***computable*** iff there is a Turing machine M that computes it and that always halts.

Note that this is different than our common use of *recursive.*

Example of Computing a Function

Let $\Sigma = \{a, b\}$. Let $f(w) = ww$.

Input: $\underline{\text{Æ}}\,w\text{Æ Æ Æ Æ Æ Æ}$ Output: $\underline{\text{Æ}}\,ww\text{Æ}$

Define the copy machine C:

$\underline{\text{Æ}}\,w\text{Æ Æ Æ Æ Æ Æ} \rightarrow \text{Æ}\ w\underline{\text{Æ}}\,w\text{Æ}$

Also use the $S_{\leftarrow}$ machine:

$\text{Æ}\ u\underline{\text{Æ}}\,w\text{Æ} \rightarrow \text{Æ}\ uw\underline{\text{Æ}}$

Then the machine to compute f is just $>C\ S_{\leftarrow}\ L_{\text{Æ}}$

More details next slide

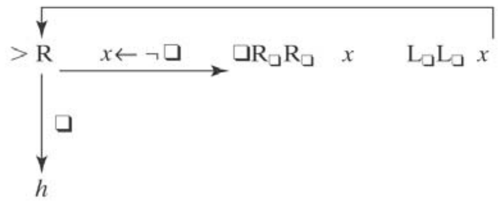
Example of Computing a Function

Let Σ = {a, b}. Let *f*(*w*) = *ww.*

Input: Æ*w*Æ Æ Æ Æ Æ Æ   Output: Æ*ww*Æ

Define **the copy machine C**:

Æ*w*Æ Æ Æ Æ Æ Æ → Æ *w*Æ*w*Æ



Then use **the $S_{\leftarrow}$ machine**:

Æ *u*Æ*w*Æ → Æ *uw*Æ

Then the machine to compute *f* is just >*C* $S_{\leftarrow}$ $L_{Æ}$

Computing Numeric Functions

For any positive integer *k*, ***value**$_k$**(n)*** returns the nonnegative integer that is encoded, base *k*, by the string *n*.

For example:

- *value*$_2$(`101`) = 5.
- *value*$_8$(`101`) = 65.

TM *M* **computes a function *f* from $\mathbb{N}^m$ to $\mathbb{N}$** iff, for some *k*:

$$value_k(M(n_1;n_2;\ldots n_m)) = f(value_k(n_1), \ldots\ value_k(n_m))$$

Note that the semicolon serves to separate the representations of the arguments

Why Are We Working with Our Hands Tied Behind Our Backs?

Turing machines	Are more powerful than any of the other formalisms we have studied so far. ☺
Turing machines	Are a **lot** harder to work with than all the real computers that are available to us. ☹

Why bother?

The very simplicity that makes it hard to program Turing machines makes it possible to reason formally about what they can do. If we can, once, show that *everything* a real computer can do can be done (albeit clumsily) on a Turing machine, then we have a way to reason about what real computers can do.