Decidable and Undecidable Languages

The Halting Problem and
The Return of Diagonalization

Tuesday, November 23, and Wednesday, November 24, 2010
Reading: Sipser 4; Kozen 31; Stoughton 5.2 & 5.3

CS235 Languages and Automata

Department of Computer Science
Wellesley College

Recursively Enumerable Languages

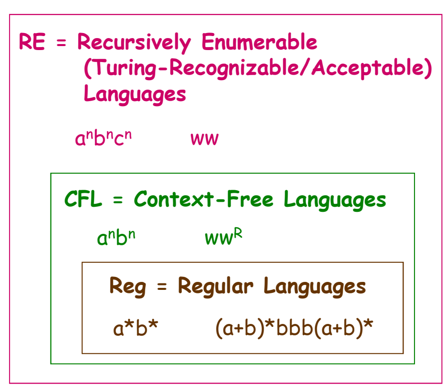
L(M) = {w | w is accepted by the Turing Machine M}

The **recursively enumerable** (r.e.) languages = the set of all languages that are the language of some Turing Machine.

These are also called **Turing-acceptable** and **Turing-recognizable** languages.

We will use **RE** to name this set.

There are many languages in **RE** that are not in **CFL**.



Decidability and Semi-Decidability

A Turing Machine **decides** a language if it rejects every string it doesn't accept – i.e., it never loops

The **recursive** languages = the set of all languages that are decided by some Turing Machine = all languages described by a non-looping TM.

These are also called the **Turing-decidable** or **decidable** languages.

We will use **Dec** to name this set.

We'll soon see examples of languages that are in **RE** but not in **Dec**. We call these languages **semi-decidable+**.

Every TM for a semi-decidable+ language halts in the accept state for strings in the language but loops for some strings not in the language.

Any language outside **Dec** is **undecidable**.
All semi-decidable+ languages are undecidable, but we'll see there are undecidable languages that aren't semi-decidable+!

RE = Recursively Enumerable (Turing-Recognizable/Acceptable) Languages

- *semi-decidable+*

Dec = Recursive (Turing-Decidable) Languages

$a^nb^nc^n$ ww
- *decidable*

CFL = Context-Free Languages

a^nb^n ww^R

Reg = Regular Languages

a*b* (a+b)*bbb(a+b)*

Dec vs. RE

For every language L in **Dec**, there is a **deciding machine** M that for an input string w is guaranteed to deliver a ball to either the accept pipe or reject pipe.

input string *w* → Turing Machine *M* for a language L in **Dec** → accept pipe / reject pipe

For every language L in **RE**, there is an **accepting machine** M that for an input string w is guaranteed to deliver a ball to the accept pipe if $w \in L$. However, if $w \notin L$, a ball *might not* be delivered to the reject pipe (M might loop).

input string *w* → Turing Machine *M* for a language L in **RE** → accept pipe / reject pipe

Game Plan for the Rest of this Lecture

Our main goal is to exhibit a language L that's semi-decidable+: L in **RE** — **Dec**.

But first:

1. we'll need to get some practice describing decidable languages that involve **language encodings**.

Then:

2. we'll define a language $HALT_{TM}$ that's in **RE** — **Dec**.

Finally:

3. we'll argue that there are languages that aren't even in **RE**!

RE = Recursively Enumerable (Turing-Recognizable/Acceptable) Languages

- *semi-decidable+*

Dec = Recursive (Turing-Decidable) Languages

$a^n b^n c^n$ ww
- *decidable*

CFL = Context-Free Languages

$a^n b^n$ ww^R

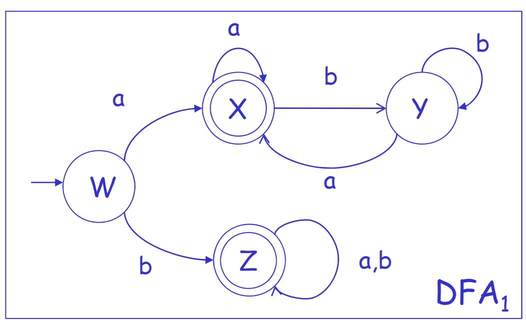
Reg = Regular Languages

a*b* (a+b)*bbb(a+b)*

Language Encodings

We will consider many languages whose strings contain encodings of DFAs, FAs, NPDAs, CFGs, and TMs. Think of such encodings as Forlan-like specifications for these machines and grammars. E.g. :

```
<DFA1> =
"({W,X,Y,Z},        <- states
 {a,b}              <- alphabet
 W,                 <- start states
 {X,Z},             <- final states
 {W,a -> X; W,b -> Z;
  X,a -> X; X,b -> Y;
  Y,a -> X; Y,b -> Y;
  Z,a -> Z; Z,b -> Z})"
transition function
```




```
<CFG1> =
"({S,A,B},  <- variables (start var first)
 {0,1},     <- terminals
 {(S,AB), (A,0A1), (A,%)
  (B,1B0), (B,%)})"   <- productions
```

S → AB
A → 0A1 | %
B → 1B0 | % CFG_1

Warm-Up: Some Decidable Languages

Show that the following languages are decidable by describing (at a high level) an algorithm that decides them (see more in Sipser 4.1)

string describing a pair of (1) a deterministic finite automaton DFA and (2) an input string w

- ACCEPT$_{DFA}$ = {(<DFA>, w) | w is in L(DFA)}
- EMPTY$_{DFA}$ = {<DFA> | L(DFA) = $\varnothing$}
- ALL$_{DFA}$ = {<DFA> | L(DFA) = Σ_{DFA}*}
- EQ$_{DFA}$ = {(<DFA$_1$>, <DFA$_2$>) | L(DFA$_1$) = L(DFA$_2$)}
- ACCEPT$_{CFG}$ = {(<CFG>, w) | w is in L(CFG)}
- EMPTY$_{CFG}$ = {<CFG> | L(CFG) = $\varnothing$}

(Warning: EQ$_{CFG}$ and ALL$_{CFG}$ are not decidable!)

What about ACCEPT$_{TM}$?

string describing a Turing Machine M and an input string w

ACCEPT$_{TM}$ = {(<M>, w) | w is in L(M) (i.e. M accepts w)}

A Turing Machine M$_{UTM}$ can accept ACCEPT$_{TM}$ as follows:

1. Check if <M> is a well-formed Turing Machine description.
   If not, M$_{UTM}$ **rejects** (<M>, w)
2. If <M> is well-formed, M$_{UTM}$ simulates the running of M on w, e.g., via a 4-tape machine:
   - Tape 1 holds <M>.
   - Tape 2 is the tape M works on (initially w).
   - Tape 3 holds the head position for tape 2.
   - Tape 4 holds the state of M.

   In this case, M$_{UTM}$ will:
   - **accept** <<M>, w> iff M accepts w.
   - **reject** <<M>, w> iff M rejects w.
   - **loop** iff M loops on w.

So ACCEPT$_{TM}$ is in **RE**.

Big question: is ACCEPT$_{TM}$ in **Dec**? I.e., is there another Turing Machine that **decides** ACCEPT$_{TM}$? (We'll see the answer is no.)

M_{UTM} is a Universal Turing Machine!

M_{UTM} is a **universal Turing Machine**

= a Turing Machine interpreter written as a Turing Machine.

There's nothing strange about this:

- We can write an SML interpreter in any language, including SML.
- We can write a Java compiler in any language, including Java.
- Why not write a Turing Machine interpreter as a Turing Machine?

The tricky bit is **bootstrapping** (take CS251 for more details):

- Our *first* SML interpreter can't be written in SML;
- Our *first* Java compiler can't be written in Java.

Self-Reference is not a Problem

Consider the following:

- A decommenting program can decomment any text file, including the decommenting program itself.
- A Java compiler can compile any Java program, including one that specifies a Java compiler.
- An SML interpreter can evaluate any SML program, including one that specifies an SML interpreter.

Moral: There is nothing inherently problematic about a program being called on "itself". The program supplied as an argument is just data, and the running program P doesn't "know" that this data describes P!

You can't eat yourself, but you can eat a *description* of yourself!

What does M_{UTM} do on the input ($\langle M_{UTM} \rangle$, ($\langle M \rangle$,w)) ?

The Halting Problem

$HALT_{TM}$ = {(<M>, w) | M halts on w (i.e. M decides w, cannot loop)}

Is $HALT_{TM}$ in **RE** (Turing-acceptable)?

Yes: Simulate M running on w. Accept if M accepts or rejects w. Loop if M loops on w.

Is $HALT_{TM}$ in **Dec** (decidable)?

No! We'll show this by diagonalization. Intuitively, the problem is that no TM for $HALT_{TM}$ can always reject (<M>, w) when M loops on w.

So $HALT_{TM}$ is semi-decidable+; our first example of such a language.

In the next lecture, we'll see that we can use $HALT_{TM}$ to show that other languages are semi-decidable+, including $ACCEPT_{TM}$.

Behavior of Turing Machines

The behavior of all Turing machines can be summarized by an infinite 2D table whose rows are Turing machines and whose columns are input strings. A table entry is A (accept), R (reject), or L (loop).

Because TM descriptions are countable, TMs are enumerable in a sequence M_1, M_2, M_3, …

input strings

Turing Machines	%	a	…	z	aa	…	zz	…
M_1	A	R	…	A	L	…	R	…
M_2	R	L	…	R	R	…	R	…
M_3	A	A	…	R	L	…	A	…
M_4	L	L	…	A	A	…	R	…
M_5	R	A	…	R	A	…	L	…
…	…	…	…	…	…	…	…	…

Towards Diagonalization

With diagonalization in mind, we focus on the subtable that results from keeping only those columns in whose input strings are valid TM descriptions.

inputs that are TM descriptions

Turing Machines

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_1	R	A	L	A	L	…
M_2	A	A	A	A	A	…
M_3	L	L	L	L	L	…
M_4	A	L	A	A	R	…
M_5	A	R	L	A	L	…
…	…	…	…	…	…	…

Suppose $HALT_{TM}$ Is Decidable

If $HALT_{TM}$ is decidable, then there is a Turing Machine M_{HALT} that, for all inputs $\langle\langle M \rangle, w\rangle$, decides if M halts on w.

inputs that are TM descriptions

Turing Machines

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_1	R	A	L	A	L	…
M_2	A	A	A	A	A	…
M_3	L	L	L	L	L	…
M_4	A	L	A	A	R	…
M_5	A	R	L	A	L	…
…	…	…	…	…	…	…

Behavior of Turing Machines on inputs that are TM descriptions

inputs that are TM descriptions

TM descriptions

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
$\langle M_1 \rangle$	A	A	R	A	R	…
$\langle M_2 \rangle$	A	A	A	A	A	…
$\langle M_3 \rangle$	R	R	R	R	R	…
$\langle M_4 \rangle$	A	R	A	A	A	…
$\langle M_5 \rangle$	A	A	R	A	R	…
…	…	…	…	…	…	…

Behavior of M_{HALT} on inputs of the form $(\langle M_i \rangle, \langle M_j \rangle)$

Diagonalization for $HALT_{TM}$

If M_{HALT} exists, then there is another machine M_{DIAG} defined as:

$M_{DIAG}(\langle M\rangle)$ = if $M_{HALT}(\ (\langle M\rangle,\langle M\rangle)\ )$ then reject else accept

M_{DIAG} inverts every entry on the diagonal of M_{HALT}.

Because M_{DIAG} is a TM, it must appear in the list of descriptions. But it can't! What is $M_{DIAG}(\langle M_{DIAG}\rangle)$?

So the assumption that M_{HALT} exists is false, and $HALT_{TM}$ isn't decidable.

inputs that are TM descriptions

TM descriptions

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\langle M_5\rangle$	…	$\langle M_{DIAG}\rangle$
$\langle M_1\rangle$	$\boxed{A}$	A	R	A	R	…	
$\langle M_2\rangle$	A	$\boxed{A}$	A	A	A	…	
$\langle M_3\rangle$	R	R	$\boxed{R}$	R	R	…	
$\langle M_4\rangle$	A	R	A	$\boxed{A}$	A	…	
$\langle M_5\rangle$	A	A	R	A	$\boxed{R}$	…	
…	…	…	…	…	…	…	
$\langle M_{DIAG}\rangle$	R	R	A	R	A		$\boxed{???}$

Behavior of M_{HALT} on inputs of the form $(\langle M_i\rangle,\langle M_j\rangle)$

Alternative Diagonalization for $HALT_{TM}$

We can instead perform the diagonalization on the original table of Turing machine behaviors

If M_{HALT} exists, then there is another machine M'_{DIAG} defined as:

$M'_{DIAG}(\langle M\rangle)$ = if $M_{HALT}(\ (\langle M\rangle,\langle M\rangle)\ )$ then loop else accept

M'_{DIAG} inverts every entry on the diagonal of the table, leading to a contradiction.

inputs that are TM descriptions

Turing Machines

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\langle M_5\rangle$	…	$\langle M'_{DIAG}\rangle$
M_1	$\boxed{R}$	A	L	A	L	…	
M_2	A	$\boxed{A}$	A	A	A	…	
M_3	L	L	$\boxed{L}$	L	L	…	
M_4	A	L	A	$\boxed{A}$	R	…	
M_5	A	R	L	A	$\boxed{L}$	…	
…	…	…	…	…	…	…	
M'_{DIAG}	L	L	A	L	A		$\boxed{???}$

Behavior of Turing Machines on inputs that are TM descriptions

The Halting Problem in Scheme*

Suppose we could write a function halts? that determines if an input function f halts when applied to its argument f:

```
(define (halts? f x) …)
```

Then we could write the following:

```
(define (loop) (loop))

(define (diag f)  (if (halts? f f) (loop) #t))
```

Suppose (diag diag) halts. Then it should loop!

Suppose (diag diag) loops. Then it should halt with #t!

This accurately captures the diagonalization dilemma, but is a bit hokey since a Scheme halts? function can't examine the structure of the input function in the way that a Turing Machine M_{HALT} can examine the structure of a TM description.

* It's tougher to show this in SML because its type system prohibits self-application.

So What? Does the Halting Problem Matter?

Yes! Compilers and other programs that analyze programs often want to perform **termination analysis** to determine whether or not the evaluation of a particular subexpression will terminate.

E.g., it's often helpful to evaluate a subexpression, but only safe to do so if evaluation will terminate (otherwise, the compiler might not terminate!)

Sadly, the halting problem says there is no iron-clad way to determine in advance whether or not subexpression evaluation will terminate.

We'll see later that various heuristics can be used, but they can only conservatively approximate exact termination analysis.

We'll also see that most forms of program analysis suffer from the same problems as termination analysis.

With Great Power Comes Great Uncomputability

Turing machines and equivalent models of computation (lambda calculus, Java, SML, etc.) are far more powerful than finite automata and pushdown automata.

But the power is gained via features that can cause programs to loop infinitely. If we want the power, we must live with the looping.

Programs that loop vs. taking a long time

How do we distinguish programs that run a long time from ones that loop?

E.g. 3x+1 problem:

$$f(x) = \begin{cases} 3x + 1, & \text{if } x \text{ is odd} \\ x/2, & \text{if } x \text{ is even} \end{cases}$$

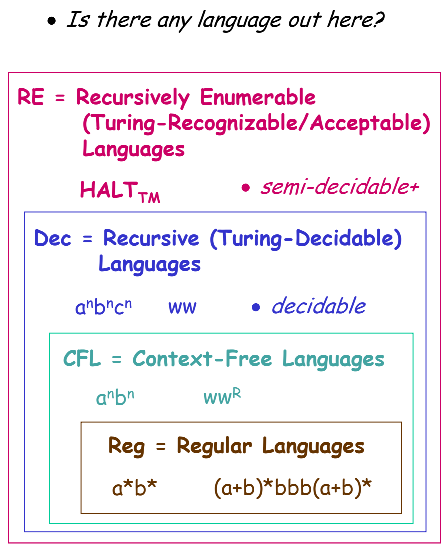
Problem: for all n, is there some i s.t. $f^i(n) = 1$? I.e., is it the case that iterating f at a starting point never loops?

No one knows! This is an open problem!

Are there non-RE Languages?

Our next big question: are there any languages that are not **RE**?

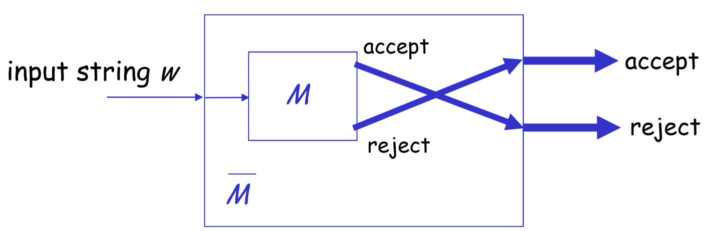
We'll see the answer is yes.
In fact, ***way*** yes (lots of them!)



Dec is Closed Under Complement

Suppose M is a Turing Machine that decides L.

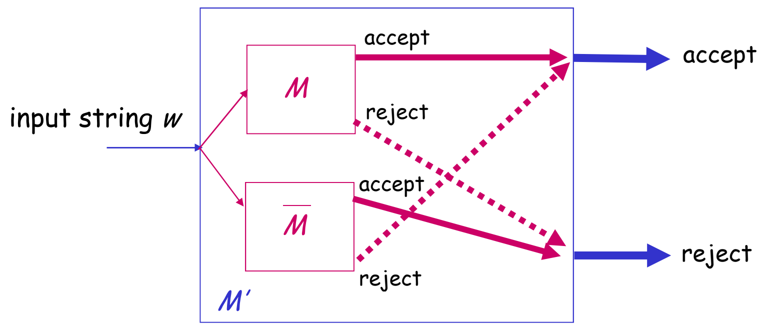
We can construct a machine $\overline{M}$ that decides $\overline{L}$:



RE is ***NOT*** Closed Under Complement

Suppose *M accepts L* and $\overline{M}$ accepts $\overline{L}$.

Then L is decidable (by M' below)!



Important detail: M and $\overline{M}$ must be run in parallel, not sequentially! See next slide.

So if *L* is semi-decidable⁺ (*L* in **RE** - **Dec**), then $\overline{L}$ ***can't*** be in **RE**!

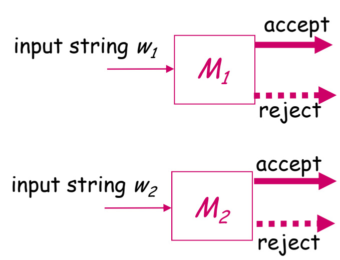
Digression: How to Run Two TMs in Parallel?

We often want to run two accepting machines M_1 and M_2 on the same or different inputs.

The machines should not be run *sequentially* (say M_1 before M_2) because if M_1 loops, M_2 will never run.

Instead, the machines are run *in parallel* by performing one step of M_1 followed by one step of M_2, alternating between the two machines.

After each step, we can check whether either machine is in its accept state or reject state. So it's possible to run M_1 and M_2 in parallel until one accepts, both accept, one rejects, or both reject.

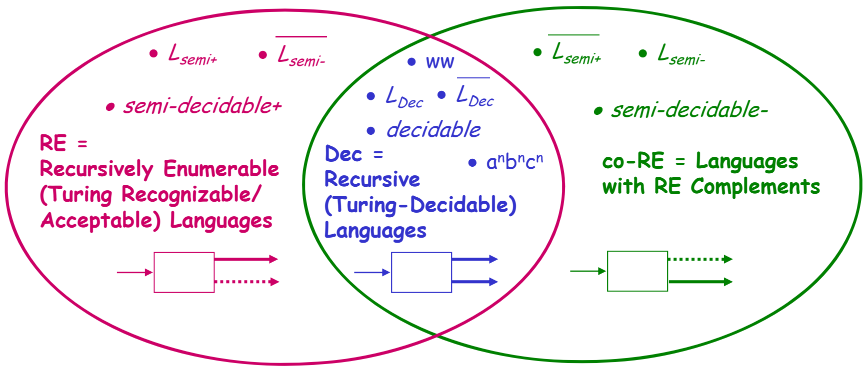


Complementary, My Dear Watson

co-RE is the set of languages whose complements are **RE**.

If L is semi-decidable+ (in **RE** — **Dec**),
then $\overline{L}$ is **semi-decidable-** (in **co-RE** — **Dec**).

We'll say L is **semi-decidable** if it's semi-decidable+ or semi-decidable-



Not-Even-Semidecidable Languages

Any Turing Machine can be described as a string in Σ^* (for an appropriate Σ),

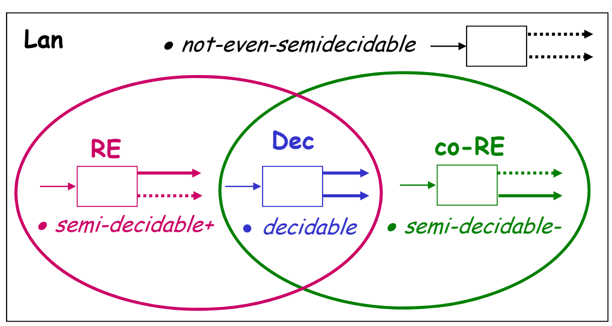
states#inputAlphabet#tapeAlphabet#transitions#start#accept#reject

We can enumerate all possible Turing Machine descriptions, so
RE and **co-RE** must be *countable*.

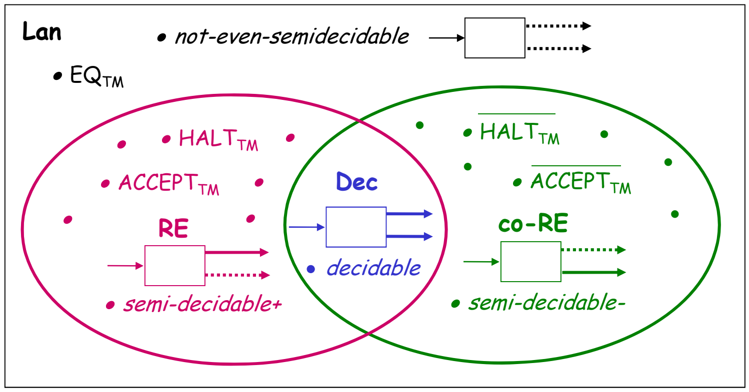
But we know that **Lan** = $P(\Sigma^*)$ is an *uncountable* set.

So **Lan** – (**RE** U **co-RE**) must contain *many* languages. We'll call these languages **not-even-semidecidable**. We'll see concrete examples of these next lecture.

For an not-even-semidecidable language, every Turing Machine must loop for some strings in the language and some not in the language.



What We're Aiming For



Note: **undecidable** =
semi-decidable+ U semi-decidable- U not-even-semidecidable

Closure Properties of **Dec** and **RE**

Dec is closed under:
- union
- intersection
- concatenation
- Kleene star
- **complement**

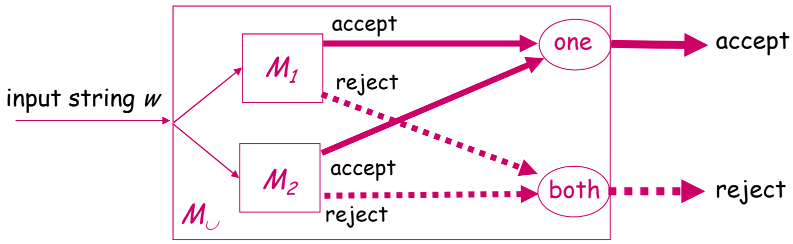
RE is closed under:
- union
- intersection
- concatenation
- Kleene star

We've already seen the complement story:
- L in **Dec** implies $\overline{L}$ in **Dec**.
- L and $\overline{L}$ are in **RE** implies L and $\overline{L}$ in **Dec**.
- L is semi-decidable+ (in **RE** — **Dec**) and $\overline{L}$ semi-decidable- (in **co-RE** — **Dec**)

Now we'll study the other properties

RE is Closed Under Union

Suppose L_1, L_2 are accepted by machines M_1, M_2, respectively.
$L_1 \cup L_2$ is accepted by the following machine $M_{\cup}$, which runs M_1 and M_2 in parallel until one accepts or both reject.



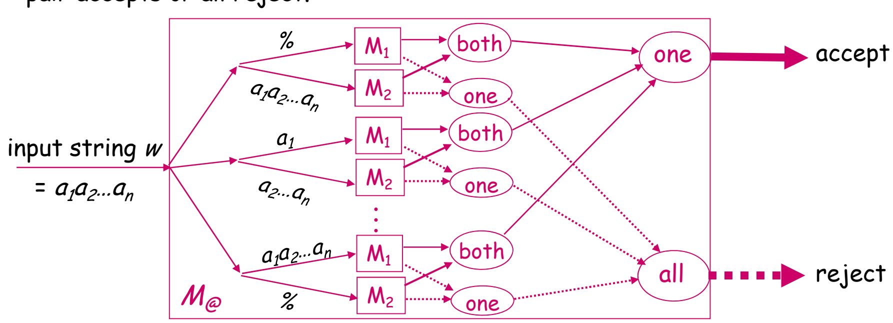
It is essential to run M_1 and M_2 in *parallel*. Why?

A similar diagram shows **Dec** is closed under union,
but in that case M_1 and M_2 can be run *sequentially*. Why?

Similar arguments (PS10) show **RE**, **Dec** are closed under intersection.

RE is Closed Under Concatenation

Suppose L_1, L_2 are accepted by machines M_1, M_2, respectively. L_1 @ L_2 is accepted by the following machine $M_@$, which runs copies of M_1 and M_2 in parallel on all possible decompositions of *w* into pairs of substrings until one pair accepts or all reject.



This diagram can be implemented by a Turing Machine program that loops over all possible decompositions, interleaving the steps for each.

Similar ideas show that **Dec** is closed under concatenation and that both **RE** and **Dec** are closed under Kleene star.