Variations:
- Multiple tracks
- Multiple tapes
- Non-deterministic

# TURING MACHINE VARIATIONS ENCODING TURING MACHINES UNIVERSAL TURING MACHINE


## Your Questions?

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

# Turing Machine Variations

There are many extensions we might like to make to our basic Turing machine model. We can do this because:

**We can show that every extended machine has an equivalent* basic machine.**

We can also place a bound on any change in the complexity of a solution when we go from an extended machine to a basic machine.

Some possible extensions:

- Multi-track tape.
- Multi-tape TM
- Nondeterministic TM

*Equivalent* means "accepts the same language," or "computes the same function."

# Multiple-track tape

We would like to be able to have TM with a multiple-track tape. On an n-track tape, Track i has input alphabet $\Sigma_i$ and tape alphabet $\Gamma_i$.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1st track | .... | 1 | B | B | B | B | B | B | B | .. |
| 2nd track | .... | B | B | 1 | B | B | B | B | B | .. |
| 3rd track | .. | B | B | B | B | 1 | B | B | B | .. |
| 4th track | .. | B | B | B | B | B | B | 1 | B | . |
| 5th track | .. | a | b | c | d | e | f | g | h | . |

# Multiple-track tape

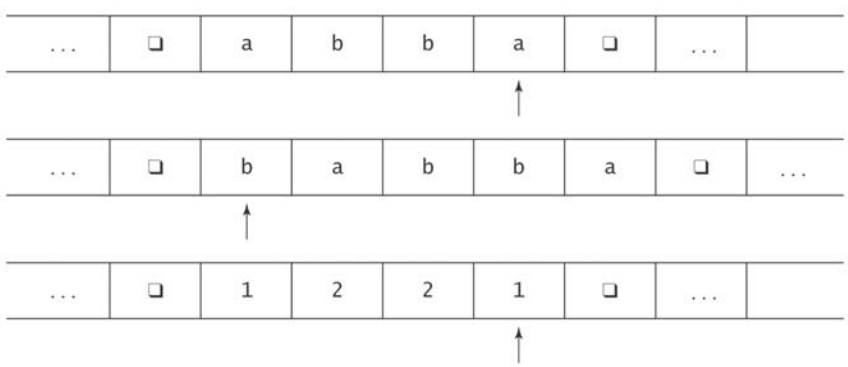
On an n-track tape, track i has input alphabet $\Sigma_i$ and tape alphabet $\Gamma_i$.

**We can simulate this with an ordinary TM.**

A transition is based on the current state and the combination of all of the symbols on all of the tracks of the current "column".

$\Gamma$ is the set of n-tuples of the form [ $\gamma_1$, …, $\gamma_n$], where $\gamma_1 \in \Gamma_i$. $\Sigma$ is similar. The "blank" symbol is the n-tuple [❑, …, ❑]. Each transition reads an n-tuple from $\Gamma$, and then writes an n-tuple from $\Gamma$ on the same "square" before the head moves right or left.

# Multiple Tapes



# Multiple Tapes

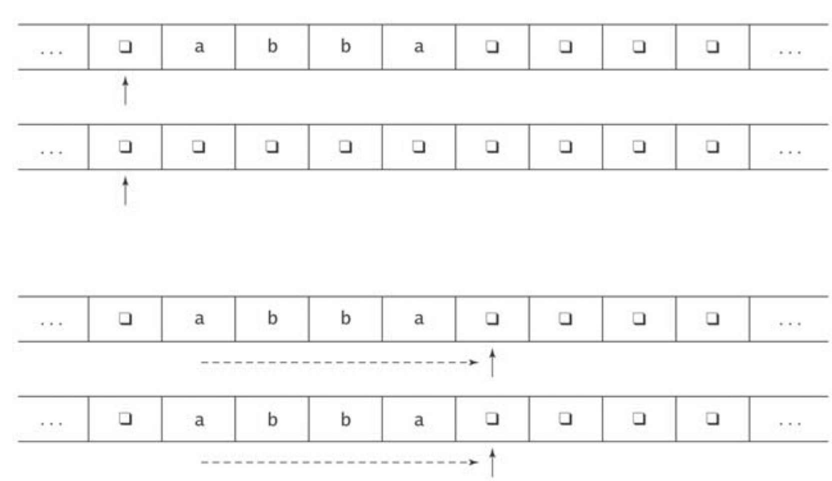
The transition function for a $k$-tape Turing machine:

$$
\begin{array}{llll}
((K\text{-}H), & \Gamma_1 & \text{to} & (K, \Gamma_{1'}, \{\leftarrow, \rightarrow, \uparrow\} \\
, & \Gamma_2 & & , \Gamma_{2'}, \{\leftarrow, \rightarrow, \uparrow\} \\
, & . & & , \; . \\
, & . & & , \; . \\
, & \Gamma_k) & & , \Gamma_{k'}, \{\leftarrow, \rightarrow, \uparrow\})
\end{array}
$$

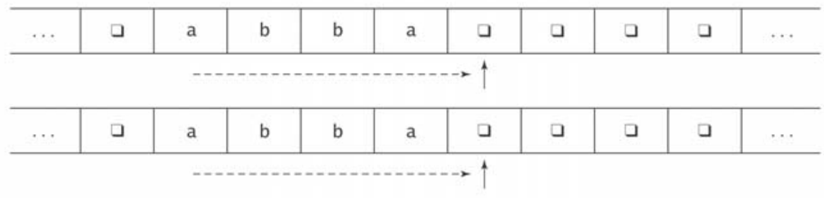
Input: initially all on tape 1, other tapes blank.
Output: what's left on tape 1, other tapes ignored.

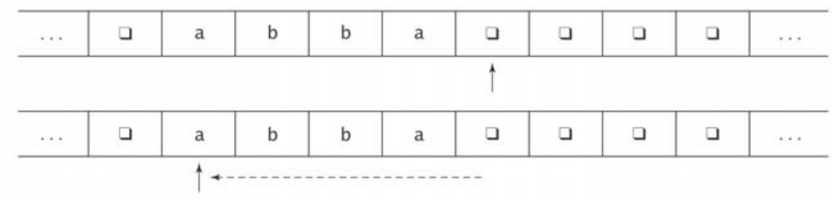
Note: On each transition, any tape head is allowed to stay where it is.

# Example: Copying a String

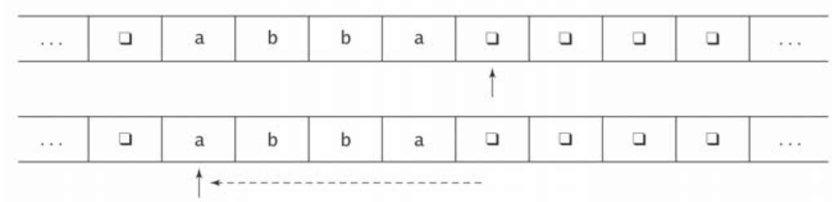


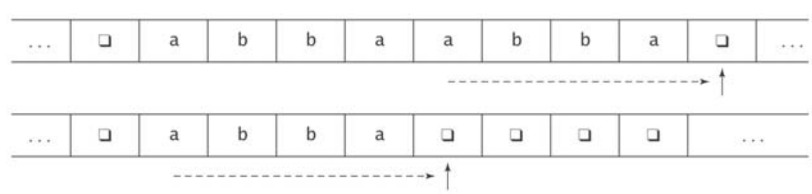
# Example: Copying a String





# Example: Copying a String





## Another Two Tape Example: Addition

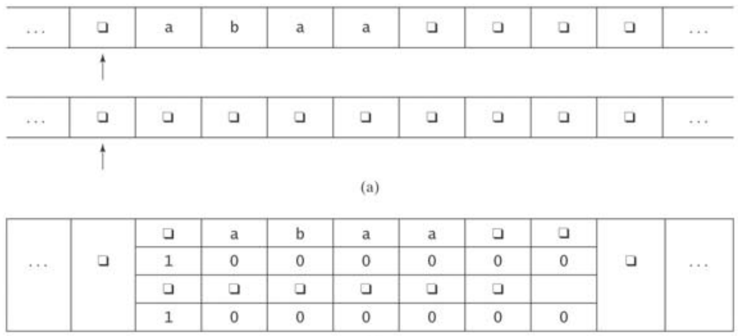

## Adding Tapes Does Not Add Power

***Theorem:*** Let $M = (K, \Sigma, \Gamma, \delta, s, H)$ be a $k$-tape Turing machine for some $k > 1$. Then there is a standard TM $M' = (K', \Sigma', \Gamma', \delta', s', H')$ where $\Sigma \subseteq \Sigma'$, and:

- On input $x$, $M$ halts with output $z$ on the first tape iff $M'$ halts in the same state with $z$ on its tape.
- On input $x$, if $M$ halts in $n$ steps, $M'$ halts in $\mathcal{O}(n^2)$ steps.

***Proof:*** By construction.

# Representation of k-tape machine by a 2k-track machine

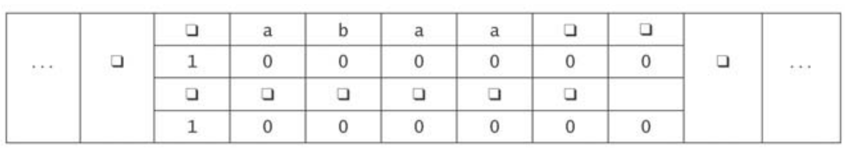


(a) (b)

Alphabet ($\Gamma$ ') of *M'* = $\Gamma \cup (\Gamma \times \{0, 1\})^k$:

Æ , a, b, (Æ , 1, Æ , 1), (a, 0, Æ ,0), (b, 0, Æ , 0), …

# The Operation of *M'*



1. Set up the multitrack tape.
2. Simulate the computation of *M* until (if) *M* would halt:
   - 2.1 Scan left and store in the state the *k*-tuple of characters under the read heads.
     Move back right.
   - 2.2 Scan left and update each track as required by the transitions of *M*. If necessary, subdivide a new (formerly blank) square into tracks.
     Move back right.
3. When *M* would halt, reformat the tape to throw away all but track 1, position the head correctly, then go to *M*'s halt state.

## How Many Steps Does *M'* Take?

Let: $w$ be the input string, and
$n$ be the number of steps it takes $M$ to execute.

Step 1 (initialization): $\mathcal{O}(|w|)$.

Step 2 ( computation):
Number of passes = $n$.
Work at each pass: **2.1** = 2 · (length of tape).
= $2 \cdot (|w| + n)$.
**2.2** = $2 \cdot (|w| + n)$.
Total: $\mathcal{O}(n \cdot (|w| + n))$.

Step 3 (clean up): $\mathcal{O}$(length of tape).

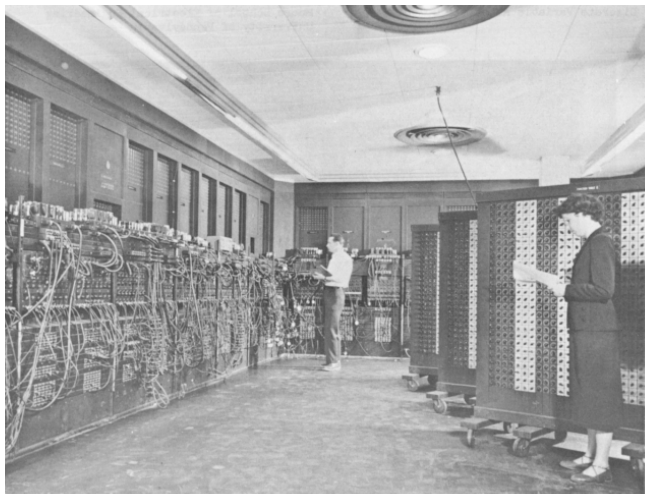
Total: $\mathcal{O}(n \cdot (|w| + n))$.
$= \mathcal{O}(n^2)$. *

* assuming that $n \geq w$

## Universal Turing Machine

# The Universal Turing Machine

**Problem**: All our machines so far are hardwired.



ENIAC - 1945

# Programmable TM?

**Problem**: All our machines so far are hardwired.

**Question**: Can we build a programmable TM that accepts as input:

*program* *input string*

executes the program on that input, and outputs:

*output string*

## The Universal Turing Machine

Yes, the ***Universal Turing Machine***.

To define the Universal Turing Machine *U* we need to:

1. Define an encoding scheme for TMs.

2. Describe the operation of *U* when it is given input <*M*, *w*>, the encoding of:

   - a TM *M*, and

   - an input string *w*.

## Encoding the States

- Let $i$ be $\lceil \log_2(|K|) \rceil$.
  Each state is encoded by a letter and a string of i binary digits.

- Number the states from 0 to $|K|$-1 in binary:
  - The start state, s, is numbered 0.
  - Number the other states in any order.

- If $t'$ is the binary number assigned to state $t$, then:
  - If $t$ is the halting state $y$, assign it the string y$t'$.
  - If $t$ is the halting state $n$, assign it the string n$t'$.
  - If $t$ is the halting state $h$, assign it the string h$t'$.
  - If $t$ is any other state, assign it the string q$t'$.

## Example of Encoding the States

Suppose *M* has 9 states.

*i* = 4

*s* = `q0000`,

The other states (suppose that *y* is 3 and *n* is 4):

```
q0001   q0010   y0011   n0100
q0101   q0110   q0111   q1000
```

## Encoding the Tape Alphabet

**The tape alphabet is Γ**

Let *j* be $\lceil \log_2(|\Gamma|) \rceil$.
Each tape alphabet symbol is encoded as
a*y* for some $y \in \{0, 1\}^+$, $|y| = j$

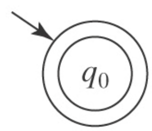
*The blank symbol is always encoded as the j-bit representation of 0*

**Example:** Γ = { Æ , b, c, d }. *j* = 2.

Æ = a00
b = a01
c = a10
d = a11

## A Special Case

We will treat this as a special case:




## Encoding other Turing Machines

**The transitions:** (state, input, state, output, move)

Example: (`q000`, `a000`, `q110`, `a000`, $\rightarrow$)

*A TM encoding is a sequence of transitions, in any order*

# An Encoding Example

Consider $M$ = ( { $s$, $q$, $h$ }, { a, b, c }, { Æ , a, b, c }, $\delta$, $s$, { $h$ } ):

| state | symbol | δ |
|---|---|---|
| $s$ | Æ | ($q$, Æ , →) |
| $s$ | a | ($s$, b, →) |
| $s$ | b | ($q$, a, ←) |
| $s$ | c | ($q$, b, ←) |
| $q$ | Æ | ($s$, a,→) |
| $q$ | a | ($q$, b, →) |
| $q$ | b | ($q$, b, ←) |
| $q$ | c | ($h$, a, ←) |

| state/symbol | representation |
|---|---|
| $s$ | q00 |
| $q$ | q01 |
| $h$ | h10 |
| Æ | a00 |
| a | a01 |
| b | a10 |
| c | a11 |

Decision problem: Given a string w, is there a TM M such that w=<M> ?

Is this problem decidable?

<*M*> = (q00,a00,q01,a00,→), (q00,a01,q00,a10,→),
(q00,a10,q01,a01, ←), (q00,a11,q01,a10,←),
(q01,a00,q00,a01,→), (q01,a01,q01,a10,→),
(q01,a10,q01,a11,←), (q01,a11,h10,a01,←)

# Enumerating Turing Machines

***Theorem:*** There exists an infinite lexicographic enumeration of:

(a) All syntactically valid TMs.

(b) All syntactically valid TMs with specific input alphabet $\Sigma$.

(c) All syntactically valid TMs with specific input alphabet $\Sigma$ and specific tape alphabet $\Gamma$.

# Enumerating Turing Machines

***Proof:*** Fix $\Sigma$ = {(, ), a, q, y, n, 0, 1, comma, $\rightarrow$, $\leftarrow$}, ordered as listed. Then:

1. Lexicographically enumerate the strings in $\Sigma^*$.
2. As each string *s* is generated, check to see whether it is a syntactically valid Turing machine description. If it is, output it.

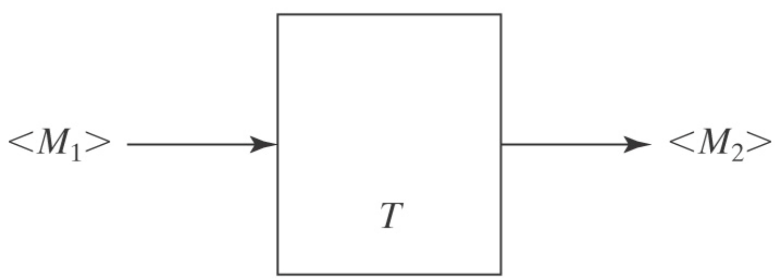
To restrict the enumeration to symbols in sets $\Sigma$ and $\Gamma$, check, in step 2, that only alphabets of the appropriate sizes are allowed.

We can now talk about the $i^{th}$ Turing machine.

# Another Benefit of Encoding

Benefit of defining a way to encode any Turing machine *M*:

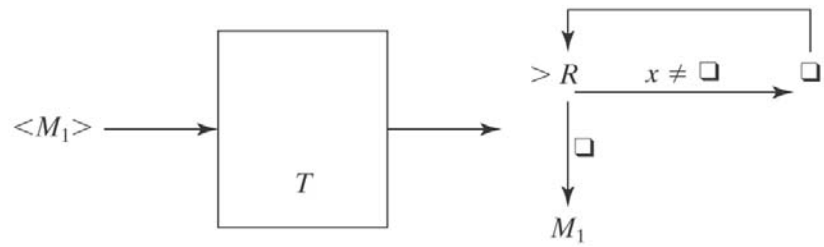
- We can talk about operations on programs (TMs).



# Example of a Transforming TM *T*:

***Input:*** a TM $M_1$ that reads its input tape and performs some operation *P* on it.

***Output:*** a TM $M_2$ that performs *P* on an empty input tape.




The machine $M_2$ (output of T) empties its tape, then runs $M_1$.

# Encoding Multiple Inputs

Let:

$<x_1, x_2, \ldots x_n>$

represent a single string that encodes the sequence of individual values:

$x_1, x_2, \ldots x_n$.

## The Specification of the Universal TM

On input <*M*, *w*>, *U* must:

- Halt iff *M* halts on *w*.
- If *M* is a deciding or semideciding machine, then:
    - If *M* accepts, accept.
    - If *M* rejects, reject.
- If *M* computes a function, then *U*(<*M*, *w*>) must equal *M*(*w*).

## How *U* Works

*U* will use 3 tapes:

- Tape 1: *M*'s tape.
- Tape 2: <*M*>, the "program" that *U* is running.
- Tape 3: *M*'s state.

# The Universal TM

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | <M-------------------M, | | | | w-------------w> | | | | |
| | 1 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| ❑ | ❑ | ❑ | ❑ | ❑ | ❑ | ❑ | ❑ | ❑ | ❑ |
| | 1 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| | ❑ | ❑ | ❑ | ❑ | ❑ | ❑ | ❑ | | |
| | 1 | 0 | 0 | 0 | 0 | 0 | 0 | | |

Initialization of *U*:
1. Copy <*M*> onto tape 2.
2. Look at <*M*>, figure out what *i* is, and write the encoding of state *s* on tape 3.

After initialization:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | ❑ | ❑ | ❑ | ❑ | <w-------------w> | | | | |
| | 0 | 0 | 0 | 0 | 1 | 0 | 0 | | |
| ❑ | <M-------------------M> | | | | ❑ | ❑ | ❑ | ❑ | ❑ |
| | 1 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| | q | 0 | 0 | 0 | ❑ | ❑ | ❑ | | |
| | 1 | ❑ | ❑ | ❑ | ❑ | ❑ | ❑ | | |

# The Operation of *U*

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | ❑ | ❑ | ❑ | ❑ | <w-------------w> | | | | |
| | 0 | 0 | 0 | 0 | 1 | 0 | 0 | | |
| ❑ | <M-------------------M> | | | | ❑ | ❑ | ❑ | ❑ | ❑ |
| | 1 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| | q | 0 | 0 | 0 | ❑ | ❑ | ❑ | | |
| | 1 | ❑ | ❑ | ❑ | ❑ | ❑ | ❑ | | |

Simulate the steps of *M* :
1. Until *M* would halt do:
   1.1 Scan tape 2 for a quintuple that matches the current state, input pair.
   1.2 Perform the associated action, by changing tapes 1 and 3. If necessary, extend the tape.
   1.3 If no matching quintuple found, halt. Else loop.
2. Report the same result *M* would report.

## How long does *U* take?

## If A Universal Machine is Such a Good Idea …

Could we define a Universal Finite State Machine?

Such a FSM would accept the language:

$L = \{<F, w> : F \text{ is a FSM, and } w \in L(F) \}$

# The Church-Turing Thesis

## Are We Done?

FSM $\Rightarrow$ PDA $\Rightarrow$ Turing machine

Is this the end of the line?

There are still problems we cannot solve with a TM:

- There is a countably infinite number of Turing machines since we can lexicographically enumerate all the strings that correspond to syntactically legal Turing machines.
- There is an uncountably infinite number of languages over any nonempty alphabet.
- So there are more languages than there are Turing machines.

## What Can Algorithms Do?

1. Can we come up with a system of axioms that makes all true statements be theorems (I.e. provable from the axioms)?
   The set of axioms can be infinite, but it must be decidable
2. Can we always decide whether, given a set of axioms, a statement is a theorem or not?

In the early 20$^{th}$ century, it was widely believed that the answer to both questions was "yes."

## Gödel's Incompleteness Theorem

Kurt Gödel showed, in the proof of his Incompleteness Theorem [Gödel 1931], that the answer to question 1 is no. In particular, he showed that there exists no decidable axiomatization of Peano arithmetic that is both consistent and complete.

Complete: All true statements in the language of the theory are theorems

## The Entscheidungsproblem

From Wikipedia: The Entscheidungsproblem ("decision problem", David Hilbert 1928) asks for an algorithm that will take as input a description of a formal language and a mathematical statement in the language, and produce as output either "True" or "False" according to whether the statement is true or false. The algorithm need not justify its answer, nor provide a proof, so long as it is always correct.

Three equivalent formulations:

1. Does there exist an algorithm to decide, given an arbitrary sentence *w* in first order logic, whether *w* is valid?
2. Given a set of axioms *A* and a sentence *w*, does there exist an algorithm to decide whether *w* is entailed by *A*?
3. Given a set of axioms, *A*, and a sentence, *w*, does there exist an algorithm to decide whether *w* can be proved from *A*?

## The Entscheidungsproblem

To answer the question, in any of these forms, requires formalizing the definition of an algorithm:

- Turing: Turing machines.
- Church: lambda calculus.

Turing proved that Turing machines and the lambda calculus are equivalent.

## Church's Thesis (Church-Turing Thesis)

All formalisms powerful enough to describe everything we think of as a computational algorithm are equivalent.

This isn't a formal statement, so we can't prove it. But many different computational models have been proposed and they all turn out to be equivalent.

# The Church-Turing Thesis

Examples of equivalent formalisms:

- Modern computers (with unbounded memory)
- Lambda calculus
- Partial recursive functions
- Tag systems (FSM plus FIFO queue)
- Unrestricted grammars:

  a*S*a → *B*
- Post production systems
- Markov algorithms
- Conway's Game of Life
- One dimensional cellular automata
- DNA-based computing
- Lindenmayer systems

# The Lambda Calculus

The successor function:

(λ *x*. *x* + 1) 3 = 4

Addition: *(λ x. λ y. x + y)* 3 4

This expression is evaluated by binding 3 to *x* to create the new function (*λ y.* 3 + *y*), which is applied to 4 to return 7.

In the pure lambda calculus, there is no built in number data type. All expressions are functions. But the natural numbers can be defined as lambda calculus functions. So the lambda calculus can effectively describe numeric functions.

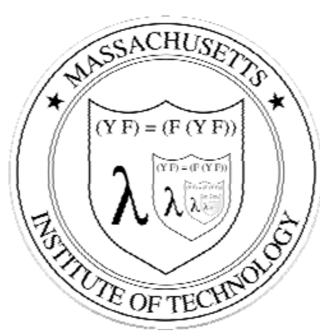
# The Lambda Calculus

```
> (define Y
    (lambda (f)
      ((lambda (x) (f (lambda (y) ((x x) y))))
       (lambda (x) (f (lambda (y) ((x x) y)))))))
> (define H
    (lambda (g)
      (lambda (n)
        (if (zero? n)
            1
            (* n (g (- n 1)))))))
> ((Y H) 5)
120
>
```

# Λ-Calculus in Scheme

**The Applicative Y Combinator**

```
> (((lambda (f)
      ((lambda (x) (f (lambda (y) ((x x) y))))
       (lambda (x) (f (lambda (y) ((x x) y))))))
    (lambda (g)
      (lambda (n)
        (if (zero? n)
            1
            (* n (g (- n 1)))))))
   5)
120
```



# Tag Systems

A tag system (or a Post machine) is an FSM augmented with a FIFO queue.

Simple for WW:
Not so simple for PalEven

# The Power of Tag Systems

Tag systems are equivalent in power to Turing machines because the TM's tape can be simulated with the FIFO queue.

Suppose that we put abcde into the queue:

| | a | b | c | d | e | |
|---|---|---|---|---|---|---|

To read the queue, we must remove the a first.

But suppose we want to remove e first:

# The Power of Tag Systems

Tag systems are equivalent in power to Turing machines because the TM's tape can be simulated with the FIFO queue.

Suppose that we push abcde onto the queue:

| | | | | | | |
|---|---|---|---|---|---|---|
| | a | b | c | d | e | |

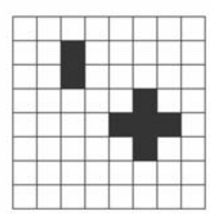
To read the queue, we must remove the a first.

But suppose we want to remove e first:

Treat the queue as a loop.

# The Game of Life

Playing the game



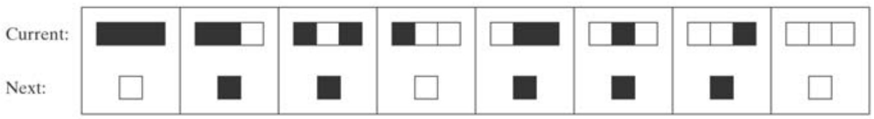
At each step of the computation, the value for each cell is determined by computing the number of neighbors (up to a max of 8) it currently has, according to the following rules:

- A dead cell with exactly three live neighbors becomes a live cell (birth).
- A live cell with two or three live neighbors stays alive (survival).
- In all other cases, a cell dies or remains dead (overcrowding or loneliness).

We'll say that a game halts iff it reaches some stable configuration.

# Elementary Cellular Automata


Wolfram's Rule 110 is a universal computer, if you can figure out how to encode the program and the input in the initial configuration:




For some fascinating pictures, look up Rule 110.
Conjectured in 1985 to be Turing complete, proved in 2000 by Matthew Cook.
Also: http://en.wikipedia.org/wiki/A_New_Kind_of_Science