“Effective” is an informal term

- Alonzo Church and (his student) Alan Turing separately tried to formalize effective computability in 1936 and 1937, respectively.

- You can think of “effective” as meaning there is an algorithm for the problem.
Effective?

- DFAs and PDAs offer a *limited* form of computability. They are effective, but not every effectively-computable function is computable by a PDA.

- Computing membership in the language \(\{a^n b^n c^n \mid n \geq 0\}\) is a case in point. There is a way to compute membership in this language, but not by a PDA.
History of Effective Computability
In 1671, Leibniz began to invent a machine that could execute all four arithmetical operations, gradually improving it over a number of years. This "Stepped Reckoner" attracted fair attention and was the basis of his election to the Royal Society in 1673. In 1679, while mulling over his binary arithmetic, Leibniz imagined a machine in which binary numbers were represented by marbles, governed by a rudimentary sort of punched cards. Modern electronic digital computers replace Leibniz's marbles moving by gravity with shift registers, voltage gradients, and pulses of electrons, but otherwise they run roughly as Leibniz envisioned in 1679.

The origin of the Entschiedungsproblem [decision problem] goes back to Gottfried Leibniz, who in the seventeenth century, after having constructed a successful mechanical calculating machine, dreamt of building a machine that could manipulate symbols in order to determine the truth values of mathematical statements. Leibniz may have been the first computer scientist and information theorist.[65] Early in life, he documented the binary numeral system (base 2), then revisited that system throughout his career.[66] He anticipated Lagrangian interpolation and algorithmic information theory. His calculus ratiocinator anticipated aspects of the universal Turing machine.

In 1671, Leibniz began to invent a machine that could execute all four arithmetical operations, gradually improving it over a number of years. This "Stepped Reckoner" attracted fair attention and was the basis of his election to the Royal Society in 1673. In 1679, while mulling over his binary arithmetic, Leibniz imagined a machine in which binary numbers were represented by marbles, governed by a rudimentary sort of punched cards. Modern electronic digital computers replace Leibniz's marbles moving by gravity with shift registers, voltage gradients, and pulses of electrons, but otherwise they run roughly as Leibniz envisioned in 1679.

http://en.wikipedia.org/wiki/Gottfried_Leibniz
Dedekind and Peano

- **Richard Dedekind** (1831-1916) in 1888 proposed an **axiomatic foundation** for the **natural numbers**, whose primitive notions were **1** and the **successor** function.

- **Giuseppe Peano** (1858–1932) in 1889 created a simpler set of axioms (also using **1**), acknowledging Dedekind.

  [Using 0 vs. 1 is a non-issue, since the axioms give an **abstract** characterization of the natural numbers. 0 is mostly used today.]
Wilhelm Ackermann

- **Ackermann** (1896-1962), a student of David Hilbert, investigated properties of recursive functions over the natural numbers.

- He defined the family of functions now known as **primitive recursive**, and found an obviously-computable function that is *not* in the family (Ackermann’s function).

- He proved the consistency of a set of axioms for arithmetic, and co-authored one of the first **mathematical logic books** with Hilbert.
Ackermann’s Function

\[
A(m, n) = \begin{cases} 
  n + 1 & \text{if } m = 0 \\
  A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\
  A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0.
\end{cases}
\]

Sometimes used as a test-case for bignum computation.

\[A(4,2) = 2^{65536} - 3,\] an integer having 19,729 decimal digits.
Kurt Gödel

- Gödel (1906-1978) proposed in 1931 a formalism for **computable functions** that extended Ackermann’s primitive recursive functions.

- He called these **general recursive functions**. The formalism is similar to functional languages.

- These played a role in his work on representing logical formulas arithmetically, and his **incompleteness theorem**.
Alonzo Church

- Church (1903–1995) was the first to prove, in 1936, that the **predicate calculus is undecidable**, i.e. there is no algorithm that will tell whether or not a formula is provable.

- This result is known as **Church’s Theorem**.

- The problem itself then went by the name **The Entscheidungsproblem** (“Decision problem”).

- Apparently his proof used some of the work of his student, Stephen Kleene.
The Lambda Calculus

• Church invented the lambda calculus in 1936. It is a system for defining functions and computing with them.

• Its use appears in many functional languages, including Lisp/Scheme and more recently Python.

• It is the residue of a logical system invented by Church that was eventually shown inconsistent, because certain expressions were equivalent to their own negation. As computation, however, this simply expresses a form of non-terminating behavior.

Let \( k = (\lambda x. \neg(x \ x)) \), then \( kk = (\lambda x. \neg(x \ x))k = \neg(kk) \).

So \( kk = \neg(kk) = \neg(\neg(kk)) = \neg(\neg(\neg(kk))) = \ldots \)
Universality of the Lambda Calculus

- Church conjectured that the lambda calculus was equivalent to effective computability.

- However, this claim was not obviously correct nor totally convincing, and some (e.g. Gödel) expressed skepticism.

- Eventually the lambda calculus was shown to be universal, in the sense that it is equivalent to Turing machines and other formalisms.
Stephen Kleene

- Kleene (1909-1994, the same Kleene who discovered regular expressions in 1951) and Turing were both students of Church.

- He wrote in 1952 the text *Introduction to Metamathematics*, which expounded on Gödel’s work and also introduced the partial recursive functions, a more structured formalism for computability.

- [This gave rise to the saying: “Kleeneness is next to Gödelness.”]
Alan Turing

- Turing (1912-1954) was Church’s student at Princeton.

- In 1937 (a year after Church published the lambda calculus), Turing published his paper "On Computable Numbers, with an Application to the Entscheidungsproblem" (Proceedings of the London Mathematical Society 42) which defined and defended a variant of what we now call the **Turing machine**, although Turing appeared to be trying to characterize computability by a *human* “computer”.

- Turing’s defense was more intuitive and grounded than Church’s defense of the lambda calculus, in my opinion.
Alan Turing, 1912-1954

Turing’s decryption computer da Bombe, 1941 which helped win WWII

Turing memorial plaque, Manchester, England

http://en.wikipedia.org/wiki/Alan_Turing
Turing Machines

- Most contemporary presentations deviate from the letter of Turing’s original model, although not in the spirit.

- There are tradeoffs involving programming convenience vs. simplicity.
TM Essentials

- There is a **tape** with cells onto which letters in a finite alphabet can be written.
- The tape can be extended arbitrarily at its extremity by adding new cells as needed.
- The read/write **head** is attached to the control and is positioned over one cell of the tape.
- The head can **move** one cell at a time in either direction.
- There is a finite-state **control** which governs writing and motion, as a function of the current cell contents and the current state.
Turing Machines

- A Turing machine in control state $q$, with a 3-letter alphabet, which includes blank (b).
- Details of the control are not shown.
TM Components (Sipser Version)

(Q, Σ, Γ, δ, q₀, q_{accept}, q_{reject}):

- Q is the set of control states
- Σ is the input alphabet
- Γ is the tape alphabet
- b ∈ Γ is the blank symbol, and Σ ⊆ Γ-{b}
- δ: Q × Γ → Q × Γ × {L, R}
- q₀ is the start state
- q_{accept} is the accept state
- q_{reject} is the reject state
TM State Transitions

\[ \delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{\text{L, R}\} \] represents

\[ \delta(q, \gamma) = (q', \gamma', m) \] where

- \( q \) is the current control state
- \( \gamma \) is the current symbol under the head
- \( q' \) is the next control state
- \( m \in \{\text{L, R}\} \) is the direction of head motion
Decider vs. Transducer

- The Sipser version assumes the TM is behaving as a **decider**, determining whether a string is in a language.

- A TM can also behave as a **transducer**, producing an output string from an input one.

- Such a TM computes a function.

- In either case, there is no result until the machine reaches a **halting state**.
Infinite Tape

- Although the tape is infinite in principle, it can only contain a finite number of non-blank characters at any one time.

- In the Sipser version, the tape is only unbounded on the right end and the left end is fixed.
Other Variations

- The tape is unbounded on both ends.
- The head can stay in the same position, rather than being required to move
  \{L, R, N\} or \{L, R, S\} rather than \{L, R\}.
  \(N = \text{no move}, \ S = \text{stationary}\)
- The control can write a symbol or move, but not both.
- There are multiple tracks on one tape.
- There is more than one tape. The input is on one tape and others are used for working memory.
- Two-dimensional “tape”.
- Other conventions regarding delimiting input.
My Favorite: Two-Stack Model

- Growth, if needed, occurs around the head, which is understood as being between the two stacks.
- Messy issues with growth, tape boundaries, and blanks are avoided.
The 5-Tuple Representation of $\delta$

$\delta(q, \gamma) = (q', \gamma', m)$ is represented by a list of 5-tuples of the form

$q, \gamma \rightarrow q', \gamma', m$
State Diagram Representation

- $q, \gamma \rightarrow q', \gamma', m$ is shown as an arc in a labeled directed graph.

JFLAP:

(# is blank)
Example $L = \{ a^n \mid n > 0 \text{ is a power of 2} \}$
The following TM decides $L$. 
Example Configurations for L

Starting on 16 a’s

q3 and q4 mark off every other a replacing it with #, effectively dividing by 2

q5 returns to left end.

q3 and q4 again mark off every other a replacing it with #, effectively dividing by 2
Example Configurations for L

q5 returns to left end.

q3 and q4 again mark off every other a replacing it with #, effectively dividing by 2

q5 returns to left end.

q3 and q4 again mark off every other a replacing it with #, effectively dividing by 2
Example Configurations for L

Make a pass to see if any a’s left over. If not, **accept**.

With 15 a’s, an odd number of a’s is detected on first pass in q4, and input rejected.

With 12 a’s, an odd number of a’s won’t be detected until the third pass.
Multiple Run

Single **a** case is detected by q0 to q1 to q2.
Other TM decidable languages
How would you program a TM for these?

• \{x \in \{0, 1, 2\}^* \mid \#_0(x) = \#_1(x) = \#_2(x) \}\}
• All palindromes over any alphabet
• \{a^n b^n c^n \mid n \geq 0\}
• \{ww \mid w \in \{0, 1\}^*\}
• \{1^p \mid p \text{ is prime}\}
• \{1^p \mid p \text{ is not prime}\}
• Any regular language.
• Any context-free language.
The Church-Turing Thesis

- Effectively-computable functions are the same as Turing-computable ones.
Turing’s Argument that every effectively computable function is computable by a Turing machine

- Everyone should read the prose parts (possibly skipping some of the notation) from the original source. (It is only a few pages long.)

- Here are on-line versions:
  
  
Excerpts from Turing’s Paper

We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions $q_1, q_2, \ldots, q_n$ which will be called “$m$-configurations”. The machine is supplied with a “tape” (the analogue of paper) running through it, and divided into sections (called “squares”) each capable of bearing a “symbol”. At any moment there is just one square, say the $r$-th, bearing the symbol $\xi(r)$ which is “in the machine”. We may call this square the “scanned square”. The symbol on the scanned square may be called the “scanned symbol”. The “scanned symbol” is the only one of which the machine is, so to speak, “directly aware”. However, by altering its $m$-configuration the machine can effectively remember some of the symbols which it has “seen” (scanned) previously. The possible behaviour of the machine at any moment is determined by the $m$-configuration $q_n$ and the scanned symbol $\xi(r)$. This pair $q_n, \xi(r)$ will be called the “configuration”: thus the configuration determines the possible behaviour of the machine.
In some of the configurations in which the scanned square is blank (i.e. bears no symbol) the machine writes down a new symbol on the scanned square: in other configurations it erases the scanned symbol. The machine may also change the square which is being scanned, but only by shifting it one place to right or left. In addition to any of these operations the $m$-configuration may be changed. Some of the symbols written down will form the sequence of figures which is the decimal of the real number which is being computed. The others are just rough notes to “assist the memory”. It will only be these rough notes which will be liable to erasure.

It is my contention that these operations include all those which are used in the computation of a number. The defence of this contention will be easier when the theory of the machines is familiar to the reader. In the next section I therefore proceed with the development of the theory and assume that it is understood what is meant by “machine”, “tape”, “scanned”, etc.
Why accept Turing’s concept?

- Turing gave a careful informal argument as to why his machine model captured the essence of computability.

- Other models of computability (a dozen or so) are provably equivalent to Turing’s machines.

- No one has come up with a function that is intuitively computable that could be proved to be not computable on a Turing machine.
Turing’s “Executive Summary”

9. The extent of the computable numbers.

No attempt has yet been made to show that the “computable” numbers include all numbers which would naturally be regarded as computable. All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically. The real question at issue is “What are the possible processes which can be carried out in computing a number?”

The arguments which I shall use are of three kinds.

(a) A direct appeal to intuition.

(b) A proof of the equivalence of two definitions (in case the new definition has a greater intuitive appeal).

(c) Giving examples of large classes of numbers which are computable.

Once it is granted that computable numbers are all “computable”, several other propositions of the same character follow. In particular, it follows that, if there is a general process for determining whether a formula of the Hilbert function calculus is provable, then the determination can be carried out by a machine.
Turing’s concept cannot be proved

• In order to prove that Turing’s model captures effective computability, we’d need another model, say an X machine, for computability that is more obviously correct, then prove that anything computable by an X machine could be computed by a Turing machine.

• But then there would remain the question of whether the X machine completely captured the notion of computability.

• This could lead to infinite regress (X machine, Y machine, Z machine, …), still with no final proof.
Turing’s idea could conceivably be disproved.

- This could be done by devising a function that is intuitively computable, then proving that it cannot be computed by a Turing machine.

- This has not been done to satisfaction of the general community.

- However, there is work in this direction:
Turing Computability Hedge

- To put arguments about computability on a precise formal basis, one can add the qualification “Turing” in front of “computability”.

- When we say “computability” here, Turing computability is what we mean.
Gödel on Turing on Computability
(from Oron Shagrir, 2004)

Quoting Gödel: “The greatest improvement was made possible through the precise definition of the concept of finite procedure, which plays a decisive role in these results. There are several different ways of arriving at such a definition, which, however, all lead to exactly the same concept. The most satisfactory way, in my opinion, is that of reducing the concept of finite procedure to that of a machine with a finite number of parts, as has been done by the British mathematician Turing.” [Gödel 1951, pp. 304–305]
Gödel on Turing on Computability
(from Oron Shagrir, 2004)

The preceding is the approach Gödel recommends in his 1934 conversation with Church, in which he rejects Church’s proposal as “thoroughly unsatisfactory.”

Gödel suggests to Church that “it might be possible, in terms of an effective calculability as an undefined notion, to state a set of axioms which would embody the generally accepted properties of this notion, and to do something on that basis.”
Emil Post

- Post (1897-1954) may have been the first to prove completeness of the propositional calculus.

- In 1936, independently of Turing, he developed his own type of machine similar to Turing’s and conjectured that it was equivalent in power to Gödel’s recursive functions.

- For this and related models, see: http://en.wikipedia.org/wiki/Post-Turing_machine
Hao Wang

• Hao Wang (1921-1955), a student of Quine, introduced in 1954 the **B-machine** model which could **not erase symbols**, only write them, yet was equivalent in power to the Turing machine.

• [How is this possible?]

• He was the first to use a program-like **statement sequence**.
Wang’s B-Machine

As defined by Wang (1954) the B-machine has at its command only 4 instructions:

(1) → : Move tape-scanning head one tape square to the right (or move tape one square left), then continue to next instruction in numerical sequence;

(2) ← : Move tape-scanning head one tape square to the left (or move tape one square right), then continue to next instruction in numerical sequence;

(3) * : In scanned tape-square print mark * then go to next instruction in numerical sequence;

(4) Cn: Conditional "transfer" (jump, branch) to instruction "n": If scanned tape-square is marked then go to instruction "n" else (if scanned square is blank) continue to next instruction in numerical sequence.
Joachim Lambek’s “Infinite Abacus” (1961)

- This machine used *registers* holding *natural numbers* instead of a tape holding symbols.

- This model is used extensively in logic texts by George Boolos and Richard Jeffrey, and was recently mentioned by Daniel Dennett.

- A host of similar models appeared around the same time, or in the decade after.
  - Counter machines (Shepherdson and Sturgis)
  - Register machines
  - RAM (Random-Access Machine)
  - RASP (Random-Access, Stored-Program) machine

  http://en.wikipedia.org/wiki/Register_machine
Universal Turing machines

- A universal Turing machine is one that can simulate any other Turing machine.

- The alphabet of a universal TM is fixed.

  So we **encode** the alphabet of the TM being simulated.

- The program of the simulated TM can be arbitrarily large.

  So we **encode and store the program** of the simulated TM on the universal machine’s tape, **as well as store any other working information** needed to carry out the simulation. Extra tapes can be used to simplify the representation.

- The universal machine would “jockey back and forth” between the program and the simulated tape, leaving markers, etc. in the form of tape symbols to keep track of current state and head position.
From each line of form \( N_1 \) let us form an expression \( q_i S_j S_k L q_m \); from each line of form \( N_2 \) we form an expression \( q_i S_j S_k R q_m \); and from each line of form \( N_3 \) we form an expression \( q_i S_j S_k N q_m \).

Let us write down all expressions so formed from the table for the machine and separate them by semi-colons. In this way we obtain a complete description of the machine. In this description we shall replace \( q_i \) by the letter “\( D \)” followed by the letter “\( A \)” repeated \( i \) times, and \( S_j \) by “\( D \)” followed by “\( C \)” repeated \( j \) times. This new description of the machine may be called the standard description (S.D). It is made up entirely from the letters “\( A \)” , “\( C \)” , “\( D \)” , “\( L \)” , “\( R \)” , “\( N \)” , and from “;”.

If finally we replace “\( A \)” by “\( 1 \)” , “\( C \)” by “\( 2 \)” , “\( D \)” by “\( 3 \)” , “\( L \)” by “\( 4 \)” , “\( R \)” by “\( 5 \)” , “\( N \)” by “\( 6 \)” , and “;” by “\( 7 \)” we shall have a description of the machine in the form of an arabic numeral. The integer represented by this numeral may be called a description number (D.N) of the machine. The D.N determine the S.D and the structure of the machine uniquely. The machine whose D.N is \( n \) may be described as \( \mathcal{M}(n) \).
A number which is a description number of a circle-free machine will be called a satisfactory number. In § 8 it is shown that there can be no general process for determining whether a given number is satisfactory or not.

6. The universal computing machine.

It is possible to invent a single machine which can be used to compute any computable sequence. If this machine $\mathcal{U}$ is supplied with a tape on the beginning of which is written the S.D of some computing machine $\mathcal{M}$, then $\mathcal{U}$ will compute the same sequence as $\mathcal{M}$. In this section I explain in outline the behaviour of the machine. The next section is devoted to giving the complete table for $\mathcal{U}$. 
A Universal Turing machine

Examples of UTMs

• The Hopcroft & Ullman machine, presented in the first edition of their textbook (1969), has 40 states and 12 tape symbols, used to simulate machines with 2 tape symbols only. (This was not the first UTM.)

• Machines with more than 2 tape symbols can be transformed into ones with only 2 by encoding the symbols into binary.

• The state table of this machine is 3 pages of a textbook.

• See: http://www.rdrop.com/~half/General/UTM/UTMStateTable.html
A small portion of a Universal TM State Table

http://www.rdrop.com/~half/General/UTM/UTMStateTable.html

Correct State Table for Hopcroft & Ullman's Universal Turing Machine

Corrected transitions are **boldface**.

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>c</th>
<th>L</th>
<th>R</th>
<th>b</th>
<th>(m0)</th>
<th>(m1)</th>
<th>(mc)</th>
<th>(mL)</th>
<th>(mR)</th>
<th>(mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_b</td>
<td>C_b,0</td>
<td>C_b,1</td>
<td>C_b,c</td>
<td>C_b,L</td>
<td>C_b,R</td>
<td>C_b,0</td>
<td>C_b,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C_b,1</td>
</tr>
<tr>
<td>C_0</td>
<td>C_0,0</td>
<td>C_0,1</td>
<td>C_0,c</td>
<td>C_0,L</td>
<td>C_0,R</td>
<td>C_0,0</td>
<td>C_0,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_1</td>
<td>C_1,0</td>
<td>C_1,1</td>
<td>C_1,c</td>
<td>C_1,L</td>
<td>C_1,R</td>
<td>C_1,0</td>
<td>C_1,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D_b</td>
<td>V,0</td>
<td>V,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D_0</td>
<td>D_0,0</td>
<td>D_0,1</td>
<td>D_0,c</td>
<td>D_0,L</td>
<td>D_0,R</td>
<td>D_0,0</td>
<td>D_0,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D_1</td>
<td>D_1,0</td>
<td>D_1,1</td>
<td>D_1,c</td>
<td>D_1,L</td>
<td>D_1,R</td>
<td>D_1,0</td>
<td>D_1,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>E,0</td>
<td>E,1</td>
<td>F,c</td>
<td>E,L</td>
<td>E,R</td>
<td>E,0</td>
<td>E,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>E,0</td>
<td>E,1</td>
<td>G,c</td>
<td>E,L</td>
<td>E,R</td>
<td>E,0</td>
<td>E,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>E,0</td>
<td>E,1</td>
<td>H,c</td>
<td>E,L</td>
<td>E,R</td>
<td>E,0</td>
<td>E,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A 34-state, 3-tape Universal Machine in JFLAP
Zoomed in on the JFLAP 3-tape machine
Vertical bars separate the actions on different tapes.
Other UTMs

- Marvin Minsky, 1962: 4 state, 7 symbol
- (The state x symbol product is sometimes used as a measure to be reduced.)
Encoding Matters

- Obviously a big aspect of universal machine compactness is the encoding of the simulated machine being used.
Universal TM’s in Cellular Automata (CA)

- A cellular automaton is another computational model, based on an infinite array of DFAs (in 1 or more dimensions).

- It may have been first proposed and studied by Stanislaw Ulam (studying crystal growth) and John Von Neumann, who also proposed how such an automaton can self-reproduce.

- The best known model is probably John Conway’s “Game of Life”.

- Universal Turing machines have been embedded in many such models.
TM embedded in Life
Wolfram’s “Rule 110” CA machine

- Stephen Wolfram, 2002: 2 state, 5 symbol machine (proof controversial)

CA for fun and profit

The Wolfram book contains a new technical result in describing the Turing completeness of the Rule 110 cellular automaton. Very small Turing machines can simulate Rule 110, which Wolfram demonstrates using a 2-state 5-symbol universal Turing machine.

**Wolfram conjectures that a particular 2-state 3-symbol Turing machine is universal.** In 2007, as part of commemorating the book's fifth anniversary, Wolfram's company offered a $25,000 prize for proof that this Turing machine is universal. Alex Smith, a computer science student from Birmingham, UK, won the prize later that year by proving Wolfram's conjecture.

Edward Fredkin’s Digital Physics based on CA ideas

In digital physics, the Fredkin Finite Nature Hypothesis states that ultimately all quantities of physics, including space and time, are discrete and finite. All measurable physical quantities arise from some Planck scale substrate for multiverse information processing. Also, the amount of information in any small volume of spacetime will be finite and equal to a small number of possibilities.

Fredkin’s page: http://www.digitalphilosophy.org/
Fredkin’s Self-Replicating CA

This type of research generally falls into the area known as “Complex Systems”.

Figure 27. Self-replication of a pattern in the Fredkin’s CA model $A_0$. 
TM presentations are often presented informally, Algorithms

- One often uses the Church-Turing thesis to avoid having to present a Turing machine in detail.

- If we can give a description of an algorithm, the thesis says that there is a TM that carries out the same computation.
Programming Languages

- Functions that are computable in most ordinary programming languages (Racket, Java, C++, C, Python, …) are computable by Turing machines, and vice-versa.

- We assume that the platform on which programs in these languages executes has unbounded memory available (just like a Turing machine tape does).
Mutual Simulation

- It is relatively easy to show that all common programming languages can simulate an arbitrary Turing machine.

- It is also possible, but generally tedious, to show that a Turing machine can simulate programs in common languages.
Why do we care?

- In showing that there are functions that are not computable, it may be easier to use a programming language.