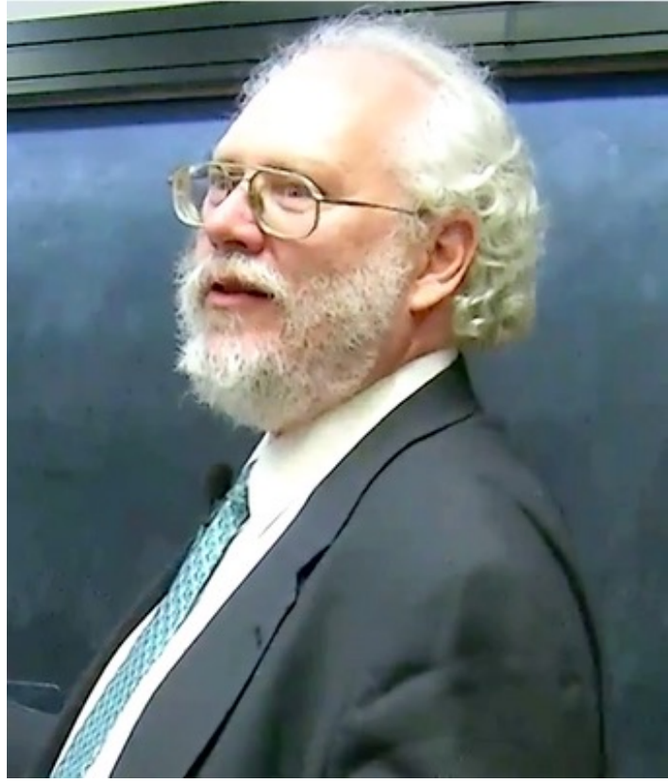
Theory of Computation
Chapter 3 Extension

Universal Turing Machine

Source: Arora & Barak, Computational Complexity
&
Hopcroft, Motwani, & Ullman,
Introduction to Automata Theory, Languages, and Computation

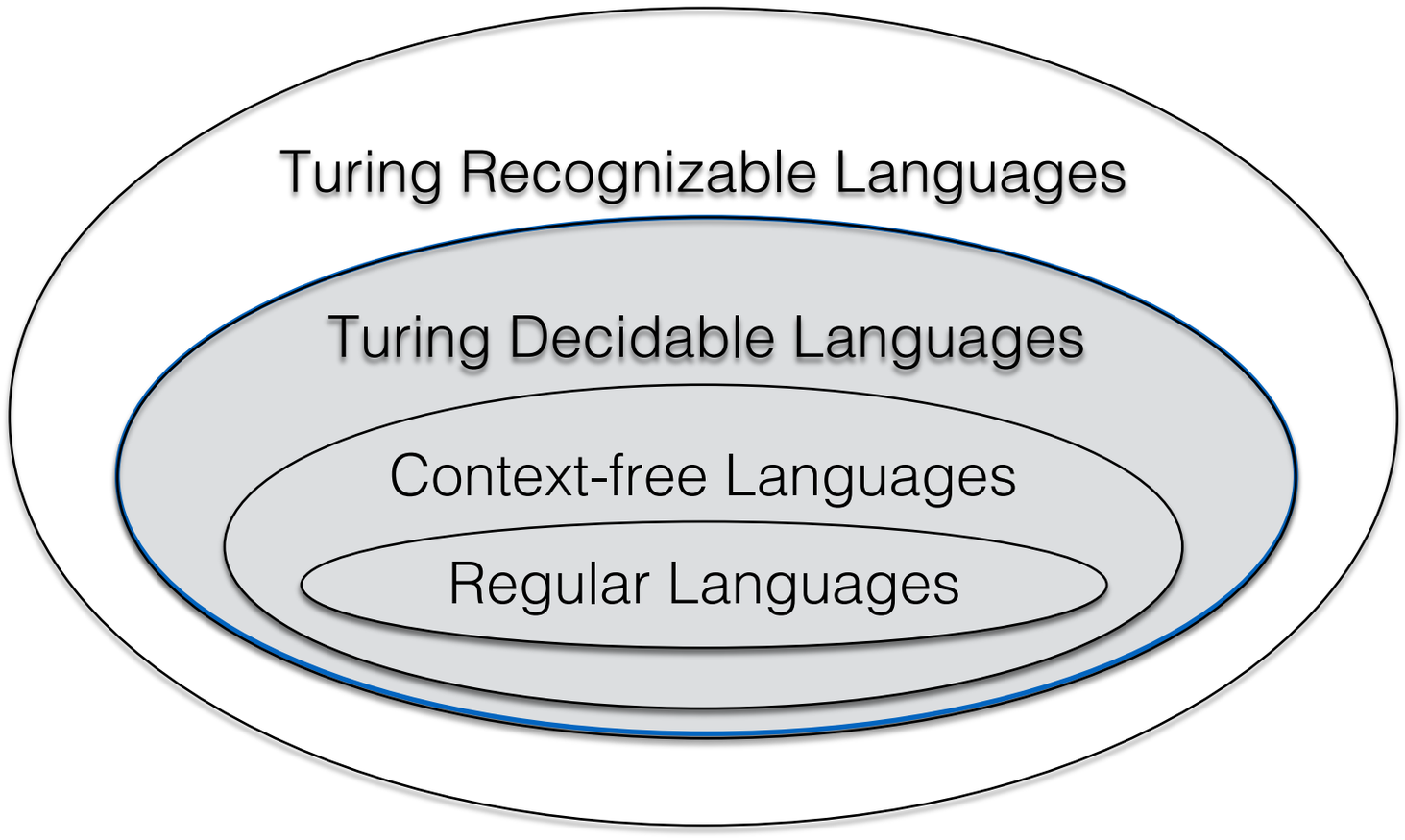
School of Engineering | Computer Science

Peter Shor



- American theoretical computer scientist known for his work on quantum computation
- Devised the Shor's algorithm, a quantum algorithm for factoring exponentially faster than the best currently-known algorithm running on a classical computer
- Professor at MIT since 2005
- Considers post-quantum cryptography to be a solution to the quantum threat

All Languages



Universal Turing Machine

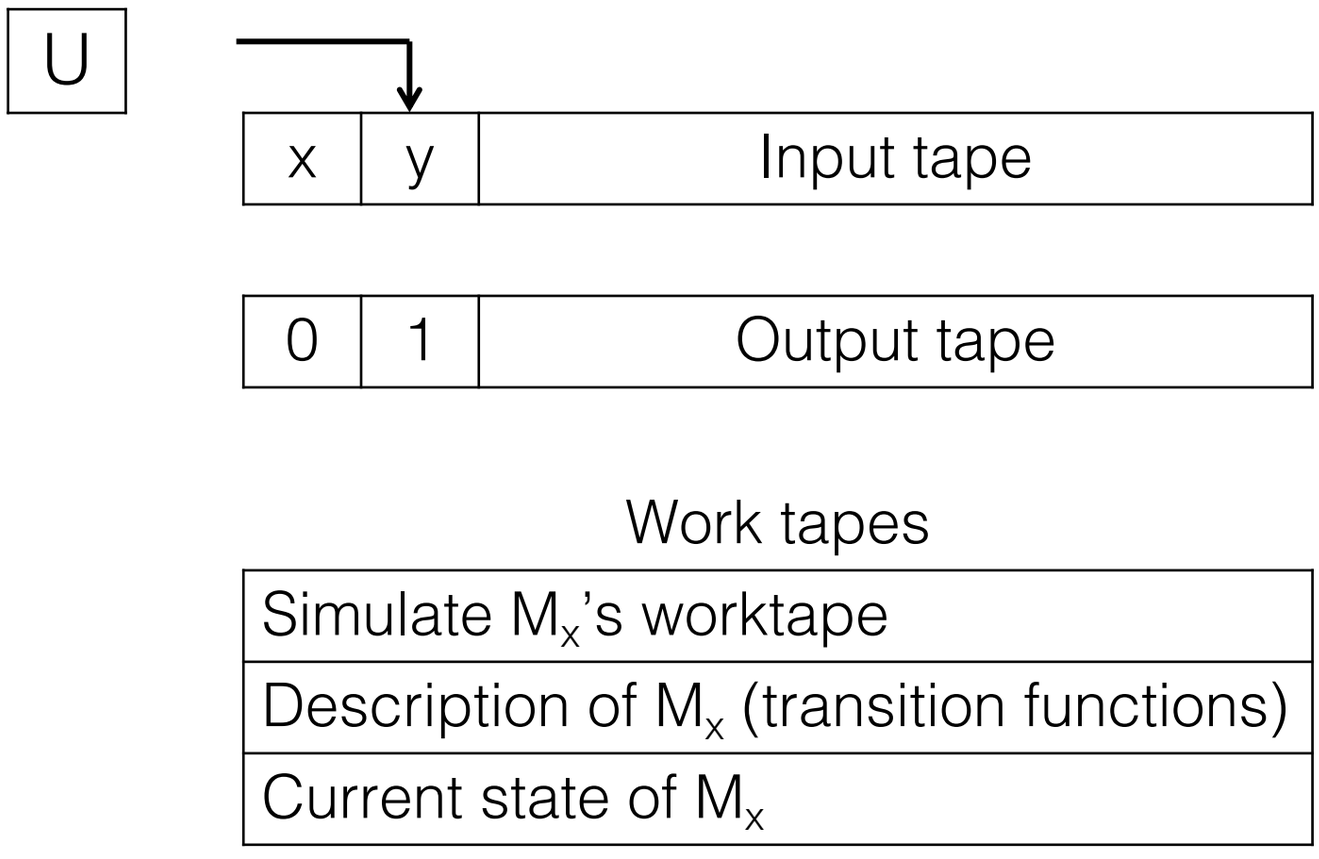
- Idea: There exists a Turing Machine that can simulate other Turing Machines as “software”
- We can encode TM as strings: Write down the entire transition function table, then pick an encoding that satisfies:
  1. Every string in {0, 1}* should represent some TM
  2. Every TM is represented by infinitely many strings (You can append an arbitrary number of 1’s to the end of each string if you wanted to, these would be ignored)
- Notation: Let x = <M> be a binary encoding of a TM M

Universal Turing Machine

- Idea: There exists a Turing Machine that can simulate other Turing Machines as “software”
- We can encode TM as strings:
- Notation: Let $x = <M_x>$ be a binary encoding of a TM M_x
- There exists a Turing Machine U such that for all x, $y \in \{0, 1\}^*$, $U(x, y) = M_x(y)$ (y is the input that M_x will run on, M_x is the TM described by the string x, where $x = < M_x > \in \{0, 1\}^*$)
- Note: If M_x halts on input y in t steps, then U halts on (x, y) in O(t log t) steps.

Universal Turing Machine

- Proof Sketch: Let's construct a TM U. Given x and y, U must output $M_x(y)$. Let's assume M_x uses the binary alphabet and a single tape.



To simulate one computational step of M_x, U scans M_x's current state and transition function to determine the 1) next state, 2) symbol to write, and 3) head movements. These are simulated on the work tapes

Why Turing Machines?

- Why are Turing Machines still relevant?
  - Church-Turing Thesis: Every physically realizable computation device can be simulated by a Turing Machine. (This has not been proven, but it is a general belief.)
  - The strong Church-Turing thesis says a TM can simulate all other computational models efficiently (with polynomial overhead)
  - Essentially, the set of "computable" problems is fully captured by Turing Machines

Why Turing Machines?

- Let's look at the Factoring Problem
  - Input: A product of pq of primes p and q
  - Output: p and q
  - Ex: Input: 15, Output: 3 5
  - It is believed that the factoring problem has no polynomial-time algorithm. (It is so difficult to crack that many encryption schemes today are "secured" assuming this is true (ex: RSA))
  - Thus, there is no TM to solve this problem
  - However, in 1994 Peter Shor showed that a quantum computer could solve the factoring problem efficiently.

Why Turing Machines?

- What are the **possible implications** of solving the factoring problem efficiently?
  1. The Church-Turing Thesis is false
  2. Quantum computers are physically realizable
  3. There exists an efficient, classical algorithm for the factoring problem
- Only time will tell the answer…
- Article on Google‘s Willow quantum processor tackling this problem: Google Willow Quantum Processor

Importance of Turing Machines

- The importance of the Turing Machine:
  1. Notation of general purpose CPU
  2. Captures everything about computing (Church-Turing Thesis)
  3. Formal definition of algorithms
  4. We can formalize the question of “What problems can computers solve?”

Introduction to Reductions (Chapter 5)

Source: Hopcroft, Motwani, and Ullman, Introduction to Automata Theory, Languages and Computation

School of Engineering | Computer Science

Undecidability

- Once we have Turing Machines and we can use those Turing Machines to run other TMs as input, we can start looking at the question of what languages can be defined by any computational device.
- Leads to question: What can computers do?
  - Recognizing strings in a language is a formal way of expressing any problem, and solving a problem is a reasonable representation of what computers can do.

Undecidability

- Let's look at a problem that the computer cannot solve:
  - Will a computer print, "Hello, World"?
  - This might seem like an obvious result in a small program that we are used to, but what about in a very large, complicated program?
  - Can we always determine this????

Undecidability

- What about this code?

```
int exp(int i, n) {
  int ans, j:
  ans = 1;
  for (j=1; j<=n; j++) ans *= i;
  return(ans);
}

main() {
  int n, total, x, y, z;
  scanf("%d", &n);
  total = 3;
  while(1) {
     for(x=1; x<=total-2; x++)
        for(y=1; y <= total-x-1; y++){
           z = total - x - y;
           if(exp(x,n) + exp(y,n) == exp(z,n))
              printf("hello, world\n");  }
     total++;  }
}
```

Can you be sure that "hello, world" will print?

Undecidability

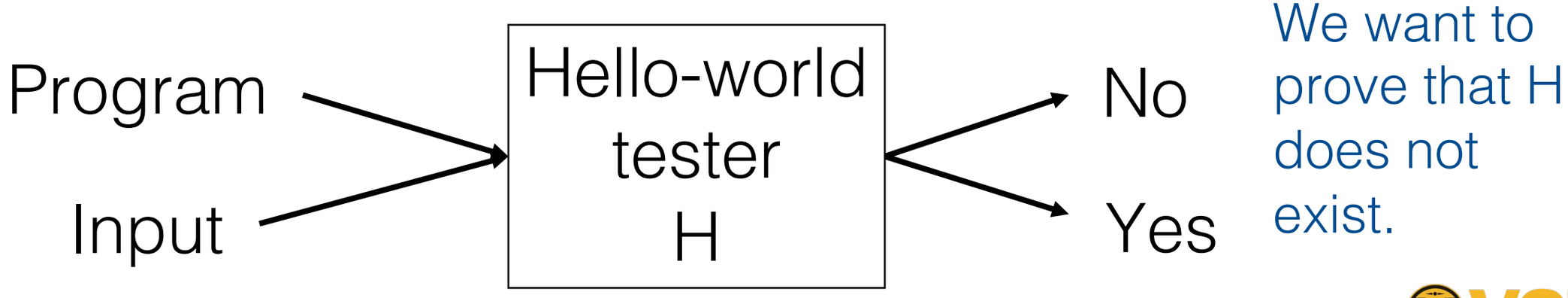
- Here is a problem that the computer cannot solve:
  - In 1900, David Hilbert published 23 problems that were unsolved. He urged the mathematical community to solve them in the upcoming century. The tenth problem listed was proven in 1970 to be undecidable by Yuri Matiyasevich
    - *Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers*.
- What does this mean to us in computer theory?

Undecidability

- Fact: Almost all problems must be undecidable by any system that involves programming.
  - A problem is the membership of a string in a language
  - The number of different languages over any alphabet of more than one symbol is not countable
    - There is no way to assign integers to the languages such that every language has an integer, and every integer is assigned to one language
  - On the other hand, programs, being finite strings over a finite alphabet are countable, thus there are infinitely fewer programs as there are problems

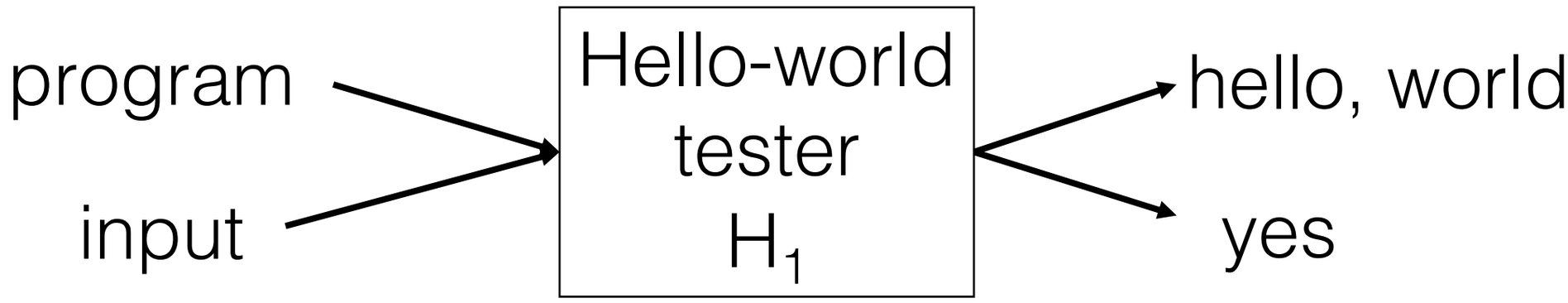
Proof of Undecidability

- To prove a language is undecidable, you need to use a proof by contradiction
  - Assume the language is decidable and prove that it is not
- Let's prove that the code before is undecidable
  - To do this we will start with a decidable program, H

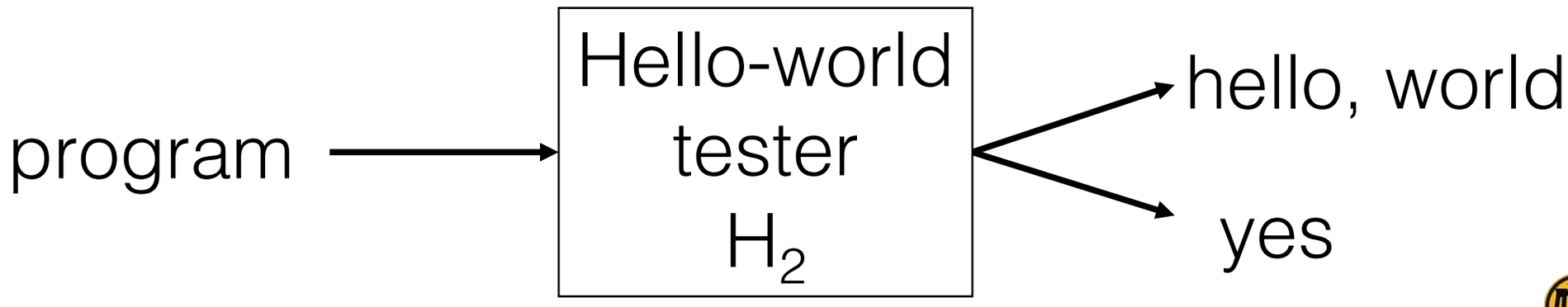


Proof of Undecidability

- We will now modify H to, instead of printing “no”, print “hello, world”, H_1

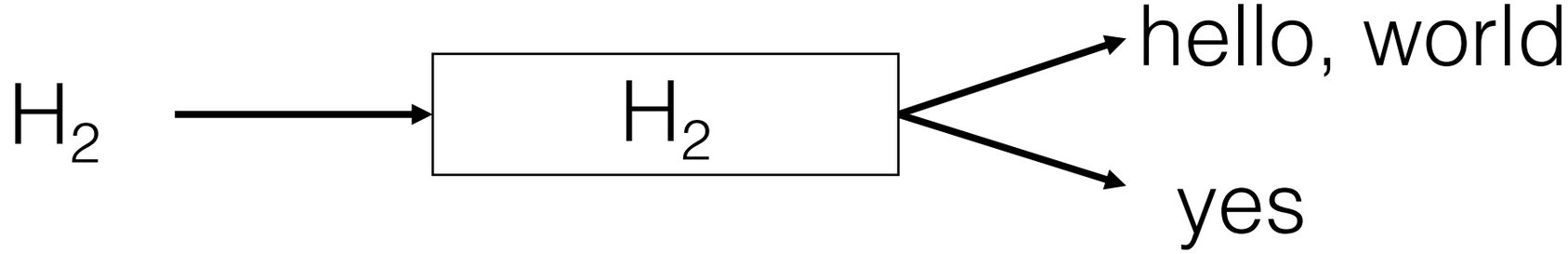


- We will modify it again to restrict the input to just the program, not the program and the input, H_2



Proof of Undecidability

- We will now prove H_2 does not exist, so H_1 does not exist and H does not exist.
  - What happens when H_2 is given itself as an input?

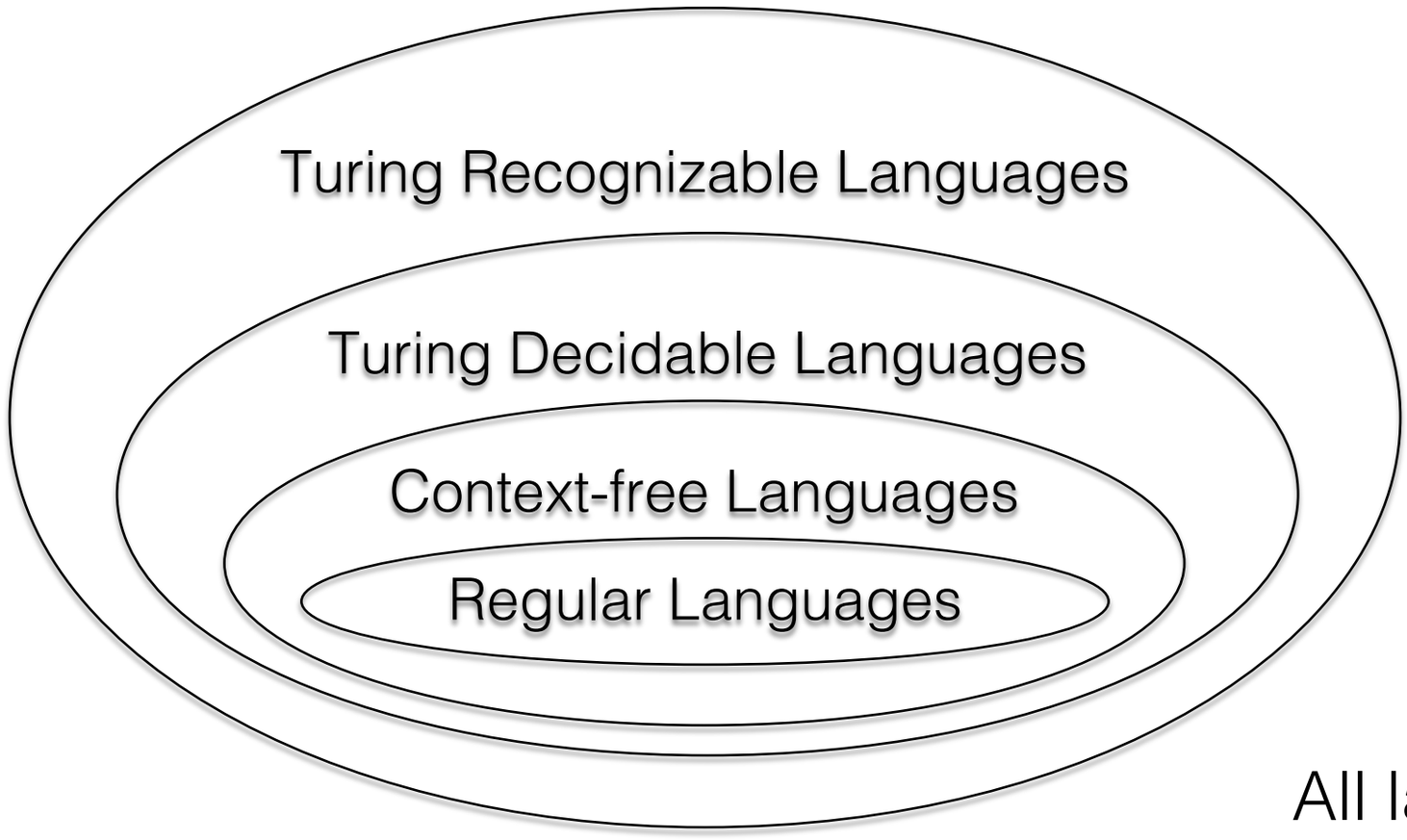


- Suppose that the H_2 box makes the H_2 program generate "yes", thus it is saying that H_2 when given itself will generate "hello, world" as its output.
- However, we just said that H_2 generates "yes" as the output, not "hello, world". Either way we have a problem.

Decidability

- Decidable problems: Problems where we can construct an algorithm to solve it in finite time. A Turing Machine will halt on every input with an accept or reject (Turing Decidable)
- Undecidable problems: Problems where we cannot construct an algorithm that can solve it in finite time. A Turing Machine cannot solve these problems.
- Semi-decidable problems: Problems where the Turing Machine accepts and halts, but also rejects or loops forever (Turing Recognizable)

Decidability



All languages

Try It

- Create an implementation-level description of a Turing Machine that shifts the entire input string one cell to the right and adds a “$” to the left-most cell.

Try It

- Create an implementation-level description of a Turing Machine that shifts the entire input string one cell to the right and adds a “$” to the left-most cell.
  1. Mark the left-most cell with a dot over the symbol.
  2. Scan to the right-most non-empty cell. Copy the value in this cell into the cell on the right.
  3. Move two cells to the left. If you have reached the start cell, move to step 5.
  4. Copy the value in this cell into the cell on the right. Repeat Step 3.
  5. Copy the original value in this cell into the cell on the right. Move one cell to the left and place a $ in that cell.