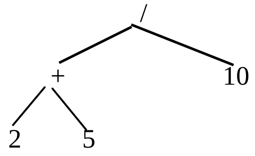
The Three Hour Tour Through Automata Theory

Read Supplementary Materials: The Three Hour Tour Through Automata Theory
Read Supplementary Materials: Review of Mathematical Concepts
Read K & S Chapter 1
Do Homework 1.

Let's Look at Some Problems

```
int alpha, beta;
alpha = 3;
beta = (2 + 5) / 10;
```

(1) **Lexical analysis**: Scan the program and break it up into variable names, numbers, etc.
(2) **Parsing**: Create a tree that corresponds to the sequence of operations that should be executed, e.g.,



(3) **Optimization**: Realize that we can skip the first assignment since the value is never used and that we can precompute the arithmetic expression, since it contains only constants.
(4) **Termination**: Decide whether the program is guaranteed to halt.
(5) **Interpretation**: Figure out what (if anything) it does.

A Framework for Analyzing Problems

We need a single framework in which we can analyze a very diverse set of problems.
The framework we will use is **Language Recognition**

A ***language*** is a (possibly infinite) set of finite length strings over a finite alphabet.

Languages

(1) $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$

$L = \{w \in \Sigma^*: \text{w represents an odd integer}\}$
$= \{w \in \Sigma^*: \text{the last character of w is 1,3,5,7, or 9}\}$
$= (0\cup1\cup2\cup3\cup4\cup5\cup6\cup7\cup8\cup9)^*$
$(1\cup3\cup5\cup7\cup9)$

(2) $\Sigma = \{(,)\}$

$L = \{w \in \Sigma^*: \text{w has matched parentheses}\}$
= the set of strings accepted by the grammar:

$S \rightarrow (S)$
$S \rightarrow SS$
$S \rightarrow \varepsilon$

(3) L = {w: w is a sentence in English}

Examples: Mary hit the ball.
Colorless green ideas sleep furiously.
The window needs fixed.

(4) L = {w: w is a C program that halts on all inputs}

Encoding Output in the Input String

(5) Encoding multiplication as a single input string

L = {w of the form: <integer>x<integer>=<integer>, where <integer> is any well formed integer, and the third integer is the product of the first two}

12x9=108 12=12 12x8=108

(6) Encoding prime decomposition

L = {w of the form: <integer1>/<integer2>,<integer3> …, where integers 2 - n represent the prime decomposition of integer 1.

15/3,5 2/2

More Languages

(7) Sorting as a language recognition task:

$L = \{w_1 \# w_2 : \exists n \geq 1,$

w_1 is of the form int_1, int_2, … int_n,

w_2 is of the form int_1, int_2, … int_n, and

w_2 contains the same objects as w_1 and w_2 is sorted}

Examples:

1,5,3,9,6#1,3,5,6,9 $\in$ L

1,5,3,9,6#1,2,3,4,5,6,7 $\notin$ L

(8) Database querying as a language recognition task:

L = {d # q # a:

d is an encoding of a database,

q is a string representing a query, and

a is the correct result of applying q to d}

Example:

(name, age, phone), (John, 23, 567-1234) (Mary, 24, 234-9876)# (select name age=23) # (John) $\in$ L

The Traditional Problems and their Language Formulations are Equivalent

By equivalent we mean:

If we have a machine to solve one, we can use it to build a machine to do the other using just the starting machine and other functions that can be built using a machine of equal or lesser power.

Consider the multiplication example:

L = {w of the form:
<integer>x<integer>=<integer>, where
<integer> is any well formed integer, and
the third integer is the product of the first two}

Given a multiplication machine, we can build the language recognition machine:

Given the language recognition machine, we can build a multiplication machine:

A Framework for Describing Languages

Clearly, if we are going to work with languages, each one must have a finite description.

Finite Languages: Easy. Just list the elements of the language.
L = {June, July, August}

Infinite Languages: Need a finite description.

Grammars let us use recursion to do this.

Grammars 1

(1) The Language of Matched Parentheses

S → (S)
S → SS
S → ε

(2) The Language of Odd Integers

S → 1
S → 3
S → 5
S → 7
S → 9
S → 0 S
S → 1 S
S → 2 S
S → 3 S
S → 4 S
S → 5 S
S → 6 S
S → 7 S
S → 8 S
S → 9 S

Grammars 2

S → O
S → A O
A →A D
A → D
D → O
D → E
O → 1
O → 3
O → 5
O → 7
O → 9
E→ 0
E→ 2
E→ 4
E→ 6
E→ 8

Grammars 3

(3) The Language of Simple Arithmetic Expressions

S → <exp>
<exp> → <number>
<exp> → (<exp>)
<exp> → - <exp>
<exp> → <exp> <op> <exp>
<op> → + | - | * | /
<number> → <digit>
<number> → <digit> <number>
<digit > → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Grammars as Generators and Acceptors

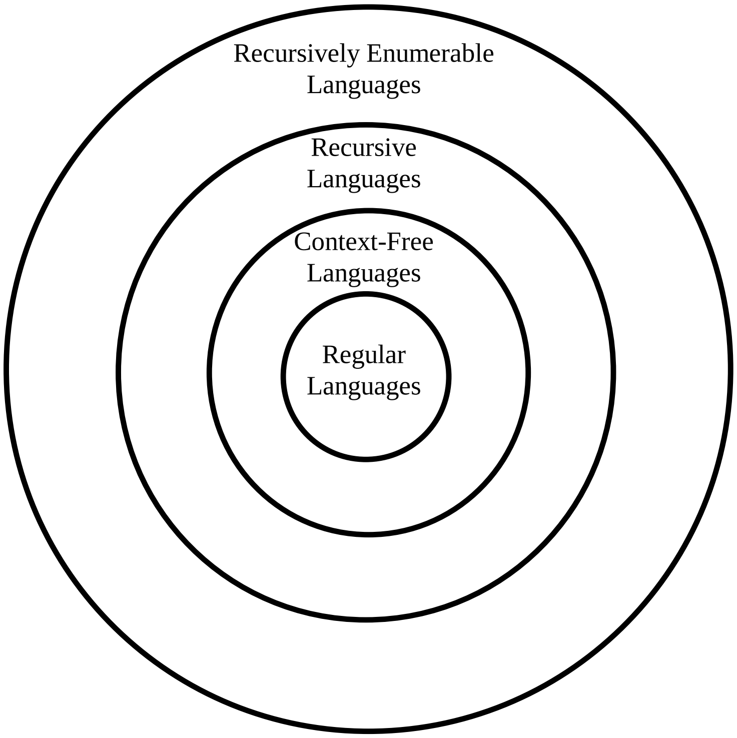
Top Down Parsing

4 + 3

Bottom Up Parsing

4 + 3

The Language Hierarchy



Regular Grammars

In a regular grammar, all rules must be of the form:

<one nonterminal> → <one terminal> or ε

or

<one nonterminal> → <one terminal><one nonterminal>

So, the following rules are okay:

S → ε
S → a
S → aS

But these are not:

S → ab
S → SS
aS → b

Regular Expressions and Languages

Regular expressions are formed from ∅ and the characters in the target alphabet, plus the operations of:

- Concatenation: $\alpha\beta$ means α followed by β
- Or (Set Union): $\alpha\cup\beta$ means α Or (Union) β
- Kleene *: α^* means 0 or more occurrences of α concatenated together.
- At Least 1: α^+ means 1 or more occurrences of α concatenated together.
- (): used to group the other operators

Examples:

(1) Odd integers:

$(0\cup1\cup2\cup3\cup4\cup5\cup6\cup7\cup8\cup9)^*(1\cup3\cup5\cup7\cup9)$

(2) Identifiers:

$(A\text{-}Z)^+((A\text{-}Z)\cup(0\text{-}9))^*$

(3) Matched Parentheses

Context Free Grammars

(1) The Language of Matched Parentheses

S → (S)
S → SS
S → ε

(2) The Language of Simple Arithmetic Expressions

S → <exp>
<exp> → <number>
<exp> → (<exp>)
<exp> → - <exp>
<exp> → <exp> <op> <exp>
<op> → + | - | * | /
<number> → <digit>
<number> → <digit> <number>
<digit > → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Not All Languages are Context-Free

English: S → NP VP
NP → the NP1 | NP1
NP1 → ADJ NP1 | N
N → boy | boys
VP →V | V NP
V → run | runs

What about “boys runs”

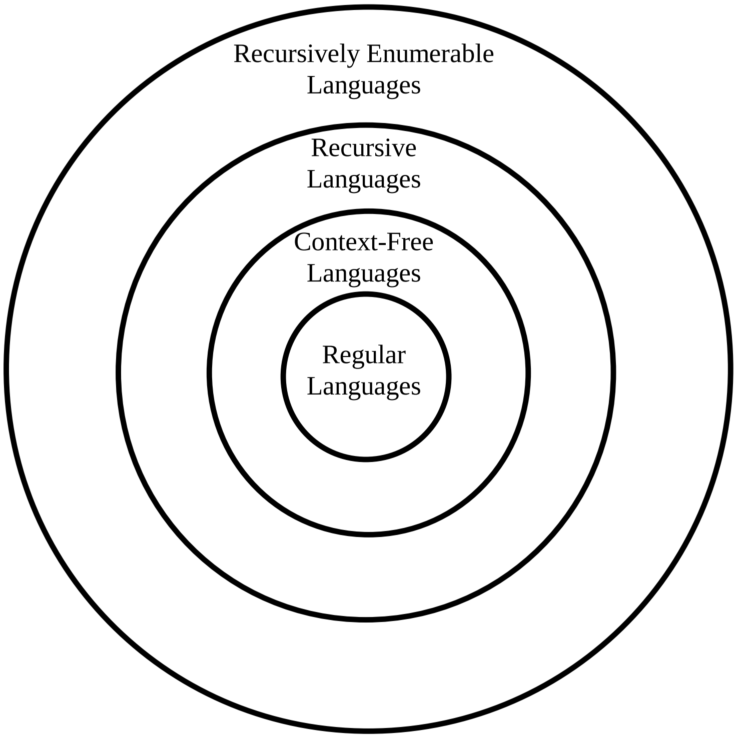
A much simpler example: $a^n b^n c^n$, $n \geq 1$

Unrestricted Grammars

Example: A grammar to generate all strings of the form $a^n b^n c^n$, $n \geq 1$

S → aBSc
S → aBc
Ba → aB
Bc → bc
Bb → bb

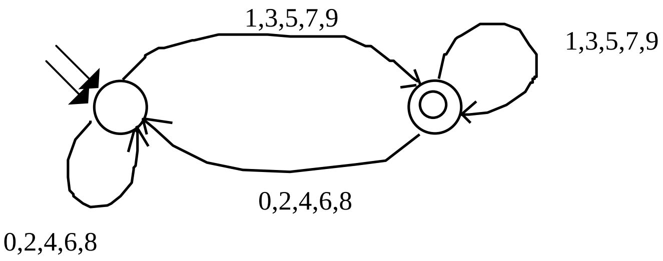
The Language Hierarchy



A Machine Hierarchy

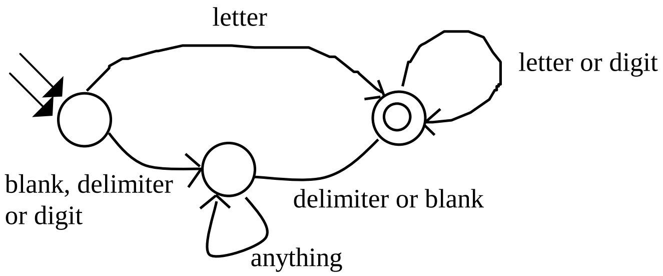
Finite State Machines 1

An FSM to accept odd integers:



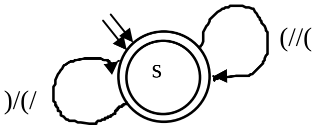
Finite State Machines 2

An FSM to accept identifiers:



Pushdown Automata

A PDA to accept strings with balanced parentheses:

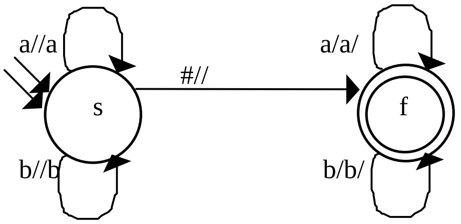


Example: (())()

Stack:

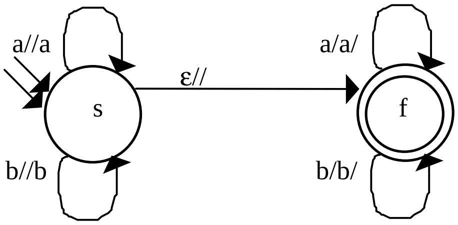
Pushdown Automaton 2

A PDA to accept strings of the form $w\#w^R$:



A Nondeterministic PDA

A PDA to accept strings of the form ww^R

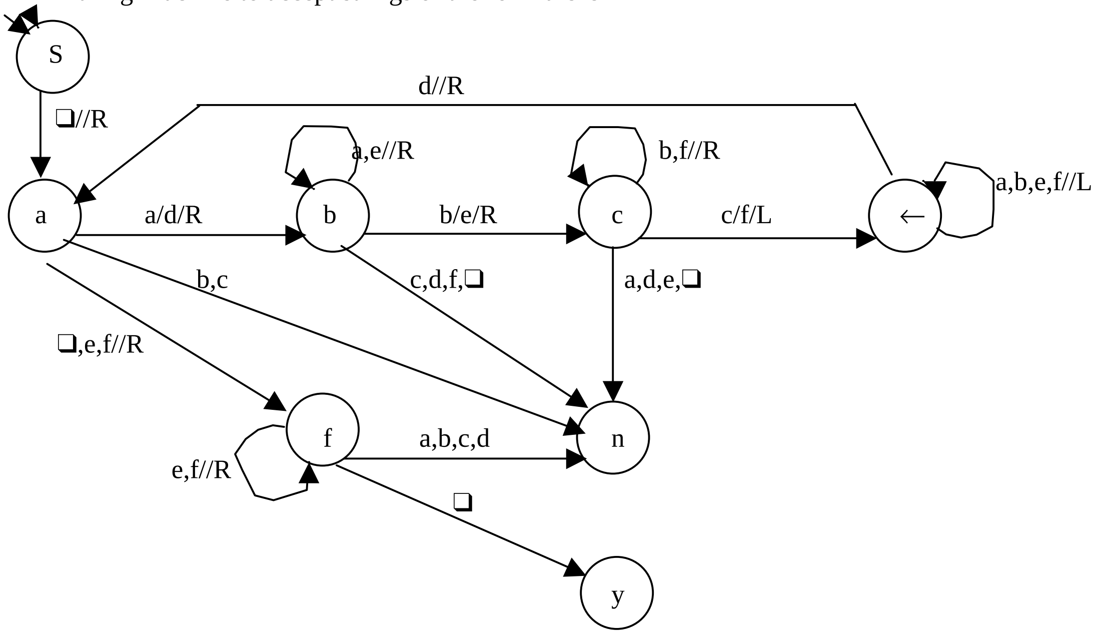


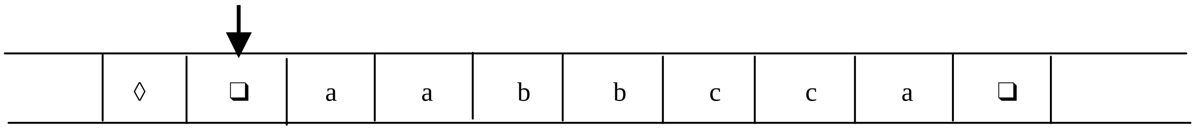
PDA 3

A PDA to accept strings of the form $a^nb^nc^n$

Turing Machines

A Turing Machine to accept strings of the form $a^nb^nc^n$



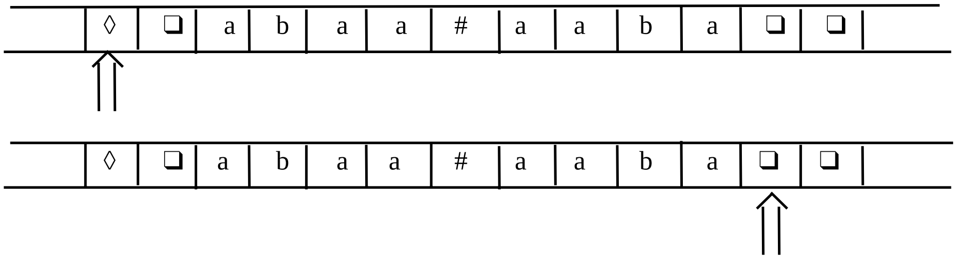


A Two Tape Turing Machine

A Turing Machine to accept {w#w^R}

	◊	❑	a	b	a	a	#	a	a	b	a	❑	❑

A Two Tape Turing Machine to do the same thing



◊	❑	a	b	a	a	#	a	a	b	a	❑	❑

◊	❑	a	b	a	a	#	a	a	b	a	❑	❑

Simulating k Tapes with One

A multitrack tape:

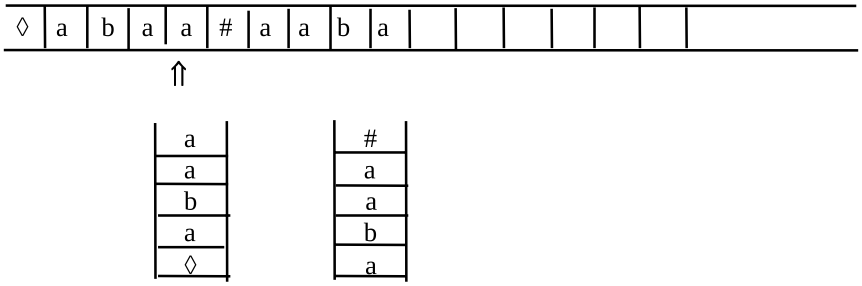
◊	◊	❑	a	b	a	❑	❑	❑	❑
	0	0	1	0	0	0	0		
	◊	a	b	b	a	b	a		
	0	1	0	0	0	0	0		

Can be encoded on a single tape with an alphabet consisting of symbols corresponding to :

{{◊,a,b,#,❑} x {0,1} x
{◊,a,b,#,❑} x {0,1}}

Example: 2nd square: (❑,0,a,1))

Simulating a Turing Machine with a PDA with Two Stacks



◊	a	b	a	a	#	a	a	b	a						

a	#
a	a
b	a
a	b
◊	a

The Universal Turing Machine
Encoding States, Symbols, and Transitions

Suppose the input machine M has 5 states, 4 tape symbols, and a transition of the form:

(s,a,q,b), which can be read as:

in state s, reading an a, go to state q, and write b.

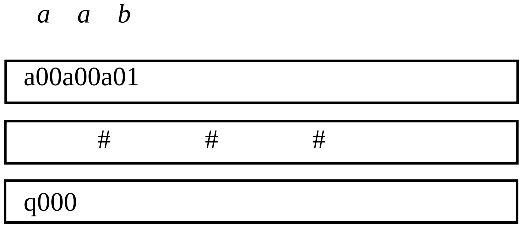
We encode this transition as:

q000,a00,q010,a01

A series of transitions that describe an entire machine will look like

q000,a00,q010,a01#q010,a00,q000,a00

The Universal Turing Machine



Church's Thesis
(Church-Turing Thesis)

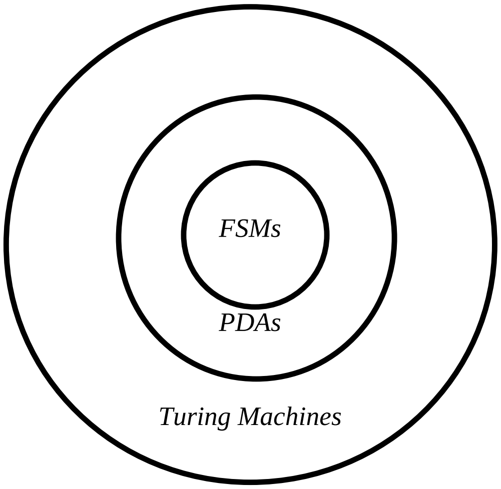
An algorithm is a formal procedure that halts.

The Thesis: Anything that can be computed by any algorithm can be computed by a Turing machine.

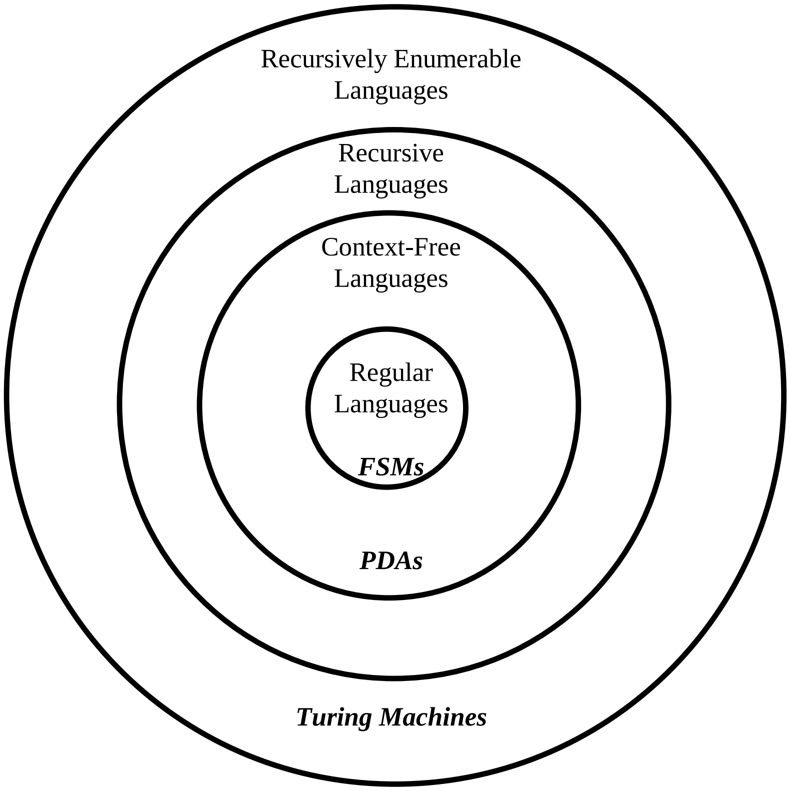
Another way to state it: All "reasonable" formal models of computation are equivalent to the Turing machine. This isn't a formal statement, so we can't prove it. But many different computational models have been proposed and they all turn out to be equivalent.

Example: unrestricted grammars

A Machine Hierarchy



Languages and Machines



Where Does a Particular Problem Go?

Showing what it is -- generally by construction of:

- A grammar, or a machine

Showing what it isn't -- generally by contradiction, using:

- Counting
  Example: $a^n b^n$
- Closure properties
- Diagonalization
- Reduction

Closure Properties

Regular Lanugages are Closed Under:

- Union
- Concatenation
- Kleene closure
- Complementation
- Reversal
- Intersection

Context Free Languages are Closed Under:

- Union
- Concatenation
- Kleene Closure
- Reversal
- Intersection with regular languages

Etc.

Using Closure Properties

Example:
$L = \{a^n b^m c^p : n \neq m \text{ or } m \neq p\}$ is not deterministic context-free.

Two theorems we'll prove later:

Theorem 3.7.1: The class of deterministic context-free languages is closed under complement.

Theorem 3.5.2: The intersection of a context-free language with a regular language is a context-free language.

If L were a deterministic CFL, then the complement of L (L') would be a deterministic CFL.

But $L' \cap a^*b^*c^* = \{a^n b^n c^n\}$, which we know is not context-free, much less deterministic context-free. Thus a contradiction.

Diagonalization

The power set of the integers is not countable.
Imagine that there were some enumeration:

	1	2	3	4	5
Set 1	1				
Set 2		1		1	
Set 3	1		1		
Set 4		1			
Set 5	1	1	1	1	1

But then we could create a new set

New Set				1	

But this new set must necessarily be different from all the other sets in the supposedly complete enumeration. Yet it should be included. Thus a contradiction.

More on Cantor

Of course, if we're going to enumerate, we probably want to do it very systematically, e.g.,

	1	2	3	4	5	6	7
Set 1	1						
Set 2		1					
Set 3	1	1					
Set 4			1				
Set 5	1		1				
Set 6		1	1				
Set 7	1	1	1				

Read the rows as bit vectors, but read them backwards. So Set 4 is 100. Notice that this is the binary encoding of 4.
This enumeration will generate all **finite** sets of integers, and in fact the set of all finite sets of integers is countable.
But when will it generate the set that contains all the integers except 1?

The Unsolvability of the Halting Problem

Suppose we could implement

```
HALTS(M,x)
    M: string representing a Turing Machine
    x: string representing the input for M
    If M(x) halts then True
           else False
```

Then we could define

```
TROUBLE(x)
    x: string
    If HALTS(x,x) then loop forever
            else halt
```

So now what happens if we invoke TROUBLE(TROUBLE), which invokes HALTS(TROUBLE,TROUBLE)

If HALTS says that TROUBLE halts on itself then TROUBLE loops. IF HALTS says that TROUBLE loops, then TROUBLE halts.

Viewing the Halting Problem as Diagonalization

First we need an enumeration of the set of all Turing Machines. We'll just use lexicographic order of the encodings we used as inputs to the Universal Turing Machine. So now, what we claim is that HALTS can compute the following table, where 1 means the machine halts on the input:

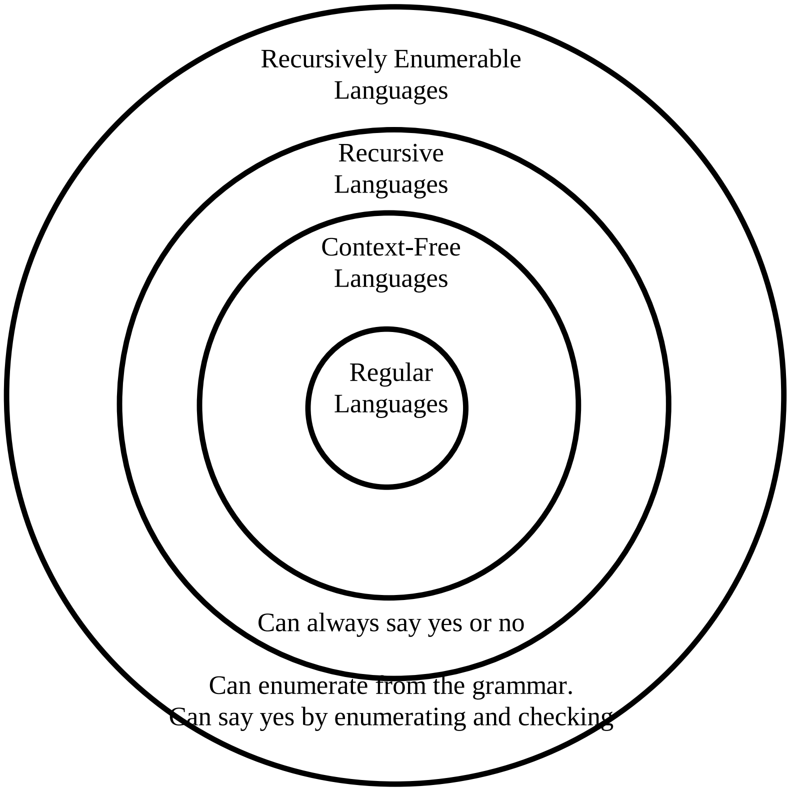
	I1	I2	I3	TROUBLE	I5
Machine 1	1				
Machine 2		1		1	
Machine 3					
TROUBLE			1		1
Machine 5	1	1	1	1	

But we've defined TROUBLE so that it will actually behave as:

TROUBLE			1	1	1

Or maybe HALT said that TROUBLE(TROUBLE) would halt. But then TROUBLE would loop.

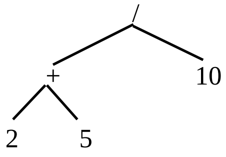
Decidability


Recursively Enumerable Languages
Recursive Languages
Context-Free Languages
Regular Languages
Can always say yes or no
Can enumerate from the grammar.
Can say yes by enumerating and checking

Let's Revisit Some Problems

```
int alpha, beta;
alpha = 3;
beta = (2 + 5) / 10;
```

(1) **Lexical analysis**: Scan the program and break it up into variable names, numbers, etc.
(2) **Parsing**: Create a tree that corresponds to the sequence of operations that should be executed, e.g.,



(3) **Optimization**: Realize that we can skip the first assignment since the value is never used and that we can precompute the arithmetic expression, since it contains only constants.
(4) **Termination**: Decide whether the program is guaranteed to halt.
(5) **Interpretation**: Figure out what (if anything) useful it does.

So What's Left?

- Formalize and Prove Things

- Regular Languages and Finite State Machines
    - FSMs
        - Nondeterminism
        - State minimization
        - Implementation
    - Equivalence of regular expressions and FSMs
    - Properties of Regular Languages
- Context-Free Languages and PDAs
    - Equivalence of CFGs and nondeterministic PDAs
    - Properties of context-free languages
    - Parsing and determinism
- Turing Machines and Computability
    - Recursive and recursively enumerable languages
    - Extensions of Turing Machines
    - Undecidable problems for Turing Machines and unrestricted grammars