Assignment 6 graded

Assignment 7 out later today

Exam 2 graded someday, someday
For Problem 3 on Assignment 6, a worrying number of people made mistakes that show they don’t understand

1. what a *language* is,
2. how *set operations* like intersection work, or
3. *both*

So, let’s travel back in time to Lecture 1.
A **string** is a sequence of characters/symbols.

In a programming language, a string is written with quotation marks, e.g., "*I already know this*".

  Punctuation goes outside the quotation marks because we’re not animals.

In language theory, we omit the quotation marks and use visible characters for spaces, e.g., *No*, *you*, *don't*.

The empty string, $\varepsilon$, is a string of length 0.
A set is an unordered collection of 0 or more objects of any type, e.g.,

- $\emptyset$
- $\{\}$
- $\{0\}$
- $\{0, 1\}$
- $\mathbb{N}$
- $\mathbb{N}_0$
A *language* is a set of strings, e.g.,

∅

{ε}

{a}

{a, b}
A language can be *finite*, i.e., only contain a fixed number of strings, even if that number is large.

Every finite language is regular.

It is also therefore, deterministic context-free, nondeterministic context-free, Turing-decidable, and Turing-recognizable because these are superset of the regular languages.

A language can be *infinite*, i.e., contain an infinite number of strings.

Infinite languages may be regular, deterministic context-free, nondeterministic context-free, Turing-decidable, Turing-recognizable, or none of the above!
If a language contains a very large number of strings – whether it is finite or infinite – we don't write them all out.

We can use an ellipsis, e.g.,

\[ \{a, \text{aa}, \text{aaa}, \ldots\} \]

but this is not very precise. E.g., we might mean the language consisting of strings composed of

one or more \text{a}s, or

either one \text{a} or a prime number of \text{a}s.
Instead, we usually give a description of the set, e.g.,

\[ \{w \mid w \in \{a, b\}^*\} \]

This means “Any string \( w \) such that \( w \) is an element of the Kleene closure of the set \( \{a, b\} \)”.

Or, more simply, “0 or more \textit{as} or \textit{bs} in any order”.

\textbf{Note}: Kleene star (\*) is an \textit{operator} that applies to languages (sets).
When we write \{… | …\}, this is *set-builder notation*. It can involve variables like \(s\), \(w\), \(x\), or \(y\) and characters like \(a\), \(b\), \(c\), …

You can tell these apart because we use letters from the end of the alphabet as variables for strings and substrings; we use letters from the beginning of the alphabet or numbers for characters.

To further distinguish them, both the textbook and I use different fonts for them.

Variables are in an *italic font*.

Characters are in a *fixed-width font*. 
Sometimes we’ve written $\{a^n b^n\}$ as a shorthand when discussing languages, but this is not a complete specification.

If you want to describe a language, you need to give the possible values for the variables:

$$\{a^n b^n \mid n \geq 0\}$$

or, to be more precise,

$$\{a^n b^n \mid n \in \mathbb{N}_0\}$$

since we can’t have non-integer numbers of a character in a string.
This is a description of a language, $L$.

The language is a set containing all strings that are composed of an equal number of (0 or more) $a$s and $b$s followed by 0 or more $c$s. E.g.,

\[
\begin{align*}
\epsilon & \in L & \text{abb} & \not\in L \\
ab & \in L & \text{aab} & \not\in L \\
c & \in L & \text{ac} & \not\in L \\
abccc & \in L & \text{bc} & \not\in L \\
aaabbb & \in L & \text{ba} & \not\in L \\
aaabbbccc & \in L & \text{ababc} & \not\in L
\end{align*}
\]
If a language is regular, we might give the description as a regular expression, which is written without curly brackets, e.g.,

\[a^*\]

This is not a string. This is a *description of a language*. That is, it is a description of a set of strings:

\[\{\varepsilon, a, aa, aaa, aaaa, aaaaa, \ldots\}\]
When we’re being precise, regular expressions are the only time we remove the curly brackets from the description of a language.

The regular expression $a$ means the language $\{a\}$.

The regular expression $a^*$ means that language $\{a\}^*$
$= \{a^i \mid i \geq 0\}$

The regular expression $a \cup b$ means the language $\{a\} \cup \{b\} = \{a, b\}$
Union and intersection are set operations. They can be applied to languages because languages are sets of strings.

They do not apply to strings. Never, ever, ever.
The intersection of two sets is a set containing the elements that are in both sets.

Therefore, the intersection of two languages is the language of strings that are in both of the original languages.

\[ \{a^i \mid i \geq 0\} \cap \{b^i \mid i \geq 0\} = \{\varepsilon\} \]

The only string in both languages is \(\varepsilon\)

\[ a^0 = \varepsilon \]

\[ b^0 = \varepsilon \]

\[ \{a^i \mid i \geq 1\} \cap \{b^i \mid i \geq 1\} = \emptyset \]

The languages (sets) are completely disjoint.
“But wait, I thought we do magic and make parts of strings go away!”

No magic. You never directly remove part of a string using operators like intersection or set difference (−). You never directly add to a string using union.

Rather, you can make a new language (set) by:

- Putting all the strings from two languages into one new language (union).
- Only including strings from language 1 that aren’t in language 2 (set difference).
- Only including strings that are in both languages (intersection).
If in doubt, draw a Venn/Euler diagram and consider what strings are in each part of it.
What is \( \{a^n b^n c^n \mid n \geq 0\} \cap \{a^i b^i c^j \mid i, j \geq 0\} \)?
What is $\{a^n b^n c^n \mid n \geq 0\} \cap \{a^i b^i c^j \mid i, j \geq 0\}$?

$\{a^n b^n c^n \mid n \geq 0\}$
What is \( \{a^n b^n c^n \mid n \geq 0\} \cap \{a^i b^i c^j \mid i, j \geq 0\} \)?

\[ \{a^n b^n c^n \mid n \geq 0\} \]

These are the strings in both languages; the first language is a proper subset of the second.
What is $\{a^n b^n c^n \mid n \geq 0\} \cap \{a^i b^i c^j d^j \mid i, j \geq 0\}$?
What is $\{a^n b^n c^n \mid n \geq 0\} \cap \{a^i b^i c^j d^j \mid i, j \geq 0\}$?

$\{\varepsilon\}$. 
What is $\{a^n b^n c^n \mid n \geq 0\} \cap \{a^i b^i c^j d^j \mid i, j \geq 0\}$?

$\{\varepsilon\}$.

Strings in the first language can have
What is \( \{a^n b^n c^n \mid n \geq 0\} \cap \{a^i b^i c^j d^j \mid i, j \geq 0\}\)?

\( \{\varepsilon\}\).

Strings in the first language can have

\textbf{as, bs, and cs}
What is $\{a^n b^n c^n \mid n \geq 0\} \cap \{a^i b^i c^j d^j \mid i, j \geq 0\}$?

$\{\epsilon\}$.

Strings in the first language can have

- as, bs, and cs

no characters
What is \( \{a^n b^n c^n \mid n \geq 0\} \cap \{a^i b^i c^j d^j \mid i, j \geq 0\} \)?

\( \{\varepsilon\} \).

Strings in the first language can have

**as, bs, and cs**

no characters

Strings in the second languages can have
What is \( \{a^n b^n c^n \mid n \geq 0\} \cap \{a^i b^i c^j d^j \mid i, j \geq 0\} \)?

\( \{\varepsilon\} \).

Strings in the first language can have

- as, bs, and cs
- no characters

Strings in the second languages can have

- as, bs, cs, and ds
What is $\{a^n b^n c^n \mid n \geq 0\} \cap \{a^i b^i c^j d^j \mid i, j \geq 0\}$?

$\{\varepsilon\}$.

Strings in the first language can have

- as, bs, and cs
- no characters

Strings in the second languages can have

- as, bs, cs, and ds
- just as and bs
What is \( \{a^n b^n c^n \mid n \geq 0\} \cap \{a^i b^i c^j d^j \mid i, j \geq 0\}\)?

\(\{\varepsilon\}\).

Strings in the first language can have

- as, bs, and cs

no characters

Strings in the second languages can have

- as, bs, cs, and ds
- just as and bs
- just cs and ds
What is \( \{a^nb^nc^n \mid n \geq 0\} \cap \{a^ib^ic^jd^j \mid i, j \geq 0\} \)?

\( \{\epsilon\} \).

Strings in the first language can have

- as, bs, and cs
- no characters

Strings in the second languages can have

- as, bs, cs, and ds
- just as and bs
- just cs and ds
- no characters
What is \( \{a^n b^n c^n \mid n \geq 1\} \cap \{a^i b^i c^j d^j \mid i, j \geq 1\} \)?
What is \{a^nb^n c^n | n \geq 1\} \cap \{a^i b^i c^j d^j | i, j \geq 1\}?

\emptyset
Oh, I have a medical condition alright. It’s called CARING TOO MUCH!

about LANGUAGE THEORY!

And it’s INCURABLE!

as long as VASSAR continues to PAY ME!
“What problems can we solve with a computer?”
“What problems can we solve with a computer?”
A *real computer* has memory limitations: You have a finite amount of RAM, a finite amount of disk space, etc.

This makes every real computer equivalent to a (large) finite automaton.

However, as computers get more and more powerful, the amount of memory available keeps increasing.

An *idealized computer* is like a regular computer, but with unlimited RAM and disk space.

It functions just like a regular computer, but never runs out of memory.
CLAIM 1: Idealized computers can simulate Turing machines.

In other words, “Anything that can be done with a Turing machine can also be done with an unbounded-memory computer.”
The Turing machine’s finite-state control can be encoded as a table, making it easy for a computer to look up transitions.
Simulating a Turing machine

To simulate a Turing machine, the computer would need to be able to keep track of

- the finite-state control
- the current position
- the position of the tape head
- the tape contents

The tape contents are infinite, but that’s because there are infinitely many blanks on both sides.

We only need to store the part of the tape that’s been read from or written to so far.
Simulating a Turing machine

To simulate a Turing machine, the computer would need to be able to keep track of

- the finite-state control
- the current position
- the position of the tape head
- the tape contents

The tape contents are infinite, but that’s because there are infinitely many blanks on both sides.

We only need to store the part of the tape that’s been read from or written to so far.
Simulating a Turing machine

To simulate a Turing machine, the computer would need to be able to keep track of

- the finite-state control
- the current position
- the position of the tape head
- the tape contents

The tape contents are infinite, but that’s because there are infinitely many blanks on both sides.

We only need to store the part of the tape that’s been read from or written to so far.
CLAIM 2: Turing machines can simulate idealized computers.

In other words, “Anything that can be done with an unbounded-memory computer can be done with a Turing machine.”
We’ve seen that Turing machines can

implement loops

make function calls (i.e., use subroutines)

keep track of natural numbers (written in unary or in decimal on the tape)

perform elementary arithmetic (equality testing, addition, subtraction, increment, decrement)

perform if–else tests (i.e., take different transitions based on different cases)

Maintain variables using different parts of the tape (e.g., the two numbers being added)
Internally, real computers execute by using basic operations like

- simple arithmetic
- memory reads and writes
- branches and jumps
- register operations

Each of these are simple enough that they could be simulated by a Turing machine.
Anything you can do with a computer can be performed by a Turing machine.

The resulting Turing machine might be very large, very slow, or both, but it would still faithfully simulate the computer.

In fact, a Turing machine can simulate *any* effective method of computation.
An **effective method of computation** is a form of computation with the following properties:

- The computation consists of a set of steps.
- There are fixed rules governing how one step leads to the next.
- Any computation that yields an answer does so in finitely many steps.
- Any computation that yields an answer always yields the correct answer.

This isn’t a formal definition, but it’s a set of properties we expect out of a computational system.
The Church–Turing thesis

Every effective method of computation is either equivalent to or weaker than a Turing machine.

I'm as good as it gets!
computation

“Behold gravity in all its glory!”
computation

“Behold gravity in all its glory!”
We can consider many other reasonable models of computation: *DNA computing*, neural networks, quantum computing…

DNA arrays displaying the Sierpinski gasket, a fractal, generated by cellular automata. Rothemund et al., 2004
We can consider many other reasonable models of computation: DNA computing, neural networks, quantum computing…
We can consider many other reasonable models of computation: DNA computing, neural networks, *quantum computing*...
We can also consider many variations on Turing machines, with multiple tapes, nondeterminism, etc.
These might be (much) faster for some computations, but experience has confirmed that every such model can be simulated by a standard Turing machine.

They do not change what is *computable*. 
Problems solvable by any feasible computing machine

- Regular languages
- Context-free languages

All languages
Problems solvable by Turing machines

Regular languages

Context-free languages
As far as we know, no device built in the physical universe can have any more computational power than a Turing machine.

This is a remarkable statement, suggesting that a universal computer with enough memory and the proper programming should be able to simulate the function of a human brain.

In other words, artificial intelligence is just a “small matter of programming”.
Turing machines $\approx$ computers

Because Turing machines have the same computational power as regular computers, we can (essentially) reason about Turing machines by reasoning about actual computer programs.

Based on what’s most convenient, we’ll switch back and forth between Turing machines and computer programs – algorithms as high-level descriptions or pseudocode.
“What problems can we solve with a computer?”

What does it mean to “solve” a problem?
Unlike finite automata, which automatically halt after reading the input, Turing machines keep running until they explicitly enter an accept or reject state.

As such, it’s possible for a Turing machine to run forever without accepting or rejecting.
If a Turing machine might run forever, how do we formally define what it means to “build a Turing machine for a language”?

What implications does this have for problem-solving?
Terminology

Let $M$ be a Turing machine.

$M$ accepts a string $w$ if it enters an accept state when run on $w$.

$M$ rejects a string $w$ if it enters a reject state when run on $w$.

$M$ loops infinitely (or just loops) on a string $w$ if, when run on $w$, it never enters an accept or reject state.
$M$ does not accept $w$ if it either rejects $w$ or loops infinitely on $w$.

$M$ does not reject $w$ if it either accepts $w$ or loops on $w$.

$M$ halts on $w$ if it accepts $w$ or rejects $w$. 
The *language of a Turing machine* $M$ is

$$L(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$

If $w \in L(M)$, $M$ accepts $w$.

If $w \not\in L(M)$, $M$ does not accept $w$.

That is, when $M$ is run on $w$, either it rejects or it loops forever.
Recognizable languages

A language is called *Turing-recognizable* (or just *recognizable*) if it is the language of some Turing machine.

A Turing machine $M$ where $L(M) = L$ is called a *recognizer* for $L$.

The set of all languages that are Turing-recognizable is called *RE*.

$$L \in \textbf{RE} \iff L \text{ is Turing-recognizable.}$$
Does this correspond to what you think it means to “solve a problem”?
If a Turing machine $M$ halts on every possible input – i.e., it never goes into an infinite loop – then we call $M$ a \textit{decider}.

For deciders, accepting is the same as not rejecting and rejecting is the same as not accepting:
Decidable languages

A language is called *Turing-decidable* (or just *decidable*) if it is the language of *some* decider.

Equivalently, a language $L$ is Turing-decidable if there is a Turing machine $M$ such that

- If $w \in L$, then $M$ accepts $w$.
- If $w \notin L$, then $M$ rejects $w$.

The set of all languages that are Turing-decidable is called $R$.

$L \in R \iff L$ is Turing-decidable.
Decidable problems – the languages in $\mathbb{R}$ – are problems that can be “solved” by a computer.

(Though that solution isn’t guaranteed to be acceptably fast.)
All regular languages are in $\mathbb{R}$. We can use a Turing machine to simulate a DFA, and DFAs always halt.

$\{0^n1^n \mid n \in \mathbb{N}_0\} \in \mathbb{R}$. Proof: The Turing machine we built is a decider; it always halts.

In fact, all context-free languages are in $\mathbb{R}$.

The proof of this is trickier. It relies on using CFGs rather than PDAs. See Sipser page 200.
R matters because it is exactly the class of languages for which there is an algorithm to decide if a string is in the language.

By the Church–Turing thesis, this isn’t just about Turing machines.

If there is any algorithm to decide membership in the language, then there is a decider for it.
Say you’re working on a computer science assignment. You wonder if your program has a bug.

**The RE perspective:** If you find a bug, you know the answer is yes. If you can’t find a bug, that doesn’t mean there isn’t one.

**The R perspective:** You know there is or isn’t. (A program that could do this would be magic!)
Every decider is a Turing machine, but not every Turing machine is a decider.

So, $\mathbf{R} \subseteq \mathbf{RE}$.

But is $\mathbf{R} = \mathbf{RE}$?

That is, if you can confirm “yes” answers to a problem, can you also solve that problem?
Is this right?

- **Regular languages**
  - **Context-free languages**
    - **RE**
  - **R**

All languages
Or this?

Regular languages

Context-free languages

R

RE

All languages
“What **problems** can we solve with a computer?”

What is a “**problem**”?
A *decision problem* is a type of problem where the goal is answer yes or no.

**Example: Bin Packing**

You’re given a list of patients who need to be seen and how much time each needs to be seen for. You’re given a list of doctors and how much free time they have. Is there a way to schedule the patients so that they can all be seen?

Note: We’re not asking *what* that way is.

**Example: Route Planning**

You’re given a transportation grid of a city, a start location, a destination location, and information about the traffic over the course of the day. Given a time limit $T$, is there a way to drive from the start location to the end location in at most $T$ hours?
input

Computational device

Yes

No
How do we represent the inputs?
Two symbols should be enough for anyone.
Everything on your computer is a string over \( \{0, 1\} \).

For instance, every image can be encoded as a sequence of \( 0 \)s and \( 1 \)s (though not every sequence of \( 0 \)s and \( 1 \)s corresponds to an image)!
Generally speaking, if $Obj$ is some discrete, finite mathematical object, then we’ll use the notation $\langle Obj \rangle$ to refer to some reasonable encoding of that object as a string of characters.

$$\langle \rangle = 110011010001011110100101 \ldots$$
Object encodings

For the purposes of what we’re going to be doing, we aren’t (usually) going to worry about exactly how objects are encoded.

Generally we’ll assume that some clever person has already figured out a way to encode what we want, and we can just say, e.g., \langle 137 \rangle to mean “some encoding of 137” without worrying about how it’s encoded.

By analogy, consider whether you need to know how the int type is represented in C to do basic C programming.
Caveat

Remember: *discrete* and *finite*! Some things can’t be encoded as strings.

There’s no general way to encode *real numbers* as strings.

Imagine a real number generated by tossing infinitely many coins, one for each digit, heads = 0, tails = 1.

There’s no general way to encode *languages* as strings.

Imagine tossing a coin for each string. Heads = the string is in the language. Tails = the string is not in the language.
Encoding groups of objects

Given a finite group of objects, $Obj_1$, $Obj_2$, …, $Obj_n$, we can create a single string encoding all of these objects.

Think of it like a .tar file (or a .zip file without the compression).

We can denote the encoding of all of these objects as a single string by $\langle Obj_1, Obj_2, \ldots, Obj_n \rangle$.

This lets us feed multiple inputs into our computational device at the same time.
A Turing machine is given an input that is appropriately encoded. The machine then determines whether to accept or reject the input.
Our goal is to speak of computers solving problems.

We model this by looking at Turing machines recognizing languages.

For decision problems that we’re interested in solving, this precisely captures what we’re interested in capturing.
“What problems can we solve with a computer?”
We haven’t answered this question yet, but we’re getting closer.
Acknowledgments

This lecture incorporates material from:

W. Daniel Hillis
Nancy Ide
Keith Schwarz
Michael Sipser