End of semester updates

1. Last class we didn’t get to the **proof of Rice’s Theorem**. I’ve updated
   the slides to include this as an appendix.
2. Tonight I will post a handout providing a **guide to doing proof by**
   **reduction**, with additional examples. This will help you with
   Assignment 8.
3. **Assignment 8 extension:**
   Regular deadline is now: 5 p.m., May 9.
   Late deadline is now: 5 p.m., May 14.
4. We’ll fill out **CEQs** at the end of class today.
5. A **study guide for Exam 3** with practice problems will be released no
   later than class on Tuesday.
6. **Exam 3** – a regularly scheduled final exam that will heavily emphasize
   the most recent material – is 9 a.m., May 17, in SP 309
First we need a definition of a computer!

What problems can be solved by computers?

I'm a computer. I compute.
We have a model of a computer.
We’re not sure what we can solve at this point, but we’ll call the languages we can capture this way the regular languages.

What other machines can we make?

Wow – these new machines are way cooler than our old ones!

Nondeterminism! Is there any path through the NFA that leads to an accept state?
I wonder if they're more powerful?

Wow – I guess not! That's surprising.
So now we have a new way of modeling computers with finite memory!

The subset construction lets us convert any NFA to a (big) equivalent DFA

I wonder how we can combine these machines together.
Cool – since we can glue machines together, we can glue languages together as well.

How are we going to do that?

$\epsilon \\ M_1 \\ M_2 \\ \epsilon \\ \epsilon \\ \epsilon$

matt@vassar.edu
matthew.vassar@vassar.edu
asprey@cs.vassar.edu
...

$a^+ (.a^+)^* @ a^+ (.a^+)^+$
Wow – we’ve got a new way of describing languages.

So, what sorts of languages can we describe this way?

\[ \varepsilon R_1 R_2 \iff \text{start} \quad q_1 \quad R_1 \quad \varepsilon \quad q_1 \quad R_2 \quad q_f \]

Any regular expression can be systematically converted into an equivalent NFA.

Awesome – we got back the exact same class of languages!
It seems like all our models give us the same power! Did we get every language?

I guess not.

We formalize the argument that a language isn’t regular using the **Pumping Lemma for Regular Languages**.

There’s no way we can build a DFA for this; we’d need an infinite number of states.

But we did learn something cool:

*We’ve just explored what problems can be solved with finite memory.*
So what else is out there?

Well, what if we add memory to our machines?

These machines can do more than our old machines!

This stack tells us we’ve seen two left parentheses and need to find two matching right parentheses.
Can we describe these languages another way?

\[ S \rightarrow 1S1 \]
\[ S \rightarrow 1S \]
\[ S \rightarrow \varepsilon \]

\[ \varepsilon, S \rightarrow 1S1 \]
\[ \varepsilon, S \rightarrow 1S \]
\[ \varepsilon, S \rightarrow \varepsilon \]
\[ \Sigma, \Sigma \rightarrow \varepsilon \]

Awesome! We can call the languages these models generate or recognize the **context-free languages**.
So, did we get every language yet?

I guess not.

Darn right.

There are languages that don't satisfy the Pumping Lemma for CFLs, meaning it's impossible to design a CFG to describe them.

So what if we make our memory a little better?
Adding an infinite tape to a finite automaton, we get a *Turing machine*.

Can we make these any more powerful?

The *Church–Turing thesis* says that we can’t!
Why is that?
Turing machines can simulate other Turing machines!

In fact, any reasonable model of computation could be simulated by a Turing machine.

So, is every language decidable?

Consider what happens when a program is run on its own source code.

(Or – equivalently – when a Turing machine is run on its own encoding.)
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("uh-oh.py").read()
    if will_accept(my_source, my_input):
        return False
    else:
        return True

What happens if...
  ...this program accepts its input?
    Then it rejects its input!
  ...this program doesn't accept its input?
    Then it accepts its input!

It's not just $A_{TM}$ and the Halting Problem; there are an infinite number of undecidable languages.

The power of self-reference immediately limits what Turing machines can do!

We can prove a particular language is undecidable by reducing a known undecidable problem to it.

E.g., if we could solve the problem of deciding whether a TM accepts the string Vassar, we could use this decider to solve $A_{TM}$ by constructing a special TM that accepts the string Vassar in exactly the cases where TM $M$ run on string $w$ would accept.
Rice's Theorem tells us that any language about a non-trivial property of a Turing-recognizable language will be undecidable for exactly this reason.

There are an infinite number of undecidable languages, but is every language at least recognizable?

Oh great. Some problems are impossible.
So, we can't recognize everything, but can we at least refute membership, i.e., recognize all the things that aren't members of the language?

**REGULAR**

\[ \text{REGULAR} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is regular} \} \]

**REGULAR \not\in \text{RE}**

**REGULAR \not\in \text{co-RE}**

In fact, almost all languages – an uncountably infinite number of them – are not in **RE** or **co-RE**.
We've gone to the absolute limits of computing.

Discovery isn’t a straight road

The ideas and results we've seen weren't discovered in this order.

- The class of regular languages was introduced in 1951, 15 years after Turing machines!
- DFAs were invented 8 years after regular expressions.

And they weren't always intended for these purposes.

- Context-free grammars were invented by Noam Chomsky in 1957 for modeling the syntax of natural languages.
- The state-elimination method was introduced for circuit design!
Where to go from here?

Congratulations on making it this far!

You've done more than tick off a bunch of boxes.
You've given yourself the foundation to tackle problems from all over computer science.
**CMPU 241: Algorithms**

A mix of theory and practice, where you’ll learn about computational complexity: Which decidable problems can we compute efficiently and which are so inefficient they might as well be undecidable? Expect more reductions!

**CMPU 331: Compiler Design**

The ideas we’ve presented on defining languages, writing grammars, parsing strings, and writing finite automata form the basis for turning computer programs from strings of symbols into action. All computer programming rests on what we’ve learned.

**COGS 101: Introduction to Cognitive Science**

and

**CMPU 365: Artificial Intelligence**

If there’s a hero of this course, it’s Alan Turing. His work in theoretical computer science was motivated by the question of how we might create a thinking machine. What would this mean? How can we go from finite automata to artificial intelligence? Fewer proofs, but plenty of big ideas and big problems.

**CMPU 336: Computational Linguistics**

This is my area of research, where many of the ideas we use are applied to human languages. While programming languages are unambiguous and we know when we’ve understood them correctly, human languages are fascinating collections of ambiguity! Ideas of formal grammars, Chomsky normal form, and parse trees are important here.
Final thoughts

CS theory is all about asking what's possible in computer science.

There's so much more to explore and so many big questions to ask – *many of which haven't been asked yet!*

A whole world of theory and practice awaits.
That’s it!

Next time, we’ll review for the final exam by working through practice problems and answering your questions.

Acknowledgments

This lecture incorporates material from:

Keith Schwarz