Comp 598 Notes

Cole Killian

Summer 2020

This is McGill's undergraduate Math 598, "Topics in Science 1" instructed by Prakash Panangaden. This semester it is focusing on Automata and Computability. You can find this and other course notes here: https://colekillian.com/course-notes

Contents

1 05-04 **3**
- 1.1 Logisitics . . . 3
- 1.2 Schedule . . . 3
- 1.3 Principle of Induction . . . 5
- 1.4 Strings . . . 5

2 05-05 **6**
- 2.1 Quantifiers . . . 6
- 2.2 Design a machine that accepts string ending in "aa". . . . 6
- 2.3 NFA vs DFA . . . 6
- 2.4 Regular Languages as an algebra . . . 10
- 2.5 From DFA to reg exp . . . 12

3 05-06 **13**
- 3.1 Minimization of DFA . . . 13
- 3.2 Splitting Algorithm . . . 15
- 3.3 Brozozowski's Algorithm . . . 17
- 3.4 Infinite state automata . . . 18

4 05-07 **18**
- 4.1 Kleene Algebras . . . 21

5 05-11 **23**
- 5.1 ω regular expressions. . . . 24
- 5.2 ω-automata . . . 24

6 05-12 **29**
- 6.1 Basic Propositional Logic . . . 29
  - 6.1.1 Syntax . . . 29
  - 6.1.2 Proof Theory . . . 30

6.1.3 Semantics . . . 31
6.2 Temporal Logic . . . 31
6.2.1 Syntax . . . 32
6.3 Labelled Transition System . . . 32

7 05-13 **32**
7.1 Bisimulation . . . 32
7.2 Lattice Theory and Fixed Points . . . 34
7.3 Logical Characterization of Bisimulation . . . 36
7.4 Probabilistic Bisimulation and Logical Characterizaiton . . . 37

8 05-14 **37**
8.1 Learning Automata . . . 37
8.2 Fixed Point Operators and LTL . . . 38

9 05-19 **40**
9.1 Context Free Languages . . . 40
9.2 Designing CFG . . . 43
9.3 Closure Properties of CFL's . . . 44
9.4 Algorithms For CFLs . . . 45
9.5 Pushdown Automata . . . 46

10 05-20 **48**

11 05-21 **54**
11.1 Existence of Unsolvable Problems . . . 54
11.2 Turing Machine . . . 55
11.3 Models of Computation . . . 57
11.4 Theory of Computability . . . 57

12 05-25 **59**
12.1 Reductions . . . 59
12.2 Sharper Notion of Reduction . . . 61
12.3 Turing Reduction . . . 62

13 05-27 **63**
13.1 Universal Functions . . . 63
13.2 Total Computable Universal Function . . . 64
13.3 Compositional Programming . . . 64
13.4 Primitive Recursive Functions . . . 65
13.5 Degrees of Unsolvability . . . 65

§1 05-04

§1.1 Logisitics

1. Potentially an oral final exam
2. Too many assignments to grade at the fast pace. Students will self grade their homeworks, and a few will be chosen at random to see if people are being honest in their self evaluation
3. All assignments are to be completed in Latex
4. Fact means he'll say it and not prove it.

§1.2 Schedule

Lecture Schedule for COMP 598 Summer 2020

Week 1	Lecture 1	4 May	Induction, strings, automata, monoids
	Lecture 2	5 May	NFA, Kleene theorem, regexp, Kleene algebra
	Lecture 3	6 May	Myhill-Nerode, minimization, duality
	Lecture 4	7 May	Pumping lemma
Week 2	Lecture 1	11 May	Temporal logic
	Lecture 2	12 May	Bisimulation
	Lecture 3	13 May	Learning automata
	Lecture 4	14 May	Weighted automata
Week 3	Lecture 1	18 May	Context-free languages, grammars and parsing
	Lecture 2	19 May	Pumping lemma for CFLs, DCFLs
	Lecture 3	20 May	Models of computation
	Lecture 4	21 May	λ-calculus
Week 4	Lecture 1	25 May	Computability, reductions
	Lecture 2	26 May	Logic and unsolvability, arithmetic hierarchy
	Lecture 3	27 May	Fixed-point theory
	Lecture 4	28 May	Godel's and Tarski's theorems

Definition 1.1 (Equivalence Relation)**.** An equivalence relation R on a set S is a set of pairs $R \subseteq S \times S$ such that (write xRy for $(x, y) \in R$)

1. xRx
2. $xRy \Rightarrow yRx$
3. $xRy \ \wedge \ yRz \Rightarrow xRz$

Definition 1.2 (Equivalence Relation)**.** $[x] \coloneqq \{y | xRy\}$ is the equivalence class of x

The set of equivalence classes forms a partition of S. Prove this as an exercise.

Definition 1.3 (Partial Order)**.** Abstraction of comparison. Not all elements can be compared. A binary relation on a set S. Big difference is no symmetry.

1. $x \leq x$
2. $x \leq y \wedge y \leq x \Rightarrow x = y$
3. $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Example of taking subsets and ordering them by inclusion. This is a typical partial ordering that is not total. The real numbers are a total ordering.

Definition 1.4 (Well-founded order)**.** Given a p.o.set (partially ordered set) $(S, \leq)$ and $U \subseteq S$, we say that $u \in U$ is a minimal element of U if $\forall v < u,\ v \notin U$. There can be infinitely many minimal elements, or none.

Example 1.5

Negative integers has no minimal element

Non negative integers in which 0 is the smallest

Strictly positive rational numbers has no minimal element

A partially ordered set $(S, \leq)$ is said to be well-founded if every non-empty subset U has (powerful word, an existential qualifier) a minimal element.

Example 1.6

Examples - Non-negative integers These are well founded. - The positive rational numbers These are not well founded - Pairs $N \times N$ where $(m, n) \leq (m', n')$ if $m < m' \vee (m = m \wedge n \leq n')$. Between (1, 17) and (2, 5) there are infinitely many elements, but do not be fooled there is still a minimal element.

Fact 1.7. An order is well founded if and only if there are no infinite strictly decreasing sequences (chain) $x_1 > x_2 > x_3 > \cdots$.
An order that is both totally ordered and well-founded is called a well order.

Theorem 1.8 (Zermelo's Theorem)

Every set can be given a well order assuming the axiom of choice. People don't want to believe this because no one can figure out how to write a well ordering of the real numbers.

Note 1.9. Zermelo's well ordering principle, axiom of choice, and zorn's lemma are equivalent. Zorn's lemma is something people don't want to give up.

§1.3 Principle of Induction

Definition 1.10 (Predicate)**.** "predicate". you're in the set if you satisfy property of predicate. "predicate is a statement that may be true or false depending on the values of its variables". "The set defined by P(x) is written as x — P(x), and is the set of objects for which P is true."

$(S, \leq)$ is inductive if $\forall P,\ \forall x \in S,\ \forall y < x, P(y) \Rightarrow P(x)$

Not every order is inductive. Minimal element takes care of the base case. Which orders are inductive, and which are not?

Theorem 1.11
An order is inductive if and only if it is well founded.

Proof.

1. Assume $V \coloneqq \{s \in S \mid \neg P(s)\}$ is not empty. Which would contradict the principle of induction. Then $\forall y < v_0,\ y \notin V$ and $P(y)$. But then this implies $P(v_0)$. So then $V = \varnothing$. i.e. $\forall x\ P(x)$.

2. Ind $\Rightarrow$ Well founded. Assume $U \subseteq S$ has no minimal element. $P(x) \coloneqq x \notin U$. $\forall x(\forall y < xP(y)) \Rightarrow P(x)$. We know this because if all y is less than x, and $y \notin U$ by the predicate, then $x \notin U$ or else it would be a minimal element of U.

   Induction says that $\forall x P(x) \Rightarrow U = \varnothing$ and hence $(S, \leq)$ is well founded because all sets have a minimal element.

□

** Emmy Noether $\Sigma = \{a, b\}.\Sigma^* = \{\epsilon, a, b, aa, ab, \cdots\}$

Under lexicographic ordering this is not well founded.

§1.4 Strings

A finite set is called an alphabet. Σ^* is a set of finite sequences. The number of such sequences is infinite. ϵ is the symbol for the empty word. A subset of the set of sequences is called a language.

Definition 1.12 (monoid)**.** A moinoid is a set S with a binary operation $\cdot$ and a unit e.

1. $\forall x, y, z \in S,\ x \cdot (y \cdot z) = (x\dot{y}) \cdot z$

2. $\forall x \in S,\ x \cdot e = e \cdot x = x$

3. Monoids are not necessarily commutative. If we have that $\forall x, y,\ x \cdot y = y \cdot x$ then we get a special kind of monoid called a commutative monoid.

4. If there is an inverse operation, it is a group. Monoid's are a general form of group.

Example 1.13

1. Σ^* with concatenation and ϵ is a monoid, but it is not commutative. $aab \cdot ba = aabba$.

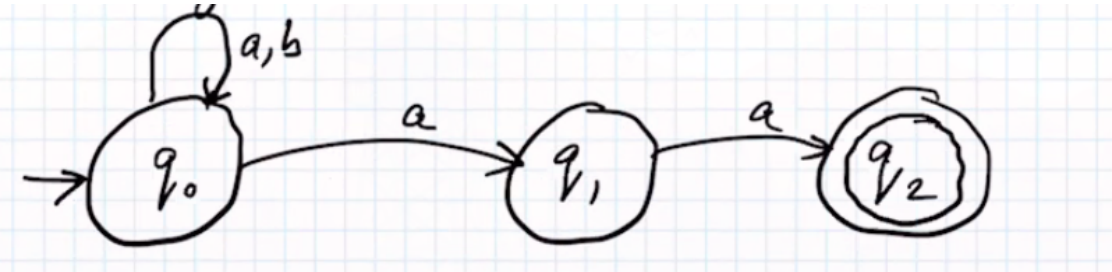
2. $\forall x, y, z$ if $xy = xz \Rightarrow y = z$ is a cancellative monoid.

§2 05-05

§2.1 Quantifiers

Can think of $\exists$ as Eloise and $\forall$ as Abeland. Think of statements as a game between these two. The statement is true if Eloise has a winning strategy, and false otherwise (i.e. Abeland has a winning strategy).

§2.2 Design a machine that accepts string ending in "aa".



§2.3 NFA vs DFA

1. There can be multiple next states or no next state. NFA's can make choices. There are multiple computation paths corresponding to different choices. A word is accepted if there exists a path leading to an accept state.

If a machine jams, that is equivalent to rejection. Therefore only sequences that end with "aa" can end up at the accept state.

Theorem 2.1
The language accepted by any NFA is a regular language. i.e. could also have been accepted by a DFA.

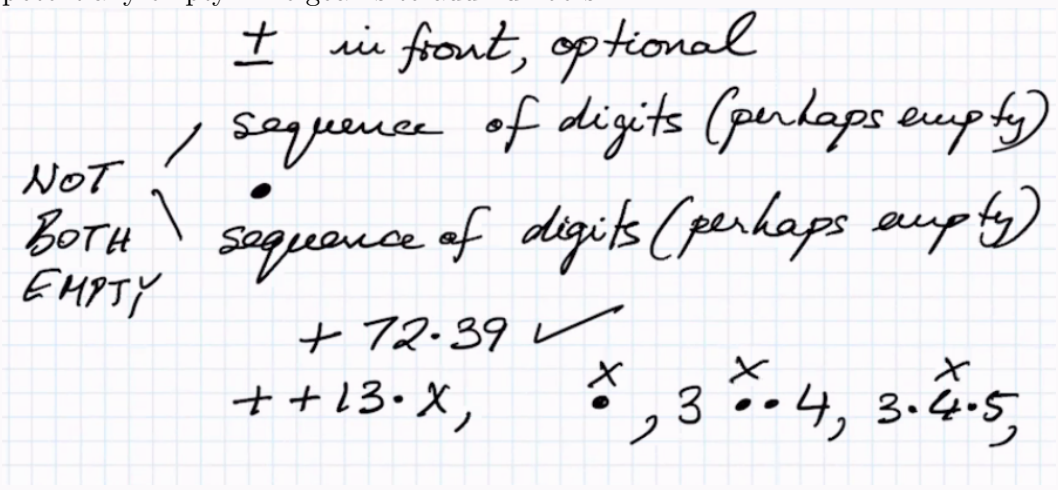
Remark 2.2. There is an algorithm for converting an NFA into an equivalent DFA. So you can go crazy with an NFA and know that it is still a regular language.

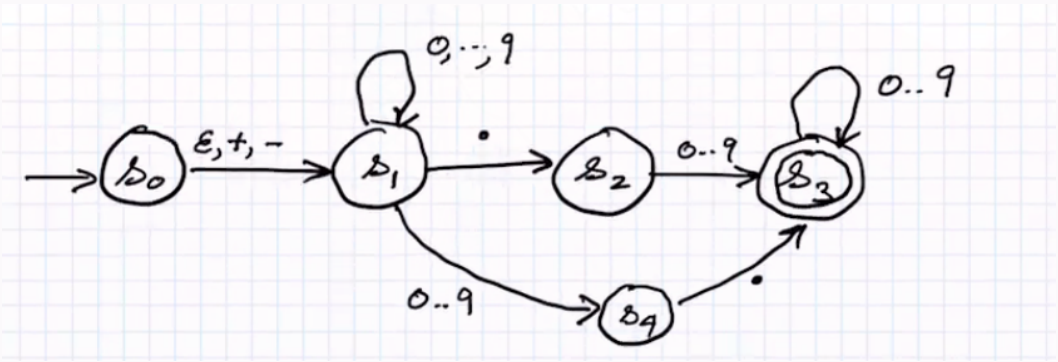
Note 2.3. Another extension: NFA + ϵ This machine can make jumps wihtout reading any input.

$$DFA \equiv NFA \equiv NFA + \epsilon \equiv \text{Regular Languages}$$

Example 2.4 (Decimal Numbers)

$\pm$ in front, optional. Two sequences of digits with one or the other (but not both) potentially empty. The goal is to add numbers.





1. Exactly plus, or exactly minus, or nothing. Otherwise the machine will jam.
2. ϵ is not just "hey I can jump to anywhere at any time"; it is part of the design of the machine.
3. There should be at least one thing after the decimal point.
4. No transition for another dot or plus or minus so any of those would cause the input to get rejected.
5. Allow for the possibility of ending with decimal point.
6. Need to two path to acceptance to distinguish between either before the decimal or after the decimal being empty, but not both.

This thing is doing a lot of guessing.

Note 2.5. In order to accept the language, it most both accept the right things and reject the wrong things. Cannot just accept everything.

Definition 2.6 (Formal definition of NFA)**.** An NFA is a 4-tuple:

1. Q: a finite set of states
2. $Q_0 \subseteq Q$: a (finite) set of start states
3. $F \subseteq Q$: a (finite) set of accept states
4. $\Delta : Q \times \Sigma \to 2^Q$.
   Or with ϵ. $\Delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$

$$\Delta^* : Q \times \Sigma^* \to 2^Q$$
$$\Delta^*(q, \epsilon) = \{q\}$$
$$\Delta^*(q, w \cdot a) = \cup_{q' \in \Delta^*(q,w)} \Delta(q', a)$$

5. The accept state

$$L(N) = \{w \in \Sigma^* \mid \exists q \in Q_0, \Delta^*(q, w) \cap F \neq \varnothing\}$$

Theorem 2.7
Given an NFA $N = (Q, Q_0, F, \Delta)$ there is a DFA $M = (S, s_0, F', \delta)$ such that $L(N) = L(M)$.

Proof. Keep track of all the places where the m/c (machine) could be.

$$S = 2^q$$
$$s_0 = Q_0$$
$$F' = \{S \subseteq Q \mid S \cap F \neq \varnothing\}$$

Take the union of all the places where it could go.

$$\delta(s, a) = \cup_{q \in s} \Delta(q, a)$$

It is easy to see that this works (favorite line of math professors) as advertised. □

Remark 2.8. It is possible to have an exponential blow up in the number of states.

Note 2.9. $S = 2^Q$ refers to the set of all subsets of Q (power set). The power set of S can be identified with the set of all functions from S to a given set of two elements 2^S.

This can be understood where you represent each of the elements in the set as a digit in a binary number. Then, in order to find all possible sets, you would find all possible binary numbers with $|S|$ digits, which gives $2^{|S|}$.

For $NFA + \epsilon$ we define ϵ-closure of a set to be the places one can get to with an ϵ-move.

Example 2.10

Let L be a regular language. Let $\frac{1}{2}L = \{w \in \Sigma^* \mid \exists x \in \Sigma^*, wx \in L \ \wedge \ |w| = |x|\}$.

Look at left half of word and guess that there is something you can tack on to the right such that the entire word is something the original machine would recognize.

Assume DFA (S, s_0, F, δ) recognizes L. We will construct an NFA for $\frac{1}{2}L$.

$$Q = S \times S \times 2^S$$

First index for . Second index for guessing where DFA will be when w ends. Third index is to try and track x the second half of the word.

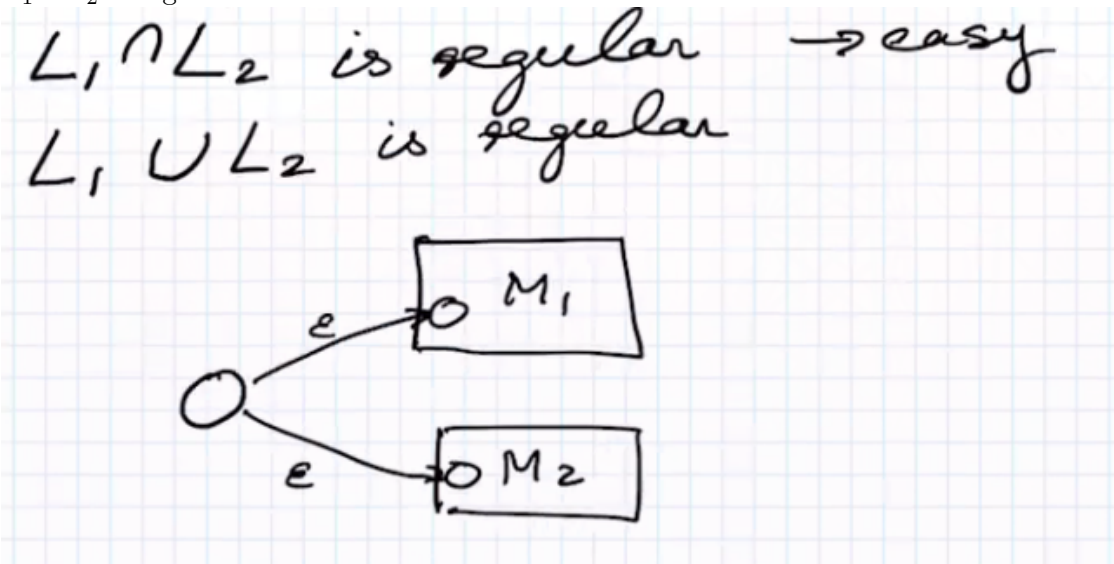
$$Q_0 = \{(s_0, s, \{s\})\}$$
$$F' = \{(s, s, X) \mid X \cap F \neq \varnothing\}$$
$$\Delta((s, \overline{s}, X), a) = \{(s', \overline{s}, X') \mid \delta(s, a) = s'\}$$

where X' is the set of states DFA can reach from x reading any symbol.

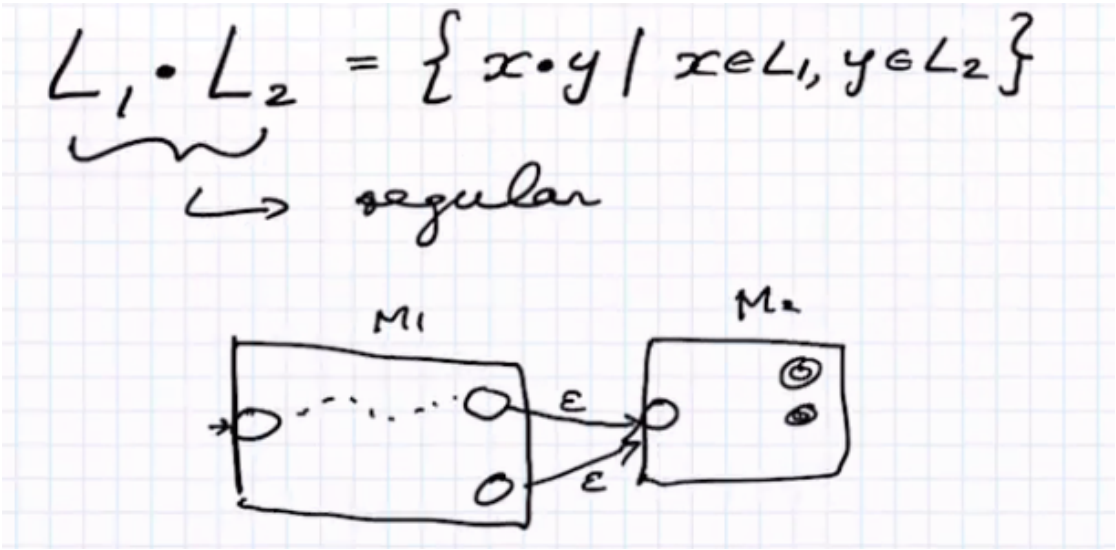
Closure properties of regular languages.

1. $L_1 \cap L_2$ is regular. Define a machine for L_1 and L_2. You create a state space with the cartesian product of states for L_1 and L_2. Then you go through in parallel and if it's in the accept state of both then it's accepted.

2. $L_1 \cup L_2$ is regular.



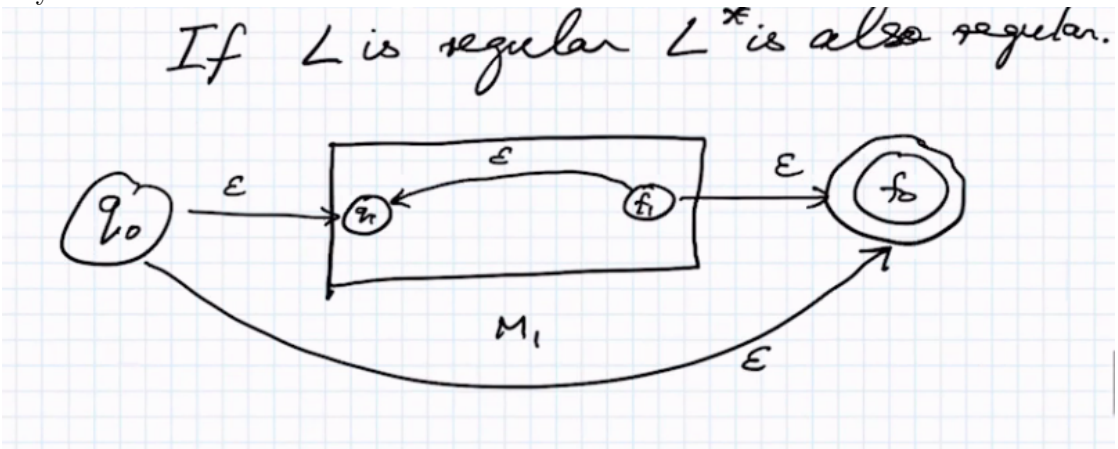
3. Concatenate language. $L_1 \cdot L_2 = \{x \cdot y \mid x \in L_1, y \in L_2\}$.



4. If L is a language, L^* is a new language where

$$L^* = \{w_1 \cdots w_k \mid w_i \in L\} \cup \{\epsilon\}$$

Basically take as many words as you like and glue them all together. Called the kleene (clay-knee) star operator. If L is regular $\Rightarrow L^*$ is regular but not the other way around.



5. Shuffle.

$$L_1 // L_2 = \{x_1 y_1 x_2 y_2 \cdots x_k y_k \mid x_1 x_2 \cdots x_k \in L_1,\ y_1 y_2 \cdots y_k \in L_2\}$$

§2.4 Regular Languages as an algebra

Definition 2.11 (Regular expressions)**.** Notation for describing regular languages in a machine independent fashion.

1. If R_1, R_2 are regular expressions, then $R_1 + R_2,\ R_1 \cdots R_2$ are regular expressions.

   If R is a regular expression, R^* is a regular expression.

A regular expression denotes a language.

$$[\![\varnothing]\!] = \varnothing,$$
$$[\![\epsilon]\!] = \{\epsilon\},$$
$$[\![R_1 \cdot R_2]\!] = [\![R_1]\!] \cdot [\![R_2]\!],$$
$$[\![R^*]\!] = ([\![R]\!])^* / [\![\overline{R}]\!] = \overline{([\![R]\!])}$$

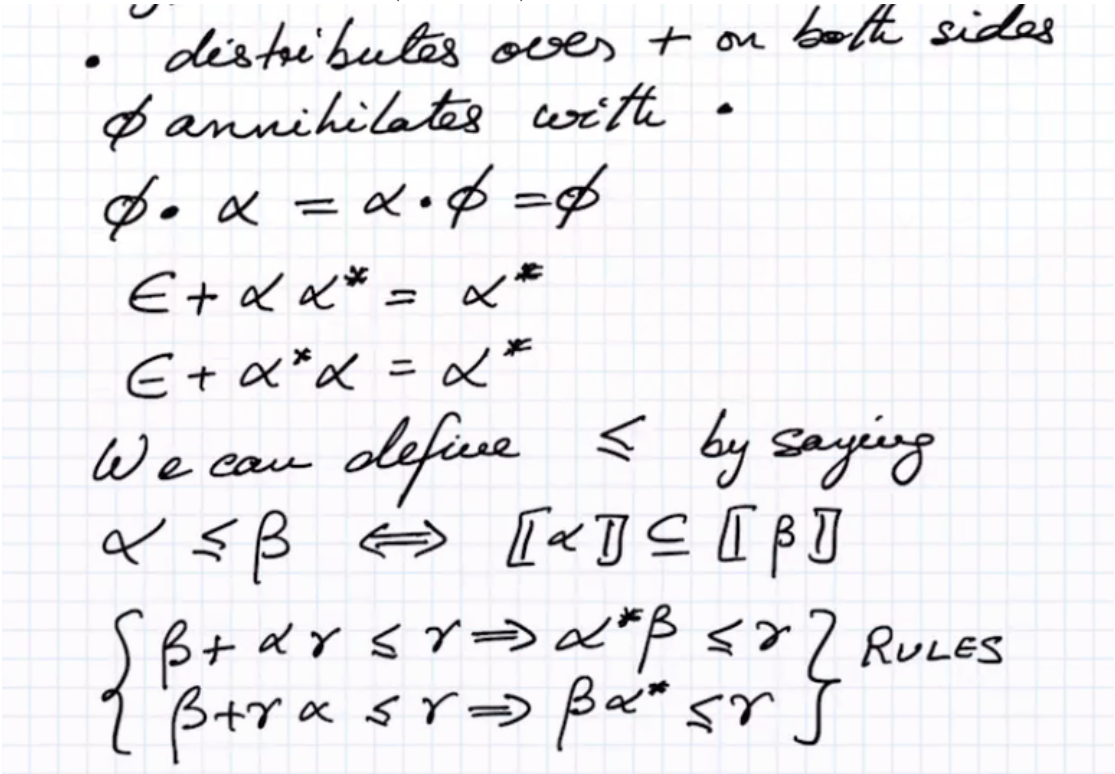
Example 2.12
$(aa + b)^* = \{aa, \epsilon, b, aab, bbaab, \cdots\}$

Theorem 2.13 (Kleene Theorem)
A language is regular iff it can be described by a regular expression.

Proof. Easy to see that reg exp implies regular. We already showed relevant closure properties. Reverse direction later if even. □

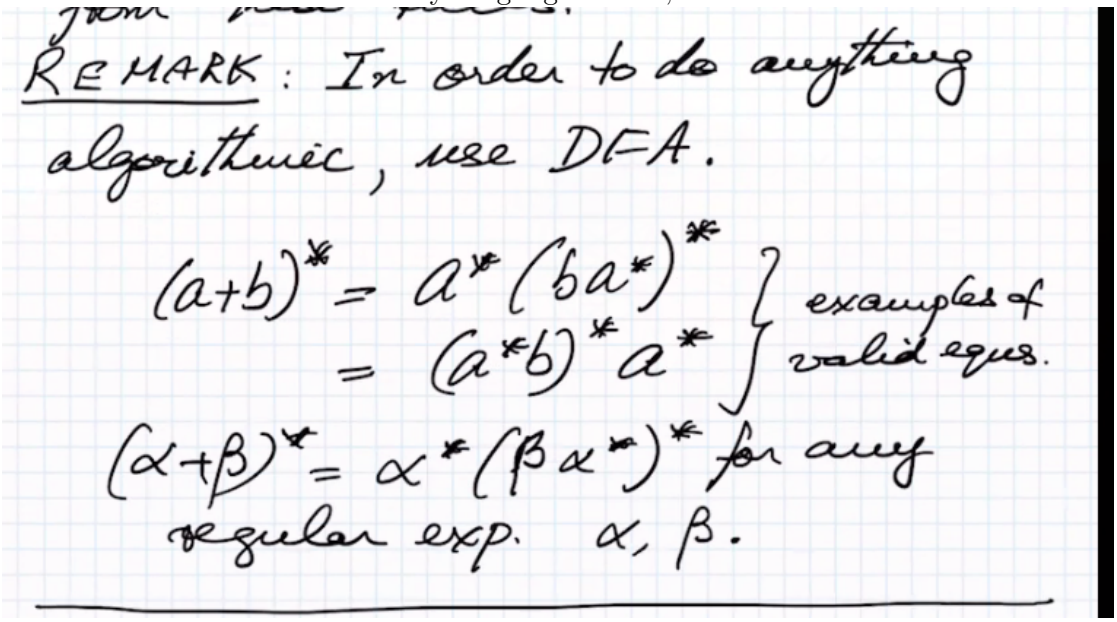
Note 2.14. "+" is notation for "or".

We have a set of equational axioms for reasoning about equality of regular expressions. Reg set of regular expressions (without -).



Theorem 2.15
These rules are complete. i.e. every valid equation can be derived from these rules.

Remark 2.16. In order to do anything algorithmic, use DFA.



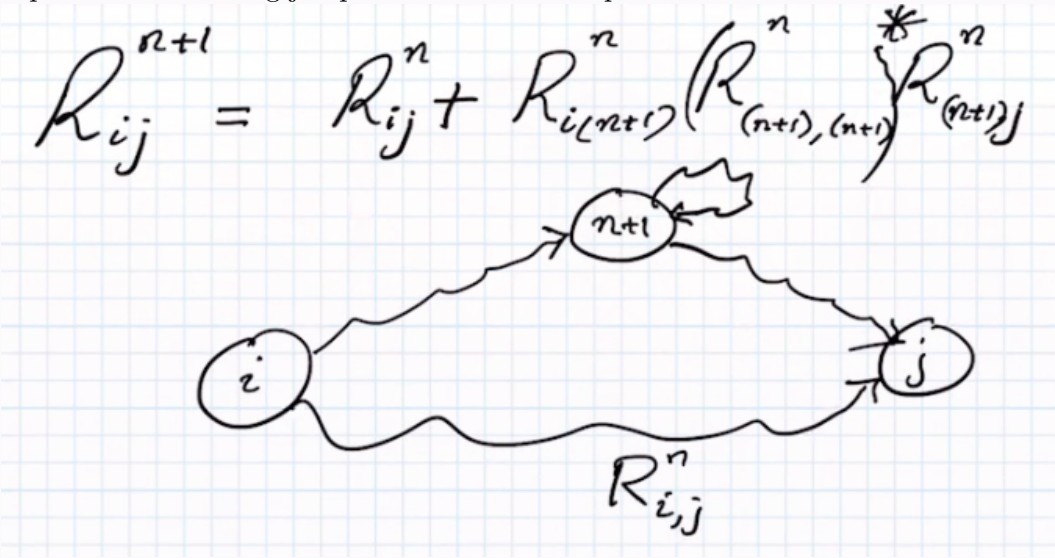
§2.5 From DFA to reg exp

Enumerate the states of a DFA $1, 2, \ldots, k$. For every pair of states i, j we define R_{ij}^n as a regular expression describing all strings that take the DFA from i to j only traversing states $1, \ldots, n$ along the way.

Example 2.17

R_{ij}^0 would represent only direct jumps from i to j. Either the empty word, or a single symbol.

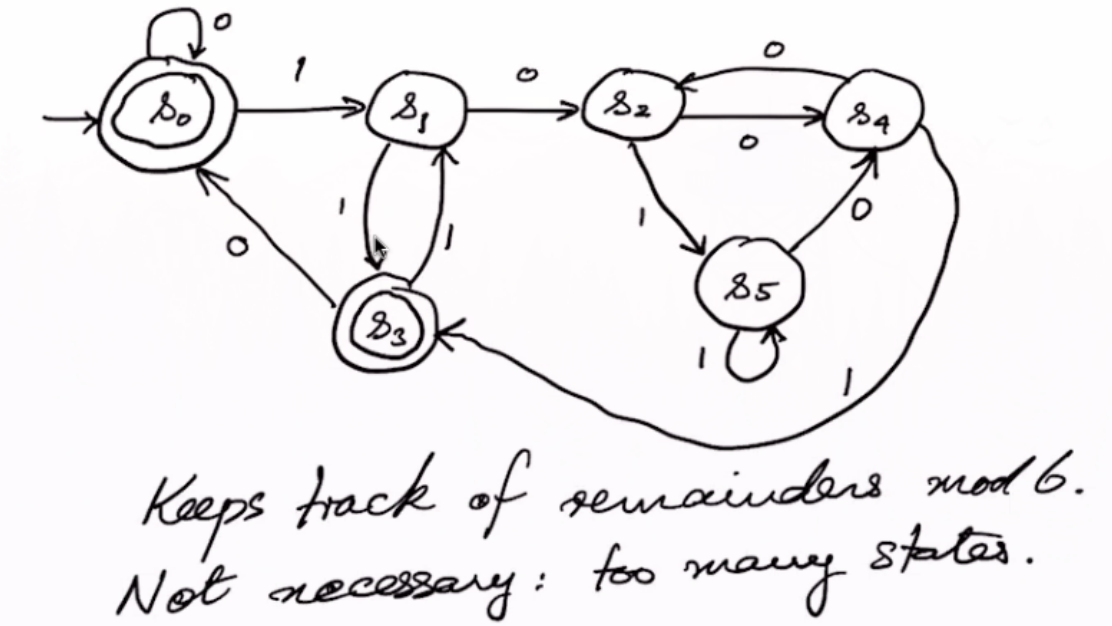
We can compute R_{ij}^{n+1} from R_{ij}^n. Then you can build the DFA by union all regular expressions describing jumps from start to accept states.



§3 05-06

§3.1 Minimization of DFA

Example of machine that checks divisibility by 3. But we already saw a three state machine that does the same thing.



$$\delta(s_0, 0) = s_0 = \delta(s_3, 0)\delta(s_1, 1) = s_1 = \delta(s_3, 1)$$

These states are doing the same thing so there's no reason to keep them separate.

Note 3.1. Once you get to a state, it doesn't matter how you got there. What happens in the future only depends on what you read in the future.

This is the fundamental meaning of the word state. It encodes the information to predict the future.

Definition 3.2. Given a DFA $M = (S, s_0, \delta, F)$, Σ, p, q are equivalent, $p \approx q$ if

$$\forall x \in \Sigma^*,\ \delta^*(p, x) \in F \Leftrightarrow \delta^*(q, x) \in F$$

So the languages defined by their starting states are the same.

Remark 3.3. $p \not\approx q$ means $\exists x \in \Sigma^*$ such that

$$(\delta^*(p, x) \in F \ \wedge\ \delta^*(q, x) \notin F)$$
$$\vee$$
$$(\delta^*(p, x) \notin F \ \wedge\ \delta^*(q, x) \in F)$$

Observe that $\approx$ is an equivalence relation.

Lemma 3.4

$p \approx q \Rightarrow \forall a \in \Sigma\ \delta(p, a) \approx \delta(q, a)$. But not necessarily $\delta(p, a) = \delta(q, a)$.

$$[p] \coloneqq \{q \mid p \approx q\},\ p \approx q \Leftrightarrow [p] = [q]$$

This means that $[p] = [q] \Rightarrow [\delta(p, a)] = [\delta(q, a)]$.

We define a new machine $M' = (S', s_0', \delta', F')$.

$$S' = S/\approx,\ s_0' = [s_0],\ F'\{[s]\ \}$$

This is a simpler draft

Lemma 3.5

$$p \in F\ \wedge p \approx q \Rightarrow q \in F$$

This is trivial when considering $\epsilon \in \Sigma^*$

Lemma 3.6
$\forall w \in \Sigma^*,\ \delta'^*([p], w) = [\delta^*(p, w)]$

Proof. By induction on the length w. □

Theorem 3.7
$L(M) = L(M')$

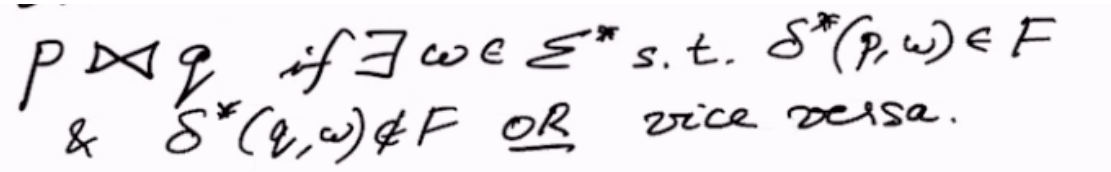
Proof.

$$x \in L(M') \Leftrightarrow \delta'^*([s_0], x) \in F'$$
$$\Leftrightarrow [\delta^*(s_0, x)] \in F' \Leftrightarrow \delta^*(s_0, x) \in F$$
$$\Leftrightarrow x \in L(M)$$

□

Quotient construction. Might be that every equivalence class contains element, but maybe each one contains several states. But definitely didn't get a bigger machine. This process is called collapsing a machine.

§3.2 Splitting Algorithm



Fact 3.8. If $\exists a \in \Sigma$ such that $\delta(p, a) bowtie \delta(q, a)$, then $pbowtieq$. The contrapositive of the above.

Matrices of booleans. You can multiply matrices of booleans by using "and" and "or". The collection of $n \times n$ matrices over a ring is always a ring.

Algorithm

1. For every (p, q) such that $p \in F \wedge q \notin F$, put 0 in the (p, q) cell of the matrix.
2. Repeat until no more changes. For each pair (p, q) that is not already marked with a 0, check if $\exists a \in \Sigma$ such that $(\delta(p, a), \delta(q, a))$ is marked with 0. If so mark (p, q) 0.

This algorithm correctly computes the equivalence classes. The time complexity is $O(n^4)$ in the naive case. $O(n^2 k)$ using data structures. $O(n \log n)$ Hopcraft method.

Theorem 3.9

If at the end two states are not marked 0 then they are equivalent.

Theorem 3.10

There is a unique minimal automaton to recognize a regular language.

This is how you prove the equivalence of regular expressions.

Definition 3.11 (Right invariant)**.** An equivalence relation R on Σ^* is said to be right-invariant if

$$\forall x, y \in \Sigma^* \text{ such that } xRy \Rightarrow \forall z \in \Sigma^*,\ azRyz$$

Example 3.12

DFA $M = (Q, \Sigma, q_0, \delta, F)$,

$$xR_m y \Leftrightarrow \delta^*(q_0, x) = \delta^*(q_0, y)$$

Example 3.13

Given $L \subseteq \Sigma^*$, not necessarily regular, we define R_L on Σ^*

$$xR_L y \Leftrightarrow xz \in L \Leftrightarrow yz \in L$$

Note 3.14. The index of an equivalence relation is the number of equivalence classes. Infinite equivalency classes means infinite index.

Theorem 3.15 (Myhill-nerode 1957)
The following are equivalent

1. L is regular
2. L is the union of some of the equivalence classes of a right-invariant equivalence relation R of finite index.
3. Any equivalence relation R with the properties described in (2) has to refine R_L.

Note 3.16. To say that an equivalence relation refines another means that it partitions all the equivalence relations.

Full proof is in the notes.

§3.3 Brozozowski's Algorithm

Will convert some DFA to NFA.

1. Reverse all the arrows change the initial states to an accept state, and make all the accept states start states.
2. Determinize it. This may cause exponential blow up because you are looking at the set of all subsets. Convert it from NFA to DFA.
3. Remove unreachable states.
4. Repeat 1.
5. Repeat 2. Might expect double exponential blow up, but instead there is a huge collapse to the minimal version of the original machine.
6. Repeat 3.

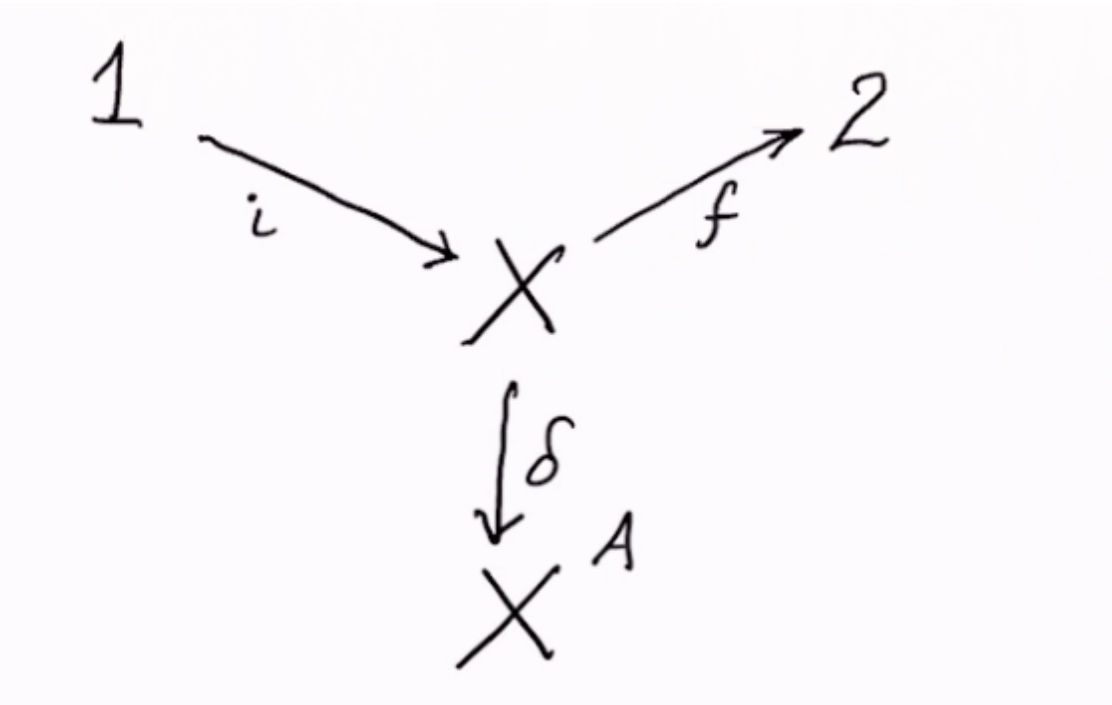
The result is the minimal machine.

Can't explain the whole thing. The point is to get interested and excited about this topic.

Note 3.17. X^A means the set of all functions from A to X. $\delta(x) : X \to X^A$.

$$X^A \approx [A \to X]$$

We can either write $\delta(x)(a)$ in which case $\delta : X \to X^A$ or $\delta(x, a)$ where $\delta : X \times A \to X$. This process is called currying.



§3.4 Infinite state automata

A^* is set of all words. Every word is a state. Initiale state is empty word.

States are words. Transition is stick the new letter at the end of the word.

There exists a unique map $r : A^* \to X$ such that the diagram above commutes.

Whatever path you follow that leads to the same end state results in the same outcome. Same as moving a different path with different endpoints.

Note 3.18. Give $f : V \to W$, $f^A : V^A \to W^A$. $\varphi \in V^A$, then $f^A(\varphi)(a) = f(\varphi(v))$. This is countably infinite.

Definition 3.19 (Reachability)**.** r maps the empty string to the . r tracks every word through the automaton with δ^*. r is the reachability map. It tells you which states can be reached. An automaton is reachable if every state can be reached.

§4 05-07

Lemma 4.1 (Pumping Lemma)

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

Not possible to recognize this language with finite automaton because it would require unbounded memory. Non regular languages could recognize this.

Suppose we have putative (generally considered or reputed to be) DFA that recognizes L. Then we can pick a word with a block of a with length greater that the number of states in the machine. This means that the machine will have to repeat a state. This rests on the pigeon hole principle.

The idea is that now you can exploit this loop as many times as you'd like by inserting the string that brings you about the loop.

Now for the lemma. Let L be a regular language. Then

$$\exists p > 0 \text{ such that } \forall w \in \Sigma^* \mid w \in L \wedge |w| \geq p$$
$$\exists x, y, z \in \Sigma^* \text{ such that } w = xyz,\ |xy| \leq p,\ |y| > 0$$
$$\forall i \in \mathbb{N},\ xy^i z \in L$$

An intuition for this is that y is the word that takes you through the loop, so you can repeat it as many times as you'd like.

Given a DFA for L choose p to be strictly greater than the number of states in the DFA. If $|w| \geq p \wedge w \in L$ then as the automaton changes state it must hit the same state twice while reading the first p letters, when p is greater than the number of states in the machine. □

Definition 4.2 (Finite Language)**.** A finite language is one containing a finite number of words.

Fact 4.3. Every finite language is regular. The p that you choose is longer than any word in the language. So then a finite language would have no words of length greater than p.

Fact 4.4. L regular $\Rightarrow$ L can be pumped. Contrapositive is that L cannot be pumped $\Rightarrow$ L not regular.

Note that it is not true that L can be pumped $\Rightarrow$ L regular.

Lemma 4.5 (Pumping Lemma Contrapositive)

$$L \subseteq \Sigma^* \ s.t. \ \forall p > 0$$
$$\exists w \in L \ with \ |w| \geq p \ s.t.$$
$$\forall x, y, z \in \Sigma^* \ with \ w = xyz, \ |xy| \leq p \wedge |y| > 0$$
$$\exists i \in \mathbb{N} \ s.t. \ xy^i z \notin L$$
$$\Rightarrow L \text{ is not regular}$$

Use games to deconstruct this statement. You and the devil. You play the $\exists$ quantifiers, and the devil plays the $\forall$ quantifiers. You must come up with a strategy to win every game.

The obvious first move is represented by a symbolic p. Your first move is explicit. The devil's move in step 3 must be analyzed by an exhaustive case analysis. Your last move must specify a response for all cases.

Example 4.6

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

1. Demon choses $p > 0$
2. You chose $w = a^p b^p$
3. The devil is constrained by $|xy| \leq p$ to choose y to consist exlucisvely of a's. So $y = a^k$ for some $0 < k \leq p$.
4. I choose $i = 2$. Didn't quite catch the rest

Thus L is not regular.

Example 4.7

$L = \{a^q \mid q \text{ a prime number}\}$

Demon picks $p > 0$. I pick a^n where $n > p$, n is a prime. Demon has to pick $y = a^k$ where $0 < k \leq p$. I pick $i > 1$, deferring the exact choice. New string $xy^i z$ is $a^{n+(i-1)k}$. Choose $i = n + 1$. Then $a^{n+nk} = a^{n(1+k)}$ which is not a prime number so L is not regular.

Example 4.8

$L = \{a^n b^m \mid n \neq m\}$

Wants you have a stock of languages that you know are not regular, you don't always have to do pumping. This example is hard to do directly with the pumping lemma.

$\overline{L}$ (L complement) is a big mess. But $\overline{L} \cap a^*b^* = \{a^n b^n \mid n \geq 0\}$. This is not a regular language to L is not regular.

Example 4.9

$L = \{a^i b^i \mid i > j\}$.

Example 4.10

$L = \{x + y = z \mid xyz \in \{0,1\}^* \wedge \text{the equation is valid}\}$

Demon picks p. I pick

$$\underbrace{111\cdots1}_{p}$$

Definition 4.11. If $S \subseteq \mathbb{N}$, define $unary(S) = \{1^n \mid n \in S\}$. $binary(S) = \{w \in \{0,1\}^* \mid w\text{read as a binary number is in}S\}\}$

If $binary(S)$ is regular does that mean $unary(S)$ is regular? No. Consider $S = \{2^n \mid n \geq 1\}$. Then binary is regular because 100^* is clearly regular. $unary(S) = \{1^{2^p}\}$ is not regular because you can pump to a non power of 2.

§4.1 Kleene Algebras

Definition 4.12 (Semi-ring)**.** A set with 2 operations. S: the carrier of the semi-ring. $+ : S \times S \to S$. $\times : S \times S \to S$. $0 : S$, $1 : S$. $(S, +, 0)$ forms a commutative monoid. $(S, x, 1)$ forms a monoid. $\times$ distributes over $+$. 0 annihilates with x i.e. $a \times 0 = 0 \times a = 0$.

Semi-ring is similar to a ring, but doesn't require that each element has an additive inverse. i.e. if $(S, +, 0)$ forms a group then it produces a ring instead of a semi-ring.

Example 4.13

$(\mathbb{N}, 0, 1, +, \times)$. This forms a semi ring, because we don't have the negative integers for the additive inverses. $(\mathbb{Z}, 0, 1, +, \times)$ would form a ring.

$\mathbb{Z} + i\mathbb{Z} = \{a + ib \mid a, b \in \mathbb{Z},\ i^2 = -1\}$ is the ring of Gaussian integers.

Example 4.14

$n \times n$ matrices over $\mathbb{N}$. Multiply by matrix multiplication and add componentwise.

Example 4.15

An idempotent semiring J is a semi ring such that $\forall x \in J, x^2 = x$. $(T, F, \vee, \wedge)$.

idempotent is an element which is unchanged in value when multiplied or otherwise operated on by itself.

Example 4.16

If we have any semiring, the set of $n \times n$ matrices with entries in this semiring form a semi-ring.

Definition 4.17 (Kleene Algebras)**.** $K = (S, +, \cdot, 0, 1, *)$.

1. $(S, +, 0)$: commutative monoid
2. $(S, \cdot, 1)$: monoid
3. $(S, +, \cdot, 0, 1)$ forms an idempotent semiring.
4. $$1 + aa^* = a^*$$ $$a + a^*a = a^*$$
5. We introduce a partial order $a \leq b \coloneqq a + b = b$ (check that this is really a partial order). 2 rules.
$$b + ac \leq c \Rightarrow a * b \leq c$$ $$b + ca \leq c \Rightarrow ba^* \leq c$$

Example 4.18

Let Σ be a finite alphabet $S =$ regular languages $\subseteq \Sigma^*$. $+$ is union. $\cdot$ concatenation. $*$ is kleene star.

Example 4.19

S any set and R the collection of binary relations on S. A binary relation $r \subseteq S \times S$. $+$ is union. $\cdot$ is relational composition. xry means $(x, y) \in r$. $x(r \cdot s)y = \exists z s.t. xrz \wedge zsy$. $0 \coloneqq \varnothing$. $1 \coloneqq \{(s, s) \mid s \in S\}$. r^* is the reflexive, transitive closure of r.

graphs are a nice way of picturing relations. r^* is reflexive, transitive closure of r. transitive closure let's you take the paths of the graph. And everything is related to itself. Directed graph because not necessarily symmetric. xr^*y if $\exists n \in \mathbb{N}, z_1, \cdots, z_n s.t. xrz_1 \wedge z_1 r z_2 \cdots z_n r y$. i.e. there exists a path from x to y.

Solving for x. $ax + b = x$. a^*b is the smallest solution. $aa * b + b = (aa * +1)b = a^*b$. Smallest solution means that if x is another solution, then $a^*b \leq x$.

Example 4.20

If K is any kleene algebra, $M_n(K)$ is $n \times n$ matrices with entries in K. Do operations as you would expect about matrices. What the heck is star though? 0 is 0. 1 is 1 along diagonal and 0 everywhere else.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^* \coloneqq \begin{bmatrix} (a + bd^*c)^* & (a + bd^*c)^* bd^* \\ (d + ca^*b)^* ca^* & (d + ca^*b)^* \end{bmatrix}$$

Now we can solve matrix vector equations.

Dexter Kozen has developed and extended the theory tremendously including probabilistic extensions.

§5 05-11

Σ corresponds to the alphabet. Σ^* represents finite words. Σ^ω is the set of infinite words over Σ.

If $\Sigma = \{a, b\}$, Σ^ω is uncountable. $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. A string in Σ^ω is countable, so positions in the string can be indexed by $\mathbb{N}$ i.e. countable.

There is a natural partial order on Σ^∞. $\alpha \leq \beta$ if α is a prefix of β. Infinite strings cannot be compared.

First ordinal. Reflexivity is part of a partial order.

Remark 5.1. We can define a metric on Σ^* by letting $d(x, y) = 2^{-n}$ where n is the first position where $x[n]! = y[n]$. The longer the shared prefix the closer two words are. This is an ultrametric which means that it satisfies the following strengthened version of the triangle inequality

$$d(x, z) \leq \max(d(x, y), d(y, z))$$

Σ^∞ is the cauchy completion of Σ^* and the metric naturally extends to Σ^*.

Infinite strings are important for the verification of reactive systems. They are designed to run forever so they have to talk about infinite computations.

§5.1 ω regular expressions.

Let E be a regular expression and $\epsilon \notin L(E)$.

E^ω is an ω-regular expression.

$$L_\omega(E^\omega) \coloneqq \{w_1 w_2 w_3 \cdots \mid w_i \in L(E)\}$$

where $L_\omega(F)$ is the ω-language defined by the ω regular expression F.

Let E_i be regular xpressions and F_i be regular expressions such that $\epsilon \notin L(F_i)$.

$$E_1(F_1)^\omega + \cdots + E_n(F_n)^\omega$$

is a general ω-regular expression.

$E^\omega = EE^\omega$ so E^ω is also an ω-regular expression.

When concatenating where $\alpha, \beta \in \Sigma^\omega$, $\alpha\beta = \alpha$.

Definition 5.2. An ω-language ($\subseteq \Sigma^\omega$) is ω-regluar if defined by some ω regular expression. This ω-regular expression may not be unique.

Note 5.3. $\cdot$ is concatenation. $+$ is union. $(.)^\omega$ is infinite repetition.

§5.2 ω-automata

Specifically non deterministic biichi automata.

$$A = (Q, \Sigma, \delta, Q_0, F)$$
$$\delta : Q \times \Sigma \to 2^Q$$
$$s' \in \delta(s, a) \equiv s \to_a s'$$

Given a word $\sigma \in \Sigma^\omega$, a run of ω in A is an infinite sequence of states $q_0, q_1, \cdots$ such that $q_0 \in Q_0$, $\forall i\ q_i \to_{\sigma[i]} q_{i+1}$.

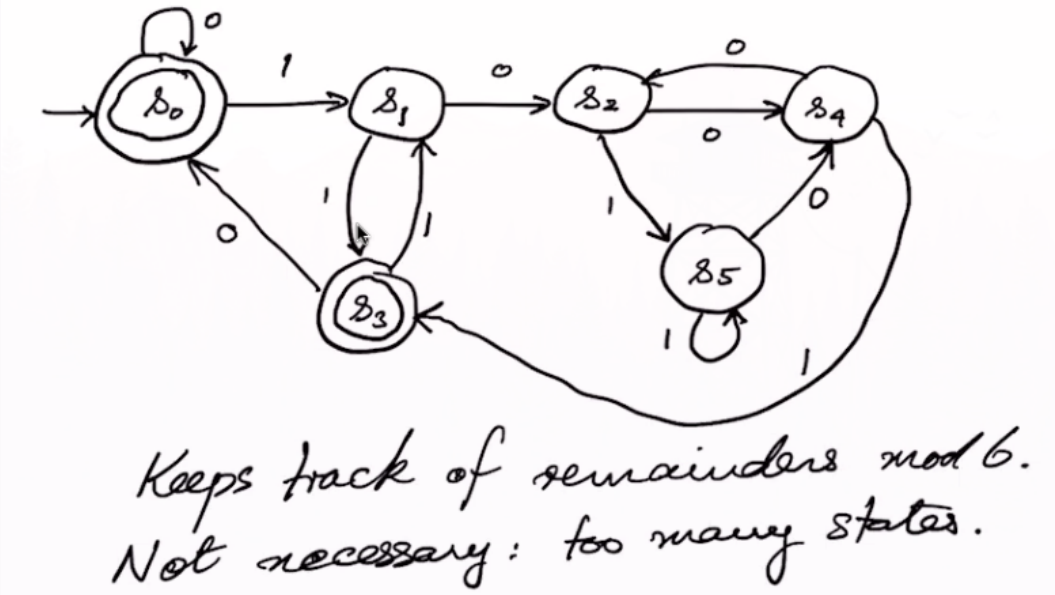
$$inf(r) = \{q \mid q \text{ occurs infinitely often in} r\}$$

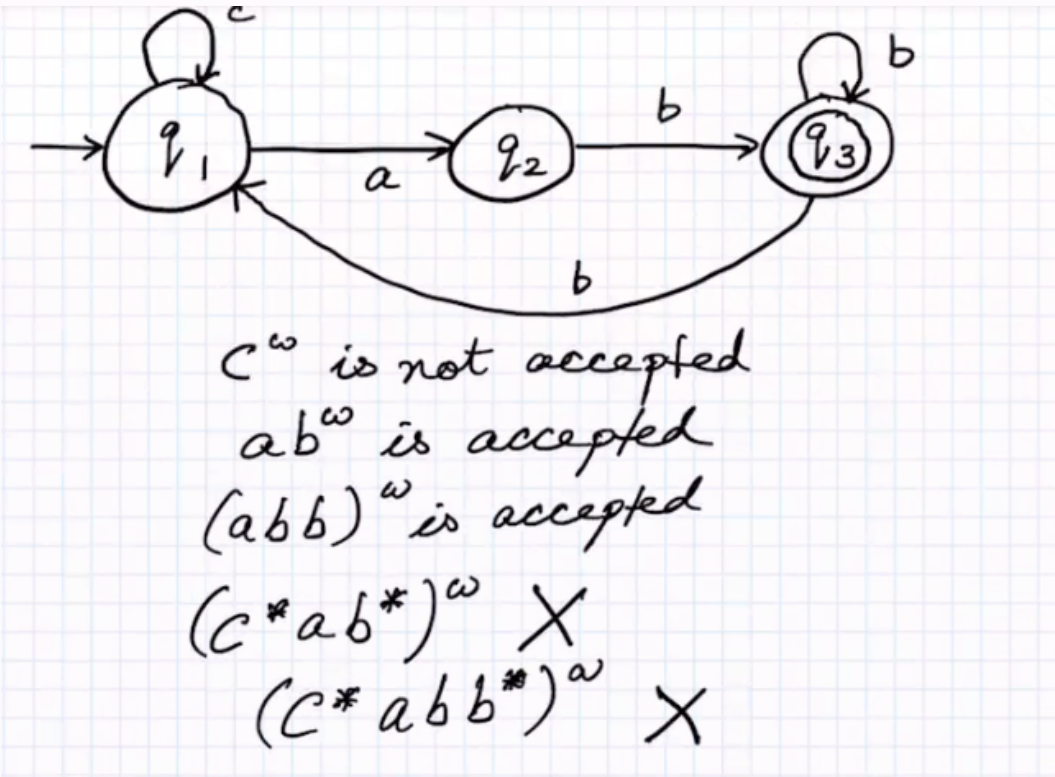
A run is accepting if it hits an accept state infinitely often. There are only finitely many accept states so one of them has to be hit infinitely many times.

$$inf(r) \cap F \neq \varnothing$$

A word is accepted by A if that word has some accepting run.

Example 5.4





$$L_\omega(A) = c^*ab(b^+ + bc^*ab)^\omega$$

We see that it doesn't necessarily have to be periodic, but perhaps a periodic aspect to it. Get it to the start state first, then think of a pattern.

Theorem 5.5

NBA's accept exactly the ω-regular languages.

Proof. Show that any ω-regular language can be captured by an NBA.

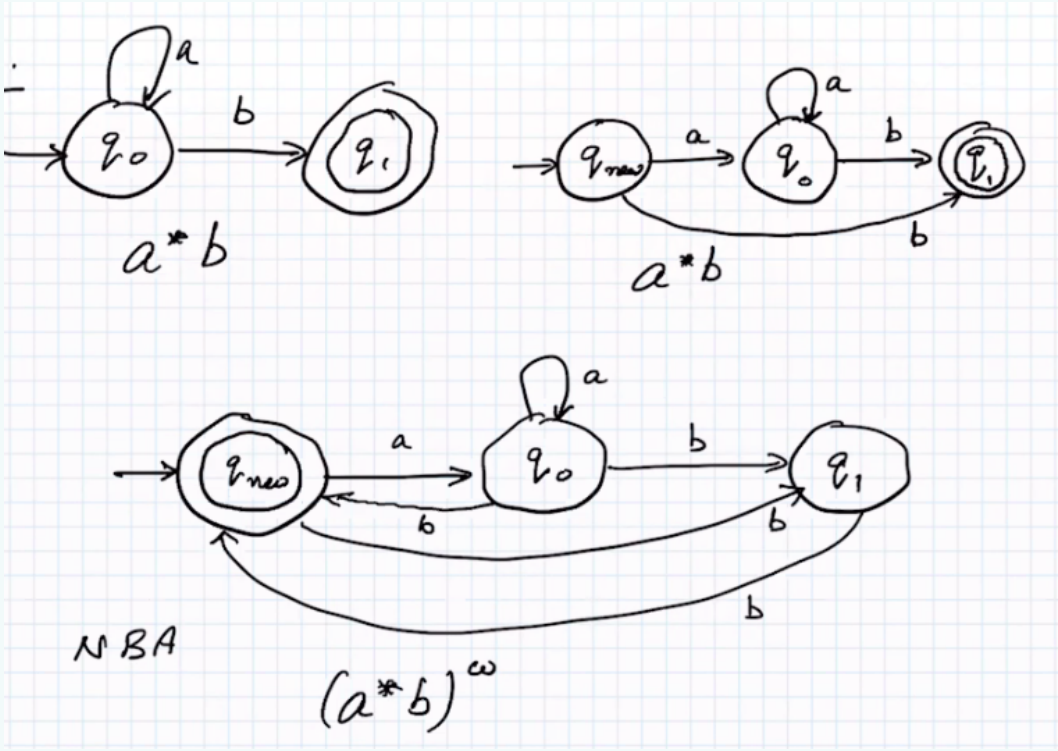
Given $A_i = (Q_i, \Sigma, \delta_i, Q_{0,i}, F_i)$, $i = 1, 2$ with $Q_1 \cap Q_2 = \varnothing$. Then $A_1 + A_2 = (Q_1 \cup Q_2, \Sigma, \delta, Q_{0,1} \cup Q_{0,2}, F_1 \cup F_2)$. You can start either in start states of A_1 or A_2.

$$\delta(q, a) = \delta_i(q, a) \text{ if } q \in Q_i$$
$$L_\omega(A_1 + A_2) = L_\omega(A_1) \cup L_\omega(A_2)$$

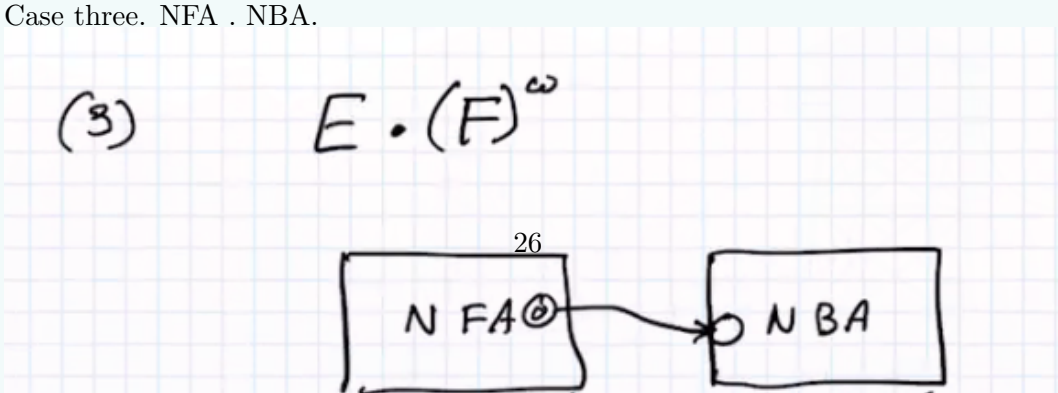
We are now considering F^ω. We have an NFA for F and assume that all the inital states are non accepting. $Q_0 \cap F = \varnothing$ and there are no incoming transition to any state in Q_0. You may worry that this will be too restricting on applicable NFAs. But if your NFA does not satisfy these assumptions we can modify it so that it does and defines the same language.

Idea for doing this: Brand new unique start state. And then leave everything afterwards untouched.

Given $A = (Q, \Sigma, \delta, Q_0, F)$. NBA $A' = (Q, \Sigma, \delta', Q_1, F')$. $L_\omega(A') = (L(A))^\omega$. The goal is to keep recognizing infinitely many words. So each time you hit an accept state, send it back to the start state.



Case three. NFA . NBA.



Theorem 5.6

Other direction

Suppose $A = (Q, \Sigma, \delta, Q_0, F)$ NBA. Fix any 2 states. $A_{qp} \coloneqq (Q, \Sigma, \delta, \{q\}, \{p\})$. This converts it to an NFA. It is not meant to read omega words.

$$L(A_{qp}) = L_{qp} = \{w \in \Sigma^* \mid \delta^*(q, w) \ni p\}$$

These are words that take you from q to p.

Now consider an accepted word σ of $L_\omega(A)$ with an accepting run. It must hit sum $q \in F$ that appears infinitely often. Therefore

$$\sigma = \underbrace{w_0}_{L_{q_0 q}} \underbrace{w_1}_{L_{qq}} \underbrace{w_1}_{L_{qq}} \cdots$$

where w_i are non empty. Any σ the can be expressed like this has an accepting run. Therefore

$$L_\omega(A) = \cup_{q_0 \in Q_0} \cup_{q \in F} L_{q_0 q} \cdot (L_{qq} \setminus \{\epsilon\})^\omega$$

Therefore anything recognized by an NBA is described by an ω-regular expression. Kleene's theorem for ω-languages.

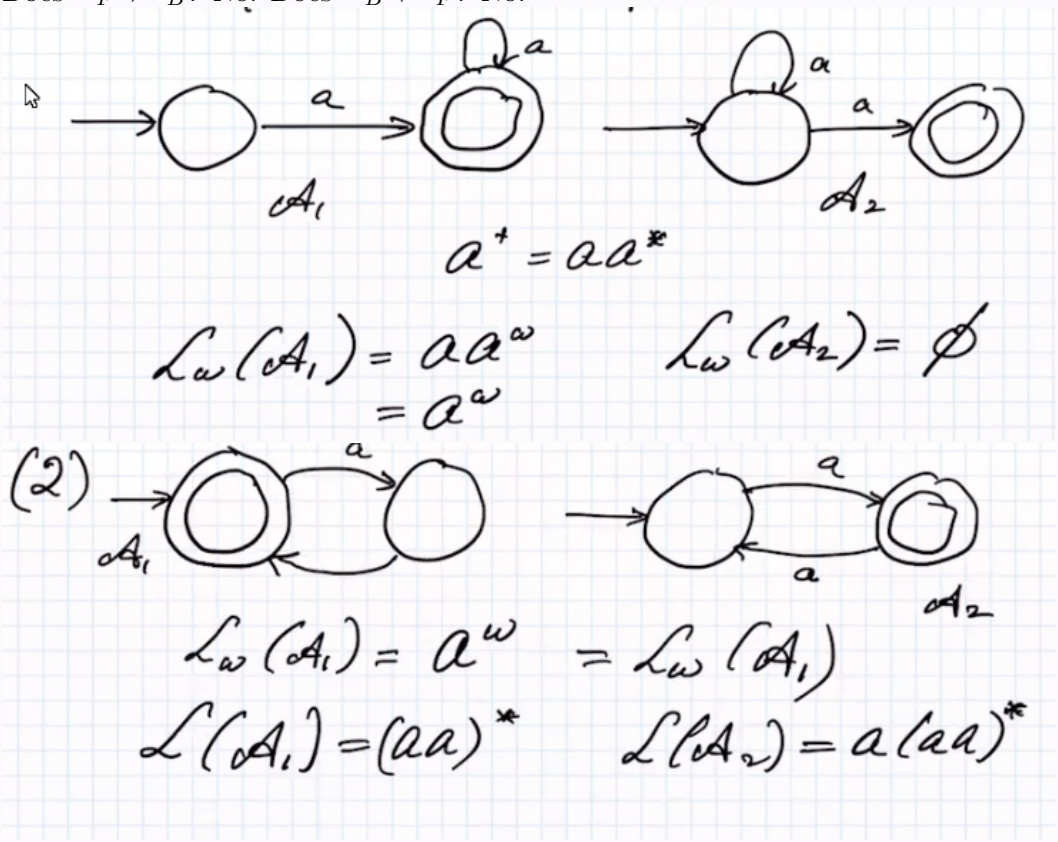
Lemma 5.7

$L_\omega(A) \neq \varnothing$ iff $\exists$ a reachable accept state that belongs to a cycle.

$A_1 \equiv_F A_2$ they are the same as NFA. $L(A_1) = L(A_2)$. $A_1 \equiv_B A_2$ they are the same as NBA. $L_\omega(A_1) = L_\omega(A_2)$.

Example 5.8

Does $\equiv_F \Rightarrow \equiv_B$? No! Does $\equiv_B \Rightarrow \equiv_F$? No!



Idea here is that it distinguishes even length from odd length, but infinity length is both even and odd.

Theorem 5.9

NBAs and DBAs are not equivalent. There is no DBA that can recognize $(a+b)^*b^\omega$. This is what makes omega automata much trickier.

Proof. Suppose we have a DBA for this language. We can define δ^* as usual on the underlying DFA. Consider the word $b^\omega \in (a+b)^*b^\omega$

Since b^ω is accepted, we must hit an accept state at some point. Therefore $\delta^*(q_0, b^{n_1})$ is an accept state. $b^{n_1}ab^\omega \in L$. This has to hit an accept state again after reading a. You can continue this process. Have to hit the same accept state twice. But then $\exists i, j, i < j$ where

$$\delta^*(q_0, b^{n_1}ab^{n_2}\cdots ab^{n_i}) = \delta^*(q_0, b^{n_1}ab^{n_2}\cdots ab^{n_j})$$

But then there is a loop. So you can include infinitely many b in the language. Contradiction. So NBA are strictly more expressive than DBA. □

Let $\alpha \in \Sigma^*\omega$ and define $\alpha|_n \in \Sigma^*$ to be the length of n finite prefix.

Definition 5.10. For $W \subseteq \Sigma^*$ we define $\vec{W} = \{\alpha \in \Sigma^\omega \mid \exists \text{ infinitely many } n \in \mathbb{N} s.t.\ \alpha|_n \in W\}$ Words where you can always make the prefix in W longer. $\vec{W}$ is called a limit language.

Theorem 5.11

An ω-regular language L is recognizable by a deterministic BA iff there is a regular language W such that $L = \vec{W}$.

Proof. DIY □

Theorem 5.12

DBA are closed under complement.

If A is a DBA, $\exists$ an NBA A' such that $L_\omega(A') = \Sigma^*\omega \setminus L(A)$.

$A' = (Q', \Sigma, \delta', Q_0', F').A = (Q, \Sigma, \delta, Q_0, F)$.

$$Q' = (Q \times \{0\}) \cup ((Q \setminus F) \cup \{1\})$$
$$Q_0' = Q_0 \times \{0\}$$

§6 05-12

§6.1 Basic Propositional Logic

Lot's of confusion about what is meant by a true statement vs valid statement vs theorem.

§6.1.1 Syntax

PROP $= \{p, q, r, \cdots\}$ are propositional variables.

φ, ψ are meta-variables for describing generic propositional formulas.

Formulas are defined inductively. Inductive because you have some formulas and you give rules for building new formulas from old ones.

This is syntax. What I'm allowed to write, but explaining anything about what they mean. Explaining what they mean is called semantics.

1. T for true is a formula, and $\perp$ for false is a forumla.
2. Any propositional variable is a formula
3. If φ is a formula, so is $\neg\phi$.

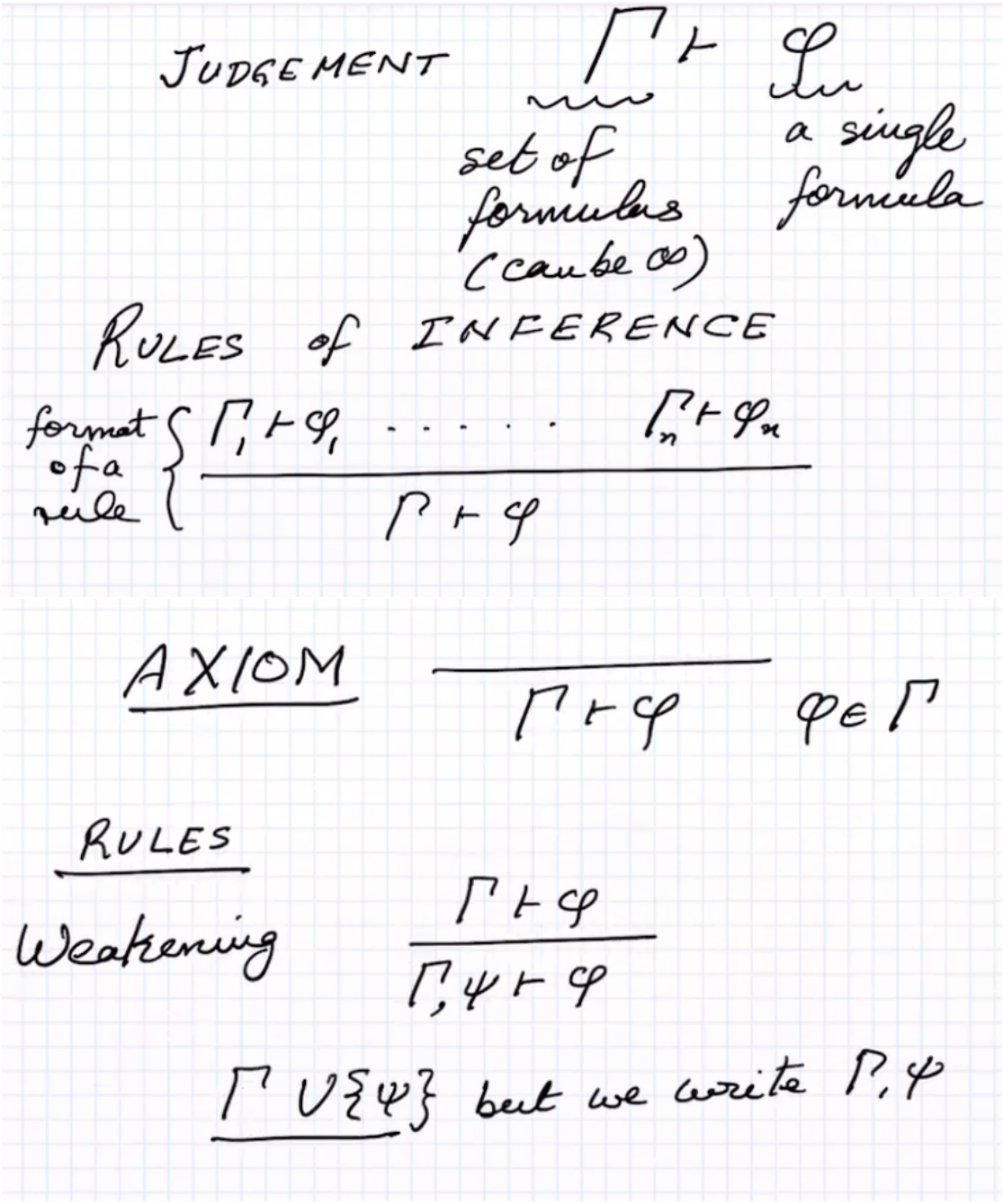
4. If φ and ψ are formulas, so are $\varphi \wedge \psi,\ \varphi \vee \psi,\ \varphi \Rightarrow \psi$

§6.1.2 Proof Theory

How to use and manipulate formulas in deduction. Proof theory is not about being true. It's about true.

Gamma is the set of assumptions you are making.

$\gamma \cup \{\psi\}$ but we write γ, ψ



You can do introduction or elimination.

$\neg\varphi$ is a marco for $\varphi \Rightarrow bot$.

A proof is a tree whose leaves are axioms, the nodes are rules instances, and the rute is a single judgement $\gamma \vdash \varphi$.

If the root has the form $\vdash$ we say that φ is a theorem.

§6.1.3 Semantics

Interpret logical formulas as elements of a simple mathematical structure.

A valuation v : PROP $\to \{0,1\}$. Extend v to a map on formulas by structural induction.

$$v(T) = tt \quad v(\bot) = ff$$
$$v(\varphi) \underbrace{\wedge}_{\text{Boolean algebra}} v(\varphi') = v(\varphi \underbrace{\wedge}_{\text{Syntax}} \varphi')$$

If $\models \varphi$, then $\forall v,\ v(\varphi) = tt$. Such a formula is called a tautology.

$\vdash$ denotes a syntactic implication while $\models$ denotes a semantic implication.

> **Theorem 6.1** (Soundness)
> If $\gamma \vdash \varphi$ then $\gamma \models$ in particular if $\vdash \varphi$ then $\models \varphi$.

> **Theorem 6.2** (Completeness)
> If $\gamma \models \varphi$ then $\gamma \vdash \varphi$.

Suppose that γ has no valuation such that $\forall \varphi \in \gamma,\ v(\varphi) = tt$. Then we say that γ is unsatisfiable.

> **Theorem 6.3** (Compactness)
> If γ is unsatisfiable, then some finite subset of γ is unsatisfiable.

Remark 6.4. Tepmoral logic fails this property.

§6.2 Temporal Logic

as opposed to propositional logic. The basic atomic properties can change their truth values in time.

A mir Puuli showed this is very useful for reasoning about code execution especially concurrent programs.

§6.2.1 Syntax

PROP $= \{p, q, r, \dots\}$. Formulas $\neg, \wedge, \dots$. Two temporal operators O, u. If φ is a formula then 0φ is a formula. If φ, ψ are formulas, then $\varphi u \psi$ is a formula.

$\varphi = true | p | \varphi_1 \wedge \varphi_2 | \neg\varphi | O\varphi | \varphi u \psi$

Intuition. Instead of a valuation we have a linear sequence of valuations. We will call these valuations states. The entire sequence is called a history.

Defined operators.

§6.3 Labelled Transition System

$(S, Act, \rightarrow, I, AP, L)$ S states perhaps infinite. Act actions. $\rightarrow \subseteq S \times A \times S$. represents function a $s \rightarrow s'$ $I \subseteq S$. initial states. AP are the set of atomic propositions.

§7 05-13

§7.1 Bisimulation

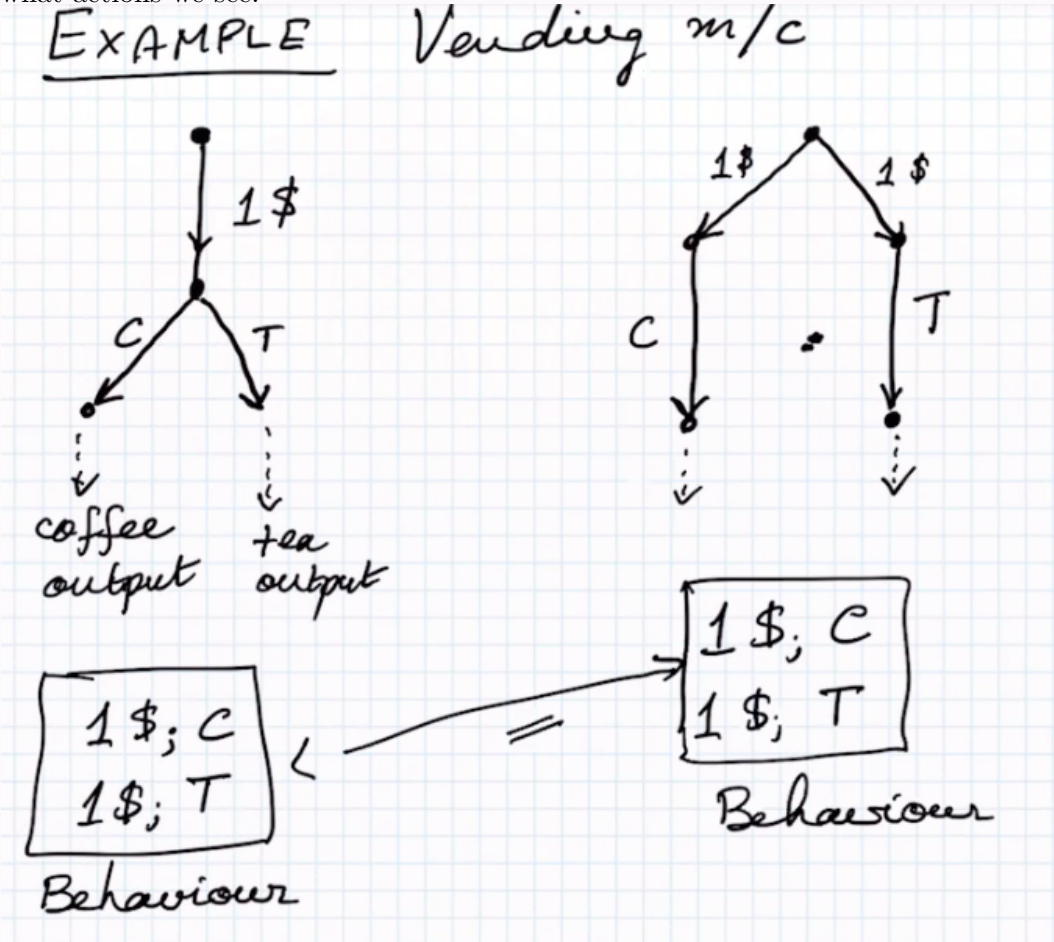
Transition systems as models of software or hardware or embedded systems.

Nondeterminism, need not be finite state.

When are two systems observably the same?

Example 7.1 (Vending Machine)

Difference from DFA, some actions get rejected. The rejection is different from the rejection in an NFA because really that encodes a state where you can't escape, while this is more true rejection. No start states or accept states. Just interested in what actions we see.



Internal vs. external choice is what makes these two machines different. This shows how sequences are inadequate description of the behavior. So how can we actually compare to LTS? With bisimulation.

Definition 7.2 (LTS (Labeled Transition System))**.**

$$S \to \text{ set of states, perhaps infinite}$$
$$A \to \text{ set of actions, finite}$$
$$\to \subseteq S \times A \times S$$
$$(s, a, s') \in \to, \quad s \xrightarrow{a} s'$$

From state s, if action a is performed, you can end up in s'.

Definition 7.3 (Bisimulation (semi-formed))**.** When comparing two systems, we form a binary relation between the states of a simple. This isn't a big deal because we could merge the two systems.

We say $s, t \in S$ are bisimilar (written $s \sim t$) if

$$\forall a \in A \; s \xrightarrow{a} s' \Rightarrow \exists t' s.t. t \xrightarrow{a} t' \text{ and } s' \sim t'$$
$$\forall a \in A t \xrightarrow{a} t' \Rightarrow \exists s' s.t. s \xrightarrow{a} s' \text{ and } s' \sim t'$$

If s can do something, t can do the same thing. And they may end up in different states, but those states themselves will be bisimilar. This should make you uneasy because it's an inductive definition with no base case.

There are 4 ways of clarifying this definition.

1. We define an $\mathbb{N}$ indexed family of equivalence relations (infinitely many of them) as follows: $s \sim_0 t$ always. Then

   $$s \sim_{n+1} \; if \; \forall a \in A \; s \xrightarrow{a} s' \Rightarrow \exists t' s.t. t \xrightarrow{a} t' \text{ and } s' \sim_n t'$$
   vice versa

   Now we say that $\sim = \cap_n \sim_n$

   In the vending machine example $s_0 \sim_1 t_0$ but $s_0 \not\sim_2 t_0$ and hence they are certainly not bisimilar systems.

2. R the family of equivalence relations $\subseteq S \times S$ order by inclusion. Smallest is everything is related to itself. Largest is everything related to everything. This forms a complete lattice. $F : R \to R$.

   $$sF(R)t \text{ if } \forall a s \xrightarrow{a} s' \Rightarrow \exists t' s.t. t \xrightarrow{a} t' \text{ and } s' R t'$$
   vice versa

   F is easily seen to be monotone i.e. if $R_1 \subseteq R_2$ then $F(R_1) \subseteq F(R_2)$. It follows that there is a unique greatest fixed point. i.e. a special $\sim \in R$ such that $F(\sim) = \sim$ and if R is any relatoin such that $F(R) = R$ then $R \subseteq \sim$. This is called fixed point bisimilarity.

3. We saw R is a dynamic relation or a bisimulation relation if whenever sRt then .

   $$\forall a \forall s' \; s \xrightarrow{a} s' \Rightarrow \exists t' s.t. t \xrightarrow{a} t' \text{ and } s' R t'$$
   vice versa

   Note that this is not circular. R is given somehow and it may or may not have this property. We say that $s \sim t$ if $\exists R$, a bisimulation relation with sRt.

   Fact 7.4. If R_i is any family of bisimulation relations, then $\cup_i R_i$ is also a bisimulation as is $\cap_i R_i$. $\sim = \cup_{R_a} R$.

   Easy to see that (2) and (3) are equivalent. (1) is not equivalent without a further assumption.

§7.2 Lattice Theory and Fixed Points

Remember that a poset is a partially ordered set. i.e. a set equipped with a partial order.

Given $(S, \leq)$, we say that u is the least upper bound of $X \subseteq S$ if

$$\forall x \in X,\ x \leq u \text{ and } \forall y \in S, if\ \forall x \in X, x \leq y \Rightarrow u \leq y$$

If $X \subseteq S$ we say v is a lower bound of x if $\forall x \in X,\ v \leq x$. If v in addition is greater than any other lower bound, we call it the Infimum.

Definition 7.5. In a lattice there is a supremum and infimum for every pair of elements. They are not necessarily unique. By induction it follows that every finite set of elements has a supremum and infimum. Infinite numbers may not.

Definition 7.6 (Complete Lattice)**.** A lattice is complete if every subset has a least upper bound.

Fact 7.7. It follows that every set has a greatest lower bound.

The least upper bound of $\varnothing$ is a least element of the whole set.

Proof. Let $X \subseteq L$, where L is a complete lattice. Let $V = \{v \in L \mid \forall x \in X\ v \leq x\}$ be the set of lower bounds of X.

Let $g = \sup(V)$. Claim g is $\inf(X)$.

Note 7.8. $\forall x \in X,\ \forall v \in V,\ v \leq x$. i.e. $\forall x \in X$, x is an upper bound for V so since g is the lesat upper bound for V we have that $\forall x \in X,\ g \leq x$, i.e. $g \in V$.

Since g is an upper bound for V it is greatre than any element of V so it is the glb of X.

□

Theorem 7.9

If $f : L \to L$ is monotone and L is a complete lattice, then the fixed points of f, i.e. $x\ s.t.\ f(x) = x$ forms a complete lattice in its own right. In particular there is a least fixed point and a greatest fixed point. There is a least fixed point and a greatest fixed point.

Proof. Find the upper bound of set of inflationary points and show that it is fixed. □

Note to be a bisimulation relation means $R \subseteq F(R)$. Least upper bound of $\{R \mid R \text{ is a bisimulation}\} = \cup_R R = gfp(F)$ (greatest fixed point). (2) and (3) are really the same. (1) is not the same unless the transition system has a special property $\forall s, a\{t \mid s \to_a t\}$ is finite. This is called image-finiteness.

If TS is image finite then (1) is equivalent to (2) and (3). The definition of bisimulation is an example of a co-inductive definition.

§7.3 Logical Characterization of Bisimulation

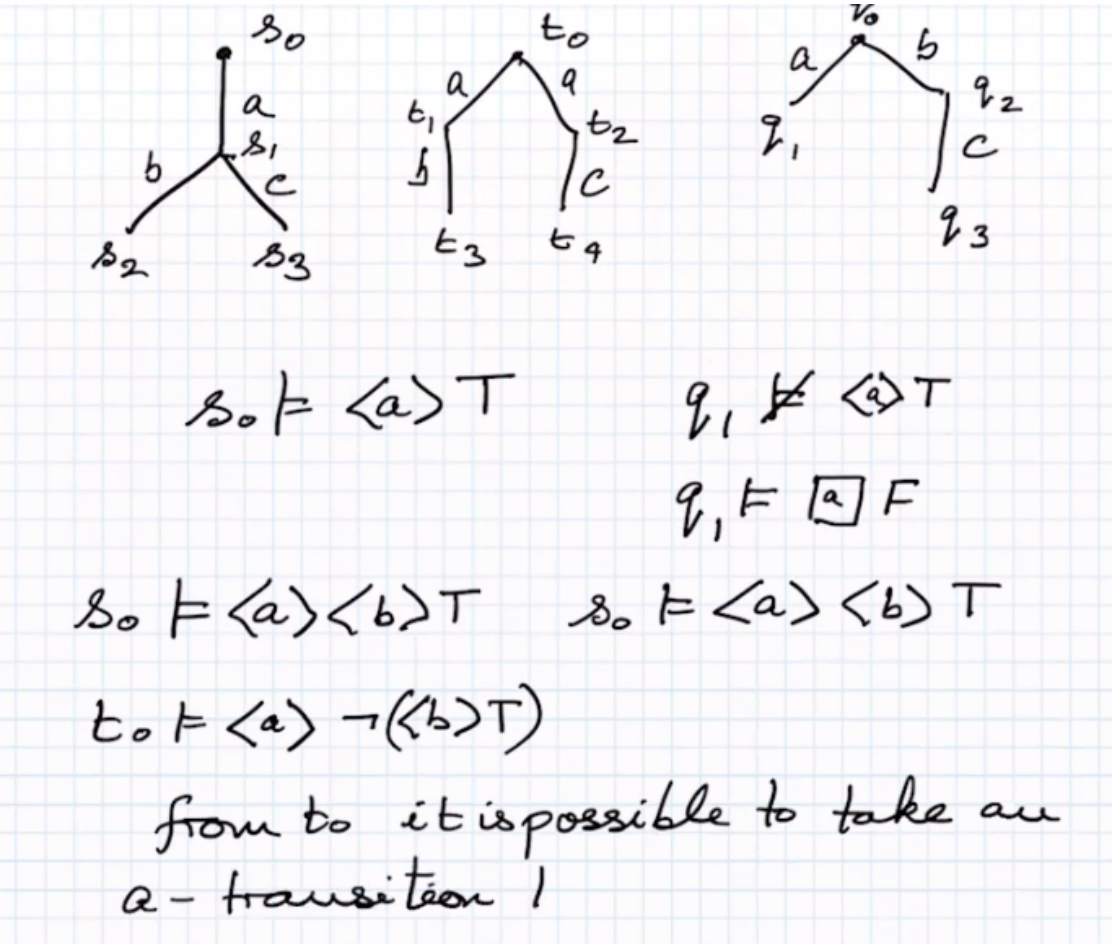
Given an LTS, we define a modal logic called Hennessy-Milner Logic.

$$\varphi ::== T \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \langle a\rangle\varphi$$

$s \models \langle a\rangle\varphi\rangle$ if $\exists t$ *s.t.* $s \to_a t$ and $t \models \varphi$. At least one state satisfies.

$[a]\varphi = \neg\langle a\rangle\neg\varphi$

Every state t such that $s \to_a t$ must satisfy φ.



Theorem 7.10
$s \sim t$ if and only if $\forall \varphi s \models \varphi \Leftrightarrow t \models \varphi$ (image finiteness assumed). Don't be afraid of infinite conjunctions.

Proof. We define a new equivalence relation $\approx$. $s \approx t \Leftrightarrow \forall \varphi\ s \models \varphi \Leftrightarrow t \models \varphi$.

Idea is to prove that $\approx$ is a bisim relation. Suppose $s \approx t$ and suppose $s \to_a s'$. We want to show that $\exists t'\ s.t. t \to_a t'$ and $s' \approx t'$.

Assume that s and t are not bisimilar. Then $\forall t'\ s.t.\ t \to_a t',\ s' \not\approx t'$. There are only finitely many such $t' : t'_1, \ldots, t'_n$. □

§7.4 Probabilistic Bisimulation and Logical Characterizaiton

Probabilistic transition systems. $s \models \langle a \rangle_q \varphi$. $a \in Act,\ q \in Q \cap [0,1]$ The probability of winding up in a state sastisfying φ after doing an a action in state $s \geq q$. Larsen and Skou proved logical characterization using

$$\langle a \rangle_q \varphi \text{ and } \neg\varphi \text{ and } \varphi_1 \wedge \varphi_2$$

They also assumed a very strong finite branching property. And probabilities must be integer multiples of some fixed rational number.

AMAZINGLY proved logical characterization with no finite branching assumption and no negation in the logic. We alo got logical characterization of simulation.

§8 05-14

§8.1 Learning Automata

Dana Angluin (1987)

Model. She assumed the existence of a teacher: minimally adequat teacher.

It's always regular languages. Trying to learn a DFA. You can ask "is w in L?".

After collecting some information you propose an answer. A teacher says "Yes" or "No" and here is a counter example: a word in L that your proposed machine rejects or a word not in L which your machine incorrectlys accepts. Teacher's choice not yours.

Theorem 8.1
With polynomially man queries you are guaranteed to learn the correct unique minimal DFA for L (even if teacher has a different more complex version).

Basic data structure: Observation table. $S, E \subseteq \Sigma^*$, both finite. Helpful to think of S as states and E as experiments.

Assume you have a table with certain properties satisfied. We can propose a DFA

An observation table is said to be closed when

$$\forall t \in S \cdot \Sigma,\ \exists s \in S\ s.t.\ row(t) = row(s)$$

An observation table is consistent when

$$\forall s_1, s_2 \in S, row(s_1) = row(s_2) \Rightarrow \forall a \in \Sigma\ row(s, a) = row(s_2, a)$$

An observation table desribes a DFA iff it is closed and consistent.

Algorithm:

1. When it's not closed, you ask for more queries.
2. When it's not consistent, you expand the number of experiments.
3. When it's closed and consistent, you present your DFA. If it's wrong, you use the counter example to expand the table and start over.

> **Example 8.2**
> This is an extended example going through the process of finding the DFA. I won't write it down. These notes do a great job

A similar algorithm works for weighted automata.

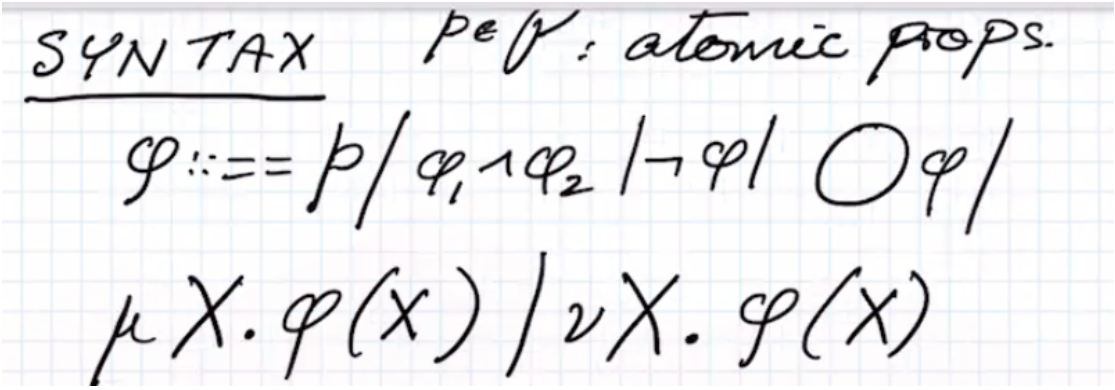
This is kinda automata theory meets machine learning. Another application is in trying to understand what an RNN does. The question of interpreting what an RNN does once it has been trained. It's important to have explainable AI.

Idea: Use an RNN as the teacher and extract a DFA as the explanation of what the RNN is doing. Gail Weiss. Alika Utepoua and Prakash are exploring this.

§8.2 Fixed Point Operators and LTL

$\Box\varphi = O\varphi \wedge OO\varphi \cdots$

This leads to infinitely many conjunctions. We should be able to prove this with induction.



X is a new syntacting entity. It is a variable that can range over formulas. μX and νX are variable binders. They are called fixed point operators.

Recall 8.3. Every monotone function from a complete lattice to itself has a least fixed point and a greatest fixed point. μ stands for least fixed point. ν stands for greatest fixed point.

We are not allowed to put a fixed point operator binding an X unless X appears in the scope of an even number of negations.

$\mu X.\varphi \wedge \neg X$ and $\mu X.X \Rightarrow \varphi$ are not allowed because they involve negated X. Remember that $x \Rightarrow y$ is the same as $\neg x \vee y$.

We defined the semantics in terms of sequences $\sigma \models \varphi$.

$$[\![\varphi]\!] = \{\sigma \mid \sigma \models \varphi\}$$

gives a definable set. Given a formula with a free variable, we interpret it not as a set but as a function from sets to sets.

$$[\![\varphi(X)]\!] = S \mapsto [\![\varphi[S/X]]\!]$$

$\varphi[S/X]$ notation for substitute the set S regarded as a formula for X in φ.

e.g. $\varphi(X) = \psi \wedge OX$.

$[\![\varphi(X)]\!](S) = \{\sigma \mid \sigma \models \psi \wedge \sigma[1..] \in S\}$

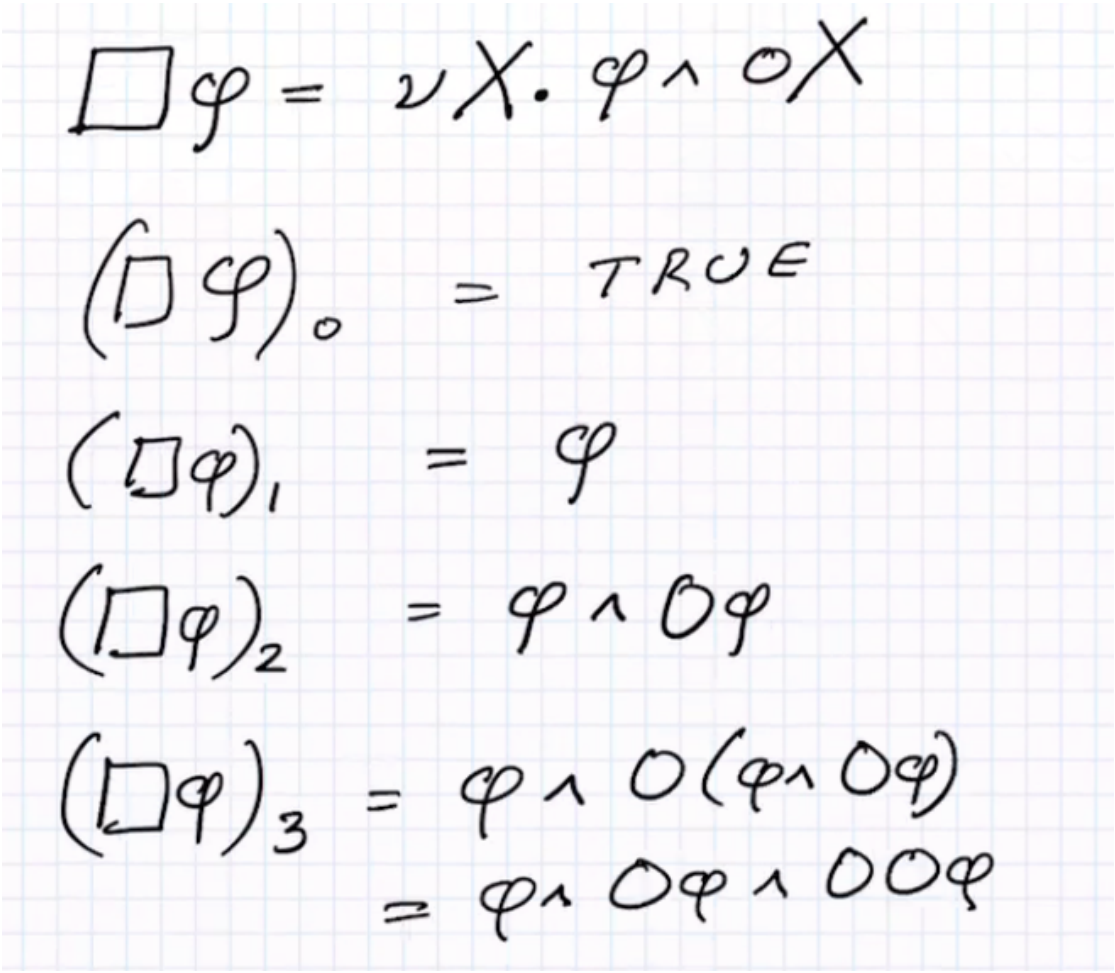
Fact 8.4. With the restriction these functions are always monotone and $P(\Sigma^\omega)$ is a complete lattice.

Now we define

$$[\![\mu X.\varphi(X)]\!] = lfp[\![\varphi(X)]\!]$$
$$[\![\nu X.\varphi(X)]\!] = gfp[\![\varphi(X)]\!]$$

It's a nightmare to interpret nested fixed points in human terms (although it is still well defined).

$\Box\varphi = \nu X.\varphi \wedge OX$



§9 05-19

Previously we were working with finite memory machines. Limitations on what we could do even when we came to omega languages.

Two ways to recognize regular languages. Regular languages in terms of deterministic finite automata, and regular expressions.

Now we are getting infinite memory. A finite state machine plus an auxiliary stack. Then in the next module we will get two stacks.

§9.1 Context Free Languages

We will define CFL's in terms of grammars.

We will define a grammar as a way to generate strings in the language. There is a lot more structure than regex had. Rules for producing strings.

Definition 9.1 (Context-free Grammar)**.** A context-free grammar consists of

1. a set of symbols called terminals (Σ alphabet)
2. Another set of symbols called non-terminals or variables V.
3. Both V, Σ are finite, and $V \cap \Sigma = \varnothing$.
4. A special variable called the start symbol, usually but not always S.
5. A set of rules called PRODUCTIONS.

$$A \to \alpha,\ A \in V,\ \alpha \in (V \cup \Sigma)^*$$

Example 9.2

Let $\Sigma = \{a, b\}$. These are our terminal symbols. Terminal suggests that something has to be terminated. Variable on the other hand suggests something that can change. $V = \{S\}$. Just the start symbol. Rules:

$$S \to \epsilon,\ S \to aSb$$

Apply any rule you like to replace a non-terminal with the RHS of a rule. It doesn't matter what the surrounding context is. This is why it's called context free.

$$S \to aSb \to aaSbb \to aaaSbbb \to aaabbb$$

This allows you to create $\{a^n b^n \mid n \geq 0\}$, which is not a regular language. CFL's are clearly more powerful than regular languages.

When you obtain a string with no more non-terminals, you have generated a string in the context free language.

$$L(G) = \{w \in \Sigma^* \mid w \text{ can be generated from the start symbol } S\}$$

Unlike with DFA the language is defined and then you have to recognize it, here you are generating the language.

Example 9.3 (Grammar for arithmetic expressions)

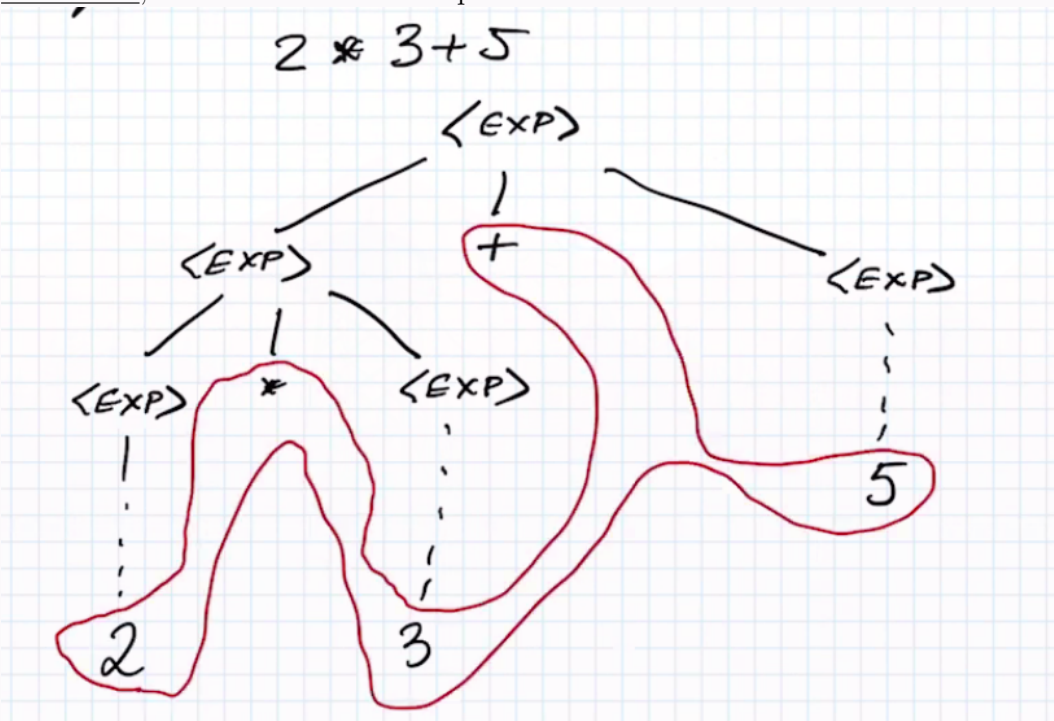
All modern programming languages, the syntax is a context free language, oftentimes it is even more restricted than that. Cobalt does not have a context free grammar, but now we've realized that this is a stupid idea. Every modern programming language is context free.

$$\Sigma = \{0, 1, \cdots, 9, \times, +, (,)\}$$
$$V = \{\langle EXP\rangle, \langle NUM\rangle, \langle NZ\rangle, \langle N\rangle\}$$

Start symbol $< EXP >$. Notation to save space.

$$\begin{aligned}
\langle EXP\rangle &\to \langle EXP\rangle + \langle EXP\rangle \mid \langle EXP\rangle \times \langle EXP\rangle \mid \langle EXP\rangle \mid \langle NUM\rangle \\
\langle NUM\rangle &\to 0 \mid \langle NZ\rangle \\
\langle NZ\rangle &\to 1\langle N\rangle \mid 2\langle N\rangle \mid \cdots \mid 9\langle NZ\rangle \\
\langle N\rangle &\to 0\langle N\rangle \mid 1\langle N\rangle \mid 2\langle N\rangle \mid \cdots \mid 9\langle NZ\rangle \mid \epsilon
\end{aligned}$$

NZ is to allow you to write zero but prevent you from writing 00. To properly display derivations, we will use trees called parse trees.



Definition 9.4 (Ambiguity in Context Free Languages)**.** The grammar allows two (or more) distinct ways of parsing an expression. Such a grammar is called ambiguous. An example of bad grammar engineering. A given language does not have a unique grammar. With regular languages there was a unique minimal automaton; there is no such concept in context free languages.

It is possible to redesign the grammar so that it is not ambiguous, at least in this case. There exist languages for which no unambiguous language is possible. Then it is inherently ambiguous.

§9.2 Designing CFG

Example 9.5 (Counting CFG)

How to produce the language $L = \{a^n b^{2n} \mid n \geq 0\}$. Rules:

$$S \to aSbb \mid \epsilon$$

Keep going until you are happy and then wipe out the S. Notice that context free languages can count

Example 9.6 (Palindromes)

$L = \{x \in \Sigma^* \mid x = x^{REV}\}$. Rules:

$$S \to aSa \mid bSb \mid a \mid b \mid \epsilon$$

Example 9.7

$L = \{x \in \Sigma^* \mid \#_a(x) = \#_b(x)\}$. A bit harder than example number 1. To make it easier let $d(x) = \#_b(x) - \#_a(x)$ and $L = \{x \mid d(x) = 0\}$.

Let $x \in L$, and u be the shortest prefix of x such that $d(u) = 0$ and suppose u starts with b. u must end with a because it starts with b. So $u = bva$ and $d(v) = 0$.

$x = uz \Rightarrow d(z) = 0$

$$S \to \epsilon \mid bSaS \mid aSbS$$
alternatively
$$S \to aSb \mid bSa \mid SS \mid \epsilon$$

Fact 9.8. Every regular language is a CFG.

Definition 9.9. A CFG is said to be in Chomsky Normal Form if every rule has the form

$$A \to BC \text{ or } A \to a$$

Here A, B, C are non-terminals and a is a terminal. We do not allow rules like $A \to \epsilon$ except for the start symbol. Some resources say that even the start symbol can't go to epsilon.

Theorem 9.10

Every context free language is generated by a context free grammar in chomsky normal form.

Proof. The idea is to systematically get rid of bad rules. e.g. $A \to \epsilon$ is a forbidden rule so throw it out and look at any rule with A on the RHS e.g. $B \to \alpha_1 A \alpha_2 A$. Add all possible rules with some of these A's removed. So in this case add the following rules.

$$B \to \alpha_1 \alpha_2 A$$
$$B \to \alpha_1 A \alpha_2$$
$$B \to \alpha_1 \alpha_2$$

Many other modifications of this type. Consider

$$A \to x_1 x_2 x_3 \cdots x_n, \ x_i \in \Sigma \cup V$$
becomes
$$A \to x_1 A_1, x_1 \to x_1 \text{ if } x_1 \in \Sigma$$
$$A_1 \to x_2 A_2, \ \ldots$$

All these A_i are new non terminals. □

You can see that there are lots of ways to modify the grammar while preserving the languages.

§9.3 Closure Properties of CFL's

1. L_1, L_2 are CFL's then $L_1 \cup L_2$ is a CFL.

   Let S_1 be the start for G_1 and $L(G_1) = L_1$, and S_2 be the start for G_2 and $L(G_2) = L_2$. Take all the rules together and add $S \to S_1 \mid S_2$.

2. $L_1 \cdots L_2$ is also a context free language.

   $$\to S_1 S_2$$

3. L^* is a context free language if L is. Add a new start symbol S' and add the rule

   $$S' \to SS' \mid \epsilon$$

4. CFL's are <u>not</u> closed under

   a) intersection

   b) complement

Easy proofs becaues of the power of grammars rather than fiddling with automata.

Example 9.11

$L_1 = \{a^n b^n c^m \mid n, m \geq 0\}$ is a CFL, and so is $L_2 = \{a^m b^n c^n \mid n, m \geq 0\}$. Then $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$. The idea is that when there is a buffer, you can't count. This is our favorite example of something that is not context free.

Tomorrow we will see a new pumping lemma for context free languages.

Note also that $L = \{a^n b^n c^n \mid n \geq 0\}$ is not a CFL but $\overline{L}$ is a CFL! Also see that if it were closed under complementation and under union it would have been closed under intersection. So it cannot possibly be closed under complementation.

Proposition 9.12

If L is a CFL and R is a regular language then $L \cup R$ is a CFL.

Fact 9.13. There is an algorithm to look at a grammar and decide if $L(G) = \varnothing$. It's easy to write grammars that don't produce anything. For example if you don't give a rule for the start symbol. No matter how many other rules you have it won't produce anything. It might be that you never get rid of the non-terminals, which by definition are not in the language. The point is that there is an algorithm that can look and decide.

Proof. DIY □

Algorithm to decide if $w \in L(G)$. Works on general context free grammar (even ambiguous ones). To reduce the amount of book keeping assume G is in chomsky normal form.

Typical dynamic programming algorithm. Given $G = (V, \Sigma, S, P)$. P for the set of rules because they are referred to as productions. Input $w = a_1 \cdots a_n \in \Sigma^*, a_i \in \Sigma$.

The idea is to work bottom up to construct a possible derivation for w. To get a specific letter a we must have used a rule of the form $A \to a$.

We define inductively a 2-indexed family of subsets of V.

$$i \leq j,\ X_{ij} \coloneqq \{A \in V \mid A \to a_i \cdots a_j\}$$
$$X_{ii} = \{A \in V \mid A \to a_i\}$$

When we come to compute X_{ij}, we assume that we have already computed X_{ij}, X_{kj} for all k between i and j When we come to compute X_{ij}, we assume that we have already computed X_{ik}, X_{kj} for all k between i and j. So if $B \in X_{ik}$, $C \in X_{(k+1)j}$ and $A \to BC$ is a rule in G then we put $A \in X_{ij}$.

§9.4 Algorithms For CFLs

1. $L(G) = \varnothing$ is decidable.
2. $L(G)$ infinite is decidable.

3. $L(G) = \Sigma^*$ is undecidable. This is surprising at first. Provably impossible.

Start of a topic called parsing. Heavily studied in a compiler class. Reached such a high state of understanding that nobody really needs to write parsers anymore because people have written parser generators.

§9.5 Pushdown Automata

This is an NFA + Stack (1 stack only, no more). Emphasis on NFA. Nondetermism is built in to the structure. DFA pushdown automata are not equivalent to NFA pushdown automata. There is an equivalent deterministic context free language.

> **Theorem 9.14**
> Every CFL can be recognized by a PDA (pushdown automata) and every language recognized by a PDA is a CFL. A language is context free if and only if it is recognized by a PDA.
>
> Use judgement. Sometimes it is much easier to do a proof via PDA, and sometimes easier via grammar.

Definition 9.15 (Pushdown Automata)**.** Q represents the states (finite).

Σ represents the input alphabet (finite). $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$.

γ represents the stack alphabet $\gamma \supset \Sigma$, $\gamma_\epsilon = \gamma \cup \{\epsilon\}$

$\delta : Q \times \Sigma_\epsilon \times \gamma_\epsilon \to P_f(Q \times \gamma_\epsilon)$

P_f is the finite power set.

Reading a letter involves (i) looking at the input and the top of the stack, (ii) changing the state, (iii) poping the stack or pushing something on the stack or both and then move to the next input symbol.

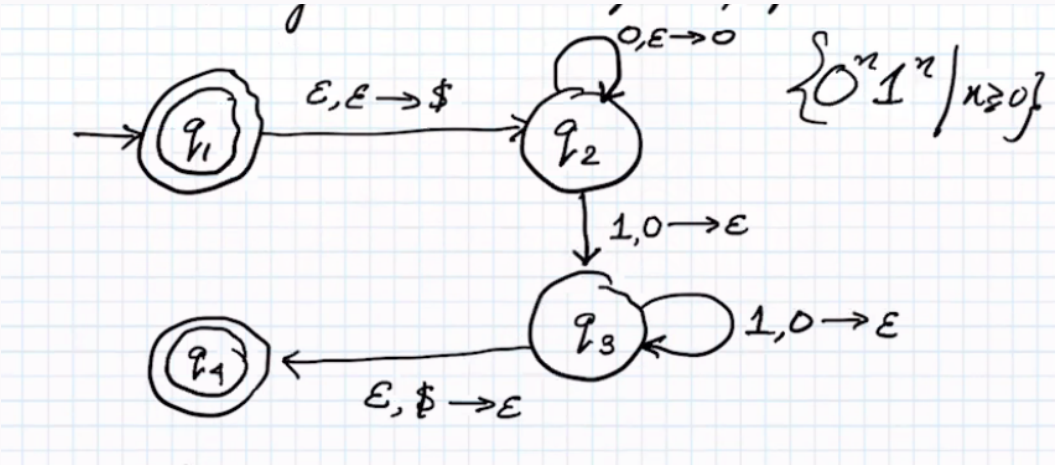
Notation, $a, b \to c$. b is seen on top of the stack, a is the next input symbol, pop the stack and replace b with c. a may be ϵ which would mean don't read input

Reading a letter involves (i) looking at the input and the top of the stack, (ii) changing the state, (iii) poping the stack or pushing something on the stack or both and then move to the next input symbol.

Notation, $a, b \to c$. b is seen on top of the stack, a is the next input symbol, pop the stack and replace b with c.

a may be ϵ which would mean don't read input. b may be ϵ which would mean just push c. c may be ϵ which would mean just pop the stack.

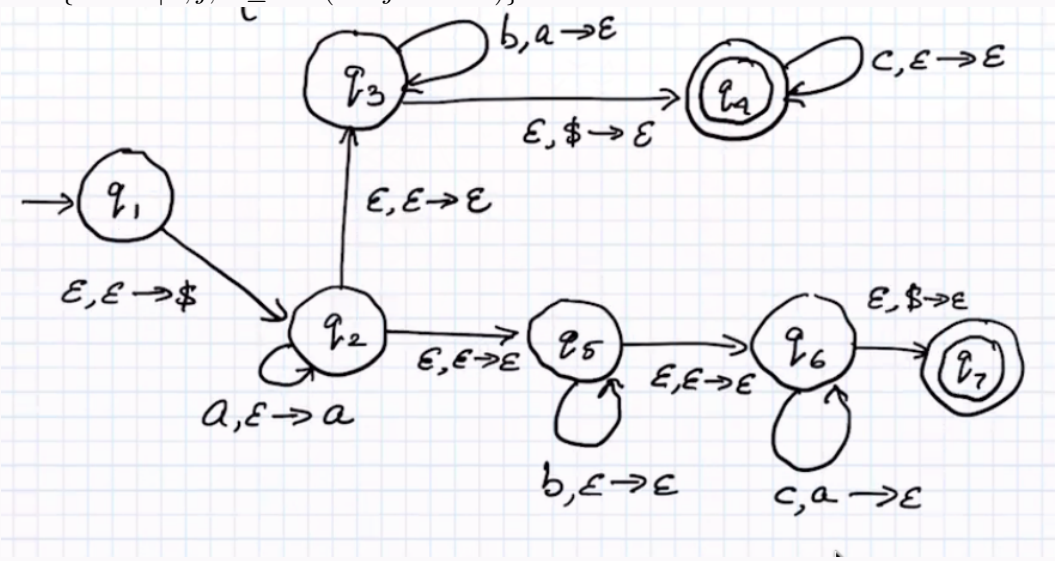
Example 9.16



You are required to go all the way to the end of the input or you are jammed and rejected. This is an example of a deterministic machine but in general you can have non determinism.

Example 9.17

$L = \{a^i b^j c^k \mid i, j, k \geq 0 \wedge (i = j \vee i = k)\}$



Note 9.18.

1. Acceptance only happens at the end
2. A PDA cannot "decide to jam" when there are valid moves.

Note 9.19. There is an alternative notion of acceptance: no accept states, we accept if the stack is empty at the end of the input. These are equivalent.

$a^n b^n c^n$ cannot be accepted by a PDA because we can the a's and count off the b's

but then we don't know what to do with the c's. If we had 2 stacks, we could accept $\{a^n b^n c^n \mid n \geq 0\}$ by pushing to the second stack as we pop from the first stack.

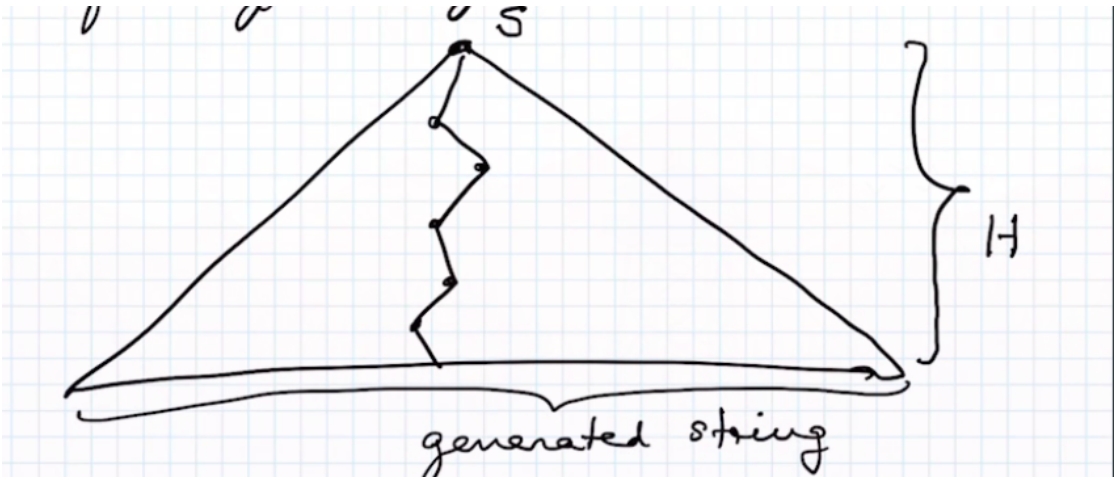
$\{a^n b^n c^n d^n \mid n \geq 0\}$ still only needs 2 stacks, because you can go back and forth between the two stacks. 2 stacks is enough for any number.

Fact 9.20. A PDA with two stacks is universal. i.e. it has the same power as a turing machine and can compute any computable function.

People realized that complexity is more significant than power, because the power of everything is the same once you get to two stacks, two queues, etc.

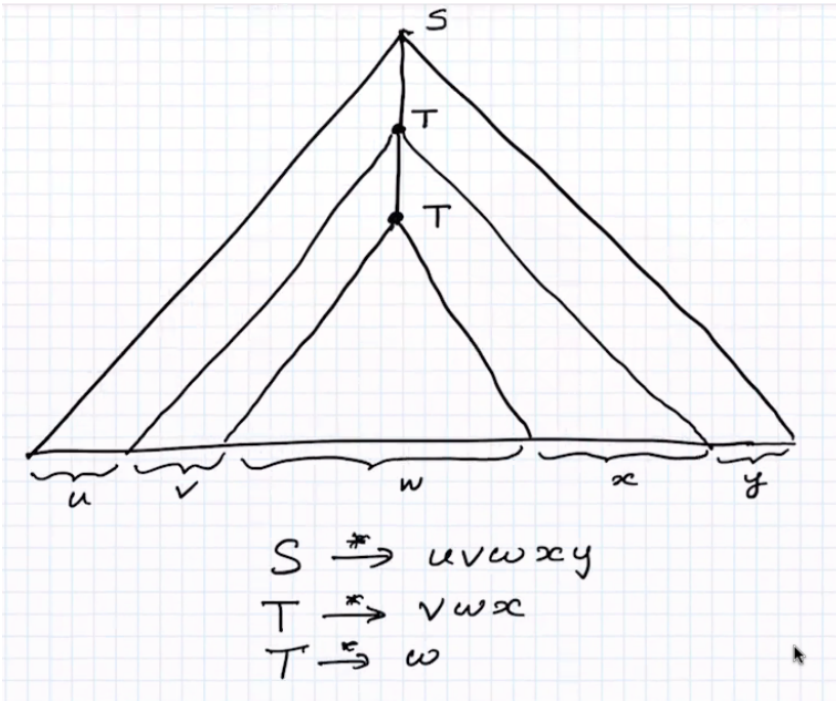
§10 05-20

Proving pumping lemma we used the way that DFA has finitely many states. But the size of the stack can be infinite. Instead we use the fact that a grammar for a CFL has only finitely many variables.



Height of parse tree increases less quickly than the length of the word. For a long enough word, $H >$ # of non terminals in the grammar.

This means that at some point there has to be a repeated non-terminal.



It is not possible that both v and x are empty. This is true if we assume Noam Chomsky form. But then we can change the second T into generating vwx which would mean thta $uv^2wx^2y \in L$. And you can do this again which would mean $uv^3wx^3y \in L$. We conclude that

$$\forall i \geq 0 \; uv^iwx^i \in L$$

Lemma 10.1 (Pumping Lemma for Context Free Languages)

$$\forall CFL, L, \; \exists p > 0$$
$$\forall s \in L \; |s| \geq p$$
$$\exists u, v, w, x, y \in \Sigma^* \text{ s.t. } s = uvwxy, \; |vx| > 0 \; , |vwx| \leq p$$
$$\forall i \geq 0, \; uv^iwx^iy \in L$$

No longer guaranteeing that is happens early on in the string.

Lemma 10.2 (Contrapositive of Pumping Lemma for Context Free Languages)
Fix some language L.

$$\forall p > 0, \; \exists s \in L, \; |s| \geq p$$
$$\forall u, v, w, x, y \in \Sigma^*$$
$$s = uvwxy, \; |vx| > 0, |vwx| \leq p$$
$$\exists i \geq 0, \; uv^iwx^iy \notin L$$
$$\Rightarrow L \text{ is not context free}$$

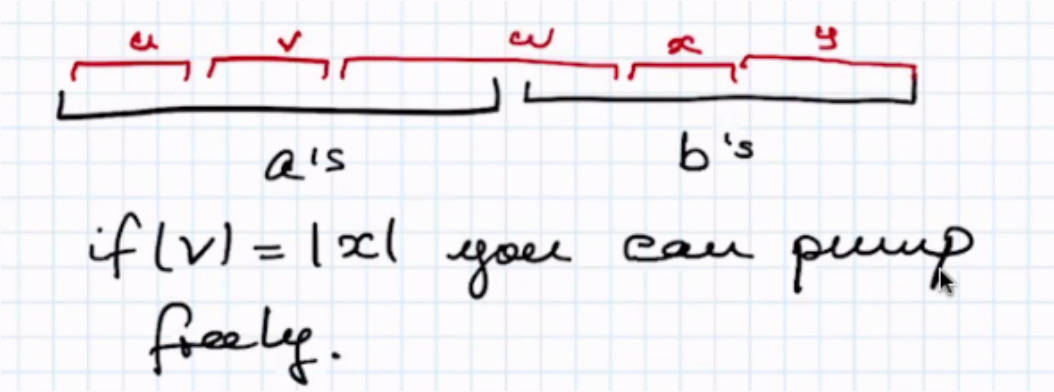
Example 10.3

$L = \{a^n b^n a^n \mid n \geq 0\}$.

1. Demon chooses p
2. We choose $a^p b^p a^p$.
3. Possibilities for $uvwxy$.
   a) If v or x stradles a block boundary, the demon loses with $i = 2$ because the a's and b's will be out of order.
   b) If v and x both contain only a's they must be in the same block. The demon loses with $i = 2$ because blocks will no longer be of the same size.
   c) If v and x contain only b's, the demon loses with $i = 2$ because the b block is too long.
   d) If v contains only a's and x contains only b's, then demon loses with $i = 2$ because the untouched block will be of different length.
4. Therefore the language is not context free because we have a winning strategy.

Example 10.4

$L = \{a^n b^n \mid n \geq 0\}$. In the regular case, the y consists of only a's so we win. But now in the context free case, there isn't a winning strategy.



Example 10.5

$L = \{a^{i+j} b^{j+k} c^{i+k} \mid i, j, k \geq 0\}$. Try to recognize instictively whether or not the language is context free before proving it.

This is context free! Trying to reason about why. $i + j$ is unbounded, but the stack is also unbounded. A lot more tricky to check CFGness.

Exercise: Cook up a CFG for this.

Example 10.6

$|\Sigma| \geq 2$. $L = \{ww \mid w \in \Sigma^*\}$, $L' = \{ww^{REV} \mid w \in \Sigma^*\}$. The first is not context free while the second is. The first is not possible because it's difficult to check if the strings are the same. The second is context free because the stack is good for checking the reversed string.

Fact 10.7. If L is CF and R is regular, then $L \cap R$ is a CFL.

Example 10.8

$L = \{ww \mid w \in \Sigma^*\}$. We use $R = a^*b^*a^*b^*$ and show $L \cap a^*b^*a^*b^*$ is not a CFL so L cannot be a CFL.

1. Demon chooses p.
2. We choose $a^p b^p a^p b^p$.
3. Possibilities for $uvwxy$.
    a) If v, x straddle block boundaries, the demon will lose
    b) If v, x are in the same block the demon loses because words aren't the same length.
    c) v and x have to be in consecutive blocks because of the bounded size of vwx. Note that the first copy of w starts with a so the second word cannot start with b. Also that the second copy ends with b so the first copy must end with b. So the boundary between 2 copies of w has to be at the end of the 2nd block. So although the two words could be the same size, they won't be the same words.
4. Therefore there is a strategy to always beat the devil so the language is not context free.

Example 10.9

$L = \{0^i 1^j \mid j = i^2\}$. This is not a CFL.

Let A be the adversary and I be us.

1. $A \to p$
2. $I \to 0^p 1^{p^2}$
3. $A \to uvwxy = 0^p 1^{p^2}$, $|vwx| \leq p$, $|vw| > 0$.
4. $I \to$ case analysis
   a) vwx all zeros pick any $i \neq 1$.
   b) vwx all ones, pick any $i \neq 1$.
   c) v, x straddles the boundary, $i = 2$ gives 0's and 1's out of order.
   d) v is all zeros and x is all ones. We pick $i = 2$. Only non trivial case is when $|v| = m > 0$, $|x| = q > 0$. Pumping to $i = 2$ gives $0^{p+m}1^{p^2+q}$. How do we know that $(p+m)^2 \neq p^2 + q$?

$$(p+m)^2 = p^2 + 2pm + m^2$$
$$2pm > p \Rightarrow 2pm + m^2 > p > q \Rightarrow p^2 + 2pm + m^2 > p^2 + q$$

Example 10.10

$L = \{a^q \mid q \text{ a prime}\}$. Even with a stack you cannot check this.

1. $A \to p$
2. $I \to a^q$ $|q > p$, prime
3. $A \to uvwxy$ s.t. $|uvwxy| = q$, $|vwx| \leq p$, $|vw| > 0$.
4. Let $vx = r > 0$. Let $i = q + 1$. Then $|uv^i wx^i y| = q + qr = q(1 + r)$. Can't be a prime because it's the product of two number where each is greater than 1.

Theorem 10.11

Over a one-letter alphabet a language is context-free if and only if it is regular.

Example 10.12

$L = \{a^{2^n} \mid n \geq 0\}$ is not a CFL. We showed that this language was not regular.

Exercise 10.13. $L = \{a^k b^j c^k \mid 0 < i < j < k\}$. Show that this is not a CFL.

Definition 10.14 (DPDA)**.** PDA's feature non determinism. This cannot be eliminated so DPDA's are strictly less powerful than PDA's.

If a CFL can be recognized by a DPDA we call it a Deterministic CFL. All modern languages are DCFL's. This makes things fast.

DPDA is a DFA with a stack.

Fact 10.15. DCFL's are closed under complement and therefore under intersection.

Example 10.16

$L = \{ww \mid w \in \Sigma^*\}$, $|\Sigma| \geq 2$. This is not a CFL, but $\overline{L}$ is a CFL.

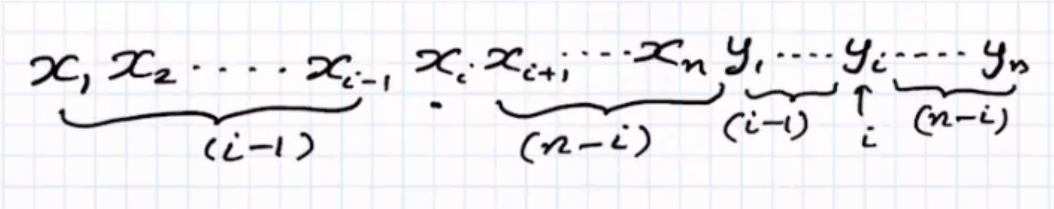
$\overline{L}$ an be recognized by a PDA. What is in $\overline{L}$.

1. Strings of odd length.
2. Might be even length. Then it can be written as concatenation of two equal length words where the words are not equal to one another. $|w_1| = |w_2|$ but $w_1 \neq w_2$. Two words are not the same if there exists i such that $w_1[i] \neq w_2[i]$.

Crucial difference is that instead of checking every position, you only have to guess one position. If there is an existing path, it works.

The idea is guess where i is and guess where the middle of the word is. It is okay that this i is unbounded because of the non determinism.

How the PDA works:



Push $x_1 \dots x_{i-1}$ onto the stack. Then remember what x_i is in the state. Start popping the stack until it is empty. Then start pushing letters onto the stack. It guesses where y_i is. It looks at y_i and compares it with x_i. It then pops the stack as it reads the remeaining letters. If the stack is empty at the end then x_i and y_i are indeed at the right positions.

$(i-1)$ symbols on the stack. $(n-i)+(i-1) = n-1$ between x_i and y_i.

After checking y_i there are $(n-i)$ letters on the stack. This exactly matches the $(n-i)$ letters after y_i.

Key point is didn't have to look at all the symbols, but used them to count.

Example 10.17

Equivalent CFG for $\overline{L}$. Let $V = \{S, A, B, C\},\ \Sigma = \{a, b\}$.

$$S \to$$

A produces odd length words with a in the middle.

B produces odd length words with b in the middle.

$S \to AB$ ultimately generates $xay\ ubv$. $n = |x| = |y|$ and $m = |u| = |v|$.

There is guaranteed to be one place where the corresponding positions are not the same.

Coming up.

1. Thursday. Concept of computability. Models of computation. Unsolvable problems. CE functions. Dovetailing.
2. Monday. Reduction.
3. Tuesday. FOC.
4. Wednesday. Lambda-calculus.
5. Thursday. Godel's Theorem. Recursion Theorem.

§11 05-21

§11.1 Existence of Unsolvable Problems

Modern language makes this proof much easier than it was for Turing.

Theorem 11.1

There is no algorithm that can analyze algorithms and their inputs and determine whether the algorithm halts or not on the given input.

Proof. Write S for an algorithm. $S(x)$ for running algorithm on input x. $\#S$ for the code of S. Everything used to have to be coded up as an integer. $S(w)\downarrow$ indicates that S halts when run on w. $S(w)\uparrow$ for when S runs forever on the given input.

Assume we have a program $H(\#S, x)$ that returns true if $S(x)\downarrow$ and false if $S(x)\uparrow$. H itself always halts.

Define $P(x) :=$ if $H(x, x)$ then loop forever else halt.

Now we give P it's own code. The question is $P(\#P)\downarrow$ or $P(\#P)\uparrow$. Two cases

1. If $P(\#P)\downarrow$, then H returns true but this means that P loops. Contradiction
2. If $P(\#P)\uparrow$, then H returns false but this means that P halts. Contradiction

Therefore a program like P cannot exist. □

§11.2 Turing Machine

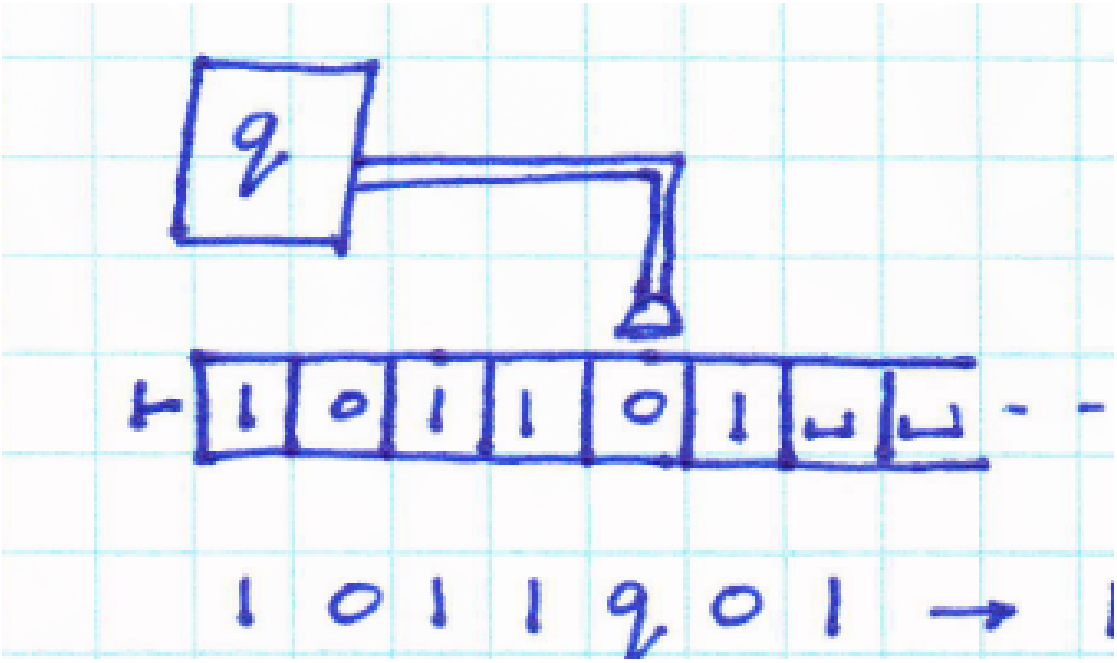
Notes

$\delta(q, a) = (q', b, L)$ means if the machine reads a and is in state q, it changes state to q', writes b in the place of a and moves one step to the left.

At each stage only a finite amount of information is processed.

Easy to see that $\{a^n b^n c^n \mid n \geq 0\}$ can be recognized by a turning machine.

A configuration of a TM is a description of its state, the tape, and the position of the reading head. Convention that you write all of the tape to the left of the state, then the state, and then the rest of the tape including the cell the head is looking at.



With this you can move backwards in the "stack" without deleting information.

uaq_ibv yields uq_jacv if $\delta(q_i, b) = (q_j, c, L)$. Emphasis on how only a three letter window is affected.

Acceptance of w by M. Start in a start configuration q_0w. Follow the transitions and end up in the accept state. Once you get to the accept state you can't leave.

A new phonomenon: The machine may go into an infinite loop. This doesn't happen in DFA or NFA because you read left to right and you are done.

The language of a machine. $L(M) = \{w \mid M \text{ halts and accepts } w\}$.

If $w \notin L(M)$ it does not mean that M explicitly rejects w. It could be looping forever. Basically there is a new "I don't know" option which makes a huge difference. Not everything can be answered.

Definition 11.2 (Turing Recognizable)**.** L is Turing Recognizable if $\exists TM\ M$ such that $L = L(M)$.

We say computably enumerable or CE for recognizable.

Definition 11.3 (Turing Decidable)**.** L is Turing Decidable if $\exists TM\ M$ such that $\forall w,\ M(w) \downarrow$ and $L = L(M)$. i.e. for every word you get a Yes/No answer.

We say computable for decidable.

Fact 11.4. Any decidable language is, of course recognizable.

Note 11.5. Old terminology. Recursively enumerable (RE) for CE. Recursive for decidable.

§11.3 Models of Computation

One can think of a turing machine as a computing function $f : \mathbb{N} \to \mathbb{N}$ instead of accepting languages. They are equivalent, but sometimes it is more helpful to think in one way than the other.

People proposed many variations all of which turned out to be equivalent to turing machine. Examples include Multitape Turing Machine, Nondeterministic Turing Machine, n-dim TM, Post Machine, 2 stack NFA, 2 counter, RAM using assembler, while programs, any modern programming language, λ-calculus, conbinatory logic, phrase structure grammas, post systems. This is not to say that some are not differences in efficiency/ease.

Are all possible formalisms equivalent? We don't know for sure but it is believed so. Church-turing thesis says so.

So we believe there isa notion of computable functoin independent of any specific model of computation.

Proved that certaint hings were impossible but in a completely different sense. When referring to distributed computing, the notion of what is computible

Turing machine writes down all the prime numbers. Never going to write them all because there are infinitely many.

If you define computing to be looking at the answer one it halts, if you say you have one TM who's tape is being fed into a second TM, the first guy doesn't even have to halt before being useful.

§11.4 Theory of Computability

We say 2 infinite sets are equipollent if $\exists$ a bijection between them. There is no bijection between $\mathbb{N}$ and $2^{\mathbb{N}}$ but there is a bijection between $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Definition 11.6 (Equipollent)**.** Two sets are said to be equipollent if there is a bijective function mapping one onto the other.

"Equipollent" means "of equal power", where "power" here alludes to the size of the sets. Two sets are equipollent precisely if they have the same cardinality.

We now have algorithms that may loop forever and produce no output. We model this by partial functions, $f : \mathbb{N} \to \mathbb{N}$. Partial means that f is not defined for every $n \in \mathbb{N}$. Let $f(n) \downarrow$ mean f is defined on n and $f(n) \uparrow$ mean f is not defined on n.

$$dom(f) = \{n \mid f(n) \downarrow\}$$

$f = g$ as partial functions means $dom(f) = dom(g)$ and $\forall n \in dom(f), f(n) = g(n)$. New notion that we don't typically have is $f \leq g$ means that $dom(f) \subseteq dom(g)$, $\forall n \in dom(f)$, $f(n) = g(n)$. This gives a partial order structure.

There is a minimal element, i.e. the everywher undefined function. It's domain is the empty set.

$$range(f) = \{m \mid \exists n \in dom(f) \text{ s.t. } f(n) = m\}$$

When discussing computability, $\mathbb{N}$ is not important; no real difference between $\mathbb{N}$, $\mathbb{N}^k$, and Σ^*. $\mathbb{R}$ is not comparable because uncountable; there is a fascinating field concerning $\mathbb{R}$ but we won't discuss it here.

Note 11.7. It is not algorithmic to write things like:

1. wait for M to halt, if it doesn't halt then...
2. Check on every word if...

Definition 11.8 (Computable Function)**.** A function $f : \mathbb{N} \to \mathbb{N}$ is computable if there is an algorithm A such that for all $n \in dom(f)$, A halts and the output if $f(n)$.

Definition 11.9 (Computable/Decidable Set)**.** A set of natural numbers $X \subseteq \mathbb{N}$ is computable or decidable if its characteristic function

$$\chi_x(n) = \begin{cases} 1, & n \in X \\ 0, & n \notin X \end{cases}$$

it total computable.

Definition 11.10 (Total function)**.** A total function is defined for all possible input values.

Proposition 11.11

An infinite set of natural numbers is decidable iff it is the range of a total non decreasing computable function.

Proof. Suppose we have X, we have f as above $X = range(f)$ and we have an always terminating algorithm for f called A. Let x, want to know if $x \in X$.

Run A on $0, 1, 2, 3, \ldots$. Eventually you may get $A(n) = x$ then say yes $x \in X$. Or you may get $A(n) > x$ then stop and say no, $x \notin X$.

For the reverse direction, assume we have a decision procedure for X, call it B. Run B on $0, 1, 2, \ldots$. As soon as it says "Yes" for the the first time you say $f(0) = x_0$. Second time you say $f(1) = x_1$. This gives a total non-decreasing computable mapping whose range is X. □

Definition 11.12 (Enumerable)**.** A set $X \subseteq \mathbb{N}$ is called enumerable or CE if $\exists$ algorithm A that lists all the members of X and only the members of X in some order, not necessarily increasing order. A may not halt but it produces every element of X eventually. No guarantee of when and in what order.

Theorem 11.13
A set X is CE iff

1. It is the domain of some computable function.
2. The range of a computable function
3. S_X is computable where

$$S_X = \begin{cases} 1 & n \in X \\ ? & n \notin X \end{cases}$$

Proof. Rigorous proof in course notes. □

Note 11.14. Computable set is the same as decidable set. Enumerable is the same as computably enumerable and is weaker then computable/decidable.

Review notes online

Theorem 11.15
The union and intersection of 2 CE sets is CE. This is how computers can do things concurrently.

Remark 11.16. The complement of a CE set is not always CE.

Proposition 11.17 (Post's Theorem)
If a set X and its complement $\overline{X}$ are both CE then they are both decidable.

Theorem 11.18
A set $X \subseteq \mathbb{N}$ is CE iff $\exists a$ decidable $Y \subseteq \mathbb{N}$ such that $x \in X \Leftrightarrow \exists y \in \mathbb{N},\ \langle x, y \rangle \in Y$

Proof. If Y is decidable (or even CE) we enumerate its memebers and compute □

§12 05-25

§12.1 Reductions

P reduces to Q means if I can solve Q I can solve P. This is roughly equivalent to Q is "harder than" P.

Algorithm for solving P. First transform the problem into a Q problem. Then feed the problem to the Q solver.

If you know P is undecidable than the putative Q solver cannot exist, so Q is also undecidable.

$\leq$ is a preorder. Transitive so reductions can be chained. Not partial order because it doesn't have anti-symmetry.

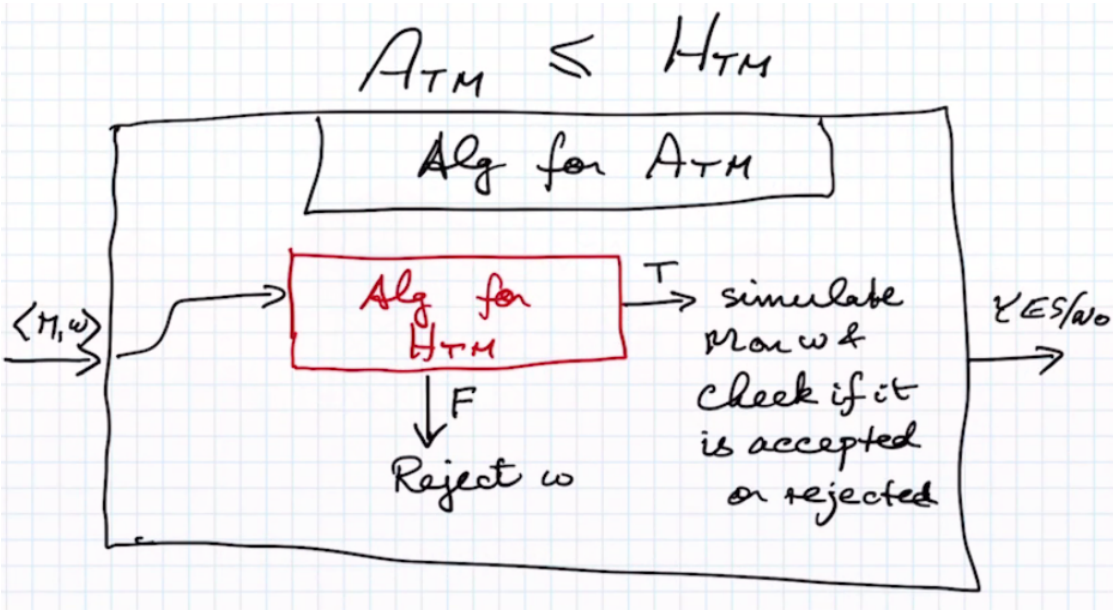
Notation. $\langle M \rangle$ is encoding of a machine. $\langle M, w \rangle$ is a machine and its input. $\langle M_1, M_2 \rangle$ is two machines. $\langle G \rangle$ encodes a CFG.

$$H_{TM} = \{\langle M, w \rangle \mid M \text{ halts on } w\}$$
$$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts } w\}$$

H_{TM} accepts the set of machines and input that halt. A_{TM} accepts the set of machines and inputs that those machines accept. $H_{TM} \leq A_{TM}$.

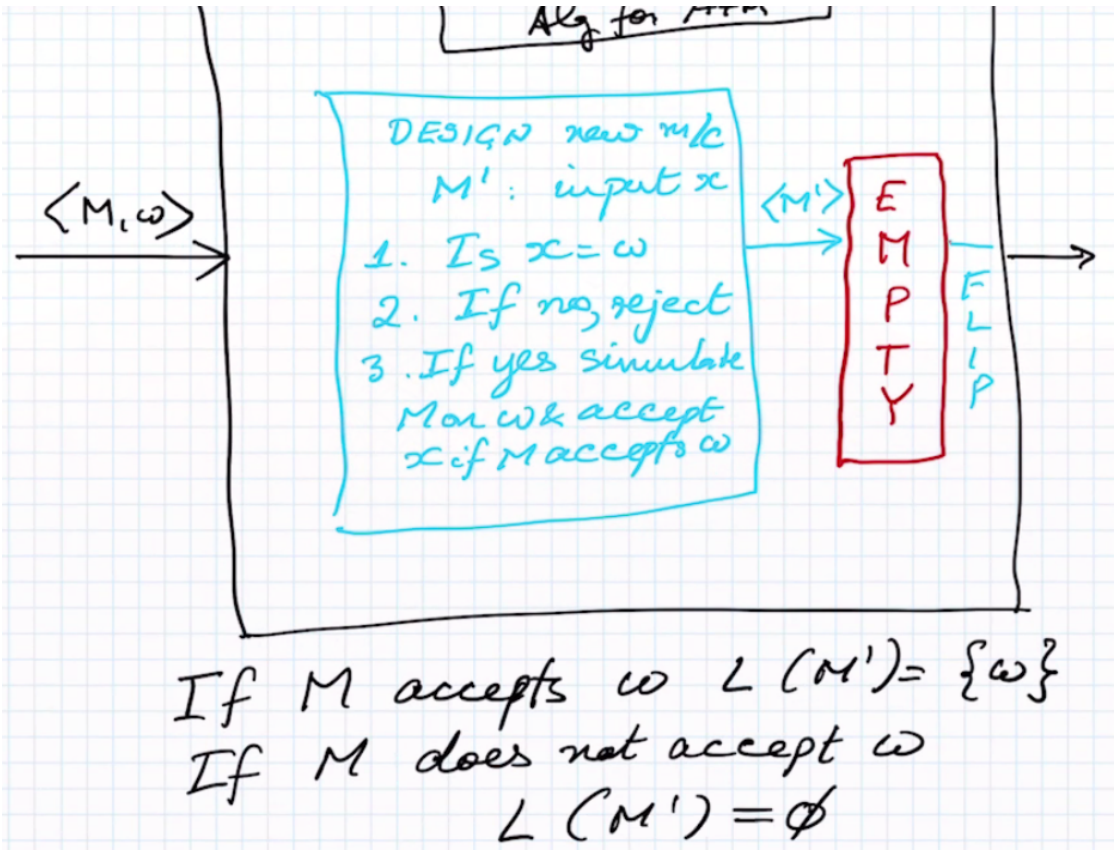
Note 12.1. Constructing a machine should be thought of as writing the code for the machine.

Algorithm for A_{TM} cannot exist because H_{TM} reduces to it.



$EMPTY_{TM} = \{\langle M \rangle \mid L(M) = \varnothing\}$.

$A_{TM} \leq EMPTY_{TM}$. Machine:



REG, is $L(M)$ a regular language? Construct machine.

Note 12.2. The more powerful a gadget is, the easier it is to build a reduction to that gadget.

$EQ_{TM} = \{\langle M_1, M_2\rangle \mid L(M_1) = L(M_2)\}$. $EMPTY_{TM} \leq EQ_{TM}$. Construct a machine that rejects all words, and compare equality of input to $EMPTY$ with this machine using EQ machine. If they are equal, then M is empty.

Exercise 12.3. Show that the following are undecideable.

1. $L(M) = L(M')$ where M' always halts.
2. $L(M)$ is context free.
3. $|L(M)| < \infty$.
4. $L(M) = \Sigma^*$.

§12.2 Sharper Notion of Reduction

Mapping reduction. Suppose $L_1, L_2 \subseteq \Sigma^*$. We say L_1 is mapping reducible to L_2

$$L_1 \leq_m L_2$$

if there exists a total computable function $f : \Sigma^* \to \Sigma^*$ such that $\forall w \in \Sigma^*,\ w \in L_1 \Leftrightarrow f(w) \in L_2$.

Note 12.4.

1. $L_1 \leq_m L_2$ then $\overline{L_1} \leq_m \overline{L_2}$. This is not true for general reductions.
2. $\leq_m$ has a direction because f has a direction. No guarantee that you can go both ways. $L_1 \leq_m L_2$ does not mean $L_2 \leq_m L_1$.

Fact 12.5. Facts about $\leq_m$.

1. $P \leq_m Q$ and P is undecidable then Q is undecidable
2. $P \leq_m Q$ and Q is decidable then P is decidable.
3. $P \leq_m Q$ and Q is computably enumerable then P is also computably enumerable.
4. $P \leq_m Q$ and P is not computably enumerable, then Q cannot be computably enumerable.
5. If $P \leq_m Q$ and P is not co-CE then Q cannot be co-CE. co-CE means if the algorithm is NO your algorithm will definitely tell you. CE means if the answer is YES your algorithm will definitely tell you.

   H_{TM}, A_{TM} are CE but not coCE. Run it and it will tell you if the answer is yes.

   $\overline{A_{TM}}, EMPTY_{TM}$ are co-CE. You will definitely find out if it is not empty with dove tailing.

Semi decision problem, computably enumerable set.

$A_{TM} \leq EMPTY_{TM}$ but this is not a mapping reduction. Suppose we had $A_{TM} \leq_m EMPTY_{TM}$, then $\overline{A_{TM}} \leq_m \overline{EMPTY_{TM}}$. But this is not possible because $\overline{A_{TM}}$ is co-CE while $\overline{EMPTY_{TM}}$ is CE.

§12.3 Turing Reduction

$P \leq_T Q$. I get to use a Q oracle as many times as I want and I can do any computable post processing I want.

$P \leq_m Q$. I get to do some total computable preprocessing and then ask my Q oracle 1 question and output the answer without post processing. Can't even flip the output.

> **Theorem 12.6**
> EQ_{TM} is not CE or coCE. More difficult than halting problem.

Fact 12.7. Halting problem is complete for all CE problems. CE complete.

Non-halting problem is coCE complete.

$|L(M)| = \infty$. INF $= \{\langle M\rangle \mid |L(M)| = \infty\}$

Claim: $\overline{H_{TM}} \leq_m INF$.

Reducing of non halting problem. Let input to M' be X, then run M on w $|x|$ times and reject x if it halts, accept otherwise. The language of M' is everything if w runs forever and is finite otherwise, which can be checked with INF.

Theorem 12.8 (Rice's Theorem)
$P : \mathbb{N} \to \mathbb{N}$. $[\![P]\!] := \{(x, y) \mid P(x) = y\}$.

$P_1 \sim P_2$ means $[\![P_1]\!] = [\![P_2]\!]$.

P_1, P_2 are extensionally equal.

$M_1 \sim M_2 \Leftrightarrow L(M_1) = L(M_2)$. $Q : PROG \to \{T, F\}$ is called a property of programs. Q is an extensional property if $P_1 \sim P_2 \Leftrightarrow Q(P_1) = Q(P_2)$. Q only depends on the IO behavior. Q only depends on the functional spec.

Q always true or Q always false are trivial properties.

Rice's theorem: Every non trivial extensional property of programs is undecidable. Nothing that just depends on the IO spec can possibly be decidable.

Proof. Let Q be a nontrivial property of CE sets. i.e. $\exists P$ such that $Q(P) = true$ and $\exists P'$ such that $Q(P') = false$.

Assume empty does not satisfy Q. i.e. $\forall M$ if $L(M) = \varnothing$ then $Q(M) = F$.

Let M_0 be such that $Q(M_0) = T$. Then $L(M_0) \neq \varnothing$ by our assumption.

$$L_q = \{\langle M\rangle \mid Q(M) = T\}$$

Claim: $A_{TM} \leq_m L_Q$. Have gadget to solve $x \in L_Q$.

M' with input x construction: Simulate M on w. If M accepts w then simulate M_0 on x. □

§13 05-27

§13.1 Universal Functions

Definition 13.1. A binary function $U : \mathbb{N}^2 \to \mathbb{N}$ is said to be universal for the class of computable unary functions if

1. $\forall n$, $U_n : x \mapsto U(n, x)$ is computable. U_n is called a section of U. This is called currying, when you split a function that takes multiple parameters into nested unary functions.

2. $\forall$ unary computable $f : \mathbb{N} \to \mathbb{N}$, $\exists n$ such that $U_n = f$, i.e. $\forall x\ U_n(x) = f(x)$

Note that in this definition U doesn't have to be computable.

Note 13.2. The list of programs is countable.

Theorem 13.3
There is a binary computable function $U : \mathbb{N} \times \mathbb{N} \to N$ such that U is a universal function for all unary computable functions. i.e. one turing machine that can simulate all the others.

Proof. Consider your favorite programming language (YFPL), and enumerate all legal programs, $p_1, p_2, \ldots, p_n, \ldots$

$U(n, x) = p_n(x)$. So U is an "interpreter" while n is the code of the algorithm. □

Note 13.4. "Code" comes from the days of early computability theory because every program could be coded up as a number.

§13.2 Total Computable Universal Function

Does there exist a total computable universal function for the class of total computable unary functions. Total means it will be defined on all inputs, so everything must terminate. No!

Let U be any total computable function of two arguments. Define $d(n) = U(n, n) + 1$. $\forall n,\ d(n) \neq U_n(n)$ so $\forall d \neq U_n$. Can't guarantee all terminating algorithms, or must allow some options to not terminate.

Think about why doesn't this argument work for partial functions? Because $U(n, n)$ might be undefined, in which case $U(n, n) + 1$ is still undefined.

§13.3 Compositional Programming

We want to program Compositionally. If f, g rae computable functions, then $g \circ f$ is also computable. The map that figures out $g \circ f$ should be total computable.

Definition 13.5. Let S be any countable set. A map $\nu : \mathbb{N} \to S$ is called a numbering of S if ν is surjective.

A value of n such that $\nu(n) = s$ is called a code number for s.

Note the indefine article. There can be multiple such n for a given element.

We want to show that for the "right kind" of universal function U, there is a computable

function with the following properties.

$$c : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$\forall p, q, x \in \mathbb{N},\ (U_p \circ U_q)(x) = U(p, U(q, x)) = U(c(p, q), x) = U_{c(p,q)}(x)$$

Note 13.6. A set S is computable if there is a total computable function f, such that $f(n) = 1$ if $n \in S$ and $f(n) = 0$ if $n \notin S$.

Note that computable function doesn't have to be "computable" set. Only total computable function.

Definition 13.7. Let U be a universal computable function. It is called a Godel universal function if $\forall$ binary computable functions V, $\exists$ a total computable unary function $\sigma : \mathbb{N} \to \mathbb{N}$ such that $\forall m, x \in \mathbb{N},\ V(m, x) = U(\sigma(m), x)$ (σ will depend on V).

I stopped taking notes because I realized there was a handout online with detailed notes on today's lecture.

§13.4 Primitive Recursive Functions

Godel. These roughly correspond to a programming language with bounded search. i.e can't use while loops, only loops that run a set number of times.

Fortran for loops for example. Provably terminating.

Ackerman gave an example of a provably terminating function that was not primitive recursive.

This makes sense based on the proof above, whereby it's impossible to produce a universal total computable function for the class of total computable unary functions.

Kleene: PRF + unbounded search gives partial recursive functions. They include the power of while loops. Proved that these are equivalent to turing machines and lambda calculus.

§13.5 Degrees of Unsolvability

Online handout.