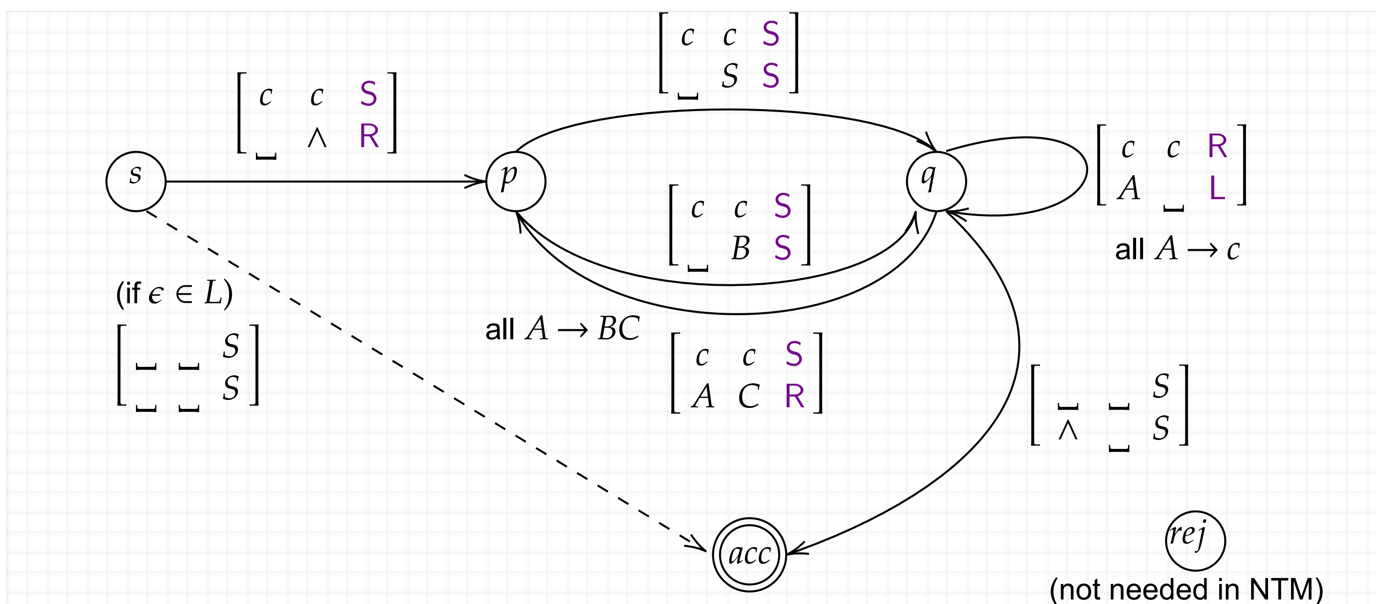
CSE396 Spr26 Lecture Tue. Apr. 7: Decision Problems and Decidability

First, we pick up a few topics from Chapter 2 that were passed over.

Theorem: A language L is a CFL if and only if there is an NPDA N such that $L(N) = L$.

Proof Sketch: Take a CFG $G = (V, \Sigma, \mathcal{R}, S)$ in Chomsky NF such that $L(G) = L \setminus \{\epsilon\}$. (As with the CFL Pumping Lemma, ChNF is not necessary and the text does without it, but IMHO it improves the visual understanding.) The NPDA N has just one "hub state" q (together with a helper state p for pushing) besides s and q_{acc} (and q_{rej} just to comply with the text's TM syntax).



The first two steps put down a bottom-of-stack marker $\wedge$ and then initialize the stack to the start symbol S of G. This sets up the

> **Computation Invariant**: The prefixes u of the input tape processed thus far, followed by the mirror reflection of the stack tape (ignoring the $\wedge$) correspond to steps of a leftmost derivation in G. If the final input blank is read when the stack is empty, then $u \in L(N)$ and $G \Longrightarrow^{*} u$.

The terminal rules $A \rightarrow c$ preserve this invariant by having N process c and pop A. The rules $A \rightarrow BC$ are handled by first overwriting A by C going from state q to state p, and then pushing B going back to state q, so that B (which is the new leftmost variable in the derivation) becomes the top of stack. Maintenance of the invariant implies $L(N) = L(G)$. [Note that N obeys the "accept by final state" and "accept by empty stack" condiitons simultaneously---this is the essence of the text's proof of their equivalence; with our TM-based definition we can just ignore that.] Example.

The proof in the other direction---from machine to equivalent grammar---is more complicated and is: *FYI, skim/skip* in the text. ☒

The above also contains the essence of proving the *equivalence* of two criteria for PDAs that one can find discussed in other sources: *acceptance by empty stack* and *acceptance by final state*. We can always make a PDA---nondeterministic or deterministic---do both simultaneously.

Theorem: Every deterministic PDA M has an equivalent DPDA M' that halts for all inputs. Hence the class DCFL of languages accepted by DPDAs is closed under complements.

Proof: If $M(x)\uparrow$, then the computation must hit a point in a state q where a character $c \in \Sigma$ is being scanned on the input tape that never gets processed and a char $d \in \Gamma$ atop the stack tape that never gets popped. The fact of (q, c, d) causing an infinite loop does not depend on anything else. There are only finitely many possible combinations, so we can "manually" determine the bad ones and edit the code of M to make those combos go right to q_{rej} in the new DPDA M'. ☒

Corollary: If L is a CFL but its complement $\widetilde{L}$ is not a CFL, then L is not a DCFL. ☒

The intersection of two DCFLs need not even be a CFL, however---recall $A = \left\{a^n b^n\right\} \cdot c^*$ and $B = a^* \cdot \left\{b^n c^n\right\}$. Note also that the complement of $A \cap B$ satisfies, by De Morgan's Laws:

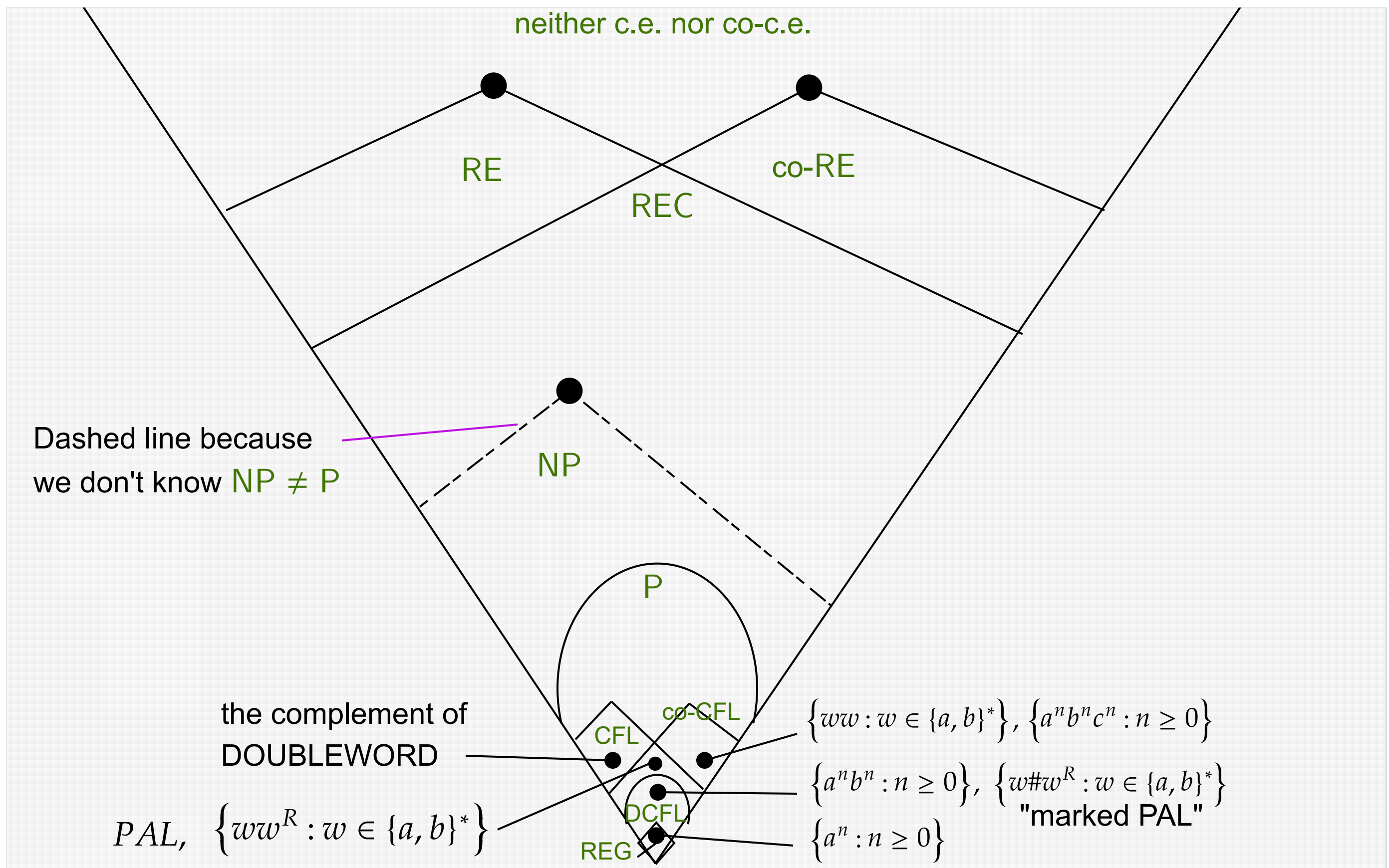
$$\sim (A \cap B) = \widetilde{A} \cup \widetilde{B},$$

which is a union to two DCFLs and hence is a CFL, but is not a DCFL. There are also languages C such that both C and its complement $\sim C$ are CFLs, but C is not a DCFL; the best example is PAL, the original language of palindromes *without* using a `#' to mark the middle point. (Proving it not a DCFL is beyond our scope.)

All such languages, indeed every CFL or its complement, can, however, be accepted by a **DTM** that is total. The DTM on any input x simulates the grammar, trying out all possible derivations of x. We will observe at the very end that though this could involve exponential branching of which-rule-to-use-when, putting the grammar into Chomsky NF yields a **dynamic programming** algorithm that runs in polynomial time. So the class **CFL**, and also **co-CFL**, is contained within the class **P**.

We would love to be able to identify tape configurations that cause DTMs not to halt, and similarly fix them. But we will see on Thursday that this is not in general possible. Indeed, we will build non-total DTMs M such that no total TM M' gives $L(M') = L(M)$. This will follow from showing that the class **RE** of all Turing-acceptable languages (*synonyms*: computably enumerable / c.e. / recursively enumerable / r.e. languages) is not closed under complements: **co-RE** $\neq$ **RE**.

As both summary and preview, here is a "cone diagram" of all the classes of languages that we have defined thus far in the course, along with some representative languages:



"Co"-classes are shown as mirror images---so classes that are closed under complements are mirror-symmetrical. The rest of this course will add more languages and a few more classes.

The Church-Turing Thesis and the Quantum Challenge

The equivalence of high-level programming languages (HLLs) and Turing machines is the greatest evidence for the following claims of universal law---in several senses of "universal":

Church-Turing Thesis (three-part version):

1. Any HLL ever devised will have the same computing power as the Turing machine. I.e., given a notion of "accepting" an input---such as outputting 1 or exit code 0 or executing `System.exit(0)`---the resulting classes RE and REC are the same as for DTMs.
2. Any physical device ever built will have no more computing power than a Turing machine.
3. For any human being H who follows a consistent functional procedure to convert (sensory) inputs x into outputs y, there exists a Turing machine M_H that on the same inputs x (under a natural string encoding, e.g., pixels for optical input) outputs the same values y. Moreover, M_H has comparable program size and efficiency to the "grey matter" of H, or better.

Plank 1 is often considered a "truism" but maybe it depends on plank 2. Plank 2 maybe harks back to plank 1 because ***computability*** is an abstract notion for languages and functions over infinitely many

strings---almost all of which have more chars than the number of quarks in te observed universe, which is under 2^{270}. Plank 3 comes from Alan Turing's seminal 1936 paper titled, "On Computable Numbers, With an Application To the *Entscheidungsproblem"*---but gets an interpretive twist from AI today. The **Turing Test** used to be considered to be *about* chatbots: can a chatbot be good enough that no one---over a connection showing just the chat---can distinguish it from a human respondent? Well, this can be more general: e.g., a chess cheater using a computer is hoping to avoid being distinguished from a human player. My formulation is more specific even in the chatbot context, though: is there a chatbot that is indistinguishable **from you**? Well, if you are really a machine to begin with, then this is tautologically true... Less tautological and more controversial, the program and memory size S needed to simulate human cognition is the threshold that "The Singularity" talks about.

The "Part Deux" of the C-T thesis is often ascribed to Alan Cobham and Jack Edmonds from papers they wrote in 1965, in which they justified **polynomial time** as a benchmark for feasible problem-solving:

Polynomial-Time C-T Thesis: As above, plus the assertion that whatever the HLL and/or device physically implementing its programs, there will always be a constant k such that whatever the program/device does in time t can be emulated by $O\left(t^k\right)$ steps of the Turing machine.

This was almost-universally believed until 1994, when Peter Shor proved:

Theorem X: Quantum computers can factor n-digit numbers in $\widetilde{O}\left(n^2\right)$ time.

This is idealized---no one has yet built quantum technology that can *scale up.* For comparison, the security of most Internet commerce and many other cryptosystems relies on concrete scaling of the belief that factoring requires roughly $2^{\Omega\left(n^{1/3}\right)}$ time, well maybe $2^{\Omega\left(n^{1/4}\right)}$ or $2^{\Omega\left(n^{1/5}\right)}$ time in most cases... [See this about the 1992 movie *Sneakers* and this review of the novel *Factor Man*.]

But as long as we stick with "classical" machines---meaning non-quantum hardware---we can take both theses as given. (Note: Actually, transistors and other chip elements *are* quantum devices, but the point is that they treat information in the classical manner of *bits*, as opposed to *qubits*.) The import is:

The classes* REC*,* RE*, and co-*RE*, and later* P*,* NP*, and co-*NP*, remain the same whenever we transfer their defining notions to any HLL or classical machine model. Moreover, it is legitimate to describe Turing machines via pseudocode---provided that gives enough detail to pin down the running time* t *within a linear* $O(t)$ *or at worst a polynomial* $t^{O(1)}$*, factor.

For example, the 2-tape TM we built to recognize $\left\{a^m b^n : m = n\right\}$ can be described by saying, "Copy leading a's to tape 2, then count against b's on the rest of tape 1, and accept iff the counts are equal and the end is reached on tape 1 without any further a appearing. Runtime: $O(m+n)$ steps, which is linear in the length $m+n$ of the input."

Decision Problems (Chapter 4, section 4.1)

The Sipser text adopts the format for specifying decision problems that came from an older text by Michael Garey and David S. Johnson:

[Name of problem in small caps]
INSTANCE: [a description of the input(s) to the problem: strings, numbers, machines, graphs, etc.]
QUESTION: [a *yes*/*no* condition where *yes* means the input is accepted]

INSTANCE is also called INPUT; one can abbreviate it to INST and QUESTION to QUES. The **language** of the problem is the set of valid instances for which the answer is *yes*. Sometimes confusingly, the name of the problem usually doubles as the name of the language. The Sipser text also *established* a standard scheme for naming various decision problems that arise with the various machine, regexp, and grammar classes in this subject. It is best described by example.

A_{DFA}: (The "**A**cceptance Problem *for* **DFA**s")
INST: A DFA $M = (Q, \Sigma, \delta, s, F)$ and a string $x \in \Sigma^*$.
QUES: Does M accept x?

The input to a decision procedure for this problem is given in the form $\langle M, x \rangle$. The language is

$$A_{DFA} = \{\langle M, x \rangle : M \text{ is a DFA and } M \text{ accepts } x\}.$$

The length N of $\langle M, x \rangle$ can be reckoned as roughly of order $m + n$ where m is the number of states in Q (note that the number of instructions for a DFA is m times $|\Sigma|$ and we can treat $|\Sigma|$ as a fixed constant such as 2) and $n = |x|$ as usual. The alphabet of the A_{DFA} language can be reckoned as ASCII or even as $\{0, 1\}$. Here is a simple statement of an algorithm to solve the A_{DFA} problem:

1. Given $\langle M, x \rangle$, first decode M and x individually. (If not possible, reject.)
2. Run $M(x)$ (using a simulator like the *Turing Kit*) until the DFA reaches the end of x.
3. Accept $\langle M, x \rangle$ if M accepted x, else halt and reject $\langle M, x \rangle$.

This pseudocode always halts because a DFA M always halts. To simulate a step of $M(x)$ takes time *at most* order-m; really it can be $O(\log m)$ time per step using good data structures (mainly being able to assign a pointer to the destination state in any executed instruction). So the running time is $O(mn)$ which gives time $O(N^2)$ taking the length $N = |\langle M, x \rangle|$ into account. Thus we can say:

- The algorithm is a **decision procedure** to solve the A_{DFA} problem.
- Hence the A_{DFA} *problem* and the A_{DFA} *language* are called **decidable**.
- In fact, they are *decidable in polynomial time*.

Now suppose we have an NFA in place of the DFA.

A_{NFA}: (The "**A**cceptance Problem *for* **NFA**s")
INST: An NFA $N = (Q, \Sigma, \delta, s, F)$ and a string $x \in \Sigma^*$.
QUES: Does N accept x?

The following qualifies as a decision procedure, albeit highly inefficient:

1. Given $\langle N, x \rangle$, first decode N and x individually.
2. Convert N into an equivalent DFA M.
3. Then run the decision procedure for A_{DFA} on $\langle M, x \rangle$ and give the same yes/no answer.

Step 3 will later be called **reducing** the (instance of the) latter problem **to** the (equivalent "mapped" instance of the) former problem. But step 2 makes this an inefficient reduction---it can require order-of 2^m time where we are now calling m the number of states in N. Then again, step 2 does always halt, so if halting is all you care about, it goes as a decision procedure. But faster is:

1. Given $\langle N, x \rangle$, first decode N and x individually.
2. Initialize R_0 to be the ϵ-closure of the start state of N.
3. For each char x_i of x, build the set R_i of states reachable from a state in R_{i-1} by processing x_i.
4. Accept $\langle N, x \rangle$ if and only if $R_n \cap F \neq \emptyset$, which is if and only if N accepts x.

For each char i, step 3 runs in time at worst $O\left(m^2\right)$ (again, one can do better with smarter data structures), so the whole time is $O\left(m^2 n\right)$, which is polynomial in $|\langle N, x \rangle| \approx m + n$.

(Non-)Emptiness Problems

This is the first of numerous problems in which the **instance type** is "Just a Machine."

NE_{DFA}:
INST: (The string code $\langle M \rangle$ of) A DFA $M = (Q, \Sigma, \delta, s, F)$.
QUES: Is $L(M) \neq \emptyset$?

The complementary problem ("E" for emptiness) is:

E_{DFA}:
INST: A DFA $M = (Q, \Sigma, \delta, s, F)$.
QUES: Is $L(M) = \emptyset$?

The solution is to use the same decision procedure, but switch the "accept" and "reject" cases:

1. On input $\langle M \rangle$, treat M as a directed graph without caring about the character labels on arcs.
2. Execute a breadth-first search in that graph from the start node s of (the graph of) M.
3. If the search terminates having visited at least one state in F, **reject** $\langle M \rangle$, else **accept**.

The corresponding problems for NFAs are just as easy: they have the same algorithms:

NE_{NFA}:
INST: An NFA $N = (Q, \Sigma, \delta, s, F)$.
QUES: Is $L(N) \neq \emptyset$?

Solution:

1. On input $\langle N \rangle$, treat N as a directed graph without caring about the character labels on arcs.
2. Execute a breadth-first search in that graph from the start node s of (the graph of) N.
3. If the search terminates having visited at least one state in F, **accept** $\langle N \rangle$, else **reject**.

This is BFS explicitly in the graph of N with node set Q. It is not the same as the BFS used to convert an NFA into a DFA, which ran implicitly on the power set 2^Q of Q. Also "the same" is:

E_{NFA}:
INST: An NFA $N = (Q, \Sigma, \delta, s, F)$.
QUES: Is $L(N) = \emptyset$?

Solution: run the decision procedure for NE_{NFA} but interchange the yes/no answers.

Now we consider a different kind of complementation:

ALL_{DFA}:
INST: A DFA $M = (Q, \Sigma, \delta, s, F)$.
QUES: Is $L(M) = \Sigma^*$?

Solution:

1. On input $\langle M \rangle$, form the complementary DFA $M' = (Q, \Sigma, \delta, s, F')$ with $F' = Q \setminus F$.
2. Feed $\langle M' \rangle$ to the decision procedure for E_{DFA}.
3. If that procedure accepts $\langle M' \rangle$, then **accept** $\langle M \rangle$, else **reject** $\langle M \rangle$.

This embodies what in Chapter 5 we will call a **mapping reduction** from ALL_{DFA} to E_{DFA}. The reduction and the whole procedure are **correct** because $L(M) = \Sigma^* \iff L(M') = \emptyset$.

This is not the same as the way we complemented NE_{DFA} to E_{DFA}, and the best way to see why it's not so simple is to consider the analogous problem for NFAs.

ALL_{NFA}:
INST: An NFA $N = (Q, \Sigma, \delta, s, F)$.
QUES: Is $L(N) = \Sigma^*$?

We can solve this by converting N into an equivalent DFA M and running the decider for ALL_{DFA} on $\langle M \rangle$. But that can take exponential time. Can we use the same idea as for ALL_{DFA} of reducting to the corresponding emptiness problem, E_{NFA}, which we solved just as efficiently as for E_{DFA}? The problem is that we can't directly complement an NFA. Surely some other idea can help? The fact is, this problem is NP**-hard**. Nobody (on Earth) knows a polynomial-time algorithm, and most (on Earth) believe that no such algorithm exists.

Two-Machine Problems

Here the input w has type "Two Machines", meaning a pair $\langle M_1, M_2 \rangle$. If the input w does not have this pair form, it is rejected to begin with.

EQ_{DFA}:
INST: Two DFAs $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$.
QUES: Is $L(M_1) = L(M_2)$?

The fact that gives an efficient decision procedure is that two sets A and B are equal if and only if their symmetric difference $A \triangle B = (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$ *is empty*. The symmetric difference is often written $A \oplus B$, with $\oplus$ also used to mean XOR. Thus if we apply the Cartesian product construction to M_1 and M_2 with XOR as the operation, to produce a DFA M_3, then the answer is yes if and only if $L(M_3) = \emptyset$.

Solution:

1. Decode a given input string $w = \langle M_1, M_2 \rangle$ into DFAs M_1 and M_2. (If w does not have that form, reject.)
2. Create the Cartesian product DFA $M_3 = (Q_3, \Sigma, \delta_3, s_3, F_3)$ with $F_3 = \{(q_1, q_2) : q_1 \in F_1 \; XOR \; q_2 \in F_2\}$.
3. Feed $\langle M_3 \rangle$ to the decision procedure for E_{DFA}, and accept $\langle M_1, M_2 \rangle$ if and only if that accepts $\langle M_3 \rangle$.

If m is the maximum of the number of states in Q_1 and in Q_2, then step 2 runs in $O(m^2)$ time (ignoring the $\log m$ length of state labels). Step 3 is run on a quadratically bigger machine, so its own quadratic time becomes $O(m^4)$ overall, but that's AOK---still polynomial in m. But how about:

EQ_{NFA}:
INST: Two NFAs $N_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $N_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$.

QUES: Is $L(N_1) = L(N_2)$?

We can get a decision procedure by converting the NFAs into DFAs M_1 and M_2 and testing whether $L(M_1) = L(M_2)$. For decidability purposes, that is all we need to say, but it is inefficient. Can't we apply the Cartesian product idea directly to N_1 and N_2? If the operation is intersection or union, this makes a good self-study question, but for difference or symmetric difference/XOR, there is a clear reason for doubt: If we could solve EQ_{NFA} efficiently in general, then we could solve it efficiently in cases where N_2 is a fixed NFA that accepts all strings. Then we would have:

$$\langle N_1, N_2 \rangle \in EQ_{NFA} \iff \langle N_1 \rangle \in ALL_{NFA}.$$

But we have already asserted above that ALL_{NFA} is NP**-hard.** So this blocks the attempt to solve EQ_{NFA}, and in fact, this shows that the EQ_{NFA} problem is NP-hard as well.

One can define all these problems when the givens are regular expressions or GNFAs rather than DFAs or NFAs. The Sipser naming scheme will write the problems as EQ_{Regexp}, A_{GNFA}, ALL_{Regexp}, NE_{GNFA}, and so on. They are all **decidable** because regular expressions and GNFAs are convertible to NFAs and DFAs, but not always efficiently to the latter. Regular expressions and NFAs convert to and from each other especially efficiently, and so the problems subscripted "$_{Regexp}$" have much the same status as those subscripted "$_{NFA}$". When we extend the problems to context-free grammars, pushdown automata, and general (deterministic) Turing machines, however, we will "lose" a lot more.