CSE396 Lecture Tue. 4/13: Decision Problems and Procedures

The TMs we have seen show off some basic capabilities of Turing Machines

- copying a string
- comparing two strings
- searching for a matching (sub-)string on a tape
- arithmetic like $3n + 1$
- branching according to what char is read
- looping.

These operations suffice to build an interpreter for assembly code.

[Show the "Universal RAM Simulator" handout, discuss the Church-Turing Thesis, and explain why it enables machines to be specified in pseudocode from here on out.]

Theorem 1: For every program P written in any known executable programming language $\mathscr{L}$ (high-level or otherwise) that uses standard input and standard output, we can build a 3-tape Turing machine M_P such that whenever P given x on standard input writes y to standard output, M_P given x on its input tape writes y to a special output tape. If $P(x)$ halts, then $M_P(x)$ halts.

Proof: First, any compiler for $\mathscr{L}$ to a known code target can be converted into a compiler from $\mathscr{L}$ to the "mini-assembler"---which is essentially similar to what the text calls a RAM. So we can compile P to make an equivalent RAM program R_P. Then take M_P to be the Turing machine T in the handout, but with the binary text of R_P already written on its input tape. More precisely, M_P begins with a series of dedicated instructions that write out R_P char-by-char in front of any input x on its first tape, so it has $R_P\#x$ there. Then it just segues to the start state of T. $\boxtimes$

Theorem 2: We can build a **universal Turing machine**, meaning a single TM U that takes inputs of the form $\langle M, x\rangle$ and simulates $M(x)$.

Here $\langle M, x\rangle$ denotes an unspecified but transparent way of combining the code of M and the bits of x into a single string over whatever alphabet we need. In the Turing Kit, user-designed Turing machines M are stored as ASCII files, so that can be the code $\langle M\rangle$ of M. ASCII can be converted to strings over $\{0, 1\}$ if we so desire. The files are self-delimiting, so we can then define $\langle M, x\rangle$ by just appending x to $\langle M\rangle$. Or, assuming that neither M nor x has any commas or angle brackets, we can regard $\langle M, x\rangle$ as literally '⟨' then whatever string code of M, then comma, then x, and finally '⟩'. The choice of **tupling scheme** does not matter in detail.

Proof: The *Turing Kit* is a high-level Java program P that reads a TM M and an input x and executes $M(x)$. That is (essentially), $P(\langle M, x\rangle) = M(x)$. Then compile P to M_P as above and call it U. Then $U(\langle M, x\rangle) = P(\langle M, x\rangle) = M(x)$. This notation includes that $U(\langle M, x\rangle)\downarrow$ if and only if $M(x)\downarrow$. (The down arrow means "halts" while $\uparrow$ is read as "diverges" or "does not halt.") $\boxtimes$

Both this and the next theorem are usually proved in more specialized ways in textbooks.

Theorem 3: For every nondeterministic TM N we can build a deterministic TM M such that $L(M) = L(N)$.

Proof: The Turing Kit could be upgraded to a version T' that simulates a given NTM N on an input x by branching to try all possibilities, accepting if and when some branch accepts x. The program T' itself is deterministic. Hence so is the equivalent Turing machine $M_{T'}$ obtained from T' via Theorem 1. ☒

The one thing we don't know how to do is make T' avoid exponential branching, which slows down the time exponentially. This is different from the situation with an NFA N on a given input x, where we can simulate $N(x)$ by the trick of maintaining the current set R_i of possible states after each bit i of x, and thus avoid the exponential blowup of converting N into a DFA. Whether we can do a similar trick for a general NTM N is the infamous NP = ? P problem, which we will confront in the last week of the course. To set this up, we jump ahead a little to make the following definition.

Definition: A Turing machine M **runs in time $t(n)$** if for all n and inputs x of length n, $M(x)$ halts within $t(n)$ steps. If M is nondeterministic, all possible computations must halt within $t(n)$ steps.

For example, every DFA---and every NFA without ϵ-transitions---runs in time $t(n) = n + 1$, which is the fastest possible time that reads every input char and the blank that says the input is terminated. (This is sometimes called running in **real time**.) It is convenient to apply O-notation to time without caring about the exact number of steps. All the 2-tape machines we have seen have run in $O(n)$ time, which is called **linear time**, but some of the 1-tape machines have run in $\Theta(n^2)$ time, which is **quadratic time**. This is no accident:

Theorem: For any k-tape TM $M = (Q, \Sigma, \Gamma, \delta, \sqcup, s, F)$ that runs in time $t(n)$, we can build a 1-tape TM M' that simulates M and runs in $O(t(n)^2)$ time.

Proof Sketch: M' uses work alphabet $\Gamma' = \Gamma^k$, which can pack the k chars in any "column j" of the k tapes of M into one "superchar" in cell j on the one tape of M'. We also need chars that say whether they are currently being scanned by a tape head of M, so we actually have $\Gamma' = (\Gamma \cup \Gamma_{\odot})^k$ where $\Gamma_{\odot}$ is a "dotted copy" of Γ. Initially, $M'(x)$ converts each char x_i into the "superchar" $[x_i \sqcup \sqcup \sqcup \cdots \sqcup]$ which packs x_i and $k-1$ blanks into one char of Γ and rewinds its single tape head onto the superchar $[\wedge_{\odot} \sqcup_{\odot} \sqcup_{\odot} \cdots \sqcup_{\odot}]$ which lines up the k "virtual" tape heads of M on $\wedge$ and blanks below it to the left of x. Thereafter, M' simulates each step of M in one left-to-right **pass** that reads the k-tuple of scanned characters according to which parts of superchars have $_{\odot}$ and then a right-to-left pass that performs the corresponding instruction of δ. The total time for each pass is initially $2n + 4$ but can grow if and when M uses more tape cells beyond the end(s) of x. The width of a pass cannot be more than (twice the) time taken by M thus far, so it is always less than $t(n)$ (or less than $2t(n)$, if M uses cells to the left of x as well). Thus the total time is $O(t(n)^2)$. ☒

There are cases where quadratic time expansion cannot be improved, but that will be AOK when running in time $n^{O(1)}$, which is called **polynomial time**, is what we care about. Also FYI, if we have a "fair cost" running time function $t(n)$ for an algorithm in our favorite high-level programming language $\mathscr{L}$, then the 3-tape TM in the "Universal RAM Simulator" handout runs in time $O\big(t(n)^4\big)$ as coded and time $O\big(t(n)^2 \log n\big)$ if coded more cleverly. So "polynomial time" is the same for the basic 1-tape TM model as it is for our real programs [maybe unless **quantum computers** ever become real]. This all has three main takeaways, IMHO:

1. The Turing machine model (especially allowing 2 or 3 tapes) remains a quite realistic model of computation.
2. The Church-Turing Thesis extends to a polynomial-time version that claims all machines ever built (will) have broadly equivalent benchmarks for *feasible* time, and extends to say that *Nature is lexical*.
3. This enables us to regard procedures specified in prose to be fully equivalent to Turing machines and other computing models---even for broadly judging their time efficiency.

Point 3 includes saying that a total Turing machine (i.e., one that halts for all inputs), a **flowchart** in which every component terminates (conventionally enclosing it in a "solid box"), and pseudocode in which every loop terminates, can all be regarded as equivalent forms of a **decision procedure**.

Decision Problems

TopHat 4899

[The next material is not on Prelim II.] 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 $O(m^2)$ (again, one can do better with smarter data structures), so the whole time is $O(m^2 n)$, 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 $L(M) \neq \varnothing$?

The QUESTION is worded oppositely from the text's wording of E_{DFA}, which we'll come to. Here is an efficient decision procedure:

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, **accept** $\langle M \rangle$, else **reject**.

The BFS in step 2 terminates---indeed, in time $O(m^2)$ at worst since the graph has m nodes. [Well, it has $O(m)$ edges, so you can get better time with random access to good data structures.] The procedure is correct because if BFS finds a path from s to a state q in F, then the chars along that path form a string in $L(M)$, so $L(M) \neq \varnothing$.

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

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

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 $L(N) \neq \varnothing$?

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 $L(N) = \varnothing$?

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 $L(M) = \Sigma^* \iff L(M') = \varnothing$.

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 $L(M_3) = \varnothing$.

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.

Problems Involving Grammars

Again, let's "accentuate the positive" and start with the nonemptiness problem rather than the emptiness problem.

NE_{CFG}:
INST: A CFG $G = (V, \Sigma, \mathcal{R}, S)$.
QUES: Is $L(G) \neq \emptyset$? (Nerdy version: Is $L(G) \cap \Sigma^* \neq \emptyset$?)

The following pseudocode to solve the problem is strongly analogous to breadth-first search:

```
bool changed = true;
set<V ∪ Σ> LIVE = Σ; //constructed to have the terminals
while (changed) {
     changed = false;
     for (each rule A → W⃗ in ℛ such that A ∉ LIVE) {
          if (W⃗ is in LIVE*) {
               LIVE = LIVE ∪ {A};
               changed = true;
          }
     }  //LOOP INV: Every variable in LIVE can derive a terminal string
}
if (S ∈ LIVE) accept; else reject;
```

By the loop invariant, if S is ever added to `LIVE` then S derives some terminal string, which means that $L(G) \neq \emptyset$. Hence the algorithm is **sound**---that is, it never gives a false positive. Why does it always terminate, and why is it comprehensive---that is, why does it halt and catch all "yes" cases?

- Each iteration of the `while` loop either adds a new variable to LIVE or leaves `changed` false.
- If `changed` is left false, the loop terminates right there.
- The number of times it can add a new variable is limited by the size of V.
- Hence the `while` loop must terminate within $|V| + 1$ iterations.
- So the pseudocode defines a **total** Turing machine, that is, a **decider**.

Now why is it comprehensive? Suppose $L(G) \neq \emptyset$. Then there is a derivation of a terminal string x from S. The string x can be ϵ; this won't matter to the logic. The derivation has some number k of steps and can be represented abstractly as

$$S \implies \vec{X}_{k-1} \implies \vec{X}_{k-2} \implies \cdots \implies \vec{X}_2 \implies \vec{X}_1 \implies x$$

The vector signs are to remind that each sentential form $\vec{X}_i$ can include multiple variables and its own terminals as well. Note the indexing of i in reverse order. Each $\vec{X}_i$ has one variable that was expanded in the step---wlog. it is the leftmost variable in $\vec{X}_i$---and we can call it A_i. It can be the same variable in different steps but we'll still call them A_i. The first point is that A_1 must be the only variable in $\vec{X}_1$, because replacing it leaves a terminal string. Put another way, the right-hand side of the rule

$A_1 \to \vec{W}_1$

that was applied in the last step must be all terminals (we could have $\vec{W}_1 = \epsilon$, that's fine) and everything else in $\vec{X}_1$ must be terminals. Since all terminals are initially in the set LIVE, we have the following facts:

- The rule $A_1 \to \vec{W}_1$ has $\vec{W}_1 \in \texttt{LIVE}^*$.
- Hence the algorithm on the first iteration includes A_1 into LIVE.
- On the next iteration, $\vec{X}_1$ belongs to $\texttt{LIVE}^*$.
- Whatever rule $A_2 \to \vec{W}_2$ was applied at the next-to-last step, it has $\vec{W}_2 \in \texttt{LIVE}^*$ in that iteration.
- Hence the second iteration adds A_2 to $\texttt{LIVE}^*$ (if A_2 wasn't there already by virtue of being the same variable as A_1).

What this adds up to is that by induction on i we can prove the statement $Q(i) \equiv$ "the variable A_i is added to the set LIVE on or before the i-th iteration." Then with $i = k$ the variable "A_k" is none other than S.

- Thus the algorithm adds S to LIVE, and so it gives the true-positive answer "yes."
- This means the algorithm captures all true positives, so it is comprehensive.
- Since it has no false positives, it is correct.
- Thus the problem NE_{CFG} is decidable.
- Since whenever we have a total Turing machine, we can complement the language by interchanging q_{acc} and q_{rej}, the complementary problem E_{CFG} $\equiv$ "Given a CFG G, is $L(G) = \emptyset$?" is likewise decidable.
- The algorithm runs within time $O(|V| \cdot |\mathcal{R}| \cdot r)$, where $|\mathcal{R}|$ means the number of rules but we also have to allow for the maximum length r of the right-hand side of a rule. Since the size of the (string encoding $\langle G \rangle$ of the) grammar G can be reckoned as order-of $(|V| + r|\mathcal{R}|)$, this is at worse quadratic. Anyway, it is a polynomial-time decider for NE_{CFG} and E_{CFG}.

Now we consider a different problem but with a closely related solution. The name is not standard but is compatible with Sipser's naming scheme.

Eps_{CFG}:

INST: A CFG $G = (V, \Sigma, \mathcal{R}, S)$.
QUES: Is $\epsilon \in L(G)$? Equivalently (nerdily), is $L(G) \cap \epsilon^* \neq \emptyset$?

The algorithm needs changing only one line:

```
bool changed = true;
set<V> NULLABLE = ∅; //constructed to be the empty set
while (changed) {
     changed = false;
     for (each rule A → W⃗ in ℛ such that A ∉ NULLABLE) {
          if (W⃗ is in NULLABLE*) {
               NULLABLE = NULLABLE ∪ {A};
               changed = true;
          }
     }
}
if (S ∈ NULLABLE) accept; else reject;
```

The trick that gets this off the ground is our old friend $\emptyset^* = \{\epsilon\}$. Thus, in the first iteration, all variables A such that $A \rightarrow \epsilon$ is a rule get added to NULLABLE. (We could also have initialized the set NULLABLE this way.) Every variable B that is later added to NULLABLE truly derives ϵ, so the soundness of this algorithm is clear, and the reason it terminates within $|V| + 1$ iterations is the same. Its correctness is a self-study exercise. Now we are ready to address the problem A_{CFG}.

A_{CFG}: (The "**A**cceptance Problem *for* **CFG**s")
INST: A CFG $G = (V, \Sigma, \mathcal{R}, S)$ and a string $x \in \Sigma^*$.
QUES: Is $x \in L(G)$?

A decision procedure:

1. If $x = \epsilon$, apply the decision procedure for Eps_{CFG} and accept $\langle G, x \rangle$ iff it accepts $\langle G \rangle$.
2. Else, convert G into a Chomsky normal form grammar $G' = (V', \Sigma, \mathcal{R}', S')$ such that $L(G') = L(G) \setminus \{\epsilon\}$, so that $x \in L(G) \iff x \in L(G')$.
3. Noting that $S' \Longrightarrow^* x$ if and only if S' derives x in exactly $2n - 1$ steps, where $n = |x|$, we can exhaustively try all derivations of $2n - 1$ steps, and accept if and only if at least one of them derived x.

Step 1 runs in polynomial time, but as-stated, steps 2 and 3 do not. The issue with step 2 as presented in many other sources is that if we have a "long rule" like $A \rightarrow B_1 B_2 \cdots B_r$ where each B_j is nullable, the conversion says to add all rules obtained by deleting any sublist of $(B_1, \ldots . B_r)$. This makes 2^r sublists, each of which might produce a different rule, and so takes exponential time. But a nifty trick is that we can first shorten the rule using $r - 2$ dedicated single-use variables:

$$A \to B_1D_1, D_1 \to B_2D_2, D_2 \to B_3D_3, ..., D_{r-3} \to B_{r-2}D_{r-2}, D_{r-2} \to B_{r-1}B_r.$$

Then the overall number of rules is multiplied by at most $2r$, which keeps the expansion of the grammar within a polynomial factor of the original data-size of G. The text does something related to this in its own incremental way of handling nullable variables when it describes the conversion to Chomsky normal form ins ection 2.1. (The remaining details of that are still *FYI, skim/skip*.)

Step 3 can exponentiate 2^{n-1} or worse if there are at least 2 choices for the $n-1$ applications of a non-terminal rule in the derivation. However, there is a nifty **dynamic programming** algorithm that is sometimes mentioned in CSE331 or software-systems courses, called **CYK** or **CKY** for its authors Cocke, Kasami, and Younger. It does step 3 in polynomial time, thus completing a polynomial-time decider for A_{CFG}. (This is mentioned later in the text, but IMHO not so clearly.)

Since we did A_{CFG}, how about the corresponding "all"-type problem?

ALL_{CFG}:
INST: A CFG $G = (V, \Sigma, \mathcal{R}, S)$.
QUES: Is $L(G) = \Sigma^*$?

Shock fact: This problem is not decidable at all. Indeed, there does not even exist a Turing machine M such that $L(M) = \left\{\langle G\rangle : L(G) = \Sigma^*\right\}$, let alone one that is total. The proof of this will come later in Chapter 5, but we will reach it by starting with **undecidable** problems that involve Turing machines themselves as the data objects.

[This will be about the middle of Thursday's lecture. It will continue into the rest of Chapter 4, but with a different order of business:

- Less attention to the analogy with showing the real numbers are uncountable by diagonalization---skim/skip that.
- Instead of using the Halting Problem as the first undecidable problem, or its close cognate the "Acceptance Problem" $A_{TM} = \{\langle M, w\rangle : w \in L(M)\}$ for deterministic Turing machines M, it will start with the "diagonal language" $D_{TM} = \{\langle M\rangle : \langle M\rangle \notin L(M)\}$.
- The text implicitly uses D_{TM} in its undecidability proof at the end of chapter 4, but refers to it in terms of an impossible machine rather than a language. I will try to make clear what stuff is real and what is nonexistent/counterfactual/"quixotic". The language D_{TM} is real.

I often leave the diagonalization proof D_{TM} as a "cliffhanger" at the end of one lecture and revisited at the start of the next lecture, but we may still be mid-lecture. Then I will finish showing that the A_{TM} language is undecidable, even though (unlike D_{TM}) it is **computably enumerable** (**c.e.**, with many synonyms for both "c.e." and "decidable" being out there).]
;