CSE396 Lecture Tue. Mar. 9: Regular Languages Summary / Context-Free Grammars

MNT Example 10: $L = \left\{a^r b^s : s \geq r-1\right\}$. Proof: Take $S = a^+$. Clearly S is infinite. Let any $x, y \in S$ $(x \neq y)$ be given. Then we can write $x = a^m, y = a^n$, where $m, n \geq 1$ and wlog. $m < n$. Note that since $1 \leq m < n$, we have $n \geq 2$, so $n-2$ is a legal number of chars for a string. Take $z = b^{n-2}$. Then $yz = a^n b^{n-2}$ which is **not** in L since $n-2$ is not $\geq r-1$ when $r = n$. But $xz = a^m b^{n-2}$ is in L since $m < n$ so $m-1 \leq n-2$ as required with $s = n-2$.
Proof does work: this shows S is PD for L, and since S is infinite, L is nonregular by MNT. ☒

The issue with the original choice $S = a^*$ is that the cases $m = 0$ and $m = 1$ are actually equivalent: $\epsilon \sim_L a$. Thus $S = a^*$ is NOT PD for L. This S is actually "too big." Fix: Take $S = a^+$. Note that "at bottom", a and aa are not equivalent: $a \nsim_L aa$ because they are distinguished by $z = \epsilon$: $az = a \in L$ because we have $r = 1, s = 0$, and $s \geq r-1$ does hold. But $yz = aa \notin L$ because that gives $r = 2$ and $s = 0$, and 0 is not $\geq 2-1$. [The proof above is now correct.]

First, a review of the course to date that paints a philosophical big-picture with emphasis on *algorithms* associated to the concepts we have learned. We have shown the equivalence of three ways of representing a regular language A:

1. Via a regular expression r such that $L(r) = A$.
2. Via an NFA N such that $L(N) = A$.
3. Via a DFA M such that $L(M) = A$.

We have also characterized nonregular languages, which don't have any of these representations. The fact that three separate formalisms (GNFAs are IMHO lumped somewhere between NFAs and regexps but not really independent) yield the same class REG of languages shows that being regular is a ***salient*** concept. (My word, not in any text, though many sources say "robust.")

The equivalence is only about potential to give a language. It does not say

- how efficiently, or
- how useful the representation is for testing strings and combining languages.

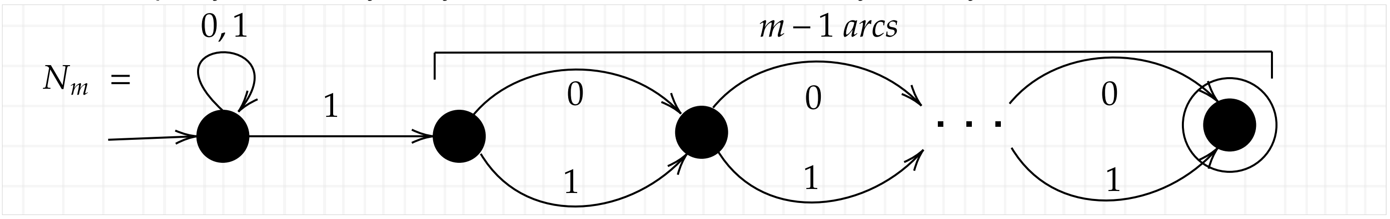
On efficiency, the languages $L_m = \left\{x \in \{0,1\}^* : \textit{the mth bit from the end is a } 1\right\}$ give a great example:

1. The regular expressions $r_m = (0+1)^* 1 (0+1)^{m-1}$ have only 12 symbols plus the bits of $m-1$ in binary notation, thanks to powering as an abbreviation, which gives size $O(\log m)$.
2. The NFAs N_m we saw have $m+1$ states and $2m+1$ instructions, for size $O(m)$. Regular expressions without powering are similar: $(0+1)^* 1 (0+1)(0+1) \cdots (0+1)$ [$m-1$ times].

3. But DFAs need exponentially many states and instructions: 2^m states is minimum, because by MNT, $S = \{0,1\}^m$ is a PD set for L_m that has size 2^m.
4. Converting any DFA or NFA back to a regexp is painful if you copy it out longhand, but if the $m \times m$ matrix operations are manipulated via references and list data structures, it counts as an $\widetilde{O}(m^3)$-time algorithm, which is a **polynomial-time** algorithm (along with 1 and 2).

```
for k = m downto 2:
    for i = 1 to k-1:
        for j = 1 to k-1:
            T(i,j) = T(i,j) + T(i,k) · T(k,k)* · T(k,j).
```

Here, regular expressions can be incredibly efficient, and NFAs are efficient too. But that ***succinctness*** can blow up in your face if you try to convert them to DFAs. Why would you want to convert them?



1. Running a k-state DFA M on a string x of length n takes time only $O(n \log m)$ with good data structures. If we ignore log factors, we can call this $\widetilde{O}(n)$, basically linear time.
2. Running a k-state NFA N on x: If you convert it to a DFA, it could take $O(2^m)$ time. Instead, track the sets R_i of possible states (from the NFA-to-DFA proof) and how they changed while reading bit i of x. Time: $\widetilde{O}(nm)$ or at worst $\widetilde{O}(nm^2)$ for "bushy" NFAs. This is an example of **polynomial time** versus **exponential time**, the last main course topic.
3. When you match a string x to a regexp r in a scripting language or OS command line, the system builds the equivalent NFA N_r and runs N_r on x. If the system disallows numerical powering, and disallows other operations in r, this doesn't blow up.

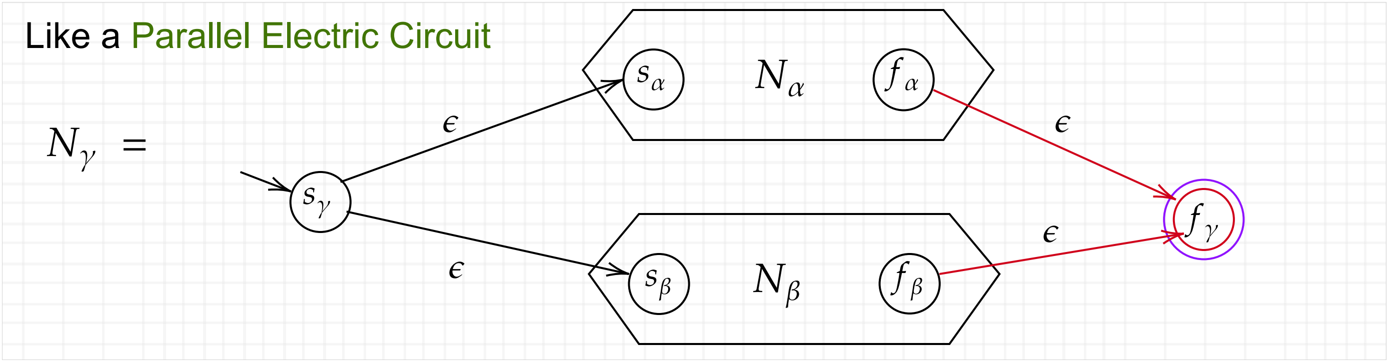
What operations should we be wary of? How about moving from a language A to its complement $\widetilde{A}$:
1. If we have a DFA $M = (Q, \Sigma, \delta, s, F)$ such that $L(M) = A$, it's cinchy: Just build $M' = (Q, \Sigma, \delta, s, Q \setminus F)$ by switching accepting and rejecting states, and then we get $L(M') = \widetilde{A}$ right away.
2. If we have an NFA $N = (Q, \Sigma, \delta, s, F)$, this trick does not work. Doing this to N_m above makes the start state eternally accepting, thus trivializing the language to be Σ^*, not $\sim L_m$. The best we know in general is to convert to a DFA M, but that can blow up exponentially.
3. If we have a regexp r, the same often goes for the NFA N_r. (Have you seen a regular expression package that allows general complementation, as opposed to just allowing the exclusion of certain sets of characters at lowest level?)

How about binary Boolean operations $C = A$ *op* B that involve negating one or both languages, such as difference $A \setminus B$ or symmetric difference $A \triangle B$ (often written $A \oplus B$ when $\oplus$ is used as the symbol for XOR). The considerations are similar to those for complementation:

1. If we have DFAs $M_A = (Q_A, \Sigma, \delta_A, s_A, F_A)$ and $M_B = (Q_B, \Sigma, \delta_B, s_B, F_B)$, then we can use the Cartesian product construction to build $M_C = (Q_C, \Sigma, \delta_C, s_C, F_C)$ by
   - $Q_C = Q_A \times Q_B$
   - $s_C = (s_A, s_B)$
   - $\delta_C((q_A, q_B), c) = (\delta_A(q_A, c), \delta_B(q_B, c))$, and finally
   - $F_C = \{(q_A, q_B) : q_A \in F_A \ \mathit{op} \ q_B \in F_B\}$.

     If M_A and M_B both have m states, then M_C has (at most) m^2 states, making this an $\widetilde{O}(m^2)$-time algorithm. Thuis this is also polynomial time.
2. But if we are given NFAs N_A and N_B, Cartesian product won't work---at least not with negation in general. We might be left needing exponential time. Note, as a general word to the wise, $\widetilde{A} = \Sigma^* \setminus A = \Sigma^* \triangle A$, so these operations include complements as a special case. Same issue if we are given regexps---why we don't have them as basic operations.

This finally brings us to *intersection* $\cap$ vis-à-vis *union* $\cup$. When we have regexps r_A and r_B, union is a basic operation: $r_C = r_A \cup r_B$ (text) or $r_C = r_A + r_B$ (notes and other sources, though regexp packages generally use $r_C = r_A \mid r_B$). If you have NFAs N_A and N_B it is almost as easy:



With a linked-list representation, this is actually $O(1)$ time. *But is there such an easy way to "rewire our brains" for intersection?*

There is some "psych" evidence of not. If we have a list of rules or requirements we have to satisfy, it is easier for us if the list is an OR, because we only have to scan to find one clause that works and can then forget the others. If it is an AND, we have to keep everything in mind until the end. If a sequence of directions is like a recipe where we can read the next step after doing the previous one, then it is like AND THEN, which is easier than when you really do have to read all the directions in advance before doing the first step. Note that AND THEN corresponds to the basic regexp operation of concatenation $\cdot$, which in turn relates to series circuits.

The "psych" evidence is even more for negation. Because $\cap$ is a monotone operation, getting a regexp r_C such that $L(r_C) = L(r_A) \cap L(r_B)$ is actually not horrible. You can convert the given regexps to NFAs N_A and N_B and do "Cartesian for $\cap$" directly on them with $F_C = F_A \times F_B =$

$\{(q_A, q_B):$ both q_A and q_B are accepting in their respective machines$\}$. Then convert the resulting NFA
N_C into r_C. Unlike with negation, this is a guaranteed polynomial-time algorithm, but it is still more painful (quadratic for Cartesian, cubic for r_C) than just putting in the $\cup$ operator is for union!

We will see the difference between AND and OR become even more "imprinted" in the next unit: **Context-Free Languages** (**CFL**s) have easy use of $\cup$, but the class CFL of CFLs will be seen not to be closed under $\cap$ at all! Nor under $\sim$ either (since it is closed under union, if it were closed under complementation then it would be closed under all Boolean ops after all).

Context-Free Grammars **[new material not on next week's exam]**

These came out in papers by Noam Chomsky and then in his 1957 book *Syntactic Structures*. Let's have a blast of syntax before we see examples and larger motivations.

Definition: A **context-free grammar** (**CFG**) is an object $G = (V, \Sigma, \mathcal{R}, S)$ where:

1. V is a finite alphabet of *variables*, aka. *non-terminals*.
2. Σ is the *terminal alphabet*.
3. S, a member of V, is the *start symbol*.
4. $\mathcal{R}$ is a finite set of *rules*, each of the form $A \to X$, where $A \in V$ and $X \in (V \cup \Sigma)^*$. More simply put, $\mathcal{R}$ is a subset of $V \times (V \cup \Sigma)^*$.

Common alternative notations are T in place of Σ and $\mathcal{P}$ for "productions" in place of $\mathcal{R}$ for "rules." Variables are commonly denoted by capital letters $A, B, C, D, \dots$ This can confuse with languages but hopefully not in-context, and besides, each variable A will stand for the language L_A of strings that can be derived from that variable, with $L(G) = L_S$. A common alternate notation is to put variables in angle-brackets, such as $\langle sentence \rangle$ in place of S. Then also the arrow is often written ":=" or "::=" which goes into **Backus-Naur Form** (**BNF**) grammars. BNF is more liberal than CFG notation but stays equivalent in the languages that can be represented.

Ordinary strings in Σ^* will be put in lowercase as $x, y, z, w, \dots$ as usual. But strings that can include variables will be emphasized by putting them in uppercase as $X, Y, Z, W, \dots$ etc. Two or more rules for the same variable are often grouped using | between the alternatives, for instance $A \to X \mid Y \mid Z$. This hints right away that CFGs are fundamentally nondeterministic and that the class of languages we get will be closed under union. Usually V, Σ, and the start symbol can be inferred from how the rules are laid out, so it is only necessary to state the rules in order to specify a context-free grammar G. Here is how to define the language $L(G)$:

Definition: Given a CFG G and two strings $Y, Z \in (\Sigma \cup V)^*$, we write $Y \Longrightarrow_G Z$ and say that Y derives Z in one step if there are strings $U, W \in (\Sigma \cup V)^*$ and a rule $A \to X$ in $\mathcal{R}$ such that

$$Y = UAW \quad \text{and} \quad Z = UXW.$$

We say that A was re-written to X by the rule. This is called "context-free" because none of the letters next to A in Y matter. We drop the subscript G as it is usually clear. Now we do one final "inductive escalation":

Definition:

- Any string $X \in (\Sigma \cup V)^*$ is considered to "derive itself in zero steps": $X \implies^0 X$.
- For $k \geq 1$, write $X \implies^k Z$ if there is a Y such that $X \implies^{k-1} Y$ and $Y \implies Z$.
- Then write $X \implies^* Z$ if there is a $k \geq 0$ such that $X \implies^k Z$.
- Finally, $L(G) = \left\{x \in \Sigma^* : S \implies^* x\right\}$.

An $X \in (V \cup \Sigma)^*$ such that $S \implies^* X$ is called a **sentential form**. Sometimes we speak of "the language of sentential forms", but strictly speaking the language of the grammar is the set of derivable terminal strings only. Note that the superscript * keeps its reading of "zero or more" (steps). Now let's see some basic examples of grammars and derivations.

Example

$G = S \rightarrow 0S1 \mid \epsilon$ Here the start symbol S is the only variable, and $\Sigma = \{0, 1\}$. Some derivations:

- $S \implies \epsilon$. This is a one-step derivation using the rule $S \rightarrow \epsilon$. The empty string counts as a terminal string since it belongs to Σ^* (as well as to $(\Sigma \cup V)^*$). So $\epsilon \in L(G)$.
- $S \implies 0S1 \implies 01$. The first step used the rule $S \rightarrow 0S1$, then we used the ϵ-rule. (Not all grammars have ϵ-rules, and we will later want to eliminate them from those that do.)
- $S \implies 0S1 \implies 00S11 \implies 0011$.

This is enough to give the idea of why $L(G) = \left\{0^n 1^n : n \geq 0\right\}$.

[I put the above example back the way it was. This is as far as I got on Tuesday. My lecture on Thursday will pick up here.]

Example

$G = S \rightarrow \epsilon \mid (S) \mid SS$. Again S is the only variable, but $\Sigma = \{(,)\}$ instead of $\{0, 1\}$.

The first two rules are much the same as in the first grammar (the order of writing the possible right-hand sides does not matter, and the only reason the order of writing rules for different variables might matter is if you need to put rule(s) for the start symbol first in order to say which it is). But the third rule "expands" by having two (or more) variables on the RHS.

- We can derive ϵ, $()$, $(())$, $((()))$, etc., much as in the previous grammar.
- But we can also do $S \implies SS \implies (S)S \implies ()S \implies ()(S) \implies ()()$ to get other kinds of balanced-parenthesis expressions.
- In fact, $L(G) = BAL$.

Example

$G' = S \to \epsilon \mid (S) \mid (S)S$ is another grammar involving parentheses.

- Anything G' can derive can be derived in G because the rule $S \to (S)S$ in G' can be **simulated** by the two steps $S \implies SS \implies (S)S$ in G.
- Hence $L(G') \subseteq L(G)$. Since we already asserted that $L(G) = BAL$, this means $L(G') \subseteq BAL$. We therefore say that G' is **sound** for BAL.
- Is $L(G') = BAL$, which would follow if $L(G') \supseteq BAL$? The latter I call G' being **comprehensive** for BAL.
- The combination of being sound and comprehensive just means being *correct*.
- In fact, yes, G' is comprehensive---but that is often not as easy to prove as soundness.

Example

$G'' = S \to \epsilon \mid 0S \mid \$S \mid \$SDS$ generates all strings in the spears-and-dragons game with unlimited spears in which the "Player" survives. We can amke it look more like G' by changing "spear" to (and "dragon" to) and ignoring 0 for "empty room":

$$G'' = S \to \epsilon \mid (S \mid (S)S.$$

Then $L(G'')$ is the language of parenthesis expressions that can be properly closed by appending zero or more right parens.

Discussion

Sense and syntax: "Colorless green ideas sleep furiously".
Balance.