## CSE396 Lecture Tue. Week 6: Regular Languages Summary / Context-Free Grammars

First, two more Myhill-Nerode examples. Here REG stands for the class of regular languages.

**Example 9**: Define $A = \{a^N : N \textit{ is a perfect square}\}$, which equals $\{a^{n^2} : n \in \mathbb{N}\}$.

Take $S = a^*$. Clearlty $S$ is infinite. Let any $x, y \in S$ ($x \neq y$) be given. Then we can write $x = a^m$ and $y = a^n$ where wlog. $m < n$. Put $k = n - m$. The key numerical fact about perfect squares is that the gaps between successive squares grow bigger and bigger. So we can find $r$ such that $(r+1)^2 - r^2 > k$, and for good measure, such that $r^2 > m$. Take $z = a^{r^2-m}$. Then $xz = a^m a^{r^2-m} = a^{r^2}$, which belongs to $A$. But

$$yz = a^n a^{r^2-m} = a^{k+m} a^{r^2-m} = a^{r^2+k},$$

which is not long enough to get up to $a^{(r+1)^2}$, which is the next member of $A$. So $yz \notin A$, giving $A(xz) \neq A(yz)$ legitimately this time. So $S$ is PD for $A$, and since $S$ is infinite, $A \notin$ REG by the Myhill-Nerode Theorem. ☒

**Example 10**: $L = \{a^r b^s : s \geq r - 1\}$. 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 choosing $S = a^*$ is that the cases $m = 0$ and $m = 1$ are actually equivalent if you're only considering $z$ as a string of $b$'s. In general, the presence of even just one equivalent pair in a set called "$S$" invalidates the proof---having infinitely many distinguishable pairs is not enough. Taking $S = a^+$ fixed this problem. (Yes, $a^*$ is PD as well,but more trouble to show.)

A more common major error is to take $S = a^*b^*$ (or $S = a^+b^+$) in cases like this. I call this the "Too Many Stars" problem. A general pair $x, y \in S$ then has to be represented as $x = a^k b^\ell$, $y = a^m b^n$ for **four** different numbers---where you could have $k = m$, as all you know is that one of $k \neq m$ and $\ell \neq n$ has to be true. Too complicated! Saying $x = a^k b^k$, $y = a^m b^m$ is not a general choice from this $S$. It would, however, be a general choice from $S' = \{a^i b^i : i \geq 0\}$ if you defined it that way from the get-go. This $S'$ is not regular, but all we need is that $S'$ is infinite. [Are there any esthetically appealing Myhill-Nerode cases where the choice of $S$ must itself be a non-regular set? Neither my brain, nor Google, nor colleagues have come up with one for me. Just $S = a^*$ and simple variations like $a^+$ or $a^*b$ (and ditto for the alphabet $\{0, 1\}$) are good enough in the vast majority of cases.]

## Distinguishing States and Unique Minimum DFAs

The full Myhill-Nerode Theorem is an exact characterization of regularity. The phrase "if and only if" is sometimes shortened to "iff".

**Theorem**. A language $A$ is regular **if and only if** all PD sets for $A$ are finite.

The direction we've proved and applied so far is that **if** there is an infinite PD set for $A$, **then** $A$ is nonregular. The logical ***contrapositive*** of this is, **if** $A$ is regular, **then** all PD sets for $A$ are finite. We need to prove the ***converse*** of this latter statement: **if** all PD sets for $A$ are finite, **then** $A$ is regular.

The definition of a PD set is that its strings all belong to different equivalence classes of the relation $\sim_A$. The number of equivalence classes of an equivalence relation is called its **index**. Hence a more abstract but pithier statement of the Myhill-Nerode Theorem is;

**Theorem′**. A language $A$ is regular **if and only if** the relation $\sim_A$ has finite index.

**Proof:** To prove the converse, we need to prove that **if** $\sim_A$ has finite index **then** $A$ is regular. Let us define $\mathbb{Q}$ to be *the* set of equivalence classes of the relation $\sim_A$. The assumption of finite index means that $\mathbb{Q}$ is finite---so it can be the set of states of a DFA $M = (\mathbb{Q}, \Sigma, \delta, s, F)$. To express the other components, we need one more definition: for any string $x$, $[x]_A$ denotes the equivalence class of $\sim_A$ that $x$ belongs to. Note that we can have $x \neq y$ and yet $[x]_A = [y]_A$, but when $x$ and $y$ belong to a PD set $S$, the definition of $S$ being PD means $[x]_A \neq [y]_A$. Now put:

- $s = [\epsilon]_A$
- $\mathcal{F} = \left\{[x]_A : x \in A\right\}$
- $\delta = \left\{([x]_A, c, [xc]_A) : x \in \Sigma^*, c \in \Sigma\right\}$.

If the language $A$ is infinite, you might think this makes $\mathcal{F}$ be an infinite set, but no---if $x$ and $x'$ are both in $A$ and in the same equivalence class of $\sim_A$, then $[x]_A$ and $[x']_A$ are the same, so they don't count as different members of the set. This goes even more for the definition of $\delta$. The definition ***works*** because whenever $x \sim_A x'$, we get $xc \sim_A x'c$ for every character $c$, so $[xc]_A$ and $[x'c]_A$ are again the same equivalence class---hence the same state of $M$. This also means that $\delta([x]_A, c) = [xc]_A$ is a function from $\mathbb{Q} \times \Sigma$ to $\mathbb{Q}$. This makes $M$ be a DFA, and $L(M) = A$ because $M$ always processes a string $x$ from $s = [\epsilon]_A$ to the state $[x]_A$, which belongs to $\mathcal{F}$ exactly when $x$ belongs to $A$. Thus $A$ is regular. ☒

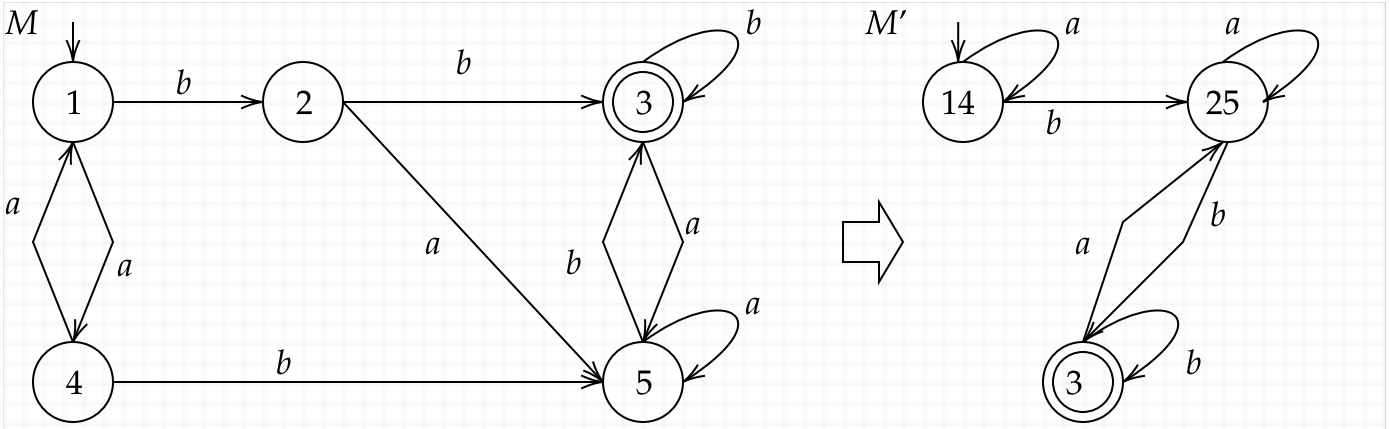
**Corollary** (to the MNT): Every regular language $A$ has a minimum-size DFA $M_A$ that is unique.

Unfortunately, the MNT does not do much to help you build an efficient *algorithm* to *find* $M_A$. The one thing we do know is that once you *have* a DFA $M = (Q, \Sigma, \delta, s, F)$ such that $L(M) = A$, no matter how wasteful, you can always efficiently *refine* it down to the unique optimal $M_A$. The key definition

uses the auxiliary notation $\delta^*(q, z)$ to mean the state that $M$ (being a DFA) uniquely ends at upon processing the string $z$ from state $q$. Inductively, $\delta^*(q, \epsilon) = q$ for any $q$, and further for any string $w$ and char $c$, $\delta^*(q, wc) = \delta\big(\delta^*(q, w), c\big)$.

**Definition**: Two *states* $p$ and $q$ in a DFA $M$ are **distinguishable** if there exists a string $z$ such that one of $\delta^*(p, z)$ and $\delta^*(q, z)$ belongs to $F$ and the other does not. Otherwise they are **equivalent**.

Two equivalent states must either be both accepting or both rejecting, because if they are one of each then they are immediately distinguished by the case $z = \epsilon$. There is a simple sufficient condition: If $p$ and $q$ are both accepting or both rejecting, and if they go to the same states on the same chars (that is, if $\delta(p, c) = \delta(q, c)$ for all $c \in \Sigma$), then they are equivalent. But otherwise, it can be hard to tell equivalence. There is an algorithm for determining this that is covered in some texts, and also appears in some Algorithms texts as an example of "dynamic programming."




States 2 and 3 are distinguished by $\epsilon$ since 2 is rejecting and 3 is accepting. Ditto 3 and 5. States 2 and 5 are equivalent, however, because both are rejecting and both go to 5 on $a$ and to 3 on $b$. States 2 and 4 are distinguished by $z = b$ because 2 goes to an accepting satte on $b$ but 4 does not. Ditto states 1 and 2. But states 1 and 2 are equivalent. This is harder to see immediately, but is because they go to each other on $a$ and go to equivalent states on $b$. The unique minimum DFA $M'$ such that $L(M') = M$ is shown at right.

We will focus on the distinguishing side instead. The following are good self-study points about any DFA $M$ with language $A = L(M)$:

- If $x \not\sim_A y$ (in words, if $x$ and $y$ are distinctive for the language $A$) then in any DFA $M$ such that $L(M) = A$, $\delta^*(s, x)$ and $\delta^*(s, y)$ must be distinguishable states---not just different states.
- If $x \sim_A y$, then $\delta^*(s, x)$ and $\delta^*(s, y)$ must be equivalent states.
- If $\delta^*(s, x)$ and $\delta^*(s, y)$ are distinguishable states, then $x \not\sim_A y$.
- If $S$ is a PD set for $A$, then the strings in $S$ must all get processed from $s$ to different states.

## What Does Regularity Mean?

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

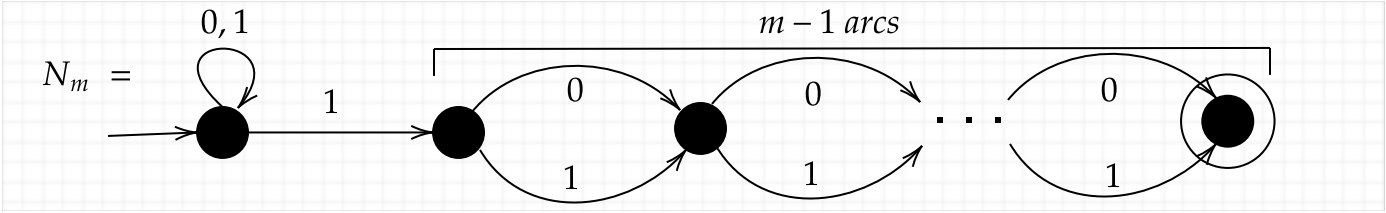
- how efficiently, or
- how useful the representation is for testing strings and combining languages.

On efficiency, the languages $L_m = \{x \in \{0,1\}^* : \textit{the mth bit from the end is a } 1\}$ 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 an $m$-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 an m-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.

## Closure Under Boolean Operations and (Brain) Efficiency

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 $\Sigma^*$, 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?)

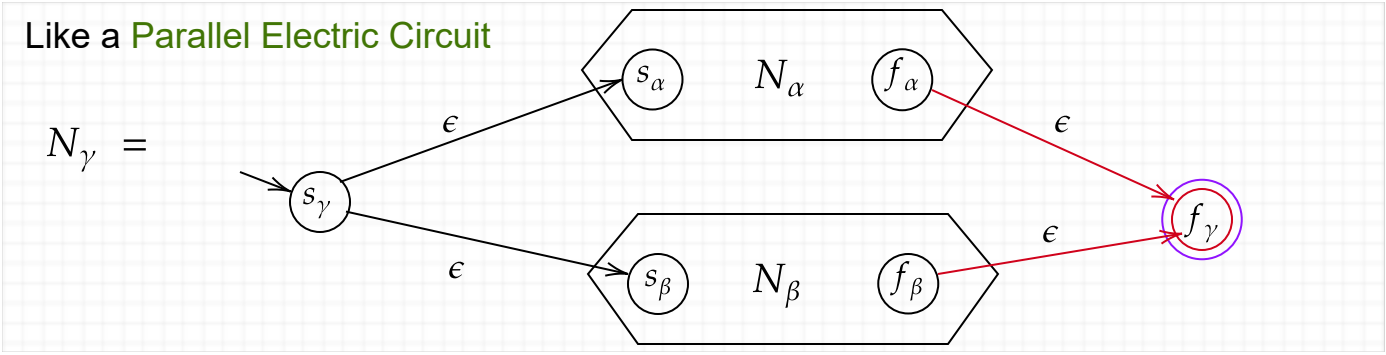
How about binary Boolean operations $C = A \; op \; B$ that involve negating one or both languages, such as difference $A \setminus B$ or symmetric difference $A \triangle B$ (often written $A \oplus B$ when $\oplus$ is used as the symbol for XOR). The considerations are similar to those for complementation:

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
   - $F_C = \{(q_A, q_B) : q_A \in F_A \; op \; q_B \in F_B\}$.

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:




The wiring takes only $O(1)$ 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 $F_C = F_A \times F_B = \{(q_A, q_B) : \text{ both } q_A \text{ and } q_B \text{ are accepting in their respective machines}\}$. 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).