Exam 2 Review Sheet

Regular operations and Regular expressions

Regular operations are the three operations that are applied to sets (for our purposes, those sets are languages Σ).

They are as follows:

operation	symbol	what it does
union	$\cup$	usual union set op.
concatenation	$\circ$, $\cdot$, or none	glues strings together
Kleene star	*	zero or more of elt.

Let the number of elements in sets A and B be $|A|$ and $|B|$. Then:

- The number of elements in $|A \cup B|$ is $|A|+|B|-|A \cap B| \leq |A| + |B|$
- The number of elements in $|A \circ B|$ is at most $|A| \times |B|$ (because concatenations are basically ordered pairs) but can be as low as 0 if either A or B is $\emptyset$
- The number of elements in $|A^*|$ is either finite, in which case it is either 0 (if $A = \emptyset$) or 1 (if $A = \{\lambda\}$), or infinite, in which case it is $\aleph_0$

Regular expressions are strings made out of parentheses, the letters in Σ, and the symbols $\cup, \circ, *, \lambda, \emptyset$. They are made to represent a **regular language**, which is obtained by finitely many applications of regular expressions to the sets $\emptyset$, $\{\lambda\}$, and $\{i\}$ for all $i \in \Sigma$. This boils down to the fact that regular expressions are a shorthand for representing regular languages – that is, languages that are defined by the three regular operations on some alphabet.

Examples (over $\Sigma = \{a, b, c\}$):

- Even length: $((a \cup b \cup c)(a \cup b \cup c))^*$
- Odd length: $((a \cup b \cup c)(a \cup b \cup c))^*(a \cup b \cup c)$
- Length ≤ 2: $(a \cup b \cup c \cup \lambda)(a \cup b \cup c \cup \lambda)$
- Contains `abcb` or `bca`: $(a \cup b \cup c)^*(abcb \cup bca)(a \cup b \cup c)^*$
- Contains odd number of `a`'s: $((b \cup c)^*a(b \cup c)^*a(b \cup c)^*)^*a(b \cup c)^*$
- Contains as a substring both `abb` and `bca`: omitted for brevity (accounting for all ordering cases and overlaps makes it really long)

Theorem (proven in NFA part by Kleene's Thm. equivalence): If there is a regex for some language, there is a regex for its complement.

Context-free grammars and languages

A **context free grammar** is a structure defined as $G = (\Sigma, V, P, S)$ where Σ is the usual kind of alphabet (set of letters), V is the set of variables (also called an alphabet of variables) used in the grammar (which does not overlap with Σ: so, $V \cap \Sigma = \emptyset$), S is the designated start symbol (note that $S \in V$ always, so $V \neq \emptyset$), and P is the set of rules ("productions") for going from one variable to another variable or string. P is a subset of $V \times (\Sigma \cup V)^*$ because we're going from V to some number of elements of the intersection of Σ and V. (Note further that P is finite despite the fact that $|(\Sigma \cup V)^*| = \aleph_0$.)

We say that if $(A, w) \in P$, A is a variable in V, and w is some "word" made of variables and letters and is in $(\Sigma \cup V)^*$. We can also write $(A, w) \in P$ as $A \to w$, spoken "A derives w".

If $A \to w_1$, and $A \to w_2$, $A \to ...$ etc., we can write $A \to w_1|w_2|...$.

Example CFG (1):

- $G = (\Sigma, V, P, S)$
- $\Sigma = \{a, b, c\}$
- S is the start state
- $V = \{S, A, B, D\}$
- P :
  - $S \to AB|BD$
  - $A \to aA|c$
  - $B \to Bb|ca$
  - $D \to abD|\lambda$

The CFG operates as follows. Start with the start symbol S. Then, proceed to the next step, of AB and BD (recall that the pipe is the symbol for union and that the lack of space between, say, AB and BD is indicative of their concatenation). We continue substituting into variables with every possibility (with branching paths denoted by the union operators) until we have created every string possible from the CFG.

More formally, we say that a variable $E \in V$ **derives a sentence** $w \in (\Sigma \cup V)^*$ in one step if $E \to w$ is in P. For example, B derives the string ca in one step.

Then, inductively, we say that variable $E \in V$ derives a sentence $x \in (\Sigma \cup V)^*$ in $n + 1$ steps if all of the following hold:

1. w derives some $y \in (\Sigma \cup V)^*$ in n steps,
2. y is equal to some concatenation $y_1 F y_2$, where y_1 and y_2 are strings in $(\Sigma \cup V)^*$ and $F \in V$ is a variable,
3. There is some rule $F \to y_0$ in P for some string $y_0 \in (\Sigma \cup V)^*$, and
4. The concatenation $y_1 y_0 y_2$ is equal to x.

We can make context-free grammars for regular expressions using a couple constructs, which are associated with the regular operators of union, concatenation, and the Kleene star. We have already seen the union operation in the form of the pipe (|). The concatenation operation is simply denoted by

putting two variables or letters or both next to each other. Finally, the Kleene star is described using the following examples:

Example: Turn the regular expression a^*c into a CFG.

We can do this with the following (single) rule: $S \to aS|c$. This prepends zero or more a's to the beginning of the string, and adds a c to the end when it is done so that the string always ends in c. We can see that A from example (1) is the same as this expression.

Similarly, from example (1), we can see that $B \to cab^*$ because we are appending zero or more b's to the end of string B, which must begin with ca. D must go to $(ab)^*$ because we are prepending ab zero or more times to D, which must be ended by λ, the empty string.

Not every CFG needs to define a regular language. For example, the language $L = \{a^n b^n | n \geq 0\}$ cannot be defined by a regular expression. Two attempts might be $(ab)^*$ and a^*b^*, but in the first, we have $abab$ contained and in the second we have aab contained, so neither work. The CFG for L could be the usual (with $\Sigma = \{a, b\}$) with $P : S \to \lambda|aSb$. In this grammar we prepend an a and append a b the same number of times (n), which is in line with what we want from L.

A more complex example of this type might be the language $L = \{a^n ccb^{2n} | n \geq 0\}$. We say that this *telescopes* as there are equivalent exponents at either end. In this example our rule is $S \to aSbb|cc$, as we prepend one a for each two b's we append, and have two c's in the middle.

While not all languages generated by CFGs need to be regular, we can use the regular operations of union, concatenation, and the Kleene star on CFGs G_1 and/or G_2 to create a new CFG G (the set of languages generated by context-free grammars is closed under the regular operations).

For the union operator $L(G_1) \cup L(G_2)$, we create a new start state S separate from any of the states in G_1 or G_2 and add the new rule to P that $S \to S_1|S_2$.

For the concatenation operator $L(G_1)L(G_2)$, we create a new start state S separate from any of the states in G_1 or G_2 and add the new rule to P that $S \to S_1S_2$.

For the Kleene star operator $L(G_1)^*$, we create a new start state S separate from any of the states in G_1 and add the new rule to P that $S \to \lambda|SS|S_1$. This rule tells us that S can make zero or more copies of itself.

Next, we want to try turning regular expressions e into CFGs (we know that we can't necessarily do the opposite from the above telescoping examples). We define our algorithm to do this recursively, with base cases of the letters $i \ \forall i \in \Sigma$, the empty string λ, and the empty language $\emptyset$, which are the zero-operator regular expressions. The CFG rules for these are as follows:

regex	rule
$i \ \forall i \in \Sigma$	$S \to i$
λ	$S \to \lambda$
$\emptyset$	no rules

What we then do is split the regular expression e up into CFG variables by operator precedence. We consider the example $e = ab^*(a \cup b) \cup (bc \cup a)^*$:

$$\underbrace{a \underbrace{b^*}_{D} \underbrace{(a \cup b)}_{E}}_{A} \cup \underbrace{\underbrace{(bc \cup a)^*}_{F}}_{B}$$

This results in the following rules:

- $S \to A|B$
- $A \to aDE$
- $D \to bD|\lambda$
- $E \to a|b$
- $B \to \lambda|BB|F$
- $F \to bc|a$

Deterministic finite automata

We define a **deterministic finite automaton** or DFA as a structure $M = (Q, \Sigma, \delta, q_0, F)$ where these elements are defined as follows:

thing	what it is
Σ	alphabet as usual
Q	set of states, $\neq \emptyset$
q_0	initial state $\in Q$
F	accept states $\subseteq Q$
δ	transition fct $(Q \times \Sigma) \to Q$

We write that $[q, a, p] \in \delta$ if there is a transition from q to p on the receipt of input a.

The *configuration* of a DFA is a state-string pair (q, w) with $q \in Q$ and $w \in \Sigma^*$. Here, q is the current state and w is the portion of the input string that has yet to be processed. We say that a DFA starts in the configuration (q_0, w_0) where w_0 is the entire string that is to be processed.

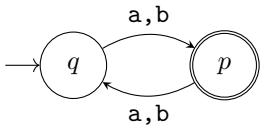
A configuration is *terminal* if it is of the form (t, λ) with $t \in Q$, and is *accepting* if $t \in F$ and otherwise *rejecting*.

We say that $L(M)$ is the set of exactly those strings that DFA M accepts (take M from the initial configuration to an accepting one).

We can write the transition function δ in two different ways. The first is a list of transitions, in a table like so:

	a	b
q	p	p
p	q	q

But we can also write this with a **state transition diagram** as follows:

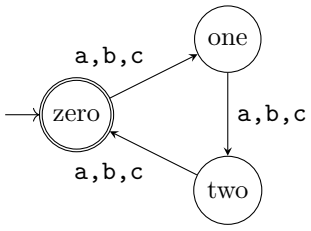


In the above diagram, the initial state is q, as can be seen by the arrow entering q from the margin. There is one accepting state, p, which can be seen by the double circle around p. We can see that on either an a or b, the automaton changes state. Since p (the accepting state) is reached by any odd number of transitions of the automaton, we can see that this automaton accepts only those strings which have an odd number of characters (since it is a DFA, we know implicitly that this automaton's alphabet is $\Sigma = \{a, b\}$).

Recall the previously-stated (but not proven) theorem that there is always a regular expression for the complement of any given regular expression. We begin to lay the groundwork for our proof of this theorem with the fact that the complement of the language defined by a DFA is exceedingly easy to make a DFA for: all we do is change F to be $Q \backslash F$ – that is, we invert the accept states. (**Note that this is *not* the same for NFAs!**)

The intersection of two DFAs is a little harder to implement, but it too can be written as a single DFA. We say that if L_1 is accepted by the usual M_1, and L_2 is accepted by the usual M_2, then their intersection $L_1 \cap L_2$ is accepted by $M = ((Q_1 \times Q_2), \Sigma, \delta, q_0 = (q_1, q_2), (F_1 \times F_2))$. Here, δ contains tuple $[(p, q), a, (s, t)]$ exactly when both $[p, a, s] \in \delta_1$ and $[q, a, t] \in \delta_2$. In essence, we have composite states, each of which is a pair of states, so that we simulate both machines at the same time.

We can also simulate *modular arithmetic* with DFAs. Consider the case in which we wanted to accept only those strings with length divisible by 3. Then, we might have the following:



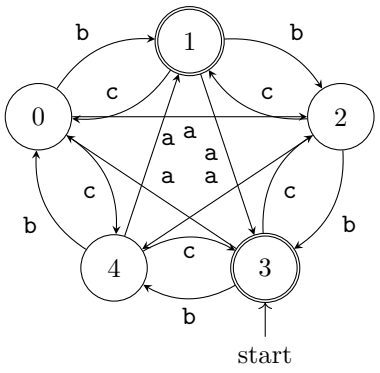
This automaton moves circularly with each transition, and only accepts when the number of transitions is zero mod 3. As such, it only accepts strings with length divisible by 3.

A more complex example is as follows. Construct a DFA to accept the set of strings for which

$$2n_a + n_b - n_c + 3 = \alpha,$$

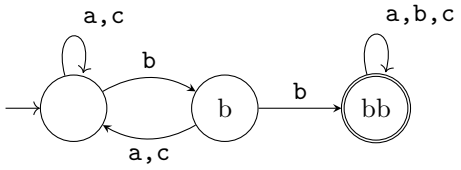
where n_a, n_b, and n_c refer to the number of a's, b's, and c's respectively, and $\alpha \mod 5$ is odd.

We can do this as follows:



Here, the accept states are 1 and 3 because, mod 5, they are the odd remainders. Then, the initial state is 3, because we explicitly add 3 to the end of the expression for α. Finally, the transitions come in the form of arithmetic mod 5 scaled by the coefficients of n_a, n_b, and n_c: we can see that each input of a corresponds to a increase in 2, each input of b corresponds to an increase of 1, and each input of c corresponds to a decrease of 1.

We can, however, do more than arithmetic with DFAs: we can also make DFAs that correspond to the containing of a substring, or having a certain start or end, etcetera. Really, we can do anything with DFAs that we can with regexes and vice versa (we will prove this later with Kleene's Thm.). For example, here is a DFA that accepts strings that contain *bb*:



However, longer substrings require ever more complex DFAs. The DFA for the set of exactly those strings that contain *babbabbba*, for instance, is significantly more difficult to draw, requiring one node for each letter in the string and then an extra one for the start, as well as 10 different transitions back, not all of which are intuitive due to the requirement that we accept overlaps in substring-finding attempts. The idea of a *non-deterministic finite automaton*, or NFA, will help us make this substring graph much easier to draw.

Non-deterministic finite automata

We define a **non-deterministic finite automaton** or NFA as a structure $M = (Q, \Sigma, \delta, q_0, F)$ (much like a DFA) where:

thing	what it is
Σ	same as DFAs: alphabet as usual
Q	same as DFAs: set of states, $\neq \emptyset$
q_0	same as DFAs: initial state $\in Q$
F	same as DFAs: accept states $\subseteq Q$
δ	transition fct $\delta : Q \times (\Sigma \cup \{\lambda\}) \to \mathcal{P}(Q)$

As can be seen above, the only difference lies in the transition

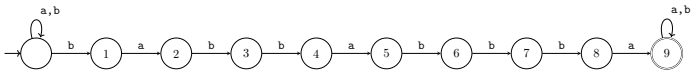
function δ: instead of going from a state-symbol pair to a single state, we can actually go from a state-symbol pair OR a state-λ pair to any number of states in Q. In fact, since $\mathcal{P}(Q)$ includes $\emptyset$, we can even have our transition "die", in which case it does not continue with evaluation. To conceptualize the idea that the automaton is at a set of states rather than a single state, we might say that it clones itself, or is operating in parallel universes, or is extending roots like a tree towards the water that is an accept state.

The aforementioned state-λ pair refers to the λ-transition: that is, the ability of a NFA to move without receiving input. Such transitions are marked on the state transition diagram with λ appropriately.

It should also be noted that the *non-deterministic* part of the NFA plays a big role in its operation. States in NFAs don't need to have a transition on every input in $\Sigma \cup \lambda$ – they might only have one, or even none at all.

An NFA accepts an input string when any one (we can use the existential quantifier $\exists$ here) of its "clones" reaches an accept state at end of the input string w_0.

Back to the substring example from the end of the DFA portion, we can quite easily make such a diagram with NFAs:

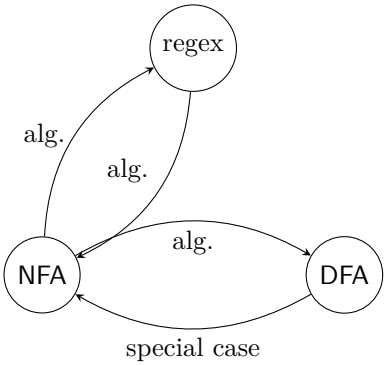


It's a little hard to see, but state (9) is the accept state for the above NFA.

Kleene's theorem

Kleene's theorem states that the classes of languages defined by regexes, DFAs, and NFAs are equivalent, and are called **regular languages**.

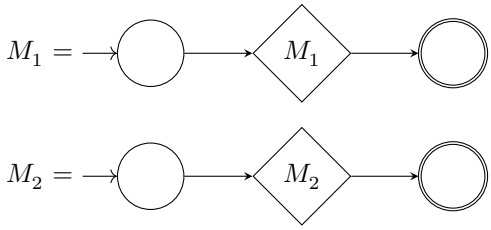
The following diagram illustrates the relationship the three have:



"alg." here refers to the existence of an algorithm. Note that DFAs are a special case of NFAs, with no λ=transitions and determinism required.

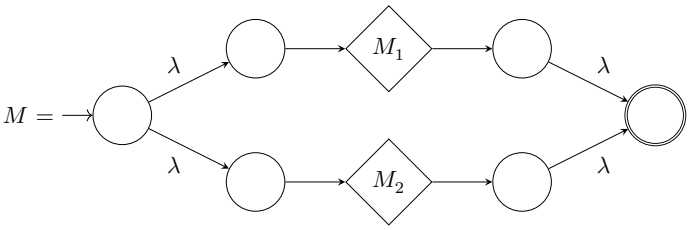
There are three major algorithms insofar as these conversions are concerned, as can be seen by the above diagram's arrows, and one extra algorithm included below for the **application of regular operations to DFAs and NFAs**. We begin with this "extra" algorithm.

For the purposes of the below three regular operation demonstrations, consider finite state automata M_1 and M_2 as defined by the state transition diagrams below:

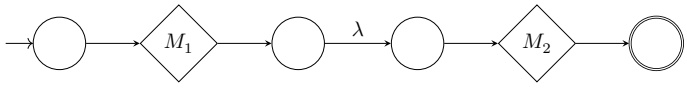


The important takeaway here is that M_1 and M_2 have *zero in-degree* initial states and *zero out-degree* accepting states, which means that they don't have edges going into the initial state (except the one from space), and that they don't have any edges coming out of accepting states. It is important that you keep this in mind, as trying to use the below rules to apply regular operations to automata may result in faulty automata if the input automata are not of this form. (If they *aren't* in this form, then we can put them into this form by adding artificial start and end nodes with λ-transitions.)

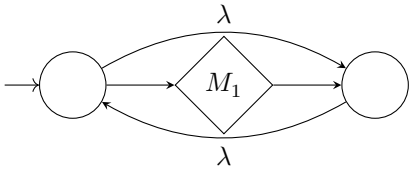
Union of $M_1 \cup M_2$ of M_1 and M_2:



Concatenation $M_1 M_2$ of M_1 and M_2:



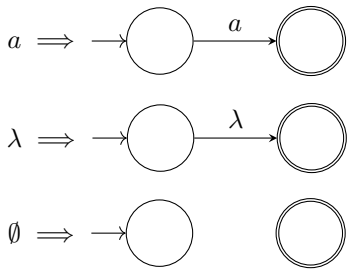
Kleene star M_1^* of M_1:



Now, onto the next algorithm, which is the **conversion of regular expressions into DFAs and NFAs**.

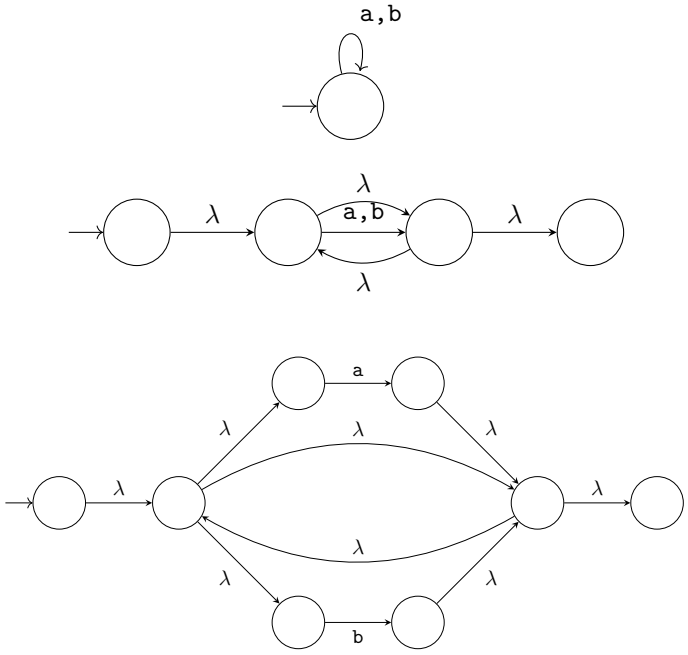
We define the algorithm for conversion of regexes to finite state automata recursively, starting from the zero-operator regular expressions and then building around them.

Recall that the zero-operator regular expressions are the letters in Σ, the empty string λ, and $\emptyset$. They become the following finite state automata:



From these base cases we can construct finite automata for all regular expressions. An example follows.

Example: Three different ways to write $(a \cup b)^*$



The next algorithm is for the **conversion between NFAs and DFAs**. The main idea here is that the states in the created DFA are each a set of states from the NFA (as would be implied by the transition function's codomain of $\mathcal{P}(Q)$). So, we first need to make some kind of table for the δ for the NFA with the states as the sets.

Then, we need to understand the notion of λ-closure. The **λ-closure** of a state x in an NFA, denoted $\mathcal{C}(x)$, is the set of states that are reachable by zero transitions (that is, the reading of the empty string λ) from the state. This becomes a recursive definition when we realize that multiple λ-transitions can be connected to each other.

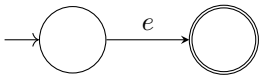
The reason λ-closure is important is that it is used in conjunction with the δ enumeration, used to ensure that we have actually enumerated all of the possible states that are transitioned to.

We can write the DFA transition function as follows, for state x and input a as

$$\delta_{\mathsf{NFA}}(x, a) = \mathcal{C}\left(\bigcup_{p \in x} \delta_{\mathsf{DFA}}(p, a)\right).$$

The last algorithm is for the conversion from **finite automaton to regular expression**. To begin, we need to understand the notion of a **generalized expression graph** (GEG). We define a GEG as a graph (*not* a finite automaton) with arcs labeled by regular expressions. (The definition of a graph is omitted here for brevity, and it is assumed that it has been covered in prerequisite material.) In terms of our previous knowledge, an NFA is a special case of a GEG, where the labels on the arc are either a single symbol or λ.

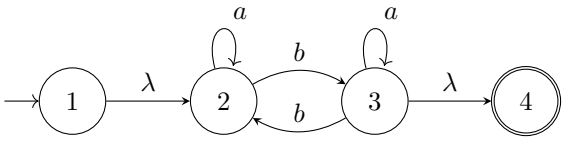
The main idea of this algorithm is to transform the original automaton, through a sequence of equivalent GEGs, until a trivial one is obtained, from which the resulting regular expression will be easily visible. This trivial GEG will be of the following form:



Here, e is our desired regular expression. We want to keep modifying the original NFA until we can get something of the above form, meaning that, for an NFA with k nodes, we'll need to eliminate $k-2$ nodes.

Before starting this algorithm, we *need to make the original automaton compliant*: that is, its initial and accept state need to have zero in- and out-degree respectively.

After we have made the automaton compliant, we need to make an adjacency matrix-like table, with each node on the rows and columns, and the transitions between them in the cells of the matrix. An example follows:



We can see that the above NFA is a compliant automaton accepting strings with an odd number of b's. This automaton can be represented by the following adjacency matrix-like table:

	1	2	3	4
1	$\emptyset$	λ	$\emptyset$	$\emptyset$
2	$\emptyset$	a	b	$\emptyset$
3	$\emptyset$	b	a	λ
4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

We iteratively rewrite this table, until we are left with two nodes. Here is the first iteration:

	1	3	4
1	$\emptyset$	a^*b	$\emptyset$
3	$\emptyset$	$a \cup ba^*b$	λ
4	$\emptyset$	$\emptyset$	$\emptyset$

We are essentially reversing the regular operations from a previous algorithm in this section.

The final iteration:

	1	4
1	$\emptyset$	$a^*b(a \cup ba^*b)^*$
4	$\emptyset$	$\emptyset$

So, we can see that the regular expression for an odd number of b's is $a^*b(a \cup ba^*b)^*$.

Exam on Monday. Good luck!