Automata and Grammars

Carlos Martínez

Abstract

Over the next few days, we'll study two types of objects: finite automata, and formal grammars. They are two sides of the same coin: formal languages (whatever a formal language is, we'll learn in due time). These find applications everywhere: in programming language parsing and compilation, mathematical statements, and even certain aspects of natural, human languages!

Often we care about strings of symbols: whether they are computer programs, or mathematical statements, or just plain old text in natural languages, we need a way to analyze them – in particular, finding ways to distinguish valid strings, sentences (or even sequences of actions!) from invalid ones.

Example 0.1. (stopwatch) You can model the behavior of a stopwatch with two buttons: *reset* and *start/stop*, and three states, as shown in the picture. We can, for example, consider the set of sequences of actions that end with the stopwatch being idle.

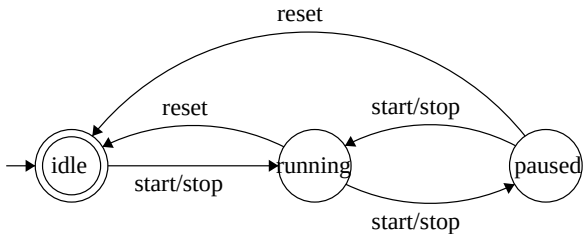


Figure 0.1: A humble stopwatch.

Example 0.2. (arithmetic expressions) An expression like $(2+(3-5))$ makes sense, but an expression like $5+-)($ doesn't really. We can generate valid arithmetic expressions with a *formal grammar*:

$$S \to \text{expr}$$
$$\text{number}_{\text{non-empty}} \to \text{digit number}$$
$$\text{number} \to \text{digit number} \mid \epsilon$$
$$\text{digit} \to 0 \mid 1 \mid 2 \mid \cdots \mid 9$$
$$\text{expr} \to (\text{expr}+\text{expr}) \mid (\text{expr}\cdot\text{expr}) \mid (\text{expr}-\text{expr}) \mid \text{number}_{\text{non-empty}}$$

1. Alphabets and Languages: What's in a Name?

Before we get into the fun stuff, here are some useful definitions just for reference more than anything else.

Definition 1.1. (alphabet) An *alphabet* Σ is any non-empty, finite set of symbols.

Don't be misled by the name – elements of the alphabet can be anything, really: the keywords of your favorite programming language (`if`, `else`, `def`...), mathematical symbols, a set of English words, and of course the letters of the English alphabet.

Definition 1.2. (word) A *word* over the alphabet Σ is a finite string of symbols from Σ. Furthermore, we use Σ^* to denote the set of all words over Σ.

Example 1.3. 11011000001 is a word over $\{0,1\}$, `tacocat` is a word over the English alphabet, and `def class lambda lambda try` is a word over the alphabet of Python keywords (whether it makes sense or not is a different story).

Note that in this definition, nowhere did we specify that the string must be non-empty. In fact, we always consider the *empty string* as a word over our alphabet. It is denoted by ϵ, and it will be quite useful later.

Definition 1.4. (language) A *language* over the alphabet Σ is a subset of Σ^* – that is, a set of words over Σ (this set need not be finite).

Example 1.5. The set of binary representations of the prime numbers is a language over $\{0,1\}$; the set of English words is a language over the English alphabet; and the set of all true mathematical statements is a language over some appropriate set of mathematical symbols.

As it can be seen from the examples, the difficulty of determining whether a word belongs to a language (or generating all the words in a language) can range from the trivial to the utterly impossible.

In what follows, we'll concern ourselves mostly with two questions:

1. How can we test whether a word is an element of a given language? Here, **automata** will come to the rescue.
2. How can we generate words in a given language? Here, **grammars** will come to the rescue.

A final note on notation: we usually denote the length (in symbols) of a word w by $|w|$, and the concatenation of words x and y as xy. Finally, the concatenation of a word with itself n times is denoted just like exponentiation: x^n.

2. Finite Automata

Let's first consider *deterministic* finite automata, or DFAs for short: fun but not-too-powerful machines that we'll later give some upgrades. You can think of a DFA as a machine to which you progressively input symbols from your alphabet. As you do this, the DFA will be changing its *state* (it will always begin in a particular state called the *initial* state). Every so often, the DFA will arrive at a special type of state called *accepting*, in which case we'll consider the word consisting of all the symbols we've put in so far as being *accepted* by the DFA.

A DFA is called *deterministic* because if you're in a given state, the symbol you put in at that moment completely determines which state you must go to next (this is called a *transition* from a state to another).

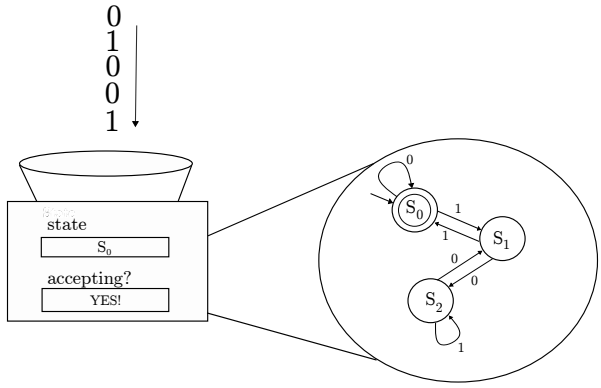


Figure 2.1: In this example, we have put in the symbols $1, 0, 0, 1, 0$, in this order. After doing this, we reach the accepting state S_0, so we say that this DFA *accepts* the word 10010. Fun fact: this DFA accepts a binary string if and only if the string represents a multiple of 3. Why?

There are two common ways to specify a DFA: we either make a *transition table* consisting of all possible pairs of states and alphabet symbols, or *state diagrams*. Tables are no fun, so we'll stick to state diagrams.

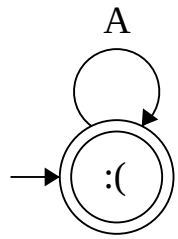


Figure 2.2: Some DFAs are conscious: this one can't *accept* the awful reality of having no free will (being *deterministic* and all), so it can only scream.

Definition 2.1. (state diagram) A *state diagram* is a labeled directed graph where the vertices correspond to states and the directed edges (labelled by symbols from the alphabet) correspond to transitions.

Remark 2.2. The initial state is indicated by having an incoming edge which doesn't come from any other vertex (see, for example, Figures 2.1 and 2.2). Accepting states are indicated by drawing an additional smaller concentric circle.

2.1. (Exercise) Explain the joke in Figure 2.2 (besides the two puns, of course).

2.2. (Challenge) Can you provide a formal definition of what a DFA is? (not too difficult) What about the notion of a word being accepted by a DFA? (somewhat harder)

Since we're doing mathematics and not philosophy, let's provide some more formal definitions for all of this.

Definition 2.3. (DFA) A *deterministic finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

1. Q is the set of states.
2. Σ is our alphabet of choice.
3. $\delta : Q \times \Sigma \to Q$ is the *transition function*, which specifies which state to go next given the current state and the current symbol we've put in the DFA.
4. $q_0 \in Q$ is the initial state.
5. $F \subseteq Q$ is the set of accepting/final states.

Note that our current definition is a bit limited: it doesn't actually tell us anything about what to do with words, only individual symbols. To deal with words, we need the *extended transition function*, which allows us to "chain" transitions:

Definition 2.4. (extended transition function) Given that we have already defined a DFA as in the previous definition, the *extended transition function* is a function $\delta^* : Q \times \Sigma^* \to Q$ defined recursively as follows:

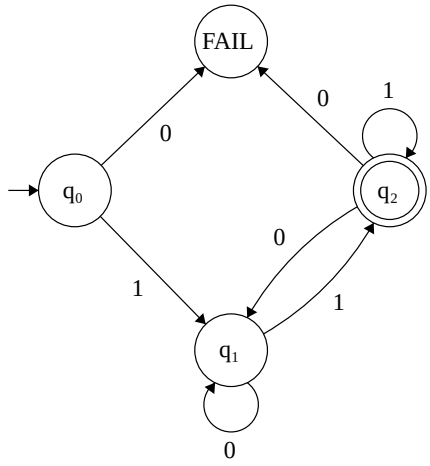
1. For all states $q \in Q$, $\delta^*(q, \epsilon) := q$.
2. For all words $w \in \Sigma^*$, symbols $x \in \Sigma$, and states $q \in Q$, $\delta^*(q, wx) := \delta(\delta^*(q, w), x)$.

In essence, what this definition is telling us is that $\delta^*(q, w)$ is the state we'll end up at if we start at state q, and process all of the letters in w in order.

Using this, we can now say that a word $w \in \Sigma^*$ is accepted by the given DFA if and only if $\delta^*(q_0, w) \in F$ (why?) and finally, we can define the *language* of a DFA A as follows:

$$L := \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$$

2.3. (**Exercise**) Give an interpretation of the language of the following DFA.



2.4. (Problem) Suppose you're given a fixed positive integer n and a fixed base b. How would you construct a DFA whose language is precisely the set of all strings in base b that represent an integer divisible by n?

2.5. (Exercise – probably slightly time consuming) If you happen to know the rules of tennis, can you make a DFA that models a game of tennis? (Note: I do not remember the rules of tennis, but back when I did, it was a fun exercise).

2.6. (Challenge) Suppose you have DFAs A and B. How would you make a DFA that accepts precisely the words accepted by both A and B?

It turns out that languages that can be accepted by a DFA are quite special, so we call them *regular* languages (spoiler: the *regular* in "regex" and the *regular* here are related), and we can ask many questions about them: are there more compact representations of regular languages? Under what operations are they closed? In particular, Challenge 2.6 is asking whether regular languages are closed under intersection. To answer this question, we'll do some dark magic and multiply DFAs.

3. The Product Automaton

Suppose that you have DFAs A and B with the same alphabet Σ, which accept the regular languages $L(A)$ and $L(B)$. To ask whether the language $L(A) \cap L(B)$ is regular or not is to ask whether it is possible to make a DFA that accepts precisely those words that are accepted by both A and B.

In essence, we want to simulate two DFAs A and B using one DFA. If Q_A and Q_B are the set of states of A and B, respectively, then one thing we can do is to make the set of states of our new DFA the *the set of all possible pairs of states of A and B*, also known as $Q_A \times Q_B$. Then, if we are in a particular pair of states and put in some symbol $s \in \Sigma$, we'll move to the pair consisting of having put in s in A and B separately. Finally, we make our accepting states to be precisely the set of pairs where each component is an accepting state. Or, more formally:

Definition 3.1. (product automaton of A and B) Given that $A = (Q_A, \Sigma, \delta_A, q_{0A}, F_A)$ and $B = (Q_B, \Sigma, \delta_B, q_{0B}, F_B)$, then we define the product automaton of A and B as follows:

$$A \times B = (Q_A \times Q_B, \Sigma, \delta, (q_{0A}, q_{0B}), F_A \times F_B)$$

where $\delta : (Q_A \times Q_B) \times \Sigma \to (Q_A \times Q_B)$ is a function such that for all $x \in \Sigma$ and pairs $(p, q) \in Q_A \times Q_B$,

$$\delta((p, q), x) := (\delta_A(p, x), \delta_B(q, x))$$

For some intuition on what this means, see Figure 3.1 (or, you know, ask me).

3.1. (Exercise) How would you modify the definition of the product automaton to prove that if R and S are regular languages over some alphabet Σ, then $R \cup S$ is also regular?

3.2. (Problem) If you're not convinced, you can try proving formally that the language accepted by $A \times B$ truly is $L(A) \cap L(B)$.

3.3. (Exercise) Suppose that $\Sigma = \{a, b\}$. Prove that the set of words over Σ that begin and end with the same symbol is regular.

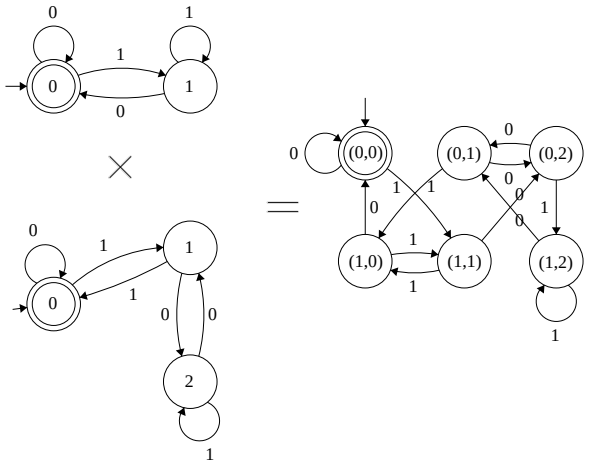


Figure 3.1: An example of how you would take the product of two DFAs. Can you figure out what are the languages they accept?

4. The Pumping Lemma

We've finally reached fancy-sounding territory, and, quite unfortunately, the limits of the power of regular languages.

You may have noticed early on that DFAs have very, *very* poor memory – in fact, since they can't remember the past, all they can really store must be stored in the states themselves – which is a huge constraint given that the number of states is finite. The Pumping Lemma will allow us to formalize this idea, and also let us prove that some languages are *not* regular.

The main idea is this: given that you only have a finite number of states, if you put in a sufficiently large number of symbols, the DFA must reach a state it had reached before – this sounds suspiciously like the Pigeonhole Principle, if you ask me.

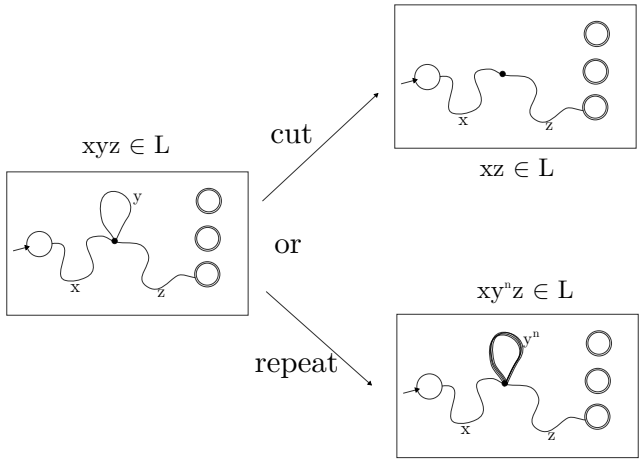


Figure 4.1: A visual proof of the Pumping Lemma.

The Pumping Lemma is one of the few theorems I know where remembering the proof is actually easier than remembering the statement, so we'll give a sketch of the proof first and only then we'll state the theorem:

Theorem 4.1. (The Pumping Lemma)

Proof. Suppose that we have a regular language L. Then, for some DFA A, we'll have that $L = L(A)$, and now, let $N \in \mathbb{N}$ be the number of states of A. If there exists a word $w \in L$ of length at least N, when inserting the symbols of the word, we would have traversed at least $N+1$ states (including the starting state), and would have ended at some accepting state.

Then, by the Pigeonhole Principle, we must have visited some state twice; in particular, the first such loop must have ended at most N symbols from the beginning. So now, we can divide our path (that is, the word w) from the starting state to the last visited state into three words: x, the word inserted up to the beginning of the loop, y, consisting of the loop itself, and z, the word inserted from the end of the loop to the last state, so that $w = xyz$.

But we can just remove the loop and still reach the same last (accepting) state, meaning that $xz \in L$; and more strikingly, we can instead repeat the loop *as many times as we want* and still get a path to the last state: meaning that for all non-negative integers n, $xy^n z \in L$! □

From this proof we can easily extract the actual statement of the theorem:

The Pumping Lemma. If L is regular, then there exists some $N \in \mathbb{N}$ such that for all $w \in L$ such that $|w| \geq N$, we can break w into three words, $w = xyz$, (not necessarily in the language), such that:

- y is not the empty word.
- $|xy| \leq N$.
- For all non-negative integers n, $xy^n z \in L$.

Unfortunately, this doesn't yet allow us to fully capture the idea of regularity, since this is only a necessary, but not sufficient, condition; but it's still quite useful.

4.1. (Problem) Let L be a regular language, and let N be the constant given by the Pumping Lemma. Prove that L contains an infinite number of words if and only if there exists a word in L with length at least N and at most $2N-1$.

4.2. (Exercise) Suppose that $\Sigma = \{0,1\}$. Prove that the language consisting of all words with the same number of 0s and 1s is *not* regular.

4.3. (Exercise) Suppose that $\Sigma = \{1\}$. Prove that the language consisting of all words with a prime number of 1s is *not* regular.

5. Nondeterminism

Given that the number of states in the product of two DFAs is the product of their number of states (why?), just taking the product haphazardly can blow up the size of the DFA – so we may want a more compact representation. I propose that we make our automaton capable of, when inserting a symbol, "choosing" non-deterministically the next state among a set of possible next states – and even further: letting the automaton jump to another state, even when no symbol has been inserted! To make this clear, what I'm proposing is this:

- Previously, for a given symbol $s \in \Sigma$, we only allowed a single transition labelled with this symbol to come out of any given state. Now, we're allowing any number of transitions labelled with s to come out of a state.
- We allow for ϵ-transitions – i.e. transitions without having to insert any symbol.
- Now, we say $w \in \Sigma^*$ is accepted by the automaton if, among all the possible paths the automaton could have taken while reading w, there was at least one that ended at an accepting state.

This may still not be particularly clear. More formally, what we are trying to do is tweaking the definition of DFA slightly, so that now the transition function δ, instead of being of type $Q \times \Sigma \to Q$, will now be of type $Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$; that is, δ will now map pairs of states and symbols (or ϵ) to *sets* of states. We'll modify the extended transition function δ^* accordingly so that, having inserted a word $w \in \Sigma^*$, it will give us a set of states the automaton could be in.

Definition 5.1. (ϵ-NFA) A *non-deterministic finite automaton with ϵ-transitions* is an ordered 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

1. Q is the set of states.
2. Σ is our alphabet.
3. $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ is our new transition function – note that we now accept ϵ as an argument!
4. q_0 is the starting state.
5. $F \subseteq Q$ is the set of accepting states.

Just like in the case for DFAs, given a state $q \in Q$ and a word $w \in \Sigma^*$, we want $\delta^*(q, w)$ to represent the set of all states the automaton could be in, starting from q and after inserting w. Now, recall that we talked of ϵ-transitions as those that are allowed to take place even when no symbol has been inserted.

This means that at any given point in time, if a particular state $r \in Q$ is in the set of states the automaton could be in, then *all the other states than can be reached from r through ϵ-transitions must be in the set.* This idea can be formalized through the concept of ϵ-closure:

Definition 5.2. (ϵ-closure) Given a set of states $S \subseteq Q$, we define the set of states $\epsilon\text{−CLOSE}(S)$ recursively, as follows:

- $S \subseteq \epsilon\text{−CLOSE}(S)$
- If $p \in \epsilon\text{−CLOSE}(S)$ and $\delta(p, \epsilon) = r$, then $r \in \epsilon\text{−CLOSE}(S)$.

Using this, we can finally provide a formal definition for the new extended transition function:

Definition 5.3. (extended transition function for ϵ-NFA) We define $\delta^* : Q \times \Sigma^* \to \mathcal{P}(S)$ recursively, as follows:

- For all states $q \in Q$, $\delta^*(q, \epsilon) := \epsilon\text{−CLOSE}(\{q\})$.

- For all states $q \in Q$, words $w \in \Sigma^*$ and symbols $x \in \Sigma$,

$$\delta^*(q, wx) := \epsilon\text{−CLOSE}\left(\bigcup_{s \in \delta^*(q,w)} \delta(s, x)\right)$$

And given this definition, we can finally say that the automaton accepts a word $w \in \Sigma^*$ precisely when $\delta^*(q_0, w) \cap F \neq \emptyset$.

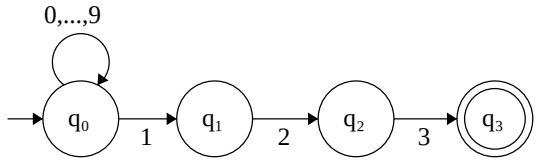


Figure 5.1: What does this do?

We first make the following observation: given a DFA $A = (Q, \Sigma, \delta, q_0, F)$ that accepts a language L, we can easily construct an ϵ-NFA N that accepts the same language L by defining $N = (Q, \Sigma, \delta_N, q_0, F)$, where $\delta_N(q, x) = \{\delta(q, x)\}$ for all states $q \in Q$ and symbols $x \in \Sigma$.

But what about the other way around: it sounds much more difficult, because at least intuitively, an ϵ-NFA feels way more powerful than a DFA.

WRONG! Remarkably, given any language L accepted by some ϵ-NFA, we can make a DFA that accepts precisely the same language – meaning that *all languages accepted by an ϵ-NFA are just regular.*

To convert an ϵ-NFA into a DFA, we perform what's called the *subset construction* – the main idea is to let the set of states be the set of all subsets of states in the ϵ-NFA; then, after introducing a symbol, the DFA will transition to the state that represents the set of states the ϵ-NFA could be in.

Definition 5.4. (The subset construction) Let $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ be an ϵ-NFA. Then, let $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ be a DFA such that:

- $Q_D = \{S \subseteq Q_N | S = \epsilon\text{−CLOSE}(S)\}$ (that is, states in D will just be sets of states in N that are closed under ϵ-transitions).
- $\delta_D(S, a) = \epsilon\text{−CLOSE}(\bigcup_{s \in S} \delta(s, a))$ (notice we're still making sure that the resulting set is ϵ-closed).
- $F_D = \{S \in Q_D | S \cap F_N \neq \emptyset\}$

5.1. (Problem – very optional!) Prove that in fact, the DFA D defined above accepts the same language as the ϵ-NFA N.

Now, if you have an NFA without any ϵ-transitions with N states, the resulting DFA from the subset construction will have 2^N states, which seems like too much. In some cases, you might be able to construct smaller DFAs – unfortunately, sometimes it's the best we can do.

5.2. (Problem) Can you construct an ϵ-NFA with $\Theta(N)$ states such that the smallest DFA that accepts the same language has $\Theta(2^N)$ states? (Hint: there's no need for ϵ-transitions).

5.3. (Problem) If L is a regular language and L^R is the language consisting of all the words of L reversed, prove that L^R is regular as well.

6. Formal Grammars

Now, we'll see a different way to look at languages – one that will allow us to classify them beyond just regular languages (and we'll find out that regular languages are actually the simplest in the whole hierarchy).

The idea of a grammar is allowing you to generate words for a language. To do this, you'll begin with a *starting symbol* and apply a set of production (read: substitution) rules until a word is generated. More formally:

Definition 6.1. (formal grammar) A 5-tuple $G = (V, T, \mathcal{P}, S)$, where:

- V is a finite nonempty set of **variables** – these are symbols that are meant to be substituted for something else.
- T is a finite set of **terminal symbols** (which must not contain any variables) – these are symbols that can't be substituted anymore, meaning they're meant to be symbols in the generated word at the end.
- $S \in V$ is the **starting symbol**, from which we start applying the production rules.
- $\mathcal{P}$ is a finite set of **production rules** of the form $\alpha \to \beta$; these will form the bulk of our grammar. The left side is called the **head**, and it is a non-empty string of variables and terminals (which must contain at least one variable), while the right side is called the **body**, and it can be any string of variables and terminals. It can even be the empty string ϵ!

Given this, we say $\gamma \Rightarrow_G \delta$ (that is, δ *one-step derives* from γ) if we can obtain δ from γ by applying exactly one production rule from G, and we say that $\gamma \Rightarrow_G^* \delta$ if we can obtain δ from γ through a chain of one-step derivations. Then, the *language generated by* G is just the set of words consisting only of terminals that can be derived from S:

$$L(G) := \{w \in T^* | S \Rightarrow_G w\}$$

Example 6.2. We can make a grammar for the language of binary palindromes as follows:

$$S \to 1S1$$
$$S \to 0S0$$
$$S \to 0$$
$$S \to 1$$
$$S \to \epsilon$$

6.1. (Problem) If $\Sigma = \{a, b\}$, make a grammar that generates

$$L = \{w \in \Sigma^* | \text{the number of } b\text{'s in } w \text{ is divisible by } 3\}$$

6.2. (Problem – more challenging) If $\Sigma = \{a, b\}$, make a grammar that generates

$$L = \{ww | w \in \Sigma^*\}$$

(that is, the set of words that consist of two words repeating themselves).

We can classify a language depending on the grammar that generates it. One very useful hierarchy is the Chomsky hierarchy of languages:

a. **Type 0 (general):** Rules of the form $\alpha_1 A \alpha_2 \to \beta$, with α_1, α_2 being string, and A a variable.

b. **Type 1 (context-sensitive):** Rules of the form $\alpha_1 A \alpha_2 \to \alpha_1 \gamma \alpha_2$, with α_1, α_2 being any string, A a variable, and γ a non-empty string.

c. **Type 2 (context-free):** Rules of the form $A \to \beta$, with A a variable, and β being any string.

d. **Type 3 (regular):** Rules of the form $A \to \omega$ or $A \to \omega B$, where $\omega \in T^*$ and B is a variable.

It turns out, we're already very familiar with the languages generated by regular grammars:

Theorem 6.3. A language L is regular if and only if there exists a regular grammar that generates it.

Proof. If L is regular, take any DFA that accepts it, with Q as its set of states. The grammar will have Q as its set of variables, and we make the starting state the starting variable. Then, for every transition $\delta(P, s) = Q$, we add a rule $P \to sQ$, and for all accepting states P, we add a rule $P \to \epsilon$.

For the other direction, suppose we have a regular grammar that generates a language M. First, for any rule of the form $A \to \omega$ or $A \to \omega B$ with ω of length $n \geq 2$, such that $\omega = s_1 s_2 \cdots s_n$, we 'split' it into rules $A \to s_1 A_1$, $A_1 \to s_2 A_2$, ..., $A_{n-1} \to s_n A_n$, and either $A_n \to \epsilon$ or $A_n \to B$. Furthermore, any rule of the form $A \to s$ (for $s \in \Sigma$) is split into $A \to sA'$, $A' \to \epsilon$. From this modified grammar, we construct an ϵ-NFA with the variables becoming states:

- Rules of the form $A \to B$ for $A, B \in \Sigma$ becomes an ϵ-transition from A to B.
- Rules of the form $A \to \epsilon$ for $A \in \Sigma$ mean that A is an accepting state of the ϵ-NFA.
- Rules of the form $A \to sB$ for $A, B \in \Sigma$ mean that there is an s-transition from A to B.
- The starting variable becomes the starting state.

□

6.3. (Problem) Can you create a context-free grammar for a small subset of the C language?

7. Context-Free Grammars

By far the most useful type of grammar in the Chomsky hierarchy are context-free grammars, in part because they allow you to define the syntax of programming languages.

Recall that a context-free grammar has rules of the form $A \to \beta$, where A is a variable and β is any string. Without modifying the grammar, sometimes it can be very difficult to determine if a particular word can be generated from a particular context-free grammar – but this can be fixed by converting the grammar into an equivalent grammar in a special form called the *Chomsky normal form*.

Theorem 7.1. (Chomsky Normal Form) Given any context-free grammar G, we can make a grammar C that generates the same language, such that all the rules of C are of one of these forms:

- $A \to BC$, where A, B and C are variables (but B and C can't be the starting variable).
- $A \to s$, where A is variable and s a terminal.
- $S \to \epsilon$, where S is the starting variable.

Proof. **(Sketch)** We first remove rules of the form $A \to \epsilon$ (except for possibly $S \to \epsilon$) by removing all possible subsets of occurrences of A in each rule (this sounds a bit too convoluted – we'll see an example in the lecture). Then, we remove all of the rules of the form $A \to B$ (where B is a variable) (called unit productions) by adding $A \to w$, for all words w that can be reached by following unit productions. Finally, we decompose all the rules of the form $A \to s_1 s_2 \cdots s_n$ into $A \to s_1 A_1$, $A_1 \to s_2 A_2$, ... making sure that s_i is not a terminal by adding unit productions of terminals. □