Automata and Formal Languages

Yoav Rodeh

Jerusalem College of Engineering

Generally something about: What is computation? that is what we try to answer.

1 Regular Expresssions and DFA's

1.1 Languages

- A language is a set of *words*.
- A word is a string of *letters*.
- Letters are taken from alphabet Σ.

Example 1. $L_1 = \{a, ab\}$, $L_2 = \{aaa, \epsilon\}$

Some special symbols:

- the empty word is marked ϵ
- $\Sigma^\star$ is the language containing all words over Σ
- The empty language is marked $\emptyset$

Operations over languages:

- All set operations : intersection, union, negation, etc.
- concatenation: $L_1L_2 = \{uv \mid u \in L_1, v \in L_2\}$ Example: $L_1 = \{a, aaa\}$, $L_2 = \{bb, bbb\}$.
- Kleene closure : $L^\star = \{u_1 \dots u_n \mid n \geq 0, \forall i, u_i \in L\}$ Example: $L = \{aa, b\}$.

1.2 Regular Expressions

Definition 1. *A regular expression can use:*

1. *letters from Σ*
2. *ϵ, $\emptyset$*
3. *$\cdot, \cup, \star$*

A note: many texts use $+$ and not $\cup$.

Example 2. 1. $(a \cup bb)^\star \cup abb$ - just for show, really explain this one last.
2. a
3. ab
4. $(a \cup b)^\star$. We many times write Σ instead of $a \cup b$.
5. $(a^\star(a \cup b)) \cup aaa(a \cup b)$
6. $(a^\star b)^\star = \{\epsilon\} \cup \Sigma^\star b$

Example 3. Find regular expressions describing the language of all words:

1. ending with a
2. starting with a, having ab somewhere in the middle.
3. starting with a and ending with b or the other way around.

1.3 Deterministic Finite Automata - DFA

Our purpose is to formulate the simplest mathematical model for a computer. This is our first attempt. Explain the name (and skip "deterministic").

To make our lives simple we think of a box. It has a green light, and a red light. It gets letter by letter as input until the word is finished and then one of the lights turns.

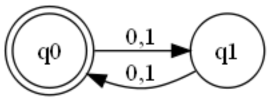
For example... show some words that have only a and c - they turn green, and some that have also b they give red. So this box checks if a word does not have a b. Of course *green* is good.

We can think of machines that do complicated stuff, like checking if a id number is valid (that is what the last digit is for). Checking if a credit card number is valid. Checking if a number is a prime, etc.

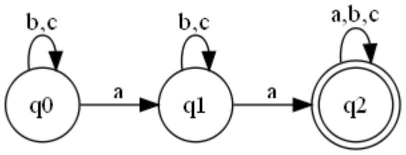
This is the general idea, but DFA's have rules of how they look inside:

Example 4. For each one also show the regular expression.

1. All words that start with *abb*. Don't forget to add the rejecting state.
2. All words of even length.



3. all words with at least 2 a's:



4. all words containing *abab*.

In last two examples, explain the idea that automata states are really states of the computation. they are our memory.

Formally (show everything on last example):

Definition 2. *a DFA is :* $\langle \Sigma, Q, q_0, F, \delta \rangle$

- Q *is the set of states.*
- $q_0 \in Q$ *is the initial state.*
- $F \subseteq Q$ *is the set of accepting states.*
- $\delta : Q \times \Sigma \to Q$ *is the transition function.*

1.4 $\delta^\star$

Here is already the second week...

Definition 3. $\delta^\star : Q \times \Sigma^\star \to Q$,

- $\delta^\star(q, \epsilon) = q$
- $\delta^\star(q, a \cdot w) = \delta^\star(\delta(q, a), w)$

We show some simple example from previous automata.

Definition 4. *The language accepted by automata A is:* $L(A) = \{w \in \Sigma^\star \mid \delta^\star(q_0, w) \in F\}$

1.5 regular languages

Are those that are accepted by DFA's.

1. The empty language.
2. one word languages. by induction on the length of the word. for empty language is easy, then its easy to extend.

1.6 Exercises

Construct automata for, $\Sigma = \{a, b\}$:

1. words of length a multiple of 3.
2. build an automata where the difference between number of a's and number of b's is a multiple of 3. Two solutions.
   (a) count $+1$ for a and -1 for b, accepting on 0.
   (b) Two different paths, one if we started counting a's, one if we started with b's.
3. each a is immediately preceded by b. This means that there are no two a's in a row, and a is not first.
4. $abab$ is a substring.
5. neither aa nor bb as sub-strings. So we should be alternating constantly.
6. odd number of a's and even number of b's 4 states - think of multiplication automata.
7. both ab and ba as sub-strings. All this means is that we should alternate twice. $a^+b^+a^+ \cup b^+a^+b^+$

Build automata for:

1. All words containing 3 0's in a row.
2. Not containing 3 0's in a row (replace accepting and non accepting states)
3. Where number of $1, 2$ is a multiple of 3 (0 is also in Σ).
4. $\Sigma = \{0, \ldots, 9\}$. All words representing valid numbers that are divisible by 3. (remember to handle a first 0 separately). Otherwise have 3 states representing the 3 moduli.
5. A can machine, with 2 coins 1,5, and which needs 6 shekels and returns change.

2 Non Deterministic Finite Automata - NFA

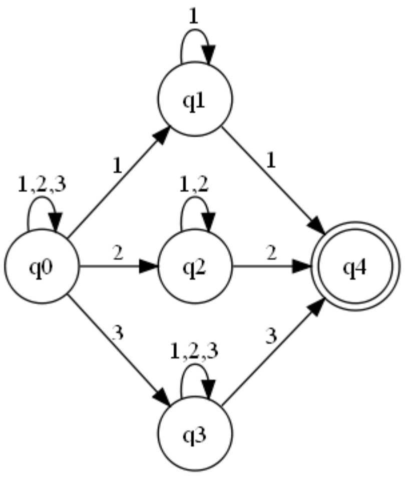
2.1 Reminder

Build an automata for the regular expression: $(a \cup abb)^\star b$ - its not a simple example and later with NFA's it will be easy.

These automata have the power of guessing. If there is a way to reach the target then they accpet the word.

Example 5.

1. $L = \Sigma^{\star}111$
2. $L = 1\Sigma^{\star}0$
3. $L = a^{\star}(ab)^{\star}a$ Notice how we don't accept wrong words.
4. $\Sigma = \{1, 2, 3\}$ L is the set of words where the last number appears previously somewhere without any higher number between.



2.2 definition

We make δ into $\delta \subseteq Q \times \Sigma \rightarrow P(Q)$.

1. We can to move from a state to a set of states.
2. We can suddenly not have any state to move to.

Definition 5. *an NFA is :* $\langle \Sigma, Q, q_0, F, \delta \rangle$,

- Q *is the set of states.*
- $q_0 \in Q$ *is the initial state.*
- $F \subseteq Q$ *is the set of accepting states.*
- $\delta : Q \times \Sigma \rightarrow P(Q)$ *is the transition function.*

A note: We will many times extend functions that work on states to work on sets of states, In a natural way. For example for a set of states $A \subseteq Q$, we say: $\delta(A, \sigma) = \cup_{q \in Q} \delta(q, \sigma)$

Definition 6. *transitive closure of* δ*:*

1. $\delta^{\star}(q, \epsilon) = \{q\}$
2. $\delta^{\star}(q, aw) = \delta^{\star}(\delta(q, a), w)$

Note the shortcut writing in the definition. expand it.

Definition 7. *The language accepted by NFA A, is*

$$L(A) = \{w \in \Sigma^{\star} \mid \delta^{\star}(q_0, w) \cap F \neq \emptyset\}$$

Show language on examples above. We can explain NFA's in several ways. I like most the idea that it can guess the correct path to go. We can say it tests all possibilities. Another good one to use is that it all the time works with a subset.

2.3 ϵ-NFA

Can make life easier.

Example 6. 1. $a^\star b^\star c^\star d^\star$.
2. $(ab)^\star(ba)^\star \cup aa^*$
3. $((ab \cup aab)^\star a^\star)^\star$
4. $((a^\star b^\star a^\star)^\star b)^\star$
5. $(ba \cup b)^\star \cup (bb \cup a)^\star$

I don't teach these formalities for ϵ-automata:

We change δ, $\delta : Q \times (\Sigma \cup \epsilon) \to P(Q)$.
Definition 8. *Closure of a state q: $CL^\epsilon(q)$ is the set of states reachable by ϵ moves only. We extend this definition to set as usual : $CL^\epsilon(R) = \cup_{q\in R} CL^\epsilon(q)$ Show an example.*

$\delta^\star$ changes:

1. $\delta^\star(q,\epsilon) = CL^\epsilon(q)$
2. $\delta^\star(q,\sigma w) = CL^\epsilon(\delta^\star(\delta(CL^\epsilon(p),\sigma),w))$

Notice we use our standard extension even while defining $\delta^\star$.

Definition 9. *For ϵ-NFA A, we define $L(A) = \{w \in \Sigma^\star \mid \delta^\star(q_0,w)) \cap F \neq \emptyset\}$*

Show this on an example with no loops and not too many ϵ in a row.

3 Relationship Between the Different Automata

We now have 3 different kinds of automata. how do they relate to each other?

3.1 ϵ moves are unnecessary

Theorem 1. *If L has an ϵ-NFA, then it is has an NFA*

Proof. Work constantly with an example: $a^\star b^\star c^\star d^\star$ We show the construction of NFA from given ϵ-NFA:

1. $Q_{new} = Q$
2. $q_{0new} = q_0$
3. Each state that can reach a final state with ϵ moves, make it into a final state. Notice this does not change the language accepted by the NFA.
4. For each path of the form $q \xrightarrow{\epsilon} q_1 \xrightarrow{\epsilon} q_2 \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} q_n \xrightarrow{\sigma} p$, we add an arrow: $q \xrightarrow{\sigma} p$.
   Again, this step does not change the language. We added no new words.
5. We remove all ϵ moves. This step is also fine, because any word accepted before can also be accepted now with the new edges (explain).

□

Example 7. $(ab)^\star(ba)^\star \cup aa^*$

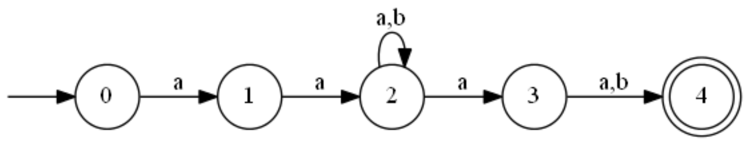
There are alternative constructions - that also "short-cut" paths with ϵ after the σ, and then one doesn't need the first step.

Note that the NFA we got has the same number of states as the original automata. So that basically these two models are the same.

3.2 NFA's are equivalent to DFA's

The idea is that a run of an NFA can be thought of as having states. Each state is a subset of the states of the NFA. Then we are deterministic.

Example 8. Write the NFA for $aa(a \cup b)^\star a(a \cup b)$, and first show a run of a word in terms of sets of states:



And an example run of the word *aaababaa*:

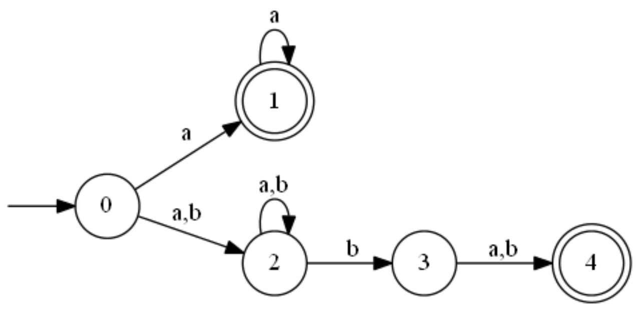
$$\{0\} \xrightarrow{a} \{1\} \xrightarrow{a} \{2\} \xrightarrow{a} \{2,3\} \xrightarrow{b} \{2,4\} \xrightarrow{a} \cdots$$

Now build the actual DFA, starting from the initial state and going on.

Formally, For NFA $N = \langle \Sigma, Q_N, q_{N0}, F_N, \delta_N \rangle$, we construct a DFA $D = \langle \Sigma, Q_D, q_{D0}, F_D, \delta_D \rangle$, s.t. $L(N) = L(D)$.

- $Q_D = P(Q_N)$
- $q_{D0} = \{q_{N0}\}$
- $\delta_D(A, \sigma) = \bigcup_{q \in A} \delta_N(q, \sigma)$
- $F = \{A \subseteq Q_N \mid A \cap F_N \neq \emptyset\}$

If we want we can have a real algorithm for it. Let's look at the NFA:



qs	a	b
0	1, 2	2
1	1	
2	2	2, 3
3	4	4
4		
1, 2	1, 2	2, 3
2, 3	2, 4	2, 3, 4
2, 4	2	2, 3
2, 3, 4	2, 4	2, 3, 4

And of course mark the final states.

3.2.1 I don't really show this exact proof

Lemma 1. $\delta^\star_D(A, w) = \delta^\star_N(A, w)$

Proof. By induction on w. If $w = \epsilon$, we get both sides are A. If $w = \sigma v$, then:

$$\delta^\star_D(A, \sigma v) = \delta^\star_D(\delta_D(A, \sigma), v) = \delta^\star_N(\delta_D(A, \sigma), v) = \delta^\star_N(\delta_N(A, \sigma), v) = \delta^\star_N(A, \sigma v)$$

□

If $w \in L(N)$, then $\delta^\star_N(q_0, w) \cap F_N \neq \emptyset$, so $\delta^\star_D(q_0, w) \cap F_N \neq \emptyset$, meaning its an accepting state. This goes the other way in the same way.

3.3 Some NFA's give exponential sized DFA's

$L = \Sigma^\star a \Sigma^{n-1}$. NFA is very simple (guesses when to step out of initial state with an a and then goes $n - 1$ steps to the accepting state.

Claim. Any DFA accepting L has at least 2^n states.

Proof. Assume it does. From Σ^n there are some u, v s.t. $\delta^\star(q_0, u) = \delta^\star(q_0, v)$ - because number of states is smaller than $|\Sigma^n|$. Take a place u and v differ. w.l.o.g., assume $u = u_1 a u_2$, $v = v_1 b v_2$. examine $ub^{(n-1)-|u_2|}$ and $vb^{(n-1)-|u_2|}$, they will both be accepted, but only the u extension should be. □

4 Regular language closure

What is closed? like addition and multiplication over the natural numbers. division? no. what does it mean in our case?

Claim. Regular languages are closed under union. put in a different way: If L_1 is regular, and L_2 is regular, then so is $L_1 \cup L_2$

show the construction. notice the size of the resulting automata and that it is non-deterministic.

Claim. Regular languages are closed under complementation.

Show an example: $L = a^\star$, $L^c = \Sigma^\star b \Sigma^\star$. Construction takes a DFA for reverses accepting states. show it on example. Would this construction work for NFA's? explain why not.

Claim. Regular languages are closed under intersection

One way is to use De-Morgen combined with previous results. However this will cause a potentially exponential blow-up, because last complementation will require a move to a DFA.

4.1 product automata

Another way to prove this claim, is by construction of a *product automata*: First show example on these automata: $(0^\star 10^\star 10^\star 10^\star)^\star$ and $\Sigma^\star 1 \Sigma^\star$. Then formally:

- $Q = Q_A \times Q_B$
- $q_0 = Q_{A0} \times Q_{B0}$
- $\delta((q_A, q_B), \sigma) = (\delta(q_A, \sigma), \delta(q_B, \sigma))$
- $F = F_A \times F_B$

What is the size of the result?

4.2 other basic results

Claim. Every finite language is regular.

Proof. Easy proof - by induction using our theorem. we can also build a tree shaped automata. □

Claim. regular languages are closed under concatenation.

Simple construction.

Claim. regular languages are closed under Kleene star.

Series of attempts:

1. add ϵ moves from the final states to the initial state. Then empty word is not accepted.
2. so, mark the initial state as final. Doesn't work because initial state may actually act as an intermidiate state as well and we've changes the language. Think of NFA for $a^\star b$
3. Instead of previous attempt, add new initial accepting state, with ϵ move to old initial state.

4.3 more closure properties

This one is actually important for a future lecture.

Definition 10.

- *For word $w = \sigma_1\sigma_2 \cdots \sigma_n$, we mark $w^R = \sigma_n\sigma_{n-1} \cdots \sigma_1$.*
- *For language L, we mark $L^R = \{w^R \mid w \in L\}$*

Claim. Regular languages are closed under reversing.

We take the original DFA, reverse all its arrows (this gives an NFA already) Change the initial state to a final state and change the final states to initial states. We do this by adding a new initial state and ϵ moves to all old final states. Here would be nice to have an example...

Claim. Closure under prefix. $prefix(L) = \{w \in \Sigma^\star \mid \exists v \in \Sigma^\star, wv \in L\}$

Mark all states that are not dead ends as accepting (an accepting state is reachable from them).

Claim. Mark $prefix(w)$ all prefixes of w. Mark $min(L) = \{w \in L \mid prefix(w) \cap L = \{w\}\}$. If L is regular than so is $min(L)$.

Drop all outgoing edges from accepting states

5 Regular Expressions are equivalent to Automata

So far we talked of

1. Regular expressions.
2. Regular languages.

Now we show they are equivalent:

Theorem 2. *language L has a regular expression iff it ha an automata accepting it.*

Which side is easier? Remember we saw that regular languages are closed under union, concatenation, and Kleene star.

5.1 From Regular expression to Automata

Remember, that a regular expression is built of three base cases: $\emptyset$, ϵ, $\sigma \in \Sigma$, and three operators: $\cup, \cdot, \star$.

For every expression that is a base case, we can easily find an automata giving the same language - show this, so they give rise to a regular language.

Since we've seen that regular languages are closed under the three operators, and that all base cases give regular languages, it means that all expressions give rise to regular languages. (formal proof, actually needs induction).

Example 9. Remind of the three automata constructions used for showing the closure of regular languages (last class). Show how to build an automata for $L = (ab \cup \epsilon)^\star a$, stage by stage.

For a regular expression e, we will get some automata $A(e)$. What is the connection between $|e|$ (the length of the expression as text) and $|A(e)|$ (the number of states)

Claim. $|A(e)| \leq 2|e|$

Proof. By induction on $|e|$. Base cases are clear. If e is not a base case, it is one of:

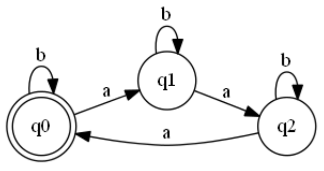
1. $e = e_1 \cdot e_2$. we get that $|e| = |e_1| + |e_2| + 1$. Then
$$|A(e)| = |A(e_1)| + |A(e_2)| \leq 2|e_1| + 2|e_2| = 2(|e_1| + |e_2|) < 2|e|$$
2. $e = e_1 \cup e_2$. we get that $|e| = |e_1| + |e_2| + 1$. Then
$$|A(e)| = |A(e_1)| + |A(e_2)| + 1 \leq 2|e_1| + 2|e_2| + 1 = 2(|e_1| + |e_2| + 1) - 1 < 2|e|$$
3. $e = e_1^\star$. we get that $|e| = |e_1| + 1$. Then
$$|A(e)| = |A(e_1)| + 1 \leq 2|e_1| + 1 = 2(|e_1| + 1) - 1 < 2|e|$$

□

5.2 From automata to regular expressions

Assume all our states are numbered $\{0, \ldots, n\}$, and that the initial state is marked 0.

Mark: $R(i, j, k)$ the set of words that move from state q_i to q_j without going through any state that is $k + 1$ or more (except for i and j themselves which may be greater). Show quite a number of examples on:



Now some general questions:

1. What is $R(i, j, -1)$? It has ϵ if $i = j$, and it has σ if there is a direct move from i to j with using σ (regardless of whether $i = j$ or not).
2. What is $R(i, j, n)$?
3. How can you express the language of the automata using $R(i, j, k)$'s?

Our goal now will be to prove that every $R(i, j, k)$ can be expressed as a regular expression. If we do this, then by what we just said, the language of the automata can be expressed as a regular expression.

We do this by induction of k. That was actually the whole point of introducing $R(i,j,k)$. For $k=-1$, the base case, we've seen that $R(i,j,-1)$ is very simple, and we can write it as a regular expression.

The induction step. Assume all the $R(i,j,k)$ have regular expressions. Lets build one for $R(i,j,k+1)$. Lets examine the path of a word in $R(i,j,k+1)$. It takes us from i to j and on the way uses only states from $\{1,\ldots,k+1\}$. lets mark all the uses of $k+1$ on the path:

$$i \rightsquigarrow k+1 \rightsquigarrow k+1 \rightsquigarrow k+1 \rightsquigarrow j$$

Do you see our induction hypothesis here? the sub-words leading between what we marked use only states in $\{0,\ldots,k\}$:

$$i \overset{R(i,k+1,k)}{\rightsquigarrow} k+1 \overset{R(k+1,k+1,k)}{\rightsquigarrow} k+1 \overset{R(k+1,k+1,k)}{\rightsquigarrow} k+1 \overset{R(k+1,j,k)}{\rightsquigarrow} j$$

So actually we can describe each words of $R(i,j,k+1)$ as:

$$R(i,j,k+1) = R(i,j,k) \cup R(i,k+1,k)R(k+1,k+1,k)^\star R(k+1,j,k)$$

By the induction hypothesis, all the $R(\cdots)$'s on the right side have regular expressions, we used regular expression operators, and so the whole thing is a regular expression.

5.2.1 Example of Transformation

We use the same example. we get for -1:

	0	1	2
0	$\epsilon \cup b$	a	
1		$\epsilon \cup b$	a
2	a		$\epsilon \cup b$

Then for $k=0$, for example:

$$R(2,1,0) = R(2,1,-1) \cup R(2,0,-1)R(0,0,-1)^\star R(0,1,-1) = \emptyset \cup a(\epsilon \cup b)^\star a = ab^\star a$$
$$R(2,0,0) = R(2,0,-1) \cup R(2,0,-1)R(0,0,-1)^\star R(0,0,-1) = a \cup a(\epsilon \cup b)^\star(\epsilon \cup b) = ab^\star$$
$$R(1,0,0) = R(1,0,-1) \cup R(1,0,-1)R(0,0,-1)^\star R(0,0,-1) = \emptyset \cup \emptyset(\epsilon \cup b)^\star(\epsilon \cup b) = \emptyset$$
$$R(1,1,0) = R(1,1,-1) \cup R(1,0,-1)R(0,0,-1)^\star R(0,1,-1) = b \cup \emptyset(\epsilon \cup b)^\star a = b$$

For $k=1$, for example:

$$R(2,0,1) = R(2,0,0) \cup R(2,1,0)R(1,1,0)^\star R(1,0,0) = ab^\star \cup ab^\star a \cdot b^\star \cdot \emptyset = ab^\star$$

And for seeing the end (do this with guessing the lesser k's):

$$R(0,0,2) = R(0,0,1) \cup R(0,2,1)R(2,2,1)^\star R(2,0,1) =$$
$$b^\star \cup b^\star ab^\star a \cdot (ab^\star ab^\star a \cup b)^\star \cdot b^\star ab^\star$$

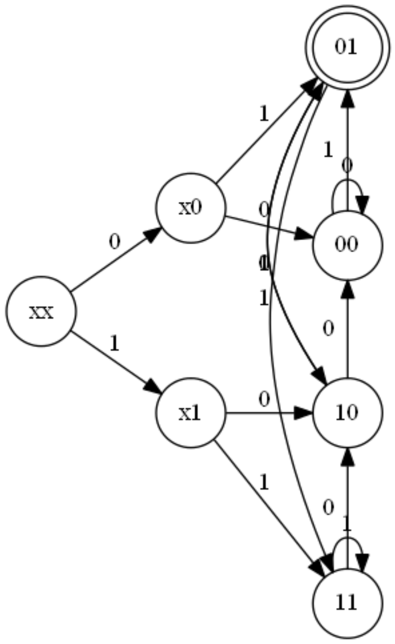
5.2.2 Size of transformation

Expression would be around 4^n where n is the number of states.

6 Minimizing automata

We've shown regular expression are equivalent to automata. That is very useful for example in looking for lines matching a certain regular expression. The problem is the automata we get may be very large. We would like to minimize them.

Examine the following automata for the language $(0 \cup 1)^{\star}01$:



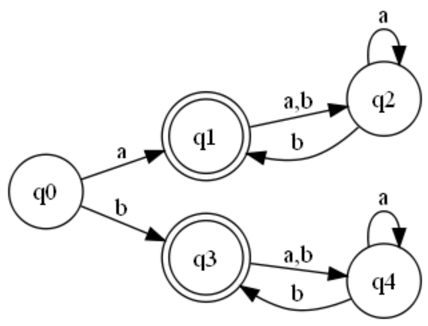
Add a couple of states that are not reachable from the init state. Note that they can be removed. Using DFS form the initial state, remove any state that is not touched.

A next attempt at minimization would be to try and see which states are equivalent. How about 10 and $x0$? They are both non-accepting, and they both lead to exactly the same states under the same letters. Go on with this and find more equivalences.

The idea later would be to merge such equivalent states - since whether we reach one or the other, we are in the same situation. Now we notice there is one special equivalence: xx and $x1$, they lead to equivalent states, and so are actually equivalent themselves.

create the new automata after all equivalences are merged (should be a 3-state thing).

So we seem to have some sort of algorithm. But what about this situation:



So no two lead to the same state but still we see the equivalence.

Our most general definition will therefore be:

Definition 11. *For $q, p \in Q$, we mark $q \sim p$ iff every $w \in \Sigma^\star$, $\delta^\star(q, w) \in F \Leftrightarrow \delta^\star(p, w) \in F$,*

Show example on example. But this definition is not practical at all.

6.1 relaxed equivalence

How do we find the equivalence classes?

Lets make life easier and think of only words of limited length, and build up.

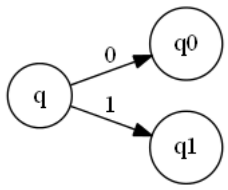
Definition 12. *For $q, p \in Q$, we mark $q \sim_n p$ iff every $w \in \Sigma^\star$, $|w| \leq n$, $\delta^\star(q, w) \in F \Leftrightarrow \delta^\star(p, w) \in F$,*

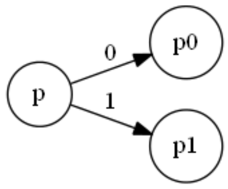
The difference in definitions is very slight. Some questions:

1. What does it mean $p \sim_0 q$?
2. In our examples who is $p \sim_1 q$?
3. Who is not? a word differentiates between two states if it leads one to an accepting state and the other to a non-accepting state. If there is one like this then they are not equivalent.
4. What is harder to be $p \sim_n q$ or $p \sim_{n+1} q$?

How can we use our knowledge of n-equivalence to calculate $n+1$-equivalence?

Examining p and q, if we want them to be $p \sim_{n+1} q$, then first they must be $p \sim_n q$. Then we look at what happens after one step:





For $n+1$ length words to not differentiate between p and q, we need $p_0 \sim_n q_0$ and $p_1 \sim_n q_1$.

6.2 algorithm

Show how we run on our first example:

1. Start with $\sim_0$: $A = \{01\}$, $B =$ all the rest.
2. $\sim_1$. We calculate for each equivalence class a table:

	0	1
xx	B	B
$x1$	B	B
$x0$	B	A
00	B	A
10	B	A
11	B	B

   This means that we split B into two classes. Now we have as $\sim_1$: $A = \{01\}$, $B = \{xx, x1, 11\}$, $C = \{x0, 00, 10\}$.
3. $\sim_2$ will turn out to be the same as $\sim_1$, and so we can stop.

The result is actually equal to the original equivalence $\sim$. We now reduce the whole thing and get a three state automata.

notes:

1. When building the automata there is no problem in uniting the states. this is a good check that you did not make mistakes.
2. Number of iteration is bound by n, since at each step we increase the number of sets, and the max number is the number of states.
3. Always check for all the sets - even if one has stabilized it can break down the next step because another set has just split.
4. Actually there is a theorem saying that this algorithm gives the minimal possible DFA.
5. How about NFAs?

A different way of viewing the process, is that we start with the most coarse equivalence relation possible, and the we each time correct it by seeing that some two states are just not possibly equivalent. and go on till we stop.

6.3 in an orderly fashion

```
remove all no reachable states.
Start with two sets:
  accepting states.
  all the rest
while (sets are changing)
  for each state:
    write down for each letter what set it is going to.
  for each set
    split it to new sets according to their behaviour as just noted.
Create the new automata:
  join each set to be one state.
```

7 Pumping Lemma

After all that we've learned about regular languages, we now wish to show that not all languages are regular. Recall one of our aims is to find a mathematical model for a computer. We can say a computer is a finite state machine, but think how many states!. Also, we will see that some very simple languages cannot be accepted by a finite automata.

7.1 The non-constructive Apporach

We can show that there is an language that is not regular, by saying that the magnitude of the set of all languages is $\aleph$ and that of the set of regular expressions is $\aleph_0$, and so the set of regular languages is the same.

I don't get into the details here (we will this proof exactly on Turing machines in the next semester). It is nice but we want to actually see a language.

Today we will see, for example, that

$$L = \{a^n b^n\} n \in \mathbb{N} = \{\epsilon, ab, aabb, aaabbb, ...\}$$

, is not regular. The intuition here is that scanning from left to right, we cannot make do with only $O(1)$ memory.

For that we will use a tool called "the Pumping Lemma"

7.2 The story

Take some regular language L. It has a DFA. Mark the number of states by N. Take some word $w \in L$ that satisfies $|w| \geq N$. Examine its path in the automata, and specifically look at the first N letter. We are sure to see a loop already there. Take one such loop. Mark x the part of w until the loop, y the part of w that takes us through the loop, and z, the last part of w that takes us all the way to the accepting state.

We clearly see that all $x \cdot y^{\star} \cdot z$ are also in L.

7.3 The Lemma

Lemma 2. ***(pumpin lemma for regular languages):*** *For a regular language, there is some N, such that for every $w \in L$, $|w| \geq N$, w can be written as $w = xyz$, and $xy^{\star}z \subseteq L$.*

In formally we say that w has a subword y within its first N letters, that is *pumpable*, i.e., no matter how much we pump it inside w, we will stay within L.

7.4 Example

Take as an example the language of all words in $\Sigma = \{a, b\}$ where the number of a's is divisible by 3. It has an automata of size 3, so take $N = 3$.

Take some word in the language of size at least 3: $w = abaabbb$, and see what sub-words y of w are pumpable. See a few examples.

The lemma actually sais that there must be a pumpable y within the first 3 letters - so in this case it is b.

7.5 How to use in Contradiction

The lemma sais that if L is regular then *blah blah*. So if we have a language and show that it does not satisfy *blah blah* then it cannot be regular.

What is the not of *blah blah*?

The original is:

> The is an N, such that for every long enough word w, there is an appropriate subword y, such that for all t, pumping y t times is in L.

negating it we get:

> For all N, the is some long enough word w, such that for all appropriate subwords y, there is a t, such that pumping y t times is not in L.

7.6 Examples

1. $L = \{a^n b^n\} n \in \mathbb{N}$, Show the whole thing.
2. $L = \left\{a^n b^k\right\} n < k$, For showing, try several choices of $w : b^N, ab^N, a^{N-1}b^N$. Last one works, but if we choose $a^N b^{N+1}$ it would be easier.
3. How about a regular language $\{a^n \mid n \in \mathbb{N}_{even}\}$.
4. $L = \left\{ww^R \mid w \in \Sigma^{\star}\right\}$, take $a^n b^n b^n a^n$
5. $L = \{a^p \mid p \in Primes\}$. Take $w = a^p$, where $p > n$. pumping: $a^x a^{ty} a^z$, where $t < n$ and $x + y + z = n$, so we get a^{p+ty}. Take $t = p$ and then $p + py$ is divisible by p.

7.7 using closure properties

This is usually don't get a chance to teach...

1. A different way to prove $L = \left\{ w \in \{a,b\}^\star \mid \#_a(w) = \#_b(w) \right\}$, intersect it with the regular $a^\star b^\star$ to get $\{a^n b^n \mid n \in \mathbb{N}\}$.
2. $L = \left\{ a^j b^k \mid j \neq k \right\}$. reverse and intersect with regular language $a^\star b^\star$ to get our standard no regular language.

8 Context Free Grammars

Instead of a process for checking words like automata, we define a process to generate words.

Show simple examples:

1. $L = \{a^n b^n \mid n \in N\}$.
2. $L = \left\{ a^i b^j \mid i \geq j \right\}$.
3. Here show L, and ask for a grammar. This shows the use of more than one variable. $L = \left\{ a^j c^t b^k \mid j < k \right\}$
4. The language of balanced parentheses. $S \to (S)|SS|\epsilon$

An important one:

Example 10. Mathematical expressions.

$$\begin{aligned}
\langle expr \rangle &\to \langle var \rangle \langle const \rangle \\
&\quad |\langle expr \rangle + \langle expr \rangle \\
&\quad |\langle expr \rangle * \langle expr \rangle \\
&\quad |\langle expr \rangle / \langle expr \rangle \\
&\quad |\langle expr \rangle - \langle expr \rangle \\
&\quad |(\langle expr \rangle) \\
\langle var \rangle &\to a|b|\dots|z \\
\langle digit \rangle &\to 0|1|\dots|9 \\
\langle const \rangle &\to digit|digit\langle const \rangle
\end{aligned}$$

8.1 definitions

A grammar G is a quadruple $\langle V, \Sigma, P, S \rangle$. V is the set of variables. Σ is the set of terminals. P is the set of rewrite rules of the form $a \to \beta$, where $a \in V$, and $\beta \in (V \cup \Sigma)^\star$. $S \in V$ is the initial symbol.

We use A, B, C to mark non-terminals, and a, b, c to mark terminals.

Definition 13. *For $\phi_1, \phi_2 \in (V \cup \Sigma)^\star$, we say ϕ_2 is immediately derived from ϕ_1, iff there is some $A \in V$, $\psi, \chi \in (V \cup \Sigma)^\star$, s.t.*

1. $\phi_1 = \psi_1 A \psi_2$
2. $A \to \alpha \in P$
3. $\phi_2 = \psi_1 \alpha \psi_2$

We mark this $\phi_1 \Rightarrow \phi_2$

Definition 14. *We mark* $\phi_1 \overset{\star}{\Rightarrow} \phi_2$ *if we can derive through a natural number of direct derivations.*

Definition 15. *The language of a grammar* G *is* $L(G) = \left\{ w \in \Sigma^\star \;\middle|\; S \overset{\star}{\Rightarrow}_G w \right\}$

8.2 Sentential forms

Definition 16. *The set of sentential forms of a grammar:*

$$T(G) = \left\{ w \in (V \cup \Sigma)^\star \;\middle|\; S \overset{\star}{\Rightarrow} w \right\}$$

1. Of a grammar for $\{a^n b^n \mid n \in \mathbb{N}\}$.
2. Of this grammar:
$$\begin{aligned} S &\to \epsilon | aA | bB \\ A &\to a | aS \\ B &\to b | bS \end{aligned}$$
3. $L = \{wavb \mid w, v \in \Sigma^\star, |w| = |v|\}$ A possible grammar is:
$$\begin{aligned} S &\to Tb \\ T &\to XTX | a \\ X &\to a | b \end{aligned}$$

Claim. $L(G) = T(G) \cap \Sigma^\star$

8.3 more examples

1. $L = \left\{ a^n b^m c^k d^l \mid n + m = k + l \right\}$ The grammar is:
$$\begin{aligned} S &\to X_{ad} \\ X_{ad} &\to aX_{ad}d | X_{bd} | X_{ac} \\ X_{ac} &\to aX_{ac}c | X_{bc} \\ X_{bd} &\to bX_{bd}d | X_{bc} \\ X_{bc} &\to bX_{bc}c | \epsilon \end{aligned}$$
2. $L = \{a^m b^n \mid m \le 2n\}$ The grammar is:
$$S \to \epsilon | Sb | aaSb | aSb$$

9 Properties of Context Free Languages

9.1 Closure rules

Theorem 3. *Context free grammars are closed under union, concatenation, and Kleene star.*

Proof. 1. union. The claim in words: If L_1 is context free (has grammar G_1) and L_2 is context free (has grammar G_2), then $L = L_1 \cup L_2$ is context free (has grammar G).
Lets look at an example:

$$L_1 = \{a^n b^n \mid n \in \mathbb{N}\}, \qquad G_1 = \{S \to aSb|\epsilon\}_{cc}$$
$$L_1 = \{a^n b^n \mid n \in \mathbb{N}\}, G_1 = \{S \to aSb|\epsilon\}$$

What is L? $L = \{a^n b^n a^m b^m \mid n, m \in \mathbb{N}\}$. We can find a grammar for it. And this brings us to the general solution:
Take both grammars, just change them so they don't have any variables in common. Let S_1 be G_1's initial variable, and S_2 be G_2's. add to these grammars one rule: $S \to S_1 \cup S_2$.

2. concatenation. Same (show the example), just extra rule is $S \to S_1 \cdot S_2$.
3. Kleene star. Same but only G_1 and the extra rule will be $S \to S \cdot S_1|\epsilon$.

□

Corollary 1. *If L is regular, then it has a context free grammar.*

Proof. If L is regular, then it has a regular expression. A regular expression is made out of base: $a \in \Sigma$ which we can build grammar for, and ϵ and $\emptyset$ which we can also build grammar for. From there it is constructed by concatenation, union and Kleene star. so we are done. □

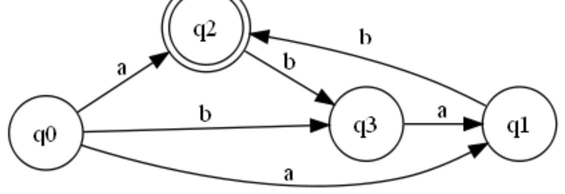
This proof is constructive. Examine what is does to the example expression $(a \cup b)^\star aa \cup \epsilon$: Stage by stage:

a	$S_1 \to a$
b	$S_2 \to b$
ϵ	$S_3 \to \epsilon$
$a \cup b$	$S_4 \to S_1 \| S_2$
$(a \cup b)^\star$	$S_5 \to S_5 S_4 \| \epsilon$
$(a \cup b)^\star a$	$S_6 \to S_5 S_1$
$(a \cup b)^\star aa$	$S_7 \to S_6 S_1$
$(a \cup b)^\star aa \cup \epsilon$	$S \to S_7 \| S_3$

What size of a grammar does this proof give? if we started with a regular expression that its linear. If we started with an automata it will be exponential.

9.2 Regular Grammars

Lets do the move from automata to grammars differently. Take an automata:



$$\begin{aligned} S &\to Q_0 \\ Q_0 &\to aQ_2|bQ_3|aQ_1|\epsilon \\ Q_1 &\to bQ_3|\epsilon \\ Q_2 &\to aQ_1 \\ Q_3 &\to bQ_2 \end{aligned}$$

Would it work with an NFA? ϵ-NFA (how would this one work) ? yes. and translation is linear.

Notice we get a grammar where all the rules are with only one non-terminal, and its always the right most. This is called *right linear* grammar.

Claim. If a grammar right linear, then the language it defines is regular.

The idea is the same, just other way around. Take an example and show the transformation.

$$\begin{aligned} S &\to aS|bbB \\ B &\to baB|aA \\ A &\to S|babB \end{aligned}$$

So we actually got a new characterization of regular languages, as the languages that have a right linear grammar.

9.2.1 left-linear grammar

There are also left-linear grammars. They are regular.

we need a lemma for this:

Lemma 3. *Given a grammar G, if we reverse all its rules to get grammar G': Instead of $X \to \alpha$, take $X \to \alpha^R$, then $L(G) = L(G')^R$.*

Proof. Given a grammar, reverse all its rules.

Show it on a simple example like $L = \left\{a^n b^n c^k \mid n, k \in bbbn\right\}$.

Show general claim: by showing the same set of derivations on original and new grammar. Say they are always reverse of each other (the sentential forms). and show the induction step: Lets say we have some crazy sentential form, and new one is reversed, then do one derivation and see that you stay reversed. □

Side note: This actually shows that context-free languages are closed under reversing.

Claim. Left linear grammar gives a regular language.

1. Take a left-linear grammar G and reverse all the rules to get G'. We get that $L(G) = L(G')^R$ (this is in the lemma).
2. G' is right-linear and so is regular, so $L(G')$ is regular.
3. We know regular languages are closed under reversing. so $L(G')^R$ is regular.
4. and so, $L(G)$ is regular.

9.3 parse trees

This is actually as far as this class goes. Move all this stuff somewhere.

Examine the language of balanced parentheses:

$$S \to (S)|SS|\epsilon$$

We can parse (())() in two different ways. We can draw the tree and see its the same. Only difference is the order. Show yield on each subtree.

Definition 17. *A parse tree T and its yield $y(T)$ in a grammar G is:*

- *For $\sigma \in \Sigma$, a leaf with σ in it. its yield $y(T) = \sigma$.*
- *For ϵ, a leaf labelled ϵ, its yield is $y(T) = \epsilon$.*
- *If $T_1, ..., T_n$ are parse trees with roots labels $A_1, \dots, A_n$, and there is a rule $A \to A_1 \dots A_n$ then there is a parse tree with A at the root, and $A_1 \dots A_n$ its children in that order. Its yield is $y(T_1) \cdots y(T_n)$*

We will be interested in parse trees whose root is labelled S.

9.4 ambiguous grammars

Example 11. Examine the following grammar:

$$E \to (E)|E + E|E * E|num$$

This is important for programming. Which tree would you build for $5 + 6 * 3$?.

Definition 18. *G is ambiguous if there are two parse trees T_1 and T_2 that yield the same word.*

For programming languages we don't like this, so we can change the grammar:

$$\begin{aligned} &E \to E + T|T \\ &T \to T * F|F \\ &F \to (E)|num \end{aligned}$$

We comment that there are context free languages that are inherently ambiguous s.t.

$$\left\{a^n b^k c^k d^n \mid n, k \in \mathbb{N}\right\} \cup \left\{a^n b^n c^k d^k \mid n, k \in \mathbb{N}\right\}$$

The word $a^n b^n c^n d^n$ will always have two parse trees.

9.5 real programming language

In real compilers. First part splits everything to *tokens*, using a regular language. We get variables, reserved words, function names, function calls, numbers, block start and ends, parentheses, etc.

Second part is defined using a context free language. For example, for one sentence we get:

$$\begin{array}{ll} S & \to S_{if}|S_{while}|S_{assign} \\ S_{if} & \to if\ Cond\ then\ S|if\ Cond\ then\ S\ else\ S \\ S_{while} & \to while\ Cond\ do\ S \\ S_{assign} & \to V = E \\ E & \to (E)|E+E|(E*E)|V|Const \end{array}$$

We have ambiguity here: *if ... then if ... then else*. Where does the second else belong? This should be solved for the language to be useful.

10 Pushdown Automata

10.1 motivation

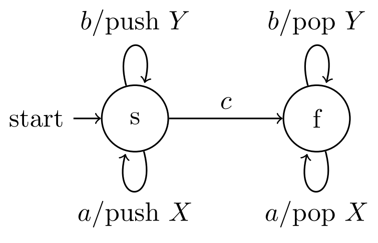
What kind of automata would we need to recognize $\left\{ww^R \mid w \in \Sigma^\star\right\}$. All we need to add to the finite automata is a stack. We also need non-determinism to guess the middle of the word.

How about the set of balanced parentheses? push when you see "(" and pop when you see ")".

Thinking again, rules such as $A \to aB$ are easy for finite automata. but $A \to aBb$? the stack is handy for pushing the b, it will resurface when we complete B.

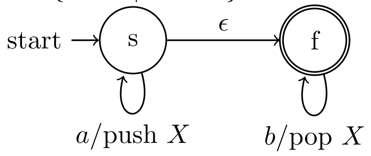
10.2 Examples

1. For language $L = \left\{wcw^R \mid w \in (a \cup b)^\star\right\}$.

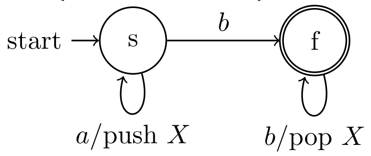


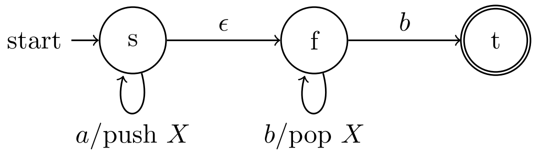
2. Show a good word, a word that cannot complete, and a word that can complete but with a non-empty stack.
   For language $L = \left\{ww^R \mid w \in (a \cup b)^\star\right\}$, just turn c transition to ϵ.

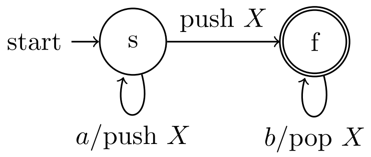
3. $L = \{a^n b^n \mid n \in \mathbb{N}\}$.



4. $L = \left\{a^n b^{n+1} \mid n \in \mathbb{N}\right\}$. A few solutions:







10.3 Formal definition

Show all these definitions on our last example.

A push-down automata is a sextuple: $M = \langle Q, \Sigma, \Gamma, \delta, q_0, F\rangle$ where

- Q is the set of states.
- Σ is the alphabet.
- Γ is the stack alphabet.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the final states.
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma^\star \to P(Q \times \Gamma^\star)$ Notice we need finiteness here.

Definition 19. *Configuration is $Q \times \Sigma^\star \times \Gamma^\star$. Our state, what is there to be read, and what is in the stack (read top down).*

Definition 20. *We say $(p, aw, \beta\alpha) \vdash (q, w, \gamma\alpha)$ iff there is a transition $(q, \beta) \in \delta(p, a, \gamma)$. note that a may be ϵ.*

Definition 21. *We say $c_1 \vdash^\star c_2$ iff there is some sequence of configurations $t_1 \ldots t_n$ where $c_1 = t_1$, $c_2 = t_n$, and $t_1 \vdash t_2 \vdash \ldots \vdash t_n$. Including the case $n = 1$.*

Definition 22. *We say M accepts w iff $(q_0, w, \epsilon) \vdash^\star (q, \epsilon, \epsilon)$ for some $q \in F$.*

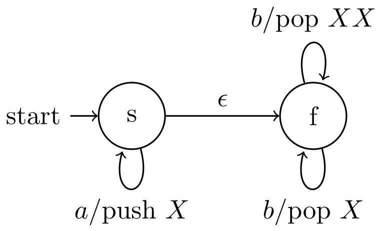
Definition 23. *The language of M, $L(M) = \{w \in \Sigma^\star \mid M \text{ accepts } w\}$.*

10.4 More examples

1. For language $L = \{w \in \Sigma^\star \mid \#_a(w) = \#_b(w)\}$. First show picture then formal stuff. We set $\Gamma = \{a, b\}$. Only one state s, $F = \{s\}$.

$$\begin{aligned} \delta(s, a, \epsilon) &= (s, a) \\ \delta(s, a, b) &= (s, \epsilon) \\ \delta(s, b, \epsilon) &= (s, b) \\ \delta(s, b, a) &= (s, \epsilon) \end{aligned}$$

2. Note that a stack automata can imitate any NFA, simply by not using the stack.
3. $L = \{a^m b^n \mid m \leq n \leq 2m\}$.



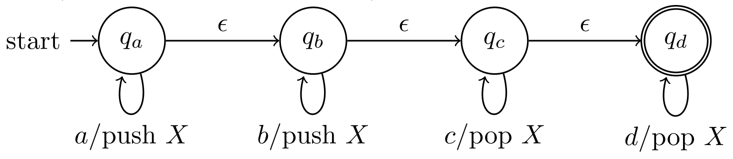
4. The language generated by the grammar:

$$S \to \epsilon | SS | [S] | (S)$$

Show picture, not this:

$$\begin{aligned} \delta(s, [, \epsilon) &= (s, [) \\ \delta(s,],]) &= (s, \epsilon) \\ \delta(s, (, \epsilon) &= (s,)) \\ \delta(s,),]) &= (s, \epsilon) \end{aligned}$$

5. $L = \left\{a^m b^n c^k d^l \mid m + n = k + l\right\}$.



6. $L = \left\{a^n b^m c^k \mid m = 2n \text{ or } m = 2k\right\}$
7. $L = \{w \in \Sigma^\star \mid \#_a(w) = 2\#_b(w)\}$

10.5 Context Free Grammars and Pushdown Automata

Theorem 4. *L has a context free grammar iff it has a stack automata*

We prove this in two parts.

10.5.1 from Grammar to Automata

Claim. If L has grammar G then it has a stack automata M.

Define M:

1. $Q = \{p, q\}$
2. p is initial state
3. $\Gamma = \Sigma \cup V$
4. Final state is q.
5. transitions: (usually I skip this and show it on the example).
   (a) $\delta(p, \epsilon, \epsilon) = (q, S)$
   (b) $\delta(q, \epsilon, A) = (q, x)$ for each rule $A \to x \in P$
   (c) $\delta(q, a, a), (q, e))$ for each $a \in \Sigma$

This automata mimics a left-most derivation.

Example 12. Grammar for $\left\{a^n c^k b^n \;\middle|\; n \in \mathbb{N}, k \in \mathbb{N}^+\right\}$,

$$\begin{aligned} S &\to aSb|X \\ X &\to cX|c \end{aligned}$$

Automata is (draw it):

1. $\delta(p, \epsilon, \epsilon) = (q, S)$
2. $\delta(q, \epsilon, S) = (q, aSb), (q, X)$
3. $\delta(q, \epsilon, X) = (q, cX), (q, c)$
4. $\delta(q, a, a) = (q, \epsilon)$
5. $\delta(q, b, b) = (q, \epsilon)$
6. $\delta(q, c, c) = (q, \epsilon)$

Run this automata on *aacbb*.

10.5.2 from Automata to Grammar

Not Taught.

10.6 Intersection of context free and regular

I don't teach this...

Theorem 5. *If L_1 is context free and L_2 is regular then $L_1 \cap L_2$ is context free.*

Proof. Take push-down automata A_1 and NFA A_2. Define pushdown automata

- $Q = Q_1 \times Q_2$
- $\Gamma = \Gamma_1$
- $q_0 = (q_{0_1}, q_{0_2}$
- $F = F_1 \times F_2$
- For every transition $\delta_1(q_1, a, \alpha) = (p_1, \beta)$ and transition $\delta_2(q_2, a) = p_2$ we define transition: $\delta((q_1, q_2), a, \alpha) = ((p_1, p_2), \beta)$. This includes the case $a = \epsilon$.

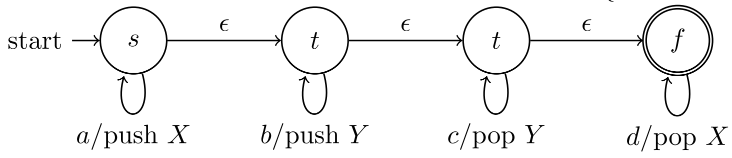
□

Example 13. The language with equal number of a's and b's where *aaba* or *babb* appears in the word is context-free.

$$\{w \in \Sigma^\star \mid \#_a(w) = \#_b(w)\} \cap \Sigma^\star(aaba \cup babb)\Sigma^\star$$

11 Alternative Definitions

a reminder for last class. Construct an automata for $L = \{a^n b^m c^m d^n \mid n, m \in \mathbb{N}\}$:



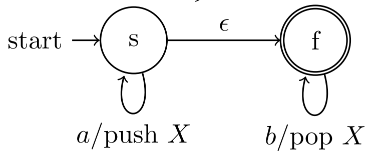
Recall our standard definition of the language of an automata $M : L(M)$.
We check out alternative definitions of a pushdown automata.

11.1 Acceptance by Final State

Definition 24. *For automata M we can define acceptance by final state:*

$$L_f(M) = \{w \in \Sigma^\star \mid (q_0, w, \epsilon) \vdash^\star (f, \epsilon, \alpha), f \in F, \alpha \in \Gamma^\star\}$$

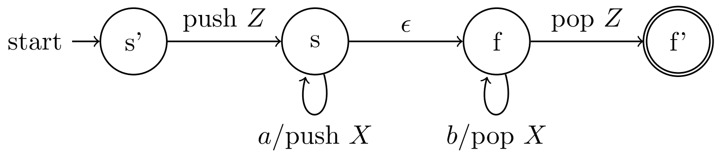
Check a few words for the last automata, and realize $L_f(M) = \left\{a^n b^m c^k \mid k < m\right\} \cup \left\{a^n b^m c^m d^k \mid k \leq n\right\}$ Another example:



In it, $L(M) = \{a^n b^n \mid n \in \mathbb{N}\}$, and $L_f(M) = \{a^n b^m \mid n \leq m\}$,

11.1.1 showing equivalence

For the last example, can we write an automata M s.t. $L_f(M) = \{a^n b^n \mid n \in \mathbb{N}\}$? yes. Using some extra symbol we put at the bottom of the stack as we start.



Generalizing this method we can claim:

Claim. For every automata M, there exists automata M' s.t. $L(M) = L_f(M)$

The way to do it is just as said: add two new states s', f', add a new stack symbol Z. s' is initial, f' is accepting. add moves:

$$\begin{aligned} \delta(s', \epsilon, \epsilon) &= (s, Z) \\ \forall f \in F\ \delta(f, \epsilon, Z) &= (f', \epsilon) \end{aligned}$$

Show this in a picture.

Can we do the other way around? Given an automata M, can we create a new automata M' s.t. $L_f(M) = L(M')$. Look at our two examples.

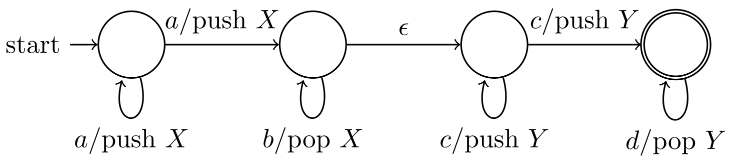
Construction adds a new emptying state f' which is made the only final state. For every $\gamma \in \Gamma$ it has a transition: $\delta(f', \epsilon, \gamma) = (f', \epsilon, \epsilon)$. For every old accepting state f we add an ϵ transition from it to f'.

11.2 Acceptance by Empty Stack

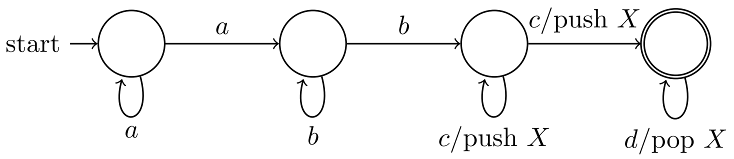
Definition 25. *For automata M we can define acceptance by empty stack:*

$$L_e(M) = \{w \in \Sigma^\star \mid (q_0, w, \epsilon) \vdash^\star (q, \epsilon, \epsilon), q \in Q\}$$

Look at automata:



$L(M) = \left\{a^{n+1}b^nc^{m+1}d^m \mid n, m \in \mathbb{N}^+\right\}$, and $L_e(M) = \left\{a^nb^nc^md^m \mid n \in \mathbb{N}^+, m \in \mathbb{N}\right\}$
another example:



$L(M) = \left\{a^nb^kc^td^t \mid n, k, t \in \mathbb{N}^+\right\}$, and

$L_e(M) = \{\epsilon\} \cup \left\{a^nb^k \mid n \in \mathbb{N}^+, k \in \mathbb{N}\right\} \cup \left\{a^nb^kc^td^t \mid n, k \in \mathbb{N}^+ t \in \mathbb{N}\right\}$

11.2.1 showing equivalence

Given an automata M, how can we change it to M' so that $L(M) = L_e(M')$? first in our examples. We see a problem. L_e will always have ϵ. so our claim is a little different:

Claim. For every M there is some M' s.t. $L(M) \cup \epsilon = L_e(M')$

In general, we want to force M to accept only when it is in a final state. so again push a Z in the beginning and from each final state pop Z and move to a new final state. Show this on example.

Note there is no accepting state here.

Other way. Given automata M we can find automata M' s.t. $L_e(M) = L(M')$? yes. We can just turn all states to be accepting.

12 Checking Membership in CFL

We are given some grammar G and a word w - we would like to check if $w \in L(G)$ in an efficient way. Exponentially this is easy.

For this we first have to change our grammar to have a standard form:

12.1 Chomsky Normal Form

A grammar is in Chomsky Normal form if all rules are of one of the forms:

1. $A \to BC$, where $A, B, C \in V$, and $B, C \neq S$.
2. $A \to a$, where $a \in \Sigma, A \in V$.
3. $S \to \epsilon$

We can change a grammar to this form. Each step handles a different kind of problem rules. We work on the example:

$$\begin{aligned} S &\to ASA|aB \\ A &\to B|S \\ B &\to b|\epsilon \end{aligned}$$

1. If S appears somewhere on the right, then make a new variable that will be the initial variable S_0 add rule $S_0 \to S$.

$$\begin{aligned} S_0 &\to S \\ S &\to ASA|aB \\ A &\to B|S \\ B &\to b|\epsilon \end{aligned}$$

2. We have a problem with these: $A \to \epsilon$, where $A \neq S$. Take such a variable A, and for every rule that uses this A on the right side, create a copy of it with A removed. This can create new $B \to \epsilon$ rule - unless this rule was already removed.
   We start with B:

$$\begin{aligned} S_0 &\to S \\ S &\to ASA|aB|a \\ A &\to B|\epsilon|S \\ B &\to b \end{aligned}$$

   Now we have to deal with A. Notice we have to take care of all possibilities:

$$\begin{aligned} S_0 &\to S \\ S &\to ASA|AS|SA|aB|a \\ A &\to B|S \\ B &\to b \end{aligned}$$

   Notice we did not add the rule $S \to S$ because it is unnecessary.
3. Handling unit rules. We want to remove all rules: $X \to Y$. For each such rule, remove it, and instead for each rule $Y \to u$ add a new rule $X \to u$, unless this new rule was already removed.
   Lets start with $A \to B$:

$$\begin{aligned} S_0 &\to S \\ S &\to ASA|AS|SA|aB|a \\ A &\to b|S \\ B &\to b \end{aligned}$$

Now $A \to S$:

$$\begin{aligned} S_0 &\to S \\ S &\to ASA|AS|SA|aB|a \\ A &\to b|ASA|AS|SA|aB|a \\ B &\to b \end{aligned}$$

Now $S_0 \to S$:

$$\begin{aligned} S_0 &\to ASA|AS|SA|aB|a \\ S &\to ASA|AS|SA|aB|a \\ A &\to b|ASA|AS|SA|aB|a \\ B &\to b \end{aligned}$$

4. Long rules. for a rule $A \to u_1...u_k$, where $k \geq 3$ we split it to small rules:

$$\begin{aligned} A &\to u_1A_1 \\ A_1 &\to u_2A_2 \\ &... \\ A_{n-2} &\to u_{n-1}A_n \end{aligned}$$

with all these new variables.

$$\begin{aligned} S_0 &\to AA_1|AS|SA|aB|a \\ S &\to AA_1|AS|SA|aB|a \\ A &\to b|AA_1|AS|SA|aB|a \\ B &\to b \\ A_1 &\to SA \end{aligned}$$

5. Mixed rules. $A \to u_1u_2$, where at least one is a terminal. for example u_1. Add a new variable A_1 and a rule $A_1 \to u_1$.

$$\begin{aligned} S_0 &\to AA_1|AS|SA|\hat{A}B|a \\ S &\to AA_1|AS|SA|\hat{A}B|a \\ A &\to b|AA_1|AS|SA|\hat{A}B|a \\ B &\to b \\ A_1 &\to SA \\ \hat{A} &\to a \end{aligned}$$

Note that this procedure may create an exponential blowup in the size of the grammar. This is only because of the step removing ϵ rules. This can be fixed by changing the order of the steps. Note that not every order will work correctly.

12.2 Algorithm for checking Membership

Our aim was, given a grammar G and a word w to check if $w \in G$. First step is to make G into Chomsky normal form.

We use the Dynamic Programming technique. Mark $w = a_1a_2 \ldots a_n$ We want a table $T[][]$, where $T[i,j]$ will hold the set of variables that can create the word $a_i...a_j$.

If we have all those, we just need to check that $S \in T[1,n]$. Let us fill the table:

1. Base step is $T[i,i]$. because of the normal form, its easy. It will contain all variable that have a rule $A \to a_i$.
2. Lets say we calculated all intervals of size $< k$, lets calculate the table for intervals of size k. We want to calculate $T[i,j]$, where $j - i + 1 = k$.
   Run over all t, from i until $j-1$. For every $B \in T[i,t]$ and $C \in T[t+1,j]$ if there is a rule $A \to BC$ then add A to $T[i,j]$.

That's it.

For a word of length n, and grammar of size t (variables, rules), what is the running time? table is $O(n^2)$, each place holds a maximum of t variables. Calculating one place in the table should around $O(nt^2)$ time (maybe slightly more). So in total: $O(n^3t^2)$.

Show this running on

$$\begin{aligned} S &\to AB|a \\ A &\to BC|a \\ B &\to AB|b \\ C &\to c \end{aligned}$$

and some simple word of length 6 (first derive it).

13 Pumping Lemma for Context-Free Languages

Our aim today is to show that some interesting languages are not context-free. They do not have a context-free grammar (equivalently: no push-down automata).

Lets look at the language $\{a^nb^nc^n \mid n \in \mathbb{N}\}$. Think of a stack automata for it. it does not seem to work. We will indeed show in this class that this language is not context-free. We will also show a general technique to show that a language is not context free.

13.1 Pumping in a parse tree

Look at this Grammar in Chomsky's normal form:

$$\begin{aligned} S &\to AB|a \\ A &\to BC|a \\ B &\to AB|b \\ C &\to c \end{aligned}$$

Show a parse tree and how we can expand it. If tree is large enough we can always do that.

13.2 The example

Assume our language $\{a^nb^nc^n \mid n \in \mathbb{N}\}$ is context-free.

1. Therefore it has a Chomsky-Normal-Form grammar.

2. Mark by N its number of variables.
3. Take the word $a^M b^M c^M$ where M is 2^N.
4. Look at the parse tree for this word. It is binary and every leaf is a letter (and not ϵ).
5. Recall that in any binary tree, the number of leafs is at most 2^d, where d is the depth when we count edges. So, if the number of leafs is l then the depth is at least $\log_2(l)$.
6. The depth in our tree is at least $\log_2(3M) > \log_2(M) = N$.
7. And so, on the path, there must be a variable that appears twice.
8. Draw the diagram that shows how this breaks up our word into $xyzuv$. Note that we can assume $|yzu| \leq M$ because we pick the first variable that repeats itself (starting from the bottom).
9. Now we can do pumping, and get all words $xy^t zu^t v$, for every $t \in \mathbb{N}$, are generated by the grammar.
10. But we easily see that taking, for example, $t = 0$, will take us out of the language.

13.3 The Lemma

Theorem 6. *For every context free language L, there is an $M > 0$ such that any $w \in L$, with $|w| \geq M$, can be written as $w = xyzuv$, so that $|xyv| \leq M$ and $|yv| > 0$, and for every $t \in \mathbb{N}$, we have $xy^t zu^t v \in L$.*

13.4 Usage

We want to show a language is not context-free.

1. Assume L is context free.
2. It has some M according to the lemma.
3. Take some word $w \in L$, $|w| \geq M$.
4. For every possible partitioning of w into $xyzuv$, with $|yzu| \leq N$, there is some t s.t. $xy^t zu^t v \notin L$
5. That is the contradiction.

13.5 Examples

I will probably manage to do the first and maybe second. Others can be in the exercise class.

1. $L = \left\{a^n b^n c^n d^m \;\middle|\; n, m \in \mathbb{N}^+\right\}$. Take $w = a^N b^N c^N d$. Split to two cases. If d not pumped, then like before. If it is, then take $t = 0$, and it disappears in contradiction.
   Notice how taking a different word would not really work, cause we cvan have one d as the pumped part.
2. $L = \left\{a^k b^l c^s \mid s = min(k, l)\right\}$. Take $w = a^N b^N c^N$. if pumped part has no c's then take $t = 0$. If it does, then take $t = 2$, since there are no a's, we contradict.

3. $L = \{ww \mid w \in \Sigma^\star\}$ Take $w = a^N b^N a^N b^N$.
   (a) $vxy = a^p b^q$. w.l.o.g it is in the first part. take $t = 0$ and you get an unbalanced word either in b's or in a's.
   (b) $vxy = b^q a^p$ works the same way.
4. $L = \{a^n \mid n \in \mathit{Primes}\}$. Take $w = a^p$ for $p > N$. Mark $vy = k$. we get $a^{p+tk} \in L$ for every $t \in \mathbb{N}$. Take $k = p$ and you contradict. Note here that every context free language over one letter alphabet is regular.

13.6 Using Closure properties

I can't believe I'll have time to teach this. We can then use closure properties to show that other languages are not context free.

Example 14. $L = \left\{w \in \{a, b, c\}^\star \mid \#_a(w) = \#_b(w) = \#_c(w)\right\}$ is not context free. If it were, intersect it with $a^\star b^\star c^\star$ to get our standard non-context free language.

Theorem 7. *Context free languages are not closed under intersection and complementation.*

Notice this does not mean every intersection or complementation will create a non-context-free language.

Proof. Take $L_1 = \{a^n b^n c^m \mid m, n \in \mathbb{N}\}$ and $L_2 = \{a^m b^n c^n \mid m, n \in \mathbb{N}\}$. They are context free but their intersection is not.

If complementation was ok, then $({L_1}^c \cup {L_2}^c)^c$ was context-free (since union is ok), but this is exactly $L_1 \cap L_2$, in contradiction. $\square$