

## Predicate:

a mathematical predicate can be True or False.  
predicates are functions with boolean return values:  
 $P, Q(n), R(x,y,z)$

## Logical Operators:

AND:  $P \wedge Q$  || OR:  $P \vee Q$  || NOT:  $\neg P$

Implication:  $P \implies Q = \neg P \vee Q$

## Distributive Rule

$$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$$

$$P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$$

## De Morgans Law

## Implication

$$\begin{aligned} \neg(P \wedge Q) &= \neg P \vee \neg Q & P \implies Q = \text{True} \& \neg P \implies Q = \text{True} \\ \neg(P \vee Q) &= \neg Q \implies \neg P & P \implies \neg Q = \text{False} \\ P \implies Q &= \neg Q \implies \neg P & \neg P \implies \neg Q = \text{True} \end{aligned}$$

## Quantors

$$\begin{aligned} \text{OR: } \bigvee_{k=0}^n P_k && \text{AND: } \bigwedge_{k=0}^n P_k \\ \text{P true for any } k \in 0..n && \text{P true for all } k \in n \\ \text{All: } \forall k \in 0..n = P_k && \text{Exists: } \exists k \in 0..n = P_k \\ \text{for all k P = True} && \text{a k exists where P = True} \end{aligned}$$

## Normalforms

### disjunctive

$$(x_1 \wedge x_2) \vee (\bar{x}_1 \wedge x_2) \vee (x_1 \wedge \bar{x}_2)$$

### conjunctive

$$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)$$

These are useful for true and false tables

This one would result to true if  $x_1$  or  $x_2$  is true.

## Quantities:

$$\emptyset = \{\} \quad [n] = \{0..n\} \quad \{a .. z\}$$

### Union:

$$A \cup B = \{x | x \in A \vee x \in B\} \quad A \cap B = \{x | x \in A \wedge x \in B\}$$

### Complement:

$$\bar{A} = \{x | x \notin A\} \quad A \setminus B = \{x \in A | x \notin B\}$$

### Pairs

$$A \times B = \{(a, b) | a \in A \wedge b \in B\}$$

### n-Tuples

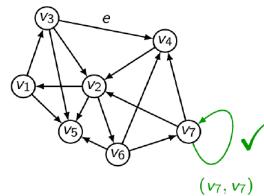
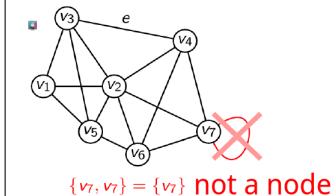
$$\bigtimes_{k=0}^n A_i = \{(a_0, a_1, \dots, a_n) | a_i \in A_i\}$$

## Undirected Graph

doesn't have directions, and therefore can't have edges to itself

## Directed Graph

This does have directions, therefore an edge to itself is valid!



Vertices V = { $v_1, v_2, \dots, v_n$ } Edge e = { $v_3, v_4$ }

EdgeCount E = {e|eEdge}

## Proofs:

### constructive Proof (proof by reforming)

Consider  $ax^2 + bx + c = 0$ , we can proof this to have 2 solutions by reforming.

$$\begin{aligned} ax^2 + bx + c &= 0 \\ x^2 + 2\frac{b}{2a}x + \frac{b^2}{4a^2} - \frac{b^2}{4a^2} + \frac{c}{a} &= 0 \\ x^2 + 2\frac{b}{2a}x + \frac{b^2}{4a^2} &= \frac{b^2 - 4ac}{4a^2} \\ \left(x + \frac{b}{2a}\right)^2 &= \frac{b^2 - 4ac}{4a^2} \\ x + \frac{b}{2a} &= -\frac{b}{2a} \pm \sqrt{\frac{b^2 - 4ac}{4a^2}} \end{aligned}$$

If  $b^2 - 4ac > 0$  then we have 2 solutions!

## Proof by contradiction

Take -2, it isn't a natural number. We can prove this by claiming the opposite.

If -2 is a natural number, then it has all the attributes of a natural number.

For example, it should be possible to take the square root of -2.

$$\sqrt{-2} = NaN$$

As you can see -2 does not have this attribute and is therefore NOT a natural number!

## Proof by Induction

This is particularly useful if you want to check an attribute for a range of numbers such as n or n+1

Base claim:

$$P(n) = \sum_{k=1}^n k = \frac{n(n+1)}{2}$$

Anker: check for n=1

$$P(1) = \frac{1(1+1)}{2} = 1$$

Hypothesis: it also works for n+1

$$\begin{aligned} P(n+1) &= \sum_{k=1}^{n+1} k = \left(\sum_{k=1}^n k\right) + n+1 \\ &= \frac{(n+1)(n+2)}{2} \\ &= \frac{(n+1)(n+1+1)}{2} \end{aligned}$$

## Alphabet and Word

$\Sigma$  = Alphabet: Nonempty Quantity of characters

$\Sigma^n = \Sigma \times \dots \times \Sigma$  = String

w  $\in \Sigma^n$  An element in that string is a Word with length n.

$\epsilon \in \Sigma^0$  The empty word, don't forget the empty word!

Quantity of all words:

$$\Sigma^* = \{\epsilon\} \cup \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup \dots = \bigcup_{k=0}^{\infty} \Sigma^k$$

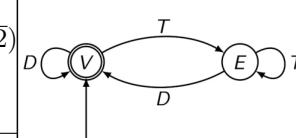
Language  $L \subset \Sigma^*$  = Language

$L = \emptyset \subset \Sigma^*$  = Empty Language

$L = \Sigma^* \rightarrow \Sigma = \{0,1\}$  all binary strings.

A language is regular if a DFA can be formed out of it.

## Deterministic Finite Automaton (DFA/DEA)

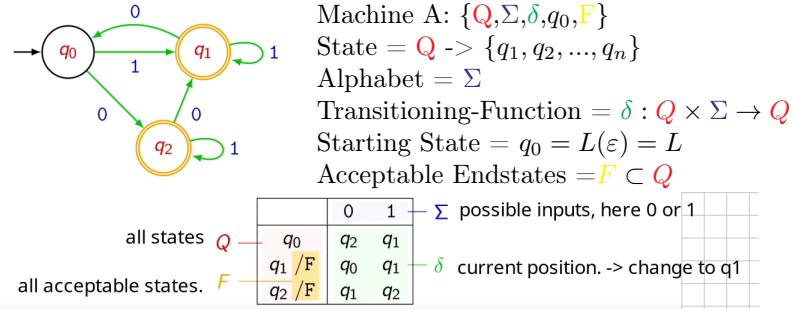


A very simple machine that accepts a variety of inputs.

Only requirement is that a D follows after T.

This means all the following inputs are valid:

(empty word!), D, DD, TD, TTTTTTTD, DDDDDTD, DDDDD, ....

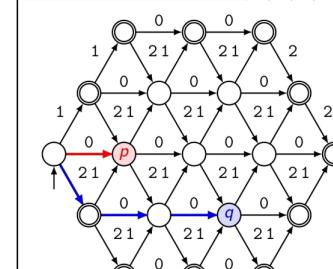
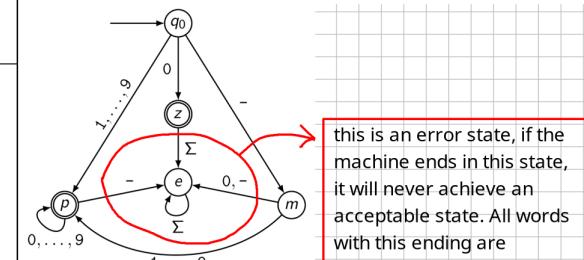


## Language of DFA A:

$$L(A) = \{w \in \Sigma^* | A \text{ accepts } w\} = \{w \in \Sigma^* | \delta(q_0, w) \in F\}$$

The language of a DFA is simply all accepted words!

## Error States in DFA



from q, many paths lead to F

This means 11, or 0 would be the "same"

$$L(q) = \{0, 10, 11, 12, \dots\}$$

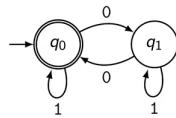
The same would obviously apply to P

## Myhill-Nerode

Adding a word to a word, to make it compatible with a language

$L(w) = \{w' | w w' \in L\}$  including:  $L(\varepsilon)$ !

w	$L(w)$	Q
$\varepsilon$	$L(\varepsilon) = L$	$q_0$
0	$L(0) = \{w \in \Sigma^*    w _0 \text{ uneven}\}$	$q_1$
1	$L(1) = \{w \in \Sigma^*    w _0 \text{ even}\} = L$	$q_0$
:	:	:



even and uneven amounts of 0s  
mod 2 zero's, 1's don't matter

## Detecting Nonregular Languages with Myhill

The examples before always had a specific amount of words/characters that one had to add, in order to accept the word.

However, there are languages that would need infinite states in order to find the entire language of a DFA

A good example for this is the language  $1^n 0^n$

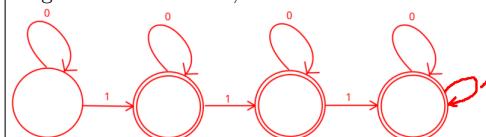
w	$L(w)$	Q
$\varepsilon$	$\{0^n 1^n   n \geq 0\}$	$q_0$
0	$\{0^n 1^{n+1}   n \geq 0\}$	$q_1$
00	$\{0^n 1^{n+2}   n \geq 0\}$	$q_2$
000	$\{0^n 1^{n+3}   n \geq 0\}$	$q_3$
$0^k$	$\{0^n 1^{n+k}   n \geq 0\}$	$q_k$
:	:	
01	$\{\varepsilon\}$	
001	$\{1\}$	
0001	$\{11\}$	
:	:	
1	$\emptyset$	e
10	$\emptyset$	e
:	:	

for every 0 that we add, we need a 1  
this means that for  $n+k$  0's we need  $k$  1's  
as  $\lim_{k \rightarrow \infty}$  we need  $\infty$  states!  
not possible with a Deterministic automaton!

also note: we have clear error states  
anything starting with 1 is an error.

## Differentiation of States

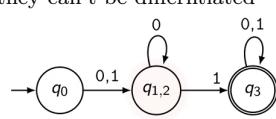
To get a minimal DFA, we eliminate all states that are superfluous.



Here the 3 acceptable states can be put together, they are the same!

	$z_0$	$z_1$	$z_2$	$z_3$
$z_0$	$\equiv$	x	x	x
$z_1$	x	$\equiv$	$\equiv$	x
$z_2$	x	$\equiv$	$\equiv$	x
$z_3$	x	x	x	$\equiv$

$z_1$  and  $z_2$  are the same!  
they can't be differentiated

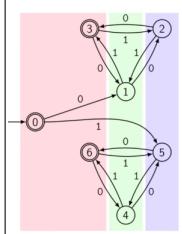


New minimal Automaton!

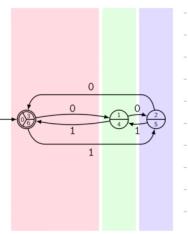
This algorithm makes it easy to see whether or not states are the same!

$$L = \{w \in \{0, 1\}^* | |w|_0 \equiv |w|_1 \pmod{3}\}$$

$$L = \{w \in \{0, 1\}^* |$$



0	1	2	3	4	5
=	=	=	=	=	=
0	=	=	=	=	=
1	=	=	=	=	=
2	=	=	=	=	=
3	=	=	=	=	=
4	=	=	=	=	=
5	=	=	=	=	=



## Beauty of a language / Pumping Lemma

a language L can be pumped if the following is valid:

$$\exists N > 0 \text{ where } w \in L \wedge |w| \geq N$$

If this word can be divided into 3 parts x,y,z while:

$$|xy| \leq N \wedge |y| > 0 \wedge |x| \geq 0 \wedge |z| \geq 0 \wedge xy^k z \in L \rightarrow \forall k \in \mathbb{N}$$

Find a number N that is bigger than 0 but less than the length of xy while the length of y is greater than 0 and  $xy^k z$  is still part of the language

Note: The power of  $xy^k z$  is NOT a power, but an indicator how many y's!!!

Any language that can be pumped is a regular language, and is therefore "schön"

Consider the following 2 examples:

$$L(s1) = \{0, 1 \text{ ending with 1}\}$$

$z = 1, xy = \text{any combination of 1 and 0}$

try:  $x=0 y=1010111 z=1, \text{True} \rightarrow N \geq 7$

try:  $x=0 y=1, \text{True} \rightarrow N \geq 1$

This can be done with any y, x

It will always satisfy all requirements of the pumping lemma.

this language is regular.

$$L(s1) = \{0^n, 1^n | n \geq 0\}$$

if  $z=1$ , more 1's than 0's, FALSE

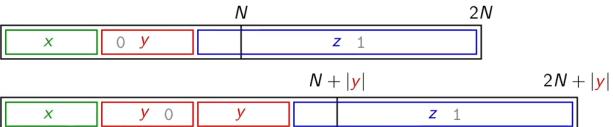
if  $x=0$ , more 0's than 1's, FALSE

This means we need  $x=0, z=0$

If we do that, then y is empty

No matter what we do, it is FALSE

$0^n 1^n$  is therefore not regular



## Pumping Lemma usage guide !!FOLLOW THIS!!

1. Claim that L is regular

2. According to pumping lemma  $\exists N$

Don't make claims about the size of N!

3. Choose a word  $w \in L | |w| \geq N$

Definition with N has to be written!

4. Division into parts according to Pumping Lemma

$$w = xyz, |xy| \leq N, |y| > 0, \text{etc}$$

5. Check if word is in language

$$\min 1 \text{ word not in language: } xy^k z \notin L | k \in \mathbb{N}$$

Explain why this word is not in the language

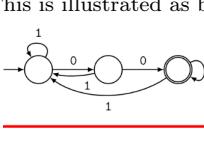
6. Contradiction, aka explain that this language is not regular.

## Non-Deterministic Finite Automaton (NFA / NEA)

Before every step was clear, there was no other determinism

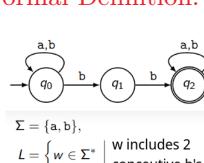
other than the input and the current state. A non-deterministic automaton Can have other things, like only accept the last 2 0's

This is illustrated as both a DFA and an NFA:



Both have the same goal, only the last 2 zero's lead to the acceptance state. However one is obviously easier, while not giving clear info on in what state it is, it is hence non-deterministic.

## Formal Definition:



Machine A:  $\{Q, \Sigma, \delta, q_0, F\}$

State  $= Q \rightarrow \{q_1, q_2, \dots, q_n\}$

Alphabet  $= \Sigma$

Transitioning-Function  $= \delta : Q \times \Sigma \rightarrow P(Q)$

Starting State  $= q_0 = L(\varepsilon) = L$

Acceptable Endstates  $= F \subset Q$

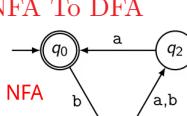
## P(Q) is the Potence Quantity!!

Note the P(Q), it means that we have more complex transitions!

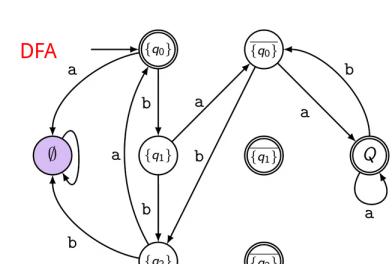
no arrow, multiple arrows for a certain character. See b in example

General tip for NFA, only write what you need to accept the word!

## NFA To DFA



NFA



DFA

The is also called a Thompson NFA

$\bar{q}_1$  is the complement Quantity. It means, any state other than  $q_1$ . Q is the full Quantity, in here the NFA can be in any state.  $\emptyset$  is the error state.

## Formal Definition of the Transition

DFA is marked with ', the regular expression is for NFA

Given  $\delta$  of an NFA and Transitions  $M \subset Q$

$$\delta' : Q \times \Sigma \rightarrow P(Q) : (M, a) \mapsto \delta'(M, a) = \bigcup_{q \in M} \delta(q, a)$$

$$Q' = P(Q)$$

$$\Sigma' = \Sigma$$

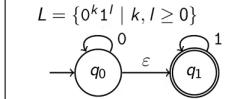
$$q'_0 = \{q_0\}$$

$$F' = \{M \in P(Q) | F \cap M \neq \emptyset\}$$

$M$  is the union of all possible endstates.

Note: a Thompson NFA needs at least 1 acceptable endstate!

## $\epsilon$ Transitions in NFA's



$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow P(Q)$$

This means that  $\epsilon$  signifies a transition without using a character!!

## Conversion from $\epsilon$ -NFA to regular NFA

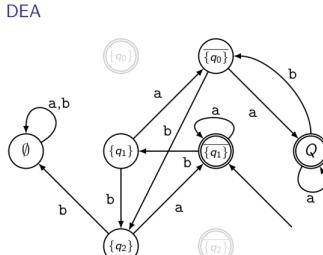
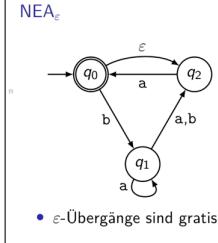
Any  $\epsilon$ -NFA can be converted to a regular NFA!

$$E(q) = \text{Quantity of all } \epsilon\text{-Transitions from } q$$

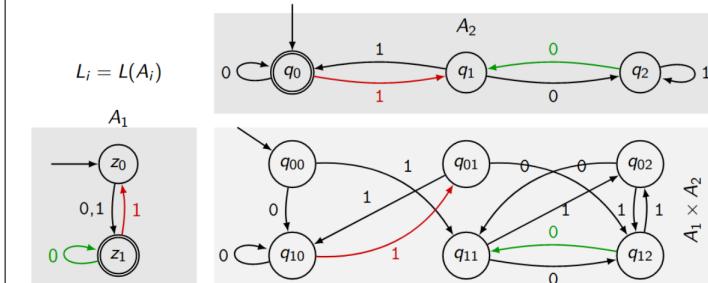
$$E(M) = \bigcup_{q \in M} E(q) \text{ Quantity of all } \epsilon\text{-Transitions}$$

$$\delta = Q \times (\Sigma \cap \{\epsilon\}) \rightarrow P(Q) : (q, a) \mapsto E(\delta(q, a))$$

## $\epsilon$ -NFA to DFA:



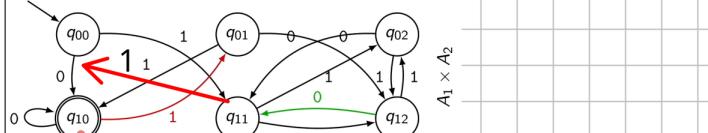
## Set-Operations with Automata



carthesian product -> acceptable endstate which both share

$$\text{Intersection } L_1 \cap L_2$$

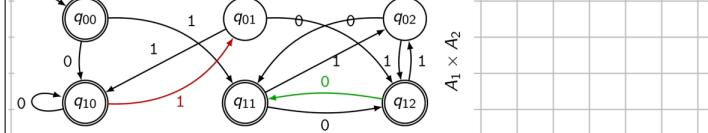
$$F = F_1 \times F_2$$



acceptable endstate of both combined

$$\text{Union } L_1 \cup L_2$$

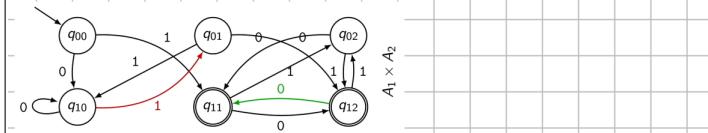
$$F = F_1 \times Q_2 \cup Q_1 \times F_2$$



acceptable endstate of one without the ones from the other

$$\text{Difference } L_1 \setminus L_2$$

$$F = F_1 \times (Q_2 \setminus F_2)$$



## Pump-able, but not regular

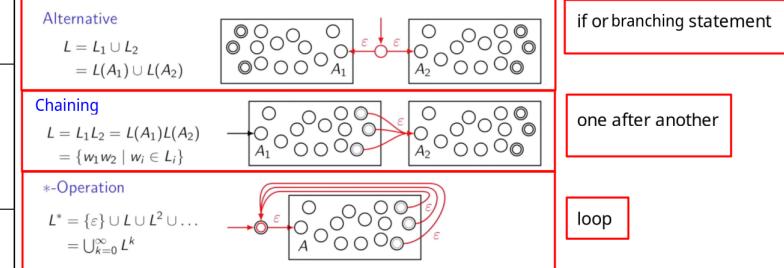
$$L = \{a^i b^j c^k \mid i = 0 \vee j = k\} = \underbrace{\{b^j c^k \mid j, k \geq 0\}}_{= L_1} \cup \underbrace{\{a^i b^j c^k \mid i > 0 \wedge j = k\}}_{= L_2}$$

The first part L1 is regular, but the second isn't

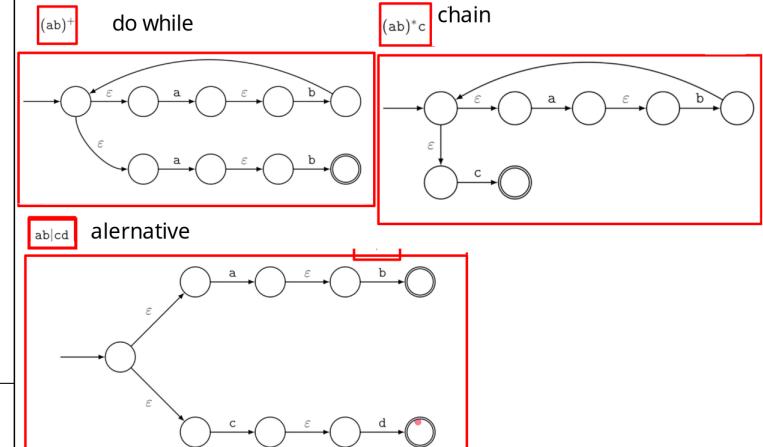
The only way we can figure that out is by using the Myhill method!

Because L2 is not regular, the composite Language L is not regular

## Regular Operations



## Regular Expressions



## Regular expression

String r to describe a language  
 $L = L(r)$

## Primitive regular expressions

expressions for Words  
with length  $\leq 1$

$L = L(r)$	$r$	NEA
$\emptyset$	$\emptyset$	
$\{\epsilon\}$	$\epsilon$	
$\{a\}$	$a$	
$\{o, s, t\}$	[ost]	
$\{a, b, \dots, s\}$	[a-s]	
$\Sigma$	.	

## Regular Operations

$$L(r_1) \cup L(r_2) = L(r_1 | r_2) \quad \text{Alternative}$$

$$L(r_1)L(r_2) = L(r_1 r_2) \quad \text{Chaining}$$

$$L(r_1)^* = L(r_1*) \quad \text{* - Operation}$$

## Contractions

$$\begin{array}{ll} r + rr^* & r? = \epsilon | r = \overline{r} \\ r\{2\} = rr & r\{2, 3\} = rr | rrr \\ r\{3\} = |r | rr | rrr & r\{3, \} = rrrr^* \end{array}$$

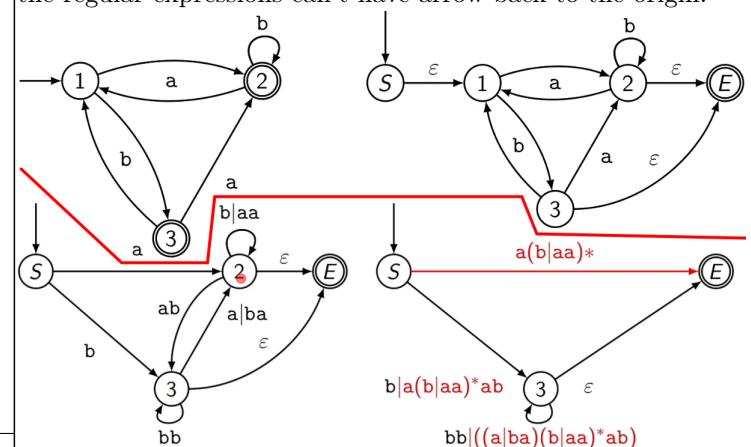
## Generalized NFA/NEA

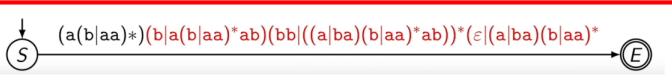
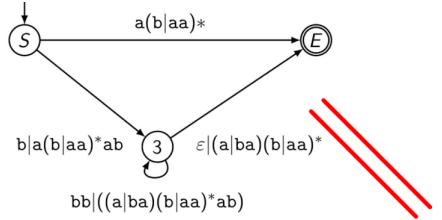
NEA<sub>ε</sub>, that has its transitions described  
NFA<sub>ε</sub>, as regular expressions

There is a DFA for every regular expression!

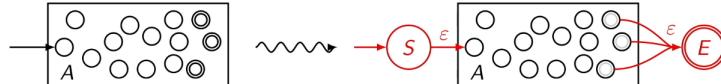
## VNEA / VNFA to regular expression

1. add a new start and stop state. This is required as the regular expressions can't have arrow back to the origin.

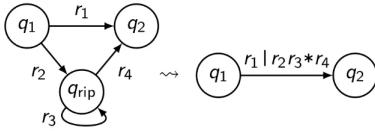




no transitions back to the starting point, and only 1 acceptable state, this simplifies the implementation

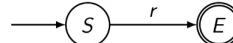


Reduction



Regular Expression

Remove all transitions until you have 1 single regular expression  $r$  in A.



## Contextfree-Grammar (CFG) » $G = (V, \Sigma, R, S)$

$V$ : Variables / Non-Terminalsymbols

$\Sigma$ : Terminalsymbols (Alphabet)

$R$ : Rules of Form  $A \rightarrow x_1 X_2 \dots x_n$

with  $A \in V, x_i \in V \cup \Sigma$

$S$ : Starvariable

Rule A  $\rightarrow w$  generated out of uAv

$uvw : uAv \implies uwv$

derivate v from u:

$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_n \Rightarrow v$  or  $u \Rightarrow^* v$

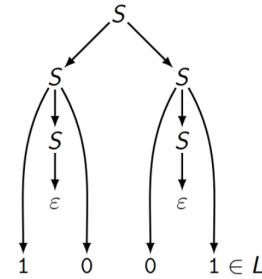
Context free grammar generated from G:

$L(G) = \{w \in \Sigma^* | S \Rightarrow^* w\}$

Parse-Tree  
 $L = \{w \in \{0, 1\}^* | |w|_0 = |w|_1\}$

with grammar:

$S \rightarrow 0S1 | 1S0 | SS | \epsilon$



Example:

Grammar  $L = \{0^n 1^n | n \geq 0\}$

Variables:  $V = \{S\}$

$\epsilon$

Terminalsymbols:  $\Sigma = \{0, 1\}$

$01 = 0\epsilon 1$

Rules:  $R = \{S \rightarrow \epsilon | 0S1\}$

$0011 = 0\ 0 1 1$

Startvariable:  $S$

## Contextfree-Grammar Term explanations

Terminalsymbols: Symbols that we can't translate further, aka they are terminal.

CFL = Contextfree-Language

$\Rightarrow^*$  means can be derived from

$uAv \Rightarrow^* uwv$ : all words that can be made with uAv

The goal of this grammar is to only have terminal symbols in the end

Aka we remove the variables one by one, via these rules

## Definition of Context-free

The context is only based on the input variable.

Take A as a variable, if it can be parsed without regard for

what is to the left or right of it, then it has a context free rule

$A \rightarrow$  something

A grammar with only variables like A is considered context free.

Rules without context:      Rules with context:

$$\begin{array}{ll} S \rightarrow A \mid C & S \rightarrow C \mid aC \mid bC \\ A \rightarrow a & aC \rightarrow A \\ A \rightarrow aAb & bC \rightarrow B \\ C \rightarrow bA \end{array}$$

This means that something like:  $S = aAb$

Turns into this:  $S \rightarrow aAb \rightarrow aab$

The A can be replaced without regard for context,  
S is therefore context free.

## Multiple Contextfree-Grammars

Context-free grammars can be interpreted in multiple ways:

This means there is no definite way to interpret something like:

$0^n 1^n$

$L = \{w \in \Sigma^* | |w|_0 = |w|_1\}$

$G_1:$

$S \rightarrow \epsilon$

$\rightarrow SS$

$\rightarrow 0S1$

$\rightarrow 1S0$

$G_2:$

$S \rightarrow SB | \epsilon$

$B \rightarrow N | E$

$N \rightarrow NN | 0N1 | 01$

$L(N) = \{w | w \text{ 0-1-expression}\}$

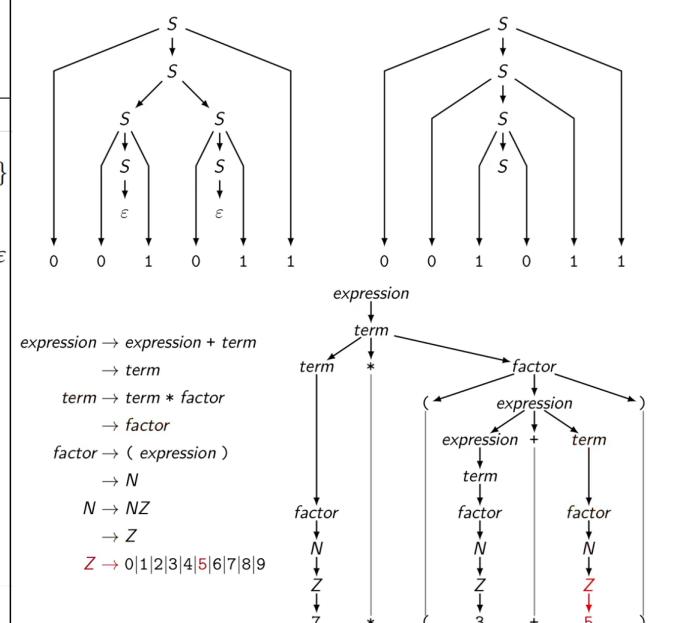
$E \rightarrow EE | 1E0 | 10$

$L(E) = \{w | w \text{ 1-0-expression}\}$

The grammar  $G_2$  gives us a clear Parse-tree, the first one doesn't. The reason for this is the obvious Haskell-like structure on the right. The one on the left has no idea of leafs and nodes.

Another example with parse trees:

$S \rightarrow 0S1 | 1S0 | SS | \epsilon$



## Facts about Contextfree-Grammar:

- Contextfree-Grammars can't be compared!
- Only the  $L_1 \cup L_2$  Quantity operation can be done On ANY Contextfree-Grammar
- The amount of steps to derive a word is not always clear Some Grammars allow this, some don't
- the class of regular languages is complete with regular operations

## Regular Operations on Contextfree-Grammar

Grammar for Regular Operations

$L_1$  and  $L_2$  context free languages with

Grammars  $G_i = (V_i, \Sigma, R_i, S_i)$ .

• new starting variable  $S_0$

•  $V = V_1 \cup V_2 \cup \{S_0\}$

• extended rules  $R$

$\Rightarrow G = (V, \Sigma, R, S_0)$

Alternative

Rules for  $L_1 \cup L_2$ :

$R = R_1 \cup R_2 \cup \{S_0 \rightarrow S_1, S_0 \rightarrow S_2\}$

Chaining

Rules for  $L_1 L_2$ :

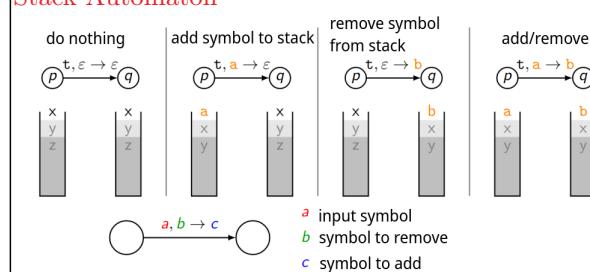
$R = R_1 \cup R_2 \cup \{S_0 \rightarrow S_1 S_2\}$

\*-Operation

Rules for  $L_1^*$ :

$R = R_1 \cup \{S_0 \rightarrow S_0 S_1, S_0 \rightarrow \epsilon\}$

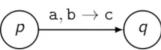
## Stack Automaton



## Definition

Stack-Automaton  $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$  input a, remove b, add c

①  $Q$ : States



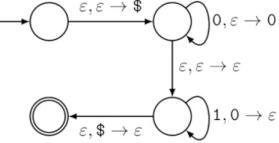
②  $\Sigma$ : Input Alphabet

③  $\Gamma$ : Stack-Alphabet

④  $\delta: Q \times \Sigma \times \Gamma \rightarrow P(Q \times \Gamma)$

Stack Automaton

⑤  $q_0 \in Q$ : Start-State



⑥  $F \subset Q$ : Accepting-State

$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ ,  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$

Note:

- always an NFA, never deterministic!
- $\Gamma \neq \Sigma$  possible! you can have different symbols in your stack!

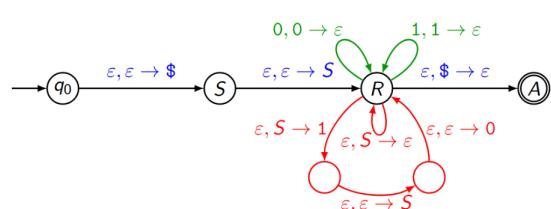
## Grammar

$S \rightarrow 0S1$

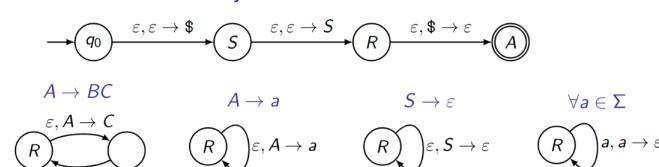
$\rightarrow \epsilon$

## Schema of the Automaton

1.  $\$$  and Startvariable  $S$
2. Transition for every Rule
3. Transition for every Terminalsymbol



## Grammar in CNF, Chomskynormalform -> Stack-Automaton



if L is a Contextfree-language, then an automaton P with  $L = L(P)$  exists.  
aka this automaton accepts the language L.

## Chomsky-Normalform

This is used to make sure, that we don't have needless Variables

### 1. Unit-rules

$$\left. \begin{array}{l} A \rightarrow B \\ B \rightarrow A \\ A \rightarrow a \end{array} \right\} \Rightarrow A \rightarrow B \rightarrow A \xrightarrow{\text{can be infinite!!}} B \rightarrow \dots \rightarrow A \rightarrow a$$

### 2. endless empty Variables

$$\left. \begin{array}{l} A \rightarrow ABCDE | a \\ B \rightarrow \epsilon \\ C \rightarrow \epsilon \\ D \rightarrow \epsilon \\ E \rightarrow \epsilon \end{array} \right\} \Rightarrow A \rightarrow ABCDE \rightarrow ABCD \rightarrow ABC \rightarrow AB \rightarrow A \rightarrow a$$

same here !!

In order to solve this, we need to follow these rules:

1. No Unit Rules:  $A \rightarrow B$
2. No Rules like:  $A \rightarrow \epsilon$  unless necessary:  $S \rightarrow \epsilon$
3. Right side has exactly 2 Variables, or 1 terminalsymbol.  
Everything else is forbidden on the right side.

Note this includes ANY other formation, including  $aB, aa, ABC, aAB$

## Transforming into Chomsky-Normalform

### Transformation into Chomsky-Normalform

#### 1. new starting variable $S_0 \rightarrow S$

$$\left. \begin{array}{l} A \rightarrow \epsilon \\ B \rightarrow AC \end{array} \right\} \Rightarrow A \text{ is pointless, as it leads to } \epsilon. \quad \Rightarrow \left\{ \begin{array}{l} B \rightarrow AC \\ \epsilon \rightarrow \epsilon \end{array} \right.$$

$$\left. \begin{array}{l} A \rightarrow B \\ B \rightarrow CD \end{array} \right\} \Rightarrow A \text{ leads to } B, \text{ which simply leads to } CD. \text{ Replace } B \text{ with } CD \text{ in } A. \quad \Rightarrow \left\{ \begin{array}{l} A \rightarrow CD \\ B \rightarrow CD \end{array} \right.$$

$$\left. \begin{array}{l} \text{Chaining: } A \rightarrow u_1 u_2 \dots u_n \\ A \rightarrow u_1 A_1, A_1 \rightarrow u_2 A_2, \dots, A_{n-2} \rightarrow u_{n-1} u_n \text{ and if } u_i = \text{terminalsymbol:} \\ A_{i-1} \rightarrow U_i A_i, U_i \rightarrow u_i. \end{array} \right.$$

In-depth explanation:

-rule 2a  $\rightarrow \epsilon$  only for startvariable

$S \rightarrow ASA|abB$

$A \rightarrow B|IS$

$B \rightarrow b|\epsilon$

here is how chomsky would look:

$S_0 \rightarrow ASA|abB$

$A \rightarrow B|IS$

$B \rightarrow b$

essentially, we moved the useless  $\epsilon$  in  $B$  up, so that  $B$  is now always  $b$ .

- rule 2a  $\rightarrow \epsilon$  only from startvariable  $S_0$

$$\left. \begin{array}{l} S_0 \rightarrow S \\ S \rightarrow ASA|abB \\ A \rightarrow B|IS \\ B \rightarrow b \end{array} \right\} \Rightarrow S \rightarrow ASA|AS|SA|abB|a$$

$\epsilon$  now we have  $\epsilon$  here....

- rule 3a  $\rightarrow$  no unit rules  $\rightarrow S$  can be skipped  $\rightarrow$  no single defined variable

$$S_0 \rightarrow S \Rightarrow ASA|AS|SA|abB|a$$

$$S \rightarrow ASA|AS|SA|abB|a$$

$$A \rightarrow B|IS \Rightarrow A \rightarrow ASA|AS|SA|abB|a|B$$

$$B \rightarrow b \Rightarrow B \rightarrow b$$

- rule 3a  $\rightarrow$  no unit rules  $\rightarrow B$  can be skipped  $\rightarrow$  no single defined variable

$$S_0 \rightarrow ASA|AS|SA|abB|a \Rightarrow ASA|AS|SA|abB|a$$

$$S \rightarrow ASA|AS|SA|abB|a \Rightarrow ASA|AS|SA|abB|a$$

$$A \rightarrow ASA|AS|SA|abB|a|B \Rightarrow ASA|AS|SA|abB|a|b$$

$$B \rightarrow b \Rightarrow B \rightarrow b$$

- rule 3b  $\rightarrow$  no more than 2 variables on the right

$$S_0 \rightarrow ASA|AS|SA|abB|a \Rightarrow AA_1|SA|AS|laB|a$$

$$S \rightarrow ASA|AS|SA|abB|a \Rightarrow AA_1|SA|AS|laB|a$$

$$A \rightarrow ASA|AS|SA|abB|a|b \Rightarrow AA_1|SA|AS|laB|alb$$

$$B \rightarrow b \Rightarrow B \rightarrow b$$

$$A_1 \rightarrow SA$$

- rule 4b  $\rightarrow$  no terminal symbol next to variable

$$S_0 \rightarrow AA_1|SA|AS|laB|a \Rightarrow AA_1|SA|AS|laB|a$$

$$S \rightarrow AA_1|SA|AS|laB|a \Rightarrow AA_1|SA|AS|laB|a$$

$$A \rightarrow AA_1|SA|AS|laB|alb \Rightarrow AA_1|SA|AS|laB|alb$$

$$B \rightarrow b \Rightarrow B \rightarrow b$$

$$A_1 \rightarrow SA$$

$$chomsky form$$

## Usage of Chomsky-Normalform Grammar G

The derivation of a word  $w \in L(G)$  is always possible within  $2 * |w| - 1$  Rule applications.

Reason:

1. Apply  $|w| - 1$  Rules of form  $A \rightarrow BC$  to generate a word with length  $|w|$  out of  $S$ .

2. Apply  $|w|$  Rules of form  $A \rightarrow a$  to generate the word  $w$

This is a total of  $2 * |w| - 1$  Rule applications

## Deterministic Parsing CYK-Algorithm

Deterministic Parsing is used to check if a word can be derived from either  $S$ , aka the entire Grammar  $S \Rightarrow^* w$ ,

or just a variable of it for example  $A$ .  $S \Rightarrow^* w | A \in V$

This is especially easy with the Chomsky-Normalform!!

Given: Word  $w \in \Sigma^*$  Variable  $A \in V$  Grammar  $G = (V, \Sigma, R, S)$

$A \xrightarrow{*} w$  possible??

$$\triangleright w = \epsilon: \quad A \xrightarrow{*} \epsilon \Leftrightarrow A \rightarrow \epsilon \in R$$

$$\triangleright |w| = 1: \quad A \xrightarrow{*} w \Leftrightarrow A \rightarrow w \in R$$

$$\triangleright |w| > 1: \text{divide and conquer} \quad A \xrightarrow{*} w \Rightarrow \exists \begin{cases} A \rightarrow BC \in R \\ w = w_1 w_2 \quad w_i \in \Sigma^* \end{cases} \text{ with } \begin{cases} B \xrightarrow{*} w_1 \\ C \xrightarrow{*} w_2 \end{cases}$$

Wort: () []

$$S \rightarrow AB | CD | AT | CU | SS$$

$$T \rightarrow SB$$

$$U \rightarrow SD$$

$$A \rightarrow ($$

$$B \rightarrow )$$

$$C \rightarrow [$$

$$D \rightarrow ]$$

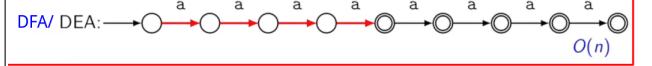
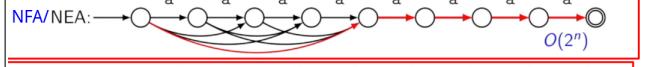
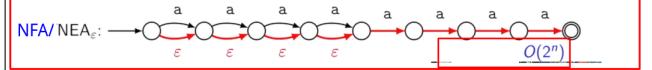
okay this works, but what is the catch?  
Well O(n³) ... hahaha .....

	(	)	[	]	(	)	[	]
0	{A}	{B}	{C}	{A}	{B}	{D}		
1	{S}	{}	{}	{S}	{}			
2	{}	{}	{}	{}	{}			
3	{}	{}	{S}					
4	{}	{}						
5	{S}							

## Performance Comparison NFA/DFA

Regex-Stressest:  $a? a? a? a? aaaa$

Runtime/Complexity of accepting the word aaaa



So what is this? Essentially, because we removed the variables and possibilities, we now have a runtime or complexity of  $O(n)$  instead of something with an exponential increase.

## Standardization of Stack Automatons (PDA to CFG)

PDA, Push-Down-Automaton = Stack Automaton

The idea is to check whether or not Contextfree-Grammar can be expressed using only regular expressions.

One way of trying this is to create a Stack Automaton, that is based on such expressions.

first need a Variable that handles words from input to stack  $A_{pq}$

Then we need rules for this  $A_{pq} \rightarrow A_{pr}A_{rq}$  from p to r and from r to q.

Note that  $A_{pq}$  signifies the amount of words that lead from p to q

However, this is only the case because it is implied in the name, it is otherwise just a Non-Terminalsymbol!!!

Turning the regular Stack Automaton to a CFG:

1. only 1 accepting state, q will be degraded.

all previous accepting states now lead to  $q_a$ :

2. on input and output we need to add and remove the \$ symbol to signify an empty stack



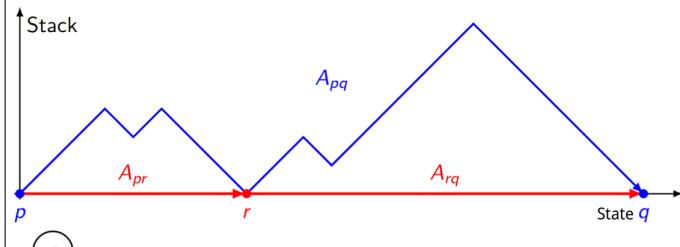
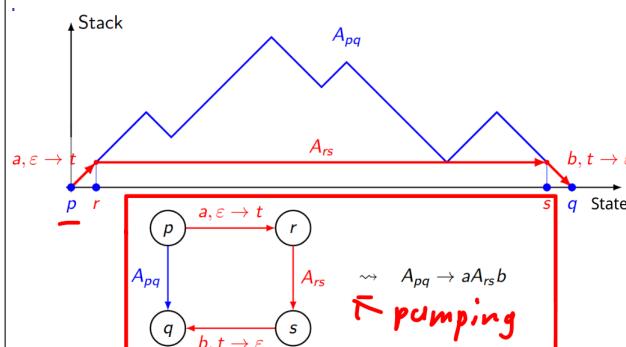
3. now all transitions need to involve the stack



This means that adding or removing need to be different operations.

It also means, that no symbol leads to the stack adding and removing a placeholder.

Here this is t.



## Derived Rules from the Automaton:

standardized Grammar with starting state/accepting state:  $q_0/F = \{q_a\}$ .

① Startvariable:  $A_{q_0,q_a}$

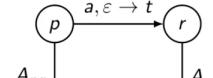
② Rules :



$A_{pp} \rightarrow \epsilon$



$A_{pq} \rightarrow \epsilon$



$A_{rs} \rightarrow b, t \rightarrow \epsilon$

$A_{pq} \rightarrow aA_{rs}b$



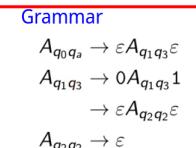
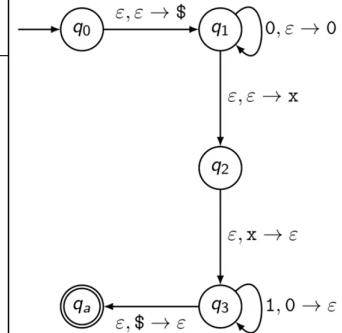
$A_{pr} \rightarrow \epsilon$



$A_{rq} \rightarrow A_{pq}$

$A_{pq} \rightarrow A_{pr}A_{rq}$

## PDA → CFG: example $1^n 0^n 1^n$



Simplification, and Chomsky compliant!!

$S \rightarrow 0S1$   
 $\rightarrow \epsilon$

Grammar for:  $L = \{0^n 1^n \mid n \geq 0\}$

every CFG can be converted to a Stack Automaton.

## Regularity Check for CFG / Pumping Lemma for CFG

Grammar G in CNF

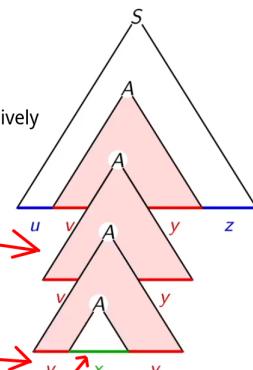
$w \in L(G) \Rightarrow S \xrightarrow{*} w$

Reusable Variable, RECURSIVE Variable

$|w| \geq N$  considering N big enough, we can recursively use variables.

In this case, the lowest variable is A, so we use this one to recursively call.

$A \xrightarrow{*} vxy$   
 $A \xrightarrow{*} x$



Pumping

$A \xrightarrow{*} vxy$  instead of simply  $A \xrightarrow{*} x$

## Pumping Lemma for CFL

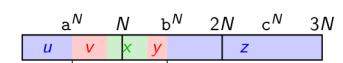
if L is a CFL, then there is a Number N (Pumping Length) for which:

each word  $w \in L$  with  $|w| \geq N$  can be divided into  $w = uvxyz$

1.  $|vy| > 0$
2.  $|vxy| \leq N$
3.  $uv^kxy^kz \in L \forall k \in \mathbb{N}$
4. While pumping, the amount of a and b increases, the amount of c doesn't  
 $\Rightarrow uv^kxy^kz \notin L \forall k \neq 1$

Example:  $\{a^n b^n c^n \mid n \geq 0\}$

1. Claim :  $L = \text{CFL}$
2. Pumping length  $N$
3. Word :  $w = a^N b^N c^N$
4. Division :



6. Contradiction! L is not a CFL!!

increasing a and b leads to c being smaller!

increasing b and c leads to a being smaller!

even if you increase ALL of them, you still have aa... (your pump)...bb which is NOT correct!

## Bakus-Naur Form

-Naur is a machine specification for Rules of a CFG.

- Variables:  $\langle \text{variable-name} \rangle$

- single Symbols: A

- Strings: 'Example'

- Rules:  $\langle \text{variable-name} \rangle ::= \text{expression}$

Haskell says hello!

Expressions are a series of variables, single symbols or strings, separated by '|'

$\langle \text{expression} \rangle ::= \langle \text{expression} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle$

$\langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$

$\langle \text{factor} \rangle ::= (\langle \text{expression} \rangle) \mid \langle \text{number} \rangle$

$\langle \text{number} \rangle ::= \langle \text{number} \rangle \langle \text{digit} \rangle \mid \langle \text{digit} \rangle$

$\langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4$

$\mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

## Extended-Backus-Naur-Form EBNF

- Literals: "" or "
- Rules are annotated with = instead of ::=
- Variables without <> and can include symbols
- comma for chaining: term, "/\*", expression
- semicolon for end, just like cpp: ;
- Comments: (\* ... \*)
- Optional recursion: {...}
- Option: [] - Groups: (...)
- Exception: - (without, not error handling)

**expression** = **expression**, "+", **term** | **term** ;  
**term** = **term**, "\*", **factor**, | **factor** ;  
**factor** = "(", **expression**, ")" | **number** ;  
**number** = **digit**, { **digit** } ;  
**digit** = "0" | "1" | "2" | "3" | "4" |  
        "5" | "6" | "7" | "8" | "9" ;

Both of these Naur implementations do of course resemble this:

$$\begin{array}{c} \text{expr} \rightarrow \text{term expr}' \quad \text{term} \rightarrow \text{factor term}' \quad \text{factor} \rightarrow (\text{expr}) \\ \text{expr}' \rightarrow + \text{term expr}' \quad \text{term}' \rightarrow * \text{factor term}' \quad \rightarrow \text{integer} \\ \rightarrow \varepsilon \quad \quad \quad \rightarrow \varepsilon \end{array}$$

One small problem with all this, these specifications change often

So even EBNF is likely soon to be out-of-date....

**Note, EBNF does NOT save you from misimplementing things!**

```
<expression> ::= <numeric_expression>
...
<numeric_expression> ::=
  ( ("-" | "+" | "--") <expression> )
 | ( <expression> ("+" | "--" ) )
 | ( <expression> ("+" | "-" | "=" | "*" | "/"
    | "/" | "=" | "%" | "%=") <expression> ).
```

Wrong implementation :)

Without operator prioritization, + would be calculated before the \*.

This means that  $5 + 6 * 2$  would evaluate to 22 instead of 17.

aka this implementation considers this to be:  $(5 + 6) * 2$

instead of  $5 + (6 * 2)$

In this picture the Operator-prioritization was not implemented!

This means this implementation is not correct, it calculates trash!

## The stack problem

The stack has enabled us to understand the Language  $0^n 1^n$

However the language  $a^n b^n c^n$  remains uninterpretable.

This can be solved by extending the stack.

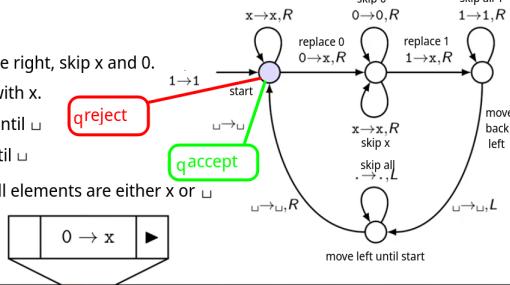
One idea is to double the stack, giving us access to every element  
The only problem is that this is a slow and tedious process.

**This is where a turing machine comes into play with its band!!**

## Turing Machine (TM)

### Pseudocode

- 1 replace 0 with x
- 2 when moving to the right, skip x and 0.
- 3 replace the first 1 with x.
- 4 move to the right until  $\square$
- 5 move to the left until  $\square$
- 6 go to step 1 until all elements are either x or  $\square$
7. stop at the start.



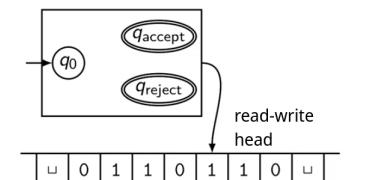
...	□	□	□	x	0	0	x	1	1	□	□	...
-----	---	---	---	---	---	---	---	---	---	---	---	-----

## Definition Of Deterministic Turing Machine

### Turing Machine

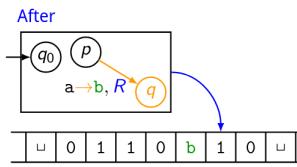
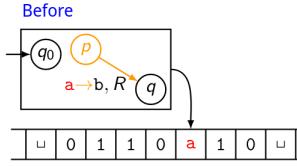
$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$

- $Q$  : States
- $\Sigma$  : Alphabet
- $\Gamma$  : Band-Alphabet:  $\sqcup \in \Gamma \setminus \Sigma$
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
- $q_0 \in Q$ : Startstate
- $q_{\text{accept}} \in Q$ : accepting state
- $q_{\text{reject}} \in Q$ : rejecting state



- Transition possible if  $a$  under read-write head
- Current field of read-write head is overwritten with  $b$
- movement head: left  $L$ , right  $R$

**State Diagram:**  
 $\delta(p, a) = (q, b, L)$ :



**Programstructure**

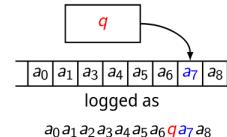
- ▶ Input-Word on Band
- ▶ Read-Write Head positioned on 1. element
- ▶ start Machine, complete  $t(w)$  transitions
- ▶ Machine stops in  $q_{\text{accept}}$  or  $q_{\text{reject}}$

**M a Turing Machine**

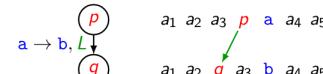
$$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$$

### Calculation History

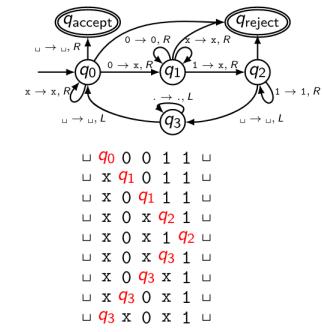
#### Notation for Machine State



#### Transition



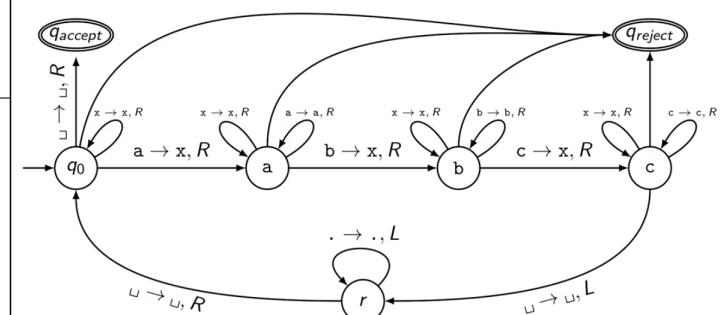
#### Protocol of a Calculation



## $a^n b^n c^n$

Before we asked the question of how we could create a Turing Machine, that understands the language  $a^n b^n c^n$

Is this already possible? YES!

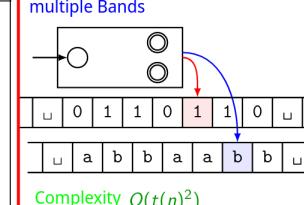


## Turing Machine Variants

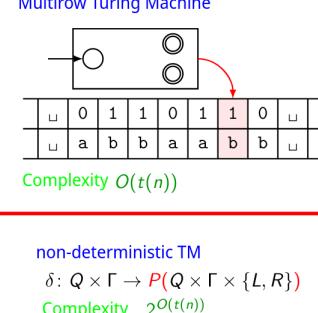
### different Alphabet

$\Gamma$  Binary encoded Bandalphabet  
 $\Gamma_0 = \{0, 1\}$   
**Complexity**  $O(t(n))$

### multiple Bands



### Multirow Turing Machine



**non-deterministic TM**  
 $\delta : Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\})$   
**Complexity**  $2^{O(t(n)^2)}$

## Nondeterministic TM Transitioning Function

on each step max N different possibilities  
 $\delta: Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\})$

$\Rightarrow$  multiple directions of movement

### accept Word

$w \in L(M)$

a word is acceptable if there is at least one possible way to accept it. The rest do not matter!

### Simulation Idea:

Try all possibilities  $\max N^t(n)$

### Simulation on standard TM

- use 3 bands
- 1. Working Band
- 2. copy of w (word)
- 3. A list of all sequences of possibilities

### Simulation

1. copy word to band 1
2. execute TM on band 1 with possibilities of band 3
3. start again at step 1 and choose the next possibility from band 3

Complexity:  $O(N^{t(n)}) = 2^{O(t(n))}$

## Counter for TM

A language of a Turing Machine is countable

It essentially allows you to see all possible words.

### Definition

A counter is a TM which allows you to print words

### recursively countable Language

L is recursive countable if there is a counter that counts it.

Countable Language == Turing Language

## Definitions for Computability (Entscheidbarkeit)

### Decider

### Definition

A Decider is a Turing-Machine that stops on ANY input

### Definition

A Language L is computable, if there is a Decider M with  $L = L(M)$ . M decides L.

### Language-Problem

Every Problem P can be turned into a Language Problem

$$L_P = \left\{ w \in \Sigma^* \mid w \text{ is the solution of Problem } P \right\}$$

### Examples:

#### Empty-Problem :

$$E_{DEA} = \left\{ \langle A \rangle \mid A \text{ a DFA, and } L(A) = \emptyset \right\}$$

#### Equality Problem :

$$EQ_{CFG} = \left\{ \langle G_1, G_2 \rangle \mid G_i \text{ CFGs and } L(G_1) = L(G_2) \right\}$$

#### Acceptance Problem :

$$A_{DEA} = \left\{ \langle A, w \rangle \mid A \text{ a DFA that accepts } w \right\}$$

#### Halting Problem:

$$HALT_{TM} = \left\{ \langle M, w \rangle \mid M \text{ stops at Input } w \right\}$$

## There are clear definite problems that we can solve:

### Language Problem Calc.

find the solution of the quadratic formula

$$x^2 - x - 1 = 0$$

a b c and x are decimals, aka numbers

$$w = a=a, b=b, c=c, x=x$$

Computability problem: is this a solution? Yes? No?

$$a=1, b=-1, c=-1, x=3 \in L?$$

### $\varepsilon$ acceptance Problem

can the Machine accept the empty word?

$$L = \{\langle A \rangle \mid \varepsilon \in L(A)\}$$

### Compute-Algorithm

1. Turn A into a DFA
2. is the startstate a accepting state?

3. does the Turing Machine stop in qaccept?

### Language Problem 1 word

can the Machine accept the word w?

$$L = \{\langle A, w \rangle \mid w \in L(A)\}$$

### Compute-Algorithm

1. turn A into a DFA
2. Simulate A with a Turing Machine
3. does the Turing Machine stop in qaccept?

Given n calculate the first 10 decimals of its square root

L is the language of strings of form n,x where n is a decimalform of a natural number and x is the first 10 decimals of the root

Computability is this a solution? Yes? No?

$$2,1.41213562 \in L?$$

or  $\langle n, x \rangle \rightarrow \langle A \rangle, \in L?$

Do Machine A1 and A2 accept the same language?

$$L(A_1) = L(A_2)?$$

$$L = \{\langle A_1, A_2 \rangle \mid L(A_1) = L(A_2)\}$$

### Computability

1. Turn A1 into a minimal Automaton A'1.
2. Turn A2 into a minimal Automaton A'2.
3. is A'1 = A'2?

Can the word w be produced from the grammar G?

$$L = \{\langle G, w \rangle \mid w \in L(G)\}$$

### Computability

1. Check if  $\langle G \rangle$  is actually a Grammar
2. Bring Grammar into CNF (Chomsky)
3. check with the CYK algorithm if w can be produced out of G.

And there are some that we can't solve:

### Theorem (Alan Turing)

$A_{TM}$  is not computable

### Proof:

Generate from a decider H for  $A_{TM}$  a machine D with Input  $\langle M \rangle$

1. let H on Input  $\langle M, \langle M \rangle \rangle$

2. if H accepts :  $q_{reject}$

3. if H rejects :  $q_{accept}$

Now use D on  $\langle D \rangle$

$D(\langle D \rangle)$  accepts  $\Leftrightarrow D$  rejects  $\langle D \rangle$

$D(\langle D \rangle)$  rejects  $\Leftrightarrow D$  accepts  $\langle D \rangle$

Contradiction!

### Special-Halting Problem

$$HALT_{\varepsilon TM} = \left\{ \langle M \rangle \mid M \text{ is a turing machine and } M \text{ stops on empty band} \right\}$$

is not computable

### Halting Problem

$$HALT_{TM} = \left\{ \langle M, w \rangle \mid M \text{ is a turing machine and } M \text{ stops on input } w \right\}$$

is not computable

Note, there is NO SUCH THING AS A DECIDER.  
It is not possible to create it.

### Reduction

#### Reductionprojection

solvable projection  $f: \Sigma^* \rightarrow \Sigma^*$

$$w \in A \Leftrightarrow f(w) \in B$$

Notation:  $f: A \leq B$ , "A easier than B"

#### Computability

B computable ,  $f: A \leq B \Rightarrow A$  computable

#### Proof

H is a decider for B, then is  $H \circ f$  a decider for A.

#### Conclusion

A not computable ,  $A \leq B \Rightarrow B$  not computable

or for humans: if B is computable then so is A, as A is the easier problem than B.

If B is NOT computable then A is NOT computable.

#### Reduction for: $A_{TM} \leq HALT_{\varepsilon TM}$

$A_{TM}$  is not computable

is  $HALT_{\varepsilon TM}$  computable?

Does the machine M accept the word w?

There is no Decider

$$\langle M, w \rangle$$

$\mapsto$

Program S

1. run M on inputword w
2. M stops on qaccept: accept
3. M stops on qreject: endless loop

M accepts w

$\Leftrightarrow$

S stops

#### Decider for $A_{TM}$

1. Create program S
2. Use H on (S)

#### Decider for $HALT_{\varepsilon TM}$

If H is a decider for  $HALT_{\varepsilon TM}$  then we could create a decider for  $A_{TM}$

The same strategy can be used on many more examples.

Due to time constraints I did not translate these:

#### Vergleich der Entscheidbarkeit von Sprachen

##### Reduktion $A_{TM} \leq HALT_{\varepsilon TM}$

Die Sprache A ist auf B reduzierbar,  $A \leq B$ , wenn es eine berechenbare Abbildung  $f: \Sigma^* \rightarrow \Sigma^*$  gibt mit  $w \in A \Leftrightarrow f(w) \in B$

Lesung: A  $\leq$  B heißt A ist leichter entscheidbar als B.

Theorem

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

$A \leq B, B$  entscheidbar, dann ist A entscheidbar

$A \leq B, B$  nicht entscheidbar, dann ist A nicht entscheidbar

## The law of Rice

### Law of Rice

#### Properties of a language

(properties of a turing language)

$\text{REGULAR}$      $L(M)$  is regular

$E$                  $L(M)$  is empty

#### Definition

A property is nontrivial if there are 2 Turing machines that handle the same language, with one of them having the property and the other not.

$L(M_1)$  has  $P$

$L(M_2)$  doesn't have  $P$

**Reduction**  $P_{\text{TM}}$ :  $A_{\text{TM}} \leq P_{\text{TM}}$

$A_{\text{TM}}$  is not computable    is  $P_{\text{TM}}$  computable?

Does the machine M accept the word w?

**Satz von Rice**  
if  $P$  is a nontrivial property of a turing language, then

$P_{\text{TM}} = \{\langle M \rangle \mid L(M) \text{ has property } P\}$   
not computable

#### Conclusion:

It is not possible to see if a program has a nontrivial property or not!

$\langle M, w \rangle$

$\mapsto$  Programm  $S$  mit Input  $w$

- 1. Program  $T$  accepts  $w$ :  $q_{\text{accept}}$
- 2. Run M on  $w$
- 3. M accepts  $w$ :  $q_{\text{accept}}$
- 4.  $q_{\text{reject}}$

$M$  accepts  $w$

$\Leftrightarrow S$  accepts  $\Sigma^*$ , has  $P$

$M$  rejects  $w$

$\Leftrightarrow S$  accepts  $L(T)$ , doesn't have  $P$

## Overview of Computability

Problem	Word	Prerequisite		Computing algorithm / Reason
$E_{\text{DEA}}$	$\langle A \rangle$	$L(A) = \emptyset$	yes	minimal-automaton doesn't have accept state
$E_{\text{CFG}}$	$\langle G \rangle$	$L(G) = \emptyset$	yes	Chomsky-Normalform
$E_{\text{TM}}$	$\langle M \rangle$	$L(M) = \emptyset$	no	
$EQ_{\text{DEA}}$	$\langle A_1, A_2 \rangle$	$L(A_1) = L(A_2)$	yes	comparison of minimal Automatons
$EQ_{\text{CFG}}$	$\langle G_1, G_2 \rangle$	$L(G_1) = L(G_2)$	no	
$EQ_{\text{TM}}$	$\langle M_1, M_2 \rangle$	$L(M_1) = L(M_2)$	no	
$A_{\text{DEA}}$	$\langle A, w \rangle$	$w \in L(A)$	yes	Regex-Engines simulate any DEAs with any input word $w$
$A_{\text{CFG}}$	$\langle G, w \rangle$	$w \in L(G)$	yes	Cocke-Younger-Kasami algorithm
$A_{\text{TM}}$	$\langle M, w \rangle$	$w \in L(M)$	no	Halting-Problem