

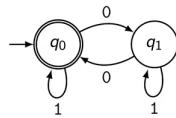


## Myhill-Nerode

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

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

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  $1^n 0^n$

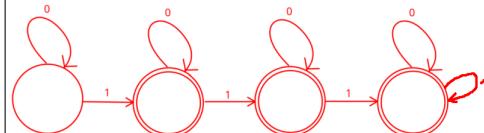
w	L(w)	Q
$\varepsilon$	$\{0^n 1^n   n \geq 0\}$	$q_0$
0	$\{0^n 1^{n+1}   n \geq 0\}$	$q_1$
00	$\{0^n 1^{n+2}   n \geq 0\}$	$q_2$
000	$\{0^n 1^{n+3}   n \geq 0\}$	$q_3$
$0^k$	$\{0^n 1^{n+k}   n \geq 0\}$	$q_k$
:	:	
01	$\{\varepsilon\}$	
001	$\{1\}$	
0001	$\{11\}$	
:	:	
1	$\emptyset$	e
10	$\emptyset$	e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

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

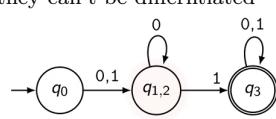
To get a minimal DFA, we eliminate all states that are superfluous.



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

	$z_0$	$z_1$	$z_2$	$z_3$
$z_0$	≡	x	x	x
$z_1$	x	≡	≡	x
$z_2$	x	≡	≡	x
$z_3$	x	x	x	≡

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

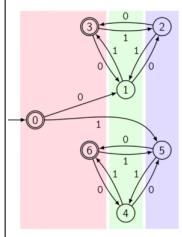


New minimal Automaton!

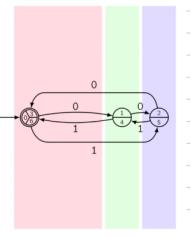
This algorithm makes it easy to see whether or not states are the same!

$$L = \{w \in \{0, 1\}^* | |w|_0 \equiv |w|_1 \pmod{2}\}$$

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



0	1	2	3	4	5	6
≡	≡	≡	≡	≡	≡	≡
0	≡	≡	≡	≡	≡	≡
1	≡	≡	≡	≡	≡	≡
2	≡	≡	≡	≡	≡	≡
3	≡	≡	≡	≡	≡	≡
4	≡	≡	≡	≡	≡	≡
5	≡	≡	≡	≡	≡	≡
6	≡	≡	≡	≡	≡	≡



## 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:  $x=0$   $y=1010111$   $z=1$ , True  $\rightarrow N \geq 7$

try:  $x=0$   $y=1$ , 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

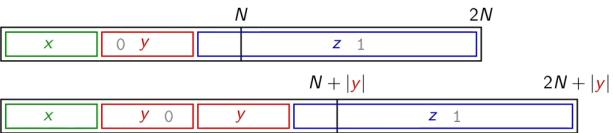
if  $x=0$ , more 0's than 1's, FALSE

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

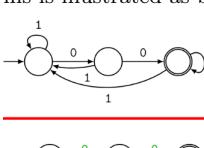
6. Contradiction, aka explain that this language is not regular.

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

Before every step was clear, there was no other determinism

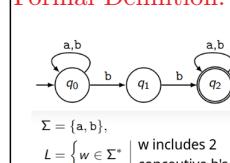
other than the input and the current state. A non-deterministic automaton Can have other things, like only accept the last 2 0's

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  $= F \subset Q$

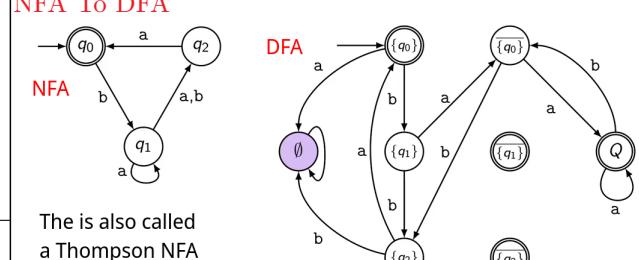
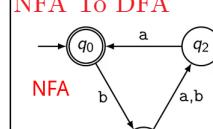
## P(Q) is the Potence Quantity!!

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



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  $M \subset Q$

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

$$Q' = P(Q)$$

$$\Sigma' = \Sigma$$

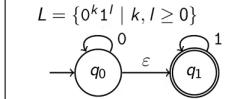
$$q'_0 = \{q_0\}$$

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

$M$  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

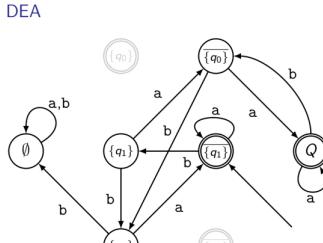
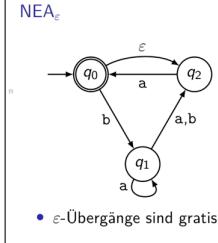
Any  $\epsilon$ -NFA can be converted to a regular NFA!

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

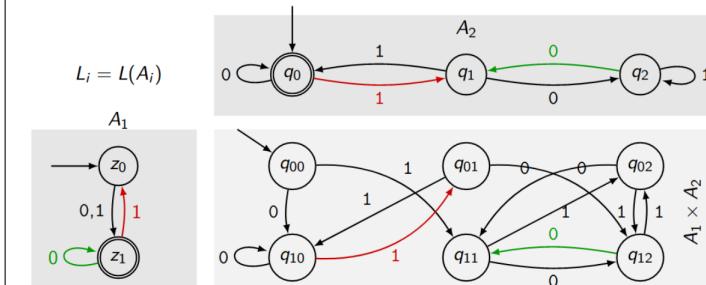
$$E(M) = \bigcup_{q \in M} E(q) \text{ Quantity of all } \epsilon\text{-Transitions}$$

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

## $\epsilon$ -NFA to DFA:



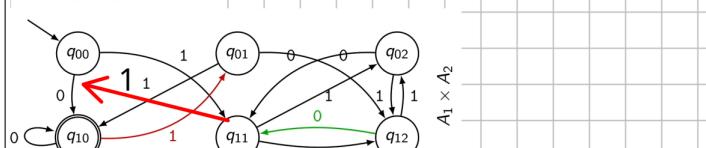
## Set-Operations with Automata



cartesian product -> acceptable endstate which both share

Intersection  $L_1 \cap L_2$

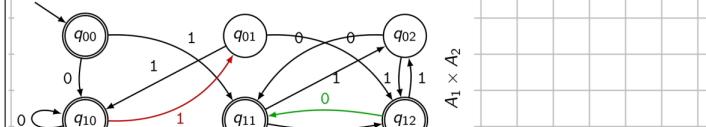
$$F = F_1 \times F_2$$



acceptable endstate of both combined

Union  $L_1 \cup L_2$

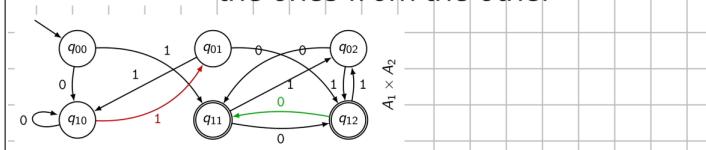
$$F = F_1 \times Q_2 \cup Q_1 \times F_2$$



acceptable endstate of one without the ones from the other

Difference  $L_1 \setminus L_2$

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



## Pump-able, but not regular

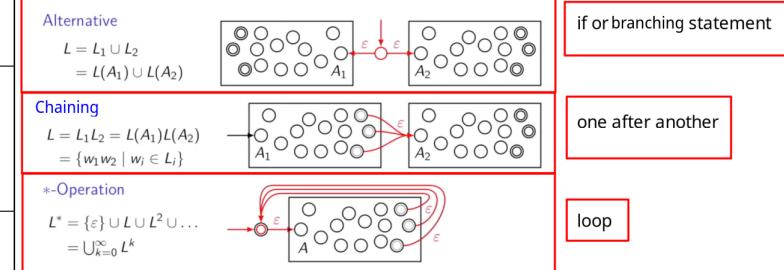
$$L = \{a^i b^j c^k \mid i = 0 \vee j = k\} = \underbrace{\{b^j c^k \mid j, k \geq 0\}}_{= L_1} \cup \underbrace{\{a^i b^j c^k \mid i > 0 \wedge j = k\}}_{= L_2}$$

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

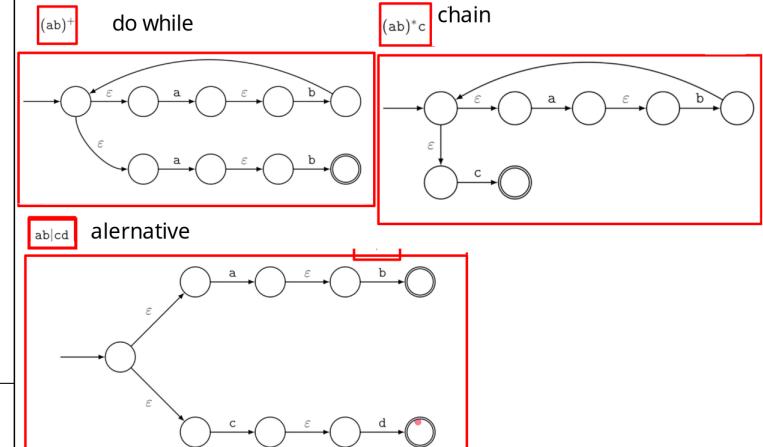
The only way we can figure that out is by using the Myhill method!

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

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

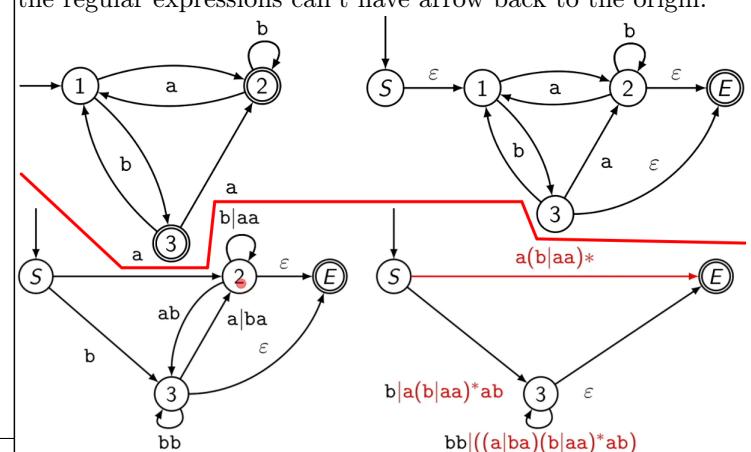
## Generalized NFA/NEA

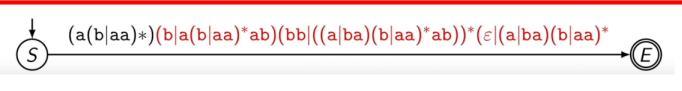
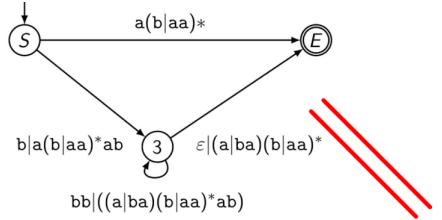
NEA<sub>ε</sub>, that has its transitions described  
NFA<sub>ε</sub>, as regular expressions

There is a DFA for every regular expression!

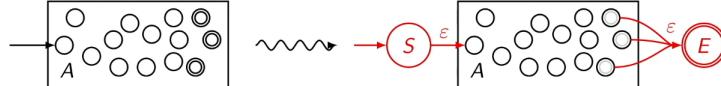
## VNEA / VNFA to regular expression

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

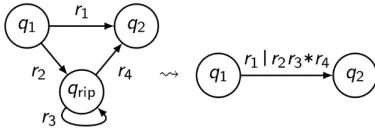




no transitions back to the starting point, and only 1 acceptable state, this simplifies the implementation

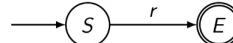


Reduction



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  $u \Rightarrow^* v$

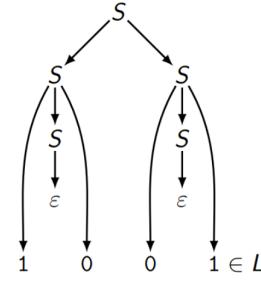
Context free grammar generated from G:

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

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:  $0^n 1^n$

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

$G_1:$

$$S \rightarrow \epsilon$$

$$\rightarrow SS$$

$$\rightarrow 0S1$$

$$\rightarrow 1S0$$

$$G_2:$$

$$S \rightarrow SB | \epsilon$$

$$B \rightarrow N | E$$

$$N \rightarrow NN | 0N1 | 01$$

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

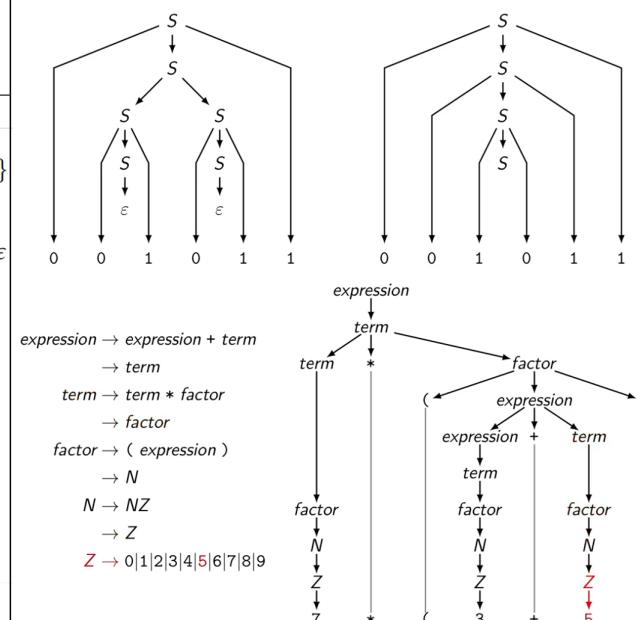
$$E \rightarrow EE | 1E0 | 10$$

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

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  $L_1 L_2$ :

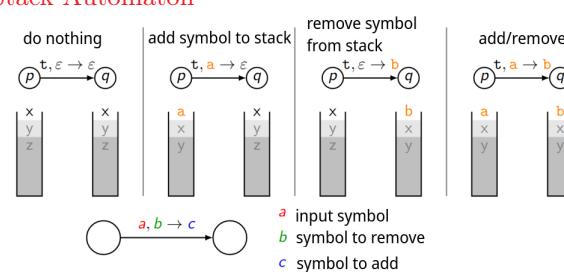
$$R = R_1 \cup R_2 \cup \{S_0 \rightarrow S_1 S_2\}$$

### \*-Operation

Rules for  $L_1^*$ :

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

## Stack Automaton

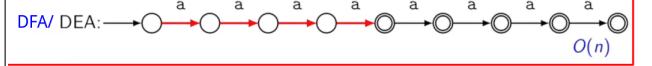
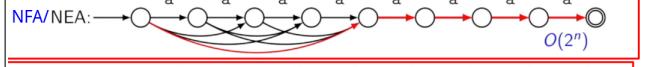
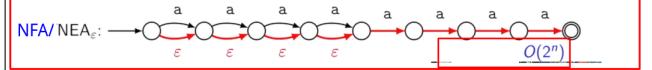




## Performance Comparison NFA/DFA

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



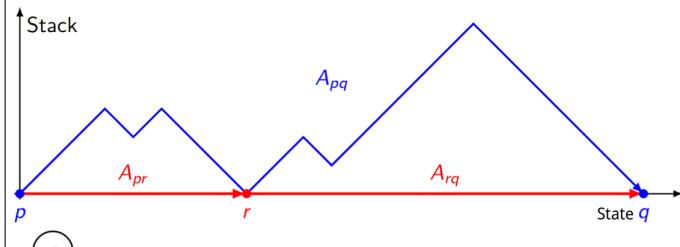
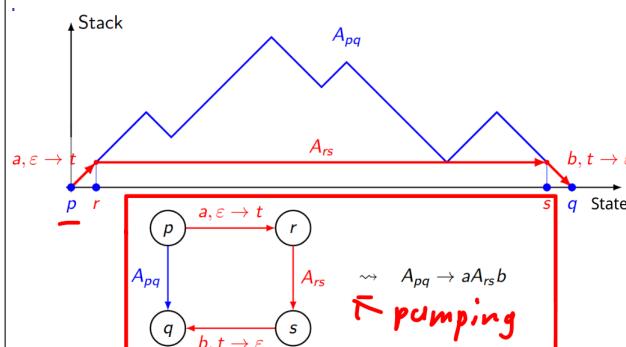
3. now all transitions need to involve the stack



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.

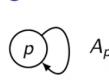


## Derived Rules from the Automaton:

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

① Startvariable:  $A_{q_0, q_a}$

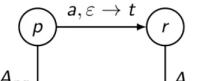
② Rules :



$A_{pp} \rightarrow \epsilon$



$A_{pq} \rightarrow a$



$A_{rs} \rightarrow \epsilon$

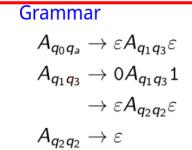
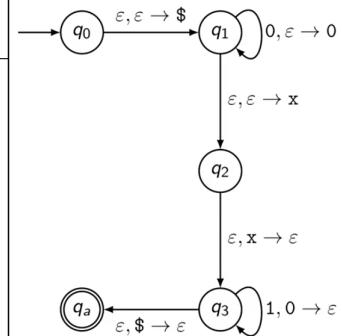


$A_{pr} \rightarrow A_{rq}$

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

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

PDA  $\rightarrow$  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

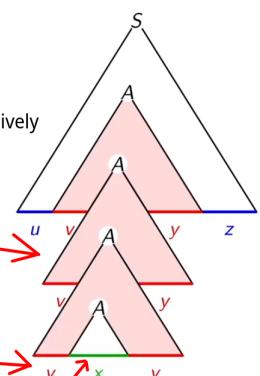
$w \in L(G) \Rightarrow S \xrightarrow{*} w$

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.  $uv^kxy^kz \in L \forall k \in \mathbb{N}$

5. 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 '|'

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

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

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

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

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

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



## 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  $\langle M \rangle$

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

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

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

Now use D on  $\langle D \rangle$

$D(\langle D \rangle) \text{ accepts } \Leftrightarrow D \text{ rejects } \langle D \rangle \text{ is not computable}$

$D(\langle D \rangle) \text{ rejects } \Leftrightarrow D \text{ accepts } \langle D \rangle$

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

## Halting Problem

The idea is that we have a Decider H and a program D that always returns the opposite. This D encompasses H, which means we give input to D, which gives the input to H. Then H returns and D inverts it, as described above.

The contradiction arises when you give D its own code as the input.

D passes this on and returns an answer, but D inverts it, aka does the opposite. This means H was wrong, aka the decider has lied, D didn't do what it said!

## Reduction

### Reductionprojection

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

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

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

### Computability

$B \text{ computable } \Rightarrow f: A \leq B \Rightarrow A \text{ computable}$

### Proof

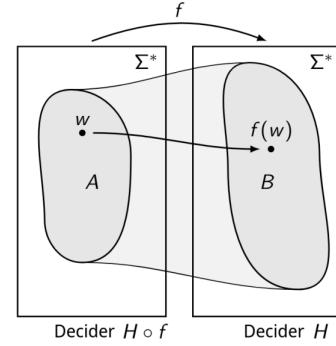
H is a decider for B, then is  $H \circ f$  a decider for A.

### 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

Does the machine M accept the word w?

is  $HALT_{\varepsilon_{TM}}$  computable?

Does the machine stop?

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

$M \text{ accepts } w$

$\Leftrightarrow$  S stops

### Decider for $A_{TM}$

1. Create program S
2. Use H on  $\langle S \rangle$

### 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

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.

Theorem:  $A \leq B$ , A nicht entscheidbar, dann ist B nicht entscheidbar.

$ATM \leq HALT_{\varepsilon_{TM}}$

### Reduktion für das Leereitsproblem: $ATM \leq ETM$

$ATM$  ist nicht entscheidbar

Ist  $ETM$  entscheidbar? Akzeptiert die Maschine M das Wort w? Ist die Sprache L(M) leer?  $\rightarrow ETM$  ist L(M)  $\neq \emptyset$

$(M, w)$

$\mapsto$

Programm S mit Input u

1. M auf w laufen lassen
2. M akzeptiert w: qaccept
3. Andernfalls: Endlosschleife

S hält genau dann, wenn M das Wort w akzeptiert:

$\langle M, w \rangle \mapsto \langle S \rangle$

ist eine Reduktion

$ATM \leq ETM$

Entscheider für  $ATM$

1. Konstruiere das Programm S
2. Wende H auf  $\langle S \rangle$  an

$\mapsto$

Entscheider für  $ETM$  Wenn H ein Entscheider für  $ETM$  wäre, könnte man daraus einen Entscheider für  $ATM$  konstruieren

### Regularitätsproblem: $ATM \leq REGULAR_{TM}$

$ATM$  ist nicht entscheidbar

Akzeptiert die Maschine M das Wort w?

Es gibt keinen Entscheider

$(M, w)$

$\mapsto$

Ist  $REGULAR_{TM}$  entscheidbar?

Ist die Sprache L(M) regulär?

Programm S mit Input u

1.  $u \notin \{0^n 1^n \mid n \geq 0\} \rightarrow qreject$
2. M auf w laufen lassen
3. M akzeptiert w: qaccept
4. qreject

M akzeptiert w

$\Leftrightarrow$

S akz.  $\{0^n 1^n \mid n \geq 0\}$ , nicht regulär

S akz.  $\emptyset$ , regulär

Entscheider für  $REGULAR_{TM}$

Wäre H ein Entscheider für  $REGULAR_{TM}$ , könnte man daraus einen

Entscheider für  $ATM$  konstruieren

## The law of Rice

### Law of Rice

#### Properties of a language

(properties of a turing language)

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

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

Does the machine M accept the word w?

Does the language  $L(M)$  have the property P?

assumptions

- $\Sigma^*$  doesn't have property P
- Testprogram T:  $L(T)$  doesn't have P

$\langle M, w \rangle$

$\mapsto$  Programm S mit Input u

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

$M$  accepts w

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

$M$  rejects w

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

## Overview of Computability

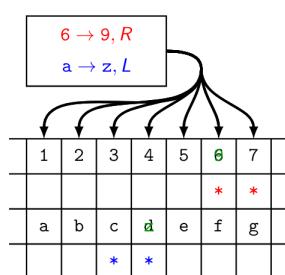
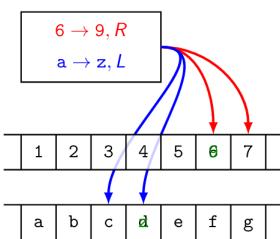
Problem	Word	Prerequisite		Computing algorithm / Reason
$E_{DEA}$	$\langle A \rangle$	$L(A) = \emptyset$	yes	minimal-automaton doesn't have accept state
$E_{CFG}$	$\langle G \rangle$	$L(G) = \emptyset$	yes	Chomsky-Normalform
$E_{TM}$	$\langle M \rangle$	$L(M) = \emptyset$	no	
$EQ_{DEA}$	$\langle A_1,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

## Efficiency and Computability

We do not just have to ask ourselves whether or not the problem is solvable, but also whether or not we can do so in a sensible amount of time.

Cracking the RSA algorithm is theoretically possible, but not practically, as it would take thousands of years to do so.

difference of Turing Machine Performance:



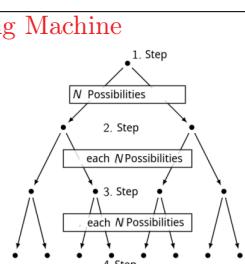
Difference: 1 step of Multiband ==  $O(t(n))$

it is the worst case, the 4row machine takes 1 full iteration, just to simulate 1 change of the multiband machine.

Multiband:  $O(n) \rightarrow$  Multirow:  $O(n^2)$

## Simulation of a Nondeterministic Turing Machine

- N possibilities.
- Every step gives  $\leq N$  possibilities
- Amount of steps:  $t(n)$
- Worst case:  $N^{t(n)} = 2^{t(n)*\log_2 N}$
- $2^{O*t(n)}$



## Difference in Hardware

### Difference between deterministic Hardware

the difference between deterministic TMs is never big. It stays polynomial!  
max change:  $O(t_1(n) = O(t_2(n)^k))$ ,  $k \geq 0$

hardware change	complexity
regular TM	$t(n)$
different Alphabet	$O(t(n))$
more rows	$O(t(n))$
more bands	$O(t(n)^2)$

### Nondeterministic Hardware:

Nondeterministic Hardware is always faster!

It can be simulated within  $2^{O(t(n))}$  Notice the exponential complexity!

The difference between TMs and NTMs is HUGE!

## P and NP Problems

P is the quantity of problems that can be solved within polynomial time by a deterministic TM

NP is the quantity of problems that can be solved within polynomial time by a nondeterministic TM

For NP there is always a deterministic TM that can verify the solution!!

NP is a bigger quantity than P!!

## Polynomial and Exponential Problems

### Polynomial

- Gauss  $O(n^3)$
- FFT  $O(n \log n)$
- Sort  $\leq O(n^2)$
- Inner Points algorithm for linear optimization
- PRIMES, Agrawal, Kayal, Saxena 2004
- Check a sudoku solution

### Exponential

- Simplex-algorithm for linear optimization
- Integer Optimization
- Factorization of big numbers (RSA)
- Discrete Logarithm (Diffie-Hellmann, elliptic curves )
- Hash-Collisions
- solve a sudoku

## Scalability

## Security

## Verification of Problems

Luckily the simple verification of a problem can always be done in polynomial time by a regular deterministic TM

### Verifier

A verifier in polynomial time is a Turing Machine V that can verify a solution of every word  $w \in \Sigma^*$  within polynomial time. While the solution is done with an NTM.

$$w \in L \Leftrightarrow V(w, c)$$

The complexity of V is polynomial within  $|w|$ .

## Complexity class NP

A Problem is ONLY in NP if the solution can be verified in polynomial time, by a deterministic TM.

## An Example for a verifier

Verifier for : Sudoku

### Computable?

yes, all  $\leq (n^2)^{n^2 \times n^2}$  possibilities have to be checked

certificate?  $c =$  all missing numbers

verification-algorithm

Nr.	to do	complexity
1	for every field, the number doesn't appear on this row anymore	$O(n^4 \cdot n^2)$
2	for every field, the number doesn't appear on this column anymore	$O(n^4 \cdot n^2)$
3	for every field, the number doesn't appear on this sub-square anymore	$O(n^4 \cdot n^2)$
4		
5		
Total		$O(n^6)$

## k-Vertex-Coloring

The idea of the vertex-coloring problem is that you have k colors and a graph G for every node, each neighbor should have a different color

The problem: is it possible for the graph G?  $\rightarrow$  NP Problem!

This problem can be converted to another problem, for example to the timetable problem

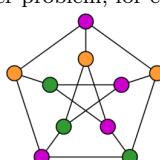
Each node is a module

Each color is a time slot

Each edge is a registration

The timetable should never overlap

k timeslots per week





## 3SAT to SUBSET-SUM

$3SAT \leq_P SUBSET-SUM$

3SAT

$$\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

SUBSET-SUM

$$\langle S = (y_i, z_i, g_k, h_k | i \leq l, k \leq n), t \rangle$$

Solution  $T \subset S: \sum_{s \in T} s = t$

- $y_i \in T \Rightarrow x_i$ : True
- $z_i \in T \Rightarrow x_i$ : False
- $g_i, h_i$ : filler, max 2 per clause

### Conclusion

$\varphi$  fullfillable  $\Leftrightarrow \langle S, t \rangle$  computable

try:  $x_1=1 x_2=1 x_3=1$   
we get the 3 numbers inside the boxes above.  
add them up,

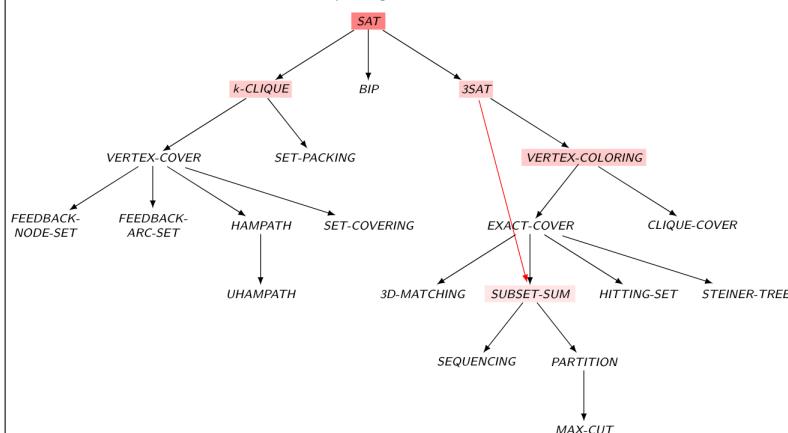
Number	$x_1$	$x_2$	$x_3$	$c_1$	$c_2$	$c_3$
$y_1$	1	0	0	1	0	0
$z_1$	1	0	0	0	1	1
$y_2$		1	0	1	1	0
$z_2$	1	0	0	0	1	1
$y_3$		1	1	0	0	0
$z_3$	1	0	1	1	1	1
$g_1$				1	0	0
$h_1$				1	0	0
$g_2$					1	0
$h_2$					1	0
$g_3$						1
$h_3$						1
$t$	1	1	1	3	3	3

we have  $1,1,1 \quad ,3, 1, 0$   
we can add  $g_2, h_2$  and  $g_3, h_3$ , however  
this only leads to  $3, 2, 0$   
This means that  $x_1=1 x_2=1 x_3=1$  is NOT  
a solution!!!

## Karp-Catalog

A catalog of NP-complete problems

The arrow denote the reducability. e.g. 3SAT to SUBSET-SUM



## Turing Machine vs PC

Turing Machine

States Q

Practically infinite memory -> Band

Read-Write Head

Stop ->  $q_{accept}$  or  $q_{reject}$

TM made for specific problem

Regular PC

Sates of CPU: Bits etc

virtual Memory (practically infinite)

Adress Register, Program Counter

EXIT\_SUCCESS, EXIT\_FAILURE

runs ANY program

## The universal Turing Machine

Alan Turing made a Turing Machine that can simulate another TM!

These are the things needed for the TM:

- Band for the transition function from TM to TM  
 $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
- Band for the current state
- Working Band

## Things that are missing on a TM

- Persistent Memory (Storage)
- Input with keyboard/mouse
- Input/Output (Monitor etc)

None of these additions change anything.

are just things we added for quality of life

aka, all of these changes can be implemented on a TM,

any issues as well! Just slow....

## Comparison of Machines

If Machine 1 can simulate Machine 2

Then Machine 1 is "more powerful" than Machine 2

$M_2 \leq_s M_1$   $M_1$  simulates  $M_2$

Unfortunately, the term is misleading, a PC is faster than a TM

But according to this an 8bit CPU and a TM are the same....

## Game of Life

### States

Cells are dead or alive

### Rules

n Count of living neighbors

1. Cell dead,  $n = 3$ : cell reborn
2. cell alive
  - $2 \leq n \leq 3$ : survives
  - else: dead

### Structures

- stable or oscillating
- mobile structures
- Web, universal TM

1	2	2	1	1	1	1	1	1	2	2	1
2	3	3	2	1	2	2	1	1	2	2	2
2	3	3	2	1	2	4	2	1	1	2	4
1	2	2	1	1	2	2	2	1	1	2	4
1	2	2	1	1	1	1	1	1	2	2	2
1	1	1	1	1	1	1	1	1	1	2	2
1	1	2	1	2	1	2	1	2	3	3	2
2	1	3	3	2	2	3	2	3	4	3	2
2	2	3	2	2	2	1	2	3	2	1	1
1	1	2	2	1	1	1	1	1	2	3	2
1	1	1	0						1	2	2
1	2	2	1						1	2	2
1	3	4	4	2	1	1	1	2	1	1	2
1	2	4	4	3	1	1	2	3	2	1	1
1	2	3	2	1					1	2	2

## Turing Complete

Do we lose any capabilities of a TM with programming languages?  
If no, then said language is Turing complete!

### $TM \leq_s$ Programming language

Or a programming language A is Turing Complete, if a projection:

$$c: A \rightarrow \Sigma^*$$

exists, where  $c(w)$  is a program of a univeral Turing machine.

examples of Turing complete languages: C, C++, javascript (Browser LOL), etc.

## Base elements of a Programming language

Both turing complete and not turing complete offer this:

- Constants c:  $N_0$   $0 \rightarrow \infty$
  - Variables  $x_1, x_n: N_0$
  - Assignment:  $x_i := c$
  - Addition:  $x_i := x_j + c$
  - Subtraction:  $x_i := x_j - c$
- Note is  $c \geq x_j$  then  $x_i = 0$
- !Not Included!:  $x_i + x_j$ ,  $x_i - x_j$ ,  $x_i * x_j$ ,  $x_i / x_j$

## Non-Turing Complete Languages

These types of languages ALWAYS halt! No halting problem!

LOOP: not turing complete

Control Structure	Addition / Subtraction	Multiplication
run a Program P exactly $x_i$ times	$x_i := x_j \pm x_k$ $x_i := x_j$	$x_i := x_j * x_k$ $x_i := 0$
LOOP $x_i$ DO P END	LOOP $x_i$ DO $x_i := x_i \pm 1$ END	LOOP $x_k$ DO $x_i := x_i + x_j$ END
		OR: $x_i := 0$
	LOOP $x_k$ DO $x_i := x_i + 1$ END	LOOP $x_j$ DO $x_i := x_i + 1$ END

We do not have conditional loops, only n says how many times we loop

This means we will ALWAYS halt, on n. This can be proven by induction, n - 1!

## While and GOTO

WHILE extension of LOOP

Definition of WHILE

- Base elements like in LOOP
- Conditional loop -> Turing Complete

WHILE  $x_i > 0$  DO P END

Do P as long as  $x_i = 0$

IF in WHILE LOOP in WHILE  
 $y := x_i$   
WHILE  $y > 0$  DO  
 $y := 0; P$   
END

$P$   
 $y := y - 1$

GOTO Definition of GOTO

► Base elements like in LOOP

► Series of statements

$M_1 : A_1$

$M_2 : A_2$

...: ...

$M_k : A_k$

contraction:

$M_{k+1} :$

$M_{k+2} : \text{IF } x_i = c \text{ THEN GOTO } M_{k+1}$

...: ...

$M_{k+n} : A_l$

Unconditional Statements

$M_{k+1} : \text{IF } x_i = c \text{ THEN GOTO } M_{k+1}$

...: ...

$M_{k+n} : A_l$

Conditional Statements

$M_{k+1} : x_i := c$

$M_{k+2} : \text{IF } x_i = c \text{ THEN GOTO } M_{k+2}$

...: ...

$M_{k+n} : A_l$

modified statement  $A'_i$

►  $A_i$ : conditional Jump statement

$M_i : \text{IF } x_i = c \text{ THEN GOTO } M_j$

will be translated to  $A'_i$ :

IF  $x_i = c$  THEN  $z := j$  ELSE  $z := z + 1$  END

►  $A_i$ : a regular statement

$M_i : A_i$

will be translated to  $A'_i$ :

$A_i; z = z + 1;$

implemented in WHILE:

$z := 1$   
WHILE  $z > 0$  DO  
IF  $z = 1$  THEN  $A'_1$  END  
IF  $z = 2$  THEN  $A'_2$  END  
:  
IF  $z = k$  THEN  $A'_k$  END  
IF  $z = k + 1$  THEN  $z := 0$  END  
END

## Implementation WHILE in GOTO Translation

A WHILE-Construct

WHILE  $x_i > 0$  DO  $P$  END

Will be translated to a GOTO-Code segment

$M_I : \text{IF } x_i = 0 \text{ THEN GOTO } M_{I+3}$   
 $M_{I+1} : P$   
 $M_{I+2} : \text{GOTO } M_I$   
 $M_{I+3} :$

### Equivalence

The languages GOTO and WHILE are equivalent

### Turing-Complete

A turing machine can be implemented in both GOTO and WHILE, therefore, both WHILE and GOTO are:

Turing-Complete

## Complexity Class BPP

- PTM = NTM with random possibility chosen
- results not always accurate!
- Algorithm to calculate error probability is polynomial!  
Then used to determine if result is "good enough"
- encompasses all Problems that can be solved in polynomial time by a PTM.
- $P \subset BPP$

-  $BPP \approx NP$  not all, choose random instead.

Example for P Algorithm vs BPP algorithm:

### Prime test

Get all primes by dividing  $p$  with all thinkable factors until the square root of  $p$

**Complexity**  
Amount of Divisions  $O(\sqrt{p})$   
dependency on bitlength  
 $n = \log_2(p)$ :  
 $\sqrt{p} = \sqrt{2\log_2(p)} = 2^{\frac{1}{2}\log_2(p)}$   
 $= 2^{O(n)}$   
exponential complexity!!

### Lehmann-Test

- choose random,  $a < p$
  - $q = a^{(p-1)/2} \bmod p$
  - $q \not\equiv \pm 1 \bmod p \Rightarrow p$  not prime
  - otherwise the chance or  $p$  not being prime  $\approx 0.5$
- complexity:  $a^{(p-1)/2} \bmod p: O(n^2)$
- Iteration**  
Lehman-test passed k times,  
probability of  $p$  not prime:  $\leq \frac{1}{2^k}$ ,  
polynomial complexity!!  $kO(n^2) = O(n^2)$

we sacrifice some accuracy to gain a LOT of speed!!!

## Quantum

- Quantummechanics: The physics of singular elements
- velocity, acceleration, force, etc do no longer matter
- Measurement always affect the element (current limitation!)

### Quantummechanics / Statemechanics

**State:** discrete states  $|0\rangle, |1\rangle, |2\rangle, \dots$

**State-space:** Vectorspace with states  $|i\rangle$  as basis

elements can be in overlapping state:

$$|\psi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle + \alpha_2|2\rangle + \dots$$

**Measurement:** a state is realized

$$|\psi\rangle \xrightarrow{P} |i\rangle \quad P^2 = P, \text{ Projectionmatrix}$$

**Hypothesis:** Probability of the measurement  $P$  to find state  $|i\rangle$

$$\text{These states are unit vectors:} \quad |\psi_+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$|\psi_-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

$|\psi_{\pm}\rangle$  This represents both vectors at once, if we do something with it, we modify both states at once!!

### Qubits

#### Singular Qubits

- Photons  $|R\rangle$  or  $|L\rangle$  polarized
- Electrons with a clockwise spin or the opposite  $| \uparrow \rangle, | \downarrow \rangle$
- Atoms in crystals, Ions,etc
- Hadamard

$n$  Qubits

$2^n$  states

2 Qubits = 4 states

$$|00\rangle, |01\rangle, |10\rangle, |11\rangle$$

3 Qubits = 8 states

$$|000\rangle, |001\rangle, |010\rangle, |011\rangle, |100\rangle, |101\rangle, |110\rangle, |111\rangle$$

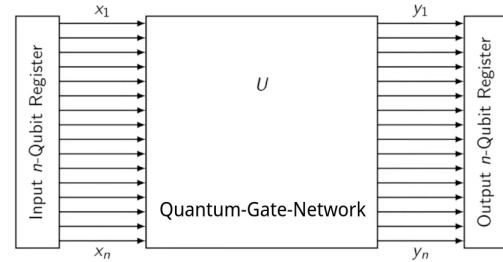
### Quantum-Gates

#### CNOT

$$|0\dots 0\rangle \xrightarrow{H} \sum_{i_1, i_2, \dots, i_n} \alpha_{i_1 i_2 \dots i_n} |i_1 i_2 \dots i_n\rangle$$

#### Toffoli-Gate

## CIRCUIT $\rightarrow$ Quantum-Algorithms



All Quantum-gates can be inverted!!

### Problem

Given a formula  $\varphi$  that takes in 2 inputs, for which half of it is constant, and the other half true.

### Verifier

Certificate: 2 inputs that give the 2 different values  $\Rightarrow$  NP Solution regular machine

Test all inputcombinations, at the max of half of all inputs, do we know what condition we have: const or true.

### Quantum solution

multiply all states  $|x\rangle$  with  $(-1)^{\varphi(x)}$

$p$  = possibility of measuring the same state

$\varphi = \text{const}$  constructive interference

$\varphi = \text{balanced}$  destructive interference

$p = 1$

$p = 0$

What is even possible?

- Can simulate a regular PC (although very slow)
- Halting problem still not computable !!!

### Quantum-SAT

Given a formula that takes in 2 inputs, for which half of it is constant, and the other half true.

### Problem

Given a formula  $\varphi$  that takes in 2 inputs, for which half of it is constant, and the other half true.

### Verifier

Certificate: 2 inputs that give the 2 different values  $\Rightarrow$  NP Solution regular machine

Test all inputcombinations, at the max of half of all inputs, do we know what condition we have: const or true.

### Quantum solution

multiply all states  $|x\rangle$  with  $(-1)^{\varphi(x)}$

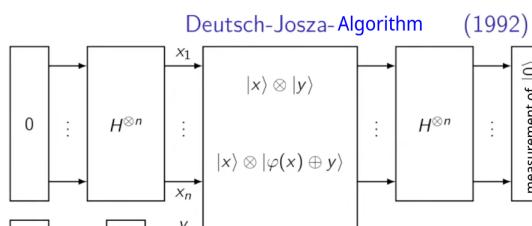
$p$  = possibility of measuring the same state

$\varphi = \text{const}$  constructive interference

$\varphi = \text{balanced}$  destructive interference

$p = 1$

$p = 0$

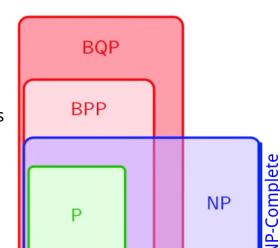


complexity:  $O(1)$   $\rightarrow$  exponential acceleration

## Complexity Class BQP

What can Quantumcomputers compute?

- Polynomial Time for the TM, which controls the Quantumoperations
- Constricted Error Probability  $\Rightarrow$  BQP



reliability achieved through:

- ① recursion, aka do again
- ② verifier, all NP problems have a polynomial verifier

Quantummechanical Process as source for randomness

Quantum Computers need thousands of QBits to be useful right now, best we can do is around 100....  
"don't hold your breath for quantum pcs :)"