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: $P \land Q$ || OR: $P \lor Q$ || NOT: $\neg P$
Implication: $P \implies Q = \neg P \lor Q$

Distributive Rule

$P \land (Q \lor R) = (P \land Q) \lor (P \land R)$
$P \lor (Q \land R) = (P \lor Q) \land (P \lor R)$

De Morgans Law

$\neg(P \land Q) = \neg P \lor \neg Q$
$\neg(P \lor Q) = \neg Q \implies \neg P$
$P \implies Q = \neg Q \implies \neg P$

Implication

$P \implies Q = True$ & $\neg P \implies Q = True$
$P \implies \neg Q = False$
$\neg P \implies \neg Q = True$

Quantors

OR: $\bigvee_{k=0}^{n} P_k$
P true for any $k \in 0..n$
All: $\forall k \in 0..n = P_k$
for all k P = True

AND: $\bigwedge_{k=0}^{n} P_k$
P true for all $k \in n$
Exists: $\exists k \in 0..n = P_k$
a k exists where P = True

Normalforms

disjunctive
$(x1 \land x2) \lor (\overline{x1} \land x2) \lor (x1 \land \overline{x2})$

conjunctive
$(x1 \lor x2) \land (\overline{x1} \lor x2) \land (x1 \lor \overline{x2})$

These are useful for true and false tables
This one would result to true if x1 or x2 is true.

Quantities:

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

Union: $A \cup B = \{x | x \in A \lor x \in B\}$
Intersection: $A \cap B = \{x | x \in A \land x \in B\}$
Complement: $\overline{A} = \{x | x \notin A\}$
Difference: $A \setminus B = \{x \in A | x \notin B\}$

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

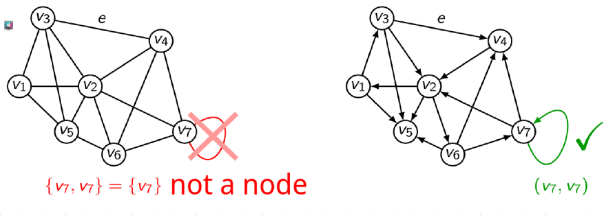
n-Tuples $\bigtimes_{k=0}^{n} A_i = \{(a_o, a_i, .., a_n) | a_i \in A_i\}$

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 = $\{v_1, v_2, ..., v_n\}$ Edge e = $\{v_3, v_4\}$
EdgeCount E = $\{e | e Edge\}$

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

Anker: check for n=1
$P(1) = \frac{1(1+1)}{2} = 1$

Hypothesis: it also works for n+1

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

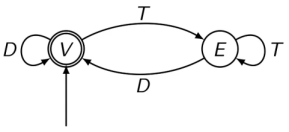
Alphabet and Word

Σ = Alphabet: Nonempty Quantity of characters
$\Sigma^n = \Sigma \times ... \times \Sigma$ = String
$w \in \Sigma^n$ An element in that string is a Word with length n.
$\varepsilon \in \Sigma^0$ The empty word, don't forget the empty word!
Quantity of all words:
$\Sigma^* = \{\varepsilon\} \cup \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup ... = \bigcup_{k=0}^{\infty} \Sigma^k$

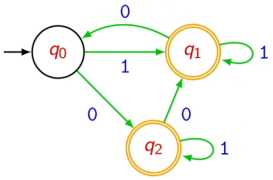
Language $L \subset \Sigma^*$ = Language

$L = \emptyset \subset \Sigma^*$ = Empty Language
$L = \Sigma^* \to \Sigma = \{0, 1\}$ 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, ...



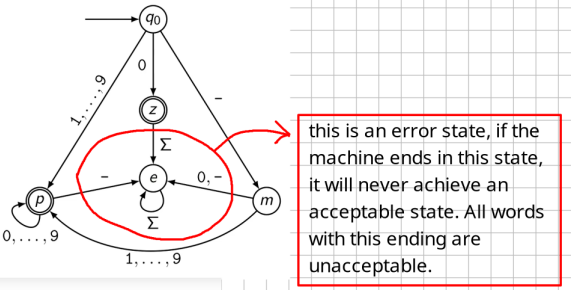
Machine A: {Q,Σ,δ,q0,F}
State = Q -> $\{q_1, q_2, ..., q_n\}$
Alphabet = Σ
Transitioning-Function = $\delta : Q \times \Sigma \to Q$
Starting State = $q_0 = L(\varepsilon) = L$
Acceptable Endstates = $F \subset Q$

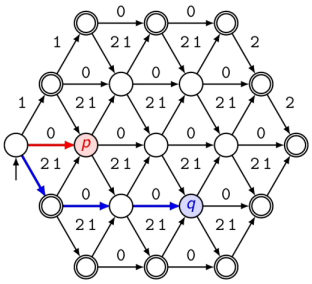
	0	1	
all states Q → q_0	q_2	q_1	← Σ possible inputs, here 0 or 1
q_1 /F	q_0	q_1	← δ current position. -> change to q1
all acceptable states. F → q_2 /F	q_1	q_2	

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





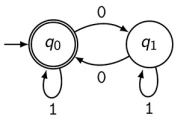
from q, many paths lead to F
This means 11, or 0 would be the "same"
$L(q) = \{0, 10, 11, 12, ...\}$
The same would obviously apply to P

Myhill-Nerode

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

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

w	$L(w)$	Q
ε	$L(\varepsilon) = L$	q_0
0	$L(0) = \{w \in \Sigma^* \mid \lvert w\rvert_0$ uneven $\}$	q_1
1	$L(1) = \{w \in \Sigma^* \mid \lvert w\rvert_0$ 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^n0^n

w	$L(w)$	Q
ε	$\{0^n1^n \mid n \geq 0\}$	q_0
0	$\{0^n1^{n+1} \mid n \geq 0\}$	q_1
00	$\{0^n1^{n+2} \mid n \geq 0\}$	q_2
000	$\{0^n1^{n+3} \mid n \geq 0\}$	q_3
0^k	$\{0^n1^{n+k} \mid n \geq 0\}$	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 $\lim_{k\to\infty}$ we need ∞ states!

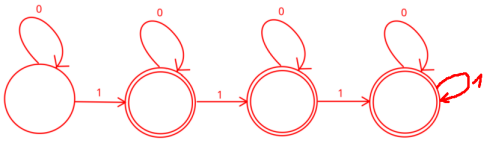
not possible with a Deterministic automaton!

also note: we have clear error states

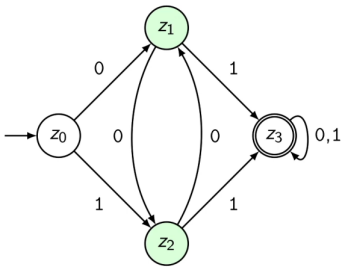
anything starting with 1 is an error.

Differentiation of States

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



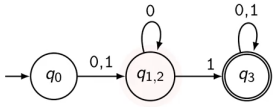
Here the 3 acceptable states can be put together, they are the same!



	z_0	z_1	z_2	z_3
z_0	≡	×	×	×
z_1	×	≡	≡	×
z_2	×	≡	≡	×
z_3	×	×	×	≡

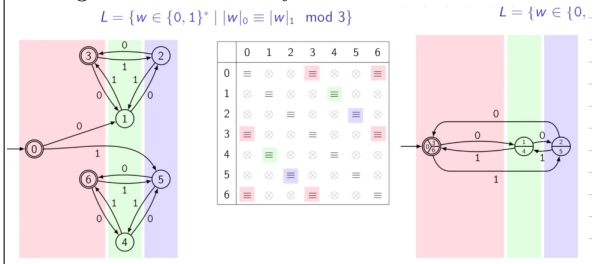
z1 and z2 are the same!

they can't be differntiated



New minimal Automaton!

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



Beauty of a language / Pumping Lemma

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

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

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

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

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^kz is still part of the language

Note: The power of xy^kz 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 = any combination of 1 and 0

try: x=0 y=1010111 z=1,True -> $N \geq 7$

try: x=0 y=1 z=1,True -> $N \geq 1$

This can be done with any y, x

It will always satisfy all requierements 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

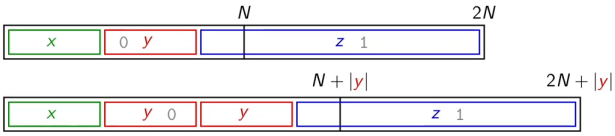
if x - 0, more 0's than 1's, FALSE

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^n1^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

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

5. Check if word is in language

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

Explain why this word is not in the language

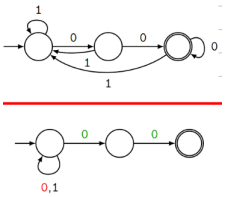
6. Contradiction, aka explain that this language is not regular.

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

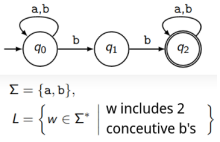
This is illustrated as both a DFA and an NFA:



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,Σ,δ,q_0,F}

State = Q -> $\{q_1, q_2, ..., q_n\}$

Alphabet = Σ

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

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

Acceptable Endstates = $F \subset Q$

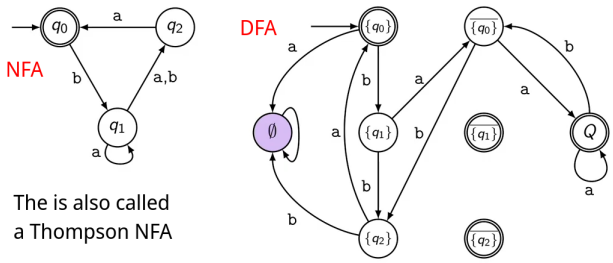
P(Q) is the Potence Quantity!!

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

NFA To DFA



$\overline{q}_1$ 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 δ of an NFA and Transitions $M \subset Q$

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

$Q' = P(Q)$

$\Sigma' = \Sigma$

$q_0' = \{q_0\}$

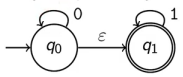
$F' = \{M \in P(Q) | F \cap M \neq \emptyset\}$

M is the union of all possible endstates.

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

ε Transitions in NFA's

$L = \{0^k 1^l \mid k, l \geq 0\}$



$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q)$

This means that ε signifies a transition without using a character!!

Conversion from ε-NFA to regular NFA

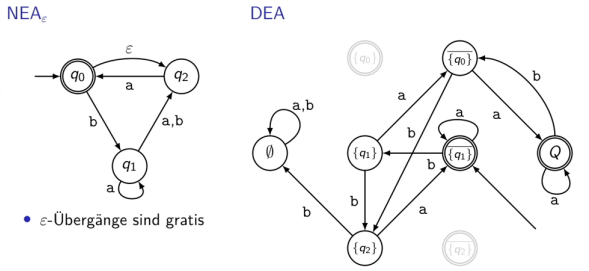
Any ε-NFA can be converted to a regular NFA!

$E(q)$ =Quantity of all ε-Transitions from q

$E(M) = \cup_{q \in M} E(q)$ Quantity of all ε-Transitions

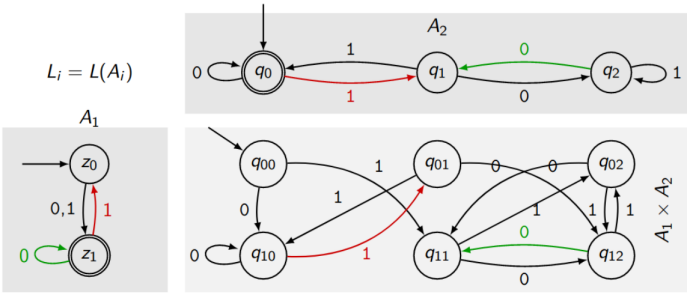
$\delta = Q \times (\Sigma \cap \{\varepsilon\} \to P(Q) : (q, a) \mapsto E(\delta(q, a)))$

ε-NFA to DFA:



Set-Operations with Automatons

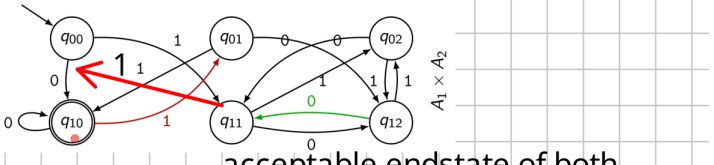
$L_i = L(A_i)$



Intersection $L_1 \cap L_2$

$F = F_1 \times F_2$

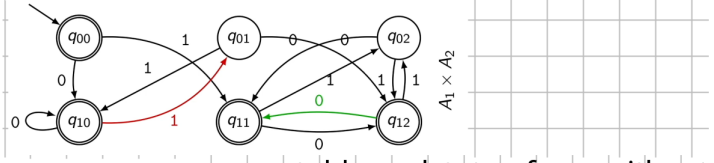
carthesian product -> acceptable endstate which both share



Union $L_1 \cup L_2$

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

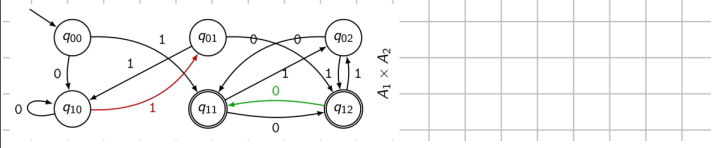
acceptable endstate of both combined



Difference $L_1 \setminus L_2$

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

acceptable endstate of one without the ones from the other



Pump-able, but not regular

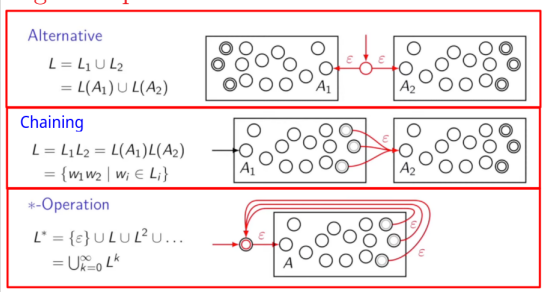
$L = \{a^i b^j c^k \mid i = 0 \vee j = k\} = \underbrace{\{b^j c^k \mid j, k \geq 0\}}_{= L_1} \cup \underbrace{\{a^i b^j c^k \mid i > 0 \wedge j = k\}}_{= L_2}$

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



if or branching statement

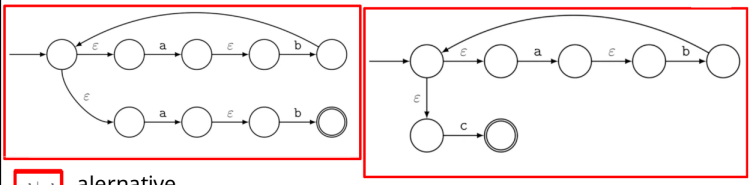
one after another

loop

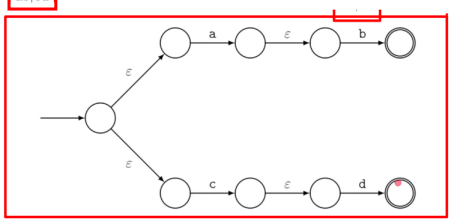
Regular Expressions

$(ab)^+$ do while

$(ab)^*c$ chain



ab|cd alernative



Regular expression

String r to describe a language

L = L(r)

Regular Operations

$L(r_1) \cup L(r_2) = L(r_1 | r_2)$ Alternative

$L(r_1)L(r_2) = L(r_1 r_2)$ Chaining

$L(r_1)^* = L(r_1 *)$ *-Operation

Primitive regular expressions

expressions for Words with length <= 1

$L = L(r)$	r	NEA
$\emptyset$	$\emptyset$	
$\{\varepsilon\}$	ε	
$\{a\}$	a	
$\{o, s, t\}$	[ost]	
$\{a, b, \ldots, s\}$	[a-s]	
Σ	.	

Contractions

$r+ = rr*$ $\quad r? = \varepsilon | r = |r$

$r\{2\} = rr$ $\quad r\{2,3\} = rr|rrr$

$r\{,3\} = |r|rr|rrr$ $\quad r\{3,\} = rrrr*$

Generalized NFA/NEA

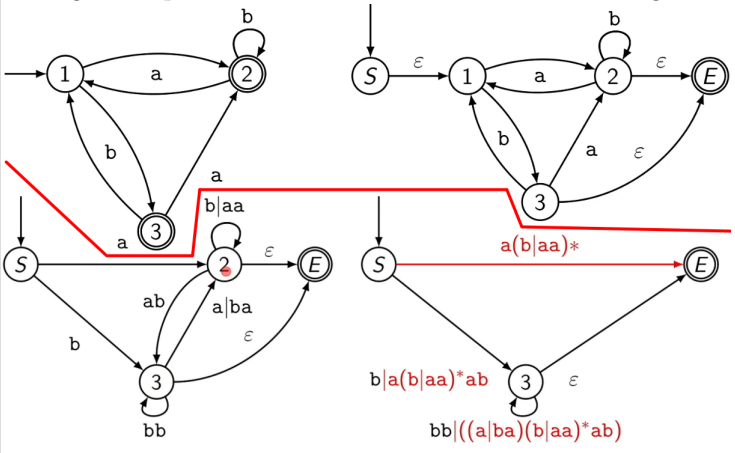
NEA$_\varepsilon$, that has it's transitions described

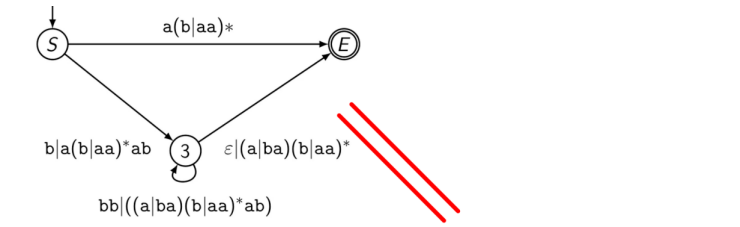
NFA$_\varepsilon$, as regular expressions

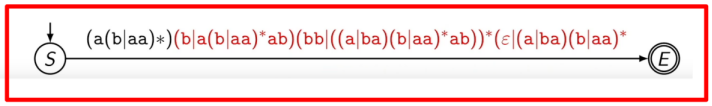
There is a DFA for every regular expression!

VNEA / VNFA to regular expression

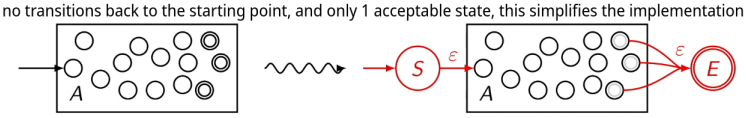
1. add a new start and stop state. This is required as the regular expressions can't have arrow back to the origin.



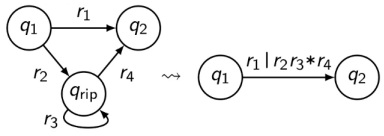




no transitions back to the starting point, and only 1 acceptable state, this simplifies the implementation

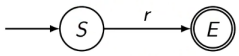


Reduction



Regular Expression

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

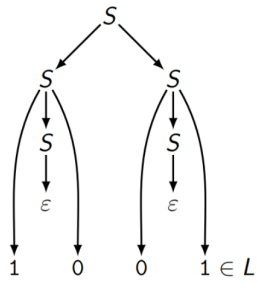


Contextfree-Grammar (CFG) » $G = (V, \Sigma, R, S)$

V: Variables / Non-Terminalsymbols
Σ: Terminalsymbols (Alphabet)
R: Rules of Form $A -> x_1X2...x_n$
with $A \in V, x_i \in V \cup \Sigma$
S: Starvariable
Rule A -> w generated out of uAv
$uwv : uAv \implies uwv$
derivate v from u:
$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow ... \Rightarrow u_n \Rightarrow v$ 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:
$S \rightarrow 0S1 \mid 1S0 \mid SS \mid \varepsilon$



Example:
Grammar $L = \{0^n1^n | n \geq 0\}$
Variables: $V = \{S\}$
Terminalsymbols: $\Sigma = \{0, 1\}$
Rules: $R = \{S \rightarrow \varepsilon | 0S1\}$
Startvariable: S

ε
$01 = 0\varepsilon1$
$0011 = 0\,01\,1$

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 wihout regard for
what is to the left or right of it, then it has a context free rule
$A \rightarrow$ something
A grammer with only variables like A is considered context free.

Rules without context:
$S \rightarrow A \mid C$
$A \rightarrow a$
$A \rightarrow aAb$
$C \rightarrow bA$

Rules with context:
$S \rightarrow C \mid aC \mid bC$
$aC \rightarrow A$
$bC \rightarrow B$

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^n1^n

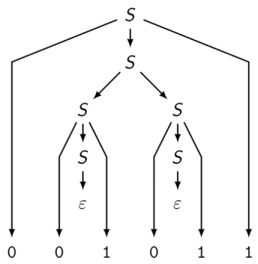
$L = \{w \in \Sigma^* \; | |w|_0 = |w|_1\}$

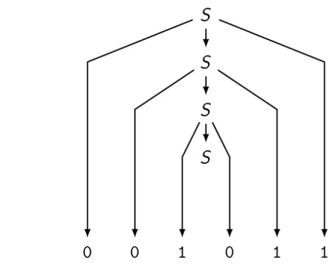
G_1:
$S \rightarrow \varepsilon$
$\rightarrow SS$
$\rightarrow 0S1$
$\rightarrow 1S0$

G_2:
$S \rightarrow SB \mid \varepsilon$
$B \rightarrow N \mid E$
$N \rightarrow NN \mid 0N1 \mid 01 \quad L(N) = \{w \mid w \text{ 0-1- expression}\}$
$E \rightarrow EE \mid 1E0 \mid 10 \quad L(E) = \{w \mid w \text{ 1-0- expression}\}$

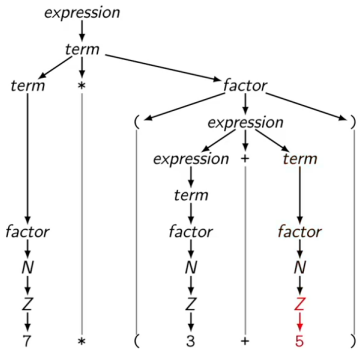
The grammar G_2 gives us a clear Parse-tree, the first one doesn't
The reason for this is the obvious haskell like structure on the right.
The one on the left has no idea of leafs and nodes.
Another example with parse trees:

$S \rightarrow 0S1 \mid 1S0 \mid SS \mid \varepsilon$





$expression \rightarrow expression + term$
$\rightarrow term$
$term \rightarrow term * factor$
$\rightarrow factor$
$factor \rightarrow (\ expression\ )$
$\rightarrow N$
$N \rightarrow NZ$
$\rightarrow Z$
$Z \rightarrow 0|1|2|3|4|5|6|7|8|9$



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

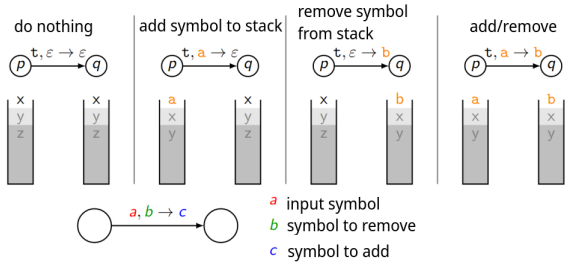
$\Rightarrow G = (V, \Sigma, R, S_0)$

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_1L_2:
$R = R_1 \cup R_2 \cup \{S_0 \rightarrow S_1S_2\}$

*-Operation
Rules for L_1^*:
$R = R_1 \cup \{S_0 \rightarrow S_0S_1, S_0 \rightarrow \varepsilon\}$

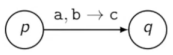
Stack Automaton



depending on the definition, the input can be anything!

Definition

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

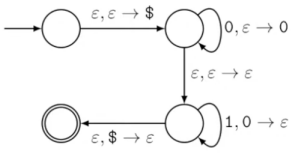


1. Q: States
2. Σ: Input Alphabet
3. Γ: Stack-Alphabet
4. δ: $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$
5. $q_0 \in Q$: Start-State
6. $F \subset Q$: Accepting-State

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

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

Stack Automaton

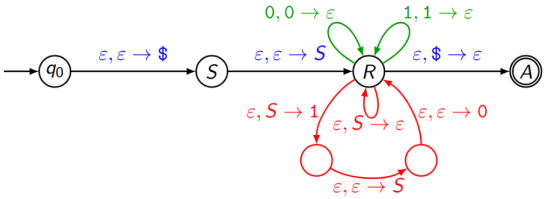


Grammar

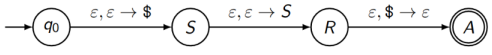
$S \to 0S1$
$\to \varepsilon$

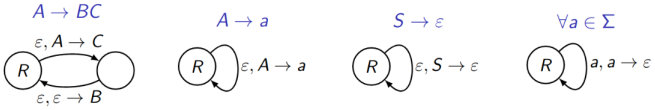
Schema of the Automaton

1. $ and Startvariable S
2. Transition for every Rule
3. Transition for everv Terminalsvmbol



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

$$\left.\begin{array}{l} A \to B \\ B \to A \\ A \to \text{a} \end{array}\right\} \Rightarrow \quad A \to B \to \underbrace{A \to B \to \cdots \to A}_{\text{can be infinite!!}} \to \text{a}$$

2. endless empty Variables

$$\left.\begin{array}{l} A \to ABCDE \mid \text{a} \\ B \to \varepsilon \\ C \to \varepsilon \\ D \to \varepsilon \\ E \to \varepsilon \end{array}\right\} \Rightarrow \quad A \to ABCDE \to ABCD \to ABC \to AB \to A \to \text{a}$$

same here !!

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

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

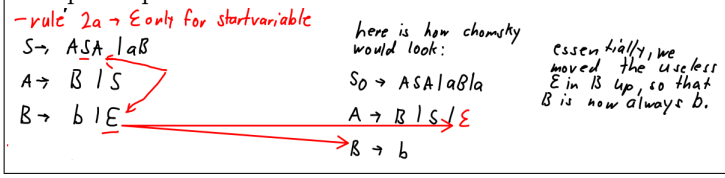
Note this includes ANY other formation, including aB, aa, ABC, aAB

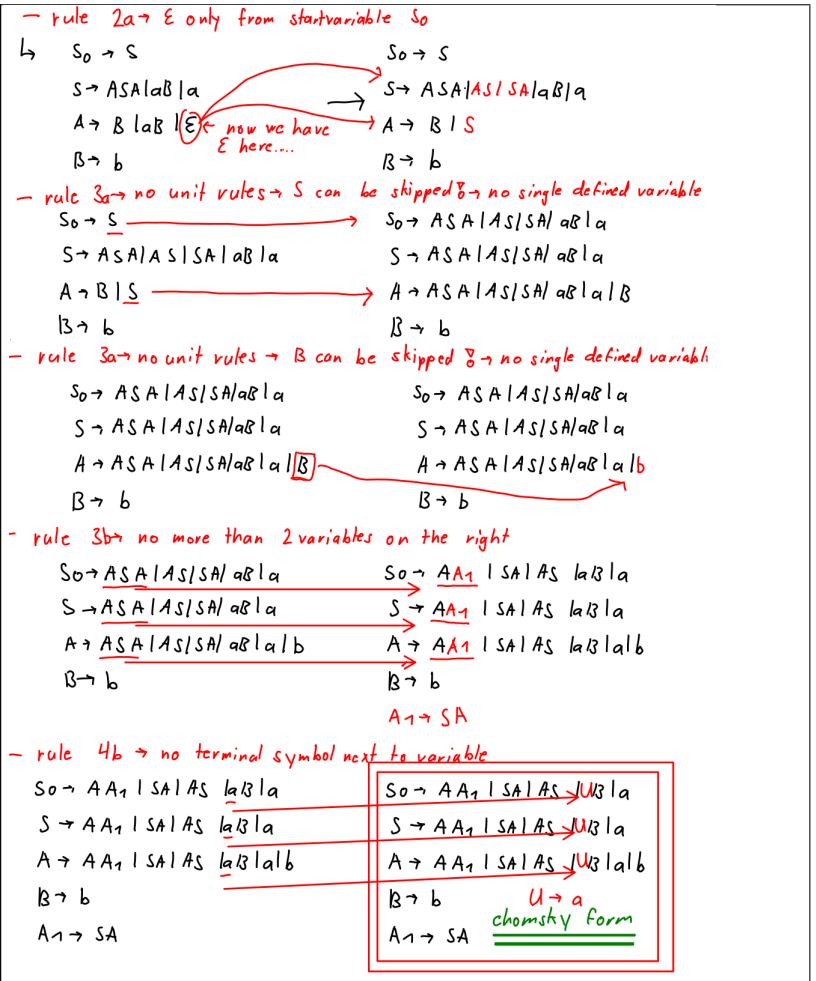
Transforming into Chomsky-Normalform

Transformation into Chomsky-Normalform

1. new starting variable $S_0 \to S$
2. ε- Rules : $\left.\begin{array}{l} A \to \varepsilon \\ B \to AC \end{array}\right\} \Rightarrow$ A is pointless, as it leads to ε. It can simply be emitted. $\Rightarrow \begin{cases} B \to AC \\ \to C \end{cases}$
3. Unit-Rules: $\left.\begin{array}{l} A \to B \\ B \to CD \end{array}\right\} \Rightarrow$ A leads to B, which simply leads to CD. Replace B with CD in A $\Rightarrow \begin{cases} A \to CD \\ B \to CD \end{cases}$
4. Chaining : $A \to u_1 u_2 \ldots u_n$
   $A \to u_1 A_1, A_1 \to u_2 A_2, \ldots, A_{n-2} \to u_{n-1} u_n$ and if u_i == terminalsymbol :
   $A_{i-1} \to U_i A_i, U_i \to u_i$.

In-depth explanation:





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 $A \to BC$ to generate a word with length $|w|$ out of S.
2. Apply $|w|$ Rules of form $A \to a$ 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 $G = (V, \Sigma, R, S)$

$A \overset{*}{\Rightarrow} w$ possible??

- $w = \varepsilon$: $A \overset{*}{\Rightarrow} \varepsilon \quad \Leftrightarrow \quad A \to \varepsilon \in R$
- $|w| = 1$: $A \overset{*}{\Rightarrow} w \quad \Leftrightarrow \quad A \to w \in R$
- $|w| > 1$: divide and conquer $A \overset{*}{\Rightarrow} w \quad \Rightarrow \quad \exists \begin{cases} A \to BC & \in R \\ w = w_1 w_2 & w_i \in \Sigma^* \end{cases}$ with $\begin{cases} B \overset{*}{\Rightarrow} w_1 \\ C \overset{*}{\Rightarrow} w_2 \end{cases}$

Wort: ()[()]

$S \to AB \mid CD \mid AT \mid CU \mid SS$
$T \to SB$
$U \to SD$
$A \to ($
$B \to)$
$C \to [$
$D \to]$

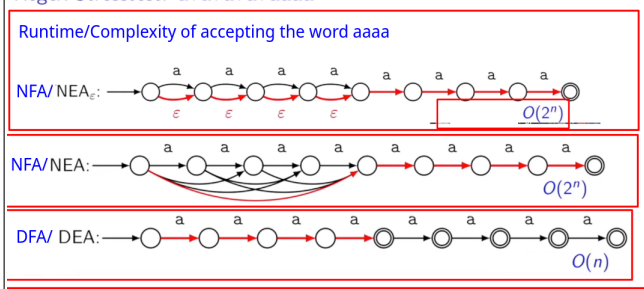
	(	)	[	(	)	]
0	{A}	{B}	{C}	{A}	{B}	{D}
1	{S}	{}	{}	{S}	{}	
2	{}	{}	{}	{U}		
3	{}	{}	{S}			
4	{}	{}				
5	{S}					

okay this works, but what is the catch?
Well $O(n^3)$ ... hahahaa .....

Performance Comparison NFA/DFA

Regex-Stresstest: 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 $A_{pq} \to A_{pr}A_{rq}$

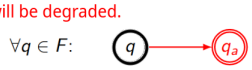
from p to r and from r to q.

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

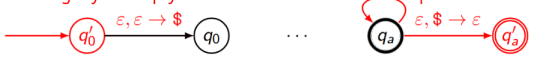
However, this is only the case because it is implied in the name, it is otherwise just a Non-Terminalsymbol!!!

Turning the regular Stack Automaton to a CFG:

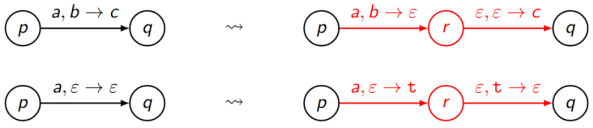
1. only 1 accepting state, q will be degraded. all previous accepting states now lead to qa



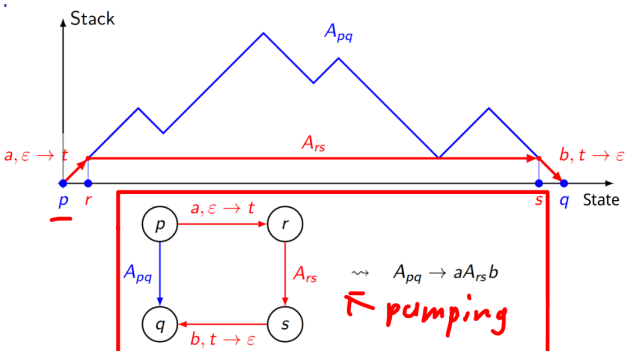
2. on input and output we need to add and remove the $ symbol to signify an empty stack

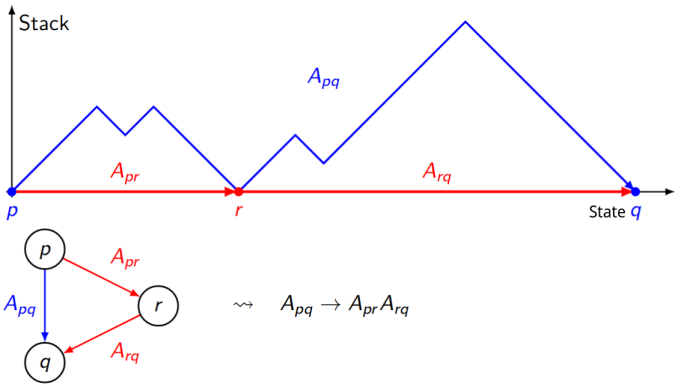


3. now all transitions need to involve the stack



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.

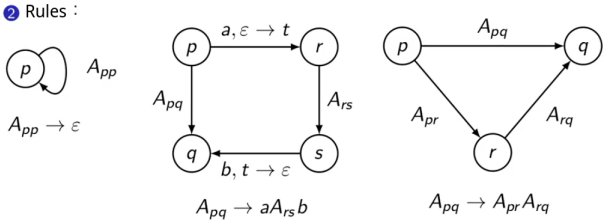




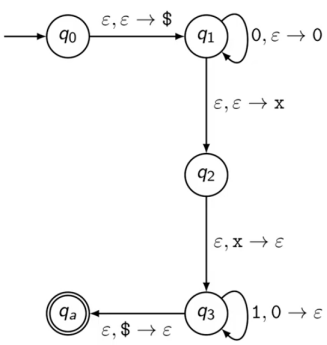
Derived Rules from the Automaton:

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

1. Startvariable: A_{q_0,q_a}
2. Rules :



PDA → CFG: example 1^n0^n



Grammar

$A_{q_0q_a} \to \varepsilon A_{q_1q_3}\varepsilon$
$A_{q_1q_3} \to 0A_{q_1q_3}1$
$\to \varepsilon A_{q_2q_2}\varepsilon$
$A_{q_2q_2} \to \varepsilon$

Simplification, and Chomsky compliant!!

$S \to 0S1$
$\to \varepsilon$

Grammar for: $L = \{0^n1^n \mid n \geq 0\}$

every CFG can be converted to a Stack Automaton.

Regularity Check for CFG / Pumping Lemma for CFG

Grammar G in CNF

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

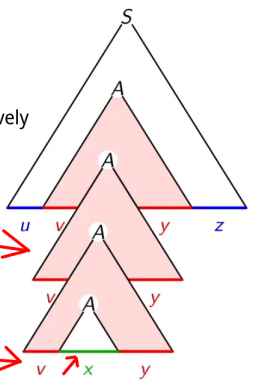
Reusable Variable, RECURSIVE Variable

$|w| \geq N$ 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.

$A \overset{*}{\Rightarrow} vxy$

$A \overset{*}{\Rightarrow} x$



Pumping

$A \overset{*}{\Rightarrow} vxy$ instead of simply $A \overset{*}{\Rightarrow} x$

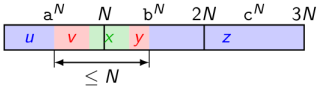
Pumping Lemma for CFL

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: $\{a^nb^nc^n \mid n \geq 0\}$

1. Claim : L = CFL
2. Pumping length N
3. Word : $w = a^Nb^Nc^N$
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: <variable-name>
- single Symbols: A
- Strings: 'Example'
- Rules: <variable-name> ::= expression

Haskell says hello!

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

```
<expression> ::= <expression> + <term> | <term>
<term>       ::= <term> * <factor> | <factor>
<factor>     ::= ( <expression> ) | <number>
<number>     ::= <number> <digit> | <digit>
<digit>      ::= 0 | 1 | 2 | 3 | 4
                 | 5 | 6 | 7 | 8 | 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:

$expr \rightarrow term\ expr'$ | $term \rightarrow factor\ term'$ | $factor \rightarrow (\ expr\ )$
$expr' \rightarrow +\ term\ expr'$ | $term' \rightarrow *\ factor\ term'$ | $\rightarrow integer$
$\rightarrow \varepsilon$ | $\rightarrow \varepsilon$ |

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 bc lol

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!

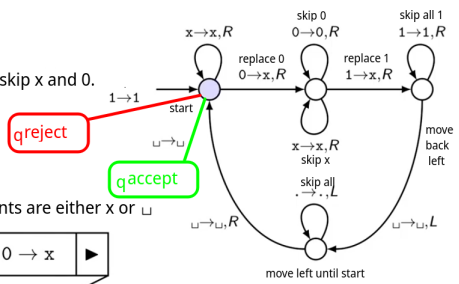
The stack problem

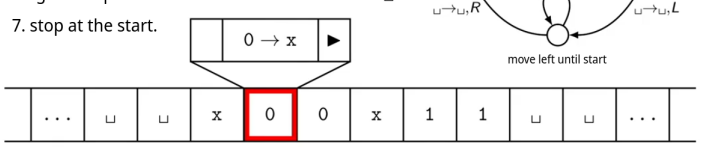
The stack has enabled us to understand the Language 0^n1^n
However the language $a^nb^nc^n$ remains uninterpretable.
This can be solved by extending the stack.
One idea is to double the stack, giving us access to every element
The only problem is that this is a slow and tedious process.
This is where a turing machine comes into play with it's 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 ␣
5. move to the left until ␣
6. go to step 1 until all elements are either x or ␣
7. stop at the start.

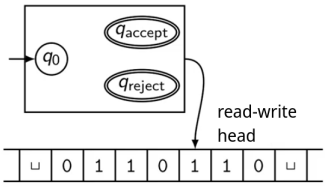




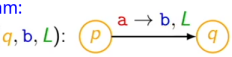
Definition Of Deterministic Turing Machine

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

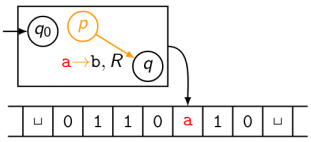
- Q : States
- Σ : Alphabet
- Γ : Band-Alphabet: $\sqcup \in \Gamma \backslash \Sigma$
- δ : $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
- $q_0 \in Q$: Startstate
- $q_{accept} \in Q$: accepting state
- $q_{reject} \in Q$: rejecting state



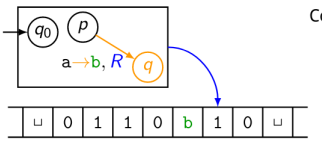
- Transition possible if a under read-write head
- Current field of read-write head is overwritten with b
- movement head: left L, right R

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

Before



After



Programstructure

- Input-Word on Band
- Read-Write Head positioned on 1. element
- start Machine, complete t(w) transitions
- Machine stops in qaccept or qreject

Complexity: $t(w)$, $t(n) = \max\{t(w) \mid |w| \leq n\}$

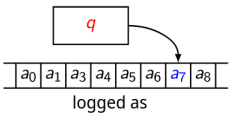
of a TM recognized language

M a Turing Machine

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

Calculation History

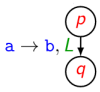
Notation for Machine State



logged as

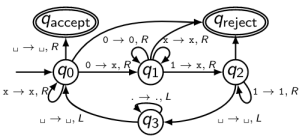
$a_0 a_1 a_2 a_3 a_4 a_5 a_6 q a_7 a_8$

Transition



$a_1\ a_2\ a_3\ p\ a\ a_4\ a_5$
$a_1\ a_2\ q\ a_3\ b\ a_4\ a_5$

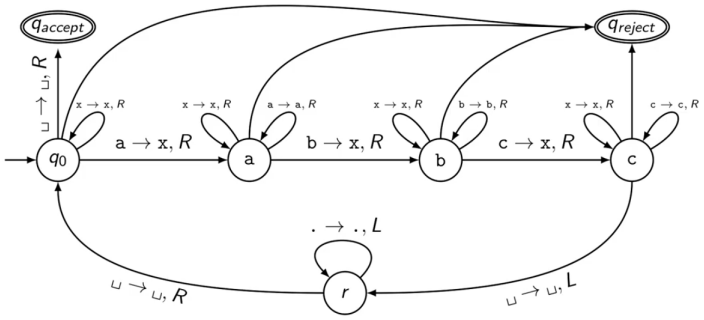
Protocoll of a Calculation



␣ q0 0 0 1 1 ␣
␣ x q1 0 1 1 ␣
␣ x 0 q1 1 1 ␣
␣ x 0 x q2 1 ␣
␣ x 0 x 1 q2 ␣
␣ x 0 x q3 1 ␣
␣ x 0 q3 x 1 ␣
␣ x q3 0 x 1 ␣
␣ q3 x 0 x 1 ␣
...

$a^nb^nc^n$

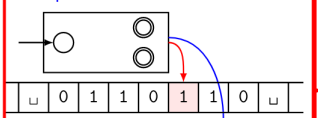
Before we asked the quetion of how we could create a Turing Machine, that understands the language $a^nb^nc^n$
Is this already possible ? YES!



Turing Machine Variants

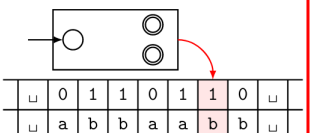
different Alphabet
Γ Binary encoded Bandalphabet
$\Gamma_0 = \{0, 1\}$
Complexity $O(t(n))$

multiple Bands



Complexity $O(t(n)^2)$

Multirow Turing Machine



Complexity $O(t(n))$

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

Nondeterministic TM

Transitioning Function

on each step max N different possibilities

$\delta: Q \times \Gamma \to P(Q \times \Gamma \times \{L, R\})$

$\Rightarrow$ 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 posibilities $\max N^{t(n)}$

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

$$L_P = \left\{ w \in \Sigma^* \;\middle|\; \begin{array}{l} w \text{ is the solution of} \\ \text{Problem P} \end{array} \right\}$$

Examples:

Empty-Problem :

$$E_{\text{DEA}} = \left\{ \langle A \rangle \;\middle|\; \begin{array}{l} A \text{ a DFA, and} \\ L(A) = \emptyset \end{array} \right\}$$

Equality Problem :

$$EQ_{\text{CFG}} = \left\{ \langle G_1, G_2 \rangle \;\middle|\; \begin{array}{l} G_i \text{ CFGs and} \\ L(G_1) = L(G_2) \end{array} \right\}$$

Acceptance Problem :

$$A_{\text{DEA}} = \left\{ \langle A, w \rangle \;\middle|\; \begin{array}{l} A \text{ a DFA that} \\ \text{accepts } w \end{array} \right\}$$

Halting Problem :

$$HALT_{\text{TM}} = \left\{ \langle M, w \rangle \;\middle|\; \begin{array}{l} M \text{ stops at} \\ \text{Input } w \end{array} \right\}$$

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

$w = \texttt{a=}a\texttt{,b=}b\texttt{,c=}c\texttt{,x=}x$

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

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

ε 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?

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?

$\texttt{2,1.414213562} \in L$?

or $\langle \text{n,x} \rangle \longrightarrow \langle A \rangle, \in L$?

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 <G> 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

$$HALT\varepsilon_{\text{TM}} = \left\{ \langle M \rangle \;\middle|\; \begin{array}{l} M \text{ is a turing machine} \\ \text{and } M \text{ stops on empty} \\ \text{band} \end{array} \right\}$$

is not computable

Halting Problem

$$HALT_{\text{TM}} = \left\{ \langle M, w \rangle \;\middle|\; \begin{array}{l} M \text{ is a turing machine} \\ \text{and } M \text{ stops on input w} \end{array} \right\}$$

is not computable

Halting Problem

The idea is that we have a Decider H and a program D that always returns the op

This D encompasses H, which means we give input to D, which gives the input to

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 $f: \Sigma^* \to \Sigma^*$

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

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

Computability

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

Proof

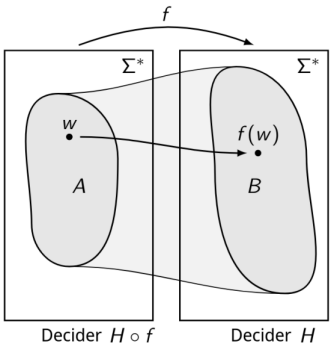
H is a decider for B, then is $H \circ f$ a decider for A.

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: $A_{\text{TM}} \leq HALT\varepsilon_{\text{TM}}$

A_{TM} is not computable

Does the machine M accept the word w?

There is no Decider

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

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

M accepts $w \quad \Leftrightarrow \quad S$ stops

is $HALT\varepsilon_{\text{TM}}$ computable?

Does the machine stop?

Decider for A_{TM}

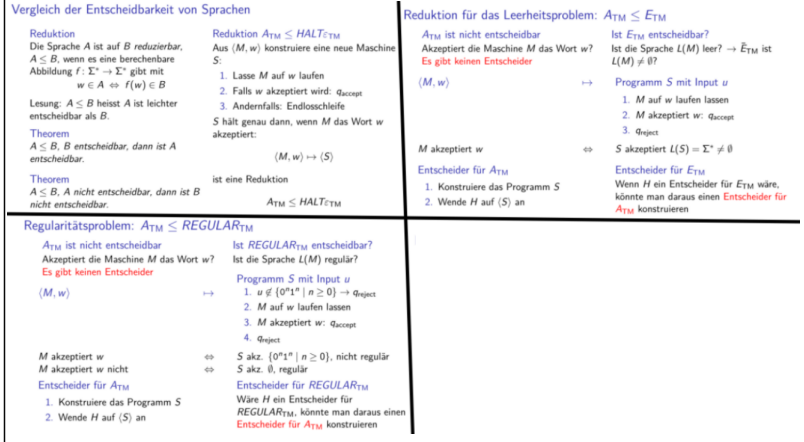
1. Create program S
2. Use H on $\langle S \rangle$

Decider for $HALT\varepsilon_{\text{TM}}$

If H is a decider for $HALT\varepsilon_{\text{TM}}$ 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:



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

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

Definition
A property is nontrivial is 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

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

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

A_{TM} is not computable
Does the machine M accept the word w?

is P_{TM} computable?
Does the language L(M) have the property P?

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

$\langle M, w\rangle \quad \mapsto$ Programm S mit Input u

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

M accepts $w \quad \Leftrightarrow \quad S$ accepts Σ^*, has P
M rejects $w \quad \Leftrightarrow \quad S$ accepts $L(T)$, doesn't have P

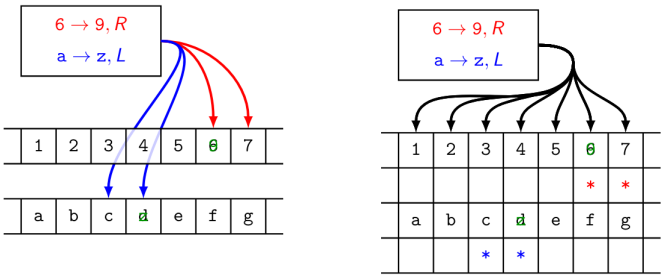
Overview of Computability

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.

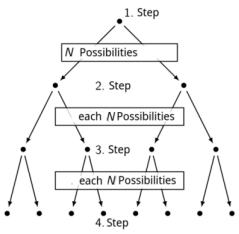
difference of Turing Machine Performance:



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

Scalability

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

Security

Verificiation 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 pylonomial time. While the solution is done with an NTM.

$w \in L \quad \Leftrightarrow \quad V \quad \langle w, c\rangle$

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

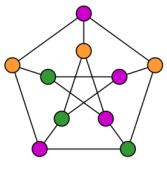
Computable?
yes, all $\leq (n^2)^{n^2 \times n^2}$ possibilities have to be checked
certificate? $c =$ all missing numbers
verification-algorithm

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? -> NP Problem!
This problem can be converted to another problem, for example to the timetable p

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



Polynomial Reduction

As the example above has already shown, we can convert problems.
Just like with regular reduction, we have converted an easier problem to a more complex one that we already have a verifier for.

If A and B are computable languages.
Then there is a projection from
A to B. see Reduction

$f : \Sigma^* \to \Sigma^* : w \mapsto f(w)$

1. $w \in A \Leftrightarrow f(w) \in B$ (Reduction)
2. $f(w)$ needs to be calculatable in polynomial time within $|w|$

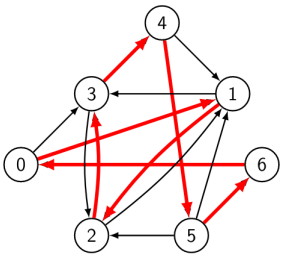
$$A \in P \Rightarrow B \in P, \quad A \notin P \Rightarrow B \notin P$$
$$A \in NP \Rightarrow B \in NP, \quad A \notin NP \Rightarrow B \notin NP$$

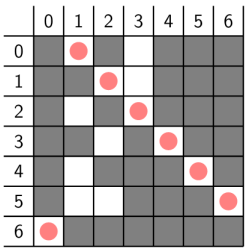
This is the same theorem as with reduction.
If A is in P then B is in P.
If A is in NP then B is in NP
etc...

$A \leq_P B.$ == A is polynomially easier to compute than B

Hampath

Sadly, not every problem is straight forward to convert, some require a bit of ingenuity.
Reduction Hampath into a quizz:



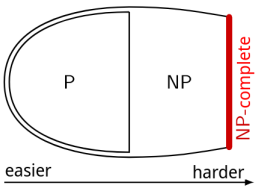


Reductioncomplexity : $O(n) \Rightarrow HAMPATH \leq_P$ quizz

P and NP

Definition
A Language is NP-Complete, if every other NP Language can be reduced to this.
Aka NP-Complete means "hardest language" to compute (There are multiple of these!!)

$A \leq_P B \quad \forall A \in \text{NP}$



Equivalence
All NP-Complete Languages are equally "hard"

$$A, B \text{ NP-complete} \Rightarrow \begin{cases} A \leq_P B \\ B \leq_P A \end{cases}$$

Rule

$$\left.\begin{array}{l} A \text{ NP-complete} \\ B \in NP \\ A \leq_P B \end{array}\right\} \Rightarrow B \text{ NP-complete}$$

If A is in NP then B must be in NP.

Is P and NP the same? If so this would have major consequences!
Our cryptography would not be secure anymore! Problem is we can't be sure about whether or not this is the case, P might be different to NP, perhaps not
We might find an algorithm to solve sudoku in polynomial time, or might not.

SAT

SAT is a Logical formula that needs to be fulfilled.
Also called Boolean satifiability problem

Fill-in Puzzle

A polynomial fill-in puzzle is a n x m table into which we enter symbols of an alphabet Σ, so that it coheres to logical rules, which can be evaluated in polynomial time.

Variables

$$x_{ijc} = \text{True} \Leftrightarrow \begin{cases} \text{Field } (i,j) \text{ contains} \\ \text{symbol c} \end{cases}$$

Only 1 symbol in field (i,j)

$$\varphi_{ij} = \bigvee_{c \in \Sigma} \Big(x_{ijc} \wedge \neg \bigvee_{d \neq c} x_{ijd} \Big)$$

c and no other symbol in (i,j)

Formula

The formula φ is the AND function over the rules that have been defined x_{ijc}
The formula returns true if all rules have been adhered to

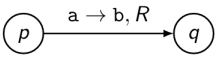
Every fill-in puzzle can be reduced to SAT.

Computation History

State of TM

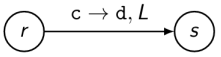
- Band content, symbols etc
- Positon and State of the TM. State is always the previous position, aka last computation
  State: s/q Position p/r

Transition R



$\ldots a_1\ a_2\ a_3\ p\ \mathrm{a}\ a_4\ a_5\ a_6 \ldots$
$\ldots a_1\ a_2\ a_3\ \mathrm{b}\ q\ a_4\ a_5\ a_6 \ldots$

Transition L



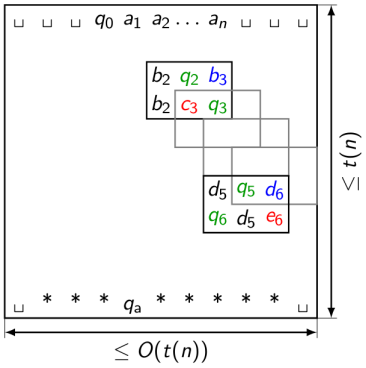
$\ldots a_1\ a_2\ a_3\ r\ \mathrm{c}\ a_4\ a_5\ a_6 \ldots$
$\ldots a_1\ a_2\ s\ a_3\ \mathrm{d}\ a_4\ a_5\ a_6 \ldots$

Cook Levins Law

1. SAT is NP-Complete
2. if $A \in NP$
   There exists an NTM for A that computes in polynomial time $O(t(n))$
   Therefore A can be reduced to SAT in polynomial time
3. $A \leq_P SAT$

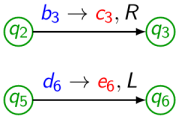
The way to do this, is by turning A into a fill-in puzzle
From there on it is easy, as we already know that fill-in puzzles can be reduced to SAT!

Reduction to polynomial fill-in puzzle



Computation History

1. 1. row: Starting State
   $w = a_0 a_1 a_2 \ldots a_n$
2. last row : $q_{\text{accept}} (= q_a)$ or q_{reject}
3. other rows : TM-Transitions



There are $O(t(n)^2)$ 2×3- Rectangles that need to be "correct"

$O(t(n)^2)$ -> Amount of symbols and states * bandlength

3SAT to k-Clique

$$3SAT \leq_P k\text{-}CLIQUE$$

Given 3SAT
φ with 3 Variables in conjunctive Normalform

Question can φ return True?

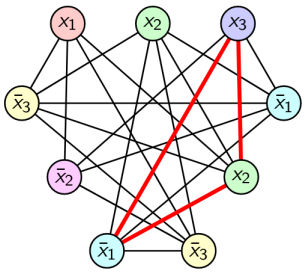
Given Graph G

Wanted k-nodes that are connected to each other. (k-Clique)

$(x_1 \vee x_2 \vee x_3)$

φ

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



Connect each node to every node that doesn't contradict itself (NOT $x1$ to $\overline{x}1$)
Only connect nodes from a different clause, (the conjunctions from 3SAT)
If you can then make a triangle with a node from each clause,
You have a TRUE from 3SAT. :)

3SAT to SUBSET-SUM

$3SAT \leq_P SUBSET\text{-}SUM$

3SAT

$$\varphi = \overbrace{(x_1 \vee x_2 \vee x_3)}^{c_1} \wedge \overbrace{(\bar{x}_1 \vee x_2 \vee \bar{x}_3)}^{c_2} \wedge \overbrace{(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)}^{c_3}$$

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

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

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
y_3			1	1	0	0
z_3			1	0	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

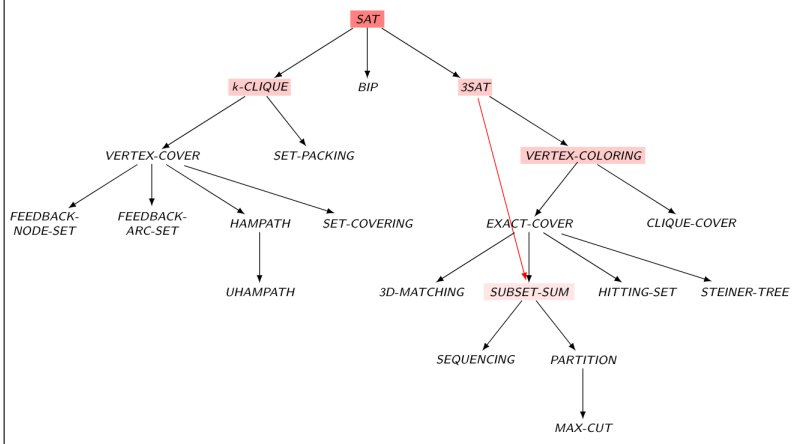
try: x1 =1 x2=1 x3=1
we get the 3 numbers inside the boxes above.
add them up,

we have 1,1,1 ,3, 1, 0
we can add g2,h2 and g3,h3, however
this only leads to 3 2 0
This means that x1=1 x2=1 x3=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	Regular PC
States Q	Sates of CPU: Bits etc
Practically infinite memory -> Band	virtual Memory (practically infinite)
Read-Write Head	Adress Register, Program Counter
Stop -> q_{accept} or q_{reject}	EXIT_SUCCESS, EXIT_FAILURE
TM made for specific problem	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
  $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
- 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

$M_2 \leq_s M_1$ M1 simulates M2

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

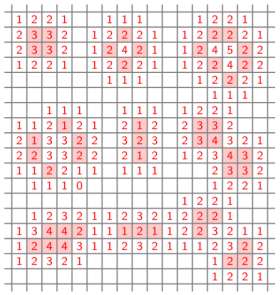
Rules

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



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 lanugage

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

$c : A \to \Sigma^*$

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: $\mathbb{N}_0$ $0 \to \infty$
- Variables $x_1, x_..$: $\mathbb{N}_0$
- Assigment: $x_i := c$
- Addition: $x_i := x_j + c$
- Subtraction: $x_i := x_j - c$

  Note is $c \geq x_j$ then $x_i = 0$

!Not Included!:

- NO addition of variables: $x_i + x_j$
- NO subtraction of variables: $x_i - x_j$
- NO division or multiplication

Non-Turing Complete Languages

These types of languages ALWAYS halt! No halting problem!

LOOP: not turing complete

Control Structure

run a Program P exactly x_i times

```
LOOP x_i DO
    P
END
```

Addition / Subtraction

$x_i := x_j \pm x_k$

```
x_i := x_j
LOOP x_k DO
    x_i := x_i ± 1
END
```

Multiplication

$x_i := x_j * x_k$

```
x_i := 0
LOOP x_k DO
    x_i := x_i + x_j
END
```

OR:

```
x_i := 0
LOOP x_k DO
    LOOP x_j DO
        x_i := x_i + 1
    END
END
```

Conditional Statement

```
IF x_i THEN
    P
END
```

implemented as:

```
y := 0
LOOP x_i DO y := 1 END
LOOP y DO P 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

```
y := x_i
WHILE y > 0 DO
    y := 0; P
END
```

LOOP in WHILE

```
y := x_i
WHILE y > 0 DO
    P
    y := y - 1
END
```

GOTO

Definition of GOTO

- Base elements like in LOOP
- Series of statements

  $M_1 : A_1$
  $M_2 : A_2$
  $\ldots : \ldots$
  $M_k : A_k$

Unconditional Statements

M_{k+1} : IF $x_i = c$ THEN GOTO M_{k+n}
$\ldots : \ldots$
$M_{k+n} : A_l$

Conditional Statements

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

contraction:

$M_{k+1} :$
M_{k+2} : GOTO M_{k+n}
$\ldots : \ldots$
$M_{k+n} : A_l$

Implementation GOTO in WHILE

GOTO-Program:

$M_1 : A_1$
$M_2 : A_2$
$\ldots : \ldots$
$M_k : A_k$

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
```

modified statement A'_i

- A_i conditional Jump statement

  M_l : IF $x_i = c$ 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; $z = z + 1$;

Implementation WHILE in GOTO

Translation

A WHILE-Construct

$$\texttt{WHILE } x_i > 0 \texttt{ DO } P \texttt{ END}$$

Will be translated to a GOTO-Code segment

M_l : IF $x_i = 0$ THEN GOTO M_{l+3}
M_{l+1} : P
M_{l+2} : GOTO M_l
M_{l+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