

# 1 Basics

## 1.1 Words and Languages

- alphabet $\Sigma$ is a finite, non-empty set. ex: $\{A, \dots, Z\}$
- language $L$ over an alphabet $\Sigma$ is a set of words over $\Sigma$: $L((ab)^*) = \{\epsilon, ab, abab, ababab, \dots\}$
- Regular vs. Not Regular
  - Regular = Track a finite set of states
    $L_1 = \{a^n b^* | n \geq 0\}$: Only need to know we are still reading $a$ or already in $b$ part $\implies$ finite states & finite state memory
  - Non-Regular = Track infinite set of states. Needs infinite memory.
    $L_2 = \{a^n b^n | n \geq 0\}$: Must count $a$ to match same number of $b$ $\implies$ infinite states & infinite state memory
- Language Sets: $REG \subsetneq CFL \subsetneq DEC$
  When looking at a Lang., think about which tool I can use to model it.
  - $REG$ular Languages: DFA & NFA
    Finite nr. of states + no memory.
  - $CFL$ Context-Free Languages: PDA
    Finite nr. of states + 1 unbounded stack
  - $DEC$idable Languages: TM that always halts
    Finite nr. of states + infinite tape.

# 2 Regular Languages

Lang. $L$ is regular $\Leftrightarrow \exists$ reg. expression $\alpha$ s.t. $L = L(\alpha)$
language is described by a regular expression $\alpha$
$\Leftrightarrow$ is recognized by a DFA/NFA

Examples:
- $L = \{a^n | n \geq 0\} = a^*$
- $L = \{w \in \{0,1\}^* |$ w has even number of 1s $\}$
  2 state DFA that flips between even and odd when it sees a 1
- $L = \{cat, dog, fox\}$ Finite Languages

## 2.1 Operations for Regular Expressions

### 2.1.1 String Operations

- Repetition: $u^n$
  $b^* \equiv$ arbitrary many bs (Kleene Star for Strings)
  $\equiv \{\epsilon_{b,}\ b,\ bb,\ bbb,\ \dots\}$
- Concatenation: $a \cdot v$
  $ab^* \equiv$ one a followed by arbitrary number of bs
  $\equiv \{a,\ ab,\ abb,\ abbb,\ \dots\}$ $u^0 = \epsilon, u^1 = u, u^2 = uu, \dots$
- Choice: $u + n$
  $b + c^* \equiv$ one b or an arbitrary number of cs
  $\equiv \{b, \epsilon_{,}\ c,\ cc,\ ccc,\ \dots\}$

### 2.1.2 Closed operations on Regular Languages

- $L_1 \circ L_2 = \{uv \mid u \in L_1, v \in L_2\}$
- $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$
- $L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$
- $\Sigma^* \setminus L(A_1)$ Complement
- $L_1^*$ Kleene Star

$L(\emptyset) = \emptyset$ • $L(\epsilon) = L(\epsilon)$ • $L(\sigma) = \sigma$, for each $\sigma \in \Sigma$

- If $\alpha$ and $\beta$ are regular expressions, then:
  - $L((\alpha\beta)) = L(\alpha) \cdot L(\beta)$
  - $L((\alpha + \beta)) = L(\alpha) \cup L(\beta)$
  - $L((\alpha^*)) = L(\alpha)^*$

### 2.1.3 EX: Prove $L$ not regular by closure properties

Let $L_1 = \{wtw | w, t \in \{00^*, 11^*\}\}$, $L_2 = \{0^a 10^a | a \geq 1\}$. $L_2$ not regular. Prove $L_1$ not regular.

- Assume for sake of contradictino $L_1$ regular
- Find $L_3 = \{0^a 10^b | a, b \geq 0\}$ to construct:
  $L_2 = L_1 \cap L_3 = \{0^a 10^a | a \geq 1\}$
- Contradiction: $L_1 \wedge L_3$ regular, so due to closure $L_2$ regular. But $L_2$ given not regular $\Rightarrow L_1$ not regular

## 2.2 Equivalences

$\alpha \equiv \beta \iff L(\alpha) = L(\beta)$

Two regular expressions are equivalent if they describe the same language.

### 2.2.1 Equivalence Rules

| | | | |
|---|---|---|---|
| Associativity | $\alpha + (\beta + \gamma) \equiv (\alpha + \beta) + \gamma$<br>$\alpha(\beta\gamma) \equiv (\alpha\beta)\gamma$ | Distributivity | $\alpha(\beta + \gamma) \equiv \alpha\beta + \alpha\gamma$<br>$\epsilon\alpha \equiv \alpha \equiv \alpha\epsilon$ |
| Commutativity | $\alpha + \beta \equiv \beta + \alpha$ | Idempotence | $(\alpha^*)^* \equiv \alpha^*$ |
| Neutral elements | $\emptyset + \alpha \equiv \alpha \equiv \alpha + \emptyset$<br>$\epsilon\alpha \equiv \alpha \equiv \alpha\epsilon$ | Zero element | $\emptyset\alpha \equiv \emptyset \equiv \alpha\emptyset$<br>$\emptyset^* \equiv \epsilon$ |

### 2.2.2 Proof Equivalence of 2 Regular Languages

Interpret Langauges as sets of strings and show equivalence through equality of sets.

1. Show that $M_1 \subseteq M_2$
2. Show that $M_2 \subseteq M_1$
3. $\forall w \in M_1$, show that $w \in M_2$
4. $\forall w \in M_2$, show that $w \in M_1$

### 2.2.3 EX: Proof Equivalence of 2 Regular Languages

Show that $\alpha(\beta+\gamma) \equiv \alpha\beta+\alpha\gamma$. In other words show that $L(\alpha(\beta + \gamma)) = L((\alpha\beta + \alpha\gamma)$

1. Let $\alpha, \beta, \gamma$ be arbitrary.
2. Here we only show: $L(\alpha(\beta + \gamma)) \subseteq L(\alpha\beta + \alpha\gamma)$
3. Let $w$ be an arbitrary string in $L(\alpha(\beta + \gamma))$ :
4. Then, $u \in L(\alpha)$ and $v \in L(\beta + \gamma)$ with $w = uv$
5. Then, $v \in L(\beta)$ or $v \in L(\gamma)$
6. We distinguish between two cases:
   (a) $v \in L(\beta)$ : Since $u \in L(\alpha)$ and $v \in L(\beta), w = uv \in L(\alpha\beta) \implies w \in L(\alpha\beta + \alpha\gamma)$
   (b) $v \in L(\gamma) : w = uv \in L(\alpha\gamma) \implies w \in L(\alpha\beta + \alpha\gamma)$
7. Same idea for $L(\alpha\beta + \alpha\gamma) \subseteq L(\alpha(\beta + \gamma))$.

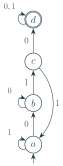
### 2.2.4 Proof of Non-Equivalence

Find counter example.
ex: $a(a + b)^* \equiv (a + b)^* a$, baba $\notin L(a(a + b)^*)$

# 3 Finite Automata

## 3.1 DFA: Deterministic Finite Automata

Automata where for every state & symbol, there is only 1 next state. i.e. symbol 0 may only go to $q_i$ and not $q_j$



- DFA $A = (Q; \Sigma, \delta, s, F)$ consists of:
  - $Q_{010} = \{a, b, c, d\}$ finite set of states
  - $\Sigma$ input alphabet
  - $\delta : Q \times \Sigma \to Q$ transition function
    takes single state & single input symbol. returns exactly one next state.
  - $s_{010} = a$ initial state
  - $F_{010} = \{d\}$ of accepting states

How to read: Beginning in the initial state, the automaton reads the input character by character. Switch states using transition function. Accepts input if the finishing state is in F.

- $L(A) = \{w \in \Sigma^* | A \text{ accepts } w\}$ = Language decided by $A$

### 3.1.1 $\delta^*$ Extended transition funct. (DFA Semantics)

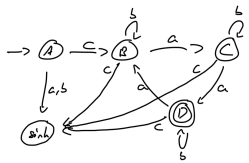
Extended transition function takes a state and an entire string (sequence of symbols) and returns the resulting state after processing the whole string.

- $\delta^* : Q \times \Sigma^* \to Q$ is defined inductively: $\delta^*(q, \epsilon) = q \circ \delta^*(q, u\sigma) = \delta(\delta^*(q, u), \sigma)$, for $u \in \Sigma^*, \sigma \in \Sigma$
- $A$ accepts $w \iff \sigma^*(s, w) \in F$

Automaton accepts string $\iff$ String applied to initial state yields accepting state.

### 3.1.2 Design a DFA for Language

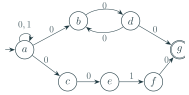
Let $L_1 = \{w \in \{a, b, c\}^* | w$ starts with c $\wedge$ contains no further $-c \wedge |w|_a mod\ 3 = 2\}$

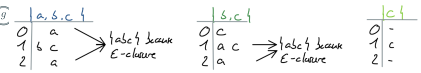


## 3.2 NFA: Non-Determ. Finite Automata

Automata where for every state & symbol, there can be many states. i.e. symbol 0 may go to $q_i$ or $q_j$

NFA $B = (Q, \Sigma, \delta, s, F)$ consists of:
- set $Q$ of states
- an input alphabet $\Sigma$
- transition relation $\delta \subseteq Q \times \Sigma \times Q$
- an initial state $s \in Q$
- a set $F$ of accepting states





- $L(A) = \{w \in \Sigma^* | s \xrightarrow{w} q, q \in F\}$.
  = set of all strings that have an accepting run in $A$. $p \xrightarrow{w} q$ : $\exists$ run from state $p$ to $q$

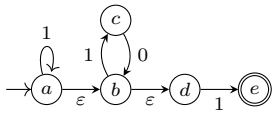
### 3.2.1 Run - Transition Relation (NFA Semantics)

Run $p$ through NFA is a sequence of the form state, symbol, state, etc. with valid relations between states. Run is a sequence $q_0, \sigma_1, q_1, \dots, \sigma_n q_n$ s.t.:

• $\forall i \in \{0, \dots, n\} : q_i \in Q$ • $\forall i \in \{0, \dots, n\} : \sigma_i \in \Sigma$ • $\forall i \in \{0, \dots, n\} : q_{i-1} \xrightarrow{\sigma_i} Q$

## 3.3 $\epsilon$-NFA

$\epsilon$-Transitions allow changing state without reading a symbol.
See right: Example Automaton for the RE $1^*(10)^*1$

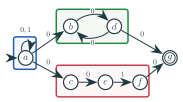


## 3.4 Convert: RE → NFA

T2.1: $\forall$ RE $\alpha$ $\exists \epsilon$-NFAs $A$ such that $L(\alpha) = L(A)$
Combine these NFA operations to build the NFA:

- $\alpha = \emptyset \equiv$ 
- $\alpha = \epsilon \equiv$ 
- $\alpha = \sigma \equiv$ 
- $\alpha = \beta\gamma \equiv$ 
- $\alpha = \beta^* \equiv$ 
- $\alpha = \beta + \gamma \equiv$ 

### 3.4.1 Example

RE: $(0+1)^*(00(00)^* + 001)0 \implies$ 

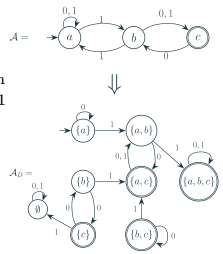
## 3.5 Convert: NFA → DFA (Powerset Automaton)

T3.1 $\forall$ NFA $A$ $\exists$ DFA $A_D$ with $L(A_D) = L(A)$

- Powerset automation is DFA created by combining multiple states of an NFA into a single one s.t. the transitions are deterministic.

### 3.5.1 EX: NFA → DFA (Powerset Automaton)

- powerset = "which NFA states are currently active after reading a symbol"
- Use tables to compute destination state for each transition. See 3.6.1
  Thought process: from a: if i apply 1, I can reach a or b. from a,b if i apply 1 from a or b, i can reach a,b,c. etc.
- Don't forget sink state $\emptyset$ or write: all missing transitions lead to sink state.



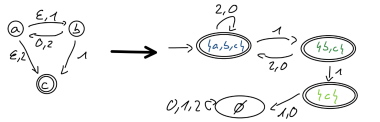
## 3.6 Convert: $\epsilon$-NFA → DFA (Powerset Automaton)

T3.2 $\forall$ $\epsilon$-NFA $A$ $\exists$ DFA $A_D$ with $L(A_D) = L(A)$

### 3.6.1 EX: $\epsilon$-NFA → DFA

Works same as NFA → DFA, but also account for $\epsilon$-Transitions.

- Initial state: $\epsilon$-closure(s) • $\epsilon - closure(p) = \{q | p \xrightarrow{\epsilon} q\}$
  Set of all reachable states from p without reading a symbol



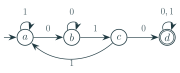


## 3.7 Convert: RE → DFA

RE → $\epsilon$ − $NFA \to DFA$

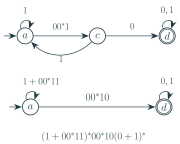
## 3.8 Convert: DFA → RE

T3.3 $\forall$ DFA $A$ $\exists$ an RE $\alpha$ with $L(\alpha) = L(A)$



### 3.8.1 DFA → RE by Hybrid Automaton (Don't use in exam)

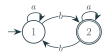
- Use hybrid automaton, whose transitions are labeled with regular expressions
- Successively remove states, until single regular expression is obtained.



### 3.8.2 DFA → RE by structural induction

- Let $A = (Q, \Sigma, \delta, s, F)$
- Let $L^k_{i,j}$ be set of all strings $w$ with: ○ $\delta^*(i, w) = j$ ○ $\forall$ proper prefixes $v \neq \epsilon$ of $w : \delta^*(i, v) \leq k$
- Claim: For each set $L^k_{i,j}$ there exists a reugular expression $\alpha^k_{i,j}$ with$L(\alpha^k_{i,j} = L^k_{i,j})$
- $k = 0$:
  - For $i \neq j : L^0_{i,j} = \{\sigma | \sigma \in \Sigma, \delta(i, \sigma) = j\}$
    $L^0_{i,j}$ consists of symbols that take automaton from state $i$ to state $j$
  - $L^0_{i,i} = \{\epsilon\} \cup \{\sigma | \sigma \in \Sigma, \delta(i, \sigma) = i\}$
    $L^0_{i,i}$ consists of $\epsilon$ and all symbols that lead from state $i$ back to state $i$
- Up to $k = |Q|$ (k = # of allowed intermediate states)
  $k > 0 : L^k_{i,j} = L^{k-1}_{i,j} \cup L^{k-1}_{i,k}(L^{k-1}_{k,k})^* L^{k-1}_{k,j}$
- Language defined by: $k = |Q|$
  $\wedge$ $i$ = initial state $\wedge$ $j$ = accepting state

### 3.8.3 EX: DFA → RE by structural induction



- $k = 0$: ○ $\alpha^0_{1,1} = \epsilon + a$ ○ $\alpha^0_{1,2} = b$ ○ $\alpha^0_{2,1} = b$ ○ $\alpha^0_{2,2} = \epsilon + a$
- $k = 1$:
  - $\alpha^1_{1,1} = \alpha^0_{1,1} + \alpha^0_{1,1}\left(\alpha^0_{1,1}\right)^* \alpha^0_{1,1} = (\epsilon + a) + (\epsilon + a)(\epsilon + a)^*(\epsilon + a) \equiv (a)^*$
  - $\alpha^1_{1,2} = \alpha^0_{1,2} + \alpha^0_{1,1}\left(\alpha^0_{1,1}\right)^* \alpha^0_{1,2} = (b) + (\epsilon + a)(\epsilon + a)^*(b) \equiv (a)^* b$
  - $\alpha^1_{2,1} = \alpha^0_{2,1} + \alpha^0_{2,1}\left(\alpha^0_{1,1}\right)^* \alpha^0_{1,1} = (b) + (b)(\epsilon + a)^*(\epsilon + a) \equiv b(a)^*$
  - $\alpha^1_{2,2} = \alpha^0_{2,2} + \alpha^0_{2,1}\left(\alpha^0_{1,1}\right)^* \alpha^0_{1,2} = (\epsilon + a) + (b)(\epsilon + a)^*(b) \equiv \epsilon + a + b(a)^* b$
- $\alpha^2_{1,2} = \alpha^1_{1,2} + \alpha^1_{1,2}\left(\alpha^1_{2,2}\right)^* \alpha^1_{2,2}$
  $= ((a)^* b) + ((a)^* b)\left(\epsilon + a + b(a)^* b\right)^* \left(\epsilon + a + b(a)^* b\right)$
  $\equiv (a)^* b\,(a + b(a)^* b)^*$

DFA Lang. corresponds to $L(\alpha^2_{1,2})$ as $\alpha^2_{1,2}$. As describes all words that take automaton from initial state 1 to accepting state 2 while allowing all intermediate states

# 4 Automata Minimization

## 4.1 Equivalence Class Automaton

= minimal DFA of $L$ defined by equival. classes of $\sim_L$.

- $x \sim_L y \implies$ States can be merged into single state.
- $\sim_L$ Equivalence Relation = binary relation that is reflexive, symmetric, transitive
- Equivalence Class of an equivalence relation $\sim$ is a maximal set $K$ of elements such that $x \sim y \forall x, y \in K$

## 4.2 $\sim_L$ Nerode Relation

- $x \sim_L y \iff \forall z \in \Sigma^*$: $(xz \in L \iff yz \in L)$
  $x$ and $y$ are Nerode equivalent $\iff$ for every possible continuation $z$, appending $z$ to $x$ and $y$ either makes both strings belong to $L$ or neither.

- Note that it might be that $z = \epsilon$. Example: $[\epsilon] = \{w \in L | w$ not contain subword aaa and not end with a$\}$
- P4.1 If index $(\sim_L) = k$ is finite $\implies$ $\exists$ DFA for $L$ with $k$ states. The equivalence class automaton $A_L = (Q, \Sigma, \delta, s, F)$ for $L$ is defined as follows:
  - Q is the set of equivalence classes of $\sim_L$
  - $s = [\epsilon]$
  - $F = \{[x] | x \in L\}$
  - $\forall x \in \Sigma^*, \sigma \in \Sigma : \delta([x], \sigma) = [x\sigma]$
- P4.2: If $A$ is a DFA with $k$ states for a Language $L$, then $Index(L) \leq k$
  $\implies$ nr. of equival. classes of the L will always be upper bounded by the nr. of states of the DFA
  $\implies$ $\nexists$ DFA with less states than the number of equivalence classes

#### 4.2.1 EX: Equivalence Classes of Nerode Relation

Give nerode equivalence classes of the language:
$L = \{w \in \{a, b\}^* | w$ has odd #a $\wedge$ last symbol b $\}$
Idea: Nerode rel. checks equiv. based on next appended symbol $z$. If append any symbol and in/out of language behavior unchanged for both words, then they're in same equiv. class.

- $[a] = \{w \in \{a, b\}^* | \ w$ has odd #a $\wedge$ last symbol a$\}$
- $[ab] = \{w \in \{a, b\}^* | \ w$ has odd #a $\wedge$ last symbol b$\}$
- $[\epsilon] = \{w \in \{a, b\}^* | \ w$ has even #a$\}$

### 4.3 Myhill-Nerode Theorem (Prove Lang. not regular)

Use to prove a Language is (not) regular by showing it has (in)finite equivalence classes.

- $L$ regular $\Leftrightarrow \sim_L$ has finite # equival. classes

#### 4.3.1 Prove $L = \{a^{n^2} | n \geq 1\}$ not regular by Myhill-Nerode

Prove $\exists$ infinite set s.t. every 2 distinct strings distinguishable.

- Let $S = \{a^{k^2} | k \geq 1\}$ infinite set of strings
  Let $s_i = a^{i^2}, s_j = a^{j^2}$ with $i, j \geq 1 \wedge j > i$
- Let $z = a^{2i+1}$ suffix s.t. 1 string in L & other not
- Check membership in $L$
  - $s_i z = a^{i^2} a^{2i+1} = a^{i^2+2i+1} = a^{(i+1)^2} \Rightarrow s_i z \in L$
  - $s_j z = a^{j^2} a^{2i+1} = a^{j^2+2i+1} \Rightarrow s_j z = a^{j^2+2i+1} \notin L$
    Because exponent lies between 2 perfect squares: $j^2 < j^2 + 2i + 1 < j^2 + 2j + 1 = (j + 1)^2$
- Thus $z$ distinguishes $s_i$ from $s_j$ for any pair of distinc integers $i, j \geq 1$ $\implies$ everay string in infinite set $S$ belongs to different equivalence class

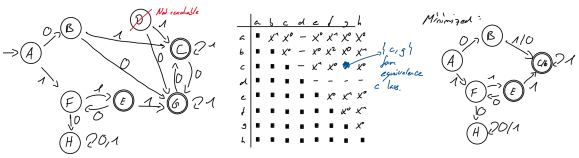
### 4.4 Uniqueness of Minimal Automaton

$A_1$ and $A_2$ are isomorphic $A_1 \cong A_2 \iff \exists$ bijection $\pi : Q_1 \to Q_2$ s.t.:

1. $\pi(s_1) = s_2$ initial state of 1st automaton is mapped to inital state of 2nd
2. $\forall q \in Q_1$ it holds that: $q \in F_1 \iff \pi(q) \in F_2$
   $\forall$ state in 1st automaton must be appropriately matched to a state in 2nd
3. $\forall q \in Q_1 \wedge \sigma \in \Sigma$ it holds that:
   $\pi(\delta_1(q, \sigma)) = \delta_2(\pi(q), \sigma)$
   $\forall$ transition in 1st automaton must be appropriately matched to transition in 2nd

$A_1 \cong A_2$ means $A_1$ and $A_2$ differ only in name of states. We get $A_2$ by renaming $A_1$'s states based on $\pi$

### 4.5 DFA Minimization w. Table Filling Algo




1. Remove all states that are not reachable from $s$
2. Table Filling Algo: Compute set of non-equivalent states $N(A) = \{(p, q) | p, q \in Q, \exists z : (\delta^*(p, z) \in F \not\Leftrightarrow \delta^*(q, z) \in F)\}$
   (a) Mark states with differing accepting status as $X^0$
   $N = \{(p, q), (q, p) | p \in F, q \notin F\}$
   (b) For all unmarked pairs:
   - Apply transitions to $p$ and $q$ to receive tuples $[\delta(P, x), \delta(Q, x)]$
   - Tuple marked in table $\implies$ mark $(p, q)$

   $N' = \{(p, q) \notin N | \exists \sigma \in \Sigma : (\delta(p, \sigma), \delta(q, \sigma)) \in N\}$
   (c) Continue while there are more unmarked steps
   $N = N \cup N'$ If $N' \neq \emptyset$, continue with 2. I.e. consider set of all non-equivalent states found so far. if there are more non-equivalent, repeat 2. else, finish.
3. Merge all unmarked state pairs into one state

## 5 Limits, Closure Properties, Algorithms

### 5.1 Pumping Lemma Regular Languages (Prove Lang. not regular)

- Show that a language $L_{ab} = \{a^n b^n | n \in \mathbb{N}_0\}$ is **not** regular.
  - Regular languages can be pumped: There must be a cycle that allows repeating a substring and staying in the language.
  - Find the cycle and show that i can find a word that's not in the Language $\implies$ $L$ not regular
- T5.1 (Pumping Lemma) $L$ is regular $\implies$ $\exists n$ s.t. each $w \in L$ with $|w| \geq n$ can be decomposed into $xyz$ s.t.:
  1. $y \neq \epsilon$ 2. $|xy| \leq n$ 3. $\forall k \geq 0 : xy^k z \in L(A)$
- T5.3 (Jaffe) $L$ is regular $\iff$ $\exists n$ s.t. each $w \in L$ with $|w| \geq n$ can written as $w = xyz$ in at least one way s.t.
  1. $y \neq \epsilon$ 2. $|xy| \leq n$ 3. $xy^k z \sim_L xyz$

#### 5.1.1 T5.2 Prove not Regular with Pumping Lemma

Let $L$ be a language. Assume $\forall n > 0 \ \exists w \in L$ with $|w| \geq n$ s.t. $\forall$ decomposition $w = xyz$ with: 1. $y \neq \epsilon$
2. $|xy| \leq n$ $\exists k_0$ s.t. $xy^k z \notin L \Rightarrow$ not regular

#### 5.1.2 ex: Prove Not Regular with Pumping Lemma

- $L_{a<b} = \{w \in \{a, b\}^* | \#_a(w) < \#_b(w)\}$
  langauge where nr. of $a$'s strictly smaller than nr. of $b$'s
- Let $n \in \mathbb{N}$ be arbitrary.
  Let $w = a^n b^{n+1} \in L_{a<b}$ $\implies$ $|w| = 2n + 1 \geq n$
- Let $x, y, z$ be arbitrary with $w = xyz$ and
  1. $y \neq \epsilon$ $\implies$ y contains at least 1 a
  2. $|xy| \leq n$ $\implies$ y contains no $b$
- Let $k \geq 2$:
  $\Rightarrow xy^2z$ contains at least $n + 1$ $a$'s
  $\wedge$ at most $n + 1$ $b$'s
  $\Rightarrow xy^2z \notin L_{a<b}$
  $xy$ contains only $a$ because 1. & 2. We pump $y$ to get more $a$ than $b$'s in $w$.
  $\Rightarrow L_{a<b}$ not regular

### 5.2 Synthesis of Finite Automata

#### 5.2.1 Boolean Operations

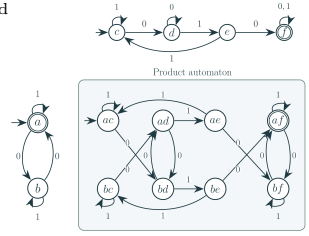
- T5.3 Let $A_1, A_2$ be DFAs. Then DFAs can be constructed for the following languages:

| | Nr. of States | Time |
|---|---|---|
| $L(A_1) \cap L(A_2)$ | $\lvert Q_1\rvert \lvert Q_2\rvert$ | $\mathcal{O}(\lvert Q_1\rvert \lvert Q_2\rvert \lvert\Sigma\rvert)$ |
| $L(A_1) \cup L(A_2)$ | $\lvert Q_1\rvert \lvert Q_2\rvert$ | $\mathcal{O}(\lvert Q_1\rvert \lvert Q_2\rvert \lvert\Sigma\rvert)$ |
| $\Sigma^* - L(A_1)$ | $\lvert Q_1\rvert$ | $\mathcal{O}(\lvert Q_1\rvert)$ |

- Cor. 5.3.1 The class of reg. languages is closed under intersection, union and complementation.

#### 5.2.2 Product automaton

- Determine initial and accepting state.
- Add transitions:
  From source graphs think in pairs.
  If I apply 1:
  $a \to a \wedge c \to c$
  $\implies$ $ac \to ac$
  If I apply 0:
  $a \to b \wedge c \to d$
  $\implies$ $ac \to bd$
- Remove unreachable states




### 5.3 Closure Properties Reg. Lang.

#### 5.3.1 Closure: Concatenation & Iteration

Let $\mathcal{A}_1, \mathcal{A}_2$ be DFAs (or NFAs) for $L_1, L_2$. Then DFAs (or NFAs) can be constructed for $L_1 \circ L_2$ and $L_1^*$

#### 5.3.2 Closure: Homomorphism

A function $h : \Sigma^* \to \Gamma^*$ is a **homomorphism** if for all strings $u, v \in \Sigma^* : h(uv) = h(u)h(v)$. From the definition it follows that $h(\epsilon) = \epsilon$
If L over *Sigma* and $\Gamma$ is an alphabet, then h(L) for each homomorphism h: $\Sigma^* \to \Gamma^*$ is regular.

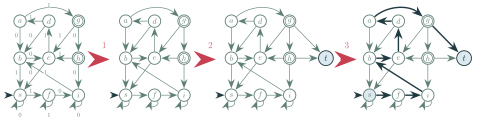
### 5.4 Further Algos for Automata

#### 5.4.1 Emptiness Test

Is there at least 1 word that is accepted?

1. Forget the edge labels
2. Insert a target node $t$ and edges from all accepting nodes to t
3. Test if there is a path from $s$ to $t$
4. If yes: output "$L(\mathcal{A}) \neq \emptyset$"

It might be simpler to backwards, from $t$ to $s$



#### 5.4.2 Word Problem

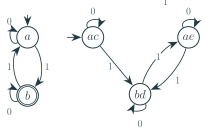
Check if specific word in a given language

| Type | How to solve | Runtime |
|---|---|---|
| DFA | simulate | $\mathcal{O}(\lvert w\rvert + \delta)$ |
| NFA | simulate powerset aut. | $\mathcal{O}(\lvert w\rvert \times \delta)$ |
| RE | $\to \epsilon$-NFA sim. powset aut. | $\mathcal{O}(\lvert w\rvert \times \alpha)$ |

#### 5.4.3 Equivalence Test for DFAs

Two finite automata are equivalent if they decide the same language.

- Show not equivalent: give a word that is accepted by one of the automata but not by the other.
- Show equivalent by:
  - Opt. 1: with minimal Automata for both Caa the states be mapped s.t. their transitions are the same?
    * Construct min. automata $\mathcal{A}_1'$ and $\mathcal{A}_2'$
    * Test if $\mathcal{A}_1'$ and $\mathcal{A}_2'$ are isomorphic
      Construct step-wise bijection $\pi$ from (the states of) $\mathcal{A}_1'$ to $\mathcal{A}_2'$
      Initialization: $\pi$ maps the initial state onto the initial state
      Then: Continue $\pi$ according to the transitions of $\mathcal{A}_1'$ and $\mathcal{A}_2'$
      "Both DFAs are now isomorphic according to $a \to c$ and $b \to d \Rightarrow$ DFAs are equiv.Älso draw the graphs!
  - Opt. 2: with product Automata
    * Construct $A = A_1 \times A_2$ with $F = \{(p_1, p_2) | (p_1 \in F_1, p_2 \notin F_2) \vee (p_1 \ not \in F_1, p_2 \in F_2)\}$
      Product automaton w. accepting states of one of the two
    * $L(A_1) = L(A_2) \iff L(A) = \emptyset$ Language is the same if product automaton accepts no words.



#### 5.4.4 Equivalence Test for NFAs and REs

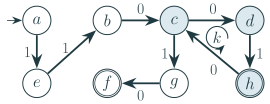
The problem is (likely) even harder than NP.

#### 5.4.5 Finiteness Test

We know that if we can pump a word, we can repeat a cycle over and over again, which would mean that the language is infinite. On the other hand, if there are no loops or cycles, the language must be finite.

The language of a DFA is infinite $\rightleftarrows$ $\exists$ state $q$ with:

- q is reachable from s,
- q lies on a cycle,
- An accepting state is reachable from q



## 6 Context Free Language

Lang. $L$ is context-free $\Leftrightarrow \exists$ CFG $G$ s.t. $L = L(G)$
language is described by a context-free grammar $G$
$\Leftrightarrow$ is recognized by a PDA

CFG $G = (V, \Sigma, S, P)$ consists of
- finite set $V$ of **variables**
- an alphabet $\Sigma$
- a **start symbol** $S \in V$
- finite set $P$ of **rules**: $P \subseteq V \times (V \cup \Sigma)^*$

where $\Sigma \cap V = \emptyset$ has to hold

- Instead of $(A, BC) \in P$ we write $A \to BC$
  - Elements of $V \cup \Sigma$: **symbols**
  - Elements of $\Sigma$: **terminals**
  - Elements of $V$: **nonterminals**
  - Rules in $P$: **Productions**

### 6.1 Derivation

- A derivation is a sequence of derivation steps
- Derivation step: variable $X$ replaced by RHS $\gamma$ of rule $X \to \gamma$ e.g: $bSB \Rightarrow_G baSab$

#### 6.1.1 Derivation Example

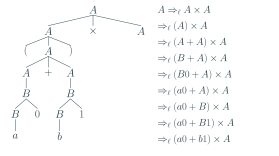
grammar $G$:
$S \to \epsilon$
$S \to a$
$S \to b$
$S \to aSa$
$S \to bSb$

the following is a derivation for the string $bbabbb$ :
$S \Rightarrow_G bSb$
$\Rightarrow_G bbSbb$
$\Rightarrow_G bbaSabb$
$\Rightarrow_G bbababb$

We can also shorten the Grammar as: $S \to \epsilon | a | b | aSa | bSb$

#### 6.1.2 Derivation Trees

$A \to \quad B \mid A + A \mid A \times A \mid (A)$
$B \to \quad Ba \mid Bb \mid B0 \mid B1 \mid a \mid b$




#### 6.1.3 Leftmost Rightmost Derivations

- In each step, the left-/rightmost variable is replaced.
- For each derivation tree, there exist corresponding LM and RM derivations.
- LM & RM derivations may not be unique in tree
- Some derivations are neither leftmost nor rightmost

#### 6.1.4 EX: Give CFG for Language

Let $L_4 = \{w \in \{a, b, c\}^* | w = ucv$ s.t. ($u$ & $v$ contain no $c$) $\wedge$ ($u$ contains exactly 2 $as$) $\wedge$ ($v$ contains even #a) $\}$

- Let CFG $G_4 = (\{S, A, B, C\}, a, b, c, S, P_4)$:
  $P_4 = \{S \to AaAaAcB,$
  $A \to bA \mid \varepsilon,$
  $B \to aC | bB | \varepsilon,$
  $C \to aB \mid bC\}$

### 6.2 Ambiguity

- Ambiguous grammar: CFG $G$ is ambiguous if $\exists$ string $w$ which admits two different derivation trees.
- Unambiguous grammar: $\exists$ single derivation tree.

Unfortunately, there is no general procedure to test if a grammar is unambiguous.

## 7 PDA: Pushdown Automata

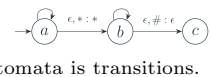
### 7.1 Definition

PDA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, \tau_0, F)$ consists of

- finite set of states $Q$
- input alphabet $\Sigma$
- stack alphabet $\Gamma$
- finite transition relation $\delta \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma) \times (Q \times \Gamma^*)$
- initial state $q_0 \in Q$
- initial stack symbol $\tau_0 \in \Gamma$
- set $F$ of accepting states

Pushdown automaton $\mathcal{A}_{eve}$ for $L_{eve} = \{ww^R | w \in \{0, 1\}^*\}$

$\mathcal{A}_{eve}$ :
$0, * : 0*$ $\quad 0, 0 : \epsilon$
$1, * : 1*$ $\quad 1, 1 : \epsilon$

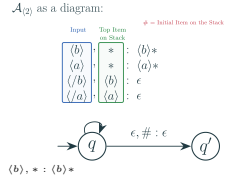



Only visible difference to finite automata is transitions.

### 7.1.1 Transition relation $\delta$

- $(p, \sigma, \tau, q, z)$
  - $p$ current state
  - $\sigma$ input symbol with $\sigma \in \{\Sigma \cup \epsilon\}$
  - $\tau$ symbol on top of stack
  - $q$ new state
  - $z$ replace $\tau$ with $z$

$\mathcal{A}_{(2)}$ as a diagram:




* is not a stack symbol, can be substituted with any other symbol in the stack alphabet.

## 7.2 Configurations and Runs

Configuration describes the current "situation".
Run is sequence $K_1, \ldots, K_n$ of configurations with $K_I \vdash_{\mathcal{A}} K_{i+1} \forall i \in \{1, \ldots, n-1\}$

### 7.2.1 Give Run for word $w$

- Configuration $(q, u, v)$ consists of:
  - a state $q \in Q$
  - the remaining input $u \in \Sigma^*$
  - the stack content $v \in \Gamma^*$

  Start configuration with input $w$ : $(s, w, \tau_0)$
- (Sucessor Configuration) Let $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, \tau_0, F)$ be a PDA. The succesor configuration $\vdash_{\mathcal{A}}$ is defined as follows:
  - $\forall p, q \in Q, \sigma \in \Sigma, \tau \in \Gamma, u \in \Sigma^*, z, v \in \Gamma^*$:
  - $(p, \sigma u, \tau v) \vdash_{\mathcal{A}} (q, u, zv)$ if $(p, \sigma, \tau, q, z) \in \delta$
  - $(p, u, \tau v) \vdash_{\mathcal{A}} (q, u, zv)$ if $(p, \epsilon, \tau, q, z) \in \delta$




A run of $\mathcal{A}$ that accepts the word $w = 10\$0010$ is

$(A, 10\$0010, \#)$
$\vdash (A, 0\$0010, 1\#)$
$\vdash (A, \$0010, 01\#)$
$\vdash (B, 0010, 01\#)$
$\vdash (B, 010, 01\#)$
$\vdash (C, 010, 01\#)$
$\vdash (C, 10, 1\#)$
$\vdash (C, 0, \#)$
$\vdash (D, 0, \#)$
$\vdash (D, \epsilon, \#)$

### 7.2.2 EX: Describe Language for PDA

Describe Language decided by PDA in 7.2.1:
$L(A) = \{x\$y | x, y \in \{0,1\}^* \text{and } x^R \text{ is a substring of y}\}$ with $x^R$ denoting the reverse of $x$

## 7.3 Empty Stack vs. Accepting States

For each PDA $\mathcal{A}$ which accepts with an empty stack, $\exists$ a PDA $\mathcal{B}$ which accepts with accepting states such thath $L(\mathcal{B}) = L(\mathcal{A})$ and vice versa.

### 7.3.1 Empty Stack → Accepting States




"add a final state after stack is empty as accepting state

### 7.3.2 Accepting States → Empty Stack




"after reaching accepting state, add state that clears stack"

## 7.4 CONVERT: CFG → PDA

- Encode entirely in Stack using single-state PDA.
- T8.2 For each context-free grammar G, there exists a pushdown automaton A with $L(A) = L(G)$.

$S \to aSb \mid bY \mid Ya$
$Y \to bY \mid aY \mid \epsilon$
$\equiv$
$\epsilon, S : aSb$
$\epsilon, S : bY$
$\epsilon, S : Ya$
$\epsilon, Y : bY$
$\epsilon, Y : aY$
$\epsilon, Y : \epsilon$
$a, a : \epsilon$
$b, b : \epsilon$




### 7.4.1 EX: CFG → PDA

See image on right.

## 7.5 Deterministic PDAs

DPDAs may only have one transition $(p, \sigma, \tau, q, w)$ in $\delta$ for each combination of $p, \sigma, \tau$. We can have an $\epsilon$-transition going out of $q$ for stack symbol $\tau$, only if for the same state $q$ and stack symbol $\tau$, there is no further transition. $\exists (p, \epsilon, \tau, q, w) \implies$ there may not be a transition $(p, \sigma, \tau, q', w')$ for any $\sigma$. (1) $(p, \epsilon, \tau, q_1, w_1)$ (2) $(p, \epsilon, \tau, q_2, w_2)$

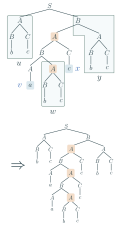
### 7.5.1 Equivalences for Language $L$

For a language L the following are equivalent:

- $L$ is context-free
- $L$ is decided by a PDA w. accepting state
- $L$ is decided by a PDA w. empty stack

## 7.6 Pumping Lemma CFL (Prove Lang. not context-free)

Pump a subtree of a context-free language using expanded pumping lemma. We pump $v$ and $x$




### 7.6.1 Pumping Lemma Cor. for CFL

Let $L$ be a language. Assume that for each $n > 0$
$\exists$ string $z \in L$ with $|z| \geq n$ s.t. $\forall$ decomposition $z = uvwxy$ with
1. $vx \neq \epsilon$ 2. $|vwx| \leq n$
$\exists k \geq 0$ where $uv^k wx^k y \notin \text{Ł} \Rightarrow L$ not context-free.

### 7.6.2 Proof: Pumping Lemma for CFL

1. Let $L_{abc} = \{a^m b^m c^m | m \geq 1\}$
2. Let $n > 0$ be arbitrary. Let $w = a^n b^n c^n \in L_{abc}$.
3. Let $z = uvwxy$ be a decomposition of $w$ with $u, v, w, x, y \in \{a, b\}^*$ s.t.
   (a) $vx \neq \epsilon$ (b) $|vwx| \leq n$
4. $(|vwx| \leq n) \wedge (n\ b's \text{ in } z) \implies$ vx can never consist of more than 2 different symbols
5. $k = 0 : uv^0 wx^0 y = uwy \notin L \implies L$ not context free

### 7.6.3 EX: Prove Context-Free

$L = \{a^n b^m c^k : n, m, k \geq 0 \wedge m \leq min(n, k)\}$

1. Let $n$ be arbitrary. Let $w = a^n b^n c^n \in L$. Let 3 regions: 1 contains all $a$, 2 all $b$, 3 all $c$.
2. Let $z = uvwxy$ be a decomposition with $u, v, w, x, y \in \{a, b, c\}^*$ s.t. (a) $vx \neq \epsilon$ (b) $|vwx| \leq n$
3. Cases:
   - If $v$ or $x$ crosses numbered regions, pump would violate form constraint. Because crossing leads to 2 symbols in v/x (ex $v = ab$). Pumping this would add $abab$ and violate first all a, then all b, then all c (form constraint)
   - (1,1): Pump out once. Reduces $\#a$ while $\#b$ unchanged. Thus $\#a < \#b \Rightarrow m \not\leq min(n, k)$
   - (2,2): Pump in once. $\#b$ increased while $min(\#a, \#b)$ same $\Rightarrow m \not\leq min(n, k)$
   - (3,3): Same as (1,1)
   - (1,2),(2,3): Pump in once. $min(\#a, \#c)$ unchanged, but $\#b$ inreased $\Rightarrow m \not\leq min(n, k)$
   - (1,3): Impossible because $|vwx| \leq n$

3 c ... c
2 b ... b
1 a ... a

## 7.7 CFG Closure Properties

The class of context-free languages is closed under:
• union • concatenation • * operator • + operator

1. If $L$ is context-free then $\{w^R \mid w \in L\}$ is as well
2. If $L \subseteq \Sigma^*$ is context-free and $h : \Sigma^* \to \Gamma^*$ a homomorphism, then $h(L)$ is context-free
3. If $L \subseteq \Sigma^*$ is context-free and $h : \Sigma^* \to \Gamma^*$ a homomorphism, then $h^{-1}(L)$ is context-free

- The class of context-free languages is **not** closed under: intersection, complementation, set difference.
- If $L_1$ is context-free and $L_2$ is regular, then $L_1 \cap L_2$ is context-free.

## 7.8 Closure Prop. of deterministic CFGs

Decided by deterministic PDA w. accepting states.

- closed under complementation.
- **not** closed under union and intersection.
- Not all context-free-langs have a DPDA
  class of deterministic context-free languages is proper subset of class of context-free languages.

## 7.9 Relation Between Classes

1. Each regular language is deterministic context-free
2. $\exists$ deterministic context-free lang. that are not regular
3. $\exists$ context-free lang. that are not deterministic context-free

# 8 Syntax Analysis

Syntax analysis is used to verify whether a given program is syntactically correct

### 8.0.1 Word Problem

- (Word Problem for CFGs)
  Input: A word $w \in \Sigma^*$ and a grammar G.
  Output: Is $w \in L(G)$
- (Syntax Analysis Problem for CFGs)
  Input: A word $w \in \Sigma^*$ and a grammar G.
  Desired output: If $w \in L(G)$: Derivation tree

## 8.1 Backtracking

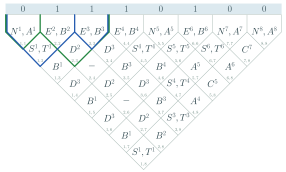
### 8.1.1 Top-down Syntax Analysis

The backtracking algo is "brute-force" iteratively trying all derivations $S \Rightarrow_l^* w[1, m]X\alpha$ of the form $X \to \beta$ to generate the leftmost derivation. It's $\mathcal{O}(p^n)$ exp.

## 8.2 CYK Algorithm

• Takes cubic time $\mathcal{O}(|w|^3 |G|)$ • Uses dynamic programming

$S \to NB \mid EA \mid \epsilon$
$T \to NB \mid EA$
$N \to 0$
$E \to 1$
$A \to 0 \mid NT \mid EC$
$B \to 1 \mid ET \mid ND$
$C \to AA$
$D \to BB$




1. Construct triangular table:
   • Word at top • Indices at bottom of each cell
   • k = split position at top of each rule
2. Next row. In cell (1,2) check for a rule that concatenates its children, (1,1) and (2,2).
3. Cell (1,3): Check extending on left: (1,1) with (2,3) and right: (1,2) with (3,3)
4. Continue until final cell while considering all possible i to j concatenations that would create sequence for current cell.
   Keep track of starting
5. Finally: Root contains S $\implies w \in L(G)$
   Word is only in language if root contains start symbol

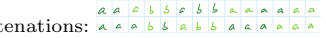
# 9 PCP: Post-Correspondence-Problem

- Let Sequence $(u_1, v_1), \ldots, (u_k, v_k)$ be pairs of non-empty strings.
- $\exists$ index sequence $i_1, \ldots, i_n$ with $n \geq 1$ s.t. $u_{i1}, u_{i2}, \ldots u_{in} = v_{i1}, v_{i2}, \ldots v_{in}$ Exists an concat arrangement that produces 2 equal strings?
- PCP is undecidable.

### 9.0.1 EX: PCP

Let $(u_1, v_1) = (aa, abb), (u_2, v_2) = (abb, b), (u_3, v_3) = (aa, aaa)$

1. Pick Tile where top and bottom start with same symbols: $(u_3, v_3)$
2. Consider the non-overlapping parts to find possible concatenations:

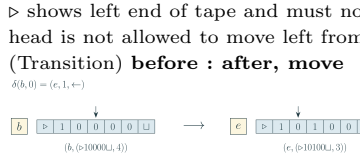


# 10 Turing Machines

## 10.1 Single-Tape Turing Machines

Extension of finite automata with tapes as memory.

- $M = (Q, \Gamma, \delta, s)$ consists of: ○ a finite set of states Q ○ working alphabet $\Gamma$ with $\triangleright \in \Gamma$ ○ transition func. $\delta$: $Q \times \Gamma \to (Q \cup yes, no) \times \Gamma \times \{\leftarrow, \downarrow, \rightarrow\}$ ○ start state $s \in Q$ where it holds that $Q, \Gamma, \{yes, no\} and \{\leftarrow, \downarrow, \rightarrow\}$ are pairwise disjoint.
- $\triangleright$ shows left end of tape and must not be overwritten
- head is not allowed to move left from $\triangleright$
- (Transition) **before : after, move**




- Configuration $M = (q, (w, z))$ with ○ state $q \in Q \cup \{yes, no\}$ ○ string-tape head description $(w, z)$

### 10.1.1 Functions - Turing Computable

use TM for calculations

- Turing Computable: Partial function $f : \Sigma^* \to \Sigma^*$ is turing computable if $f = f_M$
  Function $f$ behaves identical as function $f_M$ that is computed by Machine M
- $f_M(u)$ is only defined if the machine $M$:
  - halts with tape head leftmost in halting state
  - tape content is output string

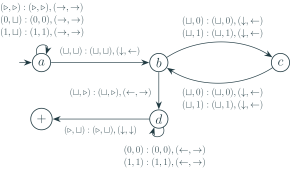
  $f_M$ is the function computed by the Machine.
- Computed function $f_M(u)$ is total $\Leftrightarrow$ TM halts for all inputs (Total func. is a partial func. that is defined $\forall$ inputs)

### 10.1.2 Single-Tape Example

Same idea as Multi-Tape. But just 1 symbol instead of tuple.

## 10.2 Multi-Tape Turing Machines

2-tape palindrome-checking TM.




When computing functions: result must be on tape 1 and tape 1 head all the way to the left.

# 11 Undecidable Problems

## 11.1 Decidability & Computability

- Decidable: Answer to yes/no question
  Set $L \subseteq \Sigma^*$ decidable $\Leftrightarrow \exists$ TM which decides $L$.
  - TM accepts every input $w \in L$
  - TM rejects every input $w \notin L$
  - TM halts on all inputs over $\Sigma^*$, including those that are not in the language
- Computable: open ended question demanding result
  A partial function $f : \Sigma^* \to \Sigma^*$ is called computable if it is Turing computable 10.1.1.
  - $\mathcal{R}$ = the set of all computable partial functions

## 11.2 Proving Closure Properties of decidable Language

### 11.2.1 Prove $D$ is closed under $L_1 \cdot L_2$

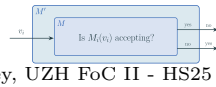
Prove Set $D$ of decidable languages is closed under Concatenation $L_1 \cdot L_2$

1. Let $L_1, L_2$ be decidable Languages
   $\Rightarrow \exists$ TM $M_1, M_2$ that halt $\forall$ inputs & accept the words in $L_1, L_2$ respectively.
2. We construct the following TM S to decide $L_1 \cdot L_2$
   (a) Let $n = |w|$. Let $\forall i \in [0, n] : Letu = w[0 \ldots i-1], Letv = w[i \ldots n-1]$
   (b) Simulate $u$ on $M_1$. If rejects, try next $i$
   (c) If $M_1$ accepts, simulate $v$ on $M_2$. If $M_2$ accepts, then accept $w$.
   (d) If $\nexists$ split that both machines accept $\forall i \in [0, n] \implies$ reject w
3. As $M_1$ & $M_2$ both halt by definition & $S$ runs the msequentially, $S$ must also halt. $\implies$ $S$ decides $L_1 \cdot L_2 \implies L_1 \cdot L_2$ is decidable.

## 11.3 TM-Diag is undecidable

Purpose: Reduce other problems to this first one to show undecidability. We show that no Turing machine can decide TM-Diag, by directly replaying Cantors diagonal argument inside the world of Turing machines.

- Enumerate all strings over $\Sigma^*$ with $\Sigma = \{0, 1\}$, thus an enumeration of all Turing machines. Consider the acceptance and termination behavior of $M_i$ for the input $v_j$ $\forall$ combos of $i$ and $j$.
- Assume $\exists$ TM $M$ that decides TM-Diag. Modify M to M' by inverting accepting behavior:



- There must exist an $M_l$ with $M_l = M'$ because we consider all acceptance & termination behaviors $M_i$.
  $\iff$ $M_l$ accepts $v_l$
  $\iff$ $M'$ accepts $v_l$ (since $M' = M'_l$)
  $\iff$ $M$ rejects $v_l$
  $\implies$ Contradiction $\implies$ TM-Diag is not decidable

## 11.4 Reductions

- Reduction
  total TM-computable funct. $F : \Sigma^* \to \Sigma^*$ is a reduction from L to $L'$ if $\forall x \in \Sigma^* : x \in L \iff f(x) \in L'$
  There exists a mapping between the two languages that preserves yes/no answers.
  We transform $L$ to $L'$. Total = defined $\forall$ inputs
- $L$ Reducible to $L'$ ($L \leq L'$)
  Reduce from $L$ to $L'$ because we want to solve $L$ using a solver for $L'$
  Solver for $L'$, would also solve $L$.
  $L'$ is at least as hard as $L$, as its Solver is at least powerful enough to solve $L$.

### 11.4.1 Implications

Let $L, L'$ be languages with

- $L \leq L' \wedge L'$ decidable $\implies$ $L$ decidable
  $L'$ is harder or eq than $L$. $L'$ can be solved. $\implies$ L can be solved
- $L \leq L' \wedge L$ **un**decidable $\implies$ $L'$ **un**decidable
  $L$ is easier or eq. than $L'$. $L$ can't be solved. $\implies$ $L'$ can't be solved.

### 11.4.2 Application: Prove $L'$ undecidable

1. Let $L$ be a undecidable problem.
2. Construct reduc. funct. from $L$ to $L'$ and prove.
   Force yes/no answers of probs. to correspond. Don't mimic original prob.!
3. Shows $L \leq L'$. Thus $L'$ at least as difficult as $L$.
4. $L$ undecidable by assumption $\Rightarrow$ $L'$ undecidable

## 11.5 TM Reductions

### 11.5.1 Prove $H \leq H_{ALL}$ by showing transition function between TMs

Let $H = \{\langle M, w\rangle | \text{M is a TM that halts on input w}\}$
Let $H_{ALL} = \{\langle M, w\rangle | \text{M is TM that halts } \forall \text{ inputs}\}$

1. Let $M$ be a TM for $H$, Let $M'$ be a TM s.t.:
   - On input x, ignore input x.
   - Simulate $M(w)$
   - If $M(w)$ halts, then halt.
   - Thus: $M(w)$ halts $\Rightarrow$ $M'$ halts $\forall$ inputs $x$
2. We see:
   (a) $f$ is a total function, because its defined $\forall$ inputs
   (b) $f$ is computable as we extend the transition function in finite steps
3. Finally: $\langle M, w\rangle \in H \Leftrightarrow M$ halts on input $w$
   $\iff \forall x, M'(x)$ halts
   $\iff M' \in H_{ALL}$
   $\iff f(\langle M, w\rangle) \in H_{ALL}$
   $\implies H \leq H_{ALL}$

## 11.6 PCP Based Reductions

### 11.6.1 EX: CFG-Non-Disjointness undecidable

$PCP \leq CFG - NON - DISJOINTNESS$

1. Let $L' = \text{CFG-NON-DISJ} = L(G_1) \cap L(G_2) \neq \emptyset$. a.k.a. CFG-Intersection
2. Let $L$ = PCP = $\exists$ index sequence $i_1, ..., i_n$ with $n \geq 1$ s.t. $u_{i1}, u_{i2}, ...u_{in} = v_{i1}, v_{i2}, ...v_{in}$
3. We construct grammars:
   - $G_1 : S_1 \to u_1 S_1 1 | \cdots | u_k S_1 k | u_1 \$ 1 | u_k \$ k$
     $L(G_1)$ contains all strings of form $u_{i1}...u_{in}\$i_n...i_1$ with $n \geq 1$
   - $G_2 : S_2 \to v_1 S_2 1 | \cdots | v_k S_2 k | v_1 \$ 1 | v_k \$ k$
     $L(G_2)$ contains all strings of form $v_{i1}...v_{in}\$i_n...i_1$ with $n \geq 1$
4. Thus PCP $\leq$ CFG-NON-DISJ w. the reduction:
   $x \in L_{PCP} \iff f(x) = (G_1, G_2) \in L_{CFG-NON-DISJ}$
   $(u_1, v_1), ...(u_k, v_k)$ has PCP solution $\iff L(G_1) \cap L(G_2) \neq \emptyset$
5. PCP undecidable $\implies$ CFG-NON-DISJ undecidable

### 11.6.2 EX: CFG-Unambiguity undecidable

$PCP \leq \overline{CFG - UNAMBIGUITY}$

1. Let $L' = \overline{\text{CFG-UNAMB}}$
   $= \exists$ string $w \in L(G)$ with at least 2 derivation trees?
2. Let $L$ = PCP = (11.6.1)
3. Construct **un**ambiguous $G_1$ & $G_2$ like (11.6.1)
4. Let G contain all rules of $G_1$ and $G_2$ plus $S \to S_1 | S_2$
5. Show: $G$ ambiguous $\iff$ $z \in PCP$
   $G$ can only be ambiguous if some string is derivable from both $G_1$ and $G_2$
   - $w \in PCP \implies G$ **not un**ambiguous:
     $w \in$ PCP $\implies w = u_i 1 ... u_{in} = v_{i1} ... v_{in}$ generated by $G_1$ & $G_2$ $\implies$ $G$ has 2 distinct derivation trees. $\implies$ G not unambiguous
   - $G$ **not un**ambiguous $\implies$ $w \in$ PCP:
     If $G$ ambiguous, one tree must use $S_1$ and one $S_2$ as $G_1$ and $G_2$ are individually unambiguous.
     $\implies w \in L(G_1) \cap L(G_2)$
     string must be in both $L(G_1)$ and $L(G_2)$ which is precisely a PCP solution.
6. $\Rightarrow \overline{\text{CFG-UNANMB}}$ undecidable
   $\Rightarrow$ CFG-UNAMB undecidable (Due to 11.7)

### 11.6.3 EX: CFG-Universality is undecidable

$PCP \leq \overline{CFG - ALL}$

1. Let $L' =$ CFG-ALL $= L(G) = \Sigma^*$ generates all possible strings
2. Let $L$ = PCP = (11.6.1)
3. Construct **un**ambiguous $G_1$ & $G_2$ like (11.6.1)
4. Construct $H_1 = \overline{G_1}$ and $H_2 = \overline{G_2}$
   $\implies L(G_1) \cap L(G_2) = \emptyset \iff L(H_1) \cup L(H_2) = \Sigma^*$
5. Let H contain all rules of $H_1$ and $H_2$ plus $S \to S_1 | S_2$
6. $z \in PCP \iff L(G_1) \cap L(G_2) = \emptyset \iff L(H) \neq \Sigma^*$

### 11.6.4 EX: CFG-Equivalence is undecidable

$CFG - ALL \leq CFG - EQUIVALENCE$

Idea: Is some arbitrary Grammar $G_1$ the same as the Grammar that generates everything $G_2$

1. Let $L'$ = CFG-EQUIVALENCE $= (L(G_1) = L(G_2))$
2. Let $L$ = CFG-ALL = 11.6.3
3. Let $G_1 = G$ be an arbitrary Grammar.
   Let $G_2$ w. $L(G_2) = \Sigma^*$ Language that definitely generates all Strings like w. rules $S \to \sigma S \ \forall \sigma \in \Sigma$, $S \to \epsilon$
4. Construct Reduction: $f(G) = (G_1, G_2) = (G, G_{\Sigma^*})$
5. $G \in$ CFG-ALL $\iff L(G) = \Sigma^*$
   $\iff L(G) = L(G_{\Sigma^*})$
   $\iff (G, G_{\Sigma^*}) \in$ CFG-EQUIV
6. CFG-ALL $\leq$ CFG-EQUIV $\wedge$ CFG-ALL undecidable
   $\implies$ CFG-EQUIV undecidable

### 11.6.5 EX: CFG-Containment is undecidable

$CFG - ALL \leq CFG - CONTAINMENT$

Idea: Is the Grammar that generates everything $G_1$ a subset of an arbitrary Grammar $G_2$

1. Let $L'$ = CFG-CONT $= (L(G_1) \subseteq L(G_2))$
2. Let $L$ = CFG-ALL = 11.6.3
3. Let $G_1$ w. $L(G_2) = \Sigma^*$ Language that definitely generates all Strings like w. rules $S \to \sigma S \ \forall \sigma \in \Sigma$, $S \to \epsilon$
   Let $G_2 = G$ be an arbitrary Grammar.
4. Construct Reduction: $f(G) = (G_1, G_2) = (G_{\Sigma^*}, G)$
5. $G \in$ CFG-ALL $\iff L(G) = \Sigma^*$
   $\iff \Sigma^* \subseteq L(G)$
   $\iff L(G_1) \subseteq L(G_2)$
   $\iff (G_1, G_2) \in$ CFG-CONT
6. CFG-ALL $\leq$ CFG-CONT $\wedge$ CFG-ALL undecidable
   $\implies$ CFG-CONT undecidable

### 11.6.6 CFG-REQ-Equivalence is undecidable

$CFG - ALL \leq CFG - REQ - EQUIVALENCE$

Idea: Is some arbitrary Grammar $G$ same as regular expression that generates everything $\alpha$?

1. Let $L'$ = CFG-REQ-EQUIV $= (L(G) = L(\alpha))$
2. Let $L$ = CFG-ALL = 11.6.3
3. Let $G$ be an arbitrary Grammar (the input).
   Let $\alpha = \Sigma^*$ Regular expression that definitely generates all Strings
4. Construct Reduction: $f(G) = (G, \alpha) = (G, \Sigma^*)$
5. $G \in$ CFG-ALL $\iff L(G) = \Sigma^*$
   $\iff L(G) = L(\Sigma^*)$
   $\iff (G, \Sigma^*) \in$ CFG-REQ-EQUIV
6. CFG-ALL $\leq$ CFG-REQ-EQUIV $\wedge$ CFG-ALL undecidable
   $\implies$ CFG-REQ-EQUIV undecidable

## 11.7 Complements of Decidable Languages

L is decidable $\implies$ $\bar{L}$ is decidable
$\bar{L}$ undecidable $\implies$ L is undecidable

## 11.8 Rice's Theorem

Shows any non-trivial semantic property of algo undecidable.

- Let S be a set of computable partial functions with $\emptyset \neq S \neq \mathcal{R}$. $\implies$ TM-Func(S) is undecidable.
  $\mathcal{R}$ = Set of all computable functions

### 11.8.1 Proof

1. Define $S$ the subset of computable functions that have the property we want to check
   $S = \{f \in \mathcal{R} | \forall x, \text{ it holds } f(x) = \bot \vee f(x) \neq 101\}$
2. Show that $S \neq \emptyset$ and $S \neq \mathcal{R}$ holds by giving an example.

### 11.8.2 EX: Rice prove undecidable

Prove: Output $f_M(x)$ of $M$ defined for infinite # inputs $x \in \Sigma^*$ where $\Sigma$ is input/output alphabet $\Sigma$ of $M$?

1. $S = \{f \in R | \{x \in \Sigma^* | f(x) \text{ is defined }\} \text{ is infinite }\}$
   S consists of all computable functions whose domain is infinite.
2. Prove $S \neq \emptyset$: $c(w) = 0 \ \forall w \in \Sigma^*$ The domain of this function is $\Sigma^*$, which is infinite. $\Rightarrow c \in S \Rightarrow S \neq \emptyset$
3. Prove $S \neq R$: $g(w) = \begin{cases} 0 & \text{if } w = \epsilon, \\ \text{undefined} & \text{otherwise.} \end{cases}$ is defined for only 1 input $\Rightarrow$ finite domain $\Rightarrow g \notin S$
4. (2)(3) $\Rightarrow \emptyset \neq S \neq R$ (S is non-trivial set of computable partial functions). Thus by Rice's Theorem it is undecidable.

# 12 Problem Classes

## 12.1 Runtime of Turing Machines

- Let $K_0 \vdash_M K_1 \vdash_M K_2 \cdots \vdash_M K_m$ be TM-computation for input x & $K_m$ halting configuration.
  - Let $t_M(x) = m$ : runtime of $M$ for the input $x$
    TM Runtime = Nr. of configuration changes
  - If $\nexists$ such finite sequence: $t_M(x) = \bot$
    Runtime undefined. Runs indefinitely.
- Input size = len(input string)
- Time Bound Upper Bounds the time for TM for an input x.
  Function $T : \mathcal{N} \to \mathbb{R}$ s.t. $\exists$ an $n_0 \in \mathcal{N}$ s.t. $\forall x \in \Sigma^*$ with $|x| > n_0 : t_M(x) \leq T(|x|)$

## 12.2 P, EXPTIME

- (TIME ($T$), P) For $T : \mathcal{N} \to \mathbb{R}$ let TIME (T) be the set of all languages L for which $\exists$ a k-tape TM M with time bound T such that L=L(M)
- P = $\bigcup_{\text{p polynomial}}$ TIME(p)
  Lang decided by **deterministic** TM in polynomial time
- EXPTIME = $\bigcup_{\text{p polynomial}}$ TIME($2^p$)
  Lang decided by TM in expontential time

### 12.2.1 EX: Prove $Triangle \in P$ (Show polynomial time algorithm)

1. Let triangle: $\exists$ triangle in G.
2. Let $G(V, E)$ be undirected Graph. We describe Algo:
   $\forall$ ordered triple of vertices $(u, v, w) \in V^3$ do:
   (a) Edge Check $\{u, v\}, \{v, w\}, \{u, w\} \in E$
   (b) If yes, accept. $\implies$ $\exists$ triangle in G. Else, reject.

Represent $G$ as adjacency matrix. $\forall$ edge check $O(1)$. $\exists$ max. $n^3$ ordered triples $\Rightarrow O(n^3) \Rightarrow Triangle \in P$

## 12.3 NP

- (NTIME($T$)) be class of all $L \subseteq \Sigma^*$ $\exists$ TM M which decides L non-deterministically s.t. $\forall$ w,y, $\in \Sigma^*$ : $t_M(w, y) \leq T(|w|)$ $t_M(w, y)$ = computation time of $M$ for input $(w, y)$
- NP = $\bigcup_{\text{p polynomial}}$ NTIME(p)
  Lang decided by **non**-determinist. TM in polynom. time
  NP is set of problems where can verify proposed solution in polynomial time, even if can't find solution efficiently

### 12.3.1 EX: Prove $ISO \in NP$ (Show polynomial time verifier)

1. Let $ISO$ be: Undirected graphs $G$ & $H$ isomorphic?
2. Let $M$ be TM receiving normal input $w = (G, H)$ & proof candidate $y$ encoding bijection $\pi : V_G \to V_H$.
3. $\forall (u, v) \in G : \{u, v\} \in E_G \iff \{\pi(u), \pi(v)\} \in E_H$
   $M$ checks $y$ is bijection $\pi$ that preserves adjacency.
4. If condition holds $\forall$ pairs, M accepts. Else rejects.

Verification in $O(n^2)$ as $n^2$ vertex pairs and adjacency test performed in constant time w. adjacency matrix.

## 12.4 P vs. NP vs. EXPTIME

- $P \subseteq NP \subseteq EXPTIME$

## 12.5 Polynomial Reductions

Polynomial Reduction $\leq_p$ is function $f$ from $L$ to $L'$ if:

- $\forall w \in \Sigma^* : w \in L \iff f(w) \in L'$ $f$ is reducction from $L$ to $L'$
- $f$ is computable in polynomial time

## 12.6 NP-Hard

- $L$ is NP-Hard $\iff \forall L' \in NP : L' \leq_p L$ Most difficult Problems. Problem is NP-Hard if it is at least as hard as any other problem in NP
- Any language $L \in NP$ can be reduced in polynomial time to TM-Halt

## 12.7 NP-Complete

The hardest problems in NP. (Because NP also contains P, aren't NP hard).

- $L$ is NP-Complete $\iff$ L is NP-Hard $\wedge L \in NP$

### 12.7.1 Implications

- Let $L, L' \subseteq \Sigma^*$ with $L \leq_p L'$:
  1. $L' \in P \implies L \in P$
  2. $L' \in NP \implies L \in NP$
- Let $L, L' \subseteq \Sigma^*$ and let $L \leq_p L'$. If $L \notin P \implies L' \notin P$ If $L$ is very hard, we can not use a solver for the easier Problem $L'$

### 12.7.2 How would we prove $P = NP$ or $P \neq NP$

- Let L be an NP-complete problem.
  - If $L \in P \implies P = NP$
    Solving single NP-complete Problem in polynomial time, would prove $P = NP$
  - If $L \notin P \implies P \neq NP$
    Showing $\nexists$ polynomial Algo for any NP-complete problem, would prove $P \neq NP$

## 12.8 Cook-Levin Theorem

- SAT is NP-complete.

## 12.9 Prove NP-Hard w. Polynomial Reductions

$L'$ NP-Hard $\wedge L' \leq_p L \implies L$ NP-Hard

### 12.9.1 Prove $SAT \leq_p$ Double-$SAT$ (Show polynomial time reduct.

- Let Double-SAT: $\varphi$ has 2 satisfying assignments?
- Let $f(\varphi) := \varphi \wedge (y \vee \neg y)$
- ($\Rightarrow$) Assume $\varphi$ satisfiable. Choose $y = 0 \wedge y = 1$ respectively. All original clauses of $\varphi$ remain satisfied while $y \vee \neg y$ trivially satisfied. We get 2 distinct satisfying assg. for $f(\varphi)) \Rightarrow f(\varphi) \in$ Double-SAT
- ($\Leftarrow$) Assume $\varphi$ not satisfiable. Then no assignment can satisfy $f(\varphi) \Rightarrow f(\varphi) \notin$ Double-SAT
- $\Rightarrow \varphi \in SAT \Leftrightarrow f(\varphi) \in$ Double-SAT w. $f$ computable in polynomial time $\Rightarrow SAT \leq_p$ Double-SAT
- $SAT \leq_p$ Double-SAT$\wedge$ SAT NP-complete $\Rightarrow$ Double-S NP-Hard