16. Fixed Points

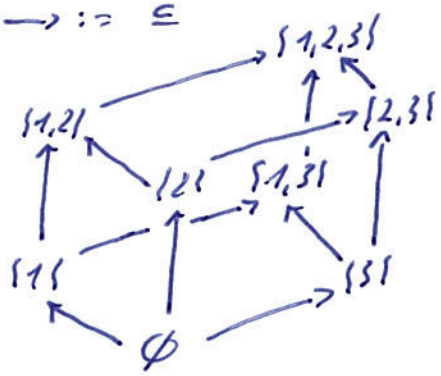
Goal: · Introduce basic notions from lattice theory
· Present the main fixed point theorems.

16.1 Complete Lattices

Goal: · Define partial orders, join and meet, lattices, complete lattices, bottom and top.

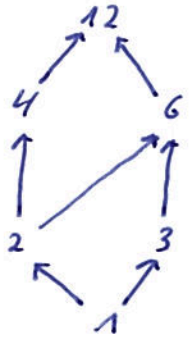
Observation: · $(\mathbb{N}, \leq)$ is totally ordered, every two elements are comparable.
· Some domains are only partially ordered:

Subsets of $\{1,2,3\}$,
$\longrightarrow := \subseteq$



$\{1,2\}$ and $\{2,3\}$ are incomparable

Divisors of 12
$\longrightarrow := |$

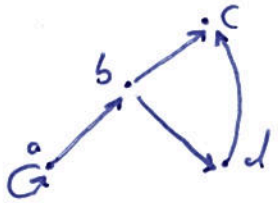


2 and 3 are incomparable.

Definition:
A partial order $(D, \leq)$ consists of a set $D \neq \emptyset$ and a reflexive, transitive, and anti-symmetric relation $\leq \subseteq D \times D$.
Recall that anti-symmetry requires $\forall x, y \in D: x \leq y \wedge y \leq x \Rightarrow x = y$.

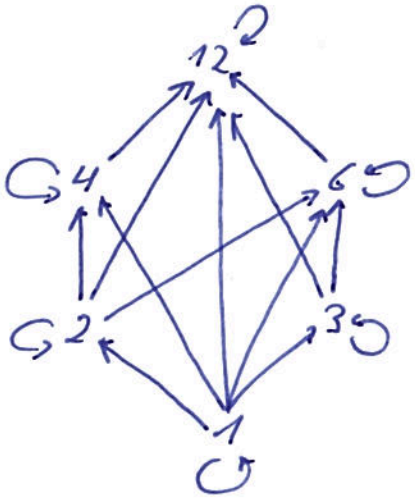
· Binary relations can be understood as directed graphs:

$\{(a,a), (a,b), (b,c), (b,d), (d,c)\}$ =

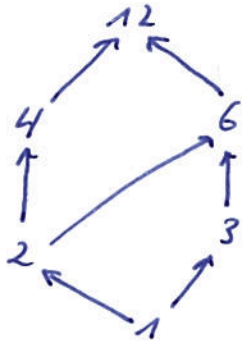


Partial orders yield particular graphs:
↳ Reflexivity = Loops at every node
↳ Anti-Symmetry = No cycles (except for loops)
↳ Transitivity = Transitivity of the edges.

Example (Divisors of 12):



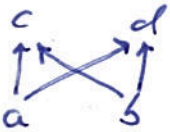
Hasse diagrams of partial orders
do not draw
- loops and
- induced edges:



Definition (Join and meet):
Let $(D, \leq)$ be a partial order.
- An element $o \in D$ is an upper bound of a set $X \subseteq D$,
  if $x \leq o$ for all $x \in X$.
- An element $o \in D$ is a least upper bound, also called join of $X \subseteq D$,
  if
  - o is an upper bound of X and
  - $o \leq o'$ for every upper bound o' of X.

  We write $\bigsqcup X$ for the join of X.
- Similarly, $u = \bigsqcap X$ is the greatest lower bound, also called meet of X.

Example:



a and b:
- have c and d as upper bounds
- but do not have a least upper bound.

Definition (Complete lattice):
- A lattice is a partial order $(D, \leq)$
  where join $a \sqcup b$ (*) and meet $a \sqcap b$ exist for all $a, b \in D$.
- A lattice is complete if every subset $X \subseteq D$ has join and meet.

-> (*) The infix notation stands for $\bigsqcup\{a, b\}$.

Example: (1)  = no lattice (2)  = no complete lattice.

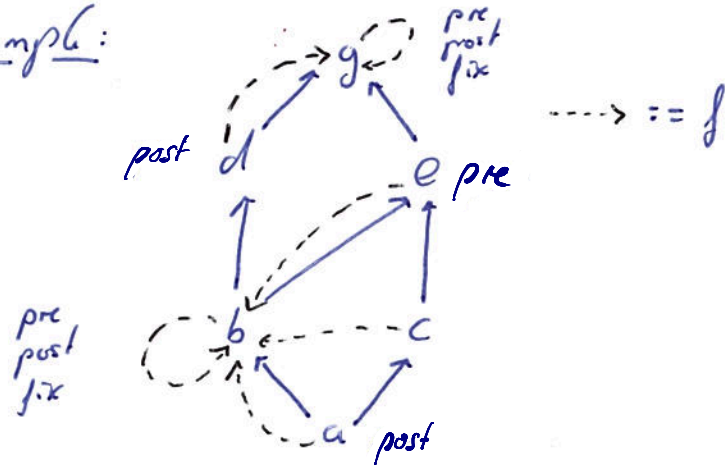
Lemma:
(1) Every complete lattice $(D, \leq)$ has a unique
↳ least element $\bot := \bigsqcup \emptyset = \bigsqcap D$
↳ greatest element $\top := \bigsqcap \emptyset = \bigsqcup D$
(2) Every finite lattice $(D, \leq)$ (with D finite) is complete.

16.2 Monotone Functions and Knaster and Tarski's Theorem:

Goal: Show the existence of fixed points for monotone functions on complete lattices.

Definition (Monotone functions and fixed points):
Let $(D, \leq)$ be a partial order.
- A function $f: D \to D$ is monotone, if $x \leq y$ implies $f(x) \leq f(y)$ for all $x, y \in D$.
- A fixed point of $f: D \to D$ is an element $x \in D$ with $f(x) = x$.
  We use $Fix(f) := \{x \in D \mid f(x) = x\}$ for the set of all fixed points of f.
- A pre-fixed point is an element $x \in D$ with $f(x) \leq x$.
  We use $Prefix(f)$ for the set of all pre-fixed points of f.
- A post-fixed point of f is an element $x \in D$ with $x \leq f(x)$.
  We use $Postfix(f)$ for the set of all post-fixed points of f.

Example:



Theorem (Knaster and Tarski '55):

Let $(D, \leq)$ be a complete lattice and
let $f: D \to D$ be monotone.

(1) Then $\sqcap$ Prefix(f) is a fixed point of f.
Moreover, it is the least fixed point,
and commonly denoted by lfp(f).

(2) Similarly, $\sqcup$ Postfix(f) is the greatest fixed point of f, gfp(f).

Proof:
We show (1), the reasoning for (2) is by duality (turn around the lattice.

- It is sufficient to show that $\sqcap$ Prefix(f) is a fixed point.
Since every fixed point is a pre-fixed point,
$\sqcap$ Prefix(f) has to be smaller than every fixed point of f.

- We use $l := \sqcap$ Prefix(f) to denote the meet.
We first show
$$f(l) \leq l.$$
Since $l \leq l'$ for all $l' \in$ Prefix(f),
and since f is monotone, we obtain
$$f(l) \leq f(l') \leq l' \quad \text{for all } l' \in \text{Prefix}(f).$$
(Prefix)

4.

This means $f(l)$ is a lower bound for $Prefix(f)$.

Since l is the greatest lower bound for $Prefix(f)$,

$$f(l) \leq l \qquad (*)$$

follows.

- We now show

$$l \leq f(l).$$

By $(*)$ and monotonicity, we have

$$f(f(l)) \leq f(l).$$

But this means $f(l) \in Prefix(f)$.

Hence,

$$l \leq f(l). \qquad (**)$$

- Anti-symmetry allows us to conclude $l = f(l)$ from $(*)$ and $(**)$. □

Note:
The above actually is a weak version of Knaster and Tarski's theorem.
The full version states that $(Fix(f), \leq)$, i.e., the set of all fixed points, again forms a complete lattice.

16.3 Chains, Continuous Functions, and Kleene's Theorem

Goal: · Knaster and Tarski's result only tells us that fixed points exist (for monotone functions on complete lattices).
· Kleene's theorem will allow us to compute them by iteration.

Definition:
Let $(D, \leq)$ be a partial order.
· A chain is a totally ordered subset $K \subseteq D$.
· We also give chains as sequences (i.e. lists) consisting of the members.

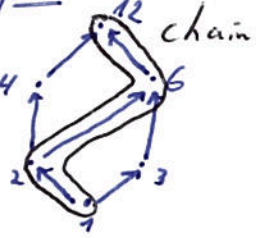
- A sequence $(h_i)_{i\in\mathbb{N}}$ is an ascending chain, if $h_i \leq h_{i+1}$ for all $i\in\mathbb{N}$.
- A sequence $(h_i)_{i\in\mathbb{N}}$ is a descending chain, if $h_i \geq h_{i+1}$ for all $i\in\mathbb{N}$.
- An ascending/descending chain $(h_i)_{i\in\mathbb{N}}$ becomes stationary, if $\exists n\in\mathbb{N}\ \forall i\geq n: h_i = h_n$.

One also says the chain stabilizes.

- $(D,\leq)$ has finite height, if every chain $K\subseteq D$ has finitely many elements.
- $(D,\leq)$ has bounded height, if there is an $n\in\mathbb{N}$ so that every chain $K\subseteq D$ has at most n elements.

Example:

(1)



chain

(2) In $(\mathbb{N},\leq)$, every descending chain becomes stationary.

Definition (chain conditions):

A partial order $(D,\leq)$

↳ satisfies the ascending chain condition (ACC)
(one also says $(D,\leq)$ is an Artinian lattice (Emil Artin, 1898-1962))
if every ascending chain $h_0\leq h_1\leq\dots$ becomes stationary;

↳ satisfies the descending chain condition (DCC)
($(D,\leq)$ is Noetherian (Emmy Noether, 1882-1935))
if every descending chain $h_0\geq h_1\geq\dots$ becomes stationary.

Note: (ACC) and (DCC) are independent of the lattice conditions.

Lemma:
If partial order has finite height iff (ACC) and (DCC) hold.

Definition (Continuity):
Let $(D, \leq)$ be a complete lattice.
A function $f: D \to D$ is

(i) $\sqcup$-continuous, if for every chain $K \subseteq D$:
($\sqcup$-join-continuous, upward-continuous)
$$f(\sqcup K) = \sqcup f(K) := \sqcup \{f(k) \mid k \in K\}.$$

(ii) $\sqcap$-continuous, if for every chain $K \subseteq D$:
($\sqcap$-meet-continuous, downward-continuous)
$$f(\sqcap K) = \sqcap f(K) := \sqcap \{f(k) \mid k \in K\}.$$

Our goal is to show that monotonicity implies $\frac{\sqcup}{\sqcap}$-continuity, provided the corresponding chain condition holds.
The proof relies on the existence of greatest and least elements.

Lemma:
Let $(D, \leq)$ be a partial order.
(i) If (ACC) holds, every non-empty chain $K \subseteq D$ has a greatest element $g \in K$ (with $k \leq g$ for all $k \in K$).
(ii) If (DCC) holds, every non-empty chain $K \subseteq D$ contains a least element $l \in K$ (with $l \leq k$ for all $k \in K$).

Proof:
We show (i), (ii) :3 by duality.
Towards a contradiction, assume
$$\neg(\exists g \in K \; \forall k \in K : k \leq g)$$
which means
$$\forall g \in K \; \exists k \in K : \; g \text{ and } k \text{ are incomparable or } g < k. \qquad (*)$$

Case (≠) cannot hold:
Since both g and h stem from K, and since K is a chain,
they have to be comparable.

We thus have
$$\forall g \in K \; \exists h \in K: \; g < h \quad (g \leq h \text{ and } g \neq h). \qquad (**)$$

This contradicts (ACC), as can be seen from the following ascending chain that does not become stationary:

$h_0 :=$ an arbitrary element from K
$h_{i+1} :=$ the $h \in K$ with $h > h_i$,
which exists by (**). □

Theorem (Monotonicity implies continuity):
Let $(D, \leq)$ be a complete lattice and let $f: D \to D$ be monotone.
(i) If $(D, \leq)$ satisfies (ACC), then f is $\sqcup$-continuous.
(ii) If $(D, \leq)$ satisfies (DCC), then f is $\sqcap$-continuous.

Proof:
We again only show (i). Consider an ascending chain $K \subseteq D$.

- To see that $\sqcup f(K) \leq f(\sqcup K)$,
note that $k \leq \sqcup K$ for all $k \in K$.
Hence, $f(k) \leq f(\sqcup K)$ by monotonicity.
Since this holds for all $k \in K$,
$f(\sqcup K)$ is an upper bound for $f(K) = \{f(k) \mid k \in K\}$.
Hence, the least upper bound is smaller, $\sqcup f(K) \leq f(\sqcup K)$.
- To see that $f(\sqcup K) \leq \sqcup f(K)$,
note that $\sqcup K = g$ with g the greatest element from the above lemma
Now $f(\sqcup K) = f(g) \leq \sqcup f(K)$.
The inequality holds as $g \in K$.
- We conclude by anti-symmetry.

□

For Kleene's fixed point theorem, we consider chains obtained by iterating the function of interest on $\bot$.
This gives us a way to actually compute the least fixed point.

Lemma:
Let $(D, \sqsubseteq)$ be a complete lattice and let $f: D \to D$ be monotone.
The sequence

$$(f^i(\bot))_{i \in \mathbb{N}} \quad \text{with } f^0(\bot) := \bot \text{ and } f^{i+1}(\bot) := f(f^i(\bot))$$

is an ascending chain.

Proof:
We proceed by induction and show that $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$ for all $i \in \mathbb{N}$.

Base case: $f^0(\bot) = \bot \sqsubseteq f(\bot)$, since bottom is the least element of D, $\bot = \sqcap D$.

Induction step: Assume $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$.
Then

$$f^{i+1}(\bot) = f(f^i(\bot)) \underset{\substack{\text{(IH +} \\ \text{Monotonicity)}}}{\sqsubseteq} f(f^{i+1}(\bot)) = f^{i+2}(\bot).$$

□

Theorem (Kleene):
Let $(D, \sqsubseteq)$ be a complete lattice and $f: D \to D$ monotone.
(i) If f is $\sqcup$-continuous, $lfp(f) = \bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \}$.
(ii) If f is $\sqcap$-continuous, $gfp(f) = \bigsqcap \{ f^i(\top) \mid i \in \mathbb{N} \}$.

Proof:
We again show (i).
- The first step is to prove that $\bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \}$ is a fixed point:

$$f(\bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \}) \underset{(\sqcup\text{-continuity})}{=} \bigsqcup \{ f^{i+1}(\bot) \mid i \in \mathbb{N} \} \underset{(\bot = \sqcap D)}{=} \bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \}.$$

- We now show that $\bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \}$ is the least fixed point.
  To this end, we consider $d \in D$ with $f(d) = d$
  and show that $\bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \}$ is smaller.
- By an induction on $i \in \mathbb{N}$, we prove $f^i(\bot) \leq d$ for all $i \in \mathbb{N}$.

  Base case: $f^0(\bot) = \bot \leq d$, since $\bot = \bigsqcap D$.

  Induction step: Assume $f^i(\bot) \leq d$.

  We then get
  $$f^{i+1}(\bot) = f(f^i(\bot)) \leq f(d) = d.$$
  (IH + monotonicity)
- Since $f^i(\bot) \leq d$ for all $i \in \mathbb{N}$, d is an upper bound for $\{ f^i(\bot) \mid i \in \mathbb{N} \}$,
  and thus $\bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \} \leq d$. □

In combination with the above theorem on monotonicity implying continuity
under appropriate chain conditions, we obtain the following.

Corollary:
Let $(D, \leq)$ be a complete lattice with (ACC) and (DCC).
Let $f : D \to D$ be monotone.
Then
- $\mathrm{lfp}(f) = \bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \}$
  $= f^n(\bot)$ with $f^n(\bot) = f^{n+1}(\bot)$.
- $\mathrm{gfp}(f) = \bigsqcap \{ f^i(\top) \mid i \in \mathbb{N} \}$
  $= f^n(\top)$ with $f^n(\top) = f^{n+1}(\top)$.

Recall that (ACC) and (DCC) hold iff the lattice has finite height.

17. Data Flow Analysis

Goal: Analyze the behavior of programs statically,
i.e., at compile-time.

Approach: Compute a fixed point on an abstract domain
of data flow values.

17.1 While-Programs

Definition:

- The syntax of labelled while-programs
is given by the following BNF:

$a ::= k \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2$ // Arithmetic expressions

$b ::= true \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2$ // Boolean expressions

$c ::= [skip]^l \mid [x := a]^l \mid c_1 ; c_2$ // Programs
$\mid$ if $[b]^l$ then c_1 else c_2 fi
$\mid$ while $[b]^l$ do c od,

where $k \in \mathbb{Z}$, $true \in \mathbb{B}$, $x \in Var$, and $l \in Lab$,
with Var and Lab sets of variables and labels, respectively.
Note that each program uses finitely many variables and labels.

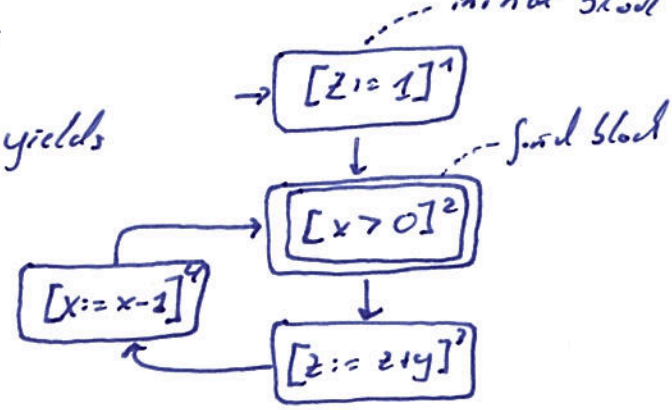
- Labelled commands are called blocks,
and we assume that blocks are uniquely labelled.

- Programs can be represented as control-flow graphs $G = (B, E, F)$
with · B the set of blocks in the program,
· $E \subseteq B$ a set of extremal blocks (initial or final), and
· $F \subseteq B \times B$ the flow relation.

- Typically, a control-flow graph represents the structure of the program of interest:

```
c = [z := 1]^1
    while [x > 0]^2 do
      [z := z * y]^3;
      [x := x - 1]^4
    od
```

yields



- For control-flow graphs, we assume that
  - the initial block has no incoming edges and
  - the final blocks have no outgoing edges.

This form can always be achieved by adding skip-commands.
The example above satisfies the condition on initial blocks but violates the condition on final blocks.

- A control-flow graph $G = (B, E, F)$ will either have the initial block as the only extremal block or the final blocks as the set of extremal blocks, but not both.
The point is that we conduct a data flow analysis either forward, from the initial block, or backward, from the final blocks.

- Indeed, there are data flow analyses that proceed against program order (backwards, e.g. live-variables). In this case, the control-flow graph does not represent the structure of the program but turns around the flow edges and starts from the final blocks.

- In general, we will always precisely define the control-flow graph of interest.

17.2 Monotone Frameworks

Monotone frameworks use a complete lattice with (ACC) as an abstract data domain, and mimic the commands of the program by monotone functions.

Definition:

A *data flow system* or *instance of a data flow analysis* is a triple $S = (G, (D, \leq), i, TF)$ with
- $G = (B, E, F)$ a *control-flow graph*,
- $(D, \leq)$ a complete lattice of *data flow values* satisfying (ACC),
- $i \in D$ an *initial value* for extremal blocks, and
- $TF = \{ f_b : D \to D \text{ monotone} \mid b \in B \}$ a family of *transfer functions*, one for each block, all monotone.

A data flow system $S = (G, (D, \leq), i, TF)$ *induces a system of equations* as follows:
- There is a variable X_b for each block $b \in B$
- For the variable, we have the equation

$$X_b = \begin{cases} i & \text{, if } b \in E \\ \bigsqcup \{ f_{b'}(X_{b'}) \mid (b', b) \in F \}, & \text{otherwise.} \end{cases}$$

A vector $(d_1, \ldots, d_{|B|}) \in D^{|B|}$ is a *solution to (the system of equations induced by) S*, if

$$d_b = \begin{cases} i & \text{, if } b \in E \\ \bigsqcup \{ f_{b'}(d_{b'}) \mid (b', b) \in F \}, & \text{otherwise.} \end{cases}$$

To relate *solutions* to the system of equations induced by S and *fixed points*, define the function:

$$g_S : D^{|B|} \to D^{|B|}$$
$$(d_1, \ldots, d_{|B|}) \mapsto (d_1', \ldots, d_{|B|}')$$

by

$$d_b' := \begin{cases} i & \text{, if } b \in E \\ \bigsqcup \{ f_{b'}(d_{b'}) \mid (b', b) \in F \}, & \text{otherwise.} \end{cases}$$

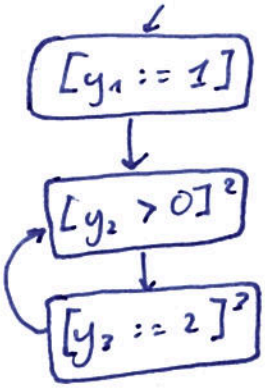
Theorem:
Vector $d \in D^{|B|}$ is a solution to the system of equations induced by S iff d is a fixed point of g_S.

We are typically interested in the least fixed point, as larger fixed points mean a loss of analysis precision.
We know how to compute least fixed points by Kleene iteration.

Example:
- We define a program analysis that computes the set of variables that could have been written when reading a block.
- Consider the program

$$c = [y_1 := 1]^1;\ \underline{\text{while}}\ [y_2 > 0]^2\ \underline{\text{do}}\ [y_3 := 2]^3\ \underline{\text{od}}$$

with $G =$



The data flow system is

$$S = (G, (\mathcal{P}(\{y_1, y_2, y_3\}), \subseteq), \emptyset, \{f_1, f_2, f_3\})$$

with $f_1, f_2, f_3 : \mathcal{P}(\{y_1, y_2, y_3\}) \to \mathcal{P}(\{y_1, y_2, y_3\})$

$$f_1(X) := X \cup \{y_1\} \qquad f_2(X) := X \qquad f_3(X) := X \cup \{y_3\}.$$

The data flow system induces the system of equations

$$X_1 = \emptyset$$

$$X_2 = \underbrace{(X_1 \cup \{y_1\})}_{= f_1(X_1)} \cup \underbrace{(X_3 \cup \{y_3\})}_{= f_3(X_3)}$$

$$X_3 = \underbrace{X_2}_{= f_2(X_2)}$$

4. A solution is $(\emptyset, \{y_1, y_3\}, \{y_1, y_3\})$. It is the least solution.

17.3 Classification of Data Flow Analyses and Examples

The explanation is given in the slides.

The programs are as follows:

Reaching Definitions:

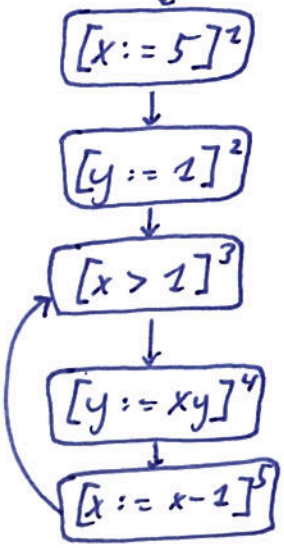
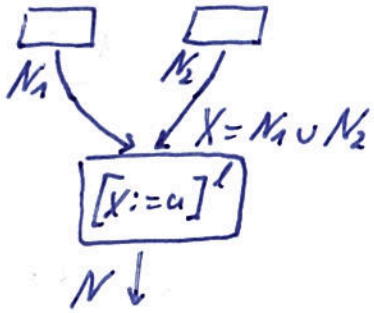


Illustration:



$= (X \setminus kill) \cup gen$.

Available Expressions:

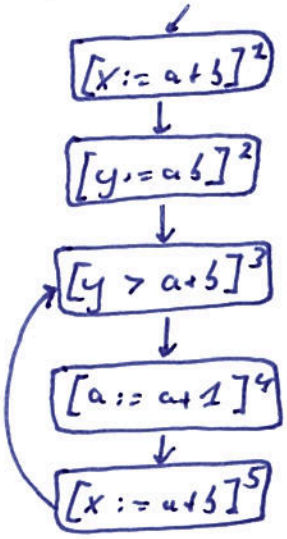
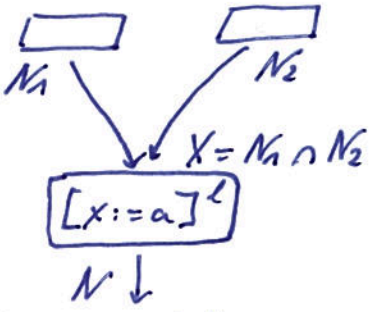


Illustration:



$= (X \setminus kill) \cup gen$

Live Variables:

```
[x := 2]^1;
[y := 4]^2;
[x := 1]^3;
if [y > 0]^4 then
  [z := x]^5
else
  [z := y]^6
fi;
[x := z]^7
```

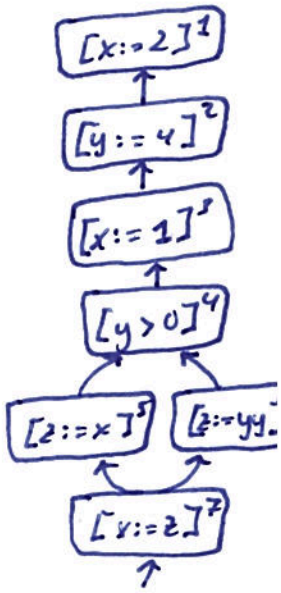
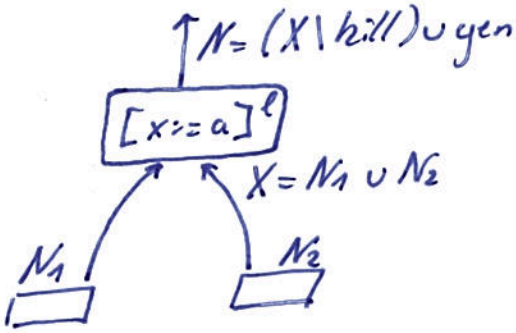


Illustration:



Busy Expressions:

```
if [a > 5]^1 then
  [x := 5-a]^2;
  [y := a-5]^3
else
  [y := 5-a]^4;
  [x := a-5]^5
fi
```

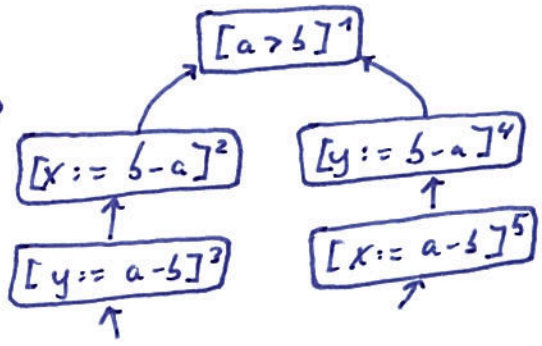
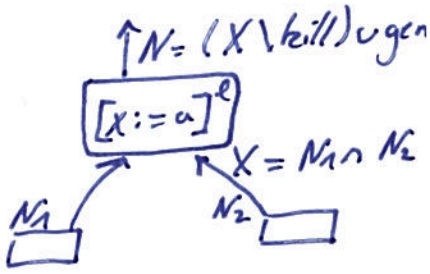
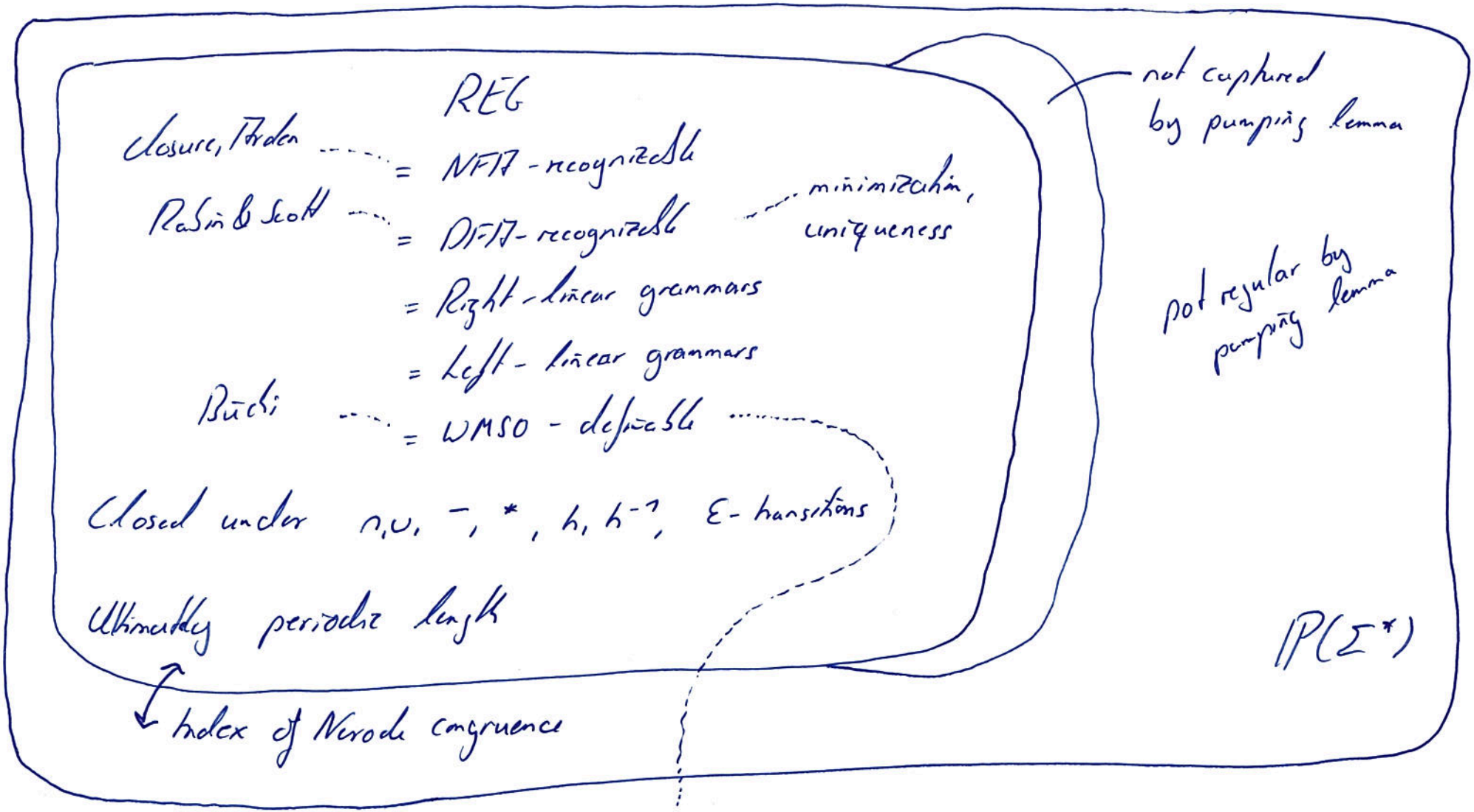


Illustration:





$$\exists x\, \exists y : Q_a(x) \wedge Q_b(y) \wedge x < y$$
$$\wedge\ \forall z : x < z \wedge z < y \rightarrow Q_c(z)$$

yields $\{a,b,c\}^* a\, c^* b\, \{a,b,c\}^*$

17 Finite words

↳ First connection between automata and logic
↳ Basic constructions

1. Regular languages and finite automata

↳ Recapitulation, notation, problems

1.1 Regular languages

↳ Basic notions
↳ Constructions from Sgdp.

Notions:

- Finite alphabet = finite set $\Sigma = \{a, b, \dots\}$
- Finite word = sequence $w = a_0 \dots a_{n-1}$ with $a_i \in \Sigma$
- Length of w, $|w| = n$
- Empty word ε with length 0
- ith symbol $w(i) = a_i$
- Σ^* set of all finite words over Σ, $\Sigma^+ := \Sigma^* \setminus \{\varepsilon\}$ non-empty words
- Concatenation of $w, v \in \Sigma^*$ is $w.v \in \Sigma^*$
- Language $L \subseteq \Sigma^*$, typically infinite
  ↳ With this definition, all set-theoretic operations also apply to languages.

$$L_1 \cup L_2, \quad L_1 \cap L_2, \quad L_1 \setminus L_2, \quad \overline{L_1} := \Sigma^* \setminus L_1$$
(union) (intersection) (difference) (complement)

↳ Concatenation:

$$L_1 . L_2 := \{u.v \mid w \in L_1 \text{ and } v \in L_2\}$$

↳ Kleene star:

$$L^* := \bigcup_{i \in \mathbb{N}} L^i \text{ with } L^0 := \{\varepsilon\}, \; L^{i+1} := L . L^i.$$

(finitely many concatenations with words in L)

$= \{w_1 \ldots w_n \mid n \in \mathbb{N}$ and $w_i \in L$ f.a. $i\}$.
$\mathbb{N} = \{0, 1, 2, \ldots\}$

<u>Definition</u>.

The class of <u>regular languages over alphabet Σ</u> is denoted by REG_Σ. It is the smallest class that satisfies

(1) $\emptyset \in REG_\Sigma$ and $\{a\} \in REG_\Sigma$ f.a. $a \in \Sigma$

(2) $L_1, L_2 \in REG_\Sigma$ implies $L_1 \cup L_2 \in REG_\Sigma$,
$L_1 . L_2 \in REG_\Sigma$,
$L_1^* \in REG_\Sigma$.

Every regular language is obtained by application of <u>finitely many operations</u> in (2) from (1).

<u>Notation</u>:

↳ Brackets: * stronger than . stronger than ∪

↳ $\{a\}$ as a (singleton sets)

<u>Example</u>:

$\varepsilon \cup (a \cup b)^* . b$

<u>Observation</u>:

↳ Every finite set of words forms a regular language

↳ Regular languages are <u>not</u> closed under infinite unions (this gives all (finite word) languages)

↳ By definition, REG closed under ∪, ., *

<u>Show</u>:

REG is also closed under remaining set operations:

$\cap$, $\overline{}$, $\setminus$, Δ

Not clear from definition. Note: $L_1 \setminus L_2 := L_1 \cap \overline{L_2}$.

- For this, need an alternative characterisation of regular languages
  ↳ Also needed for representation and operations on regular languages
  ⇒ languages are infinite objects
  ⇒ Finite representations not always easy to find (one of the goals of TCS).

1.2 Finite automata

Fix the alphabet Σ.

Definition (NFA):

A non-deterministic finite automaton (over Σ) is a tuple $A = (Q, q_0, \to, Q_F)$ with

- states Q, initial state q_0, final states Q_F, and
- transition relation $\to \subseteq Q \times \Sigma \times Q$
  (typically with $q \xrightarrow{a} q'$ instead of $(q, a, q') \in \to$)

Run of A is a sequence

$$q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \to \dots q_{n-1} \xrightarrow{a_{n-1}} q_n.$$

If $w = a_0 \dots a_{n-1}$, say this a run of A on w

Run is accepting, if $q_n \in Q_F$.

With $q_0 \xrightarrow{w} q_n$ for the fact that there are corresponding intermediary states.

Language of A

$$L(A) := \{ w \in \Sigma^* \mid q_0 \xrightarrow{w} q \text{ with } q \in Q_F \}.$$
(there is an accepting run of A on w)

Size of $\mathcal{H}$

$$|\mathcal{H}| := |Q| + |Q_I| + |\to|$$
$$\leq |Q| + |Q| + |Q|^2|\Sigma|$$
$$\in O(|Q|^2) \quad \text{for } \Sigma \text{ fixed.}$$

Number of states is important.

↳ Equations for REG can be applied to NFAs:
↳ Corresponding equations

$$L(A_1 . A_2) = L(A_1) . L(A_2) \quad \text{etc.}$$

important when we consider relationship with logics.

First goal here:

$L \in REG$ iff there is NFA A with $L = L(A)$.

Theorem:

Let A_1, A_2 NFAs.

Then

there is an NFA $A_1 . A_2$ with $L(A_1 . A_2) = L(A_1) . L(A_2)$

there is an NFA $A_1 \cup A_2$ with $L(A_1 \cup A_2) = L(A_1) \cup L(A_2)$.

Proof:

Exercise

Theorem:

Let A an NFA. There is an NFA A^* with $L(A^*) = L(A)^*$.

Construction:

Let $A = (Q, q_0, \rightarrow, Q_F)$

Then

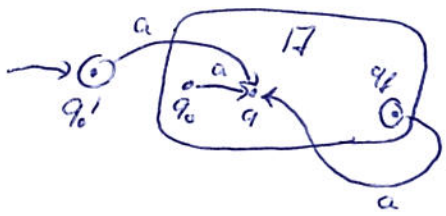
$A' := (Q \cup \{q_0'\}, q_0', \rightarrow \cup \rightarrow', Q_F \cup \{q_0'\})$

where

$q_0' \xrightarrow{a}' q$ if $q_0 \xrightarrow{a} q$

$q_f \xrightarrow{a}' q$ —"—

Illustration:



- initial state not reachable (again)
- loop back from final states.

Theorem:

If $L \in REG$, then there is an NFA A with $L = L(A)$.

Show reverse:

Let A be an NFA, then $L(A)$ is regular.

Approach:

↳ Represent automaton with n states by (recursive) system of n equations.

↳ Solve this system.

↳ Relies on Arden's lemma

Lemma (Arden '60):

Let $U, V \subseteq \Sigma^*$ with $\varepsilon \notin U$.

Consider another language $L \subseteq \Sigma^*$.

Then

$$L = U \cdot L \cup V \quad \text{iff} \quad L = U^* \cdot V.$$

Remark:

„$\Rightarrow$" Such an equation has a unique solution!

Proof:

„$\Rightarrow$" Let $L \subseteq \Sigma^*$ so that $L = U \cdot L \cup V$.

Claim: $L = U^* \cdot V$

„$\supseteq$" Show that $U^* \cdot V = \left(\bigcup_{i \in \mathbb{N}} U^i\right) \cdot V = \bigcup_{i \in \mathbb{N}} U^i \cdot V \subseteq L$

Prove by induction that $U^n \cdot V \subseteq L$ f.a. $n \in \mathbb{N}$.

IA: $U^0 \cdot V = \varepsilon \cdot V = V \subseteq U \cdot L \cup V \subseteq L$.

IS: Assume $U^n \cdot V \subseteq L$.

Then

$$U^{n+1}.V = U.(U^n.V)$$
$$\overset{(IH)}{\subseteq} U.L$$
$$\subseteq U.L \cup V \subseteq L.$$

Since $U^n.V \subseteq L$ f.a. $n \in \mathbb{N}$,
we have $\bigcup_{i \in \mathbb{N}} U^i.V \subseteq L$ □

„⊆" Assume $L \not\subseteq U^*.V$
Then there is a shortest word $w \in L$ with
$w \notin U^*.V$.
Since
$L = U.L \cup V$,
we have
$w \in U.L$ or $w \in V$.
We cannot have $w \in V$, otherwise $w \in U^*.V$ ↯.
So
$w \in U.L$, which means
$w = u.w'$ with $u \neq \varepsilon$ as $\varepsilon \notin U$.
Since
w is the shortest word in L with $w \notin U^*.V$, and
since w' is shorter,
we get
$w' \in U^*.V$.
So
$w = u.w' \in U^*.V$ ↯
Inclusion holds. □

- Apply Arden's lemma as a tool to construct regular representation for an automaton.

Theorem:

- If L is recognised by an NFA, then L is regular.

Proof:

Let $L = L(A)$ with $A = (Q, q_0, \to, Q_F)$.

W.l.o.g., $Q = \{q_0, \dots, q_{n-1}\}$

Define for every state q_i the language

X_i = "words that have an accepting run from q_i in A.

$= L(Q, q_i, \to, Q_F)$.

These languages satisfy

$$X_i = \bigcup_{a \in \Sigma} \bigcup_{q_i \xrightarrow{a} q_j} a . X_j \cup \begin{cases} \varepsilon & \text{if } q_i \in Q_F \\ \emptyset & \text{otherwise} \end{cases}$$

By

↳ mutual insertion of these equations

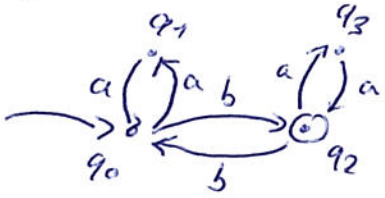
↳ applications of Arden's lemma

construct expressions that prove the languages X_i regular.

By

$L = X_0$, also L is regular. □

Example:



Corresponding system of equations:

$$X_0 = a . X_1 \cup b . X_2 \quad (1)$$

$$X_1 = a . X_0 \quad (2)$$

$$X_2 = b . X_0 \cup a . X_3 \cup \varepsilon \quad (3)$$

$$X_3 = a . X_2 \quad (4)$$

Insert (1) into (3) and (2) into (-1):

$$X_0 = a.a.X_0 \cup b.X_2$$

$$X_2 = b.X_0 \cup a.a.X_2 \cup \varepsilon$$

By Arden's lemma:

$$X_2 = (a.a)^*.(b.X_0 \cup \varepsilon)$$

Insert into X_0:

$$\begin{aligned} X_0 &= a.a\,X_0 \cup b.(aa)^*.(b.X_0 \cup \varepsilon) \\ &= (a.a \cup b(aa)^*.b).X_0 \cup b(aa)^* \\ &= (a.a \cup b.(aa)^*.b)^*.b(aa)^* \end{aligned}$$

1.1.3 Deterministic finite automata.

Definition (DFA):

An NFA $A = (Q, q_0, \to, Q_=)$ is called <u>deterministic</u>, if for all $q \in Q$ and all $a \in \Sigma$

there is precisely one $q' \in Q$ with $q \xrightarrow{a} q'$.

↳ sometimes convenient in applications.

↳ Question:

For every non-deterministic automaton A,
(equivalently, for every regular language)
<u>is there</u> a <u>deterministic</u> automaton A' with $L(A) = L(A')$?

Yes, powerset construction.

Theorem (Rabin & Scott '59)

For every NFA A with n states, there is a DFA A' with $L(A) = L(A')$ and A' has at most 2^n states.

Construction:

Let $A = (Q, q_0, \to, Q_F)$.

Define

$$A' := (\mathcal{P}(Q), \{q_0\}, \to', Q_F')$$

with

$Q_1 \xrightarrow{a}' Q_2$ where $Q_2 = \{q_2 \mid q_1 \xrightarrow{a} q_2 \text{ with } q_1 \in Q_1\}$

and

$$Q_F' := \{Q' \subseteq Q \mid Q' \cap Q_F \neq \emptyset\}$$

// sets that contain a final state

Note that A' deterministic

- For every action a, there is a good state (may be the empty set, $\emptyset \in \mathcal{P}(Q)$).
- Good state uniquely defined.

Consequence of this construction:

Closure of regular languages under complementation.

Note:

For an NFA A, not easy to find $\overline{L(A)}$

$L(A)$ = Words w so that there is run of A on w that is accepting

For complement:

$\overline{L(A)}$ = Words w so that all runs of A on w do not accept

In general, this $\forall$-quantification cannot be translated into $\exists$-quantification.

↳ For DFAs this works:

$\overline{L(A)}$ = Words w so that <u>all</u> runs of A on w do not accept

= Words w so that <u>there is</u> a run of A on w that does not accept.

<u>Theorem:</u>

Consider DFA A. Then there is DFA $\bar{A}$ with $L(\bar{A}) = \overline{L(A)}$.

<u>Construction:</u>

"swap final states"

Let $A = (Q, q_0, \to, Q_F)$.

Then

$$\bar{A} := (Q, q_0, \to, Q \setminus Q_F).$$

To sum up.

Let $L = L(A)$ for an NFA with n states.

Then there are DFAs for L and $\overline{L}$ with at most 2^n states.

↳ Bound is optimal

⇒ There are languages L_n recognized by NFA with $n+1$ states

that are not recognized by DFA with less than 2^n states.

↳ If you only consider states reachable from q_0,

you can often do with less than 2^n.

1.2 Decidability and complexity

Problems:

Given NFA A.

↳ Emptiness: $L(A) = \emptyset$?

↳ Universality: $L(A) = \Sigma^*$?

↳ Word problem: Given also $w \in \Sigma^*$. Is $w \in L(A)$?

Emptiness most important problem

↳ remaining problems can be reduced to it.

Theorem:

Emptiness for NFA A with n states can be solved in time $O(n^2)$.

Proof:

Breadth or depth-search for a final state.

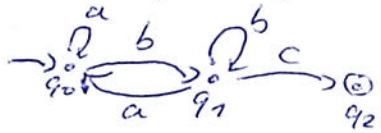
↳ Linear in number of edges, $O(n^2)$

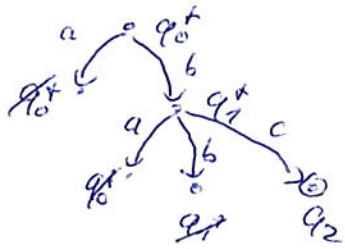
↳ If final state found, return yes.

⇒ accept word along this path

Otherwise no. □

Example for emptiness:





↳ Return "not empty"

Further problems also decidable:

↳ Intersection: $L(A_1) \cap L(A_2) = \emptyset$?

↳ Equivalence: $L(A_1) = L(A_2)$?

↳ Inclusion: $L(A_1) \subseteq L(A_2)$?

Addendum:

Definition:

Let $A = (\Sigma, Q, q_0, \to, Q_F)$ be an NFA.

We generalize the transition relation to words

$$\to \subseteq Q \times \Sigma^* \times Q.$$

The definition is by induction on the length of the word:

- For all $q \in Q$: $q \xrightarrow{\varepsilon} q$.
- For all q_1, q_2, q_3, for all $w \in \Sigma^*$, $a \in \Sigma$:

  if $q_1 \xrightarrow{w} q_2$ and $q_2 \xrightarrow{a} q_3$,

  then $q_1 \xrightarrow{wa} q_3$.

2. Closure Properties of Regular Languages

2.1 Determinisation and Complementation

see previous notes.

2.2 Homomorphisms and ε-Transitions.

Definition:

A homomorphism is a function $h: \Sigma^* \to T^*$ so that for all $x, y \in \Sigma^*$

$$h(xy) = h(x)h(y).$$

Proposition: Consider $h: \Sigma^* \to T^*$ a homomorphism.

(1) $h(\varepsilon) = \varepsilon$.

(2) Every function $f: \Sigma \to T^*$ extends uniquely to a homomorphism $h_f : \Sigma^* \to T^*$.

(3) Every homomorphism is uniquely determined by its values on Σ.
Let $f = h|_\Sigma : \Sigma \to T^*$.
Then
$$h = h_f.$$

Consequence:
We only have to define how a homomorphism acts on the letters from Σ to specify it completely.

Proof:
- To show (1), note that $h(\varepsilon) = h(\varepsilon\varepsilon)$.
We thus have
$$|h(\varepsilon)| = |h(\varepsilon\varepsilon)| = |h(\varepsilon) \cdot h(\varepsilon)| = |h(\varepsilon)| + |h(\varepsilon)|.$$
Thus $|h(\varepsilon)| = 0$, which means $h(\varepsilon) = \varepsilon$.
- Since $h_f(\varepsilon) = \varepsilon$ by (1), we have for $w = a_1 \dots a_n \in \Sigma^*$
$$h_f(w) = h_f(a_1) \dots h_f(a_n) = f(a_1) \dots f(a_n).$$
- We have $f(a) = h(a)$ for all $a \in \Sigma$.
By (2), there is only one homomorphism that we can obtain with this valuation of the letters, which means $h_f = h$.

□

Theorem (Closure under h and h^{-1}):

Consider $h: \Sigma^* \to T^*$ a homomorphism.
Let $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq T^*$ be regular languages.

Then $h(L_1) := \{ h(w) \mid w \in L_1 \} \subseteq T^*$ and (1)

$h^{-1}(L_2) := \{ w \in \Sigma^* \mid h(w) \in L_2 \} \subseteq \Sigma^*$ (2)

are regular.

Proof:

(1) Assume $L_1 = L(A)$ with A an NFA over Σ.
We replace every transition labelled by x
by the automaton for $h(x)$ (potentially inserting new states).
or contracting states).
The result is the NFA $h(A)$ over T.

Claim: $L(h(A)) = h(L(A))$.

(2) Assume $L_2 = L(B)$ with B an NFA over T.
We shortcut every transition sequence

$$p \xrightarrow{h(x)} q \quad \text{to} \quad p \xrightarrow{x} q.$$

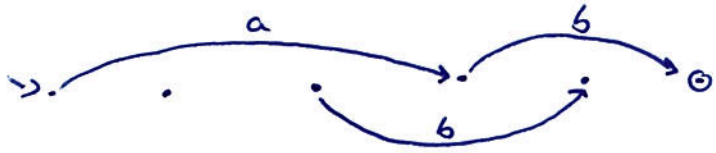
The result is an automaton $h^{-1}(B)$ over Σ.

Claim: $L(h^{-1}(B)) = h^{-1}(L(B))$. □

Example:

Let $\Sigma = \{a, b\}$ and $T = \{x, y, z\}$.
Let $h(a) = xyz$, $h(b) = zz$.

Consider B: 

Then $h^{-1}(B)$: 

Application:

We use homomorphisms to give a clean treatment of ε-transitions.

- Define an NFA with τ-transitions to be the structure

$$A = (\Sigma, \tau, Q, q_0, \to, Q_F),$$

where $\tau \notin \Sigma$ is a special symbol, and define

$$A_\tau := (\Sigma \cup \{\tau\}, Q, q_0, \to, Q_F)$$

to be the associated NFA over $\Sigma \cup \{\tau\}$.

- We say that A accepts $w \in \Sigma^*$, if

$\exists\, v \in (\Sigma \cup \{\tau\})^*$: • A_τ accepts v (using ordinary acceptance for NFAs).

• w is obtained from v by erasing all τ letters.

More formally:

$$w = h(v)$$

where $h : (\Sigma \cup \{\tau\})^* \to \Sigma^*$ is defined by

$h(\tau) := \varepsilon$ and $h(a) := a$ for all $a \in \Sigma$.

Lemma: $L(A) = h(L(A_\tau))$.

Note: One can also remove τ while doing a subset construction.

4. Minimization

Goal: • Use NFAs / DFAs as a (finite) data structure for languages (which are typically infinite objects).
• Smaller DFAs are welcome as the computation time depends on the size of the input

Approach: • Understand the requirement on the automaton imposed by the language (without looking at how the language is represented).
• Turn this requirement into an automaton.

4.1 Theorem of Myhill & Nerode

Goal: Derive an exact condition, sufficient and necessary (characterization), for regularity of $L \subseteq \Sigma^*$.

Definition:
The Nerode-right congruence $\equiv_L \subseteq \Sigma^* \times \Sigma^*$ induced by $L \subseteq \Sigma^*$ is defined by

$u \equiv_L v$: if $\forall w \in \Sigma^*: uw \in L$ iff $vw \in L$.

Idea:
$u \equiv_L v$ iff u and v behave the same for all concatenations wrt. membership in L:
Either $uw \in L$ and $vw \in L$
or $uw \notin L$ and $vw \notin L$.

Definition of a congruence $\equiv_L \subseteq \Sigma^* \times \Sigma^*$:
• Equivalence relation: Reflexiv, transitiv, symmetric.
• Compatibel with the operations (here concatenation): $u \equiv_L v \Rightarrow ux \equiv_L vx$ for all $x \in \Sigma^*$.

Note: If $u \in L$ then $[u]_{\equiv_L} \subseteq L$.

Why? Append ε.

Example:

Consider $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$.

We have

- $a \not\equiv_{L_1} aab$, since $a\underline{ab} \in L_1$ but $aab\underline{ab} \notin L_1$.
- $aab \equiv_{L_1} aaabbb$, both expecting one b.

In general:

$[a^k]_{\equiv_{L_1}} = \{a^k\}$. These are already infinitely many classes.

$[a^{k+1}b]_{\equiv_{L_1}} = \{a^{l+1}b^{l+1-k} \mid l \geq k\}$

// Everything that yields $a^{k+1}b$ if we remove as many as left as we remove bs right.

$[b]_{\equiv_{L_1}}$ = the rest.

Note: · L_1 has infinitely many $\equiv_{L_1}$-equivalence classes.

· The number of equivalence classes is called the index of an equivalence relation.

Theorem (Myhill & Nerode '57/'58):

$L \subseteq \Sigma^*$ is regular iff $\equiv_L$ has finite index.

Remark: L_1 is not regular, we need infinitely many classes.

Proof:

$\Rightarrow$: If $L \subseteq \Sigma^*$ is regular, we have $L = L(A)$

$\Leftarrow$:

for a DFA $A = (\Sigma, Q, q_0, \to, Q_F)$.

Consider the relation $\equiv_A \subseteq \Sigma^* \times \Sigma^*$ induced by A:

$u \equiv_A v$, if $\exists q \in Q: q_0 \xrightarrow{u} q$ and $q_0 \xrightarrow{v} q$.

Claim: • $\equiv_A$ is an equivalence on Σ^*

• $\equiv_A$ refines $\equiv_L$, meaning $\equiv_A \subseteq \equiv_L$.

Indeed, let $u \equiv_A v$.

To establish $u \equiv_L v$, we have to show that

for all $w \in \Sigma^*$: $uw \in L$ iff $vw \in L$.

To this end, consider a word $w \in \Sigma^*$:

$$uw \in L \iff \exists q \in Q \; \exists q_f \in Q_F : q_0 \xrightarrow{u} q \xrightarrow{w} q_f.$$

$$\overset{u \equiv_A v,\ A \text{ DFA}}{\iff} \exists q \in Q \; \exists q_f \in Q_F : q_0 \xrightarrow{v} q \xrightarrow{w} q_f.$$

$$\iff vw \in L.$$

Hence, $u \equiv_L v$.

This shows

$$\text{index } \equiv_L \;\leq\; \text{index } \equiv_A \;\leq\; |Q|.$$

"$\subseteq$" By construction of the <u>equivalence class automaton</u> (of L).

<u>Idea</u>: Store in the state which class has been reached.

Let $u_1, \dots, u_k \in \Sigma^*$ be representatives of the k equivalence classes of $\equiv_L$.

For every $w \in \Sigma^*$ there is precisely one u_i with $[w]_{\equiv_L} = [u_i]_{\equiv_L}$.

Why?

Because

$$\Sigma^* = [u_1]_{\equiv_L} \cup \dots \cup [u_k]_{\equiv_L}$$

with $[u_i]_{\equiv_L} \cap [u_j]_{\equiv_L} = \emptyset$ for all $i \neq j$.

Construct the DFA

$$A_L := (\Sigma, Q_L, q_{0L}, \to_L, Q_{F_L})$$

with
- $Q_L := \{[u_1]_{\equiv_L}, \dots, [u_k]_{\equiv_L}\}$
- $q_{0L} := [\varepsilon]_{\equiv_L}$
- $[u_i]_{\equiv_L} \xrightarrow{a}_L [u_i a]_{\equiv_L}$ for all $1 \leq i \leq k$ and all $a \in \Sigma$.
- $Q_{F_L} := \{[u_j]_{\equiv_L} \mid u_j \in L\}$.

Claim: $[\varepsilon]_{\equiv_L} \xrightarrow{w}_L [w]_{\equiv_L}$.

With this, we get

$$\boxed{L(A_L) = L.}$$

Proof: Let $w \in \Sigma^*$:

$$w \in L(A_L)$$

$$\Leftrightarrow \exists\, [u_j]_{\equiv_L} \in Q_{F_L} : [\varepsilon]_{\equiv_L} \xrightarrow{w} [u_j]_{\equiv_L}$$

$$\overset{(*)}{\Longleftrightarrow} \exists\, u_j \in L : [u_j]_{\equiv_L} = [w]_{\equiv_L}$$

$$\overset{(**)}{\Longleftrightarrow} w \in L.$$

(*) „⇒" By $[\varepsilon]_{\equiv_L} \xrightarrow{w}_L [w]_{\equiv_L}$ and determinism.

„⇐" By $[\varepsilon]_{\equiv_L} \xrightarrow{w}_L [w]_{\equiv_L}$ and $[u_j]_{\equiv_L} = [w]_{\equiv_L}$

(**) Since $u_i \equiv_L w$, we have $u_j \in L$ iff $w \in L$.

Example (Suffix detection):

$L_{01} = \{w01 \mid w \in \{0,1,2\}^*\}$ over $\Sigma = \{0,1,2\}$.

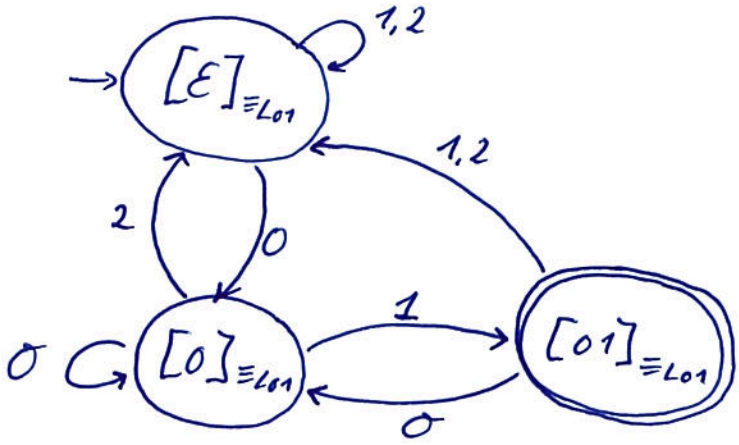
$\equiv_{L_{01}}$ has index 3:

$[0]_{\equiv_{L_{01}}}$ = all words ending in 0

$[01]_{\equiv_{L_{01}}}$ = all words ending in 01

$[\varepsilon]_{\equiv_{L_{01}}}$ = the rest.

$\mathcal{A}_{L_{01}}$:



Note: One can also construct the equivalence class automaton for L_1 but it will not have finitely many states.

4.2 Deterministic Minimal Automata

Goal: Show that
- $\mathcal{A}_L$ is a minimal deterministic automaton for L
- $\mathcal{A}_L$ is unique up to isomorphism.

Corollary (of Myhill & Nerode's Theorem):
Let $L \subseteq \Sigma^*$ be regular and with $\equiv_L$ of index $k \in \mathbb{N}$.
Every DFA accepting L has $\geq k$ states.
$\mathcal{A}_L$ reaches this minimal number.

Proof: $\text{index}_{\equiv_L} \leq |Q|$.
Construction of $\mathcal{A}_L$. □

Note: NFAs can be (a lot) more compact than DFAs
(in the sense that they have < $\text{index}_{\equiv_L}$ states).

- Consider the family of languages $(L_{\text{nondet}}(n))_{n \in \mathbb{N}}$ with

$L_{\text{nondet}}(n)$:= All words over 0,1 with 1 at the nth position from the right.

- For $L_{\text{nondet}}(n)$, an NFA can be found that has $n+1$ states.
- No DFA accepting $L_{\text{nondet}}(n)$ has less than 2^n states.

Why? Check the Nerode classes.

Intuitively:

- The DFA cannot know whether a 1 that has just been read is the appropriate one.
It thus has to remember n-digits, which yields 2^n states.
// This is an informal argument, the proof needs Myhill & Nerode.
- The NFA can guess the right 1:

Claim:
The minimal automaton is unique up to isomorphism.

Definition:

Consider $A_1 = (\Sigma, Q_1, q_{01}, \to_1, Q_{F1})$ and $A_2 = (\Sigma, Q_2, q_{02}, \to_2, Q_{F2})$.

They are called isomorphic, if there is a bijection

$$\beta : Q_1 \to Q_2$$

with $\beta(q_{01}) = q_{02}$

$\beta(Q_{F1}) = Q_{F2}$

$\forall q, q' \in Q_1 \; \forall a \in \Sigma$: $q \xrightarrow{a}_1 q'$ iff $\beta(q) \xrightarrow{a}_2 \beta(q')$.

Function β is called an isomorphism from A_1 to A_2.

(Note that isomorphism defines an equivalence on NFAs.)

Intuitively: NFAs are isomorphic if they coincide up to the names of states.

Theorem (Isomorphism theorem for DFAs):

Let $L \subseteq \Sigma^*$ be regular with index $\equiv_L$ being $k \in \mathbb{N}$.

Then every DFA A that accepts L and has k states is isomorphic to A_L.

Proof: We construct an isomorphism from A_L to A.

Let $A_L = (\Sigma, Q_L, q_{0L}, \to_L, Q_{FL})$ with $Q_L = \{[u_1]_{\equiv_L}, \dots, [u_k]_{\equiv_L}\}$.

Let $A = (\Sigma, Q, q_0, \to, Q_F)$ with $|Q| = k$.

Define function $\beta : Q_L \to Q$ by

$\beta([u_i]_{\equiv_L}) =$ the $q \in Q$ satisfying $q_0 \xrightarrow{u_i} q$.

(This is indeed a function since A is deterministic.)

Claim: β is an isomorphism from A_L to A.

Checking this is homework. □

4.3 Constructing the Deterministic Minimal Automaton

Goal: Given an arbitrary DFA, construct out of it the (up to isomorphism) unique minimal DFA.

Idea: Apply two steps:
- ↳ Eliminate unreachable states
- ↳ Collapse equivalent states.

Definition:
Let $A = (\Sigma, Q, q_0, \to, Q_F)$.
A state $q \in Q$ is reachable, if there is $w \in \Sigma^*$ with $q_0 \xrightarrow{w} q$

Note:
- If $q \in Q$ is reachable, it is reachable by w with $|w| \leq |Q|$.
- See fixed point for emptiness.

From now on:
Assume all unreachable states have been removed.

Definition:
Let $A = (\Sigma, Q, q_0, \to, Q_F)$.
Two states $q_1, q_2 \in Q$ are equivalent, $q_1 \sim q_2$,
if for all $w \in \Sigma^*$:

$$q_1 \xrightarrow{w} q_1' \in Q_F \quad \text{iff} \quad q_2 \xrightarrow{w} q_2' \in Q_F.$$

There is a close correspondence between $\sim$ and $\equiv_L$:

Lemma: Let $L = L(A)$ with $A = (\Sigma, Q, q_0, \to, Q_F)$.
Consider $q_0 \xrightarrow{u} q_1$ and $q_0 \xrightarrow{v} q_2$.
We have

$$u \equiv_L v \quad \text{iff} \quad q_1 \sim q_2.$$

Consequence of the lemma (and our understanding of the minimal automaton):

- If two states are reachable and equivalent, they should be collapsed to obtain the minimal DFA.

Phrased differently, equivalence $\sim$ identifies those states that have to collapse in the minimal automaton.

Why? These states represent the same $\equiv_L$-class.

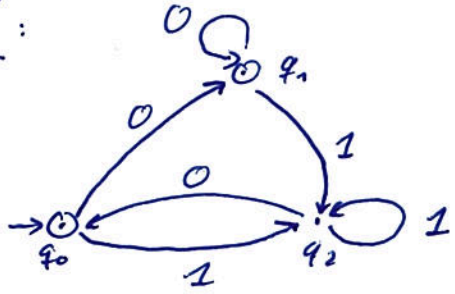
- Hence, we will construct from A an automaton where the states are the $\sim$-equivalence classes of the reachable states.

The transition relation is defined as expected:

$$[q]_\sim \xrightarrow{a} [q']_\sim, \text{ if } q \xrightarrow{a} q'.$$

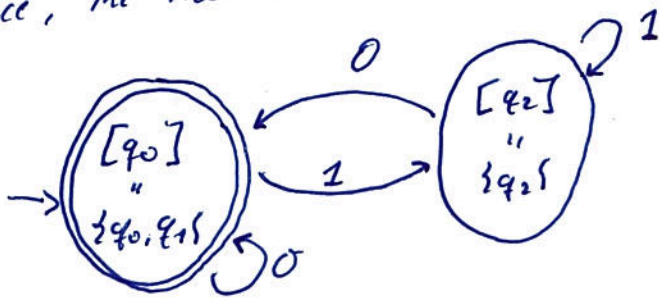
Why is this well-defined?

Example:



We have $q_0 \sim q_1$, but $q_2 \not\sim q_0$.

Hence, the new automaton is



Todo: Check whether two states are $\sim$-equivalent.

Idea: Do a fixed point that determines the inequivalent states.

Works in $O(|Q|^2)$.

Table filling algorithm:

↳ Maintain a table of pairs of states $\{q, q'\}$ with $q \neq q'$.

↳ Mark all states that have been identified as being different.

↳ Collapse all equivalent states.

If we have states $Q = \{q_0, \dots, q_{n-1}\}$,
we can do with a triangular table
of size

$$\frac{1}{2} n \cdot (n-1).$$

Example:

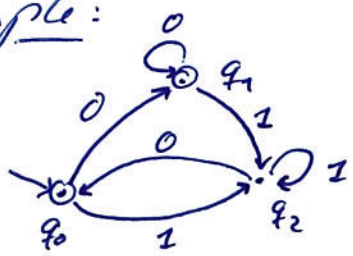
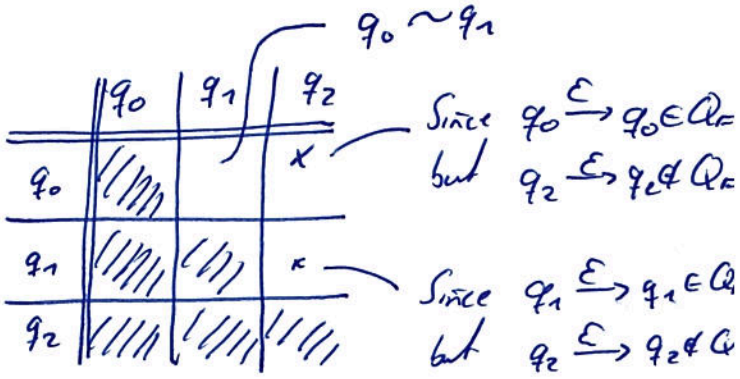


Table:



Pseudocode:

Initially, the table is unmarked;
Mark all $\{q, q'\}$ with $q \in Q_F$ and $q' \notin Q_F$ or vice versa;
while $\exists$ unmarked pair $\{q_1, q_1'\}$ and a letter $a \in \Sigma$
with $q_1 \xrightarrow{a} q_2$ and $q_1' \xrightarrow{a} q_2'$
where $\{q_2, q_2'\}$ is marked do
Mark $\{q_1, q_1'\}$;
od
collapse maximal sets of unmarked states;

Note:

- All this does not work for NFAs.
- Minimizing NFAs is an active research topic.
- One option is bisimulation, see program analysis lecture.

5. Pumping Lemma and Ultimate Periodicity

Recall: We already have a characterization of the regular/non-regular languages via Myhill & Nerode.

Goal: Establish a necessary condition for regularity (not a characterization), that has two benefits
- ↳ it can be generalized to other classes of languages
- ↳ it is easy to apply to disprove regularity.

Problem: To show that

$$L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$$

is not regular, we have to reason about all regular languages.

An argument like "L_1 can count" is not sufficient:

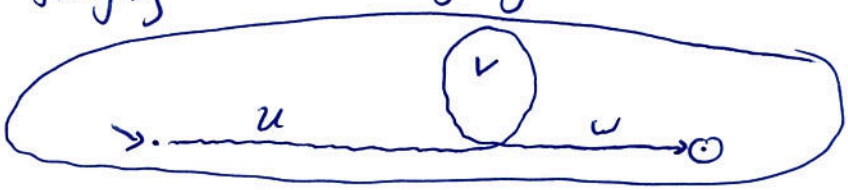
$L = \{w \in \{0,1\}^* \mid \#_0(w) = \#_1(w)\}$ is not regular but

$L' = \{w \in \{0,1\}^* \mid \#_{01}(w) = \#_{10}(w)\}$ is regular.

5.1 Pumping Lemma:

Idea:
- To show that L_1 is not regular, we prove that all regular languages have a structural property that is violated by L_1.
- The structural property states that words can be pumped if they are at least as long as a certain value. Pumping means we can indefinitely repeat an infix while staying in the language.

Intuition:



Theorem (Pumping Lemma):

For every $L \subseteq \Sigma^*$ there is a number $\rho_L \in \mathbb{N}$ so that for all $x \in L$ with $|x| \geq \rho_L$ there is a decomposition

$$x = uvw$$

satisfying the following:

(1) $|v| \geq 1$

(2) $|uv| \leq \rho_L$

(3) $uv^i w \in L$ for all $i \in \mathbb{N}$.

Note: · The number ρ_L depends on the choice of L.

· The representation of L does not go into the lemma.

Proof:

Let $L \subseteq \Sigma^*$ be regular with $L = L(A)$ and $A = (\Sigma, Q, q_0, \to, Q_F)$ an NFA.

We choose

$$\rho_L := |Q|.$$

Consider a word $x \in L$ with $|x| \geq \rho_L$.

Let $x = a_1 \dots a_r$.

Since A accepts x, we have a run

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \dots \xrightarrow{a_{r-1}} q_{r-1} \xrightarrow{a_r} q_r \in Q_F.$$

Since the run contains $r+1 > |x| \geq \rho_L = |Q|$ states, there is a state that repeats:

$$q_j = q_k \quad \text{with } j < k.$$

We consider the first such repetition,

which means

$$q_0, \dots, q_{k-1}$$

are all distinct.

Define $u := a_1 \dots a_j$

$v := a_{j+1} \dots a_k$

$w := a_{k+1} \dots a_r$.

With this choice, we have

(1) $v \neq \varepsilon$ since $j < k$.

(2) $|uv| \leq p_L$ since $q_0, \dots, q_{k-1}$ are all distinct.

(3) Automaton A can repeat the $q_j = q_k$-loop arbitrarily often when accepting. □

Example (Application of the pumping lemma):

Consider

$$L = \{ yy \mid y \in \{a,b\}^* \}.$$

To show that L is not regular, we reason towards a contradiction.

Assume L was regular.

Then there is a number $p_L \in \mathbb{N}$ with (1) to (3) as above.

Consider the word

$$x = a^{p_L} b\, a^{p_L} b \in L.$$

Since $|x| \geq p_L$,

we can decompose the word into

$x = uvw$ with (1) $v \neq \varepsilon$

(2) $|uv| \leq p_L$ and

(3) $\forall i \in \mathbb{N}: uv^i w \in L$.

Since $|uv| \leq p_L$ and $x = a^{p_L} b\, a^{p_L} b$,
u and v only consist of a's.
By Property (3), word

$$uvvw = a^{p_L + |v|} b\, a^{p_L} b \quad \text{with } |v| \geq 1$$

has to be in L. $\lightning$ There is no way to split this word into yy. $\square$

Remark:

- The pumping lemma (can also be) applied in contraposition:

$$\forall p_L \in \mathbb{N}\ \exists x \in L \text{ with } |x| \geq p_L$$
$$\forall \text{ decompositions } x = uvw \text{ with } |v| \geq 1 \text{ and } |uv| \leq p_L$$
$$\exists i \in \mathbb{N}: uv^i w \notin L$$
$$\Rightarrow L \text{ is not regular.}$$

- This is a game between defender ($\forall$) and spoiler ($\exists$):

Spoiler: I believe L is not regular.
Defender: Why not? Here is my number p_L.
Spoiler: I don't think so, here is my word x with $|x| \geq p_L$.
Does your p_L work with this?
Defender: For sure, take my decomposition
$x = uvw$
with $|v| \geq 1$ and $|uv| \leq p_L$.
Spoiler: Hm, but now you lost,
here is $i \in \mathbb{N}$ with $uv^i w \notin L$.

The pumping lemma states that

- if L is regular, defender has a winning strategy.

When applied in contraposition, it states that

- if spoiler has a winning strategy, then L is not regular.

A winning strategy is a strategy
to win no matter how well the opponent plays.

Example:

- Application of the pumping lemma in contraposition:

$$L_1 = \{a^n b^n \mid n \in \mathbb{N}\}.$$

Consider a number $p_{L_1} \in \mathbb{N}$.
We pick the word $x = a^{p_{L_1}} b^{p_{L_1}}$ with $|x| \geq p_{L_1}$.
Consider a decomposition

$$x = uvw \quad \text{with} \quad |v| \geq 1 \text{ and } |uv| \leq p_{L_1}.$$

Since $|uv| \leq p_{L_1}$, we know that u and v only consist of a's.

Take $i = 0$:

$$uv^0w = uw = a^{p_{L_1} - |v|} b^{p_{L_1}} \quad \text{with} \quad |v| \geq 1.$$

Since this word is not in L_1,
we can conclude that L_1 is not regular. □

- The following example uses the closure properties of regular languages:

$$L = \{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}.$$

If L was regular, then

$$L \cap a^*b^* = \{a^n b^n \mid n \in \mathbb{N}\} = L_1$$

was regular. We just showed that L_1 is not regular,
hence L cannot be regular. □

5.2 Ultimate Periodicity

Goal: Formalize the idea that regular languages cannot count (only count modulo).

Definition:
A set $U \subseteq \mathbb{N}$ is called *ultimately-periodic*,
if $\exists n \geq 0 \; \exists p > 0$:

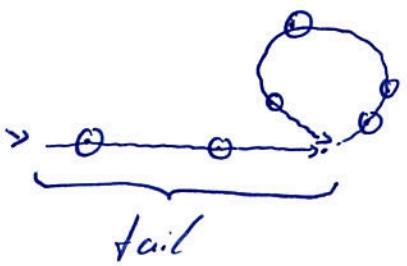
$$\forall m \geq n: \quad m \in U \text{ iff } m+p \in U.$$

Number p is called the *period* of U.

Idea: Except for an initial part,
numbers are in and out of U according to a repeating pattern.

Theorem:
Let $L \subseteq a^*$.
Then L is regular iff $\text{length}(L) := \{|w| \mid w \in L\}$
is ultimately periodic.

Proof:
"$\Rightarrow$" If L is regular, $L = L(A)$ for some DFA A.
Since there is only one letter, the DFA has the shape



Choose $n := |\text{tail}|$ = number of transitions in tail
$p := |\text{loop}|$.

"$\Leftarrow$" Build a DFA of tail length n
and loop of length p. □

Corollary:

Let $L \subseteq \Sigma^*$ be regular.

Then length(L) is an ultimately periodic set.

Proof:

Consider the homomorphism induced by

$$h: \Sigma \to \{a\}$$

(with $h(b) := a$ for all $b \in \Sigma$).

Then $h(w) = a^{|w|}$.

Since h preserves the length, we have

$$\text{length}(L) = \text{length}(h(L)).$$

But $h(L) \subseteq a^*$ is a regular language

(since the regular languages are closed under homomorphisms).

Hence, by the above theorem

length($h(L)$) is ultimately periodic

and so is length(L). □

Note:

Ultimately periodic sets are a first step towards the theory of Parikh images, semi-linear sets, and Presburger arithmetic.

Main results:

↳ Parikh(CFL) = Parikh(REG) = semi-linear sets
= Presburger-definable sets.

↳ Closure under $\cup, \cap, \bar{}, {}^*, h, h^{-1}$.

3. Decidability and Complexity

The membership problem, also called word problem, for regular languages is

WORDREG:

Given: An NFA A over Σ and $w \in \Sigma^*$.

Question: Does $w \in L(A)$ hold?

- In general, a decision problem is a set $P \subseteq S$.

Here,

$$\text{WORDREG} \subseteq \text{NFAs} \times \Sigma^*$$

with $(A, w) \in \text{WORDREG}$ iff $w \in L(A)$.

- The problem (the set) P is decidable, if the characteristic function

$$\chi_P : S \to \{0,1\}$$

is computable.

Recall that for every set $P \subseteq S$, we have (this is the definition)

$$\chi_P(x) = 1 \text{ iff } x \in P.$$

More on computability in the corresponding part of the lecture.

Theorem:

WORDREG can be solved in $O(|w| \cdot |Q|^2)$ time.

Proof:

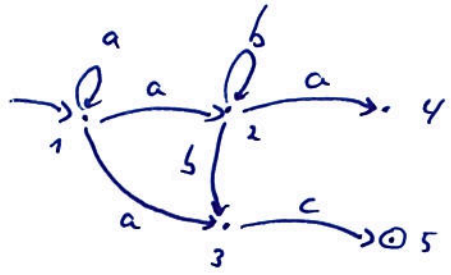
Idea: Do a powerset construction along the given word.

Return "yes", if a final state is in the last set of states.

Complexity: We have $|w|$ steps.
In each step, we have to check for at most $|Q|$ states whether there is a transition to any of the $|Q|$ states. □

Example:

$w = abc$ $\mathcal{A}$:



$\{1\} \xrightarrow{a} \{1,2,3\} \xrightarrow{b} \{2,3\} \xrightarrow{c} \{5\}$.

Return "yes".

There are more problems of importance.

EMPTYREG:

Given: $\mathcal{A}$ an NFA.

Question: $L(\mathcal{A}) = \emptyset$?

UNIVERSALITYREG:

Given: $\mathcal{A}$ an NFA over Σ.

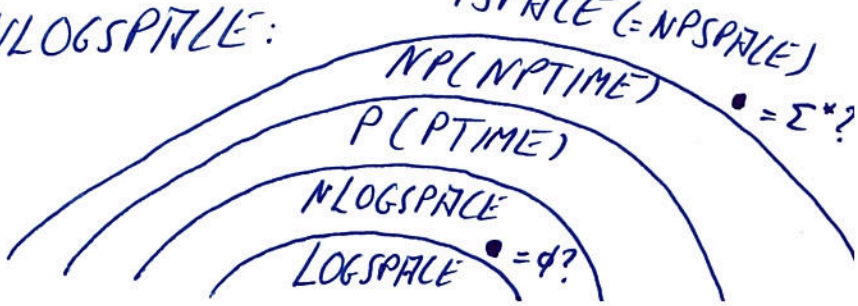
Question: $L(\mathcal{A}) = \Sigma^*$?

Remark:

The problems are interreducible, but not efficiently so.

Universality is substantially harder than emptiness, namely PSPACE vs. NLOGSPACE:

The two classes are known to be different.



Theorem:

EMPTYREG can be solved in $O(1 \to 1)$.

Idea:

- Compute the ascending chain

$$R_0 \subseteq R_1 \subseteq R_2 \subseteq \dots$$

of sets of those states that are reachable in $\leq i$ steps.

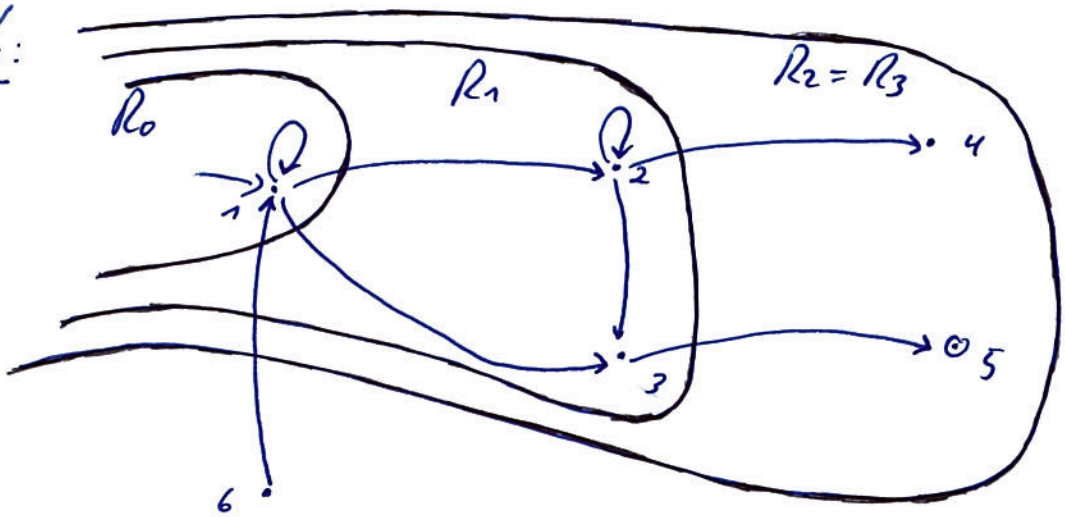
- The computation stops when $R_k = R_{k+1}$.

In this case, the fixed point has been reached:

$$\bigcup_{i \in \mathbb{N}} R_i = R_0 \cup R_1 \cup \dots \cup R_k \cup R_k \cup \dots = R_k.$$

More on fixed points in the semantics part of the lecture.

Example:



Note that the alphabet does not matter.

Proof: Let $\mathcal{A} = (\Sigma, Q, q_0, \to, Q_F)$.

Define $R_0 := \{q_0\}$

$$R_{i+1} := R_i \cup \{q' \in Q \mid \exists q \in R_i \; \exists a \in \Sigma : q \xrightarrow{a} q'\}.$$

Consider $R_k = R_{k+1}$. If $R_k \cap Q_F \neq \emptyset$, return $L(\mathcal{A}) \neq \emptyset$,

otherwise return $L(\mathcal{A}) = \emptyset$.

Complexity:

↳ The fixed point is reached after at most $|Q|$ steps.
This gives complexity $O(|Q| \cdot |\to|)$.

↳ It is sufficient to consider every transition $q \xrightarrow{a} q'$ at most once.
This gives $O(|\to|)$ as we claimed. □

Further decision problems on NFAs:

INTERSECTION REG: Given NFAs A, B, is $L(A) \cap L(B) \neq \emptyset$?

EQUIVALENCE REG: Given NFAs A, B, is $L(A) = L(B)$?

INCLUSION REG: Given NFAs A, B, is $L(A) \subseteq L(B)$?

7. Normal Forms for Context-Free Grammars

Goal: Transform a given context-free grammar G
into another grammar G'
that
- has the same language but
- a simpler structure.

Motivation:
- Algorithms can then rely on the simpler structure
which will make them faster
(unless the blow-up in the conversion is prohibitive).
- It is easier for us to understand properties of context-free languages
if the grammars have a simpler structure.

7.1 Getting Rid of ε-Productions, Unit Productions, and Useless Non-Terminals

Lemma:
For every CFG G, we can construct a CFG G'
without ε-productions $(A \to \varepsilon)$ and
without unit productions $(A \to B)$
satisfying $L(G') = L(G) \setminus \{\varepsilon\}$.

Note: The result is a statement of existence of G'.
But it says more: We also know how to obtain G'.
If we not only know an algorithm exists
but we can actually give the method,
we say the algorithm is *effective*.

Proof: Let $G = (N, \Sigma, P, S)$.
We define $\hat{G} := (N, \Sigma, \hat{P}, S)$.

Here, $\hat{P}$ is the smallest set containing P and closed under the following rules:

(a) If $A \to \alpha B \beta$ and $B \to \varepsilon$ are in $\hat{P}$, then $A \to \alpha\beta$ is in $\hat{P}$.

(b) If $A \to B$ and $B \to \gamma$ are in $\hat{P}$, then $A \to \gamma$ is in $\hat{P}$.

- Note that only finitely many productions are added. The reason is that the length of the right-hand sides is bounded by the length of the longest right-hand side in P.
- Moreover, note that $L(\hat{G}) = L(G)$.

"$\supseteq$" Holds because $P \subseteq \hat{P}$.

"$\subseteq$" Holds because each new production can be simulated by two old productions that caused it to be added (formally, this needs an induction).

Claim:
For every $w \in \Sigma^+$, every derivation $S \Rightarrow_{\hat{G}}^* w$ of minimal length neither uses ε-productions nor unit productions.

Proof:
Consider $w \neq \varepsilon$ and a shortest derivation $S \Rightarrow_{\hat{G}}^* w$.
- Towards a contradiction, assume that $B \to \varepsilon$ is used at some point, say

$$S \Rightarrow^* \alpha_1 B \alpha_2 \Rightarrow \alpha_1 \alpha_2 \Rightarrow^* w.$$

- One of α_1 and α_2 is $\neq \varepsilon$, otherwise $w = \varepsilon$.

Thus, the B is not S but has been introduced at some point, say with a production $A \to \beta_1 B \beta_2$:

$$S \Rightarrow^m \gamma_1 A \gamma_2 \Rightarrow \gamma_1 \beta_1 B \beta_2 \gamma_2 \Rightarrow^n \alpha_1 B \alpha_2 \Rightarrow \alpha_1 \alpha_2 \Rightarrow^k w.$$

- But by Rule (a), also $A \to \beta_1 \beta_2$ is in $\hat{P}$.

If we apply that production instead, we obtain

$$S \Rightarrow^m \gamma_1 A \gamma_2 \Rightarrow \gamma_1 \beta_1 \beta_2 \gamma_2 \Rightarrow^n \alpha_1 \alpha_2 \Rightarrow^k w.$$

This contradicts the assumption that the given derivation was of minimal length. □

- For unit productions, the reasoning is similar.
- Since we neither need ε-productions nor unit productions to generate $w \neq \varepsilon$, we remove them from $\hat{G}$ and obtain G' with $L(G') = L(G) \setminus \{\varepsilon\}$. □

Definition:
Let $G = (N, \Sigma, P, S)$. A non-terminal $X \in N$ is useful, if

$$S \Rightarrow^* \alpha X \beta \Rightarrow^* w \in \Sigma^* \text{ for some } \alpha, \beta \in (N \cup \Sigma)^*.$$

Otherwise, X is called useless.

Theorem:
For every CFG G, we can construct a CFG G' without ε-productions, without unit productions, and without useless non-terminals so that $L(G') = L(G) \setminus \{\varepsilon\}$.

Proof:
Consider the grammar G'' without ε-productions and without unit productions obtained with the previous lemma.

- With a fixed point computed backwards, we determine the non-terminals that can derive a terminal word.
- With a fixed point computed forwards, we determine the non-terminals that are reachable from S.
- Intersecting the sets and removing the remaining non-terminals together with the productions that use them yields the desired grammar. □

7.2 Chomsky Normal Form

Definition:
A CFG is in Chomsky normal form (CNF) if all productions take the form
$A \to BC$ or $A \to a$.

Theorem:
For every CFG G, we can construct a CFG G' in CNF and with $L(G') = L(G) \setminus \{\varepsilon\}$.

We base our construction on the previous theorem.
As a result, all non-terminals will be useful,
there are no ε-productions and no unit productions.

Proof:
- We first apply the previous theorem to turn G into G'' without useless non-terminals, ε-productions, and unit productions.

- We introduce a new non-terminal A_a for every $a \in \Sigma$ that occurs in the right-hand side of a production. We replace every occurrence of a in a right-hand side by A_a. We add the production $A_a \to a$. (Note that this step does not introduce useless non-terminals.)

- Now all productions are of the form

  $A \to a$ or $A \to B_1 \dots B_k$ with $k \geq 2$.

- For any production $A \to B_1 \dots B_k$ with $k \geq 3$, we introduce a new non-terminal C and replace the production by

  $A \to B_1 C$ and $C \to B_2 \dots B_k$.

We repeat this until all productions have length 2. □

Example:

We derive a grammar in CNF for

$\{a^n b^n \mid n \in \mathbb{N}\} \setminus \{\varepsilon\} = \{a^n b^n \mid n \geq 1\}$.

Starting from

$S \to aSb \mid \varepsilon$ for $\{a^n b^n \mid n \in \mathbb{N}\}$,

we remove ε-productions as described in the first lemma.

This yields

$S \to aSb \mid ab$ generating $\{a^n b^n \mid n \geq 1\}$.

We add non-terminals A, B and replace the productions by

$S \to ASB \mid AB$ $A \to a$ $B \to b$.

Finally, we add a non-terminal C and replace $S \to ASB$ by

$S \to AC$ $C \to SB$.

7.3 Greibach Normal Form

Definition:

A CFG G is in <u>Greibach normal form (GNF)</u>.
if every production is of the form

$$A \to a B_1 \dots B_k, \quad k \geq 0.$$

Theorem:

For every CFG G, we can construct a CFG G' in GNF
with $L(G') = L(G) \setminus \{\varepsilon\}$.

We will base our construction on the Chomsky normal form.
Therefore, we will obtain $k \leq 2$, no useless non-terminals,
no ε-productions, no unit productions.

Proof:

- Let $G = (N, \Sigma, P, S)$ be in CNF, $A \in N$, $a \in \Sigma$.
  We define

$$R_{A,a} := \{ \beta \in N^* \mid A \Rightarrow_{SL}^* a\beta \}.$$

The <u>strong left derivation relation</u> $\Rightarrow_{SL}^* \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$ is defined by

$\alpha_1 \Rightarrow_L^* \alpha_2$ if α_2 can be obtained from α_1
by a sequence of derivation steps
in which productions are only applied to the <u>leftmost symbol</u>
in which productions are only applied to the leftmost symbol

- The set $R_{A,a}$ is a <u>regular language</u> over the alphabet N.
  Indeed, take the left-linear grammar

$$(\{B' \mid B \in N\}, N, \{B' \to C'D \mid B \to CD \in P\}, A')$$
$$\cup \{B' \to \varepsilon \mid B' \to a\}$$

to generate it.

It contains a non-terminal B' for every non-terminal B in G.
The former non-terminals N now form the terminal symbols.
The productions always replace the leftmost non-terminal,
thus simulating a sequence of left derivations.
The process stops when the terminal symbol of interest a
has been found.

// Note that it is a substantial conceptual twist
to think about words over non-terminals
as a regular language.
You should really think this through.

- It is not difficult to show that for every left-linear grammar $G_{\Pi,a}$
there is a strongly right-linear grammar $G_{\Pi,a}$
with the same language.
Strong here means the productions are of the form

$$X \to B.Y \quad \text{or} \quad X \to \varepsilon.$$

where X, Y are non-terminals and B is a terminal symbol.
Let $T_{\Pi,a}$ be the start symbol of $G_{\Pi,a}$.

- We can assume that the non-terminals in all $G_{\Pi,a}$ and in G
are pairwise disjoint.
We form the grammar G_1 by adding to G
all non-terminals and productions of all $G_{\Pi,a}$.
The start symbol of G_1 is S.
The productions of G_1 are of the form

$$X \to \varepsilon \qquad X \to b \qquad X \to B.Y, \qquad X, B, Y \text{ non-terminals.}$$

Moreover, $L(G_1) = L(G)$ since none of the new non-terminals is reachable.

- We define the grammar G_2 from G_1 by replacing $X \to B.Y$ with the rules $X \to a.T_{B,a}.Y$ for all $a \in \Sigma$.

- What is going on here?
The idea is to guess the leading terminal symbol a that will be generated in a derivation sequence from B.
The possible sequences of non-terminals following B is a (that are generated from B) are captured by $G_{B,a}$ with start symbol $T_{B,a}$.
It may seem restrictive to only consider sequences of non-terminals following a.
But we do the same replacement also for the rules in $G_{B,a}$.
Therefore, we capture all sentential forms derivable from B.
The productions in G_2 are of the form

$X \to \varepsilon$, $X \to b$, and $X \to a.T_{B,a}.Y$.

- We get rid of ε-productions using the construction in the first lemma.
This construction does not introduce unit productions.
To see this, note that every non-ε-production has a terminal on the right-hand side.

- The resulting grammar G_3
  - is in GNF and
  - satisfies $L(G_3) = L(G) \setminus \{\varepsilon\}$.

- It has no ε-productions and no unit productions.
- It has no useless non-terminals if we make sure to only consider $R_{A,a} \neq \emptyset$ (and modify $G_{A,a}$ accordingly). □

Example:

Consider the grammar in CNF for balanced parantheses:

G: $S \to AB \mid AC \mid SS$ $\quad A \to [$ $\quad B \to]$

$C \to SB$

We compute the regular languages

(1) $R_{S,[} = (B \cup C).S^*$

(2) $R_{C,[} = (B+C).S^*.B$

(3) $R_{A,[} = \{\varepsilon\} = R_{B,]}$.

All other languages are $\emptyset$.

Corresponding strongly right-linear grammars are

(1) $T_{S,[} \to B.X \mid C.X$ $\quad X \to SX \mid \varepsilon$

(2) $T_{C,[} \to B.Y \mid CY$ $\quad Y \to S.Y \mid B.Z$ $\quad Z \to \varepsilon$

(3) $T_{A,[} \to \varepsilon$

$T_{B,]} \to \varepsilon$.

Combining these grammars with G, we obtain:

$S \to [. T_{A,[} . B \mid [. T_{A,[} . C \mid [. T_{S,[} . S$

$T_{S,[} \to] . T_{B,]} . X \mid [. T_{C,[} . X$

$T_{C,[} \to] . T_{B,]} . Y \mid [. T_{C,[} . Y$

$T_{A,[} \to \varepsilon$

$T_{B,]} \to \varepsilon$

$C \to [. T_{S,[} . B$

$X \to [. T_{S,[} . X \mid \varepsilon$

$Y \to [. T_{S,[} . Y \mid] . T_{B,]} . Z$

$A \to [$

$B \to]$

$Z \to \varepsilon .$

Removing ε-productions yields:

$S \to [. B \mid [. C \mid [. T_{S,[} . S$

$T_{S,[} \to] \mid] . X \mid [. T_{C,[} \mid [. T_{C,[} . X$

$T_{C,[} \to] . Y \mid [. T_{C,[} . Y$

$C \to [. T_{S,[} . B$

$X \to [. T_{S,[} . X \mid [. T_{S,[}$

$Y \to [. T_{S,[} . Y \mid]$

$A \to [$

$B \to] .$

13. Decision Algorithms for Context-free Languages

Goal: • Study algorithmic problems for context-free languages
• Focus on positive results, problems that are decidable
(that can be solved algorithmically).

13.1 Emptiness and Inclusion in a Regular Language

Goal: Develop an algorithm that decides whether a given CFL is empty.

Formally, we study the following problem:

EMPTYCFL:
Given: A CFG $G = (N, \Sigma, P, S)$
Question: Is $L(G) = \emptyset$?

Theorem: EMPTYCFL is decidable in $O(|P|^2)$.

Proof:
• We compute an ascending chain of sets of non-terminals
$N_0 \subseteq N_1 \subseteq \dots$
until we reach a fixed point $N_k = N_{k+1} = \bigcup_{i \in \mathbb{N}} N_i$.
• The idea is that N_i contains the non-terminals
from which we can derive a terminal word
with a parse tree of height $i+1$.
• Formally:
$N_0 := \{ A \in N \mid A \to w \in P \text{ with } w \in \Sigma^* \}$
$N_{i+1} := \{ A \in N \mid A \to \alpha \in (N_i \cup \Sigma)^* \}$.
• Like for finite automata, we only need to apply
each production (at most) once.
• We still have to go through the remaining productions to find
the applicable ones. □

- In program verification, we study the problem whether each word in a context-free language (modeling a recursive program) is correct wrt. a (safety) specification.
- The specification is often given as a regular language, so the (safety) verification problem amounts to:

INCLUSIONCFLREG:

Given: A CFG G in CNF and an NFA A.

Question: Does $L(G) \subseteq L(A)$ hold?

Theorem: INCLUSIONCFLREG is decidable in $O(|G|^6 \cdot 2^{6|A|})$.

Proof:
- The following equivalence is of great importance in verification:

$$L(G) \subseteq L(A) \quad \text{iff} \quad L(G) \cap \overline{L(A)} = \emptyset$$

- We can thus determinize A and invert the final states to obtain B with

$$L(B) = \overline{L(A)}.$$

This takes at most exponential time (for the powerset construction).

- The context-free languages are closed under regular intersection. We do the triple construction and obtain H with

$$L(H) = L(G) \cap L(B).$$

The triple construction introduces

$|N| \cdot (Q^{|Q|})^2$ non-terminals and

$|\Sigma| \cdot |N| \cdot (Q^{|Q|})^2$ productions $(q_1, A, q_2) \to a$ (for $A \to a$) and

$|N|^3 (Q^{|Q|})^3$ productions $(q_1, A, q_2) \to (q_1, B, q)(q, C, q_2)$ (for $A \to BC$).

Checking emptiness works in quadratic time.

All together, the construction of M works in time

$$O(|G|^3 2^{3|R|}).$$

This upper bound is in particular due to the number of productions. So we do not save by considering them separately.
Applying the emptiness check yields

$$O((|G|^2 \cdot 2^{3|R|})^2) = O(|G|^6 \cdot 2^{6|R|}).$$ □

Interestingly, the reverse inclusion

$$L(A) \subseteq L(G)$$

will turn out to be undecidable, even for a fixed language A.

UNIVERSALITYCFL:

Given: A CFG G over Σ.
Question: Is $L(G) = \Sigma^*$?

Theorem: UNIVERSALITYCFL is undecidable.

We will see the proof in later chapters.
As a consequence, deciding whether a given CFL is regular has to be undecidable.

REGULARITYCFL:

Given: A CFG G over Σ.
Question: Is $L(G)$ regular and, if so, give an NFA for it.

Theorem: REGULARITYCFL is undecidable.

In later chapters we will see that the theorem even holds without the "if so" requirement.

Proof: • Towards a contradiction, assume REGULARITYCFL was decidable.
• Using this assumption, we can construct an algorithm to solve UNIVERSALITYCFL. ↯

This is a contradiction, there is no algorithm to solve universality.
Hence, there cannot be an algorithm for regularity.

- Let G be the input to the universality problem.
We use the algorithm for REGULARITY CFL
to check whether $L(G)$ is regular.
- If not, $L(G)$ cannot be Σ^* (because Σ^* is regular)
and we return false.

If so, REGULARITY CFL returns an NFA A for $L(G)$.
We use A to check $L(A) = \Sigma^*$, and return the answer.

- Since this method solves UNIVERSALITY CFL,
the assumption that REGULARITY CFL is decidable
has to be false. □

13.2 Membership and Dynamic Programming

Goal: . Show that membership is decidable (in polynomial time)
for context-free languages.
. Introduce the algorithmic technique of dynamic programming

Dynamic programming: . Accumulate information about smaller subproblems
to solve larger problems
. Store solution to subproblems to avoid recomputing them (memoization)
(make a table where they are stored).

Example: Fibonacci

Naive algorithm:
$$\begin{aligned} fib(5) &= fib(4) + fib(3) \\ &= (fib(3) + fib(2)) + (fib(2) + fib(1)) \\ &= (\underline{fib(2)} + fib(1)) + \underline{fib(2)} + (\underline{fib(2)} + fib(1)) \end{aligned}$$

Dynamic programming algorithm: Store $mem(0) := 0$, $mem(1) := 1$ and set $mem(n) := mem(n-1) + mem(n-2)$.

Idea: Memorization, and dynamic programming in general, is like computing a fixed point on auxiliary information.

Definition:
The problem MEMBERSHIP $L(G)$ with G a context-free grammar in Chomsky normal form (over Σ) is the membership problem for the language:

Given: Input word $w \in \Sigma^*$.
Question: Does $w \in L(G)$ hold?

Idea for dynamic programming: The subproblems determine for each non-terminal A of G and for every infix v of w whether $A \Rightarrow^* v$.

Table: The algorithm enters the solution into an $n \times n$ table, $n = |w|$.
For $i \leq j$, we have
table (i,j) := Non-terminals that generate $w_i \dots w_j$.
For $i > j$, the table entries are not used.

Filling:
- Fill the table entries for all infixes of w
- Increase in length: ↳ Start from infixes of length 1
  ↳ Continue with infixes of length 2
  ↳ ...
- Key: Use entries for the shorter lengths to determine the entries for the longer lengths.

Accept: If start symbol S is in the set table$(1,n)$.

Details on filling:
- Assume we have already determined which non-terminals generate all substrings of length $\leq k$.
- To determine whether A generates w of length $k+1$,

  $w = a_1 \dots a_{k+1}$,

  split w into two non-empty pieces.
  There are k possible ways of splitting w.
- For each split position m,
  let $v_1 := a_1 \dots a_m$ and $v_2 := a_{m+1} \dots a_{k+1}$.
  We examine all rules

  $A \to BC$

  and check whether

  B generates v_1 and

  C generates v_2.

  If so, we add A to the entry for w.

Example: Consider

$G = S \to AB \mid BC$
$A \to BA \mid a$
$B \to CC \mid b$
$C \to AB \mid a$

and $w = baaba$

	1	2	3	4	5
1	$\{B\}$	$\{A,S\}$	$\emptyset$	$\emptyset$	$\{A,S,C\}$
2		$\{A,C\}$	$\{B\}$	$\{B\}$	$\{A,C,S\}$
3			$\{A,C\}$	$\{S,C\}$	$\{B\}$
4				$\{B\}$	$\{A,S\}$
5					$\{A,C\}$

We have $baaba \in L(G)$, because $S \in \{A,S,C\} = \text{table}(1,5)$.

This dynamic programming algorithm is named after Cocke, Younger, and Kasami. Who presented it (independently) in the 1960s.

CYK-Algorithm: Let $G = (N, \Sigma, P, S)$.

input: Word $w = a_1 \dots a_n$.

begin:

for all $i = 1, \dots, n$ do // Initialize diagonal
  $table(i,i) := \{A \in N \mid A \to a_i \in P\}$
od
for all $k = 2, \dots, n$ do // Length
  for all $i = 1, \dots, (n-k)+1$ do // Start of the infix
    $table(i,(i+k)-1) := \emptyset$ // Initialize $table(i,(i+k)-1)$
    for all $m = 1, \dots, k-1$ do // Split length
      $table(i,(i+k)-1) := table(i,(i+k)-1) \cup$
        $\{A \in N \mid A \to BC$ with $B \in table(i,(i+m)-1)$
        and $C \in table((i+m),(i+k)-1)\}$.
    end for all
  end for all
end for all

returns: true, if $S \in table(1,n)$
  false, otherwise.

end.

Complexity analysis:
- There are three nested loops
  - ↳ Length of the infix
    - ↳ Start position of the infix
      - ↳ Split position.
- Hence, the runtime is $O(|w|^3)$.

Theorem:
For every context-free grammar G,
MEMBERSHIP $L(G)$ can be solved in $O(|w|^3)$.

Note: The grammar is not part of the input to the problem.
Therefore, going through the rules $A \to BC$
only adds constant overhead.

13.3 Finiteness

Goal: Check whether a given CFL contains finitely many words.

Motivation: Such boundedness problems are also of importance in verification.
To turn a C-program into hardware,
we have to check that
↳ the stack is bounded in height and
↳ that it allocates a bounded amount of memory.

Note: This requires techniques different from the ones for emptiness.
We have to check that a loop can be repeated,
and hence need a kind of pumping argument.

FINITECFL:
Given: A CFG G.
Question: Is $L(G)$ finite?

Theorem: FINITECFL is decidable.

Proof: An inefficient algorithm can be derived from the pumping lemma.
We convert G into a grammar G' in Chomsky normal form.
Let k be the number of non-terminals in G'.
Let $p_L := 2^k$.

We check whether $L(G)$ contains a word w of length

$$p_L \leq |w| \leq 2p_L.$$

- If so, we return *false*, meaning the language is infinite.
  Clearly, w meets the conditions of the pumping lemma.
- If not, we return *true*, meaning the language is finite.
  Indeed, let u be the shortest word in $L(G)$
  with $|u| \geq p_L$.
  We claim that $|u| \leq 2p_L$.
  Towards a contradiction, assume $|u| > 2p_L$.
  By the pumping lemma, $u = x_1 x_2 x_3 x_4 x_5$
  with $|x_2 x_3 x_4| \leq p_L$ and $x_1 x_3 x_5 \in L$.

  Since $|x_1 x_2 x_3 x_4 x_5| > 2p_L$ and $|x_2 x_3 x_4| \leq p_L$,
  we get $|x_1 x_3 x_5| \geq p_L$.
  A contradiction to minimality of u.
- We can solve these finitely many queries using CYK. □

For a better algorithm, we turn G into a *CNF* G'
for $L(G) \setminus \{\varepsilon\}$ *without useless non-terminals*.
We have $L(G)$ finite iff $L(G')$ is finite.
Let $G' = (N, \Sigma, P, S)$.
From G' we construct a directed graph (V, E)
with $V := N$ // Every non-terminal yields a node
$E := \{A \to B \mid A \to BC \in P \text{ or } A \to CB \in P\}$.

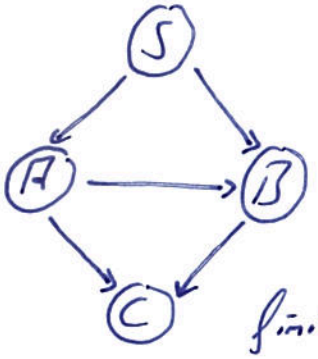
Claim: $L(G')$ is finite iff (V, E) is acyclic.

This holds since the non-terminals produced in a loop

- are guaranteed to derive a word (because they are not useless)
- and the word is not ε (because we have CNF).

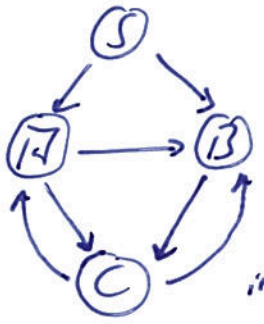
Example:

(1) $S \to AB$
$A \to BC \mid a$
$B \to CC \mid b$
$C \to a$



finite.

(2) $S \to AB$
$A \to BC \mid a$
$B \to CC \mid b$
$C \to AB$



infinite.

10. Pushdown Automata and Context-free Languages

Goal: Develop an automata-theoretic understanding of context-free languages.

Motivation:
- Will make it easier to show some closure properties
- Needed for parsing.

10.1 Pushdown Automata

Goal:
- Introduce a new computational model: finite automata that have access to a stack.
- Can store an unbounded amount of information about the history of the computation, but access this information only in a very restricted way (last-in, first-out).

Definition (Syntax of a pushdown automaton):
A (non-deterministic) pushdown automaton ((N)PDA) is a tuple $M = (Q, \Sigma, \Gamma, q_0, \delta, Q_F)$ with
- Q a finite set of states (also called control states),
- $q_0 \in Q$ the initial state, and $Q_F \subseteq Q$ the set of final states,
- Σ a (finite) input alphabet (alphabets are always finite),
- Γ a (finite) stack alphabet,
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times \Gamma^* \times Q$ a finite set of transitions.

To define the behavior of a PDA, its semantics, we need the notion of a configuration, the stack of the PDA at runtime.

The configuration should contain
- the current control state and
- the current stack content.

Definition (Semantics of a pushdown automaton):
Let $M = (Q, \Sigma, \Gamma, q_0, \delta, Q_F)$ be a PDA.

- The set of configurations of M is $Q \times \Gamma^*$.
  The initial configuration is (q_0, ε).
  A configuration is final (or accepting) if it is of the form $(q_f, \omega) \in Q_F \times \Gamma^*$.
- The labelled transition relation among configurations

$$\longrightarrow_M \subseteq (Q \times \Gamma^*) \times \Sigma \times (Q \times \Gamma^*)$$

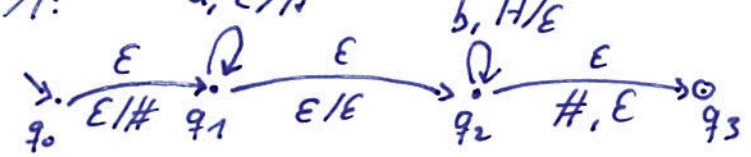
is defined by
$(q_1, \omega_1) \xrightarrow{a}_M (q_2, \omega_2)$, if $q_1 \xrightarrow[v_1/v_2]{a} q_2 \in \delta$ (this is the tuple (q_1, a, v_1, v_2, q_2))
and $\omega_1 = \omega . v_1$
and $\omega_2 = \omega . v_2$ for some $\omega \in \Gamma^*$.

We generalize the labelled transition relation to words from Σ^* in the expected way.

- The language of M is
$$L(M) := \{ w \in \Sigma^* \mid (q_0, \varepsilon) \xrightarrow{w} (q_f, u) \in Q_F \times \Gamma^* \}.$$

Example:

M:



$L(M) = \{ a^n b^n \mid n \in \mathbb{N} \}$.

Note:
If PDA M induces an infinite-state automaton

$$\mathcal{A}_M := (Q \times \Gamma^*, \Sigma, (q_0, \varepsilon), \rightarrow_M, Q_F \times \Gamma^*)$$

and $L(M) = L(\mathcal{A}_M)$
(where the language of an infinite-state automaton is defined like for NFAs).

Remark:
There are alternative definitions of PDAs in the literature.
(1) A common alternative is to define acceptance
by reaching the empty stack.
In this case, the PDA is a tuple $M = (Q, \Sigma, \Gamma, q_0, \#, \delta)$
with $\# \in \Gamma$ a symbol that is on the stack initially.
(2) In both models (acceptance via final states and via empty stack),
we may assume that transitions
- always pop an element and
- push at most two elements.

For the former restriction, we have to require
to start with an initial stack symbol
also when accepting with final states.

Theorem:
The alternative definitions accept the same languages as PDAs.

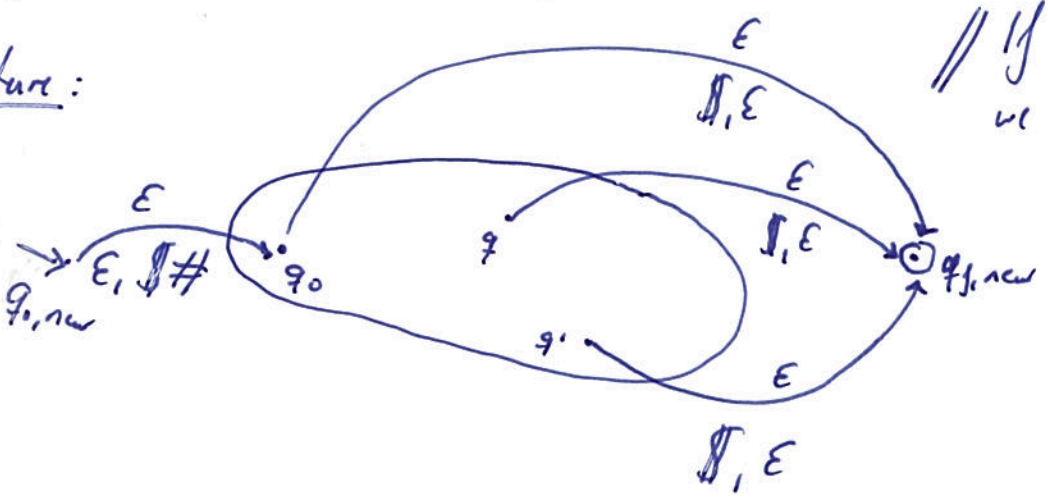
Proof (Sketch):
- To mimic empty stack acceptance with our definition,
- let $\#$ be the distinguished initial stack symbol.

We introduce
- a new initial state,
- a new final state, and
- a new bottom of stack symbol $\$$.

We introduce transitions that
- initially establish the bottom of stack $\$$
- the symbol is never removed except in the last step that takes the PDA from any state to the new final state.

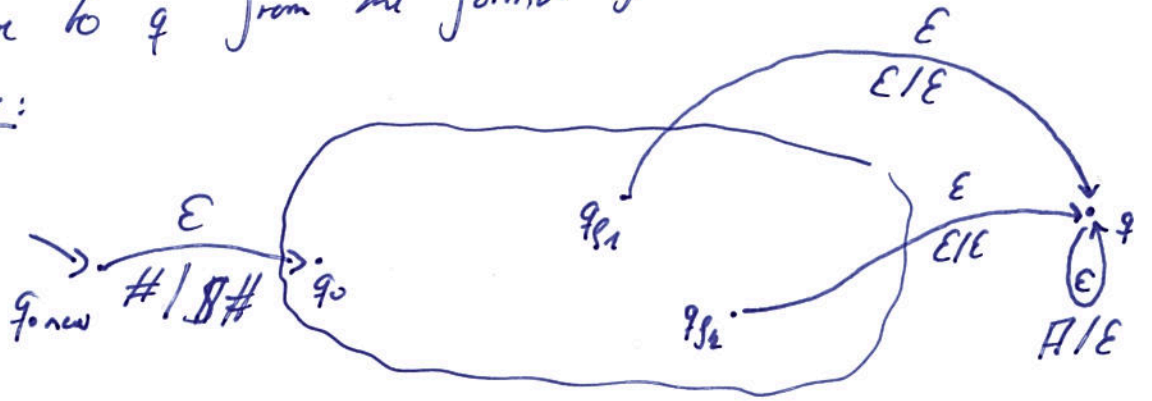
// If we can read $\$$, we are sure the stack is empty.

Picture:



- To mimic acceptance with a final state via empty stack, add a new bottom of stack symbol $\$$ so that the stack is not accidentally emptied. Moreover, add a new state q in which to empty the stack. Move to q from the former final states.

Picture:



- The remaining equivalences can be solved as homework. □

10.2 Equivalence with Context-free Languages

Goal: Show that the languages accepted by PDTAs are precisely (coincide with) the context-free languages.

We establish the two directions separately.

Theorem:
If L is context-free, then there is a PDTA A with $L = L(A)$.

Idea:
- Mimic the left-derivation relation $\Rightarrow_L$ using the stack.
- We give an elegant proof that makes use of Greibach's normal form.
- The normalization step can be avoided.

Proof:
Let $L = L(G)$ with $G = (N, \Sigma, P, S)$ a context-free grammar.
Assume the grammar is in Greibach normal form so that the productions are

$$A \to a B_1 \dots B_k \quad \text{and potentially } S \to \varepsilon.$$

We construct an equivalent PDTA A that accepts with empty stack.
Interestingly, A has only one stack.
The definition is

$$A := (\{q\}, \Sigma, N, q, \underbrace{S}_{\text{initial stack content}}, \delta)$$

(N: stack alphabet)

For every production $A \to a B_1 \dots B_k$ there is a transition

$$q \xrightarrow{a} q \quad A / B_k \dots B_1.$$

So we read letter a from the input,
read A from the top-of-stack, and
write $B_n \dots B_1$ to the top-of-stack.

Similarly, $S \to \varepsilon$ yields $q \xrightarrow[S/\varepsilon]{\varepsilon} q$.

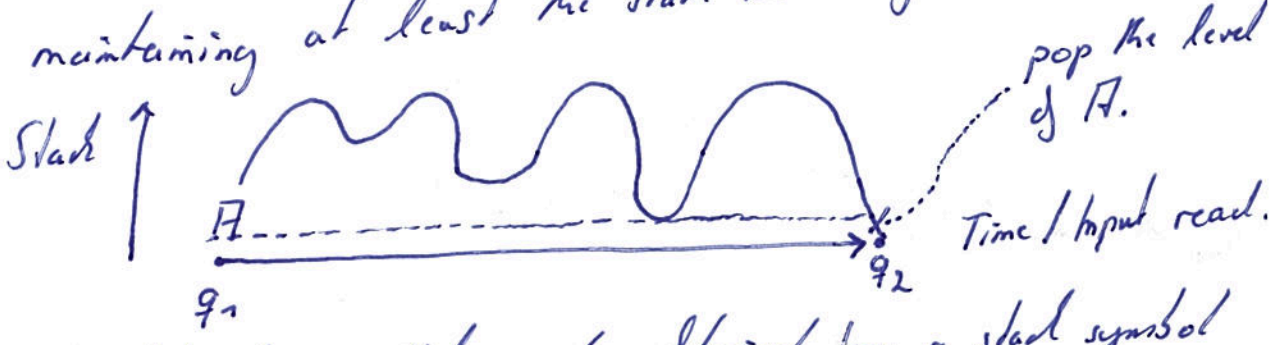
Claim: $L(M) = L(G)$. □

Discussion:

- One can give a similar construction
that does not rely on Greibach normal form.
Doing this is a good exercise.
- The benefit of the above construction is that
we definitely read a letter from the input word with every step.
So we can decide in NP whether a given word is accepted.

Theorem:
Every language $L(M)$ of a PDA M is context-free.

Idea: - We guess the changes in the control state of the PDA,
say from q_1 to q_2 (potentially far away),
that can be obtained from a stack symbol A
while maintaining at least the stack level of A.



- Guessing the stack changes that can be obtained from a stack symbol
means we use non-terminals of the form (q_1, A, q_2).

Proof:

Consider $L(M)$ with $M = (Q, \Sigma, \Gamma, q_0, \#, \delta)$ accepting with empty stack.

We assume $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times \Gamma^* \times Q$,

so we definitely read a stack symbol in every transition.

We define the grammar to be

$$G := ((Q \times \Gamma \times Q) \cup \{S\}, \Sigma, P, S)$$

with

$$P := \{ S \to (q_0, \#, q) \mid q \in Q \}$$

$$\cup \{ (q, A, q') \to a \,.\, (q'', B_m, q_1)\ldots(q_{m-1}, B_1, q') \mid$$

$q_1, \ldots, q_{m-1} \in Q$ (m may be 0) and

(*) $q \xrightarrow[A / B_1 \ldots B_m]{a} q''$ with $a \in \Sigma \cup \{\varepsilon\}\}$. □

Claim: $L(M) = L(G)$.

Putting the results together, we obtain an interesting observation about the role of control states in pushdown automata.

Corollary:

For every PDA M, there is a PDA M' that only has one control state and satisfies $L(M) = L(M')$.

(*) Rephrasing the idea of the construction:

Non-terminal (q_1, A, q_2) generates $w \in \Sigma^*$ iff upon input w and with initial stack symbol A, M moves from q_1 to q_2 and ends with an empty stack.

11. Closure Properties of Context-free Languages

11.1 Positive Results

Goal: Show closure under $\cup$, $\cdot$, *, h, h^{-1}, substitution, and regular intersection.

Theorem:
The context-free languages are closed under union, concatenation, and Kleene star.

Proof:
Assume the languages are given by context-free grammars.
The constructions are easy. □

Definition:

- A substitution is a function
$$\sigma : \Sigma \longrightarrow \mathcal{P}(\Delta^*)$$
that assigns to each letter a a language $\sigma(a)$ over Δ.
A substitution is
  - regular, if $\sigma(a)$ is a regular language for each $a \in \Sigma$
  - context-free, if $\sigma(a)$ is a context-free language for each $a \in \Sigma$.
- The application of σ to $w \in \Sigma^*$ yields the language
$$\sigma(w) := \{\varepsilon\} \quad \text{, if } w = \varepsilon$$
$$\sigma(w) := \sigma(a_1) \dots \sigma(a_n) \text{, if } w = a_1 \dots a_n.$$
The application of σ to $L \subseteq \Sigma^*$ yields
$$\sigma(L) := \bigcup_{w \in L} \sigma(w).$$

Note: Homomorphisms are substitutions where $\sigma(a)$ consists of a single word for each $a \in \Sigma$.

Theorem:

- The regular languages are closed under regular substitutions.
- The context-free languages are closed under context-free substitutions:

  If L is context-free and $\sigma(a) \subseteq \Delta^*$ is context-free for all $a \in \Sigma$,

  then $\sigma(L) \subseteq \Delta^*$ is context-free.

Proof:

We show the statement for context-free languages.

Let $G = (N, \Sigma, P, S)$ be the grammar for L.

Let $G_a = (N_a, \Delta, P_a, S_a)$ be the grammar for $\sigma(a)$, for each $a \in \Sigma$.

W.l.o.g. we assume the non-terminals in G and in the G_a are pairwise disjoint.

We replace every occurrence of a in a right-hand side of a rule in P by S_a, the start symbol of G_a. □

Corollary: The context-free languages are closed under homomorphisms.

Theorem:

The context-free languages are closed under inverse homomorphisms:

If $L \subseteq \Delta^*$ is context-free and $h: \Sigma^* \to \Delta^*$ is a homomorphism,

then $h^{-1}(L) \subseteq \Sigma^*$ is context-free.

Proof:

We use an automata-theoretic construction.

Let $L = L(M)$ with $M = (Q, \Delta, \Gamma, q_0, \delta, Q_F)$ a PDA.

We construct a PDA M' accepting $h^{-1}(L)$.

Idea:

- We let M', on input a, generate the word $h(a)$ and simulate M on $h(a)$.

- Since the length of the words is bounded (by $\max\{|h(a)| \mid a \in \Sigma\}$), we can store them in the control state of M'.
  (There is a general rule-of-thumb in automata theory:
  If some information is bounded, put it into the control state.)
- Every time M wants to read an input symbol, M' removes a letter from the currently stored word (which is a suffix of some $h(a)$).
  If the word becomes empty (ε), M' accepts the next input symbol.
- A state of M' is final, if the auxiliary word is empty (ε) and the control state of M is final.

Construction:

We define

$$M' := (Q', \Sigma, \Gamma, (q_0, \varepsilon), \delta', Q_F \times \{\varepsilon\})$$

where

$$Q' := \{(q, x) \mid q \in Q,\ x \text{ a suffix of } h(a) \text{ for some } a \in \Sigma\}.$$

The transition relation δ' is defined as follows:

↳ $(q, \varepsilon) \xrightarrow[\varepsilon / \varepsilon]{a} (q, h(a))$ for all $a \in \Sigma$ // Get new input.

↳ $(q, ax) \xrightarrow[v_1 / v_2]{\varepsilon} (p, x)$, if $q \xrightarrow[v_1 / v_2]{a} p \in \delta$ // M reads an input from the auxiliary word.

↳ $(q, x) \xrightarrow[v_1 / v_2]{\varepsilon} (p, x)$, if $q \xrightarrow[v_1 / v_2]{\varepsilon} p \in \delta$ // M does an ε-transition.

□

Theorem (Closure under regular intersection):

If $L \subseteq \Sigma^*$ is context-free and $R \subseteq \Sigma^*$ is regular,

then $L \cap R$ is context-free.

It is easy to establish the theorem using pushdown automata.

We give a grammar-based construction instead
that once more practices the idea
of summarizing a computation
that we saw when converting a PDA to a CFG.

Proof:
Let $L = L(G)$ with $G = (N, \Sigma, P, S)$ in Chomsky normal form,
potentially with $S \to \varepsilon$ and then S not in right-hand sides.
Let $R = L(A)$ with $A = (Q, \Sigma, q_0, \to, Q_F)$ an NFA
without ε-transitions.
We construct the grammar $G' = (N', \Sigma, P', S')$ for $L \cap R$ as follows.
The non-terminals are

$$N' := (Q \times N \times Q) \cup \{S'\}.$$

For the productions, we have

$$S' \to \varepsilon \quad \text{iff} \quad \varepsilon \in L \cap R.$$

Moreover, there are productions

$$S' \to (q_0, S, q_f) \quad \text{for each } q_f \in Q_F.$$

For every non-terminal (q_1, A, q_2) we have

$(q_1, A, q_2) \to a$, if $q_1 \xrightarrow{a} q_2$ in the NFA and $A \to a \in P$.

$(q_1, A, q_2) \to (q_1, B, q).(q, C, q_2)$, if $A \to BC \in P$.

There are variants of the latter production for all $q \in Q$.

Intuitively, the grammar guesses the state change in A
that is induced by the word derived from B (and from C). □

We again employ the closure properties to disprove context freeness.

Example (Application of closure properties):

We show that

$$L = \{ww \mid w \in \{a,b\}^*\} \text{ is not context-free.}$$

- Assume it was. Then by closure under regular intersection also

$$L_1 = L \cap a^+b^+a^+b^+ = \{a^ib^ia^jb^j \mid i,j \geq 1\}$$

is context-free.

But L_1 is not context-free by an application of the pumping lemma.

$\lightning$ L cannot be context-free.

- If we do not want to apply the pumping lemma to L_1, we can further reduce the language to

$$L_2 = \{a^ib^ic^id^i \mid i,j \geq 1\}.$$

Consider $h: \{a,b,c,d\}^* \to \{a,b\}^*$ defined by

$$h(a) := a =: h(c) \quad \text{and} \quad h(b) := b =: h(d).$$

Then

$$h^{-1}(L) = \{x_1x_2x_3x_4 \mid |x_1| = |x_3| \text{ and } x_1, x_3 \in \{a,c\}^* \text{ and } |x_2| = |x_4| \text{ and } x_2, x_4 \in \{b,d\}^*\}.$$

Then $h^{-1}(L) \cap a^*b^*c^*d^* = L_2$.

Moreover, if L was context-free, then so was L_2.

But L_2 is not context-free by the pumping lemma.

$\lightning$ L cannot be context-free. □

11.2 Negative Results

Goal: Show that the context-free languages are not closed under intersection and under complement.

Theorem:

(1) There are context-free languages L_1 and L_2 so that $L_1 \cap L_2$ is not context-free.

(2) There is a context-free language L where $\overline{L}$ is not context-free.

Proof:

(1) The languages are

$$L_1 = \{ a^n b^m c^n \mid n, m \in \mathbb{N} \} \text{ and}$$

$$L_2 = \{ a^n b^n c^m \mid n, m \in \mathbb{N} \}$$

with $L_1 \cap L_2 = \{ a^n b^n c^n \mid n \in \mathbb{N} \}$, known to be not context-free.

(2) If the context-free languages were closed under complement (which means for each context-free language L, $\overline{L}$ is context-free), then, due to closure under union, we would obtain

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}} \quad \text{being context-free,}$$

for all L_1 and L_2 context-free languages. $\lightning$ Contradiction to (1).

For a constructive proof, one shows that

$$L = \{a, b\}^* \setminus \{ ww \mid w \in \{a, b\}^* \}$$

is context-free.

The complement is

$$\overline{L} = \{ ww \mid w \in \{a, b\}^* \},$$

which we just proved to be not context-free. □

8. Pumping Lemmas for Context-Free Languages

Goal: Introduce pumping lemmas for context-free languages that allow us to show that a given language is not context-free.

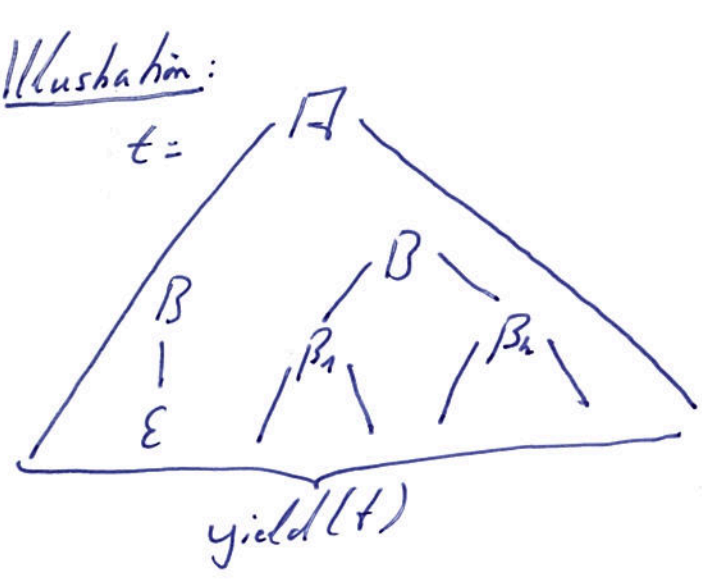
8.1 Parse Trees

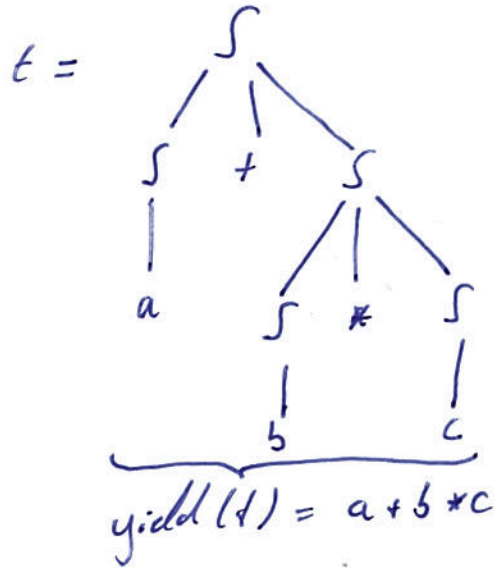
Definition:

- A parse tree from a non-terminal B in $G = (N, \Sigma, P, S)$ is a tree with the following properties:
  - Every node is labelled with a symbol from $N \cup \Sigma \cup \{\varepsilon\}$.
    The root is labelled by B.
    Every inner node is labelled by a symbol from N.
  - If B is an inner node with k children labelled $\beta_1, \dots, \beta_k$ (from left to right), then
    - $\beta_1, \dots, \beta_k \in N \cup \Sigma$ and
      $B \to \beta_1 \dots \beta_k \in P$

    or
    - $k = 1$, $\beta_1 = \varepsilon$, and $B \to \varepsilon \in P$.
- The yield of a parse tree t, denoted by $\text{yield}(t) \in (N \cup \Sigma)^*$, is the concatenation of the leaves from left to right.

Illustration:





Definition:

The parse tree associated with a derivation

$$\Pi = \alpha_0 \Rightarrow \dots \Rightarrow \alpha_n$$

is defined by induction on the length of the derivation:

$n=0$: For the empty derivation from A to A, we obtain the parse tree with a single node (root and leaf) A

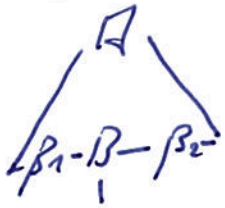
$n \to n+1$: Consider

$$\underbrace{A}_{\alpha_0} \to \dots \to \underbrace{\beta_1 B \beta_2}_{\alpha_n} \to \underbrace{\beta_1 \gamma \beta_2}_{\alpha_{n+1}}.$$

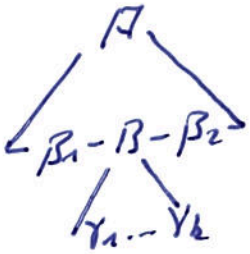
Let t be the parse tree for $A \to \dots \to \beta_1 B \beta_2$.

We extend t depending on $B \to \gamma$.

↳ If $\gamma = \varepsilon$, we obtain



↳ If $\gamma = \gamma_1 \dots \gamma_k$ with $\gamma_i \in N \cup \Sigma$, we get



Theorem:

Let $G = (N, \Sigma, P, S)$, $A \in N$, and $\alpha \in (N \cup \Sigma)^*$.

(1) $A \Rightarrow^* \alpha$ iff there is a parse tree t from A with $yield(t) = \alpha$.

(2) For every derivation $A \Rightarrow^* \alpha$, there is a unique parse tree (the one associated with it).

(3) The same parse tree may be associated with several derivations, but with only one left-derivation and only one right-derivation

8.2 The Pumping Lemma

Goal: Introduce the classic pumping lemma for CFLs.

Theorem (Pumping lemma, Bar-Hillel, Perles, Shamir '61):

Let L be a CFL.

There is a constant p_L so that for all $z \in L$ with $|z| \geq p_L$

there is a decomposition

$$z = uvwxy$$

satisfying

(1) $|vx| \geq 1$

(2) $|vwx| \leq p_L$

(3) for all $i \in \mathbb{N}$, $uv^i wx^i y \in L$.

Proof:

Let G be a context-free grammar in Chomsky normal form generating $L \setminus \{\varepsilon\}$.

Let k be the number of non-terminals in G.

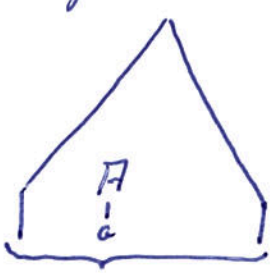
We define

$$p_L := 2^k.$$

Consider a word $z \in \Sigma^*$ with $|z| \geq p_L$.

- The derivation tree of z is – due to the Chomsky normal form – a binary tree, up to the last step $A \to a$:

$t =$

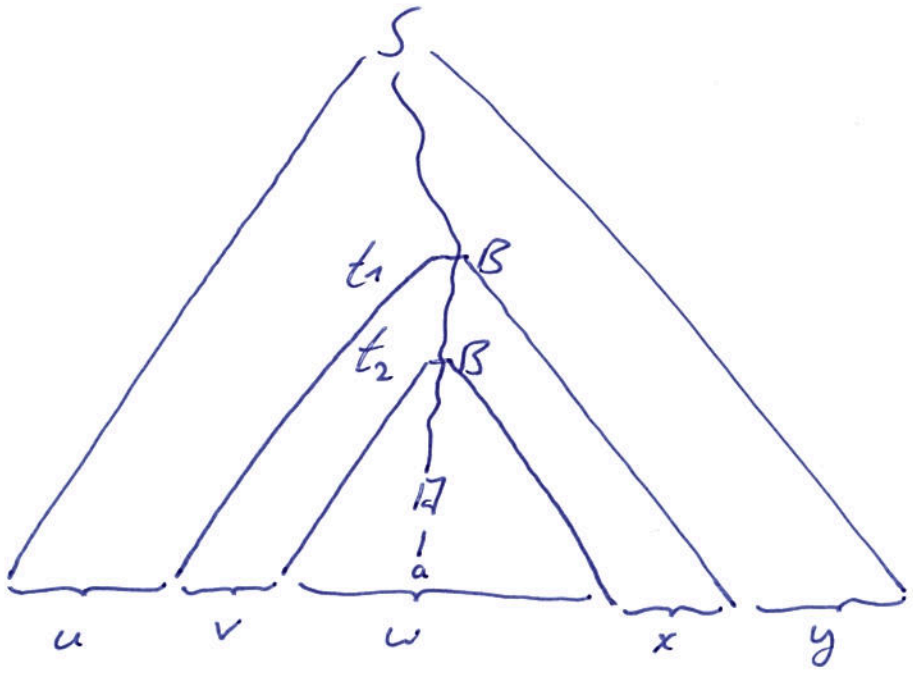


$\text{yield}(t) = z$ with $|z| \geq p_L$.

- Since the tree has at least 2^k leaves and the last step is $A \to a$, it has height at least $k+1$.

  Indeed, a full binary tree of height n has 2^n leaves.

- Fix a longest path (which has $\geq k+2$ nodes).
  There are $\geq k+1$ non-terminals on this path.
  But the grammar only has k non-terminals.
  Hence, by the pigeon hole principle, one non-terminal has to repeat.
  We consider the first repetition from the leaves.



- The upper B is at most $k+1$ steps from the leaves.
  This means the subtree t_1 rooted at the upper B has a yield of length $\leq 2^k$.
  Let t_2 be the subtree rooted at the lower B.
  Let w be the yield of t_2.
  We now have

  $$\text{yield}(t_1) = v\,w\,x \quad \text{for suitable } v, x \in \Sigma^* \text{ with } |vwx| \leq 2^k = p_L$$

- We have $v \neq \varepsilon$ or $x \neq \varepsilon$.
  To see this, note that the first production applied to the upper B is of the form $B \to CD$.
  Now t_2 is completely inside the tree rooted at C or inside the tree rooted at D.
  Hence, x is (at least partly) derived from D or v is (at least partly) derived from C.
  Therefore, x or v is not ε.
- Finally, we note that
  $$B \Rightarrow^* vBx \quad \text{and} \quad B \Rightarrow^* w.$$
  Hence, $B \Rightarrow^* v^i w x^i$ for all $i \in \mathbb{N}$. □

Example:
$L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not context-free.

Proof:
Assume L was context-free with pumping constant p_L.
Consider $a^{p_L} b^{p_L} c^{p_L}$.
We decompose the word into $uvwxy$ so as to satisfy the conditions of the pumping lemma.
We ask ourselves where v and x, the words that get pumped, could lie.
Because $|vwx| \leq p_L$, it is impossible for vx to contain a's and c's.

- If vx consists of a's only, then $uv^0wx^0y = uwy$ has p_L many b's and p_L many c's, but $< p_L$ many a's, since $|vx| \geq 1$. ↯

- If vx consists of b's only, of c's only,
  of a's and b's only, and of b's and c's only,
  the contradiction is derived similarly.

  Hence, L is not context-free. □

8.3 Ogden's Lemma

Goal: · Prove a stronger pumping lemma for context-free languages.
· The usual pumping lemma sometimes does not apply
(see the example below).

Theorem (Ogden '68):

Let L be a context-free language.
There is a constant p_L so that for all $z \in L$
where we mark $\geq p_L$ positions as distinguished
there is a decomposition

$$z = uvwxy$$

with
(1) v and x together have at least one distinguished position
(2) vwx has at most p_L distinguished positions
(3) for all $i \in \mathbb{N}$, $uv^i wx^i y \in L$.

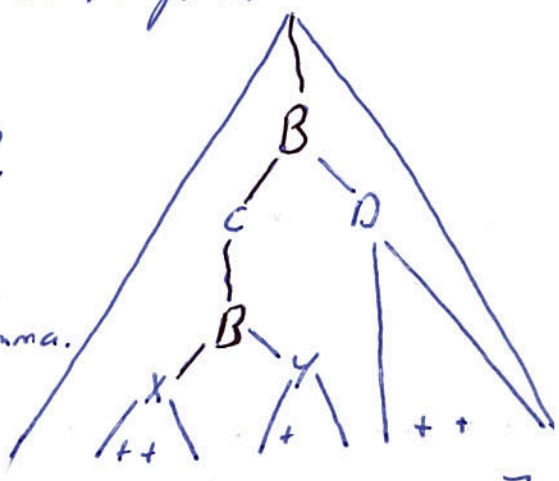
Proof:
- Let G be in Chomsky normal form generating $L \setminus \{\varepsilon\}$.
  Let G have k non-terminals and define $p_L := 2^k + 1$.
  Let z be a word with $\geq p_L$ positions marked.
- Our first goal is to construct a path
  in the parse tree for z's derivation.

Since we have to care about the distinguished positions,
we cannot just pick a longest path.
Moreover, we cannot pick arbitrary repeating non-terminals.
Instead, we have to consider branch points:
non-terminals where both children have distinguished descendants.

- We construct the path inductively:
  - ↳ The root is on the path.
  - ↳ Suppose r is the last node on the path.
    - ↳ If r is a leaf, the construction ends.
    - ↳ If r has only one child w.l.o.g. with distinguished descendants, add that child to the path and continue from there.
    - ↳ If both children of r have distinguished descendants, call r a branch point and continue from the child with the larger number of distinguished descendants (arbitrary if equal).

- Each branch point has at least half as many distinguished descendants as the previous branch point.
Since there are strictly more than 2^k (namely $\geq 2^k+1$) distinguished positions in z,
there are at least $k+1$ branch points on the path.

- Among the last $k+1$ branch points
there are two with the same non-terminal B.
Select the two closest to the leaves
and argue like in the previous pumping lemma.



□

Remark (Conservative extension):

If we apply Ogden's lemma to a word where all positions are marked, we obtain the classic pumping lemma.

Example (Application of Ogden's lemma):

$L = \{a^i b^j c^k \mid i \neq j,\ j \neq k,\ \text{and } k \neq i\}$ is not context-free.

Proof:

Suppose L was context-free.

Let p_L be the constant in Ogden's lemma.

We consider the word

$$z = a^{p_L} b^{p_L + p_L!} c^{p_L + 2p_L!} \in L.$$

Let the positions of the a's be distinguished.

Let

$$z = uvwxy$$

be a decomposition that satisfies the conditions in Ogden's lemma.

↳ If v or x contains two different symbols,

then $uv^2wx^2y \notin L$.

For example, if $v \in a^+b^+$, then vv has a b followed by an a.

↳ Now at least one of v and x must contain a's,

since only a's are distinguished positions.

So if $x \in b^*$ or $x \in c^*$, then $v \in a^+$.

Moreover, if $x \in a^+$ then $v \in a^*$.

↳ We focus on the situation where $x \in b^*$ and $v \in a^+$.

The remaining cases are similar.

Note that $1 \leq |v| \leq p_L$ since there are at most p_L-many a's.
This means $|v|$ divides $p_L!$ $(= p_L \cdot (p_L - 1) \cdot (p_L - 2) \cdot \ldots \cdot 2 \cdot 1)$.
Let q be such that $|v| \cdot q = p_L!$.

Then
$$z' = u\, v^{2q+1}\, w\, x^{2q+1}\, y \in L.$$

But z' has
$$\underbrace{p_L - |v|}_{a^{p_L - |v|}} + (2q+1) \cdot |v| = p_L + 2 \cdot \underbrace{q \cdot |v|}_{p_L!} = p_L + 2p_L! \text{ many a's.}$$

However, since v and x have no c's,
z' also has $p_L + 2p_L!$ many c's. $\lightning$ $z' \notin L$.

A similar contradiction occurs if $x \in a^+$ or $x \in c^*$.
Therefore, L is no context-free language. □

14. Context-Sensitive Languages

Goal:
- Introduce context-sensitive languages (CSLs) and Turing machines.
- Prove decidability of membership in CSLs.
- Show correspondence of CSLs and languages accepted by linear-bounded Turing machines.
- This also establishes equivalence of the recursively enumerable languages and the languages accepted by Turing machines.

Definition:
- A type-0 grammar is a grammar $G = (N, \Sigma, P, S)$ with $P \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$.
- The grammar is context-sensitive (or type-1), if for all productions $\alpha_1 \to \alpha_2 \in P$ we have $|\alpha_1| \leq |\alpha_2|$.

  Moreover, we may have $S \to \varepsilon$, but then there is no rule that has S in its right-hand side.
- A language $L \subseteq \Sigma^*$ is context-sensitive, if there is a context-sensitive grammar G with $L = L(G)$.
- A language $L \subseteq \Sigma^*$ is recursively enumerable, if $L = L(G)$ for G of type-0.

Example:

$L = \{ w \in \{a,b,c\}^* \mid \#_a(w) = \#_b(w) = \#_c(w) \}$.

We have $L = L(G)$ with

$G = S \to \varepsilon \mid R$

$R \to TABC \mid ABCR$

$T \to a \quad B \to b \quad C \to c$

$TB \to BT \quad CT \to TC$

$TC \to CT \quad CB \to BC$

$BT \to TB$

$RT \to CB$

Let $L(G)$ be a context-sensitive language (over Σ).
The membership problem is

MEMBERSHIP $L(G)$:

Given: $w \in \Sigma^*$.

Question: $w \in L(G)$?

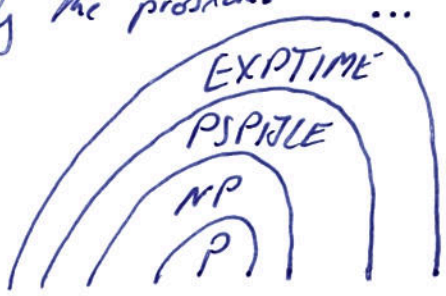
Theorem: For every context-sensitive grammar G,
MEMBERSHIP $L(G)$ can be solved in $2^{O(|w|)}$.

Proof:
- Since the productions are length preserving,
once we derived a sentential form of length $> |w|$,
there is no chance to derive w from it.
- We thus enumerate all sentential forms of length $\leq |w|$
and see whether we find w.
- There are
$$(|N| + |\Sigma| + 1)^{|w|} = 2^{O(|w|)} \text{ many,}$$
which yields the time bound.
Note that N and Σ are not part of the input
and therefore $|N| + |\Sigma| + 1 = 2^c$
for c a constant that is independent of the input (word w). □

The problem is hard:
- The correspondence with linear-bounded Turing machines
that we establish below shows that

CSL = PSPACE,

the context-sensitive languages are precisely the problems solvable with polynomial space.



- Since there are PSPACE-hard problems,
  there are context-sensitive languages $L(G)$
  for which membership is PSPACE-hard.

→ PSPACE is an important complexity class
  for verification problems:
  ↳ Reachability in Boolean shift-programs
  ↳ Reachability in multi-threaded programs.

→ Context-sensitive aspects show up in compilers
  ↳ Parameters in procedures
  ↳ Scopes of objects.

14.1 Turing Machines

Goal: Define an automaton model, the Turing machine (TM),
for context-sensitive languages and
for recursively enumerable languages.

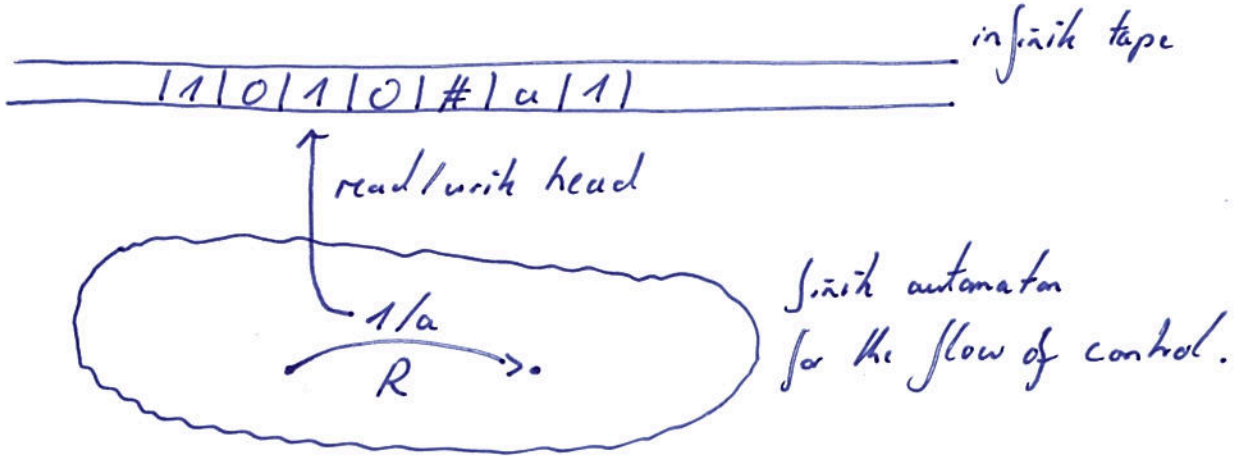
Idea:
- Drop the last-in-first-out restriction for pushdowns
  and access arbitrary information
  stored about the computation so far.
- For context sensitivity, we just bound the amount of information
  we can store.

Church's thesis ('Alonzo Church 1936):
Turing's idea was more than capturing a class of languages.
He wanted to formulate the idea of computability itself,
wanted to capture what is an algorithm.
Up to today, we believe he succeeded.
EVERY problem for which we found an algorithm (C, C++, Java)
we were able to solve with a Turing machine
(even with low overhead).

Church's thesis states that it will stay this way:

Everything that is computable is computable by a TM.

Technically:



Definition (Syntax of a TM):

A Turing machine (TM) is a tuple $M = (Q, \Sigma, \Gamma, q_0, \sqcup, \delta, Q_F)$ with

- Q a finite set of states, $q_0 \in Q$ the initial state, $Q_F \subseteq Q$ the set of final states,
- Σ the input alphabet,
- Γ the tape alphabet with $\Sigma \subseteq \Gamma$ and $\sqcup \in \Gamma \setminus \Sigma$ the blank symbol,
- $\delta : Q \times \Gamma \longrightarrow \Gamma \times \{L(eft), R(ight), N(eutral)\} \times Q$, in which case the TM is deterministic (DTM), or $\delta \subseteq Q \times \Gamma \times \Gamma \times \{L, R, N\} \times Q$ for a non-deterministic TM (NTM)

To define the semantics of a TM, we need the notion of a configuration, the state of the TM at runtime.

Definition (Semantics of a TM):

Let $M = (Q, \Sigma, \Gamma, q_0, \sqcup, \delta, Q_F)$ be a TM.

- A configuration of M is a word from $\Gamma^* Q \Gamma^+$.

If $uqav$, then uav are the cells of the tape already visited,
q is the current state and
$a \in \Gamma$ is the symbol currently under the read-write head.
Given input $x \in \Sigma^+$, the corresponding <u>initial configuration</u> is $q_0 x$.
Given input ε, the <u>initial configuration</u> is $q_0 \sqcup$.

- A configuration is <u>accepting</u> if it is from $\Gamma^* Q_F \Gamma^+$.
- The <u>transition relation</u> among <u>configurations</u>

$$\longrightarrow \subseteq (\Gamma^* Q \Gamma^+) \times (\Gamma^* Q \Gamma^+)$$

is defined by

$uaqbv \longrightarrow uq'acv$, if $q \xrightarrow[L]{b/c} q' \in \delta$

$uaqbv \longrightarrow uaq'cv$, if $q \xrightarrow[N]{b/c} q' \in \delta$

$uaqbv \longrightarrow uacq'v$, if $v \neq \varepsilon$ and $q \xrightarrow[R]{b/c} q' \in \delta$

$uqb \longrightarrow ucq'\sqcup$, if $q \xrightarrow[R]{b/c} q' \in \delta$

$qbu \longrightarrow q'\sqcup cu$, if $q \xrightarrow[L]{b/c} q' \in \delta$.

We use $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

- The <u>language of M</u> is

$$L(M) := \{ w \in \Sigma^* \mid q_0 w \longrightarrow^* u q_f v \in \Gamma^* Q_F \Gamma^+ \}.$$

($q_0 \sqcup$ for ε)

For the characterization of the context-sensitive languages,
we define Turing machines that only use the part of the tape
given by the input (plus a left and a right end marker).

<u>Definition (Linear-bounded automaton)</u>:

- A <u>linear-bounded automaton</u> (LBA) is an NTM

$$M = (Q, \Gamma, \Sigma \cup \{\$_l, \$_r\}, q_0, \delta, Q_F).$$

Instead of a blank $\sqcup$, we have a left end marker $\$_l$
and a right end marker $\$_r$.

Moreover, there are two restrictions
1) There are no moves left from $\$_l$ and right from $\$_r$.
2) The symbols $\$_l$ and $\$_r$ are never overwritten.

The Language of M is
$$L(M) := \{ w \in \Sigma^* \mid q_0 \$_l w \$_r \to^* u q_f v \in \Gamma^* Q_F \Gamma^* \}.$$

A tape compression result in complexity theory shows
that an amount of tape linear in the size of the input
leads to the same computational power (as restricting to the input itself).
Hence the name linear-bounded automaton.

14.2 Equivalence with Context-Sensitive Languages

Goal: Show that the languages accepted by LBAs
are precisely the context-sensitive languages.

Theorem (Kuroda '64):
A language $L \subseteq \Sigma^*$ is
accepted by an LBA iff it is context-sensitive.

Proof:
"$\Leftarrow$" Let $L = L(G)$ with $G = (N, \Sigma, P, S)$.
We give an NTM M that accepts L.
The idea is to apply the productions backwards
until the start symbol is found.
↳ Consider an input $x \in \Sigma^*$.
↳ M non-deterministically selects a production
$$\alpha \to \beta \in P$$
and finds an arbitrary occurrence of β in x.

↳ Now β is replaced by α,
potentially moving letters to the left if $|\alpha| < |\beta|$.

↳ If we reach S, we stop and accept.
Otherwise, we repeat the above non-deterministic procedure.

Since the productions are length preserving ($|\beta| \geq |\alpha|$),
we never exceed the part of the tape used by the input.

$\Rightarrow$ Let $L = L(M)$ for an LBA $M = (Q, T, \Sigma \dot\cup \{\$_\ell, \$_r\}, q_0, \delta, Q_E)$.

Idea:
- We give a grammar G that operates on sentential forms representing the configurations of M.
- We will have to generate the initially given word from these sentential forms.
  Therefore, they cannot be longer than the input (we cannot delete).
- Note that this bound works for the intermediary configurations due to the linear boundedness assumption.

Technically:
- We will store pairs
  $(act_1, a_1) \dots (act_n, a_n)$
  where
  $act_1 \dots act_n$ forms the current tape content and
  $a_1 \dots a_n$ is the initial input.
- The letters act_i are taken from the alphabet Δ with
  $\Delta' := T \cup (\{\$_\ell, \$_r\} \times T)$ and $\Delta := \Delta' \cup (Q \times \Delta') \cup (\{\$_\ell, \$_r\} \times Q \times T)$.
  So an intermediary configuration
  $\$_\ell \times q y a z \$_r$

derived from input

$\$_\ell\ a\ b\ a\ b\ \$_r$

has the shape

$((\$_\ell, x), a)((q, y), b)(a, a)((\$_r, \varepsilon), b)$.

- It is immediate to simulate the moves of the LBA by considering only the current configuration (one has to track a bit at the border (*)).

So if $q \xrightarrow[R]{a/b} q' \in \delta$, then we obtain the productions

$((q, a), x).(c, y) \longrightarrow (b, x).((q', c), y)$ for all $x, y \in \Sigma$

in the grammar.

(*) For the border, we use
$(q, \$_\ell, a)$ to indicate that the head is on the left end marker
and $(\$_\ell, q, a)$ to indicate that the head is on the first letter.

Construction:

We define
$G := (N, \Sigma, P, S)$

with $N := \{S, A\} \cup (\Delta \times \Sigma)$

$P := \{ S \to A.((\$_r, a), a) \mid a \in \Sigma \}$
$\cup \{ A \to A.(a, a) \mid a \in \Sigma \}$
$\cup \{ A \to ((q_0, \$_\ell, a), a) \mid a \in \Sigma \}$
$\cup$ Productions that simulate the behavior of M
$\cup \{ ((q_f, a), b) \to b \mid q_f \in Q_f,\ a \in \Delta',\ b \in \Sigma \}$
$\cup \{ (a, b) \to b \mid a \in \Delta',\ b \in \Sigma \}$.

□

Note: The construction works the same way
if we drop
↳ the length-preserving condition for grammars and
↳ the linear boundedness assumption for TMs.

Corollary:
The languages accepted by an NTM are precisely
the recursively enumerable languages.

14.3 Determinism

Recall: · For finite automata, we know that
NFAs and DFAs accept the same languages.
· For pushdown automata, we know that
DPDAs accept a strict subclass of the context-free languages.

Open: Kuroda, in his 1964 paper, posed two questions:

(1) The first question is indeed on
the role of determinism for LBAs:
Are the languages accepted by NLBAs
precisely the languages accepted by DLBAs?

It is still open!

(2) The second question was on complementation:
Are the languages accepted by NLBAs
(the context-sensitive languages)
closed under complement?

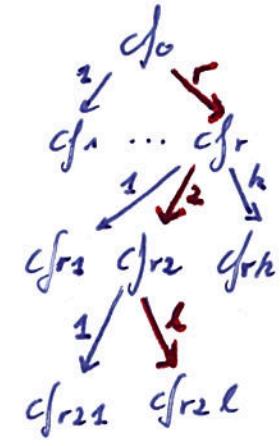
Kuroda showed that ¬(2) ⇒ ¬(1).
This did not help much:
Immerman & Szelepcsényi proved (2) to hold!
We will see the proof in the next lecture.

Goal: Show that for TM, NTMs and DTMs accept the same languages.

Theorem:
L is accepted by an NTM iff L is accepted by a DTM.

Proof:
„<=" ✓ Every DTM is an NTM.
„=>" The idea is to traverse the computation tree of the NTM in breadth-first manner.

- Let $L = L(M_1)$ with M_1 an NTM.
For any control state and any tape symbol, there is a finite number of choices for the next move of M_1.
Let $r \in \mathbb{N}$ be the maximal number of choices for any pair of control state and tape symbol.
Then any finite sequence of choices can be represented by a sequence of digits from 1 to r.
Not all such sequences may represent choices of moves, there may be fewer than r choices in some situations.



The path is given by the sequence r.2.l

- We simulate M_1 by a DTM M_2 that has three tapes.
The three tapes can be compressed into one tape by taking letters to be triples $\binom{a}{b}{c}$ or by writing the tapes one after the other.
// We will discuss such techniques when we come to computability.
The first tape of M_2 holds the input.
On the second tape, M_2 generates sequences of the digits 1 to r. in a systematic manner that implements a breadth-first search:
↳ The sequences are generated in increasing length, starting from the shortest.
↳ Sequences of equal length are generated in numerical order.

- For each sequence generated on tape 2,
  M_2 copies the input onto tape 3
  and then simulates M_1 on tape 3 —
  using the sequence on tape 2 to <u>resolve non-deterministic choices</u>.
  If there is a sequence of choices leading to acceptance of M_1,
  it will eventually be generated on tape 2.
  Then M_2 then simulates M_1, it will accept.
  If no sequence of choices of M_1 leads to acceptance,
  M_2 will not accept. □

<u>Note</u>: The construction does not carry over to LBAs
as it is not clear that it preserves linear boundedness
(the sequence of choices may be long).

15. Immerman & Szelepcsényi's Theorem

Goal: Answer Kuroda's second question from 1964 (see last lecture) positively:

The context-sensitive languages are closed under complement!

History:
- The result was obtained independently 1988 and 1987 by
  - ↪ Neil Immerman (big fish already then, Univ. of Massachusetts Amherst)
  - ↪ Robert Szelepcsényi (student in Bratislava, Slovakia).
- Both received the Gödel-Prize 1995.
- The result brought the method of inductive counting to complexity theory.

Theorem (Immerman & Szelepcsényi '88/'87):

If $L \subseteq \Sigma^*$ is context-sensitive, so is $\overline{L}$.

Goal: Let $L = L(G)$ with $G = (N, \Sigma, P, S)$ a type-1 grammar.

We construct an NLBA that, upon input $w \in \Sigma^*$, accepts if there is **no** derivation $S \Rightarrow_G^* w$

To this end, we consider $\text{Graph}_{|w|}$, the graph of all sentential forms of length $\leq |w|$ together with the derivation relation.

Formally, $\text{Graph}_{|w|} := \big(\underbrace{(\Sigma \cup N)^{\leq |w|}}_{\text{nodes}}, \underbrace{\{(\alpha, \beta) \mid \alpha \Rightarrow_G \beta\}}_{\text{edges}} \big)$.

There is no derivation $S \Rightarrow_G^* w$ iff there is no path from S to w in $\text{Graph}_{|w|}$.

Hence, the key of the proof is to show that non-reachability in a graph of exponential size can be solved non-deterministically with linear space (we discussed that we can be slightly more liberal than only using the space given by the input).

Summary: • We have to develop an algorithm (a TM) that
  • given a graph G (here $Graph_{|w|}$)
    and two nodes s (here S) and t (here w)
  • shows the <u>absence</u> of a path from s to t.
• Moreover, the algorithm should only need space (tape)
  logarithmic in the size of G.
  In our setting, being logarithmic in the size of $G = Graph_{|w|}$
  means being linear in $|w|$ (see Box *)
  Hence, the algorithm is not only a TM but indeed an NLBTM.

Idea: • To check that t is not reachable from s in G,
  enumerate all nodes that <u>are</u> reachable from s
  and check that t is not among them.
• Sounds too easy!
  How to enumerate all nodes in logarithmic space?
  Ask differently!
  How to <u>ensure</u> that all nodes reachable from s were enumerated?
  Clever idea: <u>Counting</u>!

* $Graph_{|w|}$ has $c^{|w|}$ nodes and at most $(c^{|w|})^2 = c^{2|w|}$ edges, where $c = |V| + |\Sigma| + 1$.
Then $\log c^{2|w|} = 2|w| \log c$.

In detail: • <u>Assume</u> we are given N = number of nodes reachable from s.
  We show how to compute N non-deterministically
  in logarithmic (in $|G|$) space below.

• Given N, the following <u>non-deterministic</u> algorithm
  ↪ checks that t is <u>not</u> reachable from s
    in $G = (V, \to)$ with $|V| = n$
  ↪ works in space $O(\log n)$.

```
bool unreach(G,s,t)
begin
  // Given N = number of nodes reachable from s
  count := 0;
  for every node v do
    "make a non-deterministic guess
                whether v is reachable from s";
    if guess = true then
      "non-deterministically try to guess a path
                from s to v of length ≤ n";   // Easy in non-deterministic logarithmic space: Just store last node and steps.
      if "guessed path does not lead to v" then   // Wrong path or wrong guess
        return false;
      else if v = t then   // Reachable
        return false;
      else   // Another reachable node ≠ t found
        count++;
      end if
    end if
  end for
  if count < N then   // Guessed incorrectly about reachability for some v.
    return false;
  else   // Unreachable
    return true;
  end if
end
```

Algorithm unreach runs in (non-deterministic) logarithmic space.
Indeed, N and count can be at most n.
So they can be written down in binary at length log n.

Lemma (Correctness):

Algorithm unreach(G, s, t) has a computation that returns true
iff t is not reachable from s in G.

Proof:
The algorithm makes sure it enumerates all nodes reachable from s
by comparing count to N.
The algorithm accepts iff t was not one of the N nodes
reachable from s. □

It remains to compute
N = number of nodes reachable from s.

The key idea, nowadays called method of inductive counting,
is to inductively compute the values
R(i) := number of nodes reachable from s in ≤ i steps.
Then N = R(n) (we do not have to repeat a node to solve reachability).

```
false or N  numberReach(G, s)
begin
  R(0) := 1                          // s is reachable from s in 0 steps
  for i = 1, ..., n do
    R(i) := 0;                       // initialize R(i)
(*) for every node v do
      // Try all nodes u reachable from s in ≤ i-1 steps
      // Check if v is reachable from such a u in ≤ 1 step
      count := 0;
```

```
for every node u do
    " make a non-deterministic guess
        whether u is reachable from s in ≤ i-1 steps "
    if guess = true then
        " non-deterministically try to guess a path
            from s to u of length ≤ i-1 "
        if "guessed path does not lead to u" then
            return false;
        else
            count++;        // If u is reachable, count it in.
            if u = v or u → v then
                R(i)++;
                goto (*);   // Go to next iteration
                            of "for v" loop
            end if
        end if
    end if
end for                     // Loop for u
if count < R(i-1) then      // Guessed incorrectly
    return false;           about reachability
end if                      for some u.
end for                     // Loop for v
end for                     // Loop for i
return R(n);
end
```

Remark:

At any point in time, Algorithm numberReach(-) needs to remember only two successive values $R(i-1)$ and $R(i)$.
So it can reuse space when computing $R(1), \ldots, R(n)$,
and can be made run with logarithmic space.

Lemma:

numberReach(G,s) computes the number of nodes reachable from s in G.

Proof:

We proceed by induction on i and show that
upon termination of the iteration for i:

$R(i)$ = number of nodes reachable from s in $\leq i$ steps.

Base case ($i=0$): $R(0)=1$ is correct.

Induction step ($i \rightsquigarrow i+1$): Assume the equality holds for $R(i)$
and consider $R(i+1)$.
The algorithm increments $R(i+1)$ on v
iff v is reachable from s in $\leq i+1$ steps.

To see this, note that $R(i+1)$ is not incremented
only if all nodes at distance $\leq i$ from s were tried,
and v is not reachable in ≤ 1 step from any of them.
We are sure to check all nodes at distance $\leq i$
by comparing count to $R(i)$. □

Summary:

- To check that t is not reachable from s in G,
  we first run algorithm numberReach(G, s) to compute N.
  Then we run unreach(G, s, d) with that N.
- Since both (non-deterministic) algorithms
  run in logarithmic space (tape),
  the total space required by the procedure is $O(\log |G|)$.
- We argued on Pages 1 and 2 that this entails
  the existence of an NLBM for $\overline{L}$. □