# Lecture Computability and Logic

Markus Lohrey

University of Siegen

Wintersemester 2025/2026

# Lecture Organization

Under `https://www.eti.uni-siegen.de/ti/lehre/ws2526/bul/index.html?lang=de` you can find updated lecture slides, exercise sheets, announcements, etc.

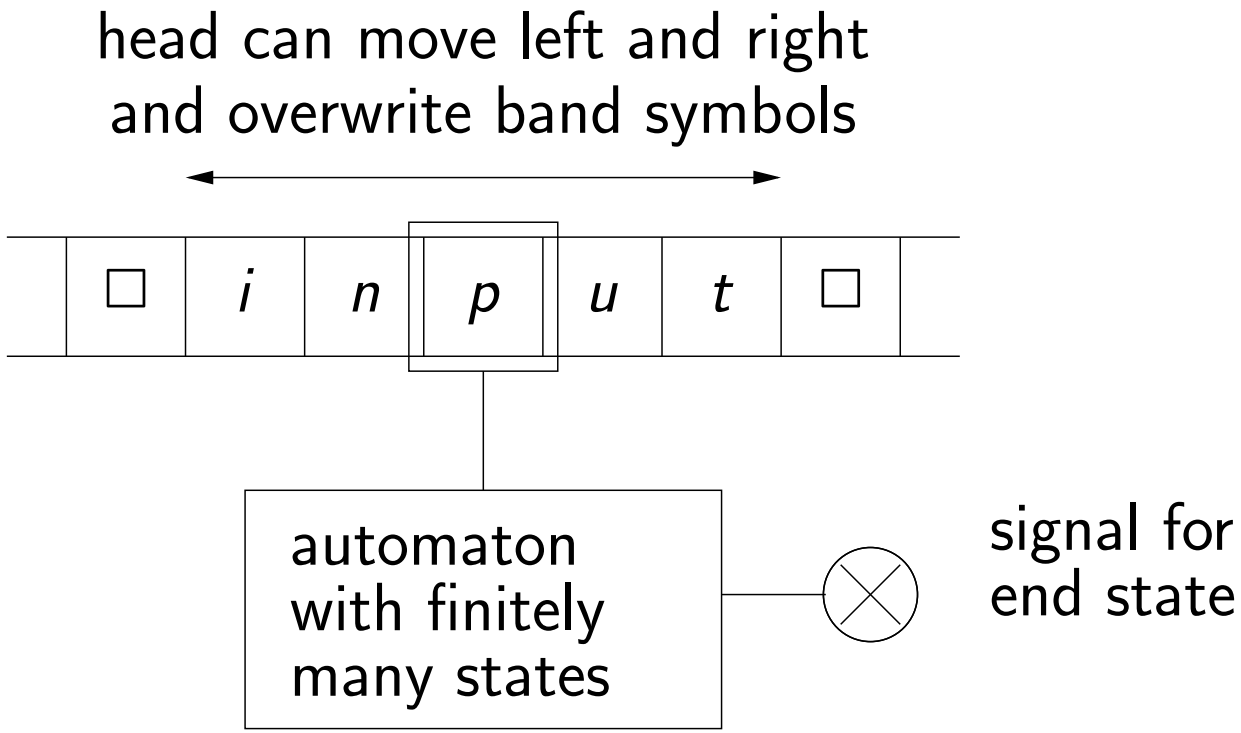
**Recommended Literature:**

- Uwe Schöning, Theoretische Informatik – kurz gefasst, Spektrum Akademischer Verlag (5th Edition): The section on computability closely follows this book in content.
- Uwe Schöning, Logic for Computer Scientists, Springer: The section on logic closely follows this book in content.
- Alexander Asteroth, Christel Baier, Theoretische Informatik, Pearson Studium: This book is structured somewhat differently from the lecture, but still serves as a very good supplement for the lecture parts on computability and propositional logic.

Alexander Thumm organizes the **exercises**.

# Computability: what we need from FSA

- In the lecture *Formal Languages and Automata*, we learned about the model of Turing machines at the end.
- Both deterministic and non-deterministic Turing machines accept exactly the Chomsky Type-0 languages, see FSA, slides 348 and 352.



# Computability: Motivation

After answering the question of which languages are machine-acceptable, we address the question of which functions are computable.

We consider the following types of functions:

- (multi-argument) functions on natural numbers (including zero):

$$f : \mathbb{N}^k \to \mathbb{N}$$

- Functions on words:

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

We also allow partial functions, which are not necessarily defined everywhere.

Formally, a partial function $f : A \to B$ can be defined as a function $f : A \to B \cup \{\bot\}$, where $\bot \notin B$ is a special element, and $f(a) = \bot$ means that $f$ is undefined at $a$.

# Computability: Motivation

## Intuitive Concept of Computability

A partial function $f : \mathbb{N}^k \to \mathbb{N}$ should be considered computable if there exists a computational procedure/an algorithm/a program that computes $f$, i.e., for an input $(n_1, \ldots, n_k)$ the program behaves as follows:

- If $f(n_1, \ldots, n_k)$ is defined, the program terminates after finitely many steps and outputs $f(n_1, \ldots, n_k)$.
- If the function is not defined for $(n_1, \ldots, n_k)$, the program should not stop (e.g., by entering an infinite loop).

# Computability: Motivation

The equivalence of many computational models (this will be shown later) and the intuitive understanding of the concept of computability lead to the following (unprovable) thesis.

## Church's Thesis

The class of functions captured by the formal definition of Turing machine computability (equivalent: WHILE-computability, GOTO-computability, $\mu$-recursiveness) coincides exactly with the class of functions computable in the intuitive sense.

**Remarks:** A computational model that is equivalent to Turing machines is also called Turing-complete. The corresponding concept of computability is called Turing-computability.

Almost all programming languages are Turing-complete.

# Computability: Motivation

## Non-computable Functions

There are functions of the form $f : \mathbb{N} \to \mathbb{N}$ that are not computable.

**Idea of the proof:** We choose an arbitrary computational model and impose only one requirement:

> Programs or machines in this computational model can be encoded as words over a finite alphabet.

Then it holds: there are at most countably many machines/programs.

# Computability: Motivation

**But:** there are uncountably many (total) functions.

We show this by contradiction: assume the set of all functions $\mathcal{F}$ on natural numbers is countable. This means there is a surjective mapping $F : \mathbb{N} \to \mathcal{F}$.

We construct the function $g : \mathbb{N} \to \mathbb{N}$ with

$$g(n) = f_n(n) + 1, \qquad \text{where } f_n = F(n).$$

Since $F$ is surjective, there must be a natural number $i$ such that $F(i) = g$. For this $i$, it follows: $g(i) = f_i(i)$. But this is a contradiction to the definition of $g$ with $g(i) = f_i(i) + 1$.

# Computability: Motivation

**Illustration:** we represent $F$ by writing for each $n$ the function $f_n$ as the sequence $f_n(0)$, $f_n(1)$, $f_n(2)$, ....

For example:

| $n$ | $f_n(0)$ | $f_n(1)$ | $f_n(2)$ | $f_n(3)$ | $f_n(4)$ | ... |
|---|---|---|---|---|---|---|
| 0 | 7 | 20 | 33 | 0 | 12 | |
| 1 | 12 | 33 | 94 | 2 | 17 | |
| 2 | 99 | 101 | 16 | 11 | 22 | |
| 3 | 2 | 0 | 14 | 99 | 42 | |
| 4 | 17 | 5 | 77 | 7 | 11 | |
| ... | | | | | | |

# Computability: Motivation

**Illustration:** we represent $F$ by writing for each $n$ the function $f_n$ as the sequence $f_n(0)$, $f_n(1)$, $f_n(2)$, ....

Use all the numbers on the diagonal ...

| $n$ | $f_n(0)$ | $f_n(1)$ | $f_n(2)$ | $f_n(3)$ | $f_n(4)$ | ... |
|---|---|---|---|---|---|---|
| 0 | 7 | 20 | 33 | 0 | 12 | |
| 1 | 12 | 33 | 94 | 2 | 17 | |
| 2 | 99 | 101 | 16 | 11 | 22 | |
| 3 | 2 | 0 | 14 | 99 | 42 | |
| 4 | 17 | 5 | 77 | 7 | 11 | |
| ... | | | | | | |

# Computability: Motivation

**Illustration:** we represent $F$ by writing for each $n$ the function $f_n$ as the sequence $f_n(0)$, $f_n(1)$, $f_n(2)$, ....

... and increment them by one. This gives us $g$.

| $n$ | $f_n(0)$ | $f_n(1)$ | $f_n(2)$ | $f_n(3)$ | $f_n(4)$ | ... |
|---|---|---|---|---|---|---|
| 0 | 8 | 20 | 33 | 0 | 12 | |
| 1 | 12 | 34 | 94 | 2 | 17 | |
| 2 | 99 | 101 | 17 | 11 | 22 | |
| 3 | 2 | 0 | 14 | 100 | 42 | |
| 4 | 17 | 5 | 77 | 7 | 12 | |
| ... | | | | | | |

# Computability: Motivation

**Illustration:** we represent $F$ by writing for each $n$ the function $f_n$ as the sequence $f_n(0)$, $f_n(1)$, $f_n(2)$, ....

However, the function $g$ cannot match any of the other functions due to this construction.

| $n$ | $f_n(0)$ | $f_n(1)$ | $f_n(2)$ | $f_n(3)$ | $f_n(4)$ | ... |
|---|---|---|---|---|---|---|
| 0 | 8 | 20 | 33 | 0 | 12 | |
| 1 | 12 | 34 | 94 | 2 | 17 | |
| 2 | 99 | 101 | 17 | 11 | 22 | |
| 3 | 2 | 0 | 14 | 100 | 42 | |
| 4 | 17 | 5 | 77 | 7 | 12 | |
| ... | | | | | | |

# Computability: Motivation

**Illustration:** we represent $F$ by writing for each $n$ the function $f_n$ as the sequence $f_n(0)$, $f_n(1)$, $f_n(2)$, ....

However, the function $g$ cannot match any of the other functions due to this construction.

| $n$ | $f_n(0)$ | $f_n(1)$ | $f_n(2)$ | $f_n(3)$ | $f_n(4)$ | ... |
|---|---|---|---|---|---|---|
| 0 | 8 | 20 | 33 | 0 | 12 | |
| 1 | 12 | 34 | 94 | 2 | 17 | |
| 2 | 99 | 101 | 17 | 11 | 22 | |
| 3 | 2 | 0 | 14 | 100 | 42 | |
| 4 | 17 | 5 | 77 | 7 | 12 | |
| ... | | | | | | |

This type of "self-referential" proof is often called a diagonalization proof because of its representation through such diagrams.

# Turing-computability

After discussing the intuitive concept of computability, we now turn to the formal concept of computability, first based on Turing machines.

We already know what it means for a Turing machine to accept a language. Now we define what it means for a Turing machine to compute a function.

For a number $n \in \mathbb{N}$, let $bin(n)$ be the binary representation of $n$:

- $bin(n) \in 1\{0,1\}^* \cup \{0\}$
- If $bin(n) = b_k b_{k-1} \cdots b_0$, then $n = \sum_{i=0}^{k} b_i 2^i$.

**Example:** $bin(5) = 101$, $bin(6) = 110$, $bin(7) = 111$, $bin(8) = 1000$.

# Turing-computability

## Turing-computable Functions on Natural Numbers

A partial function $f : \mathbb{N}^k \to \mathbb{N}$ is Turing-computable if there exists a deterministic Turing machine $M = (Z, \{0, 1, \#\}, \Gamma, \delta, z_0, \Box, E)$ such that for all $n_1, \ldots, n_k \in \mathbb{N}$:

- If $f(n_1, \ldots, n_k)$ is undefined, then $M$ does not halt on the initial configuration $z_0 bin(n_1)\#bin(n_2)\# \ldots bin(n_k)\#$, i.e., there is no configuration $c \in \Gamma^* E \Gamma^+$ such that

$$z_0 bin(n_1)\#bin(n_2)\# \ldots bin(n_k)\# \vdash_M^* c.$$

- If $f(n_1, \ldots, n_k)$ is defined, and $f(n_1, \ldots, n_k) = m$, then there exists a halting state $z_e \in E$ with

$$z_0 bin(n_1)\#bin(n_2)\# \ldots bin(n_k)\# \vdash_M^* \Box \cdots \Box z_e bin(m) \Box \cdots \Box.$$

# Turing-computability

**Intuition:**

- If you write the numbers $n_1, \ldots, n_k$ in binary representation – separated by $\#$ – onto the tape, then $M$ halts if and only if $f(n_1, \ldots, n_k)$ is defined.
- If $f(n_1, \ldots, n_k) = m$, then $M$ computes the number $f(n_1, \ldots, n_k)$ (possibly surrounded by spaces) from $n_1, \ldots, n_k$ and transitions to a halting state.

# Turing-computability

## Turing-computable Functions on Words

A partial function $f : \Sigma^* \to \Sigma^*$ is Turing-computable if there exists a deterministic Turing machine $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ such that for all $x \in \Sigma^*$:

- If $f(x)$ is undefined, then $M$ does not halt on the initial configuration $z_0 x \Box$, i.e., there is no $c \in \Gamma^* E \Gamma^+$ with $z_0 x \Box \vdash_M^* c$.
- If $f(x)$ is defined, and $f(x) = y$, then there exists a halting state $z_e \in E$ with $z_0 x \Box \vdash_M^* \Box \cdots \Box z_e y \Box \cdots \Box$.

**Intuition:**

- If you write the word $x$ onto the tape, $M$ halts if and only if $f(x)$ is defined.
- If $f(x) = y$, then $M$ computes the word $y$ from $x$ (possibly surrounded by spaces) and transitions to a halting state.

# Turing-computability

Examples of Turing-computable functions:

**Example 1:** The successor function $n \mapsto n + 1$ is Turing-computable (see FSA, Slide 324 and 325).

**Example 2:** Let $\Omega$ be the everywhere undefined function. This is also Turing-computable, for example, by a Turing machine that has no halting state. For instance, by a Turing machine with the transition rule

$$\delta(z_0, a) = (z_0, a, N) \qquad \text{for all } a \in \Gamma.$$

**Example 3:** Given a Type-0 language $L \subseteq \Sigma^*$, we consider the so-called "half" characteristic function of $L$:

$$\begin{aligned} \chi_L \colon \Sigma^* &\to \{1\} \\ \chi_L(w) &= \begin{cases} 1 & \text{if } w \in L \\ \text{undefined} & \text{otherwise} \end{cases} \end{aligned}$$

# Turing-computability

Idea for a Turing machine $M$ that computes $\chi_L$:

- We use the transformation "Grammar $\rightarrow$ Turing machine" (FSA, Slide 354), and obtain a Turing machine $M'$ with $T(M') = L$.
- The machine $M'$ is non-deterministic. Due to the equivalence of deterministic and non-deterministic Turing machines (FSA, Slide 358) one can convert $M'$ into a deterministic Turing machine $M''$ with $T(M'') = T(M') = L$.
- From $M''$ we can easily obtain a deterministic Turing machine $M$, which behaves like $M''$ except: If $M''$ is supposed to transition to a halting state (and thus accepts), then $M$ overwrites the entire tape with 1 and transitions to a halting state.
- Note: If the input $x$ does not belong to $L$, $M$ will not halt.

# Multi-tape Turing Machine

We now introduce several new computational models and show that they are all equivalent to Turing machines. The first one is the so-called multi-tape Turing machine.

## Multi-tape Turing Machine

- A multi-tape Turing machine has $k$ ($k \geq 1$) tapes with $k$ independent heads, but only one state.
- The transition function has the form

$$\delta\colon (Z \setminus E) \times \Gamma^k \to Z \times \Gamma^k \times \{L, R, N\}^k$$

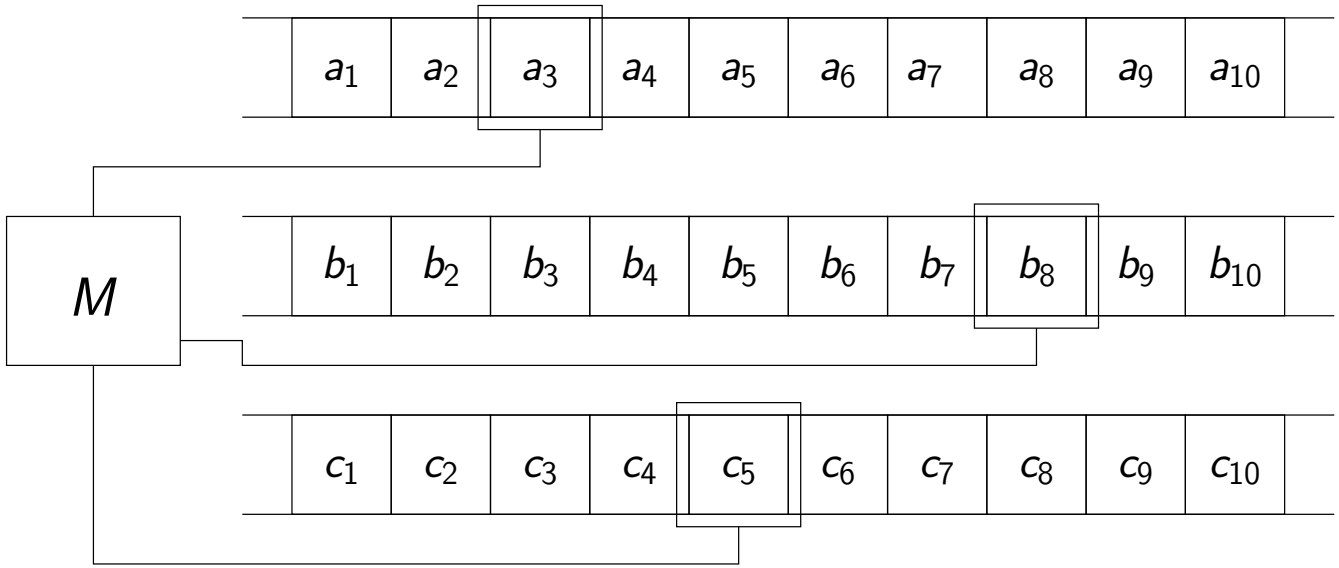
  (one state, $k$ tape symbols, $k$ movements).
- The input and output are each on the first tape. Initially, the remaining tapes are empty.

# Multi-tape Turing Machine

**Theorem 1 (Multi-tape Turing Machines $\rightarrow$ Turing Machines)**

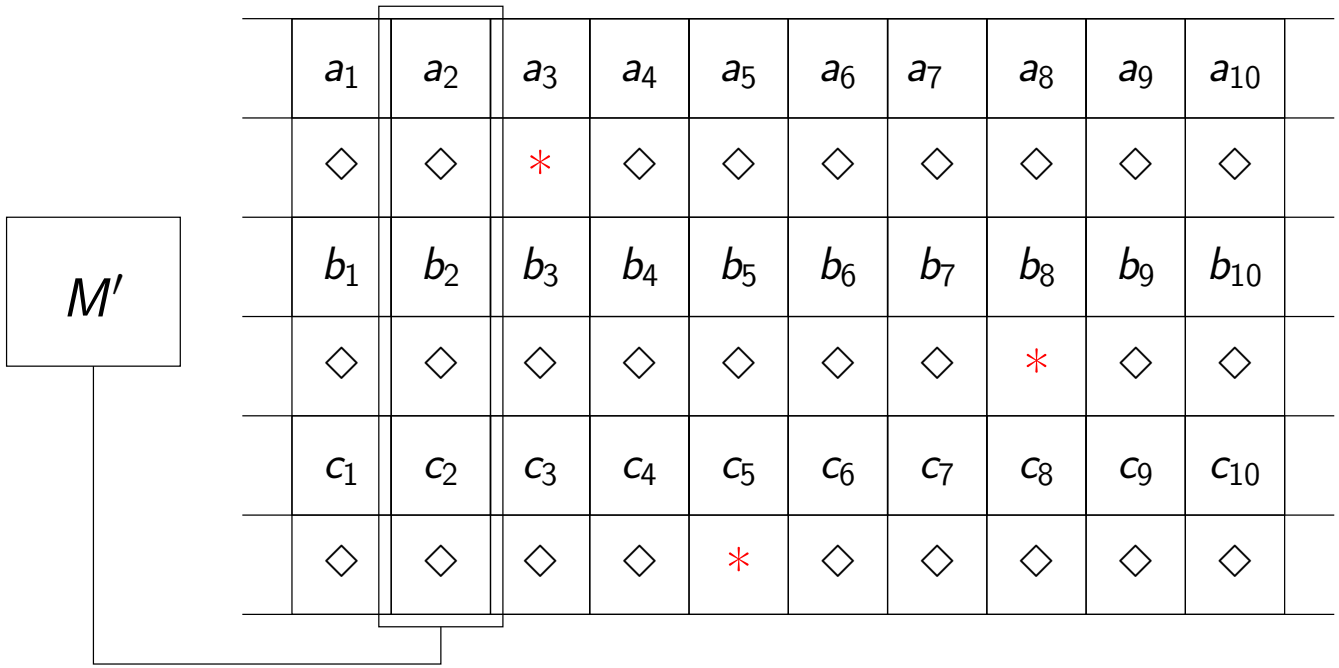
For every multi-tape Turing machine $M$, there exists a (single-tape) Turing machine $M'$ that accepts the same language or computes the same function.

**Proof Idea:** We start with the representation of a typical configuration of a multi-tape Turing machine.



# Multi-tape Turing Machine

Simulation using a single-tape Turing machine by extending the alphabet: We combine the overlapping tape entries into a field. The symbols $*$ and $\diamond$ are used to encode where the heads of the multi-tape Turing machine are located.



# Multi-tape Turing Machine

Let $k$ be the number of tapes of $M$, $\Gamma$ its tape alphabet, and $\Sigma$ the input alphabet.

Tape alphabet of $M'$: $\Gamma' = \Sigma \cup (\Gamma \times \{*, \diamond\})^k$

- Meaning of a tape symbol from $(\Gamma \times \{*, \diamond\})^k$ using the example of $(a, \diamond, b, *, c, \diamond)$, i.e., $k = 3$:

  From each of the 3 tapes of $M$, $M'$ (the single-tape Turing machine) is currently scanning one cell. In the cell scanned from tape 1 (or 2, 3) of $M$, there is currently $a$ (or $b$, $c$).

  The $M$-head of tape 2 is currently positioned on the cell of tape 2 scanned by $M'$ (indicated by $*$).

  The $M$-heads of tapes 1 and 3, however, are on cells that $M'$ is not currently scanning (indicated by the two $\diamond$).

# Multi-tape Turing Machine

- Since $M'$ and $M$ have the same input alphabet, $\Sigma$ must be part of $\Gamma'$. When $M'$ receives the input $w \in \Sigma^*$, it first goes completely over $w$ in a single pass, converting $w$ into the above "multi-tape encoding."

**Problem:** A single-tape Turing machine has only one head, and it can only be in one position $\rightsquigarrow$ Simulation of a multi-tape Turing machine transition in multiple steps.

**Simulation of a transition of the multi-tape Turing machine:**

- At the beginning of simulating a step, the head of the single-tape Turing machine $M'$ is to the left of all $*$-marks.
- Then the head moves to the right, crossing all $k$ $*$-marks and recording the respective actions of the $\delta$ function. (This requires many states.)
- Then the head moves back to the left and performs all the necessary changes. □

# Loop-, While-, Goto-Computability

We now consider another computational model, which essentially represents a simple programming language with various constructs.

- These programs have variables that are assigned natural numbers. These variables may be assigned arithmetic expressions (with constants, variables, and operators $+$, $-$).
- Additionally, the programs include various loop constructs.

# Loop-, While-, Goto-Computability

In particular, we consider the following types of programs:

## Loop Programs

Contain only Loop loops and for loops, where the number of iterations is determined at the outset.

## While Programs

Contain only While loops with a condition that needs to be repeatedly evaluated.

## Goto Programs

Contain Gotos (unconditional jumps) and if-then-else statements.

We are mainly interested in the functions computed by such programs.

# LOOP Programs

## Syntactic Components for LOOP Programs

- Variables: $x_1, x_2, x_3, \ldots$
- Constants: $0, 1, 2, \ldots$
- Delimiters: ; and $:=$
- Operator Symbols: $+$ and $-$
- Keywords: LOOP, DO, END

## Inductive Syntax Definition of LOOP Programs

A LOOP program is either of the form

- $x_i := x_j + c$ or $x_i := x_j - c$ with $c \in \mathbb{N}$ and $i, j \geq 1$ (Assignment) or
- $P_1; P_2$, where $P_1$ and $P_2$ are already LOOP programs (Sequential Composition) or
- LOOP $x_i$ DO $P$ END, where $P$ is a LOOP program and $i \geq 1$.

# LOOP Programs

## Informal Description of Semantics

- A LOOP program that is supposed to compute a $k$-ary function $f : \mathbb{N}^k \to \mathbb{N}$ starts with the input values $n_1, \ldots, n_k$ in the variables $x_1, \ldots, x_k$. All other variables start with the value 0. The result $f(n_1, \ldots, n_k)$ will be in $x_1$ upon termination.
- Interpretation of assignments:
  - $x_i := x_j + c$ – as usual
  - $x_i := x_j - c$ – modified subtraction; if $c > x_j$, the result is 0
- Sequential composition $P_1; P_2$: first execute $P_1$, then execute $P_2$.
- LOOP $x_i$ DO $P$ END: the program $P$ is executed as many times as the variable $x_i$ initially specifies.

# LOOP Programs

**Note:** If in a program LOOP $x_i$ DO $P$ END the value of $x_i$ is changed within $P$, this has no effect on the number of executions of the loop body $P$.

If $x_i$ has the value $n$ before $P$ is executed for the first time, then $P$ is executed exactly $n$ times.

**Example:** $x_1 := x_1 + 3;$ LOOP $x_1$ DO $x_1 := x_1 + 1$ END

Suppose $x_1$ initially has the value 0.

Then the loop body will be executed 3 times.

Therefore, at the end $x_1$ will have the value 6.

# LOOP Programs

## Formal Description of Semantics

For every LOOP program $P$ in which no variable $x_i$ with $i > k$ occurs (i.e., only variables $x_1, \ldots, x_k$ are allowed in $P$), we first define a function

$$[P]_k : \mathbb{N}^k \to \mathbb{N}^k$$

as follows, by induction on the structure of $P$:

- $[x_i := x_j + c]_k(n_1, \ldots, n_k) = (m_1, \ldots, m_k)$ if and only if (i) $m_\ell = n_\ell$ for $\ell \neq i$ and (ii) $m_i = n_j + c$.
- $[x_i := x_j - c]_k(n_1, \ldots, n_k) = (m_1, \ldots, m_k)$ if and only if (i) $m_\ell = n_\ell$ for $\ell \neq i$ and (ii) $m_i = \max\{0, n_j - c\}$.
- $[P_1; P_2]_k(n_1, \ldots, n_k) = [P_2]_k([P_1]_k(n_1, \ldots, n_k))$
- $[\text{LOOP } x_i \text{ DO } P \text{ END}]_k(n_1, \ldots, n_k) = [P]_k^{n_i}(n_1, \ldots, n_k)$

**Intuition:** $[P]_k(n_1, \ldots, n_k)$ is computed as follows:

- Initially, set each variable $x_i$ with $1 \leq i \leq k$ to the value $n_i$.

  The initial values of variables $x_i$ with $i > k$ do not matter, as such variables do not occur in $P$.

- Now execute the program $P$.
- Suppose after executing $P$, the variable $x_i$ $(1 \leq i \leq k)$ has the value $m_i$.

  Then $[P]_k(n_1, \ldots, n_k) = (m_1, \ldots, m_k)$.

# LOOP Programs

Let $\pi_i(n_1, \ldots, n_k) = n_i$ (projection onto the $i$-th component).

## LOOP Computability (Definition)

A function $f : \mathbb{N}^k \to \mathbb{N}$ is called LOOP-computable if there exists an $\ell \geq k$ and a LOOP program $P$, using only the variables $x_1, \ldots, x_\ell$, such that:

$$\forall n_1, \ldots, n_k \in \mathbb{N} : f(n_1, \ldots, n_k) = \pi_1([P]_\ell(n_1, \ldots, n_k, \underbrace{0, \ldots, 0}_{\ell - k \text{ many}})).$$

To compute $f(n_1, \ldots, n_k)$, the variables $x_1, \ldots, x_k$ are initially set to the input values $n_1, \ldots, n_k$.

The auxiliary variables $x_{k+1}, \ldots, x_\ell$ are initialized to 0.

Then $P$ is executed.

After the execution of $P$, the value of variable $x_1$ is the function value $f(n_1, \ldots, n_k)$.

# LOOP Programs

**Remarks:**

- All LOOP programs halt after a finite amount of time (LOOP loops always terminate).
- Therefore, all LOOP-computable functions are total (i.e., defined everywhere).
- Hence, there are Turing-computable functions that are not LOOP-computable (e.g., the everywhere undefined function $\Omega$ from Slide 14).
- We will see later that there are even total Turing-computable functions that are not LOOP-computable.

# Loop Programs

Loop programs can simulate certain programming constructs that are not included in their syntax.

## If-Then

Simulation of If $x_1 = 0$ Then $A$ End

$x_2 := 1;$ Loop $x_1$ Do $x_2 := 0$ End; Loop $x_2$ Do $A$ End

## Addition

Simulation of $x_i := x_j + x_k$ (where $i \neq k$)

$x_i := x_j;$ Loop $x_k$ Do $x_i := x_i + 1$ End

## Multiplication

Simulation of $x_i := x_j \cdot x_k$ (where $k \neq i \neq j$)

$x_i := 0;$ Loop $x_k$ Do $x_i := x_i + x_j$ End

# LOOP Programs

We will also use such constructs in While programs. We will assume that they are suitably simulated as above.

**Analogously:** Integer division ($x$ DIV $y$) and remainder ($x$ MOD $y$).

# WHILE-Programs

We now extend the syntax of LOOP-Programs to WHILE-Programs by allowing an additional type of loop construct besides LOOP loops.

## Syntax of WHILE-Programs

If $P$ is a WHILE-Program and $i \geq 1$, then

WHILE $x_i \neq 0$ DO $P$ END

is also a WHILE-Program.

All constructs allowed in LOOP-Programs (see Slide 23) are also permitted in WHILE-Programs.

**Intuition:** Program $P$ is executed as long as the value of $x_i$ is not equal to 0.

## Semantics of WHILE Programs

As with LOOP programs, we first define, for each WHILE program $P$ that contains no variable $x_i$ with $i > k$, a (partial) function $[P]_k : \mathbb{N}^k \to \mathbb{N}^k$ inductively.

For the constructs available in LOOP programs, we adopt the definitions from Slide 26.

Let $P = \text{WHILE } x_i \neq 0 \text{ DO } A \text{ END}$ $(i \leq k)$ and $(n_1, \ldots, n_k) \in \mathbb{N}^k$.

If there exists a number $\tau$ such that $\pi_i([A]_k^\tau(n_1, \ldots, n_k)) = 0$, let $t$ be the smallest number with this property. Otherwise, let $t$ be undefined.

Then define

$$[P]_k(n_1, \ldots, n_k) = \begin{cases} [A]_k^t(n_1, \ldots, n_k) & \text{if } t \text{ is defined} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

# While Programs

**Explanation:** $[A]_k^\tau$ denotes the $\tau$-fold composition of $[A]$:

$$[A]_k^\tau(n_1, \ldots, n_k) = \underbrace{[A]_k([A]_k([A]_k(\cdots [A]_k}_{\tau \text{ times}}(n_1, \ldots, n_k)\cdots)))$$

Thus, $t$ is the smallest number $\tau$ such that after $\tau$ executions of $A$ (starting with the initial values $n_1, \ldots, n_k$ for the variables $x_1, \ldots, x_k$), the variable $x_i$ has the value 0.

If $x_i$ never takes the value 0, the While loop does not terminate, and $[P]_k(n_1, \ldots, n_k)$ is undefined.

**Note:** A Loop loop Loop $x$ Do $P$ End can be simulated by

$y := x;$
While $y \neq 0$ Do $y := y - 1; P$ End

**Important:** $y$ is a new variable that does not appear in $P$.

## WHILE Computability (Definition)

A partial function $f : \mathbb{N}^k \to \mathbb{N}$ is called WHILE-computable if there exists an $\ell \geq k$ and a WHILE program $P$, using only the variables $x_1, \ldots, x_\ell$, such that for all $n_1, \ldots, n_k \in \mathbb{N}$:

- $f(n_1, \ldots, n_k)$ is defined $\iff$ $[P]_\ell(n_1, \ldots, n_k, \underbrace{0, \ldots, 0}_{\ell - k \text{ many}})$ is defined
- If $f(n_1, \ldots, n_k)$ is defined, then $f(n_1, \ldots, n_k) = \pi_1([P]_\ell(n_1, \ldots, n_k, \underbrace{0, \ldots, 0}_{\ell - k \text{ many}}))$.

# WHILE Programs

**Theorem 2 (WHILE-Programs $\rightarrow$ Turing Machines)**

Every WHILE-computable function is also Turing-computable.

In other words: Turing machines can simulate WHILE programs.

**Proof Idea:**

- We use a multi-tape Turing machine, where each tape stores a different variable of the WHILE program in binary representation.
  $k$ variables $\rightsquigarrow$ $k$ tapes
- $x_i := x_j + c$ can be performed by the Turing machine by executing the increment function $(+1)$ $c$ times.
- $x_i := x_j - c$ works similarly.

# While Programs

- Sequential Composition $P_1; P_2$:

  We inductively determine Turing machines $M_1$, $M_2$ for $P_1$, $P_2$.

  We then construct a Turing machine for $P_1; P_2$ as follows:

  - Union of the state sets, tape alphabets, and transition functions
  - The initial state is the initial state of $M_1$. The final states are the final states of $M_2$.
  - Instead of transitioning to a final state of $M_1$, a transition to the initial state of $M_2$ is made.

    (Compare with the concatenation construction for finite automata (FSA, Slides 99 and 100).

# While Programs

- While Loop While $x_i \neq 0$ Do $P$ End:

  First, determine a Turing machine $M$ for $P$.

  Modify $M$ as follows:
  In the new initial state, first check if 0 is on the $i$-th tape.
  - If yes: transition to the final state
  - If no: $M$ is executed

  Instead of transitioning to the final state: transition to the new initial state.

# GOTO Programs

## Syntax of GOTO Programs

Possible instructions for GOTO programs:

Assignment: $x_i := x_j + c$ or $x_i := x_j - c$ (with $c \in \mathbb{N}$)

Unconditional Jump: GOTO $M_i$

Conditional Jump: IF $x_i = c$ THEN GOTO $M_i$

Halt Instruction: HALT

A GOTO program consists of a sequence of instructions $A_i$, each preceded by a (jump) label $M_i$.

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

(If labels are not jumped to, we sometimes omit them.)

# Goto Programs

## Intuitive Semantics of Goto Programs

- The instructions of a Goto program are executed sequentially.
- Exception: Goto $M$ jumps to the instruction with label $M$.
- If statements are interpreted as usual.
- Halt instructions terminate Goto programs. (The last instruction of a program should be a Halt or an unconditional jump.)

This is not a truly formal semantics definition. As an exercise, write a formal semantics definition (analogous to While programs).

Like While programs, Goto programs can also enter infinite loops.

# Goto Programs

Goto-computable functions are defined analogously to While-computable functions.

**Example:** a Goto program for computing $x_1 := x_1 + x_2$

$M_1$ : If $x_2 = 0$ Then Goto $M_2$;

$x_1 := x_1 + 1$;

$x_2 := x_2 - 1$;

Goto $M_1$;

$M_2$ : Halt

**Satz 3 (WHILE Programs → GOTO Programs)**

Every WHILE program can be simulated by a GOTO program, i.e., every WHILE-computable function is GOTO-computable.

**Proof:**

A WHILE loop

WHILE $x \neq 0$ DO $P$ END

can be simulated by

$M_1$: IF $x = 0$ THEN GOTO $M_2$;
$P$;
GOTO $M_1$;
$M_2$: ...

□

# GOTO Programs

The less obvious reversal also holds:

**Satz 4 (GOTO Programs → WHILE Programs)**

Every GOTO program can be simulated by a WHILE program, i.e., every GOTO-computable function is WHILE-computable.

This is one of the reasons why modern programming languages generally do not use GOTOS.

**Additional Reasons:** Spaghetti code with the use of GOTOS, see also Edsger W. Dijkstra: "Go To Statement Considered Harmful" (1968), `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF`.

**Proof (Goto Programs $\rightarrow$ While Programs):**

Let $P = (M_1 : A_1; M_2 : A_2; \ldots M_k : A_k)$ be a GOTO program.

We simulate $P$ with the following WHILE program $Q$ that contains only one WHILE loop:

count := 1;
WHILE count $\neq$ 0 DO
    IF count = 1 THEN $A'_1$ END;
    IF count = 2 THEN $A'_2$ END;
        $\vdots$
    IF count = $k$ THEN $A'_k$ END
END

# GOTO Programs

Here, $A'_i$ is the following WHILE program:

$$A'_i = \begin{cases} x_j := x_\ell \pm c;\ \text{count} := \text{count} + 1 & \text{if } A_i = (x_j := x_\ell \pm c) \\ \text{count} := n & \text{if } A_i = (\text{GOTO } M_n) \\ \text{IF } x_j = c \text{ THEN count} := n & \text{if } A_i = (\text{IF } x_j = c \text{ THEN} \\ \quad \text{ELSE count} := \text{count} + 1 \text{ END} & \qquad \text{GOTO } M_n) \\ \text{count} := 0 & \text{if } A_i = \text{HALT} \end{cases}$$

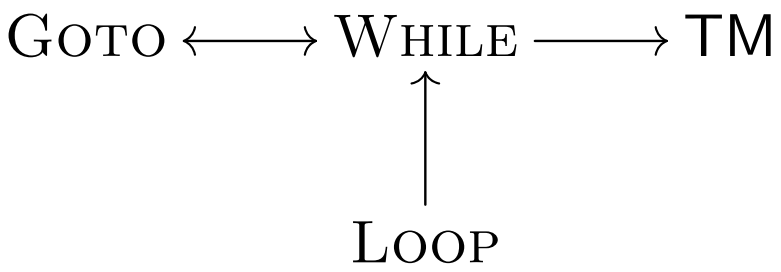
□

The simulation of GOTO programs by WHILE programs uses only one WHILE loop (if IF THEN ELSE is allowed as an elementary construct).

This means: A WHILE program can be transformed into an equivalent WHILE program with a WHILE loop (Kleene Normal Form for WHILE programs) by converting it into a GOTO program and then back to a WHILE program.

# Goto Programs

What transformations have we carried out so far?




To demonstrate the equivalence of Goto, While, and Turing computability, we still need to show the direction

$$\text{TM} \to \text{Goto}$$

# GOTO Programs

**Theorem 5 (TM → GOTO Programs)**

Every Turing machine can be simulated by a GOTO program. That is, every Turing-computable function is GOTO-computable.

**Proof:**

Let $M = (Z, \Sigma, \Gamma, \delta, z_0, E, \square)$ be a deterministic Turing machine that computes a function $f : \mathbb{N}^k \to \mathbb{N}$.

Without loss of generality, assume $\Gamma = \{0, \ldots, m-1\}$, $\square = 0$, and $Z = \{0, \ldots, n-1\}$.

For $a_1 a_2 \cdots a_p \in \Gamma^*$, let

$$(a_1 a_2 \cdots a_p)_m = \sum_{i=1}^{p} a_i \cdot m^{i-1}$$

be the value of $a_1 a_2 \cdots a_p$ in base $m$ (least significant digit on the far left).

# Goto Programs

A configuration $a_1 \cdots a_p \, z \, b_1 \cdots b_q$ is represented by the triple

$$\big((a_p \ldots a_1)_m, z, (b_1 \cdots b_q)_m\big) \in \mathbb{N} \times \{0, \ldots, n-1\} \times \mathbb{N}$$

Such a triple is stored using the three variables $x$, $z$, and $y$ as follows:

- $z$ = current state
- $x$ = encoding of the tape contents to the left of the head
- $y$ = encoding of the tape contents to the right of the head, including the currently read cell

Note: $(a_p \ldots a_1 \square)_m = (a_p \ldots a_1)_m$ and $(b_1 \cdots b_q \square)_m = (b_1 \cdots b_q)_m$, since $\square = 0$ and the highest digit is on the far right.

This is convenient because $a_1 \cdots a_p \, z \, b_1 \cdots b_q$ and $\square a_1 \cdots a_p \, z \, b_1 \cdots b_q \square$ represent the same configuration.

# GOTO Programs

To simulate Turing machine operations, we use the arithmetic operations DIV and MOD (integer division and remainder), always dividing by our base $m$. These operations can be easily implemented using GOTO programs.

**Example:** In base $m = 10$, we have (here we write numbers as usual, i.e., the least significant digit is on the right)

$$\begin{aligned} 5634 \text{ DIV } 10 &= 563 \\ 5634 \text{ MOD } 10 &= 4 \end{aligned}$$

**Simulation of Turing machine operations:**

Head reads symbol: $a := y \text{ MOD } m$

Write symbol $b$ on the tape: $y := (y \text{ DIV } m) \cdot m + b$

Move head left: $y := y \cdot m + (x \text{ MOD } m);\ x := x \text{ DIV } m$

Move head right: $x := x \cdot m + (y \text{ MOD } m);\ y := y \text{ DIV } m$

**Explanation:** Let $a_1 \cdots a_p\, z\, b_1 \cdots b_q$ be the current configuration, and thus

$$x = (a_p \ldots a_1)_m, \quad y = (b_1 \cdots b_q)_m \quad (\text{and } z = z)$$

Then $y \text{ MOD } m = (b_1 + m \cdot (b_2 \cdots b_q)_m) \text{ MOD } m = b_1$.

Therefore, the variable $a$ is set to the currently read symbol.

Furthermore,

$$\begin{aligned}
(y \text{ DIV } m) \cdot m + b &= ((b_1 \cdots b_q)_m \text{ DIV } m) \cdot m + b \\
&= (b_2 \cdots b_q)_m \cdot m + b \\
&= (0 b_2 \cdots b_q)_m + b \\
&= (b b_2 \cdots b_q)_m
\end{aligned}$$

So, the operation $y := (y \text{ DIV } m) \cdot m + b$ indeed writes $b$ into the currently scanned tape cell.

It holds that

$$x \text{ DIV } m = (a_p a_{p-1} \dots a_1)_m \text{ DIV } m = (a_{p-1} \dots a_1)_m$$

and

$$\begin{aligned} y \cdot m + (x \text{ MOD } m) &= (b_1 \cdots b_q)_m \cdot m + ((a_p a_{p-1} \dots a_1)_m \text{ MOD } m) \\ &= (0 b_1 \cdots b_q)_m + a_p \\ &= (a_p b_1 \cdots b_q)_m \end{aligned}$$

Thus, the operation

$$y := y \cdot m + (x \text{ MOD } m);\ x := x \text{ DIV } m$$

correctly implements a leftward movement of the head.

Similarly, we can show that

$$x := x \cdot m + (y \text{ MOD } m);\ y := y \text{ DIV } m$$

correctly implements a rightward movement of the head.

# GOTO Programs

Let $(n_1, \ldots, n_k) \in \mathbb{N}^k$ be an input tuple.

The GOTO program to be created consists of the following three parts:

**Part 1:** Initialization of the variables $x, y, z$ with the initial configuration.

The initial configuration corresponding to the input tuple $(n_1, \ldots, n_k)$ is

$$z_0 \, \mathrm{bin}(n_1) \# \, \mathrm{bin}(n_2) \# \cdots \# \, \mathrm{bin}(n_k) \#$$

So we initialize $x, y, z$ with

$$x := 0; \;\; z = z_0; \;\; y := (\mathrm{bin}(n_1) \# \mathrm{bin}(n_2) \# \cdots \# \mathrm{bin}(n_k) \#)_m.$$

In $\mathrm{bin}(n_i)$, the binary digits 0 and 1 are represented by symbols $a \in \Gamma \setminus \{\Box\}$ and $b \in \Gamma \setminus \{\Box\}$ respectively (e.g., $0 \to 1$, $1 \to 2$). This is necessary because 0 is already reserved for the blank symbol $\Box$.

# GOTO Programs

The number $(\text{bin}(n_1)\#\text{bin}(n_2)\#\cdots\#\text{bin}(n_k)\#)_m$ must first be calculated from the input numbers $n_1, \ldots, n_k$ using a GOTO program.

This is somewhat tedious but not fundamentally difficult.

**Part 2:** The Turing machine computation is simulated by manipulating $x$, $z$, and $y$ until finally $z \in E$ holds.

**Part 3:** The number stored in $y$ is converted into the actual output value:

If $y$ has the value $(a_1 a_2 \cdots a_n)_m$ with $a_1, \ldots, a_n \in \{a, b\}$, then the unique number $n$ with $\text{bin}(n) = a_1 a_2 \cdots a_n$ must be computed.

This arithmetic transformation can again be realized by a GOTO program.

**Remark:** Only the second part depends on the transition function $\delta$.

**Schema for Part 2:**

$M_2$: IF $(z \in E)$ THEN GOTO $M_3$;
$a := y$ MOD $m$; (Read symbol)
IF $(z = 0)$ AND $(a = 0)$ THEN GOTO $M_{0,0}$;
IF $(z = 0)$ AND $(a = 1)$ THEN GOTO $M_{0,1}$;
$\vdots$

$M_{0,0}$: $P_{0,0}$; (Execute action $\delta(0,0)$)
GOTO $M_2$;

$M_{0,1}$: $P_{0,1}$; (Execute action $\delta(0,1)$)
GOTO $M_2$;
$\vdots$

$M_3$: Execute Part 3

In program part $P_{i,j}$, the action described by $\delta(i,j)$ is simulated as detailed at the bottom of slide 49. □

# Primitive Recursive and $\mu$-Recursive Functions

LOOP-, WHILE-, and GOTO programs are simplified imperative programs and represent imperative programming languages, where programs are seen as sequences of commands.

In parallel, there are also functional programs, whose main component is the definition of recursive functions. There are computation concepts that align more closely with functional programs.

For example:

- $\lambda$-Calculus (Alonzo Church, 1932)
- $\mu$-recursive and primitive recursive functions (which we will cover in this lecture)

# Primitive Recursive and $\mu$-Recursive Functions

We now define classes of functions of the form $f : \mathbb{N}^k \to \mathbb{N}$.

## Primitive Recursive Functions (Definition)

The class of primitive recursive functions is inductively defined as follows:

- All constant functions of the form $k_m : \mathbb{N}^0 \to \mathbb{N}$ with $k_m() = m$ (for a fixed $m \in \mathbb{N}$) are primitive recursive.
- All projections of the form $\pi_i^k : \mathbb{N}^k \to \mathbb{N}$ with $\pi_i^k(n_1, \ldots, n_k) = n_i$ for $1 \leq i \leq k$ are primitive recursive.
- The successor function $s : \mathbb{N} \to \mathbb{N}$ with $s(n) = n + 1$ is primitive recursive.
- If $g : \mathbb{N}^k \to \mathbb{N}$ and $f_1, \ldots, f_k : \mathbb{N}^r \to \mathbb{N}$ $(k \geq 0)$ are primitive recursive, then the function $f : \mathbb{N}^r \to \mathbb{N}$ defined by

$$f(n_1, \ldots, n_r) = g(f_1(n_1, \ldots, n_r), \ldots, f_k(n_1, \ldots, n_r))$$

is also primitive recursive (Substitution/Composition)

# Primitive Recursive and $\mu$-Recursive Functions

## Primitive Recursive Functions (Definition, Continuation)

- Any function $f$ that is obtained through primitive recursion from primitive recursive functions is primitive recursive.

  This means $f\colon \mathbb{N}^{k+1} \to \mathbb{N}$ must satisfy the following equations (for primitive recursive functions $g\colon \mathbb{N}^k \to \mathbb{N}$, $h\colon \mathbb{N}^{k+2} \to \mathbb{N}$):

$$\begin{aligned} f(0, n_1, \ldots, n_k) &= g(n_1, \ldots, n_k) \\ f(n+1, n_1, \ldots, n_k) &= h(f(n, n_1, \ldots, n_k), n, n_1, \ldots, n_k) \end{aligned}$$

**Intuitively:** in primitive recursion, the definition of $f(n+1, \ldots)$ is reduced to $f(n, \ldots)$. This means that primitive recursion always terminates.

Thus, it is a computational model analogous to LOOP-programs.

**Examples** of primitive recursive functions:

**Addition Function**

The function $add : \mathbb{N}^2 \to \mathbb{N}$ with $add(n, m) = n + m$ is primitive recursive.

$$\begin{aligned} add(0, m) &= m \\ add(n+1, m) &= s(add(n, m)) \end{aligned}$$

Strictly speaking, we should write:

$$\begin{aligned} add(0, m) &= g(m) \\ add(n+1, m) &= h(add(n, m), n, m) \end{aligned}$$

where $g = \pi_1^1$ and $h : \mathbb{N}^3 \to \mathbb{N}$ is the following primitive recursive function:

$$h(x, y, z) = s(\pi_1^3(x, y, z))$$

However, we will usually not be so precise.

## Multiplication Function

The function $mult\colon \mathbb{N}^2 \to \mathbb{N}$ with $mult(n, m) = n \cdot m$ is primitive recursive.

$$\begin{aligned} mult(0, m) &= 0 \\ mult(n+1, m) &= add(mult(n, m), m) \end{aligned}$$

Instead of $mult(0, m) = 0$, we should strictly speaking write $mult(0, m) = k_0()$, which is allowed since in the composition case on Slide 56, we allow $k = 0$.

# Primitive Recursive and $\mu$-Recursive Functions

## Decrement

The function $dec\colon \mathbb{N} \to \mathbb{N}$ with $dec(n) = n - 1$ is primitive recursive.

$$\begin{aligned} dec(0) &= 0 \\ dec(n+1) &= n \end{aligned}$$

## Subtraction

The function $sub : \mathbb{N}^2 \to \mathbb{N}$ with $sub(n, m) = \max\{0, n - m\}$ is primitive recursive.

$$\begin{aligned} sub(n, 0) &= n \\ sub(n, m+1) &= dec(sub(n, m)) \end{aligned}$$

# Primitive Recursive and $\mu$-Recursive Functions

**Reminder:** The binomial coefficient $\binom{n}{k}$ is defined for $n \geq 0, k \geq 1$ by

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \cdot (n-1) \cdots (n-k+1)}{k \cdot (k-1) \cdots 1}.$$

The unary function $n \mapsto \binom{n}{2} = \frac{(n-1)n}{2}$ is primitive recursive.

$$\binom{0}{2} = 0$$

$$\binom{n+1}{2} = \frac{n(n+1)}{2} = \frac{(n-1)n}{2} + n = \binom{n}{2} + n$$

By composition, it follows that the function $c : \mathbb{N}^2 \to \mathbb{N}$ with

$$c(n,m) = n + \binom{n+m+1}{2}$$

is also primitive recursive.

# Primitive Recursive and $\mu$-Recursive Functions

The function $c$ is a bijection from $\mathbb{N}^2$ to $\mathbb{N}$ (pairing function).

| | $n = 0$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| $m = 0$ | 0 | 2 | 5 | 9 | 14 |
| 1 | 1 | 4 | 8 | 13 | 19 |
| 2 | 3 | 7 | 12 | 18 | 25 |
| 3 | 6 | 11 | 17 | 24 | 32 |
| 4 | 10 | 16 | 23 | 31 | 40 |

Note: $c(n, m) = n + \sum_{i=1}^{n+m} i$.

The function $c$ can be used to encode arbitrary $k$-tuples ($k \geq 2$) of natural numbers into a single number:

$$\langle n_1, n_2, \ldots, n_k \rangle = c(n_1, c(n_2, \ldots, c(n_{k-1}, n_k) \ldots))$$

The mapping $(n_1, n_2, \ldots, n_k) \mapsto \langle n_1, n_2, \ldots, n_k \rangle$ is also primitive recursive (for each fixed $k$).

# Primitive Recursive and $\mu$-Recursive Functions

Let $\ell : \mathbb{N} \to \mathbb{N}$ and $r : \mathbb{N} \to \mathbb{N}$ be the unique functions with:

$$\forall n, m \in \mathbb{N} : \ell(c(n, m)) = n \text{ and } r(c(n, m)) = m$$

We will now show that the functions $\ell$ and $r$ are also primitive recursive. With these, we can define primitive recursive decoding functions for encoded $k$-tuples:

$$\begin{aligned}
d_1(n) &= \ell(n) \\
d_2(n) &= \ell(r(n)) \\
&\vdots \\
d_{k-1}(n) &= \ell(\underbrace{r(r(\cdots r}_{k-2 \text{ times}}(n)\cdots)) \\
d_k(n) &= \underbrace{r(r(\cdots r}_{k-1 \text{ times}}(n)\cdots))
\end{aligned}$$

Then: $d_i(\langle n_1, n_2, \ldots, n_k \rangle) = n_i$.

# Primitive Recursive and $\mu$-Recursive Functions

Let $P : \mathbb{N}^{k+1} \to \{0, 1\}$ be a predicate (a function with range $\{0, 1\}$). Then the function $q : \mathbb{N}^{k+1} \to \mathbb{N}$ defined by

$$q(n, n_1, \ldots, n_k) = \begin{cases} 0 & \text{if } P(x, n_1, \ldots, n_k) = \\ \max\{x \leq n \mid P(x, n_1, \ldots, n_k) = 1\} & \text{otherwise} \end{cases}$$

is defined by applying the bounded max-operator to $P$.

If $P$ is primitive recursive, then $q$ is also primitive recursive.

$$\begin{aligned} q(0, n_1, \ldots, n_k) &= 0 \\ q(n+1, n_1, \ldots, n_k) &= \begin{cases} n+1 & \text{if } P(n+1, n_1, \ldots, n_k) = 1 \\ q(n, n_1, \ldots, n_k) & \text{otherwise} \end{cases} \\ &= q(n, n_1, \ldots, n_k) + \\ &\quad P(n+1, n_1, \ldots, n_k) * (n+1 - q(n, n_1, \ldots, n_k)) \end{aligned}$$

# Primitive Recursive and $\mu$-Recursive Functions

Let $P : \mathbb{N}^{k+1} \to \{0,1\}$ be a predicate. Then the predicate $Q : \mathbb{N}^{k+1} \to \{0,1\}$ defined by

$$Q(n, n_1, \ldots, n_k) = \begin{cases} 1 & \text{if } \exists x \leq n : P(x, n_1, \ldots, n_k) = 1 \\ 0 & \text{otherwise} \end{cases}$$

is defined by applying the bounded existential quantifier to $P$.

If $P$ is primitive recursive, then $Q$ is also primitive recursive.

$$\begin{aligned} Q(0, n_1, \ldots, n_k) &= P(0, n_1, \ldots, n_k) \\ Q(n+1, n_1, \ldots, n_k) &= P(n+1, n_1, \ldots, n_k) + Q(n, n_1, \ldots, n_k) \\ &\quad - P(n+1, n_1, \ldots, n_k) * Q(n, n_1, \ldots, n_k) \end{aligned}$$

# Primitive Recursive and $\mu$-Recursive Functions

We can now show that the inverse functions $\ell$ and $r$ of $c : \mathbb{N}^2 \to \mathbb{N}$ are also primitive recursive.

The predicate $C : \mathbb{N}^3 \to \{0, 1\}$ defined by

$$C(x, y, z) = \begin{cases} 1 & \text{if } c(x, y) = z \\ 0 & \text{if } c(x, y) \neq z \end{cases}$$

is primitive recursive:

$$C(x, y, z) = \Big(1 - (c(x, y) - z)\Big) * \Big(1 - (z - c(x, y))\Big).$$

Therefore, the following functions $\ell'$ and $r'$ are also primitive recursive:

$$\begin{aligned} \ell'(k, m, n) &= \max\{x \leq k \mid \exists y \leq m : C(x, y, n) = 1\} \\ r'(k, m, n) &= \max\{y \leq k \mid \exists x \leq m : C(x, y, n) = 1\} \end{aligned}$$

# Primitive Recursive and $\mu$-Recursive Functions

Finally, it holds that $\ell(n) = \ell'(n, n, n)$ and $r(n) = r'(n, n, n)$:

Note that $x \leq c(x, y)$ and $y \leq c(x, y)$ for all $x, y \in \mathbb{N}$.

Further, we have:

$$\begin{aligned}
& \ell'(c(x,y), c(x,y), c(x,y)) \\
= \; & \max\{x' \leq c(x,y) \mid \exists y' \leq c(x,y) : C(x', y', c(x,y)) = 1\} \\
= \; & \max\{x' \leq c(x,y) \mid \exists y' \leq c(x,y) : c(x', y') = c(x,y))\} \\
= \; & \max\{x' \leq c(x,y) \mid \exists y' \leq c(x,y) : x' = x \text{ and } y' = y\} \\
= \; & \max\{x' \leq c(x,y) \mid x' = x\} = x
\end{aligned}$$

and similarly $r'(c(x, y), c(x, y), c(x, y)) = y$.

From the definition of the functions $\ell$ and $r$ (Slide 63) it follows that $\ell(n) = \ell'(n, n, n)$ and $r(n) = r'(n, n, n)$.

# Primitive Recursive and $\mu$-Recursive Functions

**Theorem 6 (Primitive Recursive Functions = LOOP-Computable Functions)**

The class of primitive recursive functions is exactly the same as the class of LOOP-computable functions.

**Proof:** Let $f : \mathbb{N}^r \to \mathbb{N}$ be LOOP-computable.

Then there exists a LOOP program $P$, in which only the variables $x_1, \ldots, x_k$ $(k \geq r)$ appear, such that (see Slide 28)

$$\forall n_1, \ldots, n_r \in \mathbb{N} : f(n_1, \ldots, n_r) = \pi_1([P]_k(n_1, \ldots, n_r, 0, \ldots, 0)).$$

By induction on the construction of $P$, we define a primitive recursive function $g_P : \mathbb{N} \to \mathbb{N}$ such that

$$\forall n_1, \ldots, n_k \in \mathbb{N} : g_P(\langle n_1, \ldots, n_k \rangle) = \langle [P]_k(n_1, \ldots, n_k) \rangle.$$

Since $f(n_1, \ldots, n_r) = d_1(g_P(\langle n_1, \ldots, n_r, 0, \ldots, 0 \rangle))$, it follows that $f$ is also primitive recursive.

**Case 1.** $P = (x_i := x_j \pm c)$: Define

$$g_P(n) := \langle d_1(n), \ldots, d_{i-1}(n), d_j(n) \pm c, d_{i+1}(n), \ldots, d_k(n) \rangle.$$

Then

$$\begin{aligned}
& g_P(\langle n_1, \ldots, n_k \rangle) \\
= \quad & \langle d_1(\langle n_1, \ldots, n_k \rangle), \ldots, d_{i-1}(\langle n_1, \ldots, n_k \rangle), d_j(\langle n_1, \ldots, n_k \rangle) \pm c, \\
& \quad d_{i+1}(\langle n_1, \ldots, n_k \rangle), \ldots, d_k(\langle n_1, \ldots, n_k \rangle) \rangle \\
= \quad & \langle n_1, \ldots, n_{i-1}, n_j \pm c, n_{i+1}, \ldots, n_k \rangle \\
\overset{\text{Slide 26}}{=} \quad & \langle [P]_k(n_1, \ldots, n_k) \rangle.
\end{aligned}$$

Moreover: Since all functions $d_1, \ldots, d_k$ as well as $n \mapsto n \pm c$ and $(n_1, \ldots, n_k) \mapsto \langle n_1, \ldots, n_k \rangle$ are primitive recursive, $g_P$ is also primitive recursive by the above definition.

**Case 2.** $P = (Q; R)$: Define

$$g_P(n) := g_R(g_Q(n)).$$

By induction, for all $n_1, \ldots, n_k \in \mathbb{N}$:

$$\begin{aligned} g_Q(\langle n_1, \ldots, n_k \rangle) &= \langle [Q]_k(n_1, \ldots, n_k) \rangle \\ g_R(\langle n_1, \ldots, n_k \rangle) &= \langle [R]_k(n_1, \ldots, n_k) \rangle \end{aligned}$$

Thus

$$\begin{aligned} g_P(\langle n_1, \ldots, n_k \rangle) &= g_R(g_Q(\langle n_1, \ldots, n_k \rangle)) \\ &= g_R(\langle [Q]_k(n_1, \ldots, n_k) \rangle) \\ &= \langle [R]_k([Q]_k(n_1, \ldots, n_k)) \rangle \\ &\overset{\text{Slide 26}}{=} \langle [P]_k(n_1, \ldots, n_k) \rangle. \end{aligned}$$

Moreover: Since $g_Q$ and $g_R$ are primitive recursive by induction, $g_P$ is also primitive recursive by the above definition.

**Case 3.** $P = (\text{LOOP } x_i \text{ DO } Q \text{ END})$:

First, define the primitive recursive function $h$ by

$$\begin{aligned} h(0, m) &= m \\ h(n+1, m) &= g_Q(h(n, m)) \end{aligned}$$

Thus $h(n, m) = g_Q^n(m)$ (the $n$-fold application of $g_Q$ to $m$).

Finally, define $g_P(x) = h(d_i(x), x)$.

Then:

$$\begin{aligned} g_P(\langle n_1, \ldots, n_k \rangle) &= h(d_i(\langle n_1, \ldots, n_k \rangle), \langle n_1, \ldots, n_k \rangle) \\ &= h(n_i, \langle n_1, \ldots, n_k \rangle) \\ &= g_Q^{n_i}(\langle n_1, \ldots, n_k \rangle) \\ &\overset{\text{Ind.hyp.}}{=} \langle [Q]_k^{n_i}(n_1, \ldots, n_k) \rangle \\ &\overset{\text{Slide 26}}{=} \langle [P]_k(n_1, \ldots, n_k) \rangle. \end{aligned}$$

# Primitive Recursive and $\mu$-Recursive Functions

Let $f : \mathbb{N}^r \to \mathbb{N}$ be primitive recursive.

By induction on the construction of $f$, we show that $f$ is LOOP-computable.

**Case 1**: $f$ is one of the basic functions (constant functions, projections, successor function).

Then $f$ is obviously LOOP-computable.

**Case 2**: There are primitive recursive functions $g, f_1, \ldots, f_k$ such that

$$f(n_1, \ldots, n_r) = g(f_1(n_1, \ldots, n_r), \ldots, f_k(n_1, \ldots, n_r)).$$

By induction, $g, f_1, \ldots, f_k$ are LOOP-computable.

Let $G, F_1, \ldots, F_k$ be LOOP-programs for computing $g, f_1, \ldots, f_k$.

Then the following LOOP-program computes $f$:

# Primitive Recursive and $\mu$-Recursive Functions

$y_1 := x_1; \ldots; y_r := x_r;$
$F_1;$
$z_1 := x_1;$
$x_1 := y_1; \ldots; x_r := y_r;$
$F_2;$
$z_2 := x_1;$
$x_1 := y_1; \ldots; x_r := y_r;$
$F_3;$
$z_3 := x_1;$
$\vdots$
$x_1 := y_1; \ldots; x_r := y_r;$
$F_k;$
$z_k := x_1;$
$x_1 := z_1; \ldots; x_k := z_k;$
$G$

# Primitive Recursive and $\mu$-Recursive Functions

**Explanation:**

- The input numbers $n_1, \ldots, n_r$ are initially in the variables $x_1, \ldots, x_r$. These are saved using $y_1 := x_1; \ldots; y_r := x_r$ into the variables $y_1, \ldots, y_r$.
- Using the program $F_i$, the value $f_i(n_1, \ldots, n_r)$ is computed and this value is in the variable $x_1$ after executing $F_i$.
- By using $z_i := x_1$, the value $f_i(n_1, \ldots, n_r)$ is saved in the variable $z_i$.
- Before starting $F_{i+1}$, the variables $x_1, \ldots, x_r$ must be restored to the input values $n_1, \ldots, n_r$ using $x_1 := y_1; \ldots; x_r := y_r$.
- Finally, each variable $z_i$ $(1 \leq i \leq k)$ contains the value $f_i(n_1, \ldots, n_r)$.
- Using $x_1 := z_1; \ldots; x_k := z_k$, these values are copied into the variables $x_1, \ldots, x_k$. After that, $G$ is executed.
- After $G$ finishes, the variable $x_1$ contains the desired value $g(f_1(n_1, \ldots, n_r), \ldots, f_k(n_1, \ldots, n_r)) = f(n_1, \ldots, n_r)$.

**Case 3**: $f$ is derived from $g$ and $h$ through primitive recursion.

There exist primitive recursive functions $g\colon \mathbb{N}^{r-1} \to \mathbb{N}$ and $h\colon \mathbb{N}^{r+1} \to \mathbb{N}$ such that

$$\begin{aligned} f(0, n_2, \ldots, n_r) &= g(n_2, \ldots, n_r) \\ f(n_1 + 1, n_2, \ldots, n_r) &= h(f(n_1, n_2, \ldots, n_r), n_1, n_2, \ldots, n_r) \end{aligned}$$

The function $f$ can then be computed by the following (pseudocode) LOOP program:

$y := g(x_2, \ldots, x_r);\ k := 0;$
LOOP $x_1$ DO
    $y := h(y, k, x_2, \ldots, x_r); k := k + 1$
END

Using LOOP programs for $g$ and $h$ and intermediate storage similar to Case 2, this pseudocode
can be converted into a LOOP program for $f$. □

# Primitive Recursive and $\mu$-Recursive Functions

We now define another class that is equivalent to WHILE-programs, GOTO-programs, and Turing machines.

## $\mu$-Recursive Functions (Definition)

The $\mu$-recursive functions use the same basic functions (constant functions, projections, successor function) and operators (substitution, primitive recursion) as primitive recursive functions.

Additionally, the $\mu$-Operator may be used.

The $\mu$-Operator transforms a function $f : \mathbb{N}^{k+1} \to \mathbb{N}$ into a function $\mu f : \mathbb{N}^k \to \mathbb{N}$ with

$$\mu f(n_1, \ldots, n_k) = \min\{n \mid f(n, n_1, \ldots, n_k) = 0 \text{ and } \forall m < n : f(m, n_1, \ldots, n_k) \text{ is defined}\}$$

where $\min \emptyset = \textit{undefined}$.

# Primitive Recursive and $\mu$-Recursive Functions

**Intuitive Calculation** of $\mu f(n_1, \ldots, n_k)$:

- Compute $f(0, n_1, \ldots, n_k)$, $f(1, n_1, \ldots, n_k)$, $\ldots$
- If for some $n$ it holds that $f(n, n_1, \ldots, n_k) = 0$, then return $n$ as the function value.
- If $f(m, n_1, \ldots, n_k)$ is undefined (without the function value being 0 previously), or if the function value 0 is never reached: the intuitive calculation does not terminate.
  In this case, $\mu f(n_1, \ldots, n_k) = \min \emptyset = \textit{undefined}$.

**Analogy** to WHILE-programs: it is not clear whether the termination condition will ever be met.

# Primitive Recursive and $\mu$-Recursive Functions

With the $\mu$-operator, we can now also create partial functions that are not total.

## Total Undefined Function

The function $\Omega\colon \mathbb{N} \to \mathbb{N}$ with $\Omega(n) = \textit{undefined}$ for all $n \in \mathbb{N}$ is $\mu$-recursive.

For example, use the binary function $f : \mathbb{N}^2 \to \mathbb{N}$ with $f(n, m) = n + 1$ for all $n, m$.

It holds that $f(n, m) = s(\pi_1^2(n, m))$, where $s$ is the successor function.

Then $\Omega = \mu f$.

**Another Example:**

**Square Root Function**

The function $sqrt : \mathbb{N} \to \mathbb{N}$ with $sqrt(n) = \lceil \sqrt{n} \rceil$ is $\mu$-recursive.

Here, $\lceil \ldots \rceil$ rounds a real number up to the nearest (or equal) integer.

Let $f(m, n) = n - m \cdot m$. (Note: multiplication and subtraction functions are primitive recursive).

Then $sqrt = \mu f$, because: $\mu f(n) = \min\{m \in \mathbb{N} \mid n - m \cdot m = 0\} = \lceil \sqrt{n} \rceil$

However, this function is also primitive recursive.

Intuition: calculating $sqrt(n)$ requires at most $n$ iterations.

# Primitive Recursive and $\mu$-Recursive Functions

**Theorem 7**

The class of $\mu$-recursive functions is exactly the same as the class of WHILE-(GOTO-, Turing-) computable functions.

**Proof:**

We show that the class of $\mu$-recursive functions coincides with the class of WHILE-computable functions.

To do this, it is sufficient to extend the proof of the theorem on Slide 68 (primitive recursive functions $=$ LOOP-computable functions) to include the $\mu$-operator and the WHILE loop.

Let $P = (\text{WHILE } x_i \neq 0 \text{ DO } Q \text{ End})$ be a WHILE-program.

We need to show that the function $g_P : \mathbb{N} \to \mathbb{N}$ defined on Slide 68 is $\mu$-recursive.

By induction, this is already the case for $g_Q$.

# Primitive Recursive and $\mu$-Recursive Functions

As in the proof of the theorem on Slide 68 (Case 3 on Slide 71), we can define a $\mu$-recursive function $h : \mathbb{N}^2 \to \mathbb{N}$ with $h(n, m) = g_Q^n(m)$.

Define $j : \mathbb{N}^2 \to \mathbb{N}$ by $j(n, m) = d_i(h(n, m))$ and

$$g_P(x) := h((\mu j)(x), x).$$

Note: $(\mu j)(\langle n_1, \ldots, n_k \rangle)$ is defined if and only if there exists a number $t$ such that

$$\begin{aligned} 0 &= d_i(h(t, \langle n_1, \ldots, n_k \rangle)) \\ &= d_i(g_Q^t(\langle n_1, \ldots, n_k \rangle)) \\ &= d_i(\langle [Q]_k^t(n_1, \ldots, n_k) \rangle) \\ &= \pi_i^k([Q]_k^t(n_1, \ldots, n_k)) \end{aligned}$$

and $[Q]_k^s(n_1, \ldots, n_k)$ is defined for all $s < t$.

# Primitive Recursive and $\mu$-Recursive Functions

In this case, $(\mu j)(\langle n_1, \ldots, n_k \rangle)$ is the smallest such number $t$.

Therefore, $(\mu j)(\langle n_1, \ldots, n_k \rangle)$ is defined if and only if the program $P = (\text{WHILE } x_i \neq 0 \text{ DO } Q \text{ End})$ terminates with input $n_1, \ldots, n_k$.

Thus: $g_P(\langle n_1, \ldots, n_k \rangle)$ is defined $\iff$ $\langle [P]_k(n_1, \ldots, n_k) \rangle$ is defined.

Moreover: If $(\mu j)(\langle n_1, \ldots, n_k \rangle)$ is defined and equal to $t \in \mathbb{N}$, then $t$ is the number of iterations of the WHILE loop for input $n_1, \ldots, n_k$.

In this case:

$$
\begin{aligned}
g_P(\langle n_1, \ldots, n_k \rangle) &= h((\mu j)(\langle n_1, \ldots, n_k \rangle), \langle n_1, \ldots, n_k \rangle) \\
&= h(t, \langle n_1, \ldots, n_k \rangle) \\
&= g_Q^t(\langle n_1, \ldots, n_k \rangle) \\
&\overset{\text{Ind.hyp.}}{=} \langle [Q]_k^t(n_1, \ldots, n_k) \rangle \\
&\overset{\text{Folie 33}}{=} \langle [P]_k(n_1, \ldots, n_k) \rangle.
\end{aligned}
$$

# Primitive Recursive and $\mu$-Recursive Functions

Let $f = \mu g : \mathbb{N}^r \to \mathbb{N}$ be a $\mu$-recursive function $g : \mathbb{N}^{r+1} \to \mathbb{N}$.

Then $f$ can be computed by the following (pseudocode) WHILE-program:

$y := 0;\ z := g(0, x_1, \ldots, x_r);$
WHILE $z \neq 0$ DO
    $y := y + 1;\ z := g(y, x_1, \ldots, x_r);$
END;
$x_1 := y$

# Primitive Recursive and $\mu$-Recursive Functions

We will now show that there are total Turing-computable/$\mu$-recursive functions that are not primitive recursive.

A classic example of this is the so-called Ackermann Function.

**Ackermann Function $a\colon \mathbb{N}^2 \to \mathbb{N}$ (Ackermann 1928)**

$$a(0,y) = y+1 \tag{1}$$
$$a(x,0) = a(x-1,1), \text{ if } x > 0 \tag{2}$$
$$a(x,y) = a(x-1,a(x,y-1)), \text{ if } x,y > 0 \tag{3}$$

**Lemma 8**

The Ackermann function is a total function.

**Proof:** By induction on the first argument $x$.

Let $x, y \in \mathbb{N}$.

If $x = 0$, then $a(x, y) \overset{(1)}{=} y + 1$.

If $x > 0$, then

$$\begin{aligned} a(x,y) &\overset{(3)}{=} a(x-1, a(x, y-1)) \\ &\overset{(3)}{=} a(x-1, a(x-1, a(x, y-2))) = \cdots \\ &= \underbrace{a(x-1, a(x-1, \ldots a(x-1}_{(y+1)\text{-times}}, 1) \ldots)). \end{aligned}$$

Since by induction all values $a(x-1, z)$ (for $z \in \mathbb{N}$) are defined, $a(x, y)$ is also defined. □

# Primitive Recursive and $\mu$-Recursive Functions

Value table of the Ackermann function for small values:

| $y =$ | 0 | 1 | 2 | 3 | 4 | ... | $a(x, y)$ |
|---|---|---|---|---|---|---|---|
| $x = 0$ | 1 | 2 | 3 | 4 | 5 | ... | $y + 1$ |
| $x = 1$ | 2 | 3 | 4 | 5 | 6 | ... | $y + 2$ |
| $x = 2$ | 3 | 5 | 7 | 9 | 11 | ... | $2y + 3$ |
| $x = 3$ | 5 | 13 | 29 | 61 | 125 | ... | $2^{y+3} - 3$ |
| $x = 4$ | 13 | 65533 | $> 10^{19727}$ | | | | $\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{y+3 \text{ twos}} - 3$ |
| ... | | | | | | | |

## Satz 9

The Ackermann function is WHILE-computable, but not primitive recursive or LOOP-computable.

**Proof:**

We first show that the Ackermann function is WHILE-computable (the Ackermann function is certainly computable in the intuitive sense).

For this purpose, it is useful to introduce stacks (LIFO) of natural numbers.

A sequence $(n_0, n_1, \ldots, n_k)$ of natural numbers can be stored in a single number using the encoding function $(n_0, n_1, \ldots, n_k) \mapsto \langle n_0, n_1, \ldots, n_k \rangle$ (see Slide 62).

Let stack be an integer variable representing a stack of natural numbers. We define the following operations:

- INIT(stack): stack $:= 0$
- PUSH$(n,$ stack$)$ for $n \in \mathbb{N}$: stack $:= c(n+1,$ stack$)$
- $x :=$ POP(stack): $x := \ell($stack$) - 1;$ stack $:= r($stack$)$

Additionally, we use size(stack) $\neq 1$ as a shorthand for $r($stack$) \neq 0$.

**Note:** In a PUSH operation, the argument to be pushed onto the stack is incremented by 1, and in a POP operation, it is decremented again.

This is necessary to distinguish these arguments from the bottom stack symbol 0.

If we did not perform this increment, all stacks of the form $(0, 0, \ldots, 0)$ would be encoded by the number 0.

Using these operations, the Ackermann function can be computed by the following WHILE-program:

# Primitive Recursive and $\mu$-Recursive Functions

```
INIT(stack);
PUSH(x_1, stack);
PUSH(x_2, stack);
WHILE size(stack) ≠ 1 DO
    y := POP(stack);
    x := POP(stack);
    IF x = 0 THEN PUSH(y + 1, stack);
    ELSEIF y = 0 THEN PUSH(x − 1, stack); PUSH(1, stack);
    ELSE PUSH(x − 1, stack); PUSH(x, stack); PUSH(y − 1, stack);
    END (if)
END (while)
x_1 := POP(stack);
```

We will now show that the Ackermann function grows faster than any primitive recursive function.

For this, we prove a series of lemmas.

**Lemma 10**

$\forall x, y \in \mathbb{N} : y < a(x, y)$

**Proof:** Induction on $x$.

Base Case: $x = 0$.

It holds that $y < y + 1 \overset{(1)}{=} a(0, y)$ for all $y \in \mathbb{N}$.

Inductive Step: Assume $\forall y \in \mathbb{N} : y < a(x, y)$ (IH 1).

We will now show by induction on $y \in \mathbb{N}$ that $\forall y \in \mathbb{N} : y < a(x + 1, y)$.

Base Case: $y = 0$.

It holds that $1 < a(x, 1)$ (by IH 1) and thus $0 < 1 < a(x, 1) \overset{(2)}{=} a(x + 1, 0)$.

Inductive Step: Assume $y < a(x + 1, y)$ (IH 2).

We will now show $y + 1 < a(x + 1, y + 1)$.

First, by (IH 1): $a(x+1,y) < a(x,a(x+1,y))$
(replace $y$ in (IH 1) with $a(x+1,y)$).

Therefore: $y+1 \overset{\text{(IH 2)}}{\leq} a(x+1,y) < a(x,a(x+1,y)) \overset{(3)}{=} a(x+1,y+1)$. □

**Lemma 11**

$\forall x,y \in \mathbb{N} : a(x,y) < a(x,y+1)$

**Proof:**

For $x = 0$, it holds that $a(0,y) \overset{(1)}{=} y+1 < y+2 \overset{(1)}{=} a(0,y+1)$ for all $y \in \mathbb{N}$.

For $x > 0$, we have $a(x,y) < a(x-1,a(x,y))$ by Lemma 10
(replace $x$ in Lemma 10 with $x-1$ and $y$ with $a(x,y)$).

This yields $a(x,y) < a(x-1,a(x,y)) \overset{(3)}{=} a(x,y+1)$. □

**Lemma 12**

$\forall x, y \in \mathbb{N} : a(x, y+1) \leq a(x+1, y)$

**Proof:** Induction on $y$.

IA: $y = 0$.

It holds that $a(x, 1) \overset{(2)}{=} a(x+1, 0)$ for all $x \in \mathbb{N}$.

IS: Assume $\forall x \in \mathbb{N} : a(x, y+1) \leq a(x+1, y)$ (IH).

We show $a(x, y+2) \leq a(x+1, y+1)$ for all $x \in \mathbb{N}$.

By Lemma 10, $y + 1 < a(x, y+1)$.

Thus, $y + 2 \leq a(x, y+1) \overset{\text{(IH)}}{\leq} a(x+1, y)$.

Lemma 11 implies $a(x, y+2) \leq a(x, a(x+1, y))$.

Therefore, $a(x, y+2) \leq a(x, a(x+1, y)) \overset{(3)}{=} a(x+1, y+1)$. □

**Lemma 13**

$\forall x, y \in \mathbb{N} : a(x, y) < a(x + 1, y)$

**Proof:**

Lemma 11 $\rightsquigarrow$ $a(x, y) < a(x, y + 1)$.

Lemma 12 $\rightsquigarrow$ $a(x, y + 1) \leq a(x + 1, y)$. □

From Lemma 11 and Lemma 13, it follows that the function $a$ is monotonic:

If $x \leq x'$ and $y \leq y'$ then $a(x, y) \leq a(x', y')$. Additionally, if $x < x'$ or $y < y'$ then $a(x, y) < a(x', y')$.

# Primitive Recursive and $\mu$-Recursive Functions

Let $P$ be a LOOP program where no variable $x_i$ with $i > k$ appears.

For a tuple $(n_1, \ldots, n_k) \in \mathbb{N}^k$, we write

$$\sum(n_1, \ldots, n_k) = n_1 + \cdots + n_k.$$

We then define

$$f_P(n) = \max\{\sum[P]_k(n_1, \ldots, n_k) \mid n_1, \ldots, n_k \in \mathbb{N}, \sum(n_1, \ldots, n_k) \leq n\}.$$

**Lemma 14**

For every LOOP program $P$, there exists a number $\ell$ such that for all $n \in \mathbb{N}$ it holds: $f_P(n) < a(\ell, n)$.

**Proof:** Induction over the construction of $P$.

# Primitive Recursive and $\mu$-Recursive Functions

Without loss of generality, assume $P$ satisfies the following properties:

- For each subprogram $x_i := x_j \pm c$ in $P$, we have $c = 1$.

  For example, $x_i := x_j + 2$ can be replaced by $x_i := x_j + 1; x_i := x_i + 1$.

- For each subprogram $\text{LOOP } x_i \text{ DO } Q \text{ END}$ in $P$:
  $x_i$ does not appear in $Q$.

  If $x_i$ does appear in $Q$, replace $\text{LOOP } x_i \text{ DO } Q \text{ END}$ with
  $y := x_i;\ \text{LOOP } y \text{ DO } Q \text{ END}$, where $y$ is a new variable.

**Case 1**: $P = (x_i := x_j \pm 1)$

Then we have $f_P(n) \leq 2n + 1$.

By induction on $y$, it can be easily shown that:
$a(1, y) = y + 2$ and $a(2, y) = 2y + 3$ (see exercise).

Thus, $f_P(n) < a(2, n)$.

**Case 2**: $P = (P_1; P_2)$.

By the induction hypothesis, there exist numbers $\ell_1$ and $\ell_2$ such that

$$\forall n \in \mathbb{N} : f_{P_1}(n) < a(\ell_1, n) \text{ and } f_{P_2}(n) < a(\ell_2, n). \tag{4}$$

We first show that $f_P(n) \leq f_{P_2}(f_{P_1}(n))$.

By the definition of $f_P(n)$, there exists a tuple $(n_1, \ldots, n_k) \in \mathbb{N}^k$ with $\sum(n_1, \ldots, n_k) \leq n$ and

$$f_P(n) = \sum [P]_k(n_1, \ldots, n_k) = \sum [P_2]_k([P_1]_k(n_1, \ldots, n_k)).$$

Now, by the definition of $f_{P_1}(n)$, we have: $\sum [P_1]_k(n_1, \ldots, n_k) \leq f_{P_1}(n)$.

With the definition of $f_{P_2}(n)$, it follows that:

$$f_P(n) = \sum [P_2]_k([P_1]_k(n_1, \ldots, n_k)) \leq f_{P_2}(f_{P_1}(n)).$$

With $\ell_3 := \max\{\ell_1 - 1, \ell_2\}$ it follows:

$$\begin{aligned}
f_P(n) &\leq f_{P_2}(f_{P_1}(n)) \\
&< a(\ell_2, a(\ell_1, n)) && \text{((4) and monotonicity of } a\text{)} \\
&\leq a(\ell_3, a(\ell_3 + 1, n)) && \text{(monotonicity of } a\text{)} \\
&= a(\ell_3 + 1, n + 1) && \text{(definition of } a\text{)} \\
&\leq a(\ell_3 + 2, n) && \text{(Lemma 12)}
\end{aligned}$$

Thus, we can choose $\ell = \ell_3 + 2$ in Lemma 14.

**Case 3**: $P = (\textsc{Loop}\ x_i\ \textsc{Do}\ Q\ \textsc{End})$

By the induction hypothesis, there exists a number $\ell_1$ such that

$$\forall n \in \mathbb{N} : f_Q(n) < a(\ell_1, n). \tag{5}$$

Note: $x_i$ does not appear in $Q$.

# Primitive Recursive and $\mu$-Recursive Functions

Choose $m, n_1, \ldots, n_{i-1}, n_{i+1}, \ldots, n_k \leq n$ such that:

$$\begin{aligned} f_P(n) &= \max\{\sum[P]_k(n_1, \ldots, n_k) \mid n_1, \ldots, n_k \in \mathbb{N}, \sum(n_1, \ldots, n_k) \leq n\} \\ &= \sum[P]_k(n_1, \ldots, n_{i-1}, m, n_{i+1}, \ldots, n_k), \end{aligned}$$

where $n_1 + \cdots + n_{i-1} + n_{i+1} + \cdots + n_k + m \leq n$.

If $m = 0$, then:

$$\begin{aligned} f_P(n) &= \sum[P]_k(n_1, \ldots, n_{i-1}, 0, n_{i+1}, \ldots, n_k) \\ &= \sum(n_1, \ldots, n_{i-1}, 0, n_{i+1}, \ldots, n_k) \\ &\leq n \\ &< n + 1 \\ &= a(0, n). \end{aligned}$$

If $m \geq 1$, then we get (since $x_i$ does not occur in $Q$):

$$\begin{aligned}
f_P(n) &= \sum [P]_k(n_1, \ldots, n_{i-1}, m, n_{i+1}, \ldots, n_k) \\
&= \sum [Q]_k^m(n_1, \ldots, n_{i-1}, m, n_{i+1}, \ldots, n_k) \\
&\leq \underbrace{f_Q(f_Q(\cdots f_Q}_{m\text{-times}}(n-m)\cdots)) + m \\
&< a(\ell_1, \underbrace{f_Q(f_Q(\cdots f_Q}_{m-1\text{-times}}(n-m)\cdots))) + m \\
&\vdots \\
&< \underbrace{a(\ell_1, a(\ell_1, \cdots a(\ell_1}_{m\text{-times}}, n-m)\cdots)) + m
\end{aligned}$$

Since this estimate contains the symbol "$<$" $m$-times, we get:

$$
\begin{aligned}
f_P(n) &\leq \underbrace{a(\ell_1, a(\ell_1, \cdots a(\ell_1,}_{m\text{-times}} n-m)\cdots)) \\
&< \underbrace{a(\ell_1, \cdots a(\ell_1,}_{m-1\text{-times}} a(\ell_1+1, n-m))\cdots) && \text{(Monotonicity of } a\text{, } m \geq 1) \\
&= a(\ell_1+1, n-1) && \text{(Definition of } a) \\
&< a(\ell_1+1, n) && \text{(Monotonicity of } a)
\end{aligned}
$$

Thus, we can choose $\ell = \ell_1 + 1$ in Lemma 14. □

# Primitive Recursive and $\mu$-Recursive Functions

We can now finally conclude the proof of Theorem 9.

Suppose the Ackermann function $a$ were LOOP-computable.

Then the function $g : \mathbb{N} \to \mathbb{N}$ with $g(n) = a(n, n)$ is also LOOP-computable.

Let $P$ be a LOOP program such that

$$\forall n \in \mathbb{N} : g(n) = \pi_0([P]_k(n, 0, \ldots, 0)).$$

According to Lemma 14, there exists a constant $\ell$ such that

$$\forall n \in \mathbb{N} : f_P(n) < a(\ell, n).$$

For $n = \ell$, it follows that

$$g(\ell) = \pi_0([P]_k(\ell, 0, \ldots, 0)) \leq f_P(\ell) < a(\ell, \ell) = g(\ell).$$

This is a contradiction. □

**Overview**

- First, we formally define the concept of decidability. What does it actually mean for a problem to be decidable?
- Next, we introduce the concept of semi-decidability.
  Here, it is allowed that the decision procedure may not terminate and not provide an answer for a negative case.
- Finally, we discuss negative results.
  How can one prove that a problem is undecidable?

# Decidability

## Decidability (Definition)

A language $L \subseteq \Sigma^*$ is called decidable, if the characteristic function of $L$, i.e., the function $\chi_L : \Sigma^* \to \{0, 1\}$ with

$$\chi_L(w) = \begin{cases} 1 & \text{if } w \in L \\ 0 & \text{if } w \notin L \end{cases}$$

is computable.

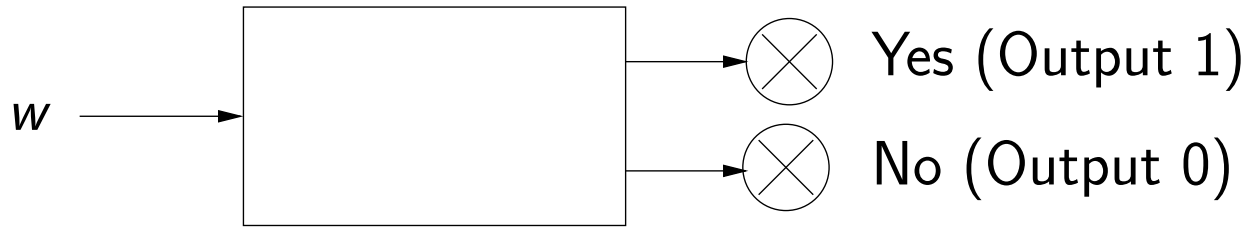
A language that is not decidable is called undecidable.

**Intuition:** There exists an algorithmic procedure that, given an input $w \in \Sigma^*$, behaves as follows:

- If $w \in L$, the procedure terminates after a finite number of steps with the output 1 ("Yes, $w$ belongs to $L$").
- If $w \notin L$, the procedure terminates after a finite number of steps with the output 0 ("No, $w$ does not belong to $L$").

# Decidability

Representation of decidability using a machine model:




For every input, the machine computes for a finite amount of time and then outputs either "Yes" or "No".

# Decidability

In semi-decidability, it is allowed that the characteristic function is undefined in the negative case, meaning no answer is returned. In concrete computational models, this results in non-termination.

## Semi-Decidability (Definition)

A language $L \subseteq \Sigma^*$ is called semi-decidable, if the "partial" characteristic function of $L$, i.e., the function $\chi'_L : \Sigma^* \to \{1\}$ with

$$\chi'_L(w) = \begin{cases} 1 & \text{if } w \in L \\ \textit{undefined} & \text{if } w \notin L \end{cases}$$

is computable.

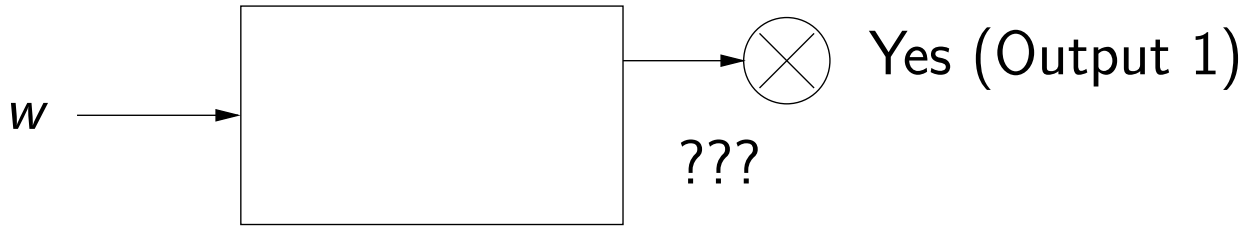
# Decidability

**Intuition:** There is an algorithmic procedure that, given an input $w \in \Sigma^*$, behaves as follows:

- If $w \in L$, the procedure terminates after a finite number of steps with output 1 ("Yes, $w$ belongs to $L$").
- If $w \notin L$, the procedure never terminates.

# Decidability

Representation of semi-decidability using a machine model:




For any input, the machine computes and outputs "Yes" after a finite amount of time if $w \in A$. If $w \notin A$, the machine never terminates.

This means you can never be sure if "Yes" will eventually be output, as the machine's response time is not bounded.

**Satz 15 (Semi-Decidability and Chomsky-0 Languages)**

A language $A$ is semi-decidable if and only if it is of type 0.

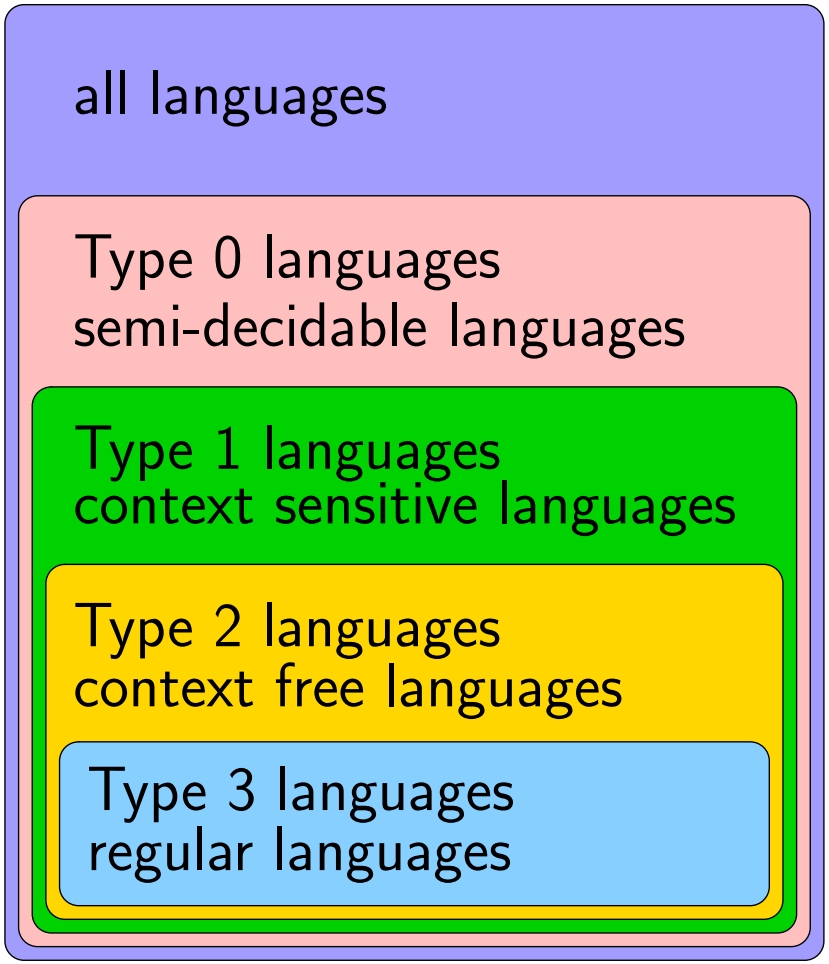
**Proof:** The Chomsky-0 languages are exactly the languages accepted by a Turing machine, see FSA, slide 354.

A Turing machine that computes the partial characteristic function $\chi'_A$ also accepts the language $A$, as it moves into an accepting state after writing the 1.

A Turing machine that accepts a language $A$ can easily be converted into a Turing machine that computes the "partial" characteristic function $\chi'_A$, by erasing the tape after reaching an accepting state and then writing a 1. □

# Decidability

As a reminder: the Chomsky hierarchy



# Decidability

**Remarks:**

- In the context of decidability questions, languages are often referred to as problems.
- Even though characteristic functions take words as arguments, they can easily be viewed as functions over natural numbers and thus computed with WHILE- or GOTO-programs. Every word from $\Sigma^*$ can be interpreted as a number in base $b$, where $b \geq |\Sigma|$ (see also the conversion of Turing machines into GOTO-programs on slide 47).
- Therefore, we will consider problems as subsets of $\mathbb{N}$ or $\mathbb{N}^k$ from now on.

# Decidability

Typical examples of problems:

## Example 1: the word problem

Given a fixed Chomsky grammar $G$, the problem is $A = L(G)$. We already know that $A$ is decidable if $G$ is a Chomsky-1 grammar. Moreover, there exist grammars for which $L(G)$ is not decidable (proof coming soon).

## Example 2: the general word problem

The general word problem is the set

$$A = \{(w, G) \mid w \in L(G),\ G \text{ Chomsky grammar over the alphabet } \Sigma\},$$

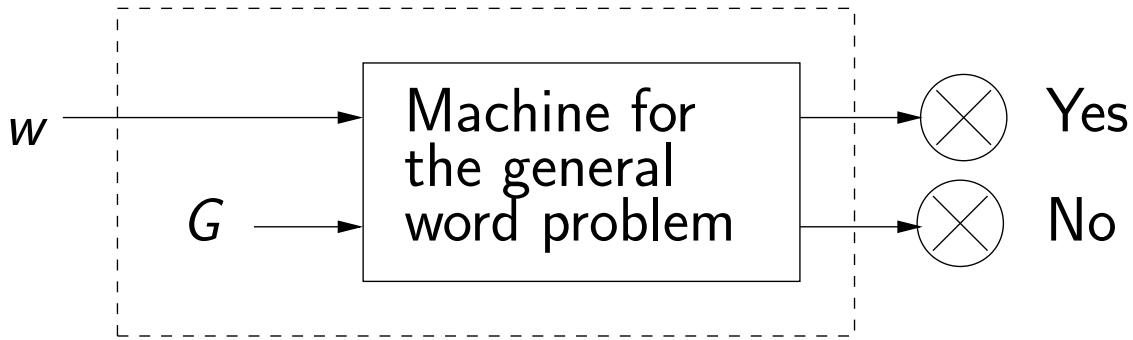
where the pairs $(w, G)$ must be suitably encoded as words (e.g., by listing all productions of $G$, separated by a symbol #).

## Satz 16 (Undecidability of the General Word Problem)

The general word problem is undecidable.

**Proof:** Let $G$ be a grammar for which the word problem $L(G)$ is undecidable (proof to follow).

If the general word problem $A$ were decidable, then $L(G)$ would also be decidable. For a given word $w$, one would only need to check if $(w, G) \in A$. $\rightsquigarrow$ Contradiction!




Machine for the (specific) word problem

# Decidability

This means that from a machine that solves the general word problem, one could construct a machine to solve the (special) word problem. However, since such a machine does not exist for all grammars $G$, the former cannot exist either.

Such arguments ("if there is a procedure for $A$, then one can construct a procedure for $B$") are called reductions. We will use them frequently in the following.

## Example 3: The Intersection Problem

The intersection problem for context-free grammars is the set

$$A = \{(G_1, G_2) \mid G_1, G_2 \text{ are context-free grammars, } L(G_1) \cap L(G_2) \neq \emptyset\}.$$

The intersection problem is undecidable (proof still pending).

**Intuitively speaking:** There is no algorithmic procedure (e.g., a C program) that takes as input two context-free grammars $G_1$ and $G_2$ and, after a finite number of steps, produces the following output:

- 1, if $L(G_1) \cap L(G_2) \neq \emptyset$,
- 0, if $L(G_1) \cap L(G_2) = \emptyset$.

**Theorem 17 (Decidability and Semi-Decidability)**

A language $A$ is decidable if and only if both $A$ and $\overline{A}$ (the complement of $A$) are semi-decidable.

**Proof:**

Suppose $A$ is decidable.

Then the characteristic function $\chi_A$ is computable by a Turing machine $M$.

The following Turing machine $M_A$ computes the partial characteristic function $\chi'_A$:

- $M_A$ simulates the machine $M$.
- If $M$ terminates with output 0, $M_A$ enters an infinite loop.

# Decidability

The following Turing machine $M_{\overline{A}}$ computes the partial characteristic function $\chi'_{\overline{A}}$:

- $M_{\overline{A}}$ simulates the machine $M$.
- If $M$ terminates with output 1, $M_{\overline{A}}$ enters an infinite loop.
- If $M$ terminates with output 0, $M_{\overline{A}}$ terminates with output 1.

Thus, both $A$ and $\overline{A}$ are semi-decidable.

Now suppose both $A$ and $\overline{A}$ are semi-decidable.

Let $M_A$ (or $M_{\overline{A}}$) be a Turing machine that computes the partial characteristic function $\chi'_A$ (or $\chi'_{\overline{A}}$).

The following algorithm computes the characteristic function of $A$:

# Decidability

INPUT $w$
$t := 1$;
WHILE true DO
  IF $M_A$ terminates on input $w$ after $t$ steps THEN
    OUTPUT(1)
  ELSEIF $M_{\overline{A}}$ terminates on input $w$ after $t$ steps THEN
    OUTPUT(0)
  END
  $t := t + 1$
END

Note: The WHILE loop will always terminate after a finite number of steps, since either $w \in A$ or $w \notin A$.

Thus, there exists a number $t$ such that either $M_A$ or $M_{\overline{A}}$ terminates on input $w$ after $t$ steps. □

## Recursively Enumerable (Definition)

A language $L \subseteq \Sigma^*$ is recursively enumerable if $L = \emptyset$ or there exists a total and computable function $f : \mathbb{N} \to \Sigma^*$ such that

$$L = \{f(n) \mid n \in \mathbb{N}\} = \{f(1), f(2), f(3), \dots\}.$$

**Remarks:**

- Terminology: the function $f$ enumerates the language $L$.
- An equivalent definition of recursive enumerability: there exists a total, computable, and surjective function $f : \mathbb{N} \to L$.
- The mathematical concept of countability is defined similarly, but it does not require that $f$ be computable. Hence:
  $L$ is recursively enumerable $\Rightarrow$ $L$ is countable.

**Theorem 18 (Recursive Enumerability and Semi-Decidability)**

A language $L$ is recursively enumerable if and only if it is semi-decidable.

**Proof:**

Recursive Enumerability $\rightarrow$ Semi-Decidability:

Given: A recursively enumerable language $L \subseteq \Sigma^*$, described by a computable and total function $f : \mathbb{N} \rightarrow \Sigma^*$.

Goal: Construct a Turing machine that determines whether a word $w \in \Sigma^*$ is in $L$.

Solution: Compute $f(1), f(2), f(3), \ldots$ until $w = f(i)$ for some $i$. In this case, the Turing machine outputs 1; otherwise, it does not terminate.

# Decidability

Semi-Decidability $\rightarrow$ Recursive Enumerability:

Given: A semi-decidable language $L \subseteq \Sigma^*$, described by a deterministic Turing machine $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ with $L = T(M)$.

Goal: Construct a Turing machine $M'$ that computes a function $f$ such that $\{f(1), f(2), f(3), \dots\} = L$.

The case where $L$ is finite is simple: If $L = \{w_1, w_2, \dots, w_n\}$, then the function $f : \mathbb{N} \rightarrow \Sigma^*$ defined by $f(i) = w_i$ for $1 \leq i \leq n$ and $f(i) = w_n$ for all $i > n$ is computable.

Now consider the case where $L$ is infinite.

Recall how we described successful computations of $M$ using words over the alphabet $\Omega = Z \cup \Gamma \cup \{\#\}$ in FSA (Slide 359).

Since $M$ is deterministic, there is exactly one successful computation $c(w)$ for each word $w \in T(M)$.

# Decidability

$M'$ operates as follows:

INPUT: $i \in \mathbb{N}$
$c := \varepsilon$,
$j := 0$;
WHILE true DO
  $c :=$ the length-lexicographically next word after $c$ from $\Omega^*$;
  IF $c$ is a successful computation THEN
    $j := j + 1$
    IF $j = i$ THEN OUTPUT $w$ if $c = c(w)$
  END
END

Justification of correctness:

- Let $c(w_1), c(w_2), \ldots$ be the length-lexicographic listing of all successful computations of $M$.
- Then $M'$ computes the function $i \mapsto w_i$. □

# Decidability

From Theorems 7 in FSA (Slide 354), 15 (Slide 108) and 18 (Slide 119) it follows that the following statements are equivalent for a language $L$:

## Semi-Decidability and Equivalent Terms

- $L$ is semi-decidable, i.e., $\chi'_L$ is (Turing, WHILE-, GOTO-)computable.
- $L$ is recursively enumerable
- $L$ is of Type 0.
- $L = T(M)$ for some Turing machine $M$.

# Halting Problem/Encoding Turing Machines

Our goal now is to show that the so-called Halting Problem is undecidable.

**Halting Problem (informally)**

- **Input:** A deterministic Turing machine $M$ with input $w$.
- **Output:** Does $M$ halt on $w$?

To do this, we first need to define more precisely how a Turing machine can be encoded to be used as input for a computable (characteristic) function.

# Halting Problem/Encoding Turing Machines

**Goal:** Encode a deterministic Turing machine

$$M = (Z, \{0,1\}, \Gamma, \delta, z_0, \square, E)$$

by a word over the alphabet $\{0,1\}$.

Without loss of generality, we can assume the following:

$$\begin{aligned} \Gamma &= \{0,1,\ldots,m\} \text{ where } \square = m \geq 2, \\ Z &= \{1,\ldots,n\} \text{ where } z_0 = 1 \text{ and} \\ E &= \{k+1,\ldots,n\} \end{aligned}$$

In the following, $1^i$ denotes the word $\underbrace{11\cdots1}_{i \text{ ones}}$.

The most important part of $M$ is the transition function $\delta$. This can be encoded as a word over the alphabet $\{0,1\}$ as follows:

# Halting Problem/Encoding Turing Machines

For all $1 \leq i \leq k$ and $0 \leq j \leq m$, we define the word $w_{i,j}$ as follows:

Let $\delta(i,j) = (i',j',y)$. Then

$$w_{i,j} = 1^i 0 1^j 0 1^{i'} 0 1^{j'} 0 1^{\text{code}(y)} 0 \in \{0,1\}^*.$$

$$\text{where code}(y) = \begin{cases} 1 & \text{if } y = L \\ 2 & \text{if } y = R \\ 3 & \text{if } y = N \end{cases}$$

Then the deterministic Turing machine $M$ can be encoded by the following word $\text{code}(M) \in \{0,1\}^*$:

$$\text{code}(M) = 1^n 0 1^m 0 1^k 0 \prod_{1 \leq i \leq k} \prod_{0 \leq j \leq m} w_{i,j}$$

# Halting Problem/Encoding Turing Machines

**Remarks:** Many of the choices in our encoding of Turing machines are highly arbitrary. There are many other ways to encode Turing machines. What is important here is that there exists a possible encoding that we agree upon.

**Note:** Not every word $w \in \{0,1\}^*$ is the code of a Turing machine. Nevertheless, we want to assign a Turing machine $M_w$ to every word $w \in \{0,1\}^*$.

To this end, we fix an arbitrary but fixed deterministic Turing machine $\widehat{M}$ (the default machine) with input alphabet $\{0,1\}$. Then define

$$M_w := \begin{cases} M & \text{if } \mathrm{code}(M) = w \\ \widehat{M} & \text{if no Turing machine } M \text{ exists with } \mathrm{code}(M) = w \end{cases}$$

# Halting Problem/Encoding Turing Machines

We can now define two different variants of the halting problem.

## Halting Problem

The (general) halting problem is the language

$$\begin{aligned} H &= \{w\#x \mid w, x \in \{0,1\}^*, M_w \text{ halts on } x\} \\ &= \{w\#x \mid w, x \in \{0,1\}^*, x \in T(M_w)\} \end{aligned}$$

## Special Halting Problem

The special halting problem is the language

$$\begin{aligned} K &= \{w \in \{0,1\}^* \mid M_w \text{ halts on } w\} \\ &= \{w \in \{0,1\}^* \mid w \in T(M_w)\}. \end{aligned}$$

# Universal Turing Machine

Note the self-reference in the definition of $K$: A Turing machine $M = M_w$ receives its own encoding $w$ as input.

This is, however, perfectly possible.

For example, if you have written a C program $P$ that takes a text file as input, you can certainly apply the program $P$ to its own source code.

# Universal Turing Machine

Some of the following results are based on the fact that there exists a deterministic Turing machine that can simulate another Turing machine when its encoding is provided. Such a Turing machine is also called a universal Turing machine.

## Universal Turing Machine

A deterministic Turing machine $U$ is called a universal Turing machine if it behaves as follows when given input $w\#x$:

- If $M_w$ does not halt on input $x$, then $U$ does not halt on input $w\#x$ either.
- If $M_w$ halts on input $x$ with output $y$, then $U$ also halts on input $w\#x$ with $y$.

From now on, let $U$ be a universal Turing machine.

# Undecidability of the Halting Problem

**Satz 19 (Undecidability of the Halting Problem)**

The special halting problem $K$ is undecidable.

**Proof:**

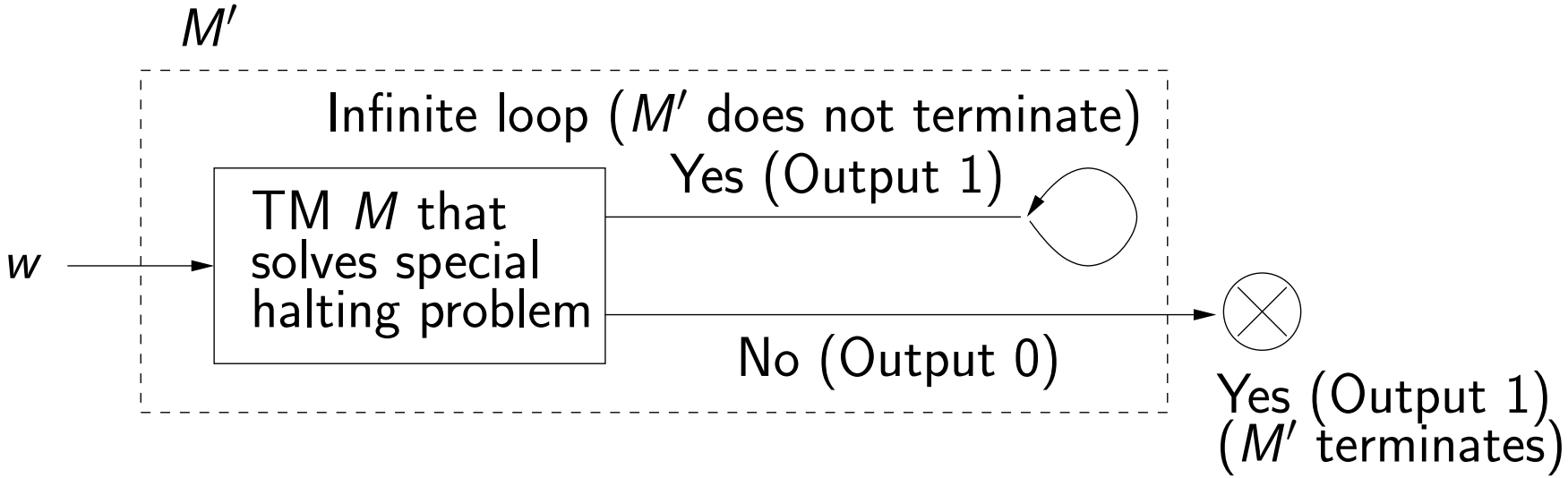
Assume that the special halting problem $K$ is decidable, i.e., the characteristic function $\chi_K : \{0,1\}^* \to \{0,1\}$ is computed by a Turing machine $M$.

We can then construct a Turing machine $M'$ that behaves as follows:

INPUT $w \in \{0,1\}^*$
Simulate $M$ on input $w$
IF $\chi_K(w) = 0$ THEN OUTPUT(1)
ELSE Enter an infinite loop
END

# Undecidability of the Halting Problem

Illustration of the Turing machine $M'$:




Let $w' \in \{0,1\}^*$ be such that $M' = M_{w'}$.

We then obtain the following contradiction:

$$
\begin{aligned}
M' = M_{w'} \text{ halts on input } w' &\iff M \text{ outputs 0 on input } w' \\
&\iff \chi_K(w') = 0 \\
&\iff w' \notin K \\
&\iff M_{w'} \text{ does not halt on input } w'
\end{aligned}
$$

# Undecidability of the Halting Problem

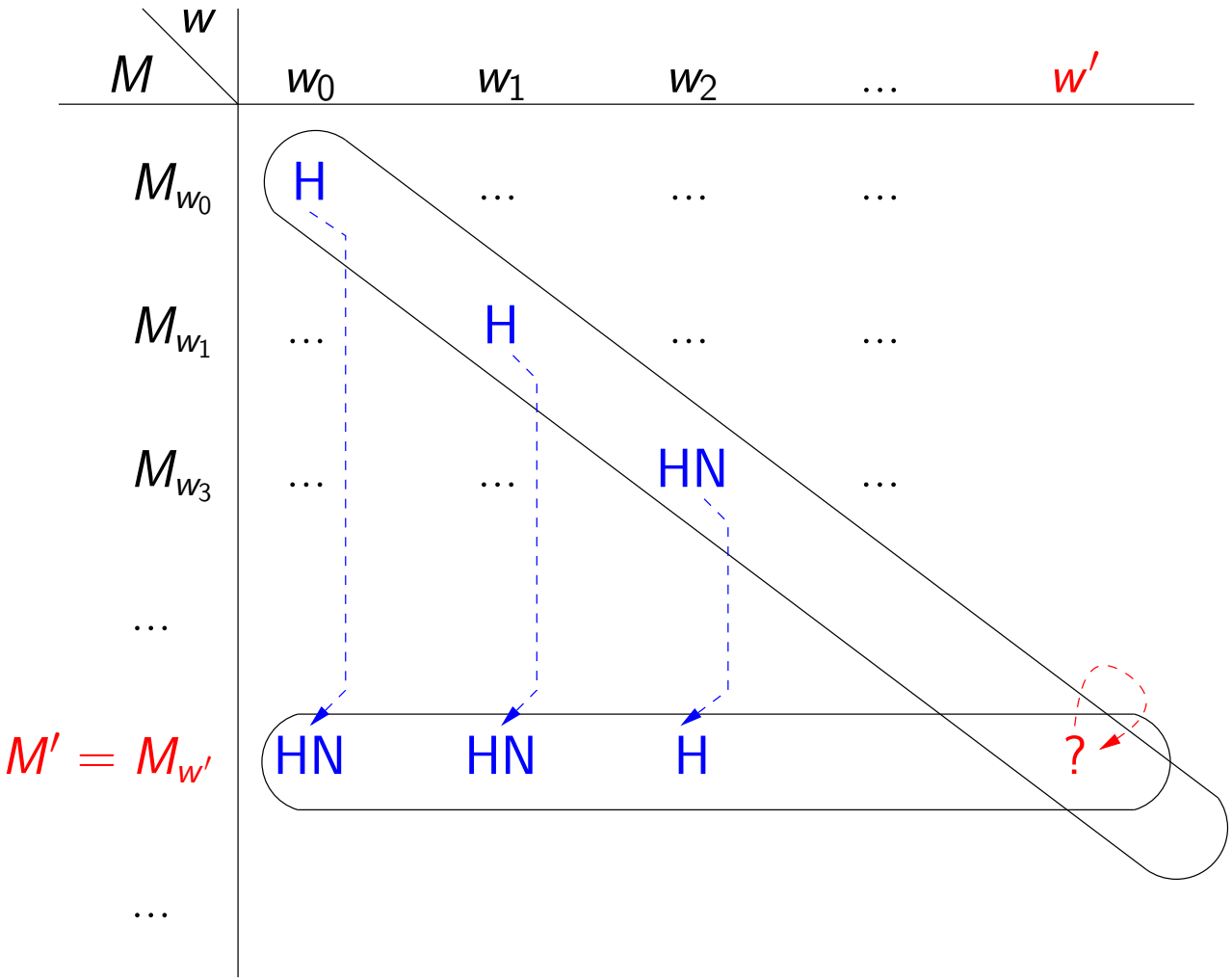
This is a diagonalization proof:

Let $w_0, w_1, w_2, \ldots$ be an enumeration of all words from $\{0,1\}^*$, for example $w_0 = \varepsilon$, $w_1 = 0$, $w_2 = 1$, $w_3 = 00$, ...

We record in a table: "How does $M_{w_i}$ behave on $w_j$?"

There are two possibilities: H ($M_{w_i}$ halts) or HN ($M_{w_i}$ does not halt).

# Undecidability of the Halting Problem



# Undecidability of the Halting Problem

The constructed Turing machine $M'$ and a $w'$ with $M' = M_{w'}$ are also entered into the table.

Due to the construction of $M'$: the diagonal entries determine the entries in the row of $M'$.

**Problem:** nothing fits in the place of the question mark!

$\rightsquigarrow$ There can be no Turing machine that solves the Halting Problem. $\square$

# Undecidability of the Halting Problem

**Theorem 20 (Semi-Decidability of the Special Halting Problem)**

The special Halting Problem $K$ is semi-decidable.

**Proof:**

The "partial" characteristic function $\chi'_K : \{0,1\}^* \to \{1\}$ can be computed as follows:

- On input $w$, we start the universal Turing machine $U$ with the input $w\#w$.
- If $U$ halts on input $w\#w$, the produced output is overwritten by 1.

□

# Reductions

We have now proven the undecidability of a problem, the special Halting Problem.

Further undecidability results should be derived from this.

This is done using arguments of the following kind:

1. If one could solve problem $B$, then one could also solve $A$. (Reduction step)
2. It follows that $B$ is harder or more general than $A$ ($A \leq B$).
3. We already know that $A$ is undecidable.
4. Therefore, the harder problem $B$ must also be undecidable.

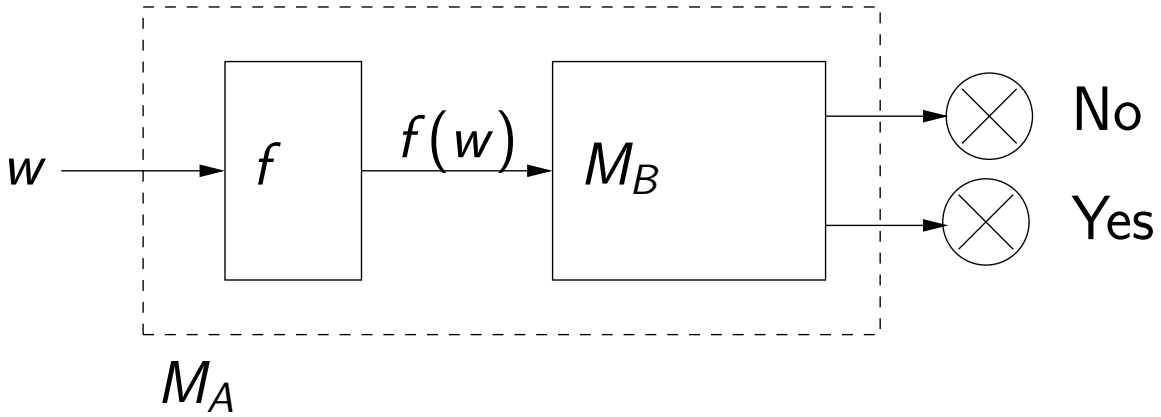
# Reductions

## Reduction/Reducibility (Definition)

Given languages $A \subseteq \Sigma^*$ and $B \subseteq \Gamma^*$, $A$ is said to be reducible to $B$ (denoted $A \leq B$) if there is a total and computable function $f : \Sigma^* \to \Gamma^*$ such that for all $w \in \Sigma^*$:

$$w \in A \iff f(w) \in B.$$

**Intuitively:** $A \leq B$ if you can construct a machine $M_A$ for $A$ from a machine $M_B$ for $B$ and a function $f$. That is, $M_B$ is called as a subroutine after preprocessing the input with $f$.



# Reductions

The following statement is then obvious:

## Lemma 21 (Reductions and Decidability)

Let $A \leq B$.

- If $B$ is decidable, then $A$ is also decidable.
- If $A$ is undecidable, then $B$ is also undecidable.

## Recipe for Showing the Undecidability of a Problem $B$

- Find a suitable problem $A$ known to be undecidable.

  So far, we only know the special halting problem $K$, but we will soon learn about other suitable problems.
- Find an appropriate function $f$ that reduces $A$ to $B$ and prove its correctness.
- It then follows immediately that $B$ is undecidable.

# Undecidability of the Halting Problem

**Satz 22 (Undecidability of the Halting Problem)**

The (general) halting problem $H$ is undecidable.

**Proof:**

Let the computable function $f$ be defined by $f(w) = w\#w$ for $w \in \{0,1\}^*$.

Then for all $w \in \{0,1\}^*$:

$$w \in K \iff w\#w \in H \iff f(w) \in H.$$

Thus, $K \leq H$.

Since $K$ is undecidable by Satz 20, $H$ must also be undecidable. □

# Undecidability of the Halting Problem

## Halting Problem on an Empty Tape (Definition)

The Halting Problem on an Empty Tape is the language

$$\begin{aligned} H_0 &= \{w \in \{0,1\}^* \mid M_w \text{ halts when started with an empty tape}\} \\ &= \{w \in \{0,1\}^* \mid \varepsilon \in T(M_w)\}. \end{aligned}$$

## Satz 23 (Undecidability of the Halting Problem on an Empty Tape)

The Halting Problem on an Empty Tape $H_0$ is undecidable.

**Proof:** We show $H \leq H_0$.

To each word $w\#x$ with $w, x \in \{0,1\}^*$, we assign a Turing machine $M(w\#x)$ that, when started on an empty tape, operates as follows:

1. Write $x$ on the tape.
2. Then simulate the machine $M_w$.

# Undecidability of the Halting Problem

It does not matter how the machine $M(w\#x)$ behaves on a non-empty tape.

We now define the function $f : \{0,1,\#\}^* \to \{0,1\}^*$ by the rule

$$f(w\#x) = \text{code}(M(w\#x)),$$

i.e., $M(w\#x) = M_{f(w\#x)}$ for all $w, x \in \{0,1\}^*$.

For words of the form $y \in \{0,1,\#\}^* \setminus \{0,1\}^*\#\{0,1\}^*$, let $f(y)$ be the code of a Turing machine $M_0$ that does not halt on an empty tape.

The function $f$ is then computable: Let $y \in \{0,1,\#\}^*$ be the input.

- If $y$ does not contain exactly one $\#$ (which can be checked with a DFA), the fixed word $\text{code}(M_0)$ is output.
- If $y = w\#x$ with $w, x \in \{0,1\}^*$, we algorithmically construct the Turing machine $M(w\#x)$ and then compute $\text{code}(M(w\#x))$.

# Undecidability of the Halting Problem

It holds that:

$$\begin{aligned} w\#x \in H &\iff M_w \text{ halts on input } x \\ &\iff M(w\#x) \text{ halts on an empty tape} \\ &\iff M_{f(w\#x)} \text{ halts on an empty tape} \\ &\iff f(w\#x) \in H_0 \end{aligned}$$

Furthermore, for all $y \in \{0,1,\#\}^* \setminus \{0,1\}^*\#\{0,1\}^*$:

$$y \notin H \text{ and } f(y) = \text{code}(M_0) \notin H_0.$$

Thus, $f$ indeed provides a reduction from $H$ to $H_0$. □

# Rice's Theorem

The next result shows that it is undecidable whether the function computed by a Turing machine $M$ has a certain property $\mathcal{S}$.

This means there is no method to make reliable statements about the functions computed by all Turing machines.

## Satz 24 (Rice's Theorem)

Let $\mathcal{R}$ be the class of all Turing-computable functions, and let $\mathcal{S}$ be any non-empty proper subset of $\mathcal{R}$ (i.e., $\mathcal{S} \neq \emptyset$ and $\mathcal{S} \neq \mathcal{R}$).
Then the language

$$C(\mathcal{S}) = \{w \in \{0,1\}^* \mid \text{the function computed by } M_w \text{ is in } \mathcal{S}\}$$

is undecidable.

# Rice's Theorem

**Proof:**

Let $\Omega$ be the function that is undefined everywhere.

Either $\Omega \in \mathcal{S}$ or $\Omega \notin \mathcal{S}$.

**Case 1:** $\Omega \in \mathcal{S}$

Since $\mathcal{S} \neq \mathcal{R}$, there exists a function $q \in \mathcal{R} \setminus \mathcal{S}$.

Let $Q$ be a Turing machine that computes $q$.

We now assign to each word $w \in \{0,1\}^*$ a Turing machine $M(w)$ that behaves as follows on input $y \in \{0,1\}^*$:

1. $M(w)$ initially ignores the input $y$ and simulates $M_w$ on an empty tape.
2. If this simulation eventually halts, then $M(w)$ simulates the machine $Q$ on $y$.

# Rice's Theorem

Then for the function $g$ computed by $M(w)$, we have:

$$g = \begin{cases} \Omega & \text{if } M_w \text{ does not halt on an empty tape, i.e., } w \notin H_0 \\ q & \text{otherwise, i.e., } w \in H_0 \end{cases}$$

The total function $f : \{0,1\}^* \to \{0,1\}^*$ given by

$$f(w) = \text{the code of the machine } M(w)$$

is obviously computable.

Note: $M(w) = M_{f(w)}$.

We obtain:

$$\begin{aligned} w \in H_0 &\implies g = q \\ &\implies \text{the function computed by } M_{f(w)} \text{ is not in } \mathcal{S} \\ &\implies f(w) \notin C(\mathcal{S}) \end{aligned}$$

# Rice's Theorem

Conversely:

$$\begin{aligned} w \notin H_0 &\implies g = \Omega \\ &\implies \text{the function computed by } M_{f(w)} \text{ is in } \mathcal{S} \\ &\implies f(w) \in C(\mathcal{S}) \end{aligned}$$

Thus, $w \in \overline{H_0} \iff f(w) \in C(\mathcal{S})$, i.e., $\overline{H_0} \leq C(\mathcal{S})$.

Since $H_0$ is undecidable by Theorem 23, $\overline{H_0}$ and thus $C(\mathcal{S})$ are undecidable.

**Case 2:** $\Omega \notin \mathcal{S}$

Since $\mathcal{S} \neq \emptyset$, there is a function $q \in \mathcal{S}$.

Let $Q$ be a Turing machine that computes $q$.

For $w \in \{0,1\}^*$, let the machine $M(w)$ and the computable total function $f : \{0,1\}^* \to \{0,1\}^*$ be defined exactly as in Case 1.

# Rice's Theorem

This time we get:

$$\begin{aligned} w \in H_0 &\implies g = q \\ &\implies \text{the function computed by } M_{f(w)} \text{ is in } \mathcal{S} \\ &\implies f(w) \in C(\mathcal{S}) \end{aligned}$$

Conversely:

$$\begin{aligned} w \notin H_0 &\implies g = \Omega \\ &\implies \text{the function computed by } M_{f(w)} \text{ is not in } \mathcal{S} \\ &\implies f(w) \notin C(\mathcal{S}) \end{aligned}$$

Thus, the undecidability of $C(\mathcal{S})$ follows as in Case 1. □

# Rice's Theorem

## Consequences of Rice's Theorem

The following problems are undecidable:

- Constant Function: $\{w \mid M_w \text{ computes a constant function}\}$
- Identity: $\{w \mid M_w \text{ computes the identity function}\}$
- Total Function: $\{w \mid M_w \text{ computes a total function}\}$
- Totally Undefined Function: $\{w \mid M_w \text{ computes } \Omega\}$

Rice's Theorem allows us to show undecidability for the properties of the function computed by a Turing machine, but not for other properties of a Turing machine (such as the number of its states or the tape alphabet).

**Consequence of Rice's Theorem on Program Verification:** No program can automatically verify the correctness of software.

# Rice's Theorem

Rice's Theorem and its variants also apply to other universal computation models.

**Satz 25 (Halting Problem for GOTO/WHILE Programs)**

For a given GOTO/WHILE program and initial values for the variables, it is undecidable whether the program halts on that input.

**Proof:**

The halting problem for Turing machines can be reduced to this problem. It suffices to translate the Turing machine into the corresponding GOTO/WHILE program and the machine's input into the corresponding variable assignments.

See the transformation "Turing Machine $\rightarrow$ GOTO Program" as the reduction function $f$. □

# Rice's Theorem

The following problem is already undecidable:

**Satz 26 (Halting Problem for GOTO Programs with Two Variables)**

For a given GOTO program with two variables, both initialized to 0, it is undecidable whether the program halts.

(Proof omitted)

For GOTO programs with only one variable, the halting problem is decidable, as one variable can be simulated by a pushdown automaton.

# Non-computable functions

Similar to the undecidability of problems, it can also be shown that certain functions are not computable. One example of this is the so-called Busy Beaver function.

## Busy Beaver

We consider all Turing machines with the two-element tape alphabet $\Gamma = \{1, \square\}$ and $n$ states. Of these machines, some halt on the empty tape, while others do not terminate.

The value of the Busy Beaver function $\Sigma$ at position $n$ is the maximum number of ones that can be written by a machine with $n$ states that terminates on the empty tape.

# Non-computable functions

The following is known about the Busy Beaver function $\Sigma\colon \mathbb{N} \to \mathbb{N}$:

- It is not computable.
- The following values of the function are known:

| $n$ | $\Sigma(n)$ |
|---|---|
| 1 | 1 |
| 2 | 4 |
| 3 | 6 |
| 4 | 13 |
| 5 | $\geq 4098$ |
| 6 | $\geq 1.29 \times 10^{865}$ |

The function value at position $n = 5$ has not yet been exactly determined.

# Undecidable Problems

We will now use the reduction proof technique to show the undecidability of the following problems:

- Post's Correspondence Problem (PCP)

  PCP: a combinatorial problem on words, an important (auxiliary) problem used to demonstrate the undecidability of other problems.
- Intersection Problem for Context-Free Grammars

# Post's Correspondence Problem

We now consider an important undecidable problem that is used to demonstrate the undecidability of many other problems:

## Post's Correspondence Problem (PCP)

- **Input:** A finite list of word pairs $I = ((x_1, y_1), \ldots, (x_k, y_k))$ with $x_i, y_i \in \Sigma^+$.

  Here, $\Sigma$ is an arbitrary alphabet.
- **Question:** Is there a sequence of indices $i_1, \ldots, i_n \in \{1, \ldots, k\}$ with $n \geq 1$ such that $x_{i_1} \cdots x_{i_n} = y_{i_1} \cdots y_{i_n}$?

A sequence $(i_1, \ldots, i_n)$ with $n \geq 1$, $i_1, \ldots, i_n \in \{1, \ldots, k\}$ such that $x_{i_1} \cdots x_{i_n} = y_{i_1} \cdots y_{i_n}$ is called a solution for the PCP input $I = ((x_1, y_1), \ldots, (x_k, y_k))$, and $x_{i_1} \cdots x_{i_n}$ is the solution word.

# Post's Correspondence Problem

**Example 1:** The following PCP input is solvable:

$$\begin{array}{llllll} x_1 & = & 0 & x_2 & = & 1 & x_3 & = & 0101 \\ y_1 & = & 010 & y_2 & = & 101 & y_3 & = & 01 \end{array}$$

A possible solution: $(3, 3, 1, 2)$:

$$\begin{array}{l} 01 \quad 01 \mid 010 \quad 1 \mid 0 \mid 1 \\ 01 \mid 01 \mid 010 \mid 1 \quad 0 \quad 1 \end{array}$$

Another (shorter) solution is: $(3, 1)$

# Post's Correspondence Problem

**Example 2:** The following PCP input is solvable:

$$\begin{array}{cccccccccccc} x_1 & = & 001 & x_2 & = & 01 & x_3 & = & 01 & x_4 & = & 10 \\ y_1 & = & 0 & y_2 & = & 011 & y_3 & = & 101 & y_4 & = & 001 \end{array}$$

A shortest solution consists of 66 indices:
$(2, 4, 3, 4, 4, 2, 1, 2, 4, 3, 4, 3, 4, 4, 3, 4, 4, 2, 1, 4, 4, 2, 1, 3, 4, 1, 1, 3,$
$4, 4, 4, 2, 1, 2, 1, 1, 1, 3, 4, 3, 4, 1, 2, 1, 4, 4, 2, 1, 4, 1, 1, 3, 4, 1, 1, 3,$
$1, 1, 3, 1, 2, 1, 4, 1, 1, 3)$.

The complexity of this solution already shows the difficulty of the problem.

# Post's Correspondence Problem

**Satz 27 (Semi-Decidability of PCP)**

The Post's Correspondence Problem is semi-decidable.

**Proof:**

Try all index sequences of length 1, then all index sequences of length 2, and so on.

If a matching index sequence is found at some point, output 1. □

# Post's Correspondence Problem

The first step of the undecidability proof is to consider the following modified problem.

## Modified PCP (MPCP)

- **Input:** $I$ as in PCP.
- **Question:** Is there a solution $(i_1, \ldots, i_n)$ for $I$ with $i_1 = 1$?

We now prove two reduction lemmas from which the undecidability of Post's Correspondence Problem follows:

## Lemma 28 (MPCP reducible to PCP)

MPCP $\leq$ PCP

# Post's Correspondence Problem

**Proof:**

Let $I = ((x_1, y_1), \ldots, (x_k, y_k))$ with $x_i, y_i \in \Sigma^+$ be a finite sequence of word pairs.

Let $\#$ and \$ be two new symbols.

For a word $w = a_1 a_2 \cdots a_n$ with $a_1, \ldots, a_n \in \Sigma$ and $n \geq 1$, we define the words ${}^{\#}w$, $w^{\#}$, ${}^{\#}w^{\#}$ as follows:

$$\begin{aligned} {}^{\#}w &= \#a_1\#a_2\#\cdots\#a_n \\ w^{\#} &= a_1\#a_2\#\cdots\#a_n\# \\ {}^{\#}w^{\#} &= \#a_1\#a_2\#\cdots\#a_n\# \end{aligned}$$

We now assign to the list $I$ the list

$$f(I) = (({}^{\#}x_1^{\#}, {}^{\#}y_1), (x_1^{\#}, {}^{\#}y_1), \ldots, (x_k^{\#}, {}^{\#}y_k), (\$, \#\$))$$

This consists of $k + 2$ pairs.

The function $f$ is clearly computable.

# Post's Correspondence Problem

We claim that $f$ provides a reduction from MPCP to PCP.

First, let $(i_1, i_2, \ldots, i_n)$ be a solution for $I$ with $i_1 = 1$ $(n \geq 1)$.

Then $(1, i_2 + 1, \ldots, i_n + 1, k + 2)$ is a solution for $f(I)$.

Now, let $(i_1, \ldots, i_n)$ be the shortest solution of $f(I)$.

Then it must hold that $i_1 = 1$, $i_2, \ldots, i_{n-1} \in \{2, \ldots, k + 1\}$, and $i_n = k + 2$.

Thus, $(1, i_2 - 1, \ldots, i_{n-1} - 1)$ is a solution for $I$. □

# Post's Correspondence Problem

**Lemma 29 (Halting Problem reducible to MPCP)**

$H \leq \text{MPCP}$

**Proof:**

Let $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ be a deterministic Turing machine (given by its encoding) and $w \in \Sigma^*$.

We will construct an MPCP input

$$I(M, w) = ((x_1, y_1), \ldots, (x_k, y_k)),$$

which is solvable if and only if $M$ halts on input $w$.

$I(M, w)$ will be defined over the alphabet $Z \cup \Gamma \cup \{\$, \#\}$ (w.l.o.g. assume $Z \cap \Gamma = \emptyset$).

# Post's Correspondence Problem

Then $I(M, w)$ looks as follows:

- 1st word pair:
  $(\$, \$\Box z_0 w \Box \#)$
- Copy pairs:
  $(a, a)$ for all $a \in \Gamma \cup \{\#\}$
- Transition pairs:
  $(za, z'c)$ if $\delta(z, a) = (z', c, N)$
  $(za, cz')$ if $\delta(z, a) = (z', c, R)$
  $(bza, z'bc)$ if $\delta(z, a) = (z', c, L)$ and $b \in \Gamma$
- Pair to add blanks at the ends if needed:
  $(\#, \Box\#)$ and $(\#, \#\Box)$
- Delete pairs:
  $(az_e, z_e)$ and $(z_e a, z_e)$ for all $z_e \in E$ and $a \in \Gamma$
- Final pair:
  $(z_e\#\#, \#)$ for all $z_e \in E$

# Post's Correspondence Problem

**Claim:** $M$ halts on input $w$ if and only if $I(M, w)$ is solvable.

Assume $M$ halts on input $w$.

Then there exists a sequence of configurations $k_0, k_1, \ldots, k_t \in \Gamma^+ Z \Gamma^+$ such that:

- $k_0 = \Box z_0 w \Box$
- $k_i \vdash_M k_{i+1}$ for all $0 \leq i \leq t-1$
- $k_t \in \Gamma^+ E \Gamma^+$

Then we obtain a solution for $I(M, w)$, where the solution word looks like:

$$\$k_0 \# k_1 \# \cdots \# k_t \# k_t' \# k_t'' \# k_t''' \cdots \# z_e \# \#.$$

Here, $k_t', k_t'', k_t''', \ldots$ are obtained from $k_t$ by deleting the symbol next to $z_e$ either to the left or to the right.

# Post's Correspondence Problem

Assume that $I(M, w)$ has a solution $(1, i_2, \ldots, i_t)$, which thus begins with $(\#, \#\square\square z_0 w \square \#)$.

As long as the "partial solution" $(x_1 x_{i_2} \cdots x_{i_n}, y_1 y_{i_2} \cdots y_{i_n})$ does not yet contain an accepting state $z_e \in E$ in $y_1 y_{i_2} \cdots y_{i_n}$ (the longer word), the computation of $M$ on input $w$ must be correctly simulated using the copy pairs, transition pairs, and the pair for adding blanks.

However, since we have a finite solution $(1, i_2, \ldots, i_t)$, there must be some $m \leq t$ such that an accepting state $z_e \in E$ occurs in $y_1 y_{i_1} \cdots y_{i_m}$.

Thus, $M$ halts on input $w$. □

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash bz_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$$\$$
$$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \#$$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash bz_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box$

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash bz_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e\#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a$

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ \ z_1$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash bz_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a \ \ b$
$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ \ z_1 \ b$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash bz_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$(\$, \$\Box z_0 ab\Box\#)$ $(x, x)$ $(z_0 a, bz_1)$ $(yz_1 b, z_2 yc)$ $(z_2 b, z_e c)$

$(\#, \Box\#)$ $(\#, \#\Box)$ $(yz_e, z_e)$ $(z_e y, z_e)$ $(z_e\#\#, \#)$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a \ \ b \ \Box$

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ \ z_1 \ b \ \Box$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash b z_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, b z_1) \quad (y z_1 b, z_2 y c) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (y z_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \#$
$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ \ z_1 \ b \ \Box \ \#$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash bz_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e\#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box$
$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ \ z_1 \ b \ \Box \ \# \ \Box$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash bz_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ \ z_1 \ b$
$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash bz_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ \ z_1 \ b \ \Box$
$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c \ \Box$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash b z_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, b z_1) \quad (y z_1 b, z_2 y c) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (y z_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \#$
$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c \ \Box \ \#$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash b z_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, b z_1) \quad (y z_1 b, z_2 y c) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (y z_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box$
$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c \ \Box \ \# \ \Box$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash bz_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b$
$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c \ \Box \ \# \ \Box \ z_e \ c$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash b z_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, b z_1) \quad (y z_1 b, z_2 y c) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (y z_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c$
$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c \ \Box \ \# \ \Box \ z_e \ c \ \ c$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash b z_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, b z_1) \quad (y z_1 b, z_2 y c) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (y z_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c \ \Box$
$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c \ \Box \ \# \ \Box \ z_e \ c \ \ c \ \Box$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash bz_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c \ \Box \ \#$
$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c \ \Box \ \# \ \Box \ z_e \ c \ \ c \ \Box \ \#$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash bz_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c \ \Box \ \# \ \Box \ z_e$
$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c \ \Box \ \# \ \Box \ z_e \ c \ \ c \ \Box \ \# \ z_e$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash bz_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c \ \Box \ \# \ \Box \ z_e \ c$
$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c \ \Box \ \# \ \Box \ z_e \ c \ \ c \ \Box \ \# \ z_e \ c$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash bz_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$\ \Box\ z_0\ a\ \ b\ \Box\ \#\ \Box\ b\ \ z_1\ b\ \Box\ \#\ \Box\ z_2\ b\ \ c\ \Box\ \#\ \Box\ z_e\ c\ \ c$
$\$\ \Box\ z_0\ a\ \ b\ \Box\ \#\ \Box\ b\ \ z_1\ b\ \Box\ \#\ \Box\ z_2\ b\ \ c\ \Box\ \#\ \Box\ z_e\ c\ \ c\ \Box\ \#\ z_e\ c\ \ c$

# Post's Correspondence Problem

**Example:** Consider a Turing machine $M$ with states $z_0, z_1, z_2, z_e$, tape symbols $a, b, c, \Box$, and the following transitions:

$$\delta(z_0, a) = (z_1, b, R) \quad \delta(z_1, b) = (z_2, c, L) \quad \delta(z_2, b) = (z_e, c, N)$$

with $z_e \in E$. Then there is the following accepting computation on input $ab$:

$$z_0 ab \vdash b z_1 b \vdash z_2 bc \vdash z_e cc$$

$I(M, ab)$ consists of the following pairs for all $x \in \{a, b, c, \Box, \#\}$ and $y \in \{a, b, c, \Box\}$

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, b z_1) \quad (y z_1 b, z_2 y c) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (y z_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c \ \Box \ \# \ \Box \ z_e \ c \ \ c \ \Box$

$\$ \ \Box \ z_0 \ a \ \ b \ \Box \ \# \ \Box \ b \ z_1 \ b \ \Box \ \# \ \Box \ z_2 \ b \ \ c \ \Box \ \# \ \Box \ z_e \ c \ \ c \ \Box \ \# \ z_e \ c \ \ c \ \Box$

# Post's Correspondence Problem

**Example (continued)**

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e\#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\cdots \Box$

$\cdots \Box \# z_e\ c\ \ c\ \Box$

# Post's Correspondence Problem

**Example (continued)**

$$(\$, \$\Box z_0 ab \Box \#) \quad (x, x) \quad (z_0 a, b z_1) \quad (y z_1 b, z_2 y c) \quad (z_2 b, z_e c)$$
$$(\#, \Box \#) \quad (\#, \#\Box) \quad (y z_e, z_e) \quad (z_e y, z_e) \quad (z_e \# \#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\cdots \Box \#$

$\cdots \Box \# \; z_e \; c \;\; c \;\; \Box \#$

# Post's Correspondence Problem

**Example (continued)**

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e\#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\cdots \Box \# z_e\ c$

$\cdots \Box \# z_e\ c\ \ c\ \Box \# z_e$

# Post's Correspondence Problem

**Example (continued)**

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e\#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\cdots \Box \# z_e c \quad c$
$\cdots \Box \# z_e c \quad c \quad \Box \# z_e c$

# Post's Correspondence Problem

**Example (continued)**

$(\$, \$\Box z_0 ab \Box \#)$ $(x, x)$ $(z_0 a, b z_1)$ $(y z_1 b, z_2 y c)$ $(z_2 b, z_e c)$

$(\#, \Box \#)$ $(\#, \#\Box)$ $(y z_e, z_e)$ $(z_e y, z_e)$ $(z_e \# \#, \#)$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\cdots \Box \# z_e c \;\; c \;\; \Box$

$\cdots \Box \# z_e c \;\; c \;\; \Box \# z_e c \;\; \Box$

# Post's Correspondence Problem

**Example (continued)**

$$(\$, \$\Box z_0 ab \Box \#) \quad (x, x) \quad (z_0 a, b z_1) \quad (y z_1 b, z_2 y c) \quad (z_2 b, z_e c)$$
$$(\#, \Box \#) \quad (\#, \#\Box) \quad (y z_e, z_e) \quad (z_e y, z_e) \quad (z_e \# \#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\cdots \Box \# z_e c \;\; c \;\; \Box \#$
$\cdots \Box \# z_e c \;\; c \;\; \Box \# z_e c \;\; \Box \#$

# Post's Correspondence Problem

**Example (continued)**

$$(\$, \$\Box z_0 ab \Box \#) \quad (x, x) \quad (z_0 a, b z_1) \quad (y z_1 b, z_2 y c) \quad (z_2 b, z_e c)$$
$$(\#, \Box \#) \quad (\#, \#\Box) \quad (y z_e, z_e) \quad (z_e y, z_e) \quad (z_e \#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\cdots \Box \# z_e c \;\; c \;\; \Box \# z_e c$
$\cdots \Box \# z_e c \;\; c \;\; \Box \# z_e c \;\; \Box \# z_e$

# Post's Correspondence Problem

**Example (continued)**

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e\#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$\cdots \Box \# z_e c \;\; c \;\; \Box \# z_e c \;\; \Box$
$\cdots \Box \# z_e c \;\; c \;\; \Box \# z_e c \;\; \Box \# z_e \Box$

# Post's Correspondence Problem

**Example (continued)**

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e\#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$$\cdots \Box \# z_e c \;\; c \;\; \Box \# z_e c \;\; \Box \#$$
$$\cdots \Box \# z_e c \;\; c \;\; \Box \# z_e c \;\; \Box \# z_e \Box \#$$

# Post's Correspondence Problem

**Example (continued)**

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e\#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$$\cdots \Box \# z_e c \;\; c \;\; \Box \# z_e c \;\; \Box \# z_e \Box$$
$$\cdots \Box \# z_e c \;\; c \;\; \Box \# z_e c \;\; \Box \# z_e \Box \# z_e$$

# Post's Correspondence Problem

**Example (continued)**

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e\#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$$\cdots \Box \# z_e c \;\; c \;\; \Box \# z_e c \;\; \Box \# z_e \Box \#$$
$$\cdots \Box \# z_e c \;\; c \;\; \Box \# z_e c \;\; \Box \# z_e \Box \# z_e \#$$

# Post's Correspondence Problem

**Example (continued)**

$$(\$, \$\Box z_0 ab\Box\#) \quad (x, x) \quad (z_0 a, bz_1) \quad (yz_1 b, z_2 yc) \quad (z_2 b, z_e c)$$
$$(\#, \Box\#) \quad (\#, \#\Box) \quad (yz_e, z_e) \quad (z_e y, z_e) \quad (z_e\#\#, \#)$$

A solution to this PCP, corresponding to the above accepting computation, then builds up as follows:

$$\cdots \Box \# z_e c \;\; c \;\; \Box \# z_e c \;\; \Box \# z_e \Box \# z_e \# \#$$
$$\cdots \Box \# z_e c \;\; c \;\; \Box \# z_e c \;\; \Box \# z_e \Box \# z_e \# \#$$

# Post's Correspondence Problem

**Theorem 30 (PCP undecidable)**

The Post Correspondence Problem is undecidable.

**Proof:** The claim follows directly from the two previous lemmas:

From $H \leq \text{MPCP} \leq \text{PCP}$ it follows that $H \leq \text{PCP}$ (by composition of the reduction mappings).

Since the general halting problem $H$ is also undecidable, PCP is also undecidable. □

# Post's Correspondence Problem

**Remarks:**

Let $\mathrm{PCP}_{m,n}$ be the restriction of the PCP to inputs of the form $((x_1, y_1), \ldots, (x_k, y_k))$ with $k \leq m$, $x_1, y_1, \ldots, x_k, y_k \in \{a_1, \ldots, a_n\}^+$ (i.e., $n$-element alphabet and at most $m$ word pairs)

- Already $\mathrm{PCP}_{5,2}$ is undecidable.
  (Turlough Neary 2015, `http://drops.dagstuhl.de/opus/frontdoor.php?source_opus=4948`
- $\mathrm{PCP}_{m,1}$ and $\mathrm{PCP}_{2,n}$ are decidable (for arbitrary $m$ and $n$).
- It is unknown whether $\mathrm{PCP}_{k,2}$ for $k \in \{3, 4\}$ is decidable.

# Intersection Problem for Context-Free Grammars

We will now use the PCP to show the undecidability of the intersection problem for context-free grammars.

## Intersection Problem for Context-Free Grammars

- **Input:** two context-free grammars $G_1$, $G_2$.
- **Question:** Is $L(G_1) \cap L(G_2) \neq \emptyset$, i.e., is there a word that is generated by both $G_1$ and $G_2$?

## Theorem 31 (Intersection Problem Undecidable)

The intersection problem for context-free grammars is undecidable.

# Intersection Problem for Context-Free Grammars

**Proof:**

Based on Theorem 30 (PCP Undecidable), it suffices to show that PCP is reducible to the intersection problem for context-free grammars.

Let $I = ((x_1, y_1), \ldots, (x_k, y_k))$ be an arbitrary PCP instance with $x_1, y_1, \ldots, x_k, y_k \in \Sigma^+$.

Let $\Gamma = \Sigma \cup \{\$, 1, \ldots, k\}$.

We now define two context-free grammars $G_1$ and $G_2$ over the terminal alphabet $\Gamma$.

Productions of $G_1$ (with $S$ as the start symbol):

$$\begin{array}{lcl} S & \to & A\,\$\,B \\ A & \to & 1\,A\,x_1 \mid 1\,x_1 \mid \cdots \mid k\,A\,x_k \mid k\,x_k \\ B & \to & y_1^{\text{rev}}\,B\,1 \mid y_1^{\text{rev}}\,1 \mid \cdots \mid y_k^{\text{rev}}\,B\,k \mid y_k^{\text{rev}}\,k \end{array}$$

# Intersection Problem for Context-Free Grammars

Here, $w^{\text{rev}}$ is the word $w$ read from right to left.

Then it holds that:

$$L(G_1) = \{i_n \cdots i_1 x_{i_1} \cdots x_{i_n} \$ (y_{j_1} \cdots y_{j_m})^{\text{rev}} j_1 \cdots j_m \mid n, m \geq 1, 1 \leq i_1, \ldots, i_n, j_1, \ldots, j_m \leq k\}.$$

Productions of $G_2$ (with $S$ as the start symbol):

$$\begin{aligned} S &\to 1S1 \mid \cdots kSk \mid T \\ T &\to aTa \text{ for all } a \in \Sigma \mid \$ \end{aligned}$$

Then it holds that: $L(G_2) = \{uv\$v^{\text{rev}}u^{\text{rev}} \mid u \in \{1, \ldots, k\}^*, v \in \Sigma^*\}$.

Thus we have: $I$ is solvable $\iff$ $L(G_1) \cap L(G_2) \neq \emptyset$. □

# Intersection Problem for Context-Free Grammars

In the previous proof, solutions to the PCP input $I$ correspond exactly to the words in $L(G_1) \cap L(G_2)$.

Now, for any PCP input $J$: $J$ is solvable if and only if $J$ has infinitely many solutions (if $(i_1, \ldots, i_k)$ is a solution, then $(i_1, \ldots, i_k, i_1, \ldots, i_k)$ is also a solution).

Thus, $I$ is solvable if and only if $L(G_1) \cap L(G_2)$ is infinite.

We obtain:

**Satz 32**

It is undecidable whether for given context-free grammars $G_1$ and $G_2$ the intersection $L(G_1) \cap L(G_2)$ is infinite.

# Intersection Problem for Context-Free Grammars

It is easy to provide a context-free grammar $G_2'$ for the language $\Gamma^* \setminus L(G_2)$ (Exercise).

Let $G_3$ be a context-free grammar for $L(G_1) \cup L(G_2')$.

Then we have:

$$\begin{aligned} L(G_1) \cap L(G_2) = \emptyset \quad &\iff \quad L(G_1) \subseteq L(G_2') \\ &\iff \quad L(G_1) \cup L(G_2') = L(G_2') \\ &\iff \quad L(G_3) = L(G_2') \end{aligned}$$

We obtain:

**Satz 33**

It is undecidable whether for given context-free grammars $G_1$ and $G_2$ it holds that:

- $L(G_1) \subseteq L(G_2)$
- $L(G_1) = L(G_2)$

# Intersection Problem for Context-Free Grammars

Finally, it can be shown that the grammars $G_1$, $G_2$, and $G_2'$ generate deterministic context-free languages, and it is possible to construct equivalent deterministic pushdown automata $A_1$, $A_2$, and $A_2'$ from $G_1$, $G_2$, and $G_2'$.

Thus we obtain:

**Satz 34**

It is undecidable whether for given deterministic pushdown automata $A_1$ and $A_2$ it holds that:

- $T(A_1) \cap T(A_2) \neq \emptyset$
- $T(A_1) \cap T(A_2)$ is infinite.
- $T(A_1) \subseteq T(A_2)$

# Intersection Problem for Context-Free Grammars

**Remark 1**: The language $L(G_1) \cup L(G_2')$ constructed on slide 195 is not necessarily deterministic context-free (the class of deterministic context-free languages is not closed under union).

In fact, it is decidable whether $T(A_1) = T(A_2)$ for two given deterministic pushdown automata $A_1$ and $A_2$ (Senizergues 1997).

**Remark 2:** The intersection problem for context-free languages is semi-decidable:

More generally: The set $\{(u, v) \mid u, v \in \{0, 1\}^*, T(M_u) \cap T(M_v) \neq \emptyset\}$ is semi-decidable, i.e., the intersection problem for Type-0 languages is semi-decidable:

The languages $T(M_u)$ and $T(M_v)$ are recursively enumerable.

Enumerate the languages $T(M_u)$ and $T(M_v)$ "in parallel".

# Emptiness of Context-Sensitive Languages

Terminate with output 1 if at any point a word $w$ appears in both enumerations.

Consequence: The complement of the intersection problem is not semi-decidable. Otherwise, it would be decidable (Slide 115).

**Theorem 35 (Emptiness of Type-1 Grammars Undecidable)**

It is undecidable whether for a given Type-1 grammar $G$ (or alternatively a linear bounded automaton) it holds that $L(G) \neq \emptyset$.

**Proof:**

We reduce the intersection problem for context-free grammars to the emptiness problem for Type-1 grammars.

With Theorem 31, this proves the theorem.

Let $G_1$ and $G_2$ be two context-free grammars.

These are in particular of Type-1.

# Emptiness of Context-Sensitive Languages

Since Type-1 languages are effectively closed under intersection, we can construct a Type-1 grammar $G$ from $G_1$ and $G_2$ with $L(G) = L(G_1) \cap L(G_2)$.

To elaborate: Construct two linear bounded automata $A_1$ and $A_2$ from $G_1$ and $G_2$ with $L(G_1) = T(A_1)$ and $L(G_2) = T(A_2)$ (see the construction in the proof of Kuroda's theorem (FSA, Slide 344)).

From $A_1$ and $A_2$, one can easily construct a linear bounded automaton $A$ with $T(A) = T(A_1) \cap T(A_2)$.

$A$ can then again be transformed into an equivalent Type-1 grammar using the construction in the proof of Kuroda's theorem. □

# Hilbert's 10th Problem

Another undecidable problem (without proof):

We consider polynomials $p(x_1, \ldots, x_n)$ (in multiple variables) with coefficients from $\mathbb{Z}$.

**Example**: $p(x_1, x_2, x_3, x_4) = -5x_1^2 x_3^4 x_4 + 3x_2^8 x_3^2 x_4^3 - 8x_1 x_2^6 + 17x_4 - 25$.

**Hilbert's 10th Problem is Undecidable (Matiyasevich 1970)**

The following problem is undecidable:

INPUT: A polynomial $p(x_1, \ldots, x_n)$ (in multiple variables) with coefficients from $\mathbb{Z}$.

QUESTION: Do there exist $a_1, \ldots, a_n \in \mathbb{Z}$ such that $p(a_1, \ldots, a_n) = 0$?

## Deterministic Time Classes

Let $f : \mathbb{N} \to \mathbb{N}$ be a monotonic function. The class DTIME($f$) consists of all languages $L$ for which there exists a deterministic Turing machine $M$ such that:

- $M$ computes the characteristic function of $L$.
- For every input $w \in \Sigma^*$, $M$ reaches a final state from the start configuration $z_0 w \Box$ in at most $f(|w|)$ computation steps (and outputs 0 or 1, depending on whether $w \notin L$ or $w \in L$).

Poly denotes the set of all functions on $\mathbb{N}$ described by a polynomial with coefficients from $\mathbb{N}$ (e.g., $n, 2n, n^2 + 3n, n^{10000}$).

## The Class P

$$\mathrm{P} = \bigcup_{f \in \mathrm{Poly}} \mathrm{DTIME}(f)$$

**Remarks:**

- P is often considered the set of all efficiently solvable problems.
- The class P is relatively robust against changes in the computational model. For example, the class P does not change if we allow multi-tape Turing machines instead of normal Turing machines (which would also be somewhat more realistic, as using single-tape Turing machines requires copying a lot of information).
- If we define P in terms of WHILE or GOTO programs, we must measure the time required for an assignment (e.g., $x_i := x_j + 1$) as the current number of bits of $x_j$ (so approximately $\log(x_j)$): logarithmic cost.

- If one were to use the uniform cost model (i.e., an assignment is counted as one step), then, for example, the following algorithm would be polynomial:

  INPUT $n$;
  $x := 2$;
  LOOP $n$ DO $x := x * x$ END;
  OUTPUT $(x)$

  This algorithm computes the number $2^{2^n}$, and writing this number in binary representation already requires $2^n$ bits.

  In this example, however, we have cheated a bit since we use multiplication as a basic operation. If we replace $x := x * x$ with a (standard) Loop program, the running time of the above algorithm becomes exponential.

## Nondeterministic Time Classes

Let $f : \mathbb{N} \to \mathbb{N}$ be a monotone function. The class NTIME($f$) consists of all languages $L$ for which there exists a nondeterministic Turing machine $M$ such that:

- For every input $w \in \Sigma^*$, $M$ reaches an accepting state from the initial configuration $z_0 w \Box$ on every computation path in at most $f(|w|)$ steps and outputs 0 or 1.
- It holds that $w \in L$ if and only if $M$ outputs 1 on at least one computation path.

## The Class NP

$$\text{NP} = \bigcup_{f \in \text{Poly}} \text{NTIME}(f)$$

**Remarks:**

- Clearly, $\mathrm{P} \subseteq \mathrm{NP}$.
- Whether $\mathrm{P} = \mathrm{NP}$ is considered the most important open question in theoretical computer science. It is generally suspected that $\mathrm{P} \neq \mathrm{NP}$.
- Why is the question of $\mathrm{P} = \mathrm{NP}$ so interesting?

  It is known that a multitude of problems lie in NP, but it is unknown whether they lie in P.

  There is even a large class of problems (the NP-complete problems, more on this shortly) of which it is known that if one of these problems belongs to P, then $\mathrm{P} = \mathrm{NP}$.
- It is not hard to see that all languages in NP are $\textsc{Loop}$-decidable (i.e., the characteristic functions of languages from NP are $\textsc{Loop}$-computable).

**Example: SUBSETSUM**

**SUBSETSUM**

INPUT: Binary-encoded numbers $t, w_1, \ldots, w_n$
QUESTION: Is there a subset $U \subseteq \{w_1, \ldots, w_n\}$ such that $t = \sum_{w \in U} w$?

**Theorem 36**

SUBSETSUM $\in$ NP

**Remark:** This problem belongs to P if the numbers $t, w_1, \ldots, w_n$ are encoded in unary.

In unary encoding, the number $n$ is represented by the word $a^n$ (for a symbol $a$).

Reductions, as we learned in the section on (Un)decidability (Slide 137), are not very informative for decidable problems:

**Lemma 37**

Let $A, B \subseteq \Sigma^*$ be decidable languages with $\emptyset \neq B \neq \Sigma^*$. Then $A \leq B$.

Choose two elements $x \in B$ and $y \in \Sigma^* \setminus B$.

Define the function $f : \Sigma^* \to \Sigma^*$ by

$$f(w) = \begin{cases} x & \text{if } w \in A \\ y & \text{if } w \notin A \end{cases}$$

Since $A$ is decidable, $f$ is computable, and it holds that $w \in A \iff f(w) \in B$, i.e., $A \leq B$.

## Polynomial Reducibility

A function $f : \Sigma^* \to \Gamma^*$ is polynomially computable if there exists a deterministic Turing machine $M$ and a polynomial $p(n)$ such that for all $w \in \Sigma^*$:

When $M$ is started with input $w$, it halts after at most $p(|w|)$ steps with the output $f(w)$ on the work tape.

A language $A \subseteq \Sigma^*$ is polynomially reducible to a language $B \subseteq \Gamma^*$ (denoted $A \leq_p B$) if there exists a polynomially computable function $f : \Sigma^* \to \Gamma^*$ such that

$$\forall w \in \Sigma^* : w \in A \iff f(w) \in B.$$

**Lemma 38**

If $A \leq_p B$ and $B \in \mathrm{P}$ (or $B \in \mathrm{NP}$), then it follows that $A \in \mathrm{P}$ (or $A \in \mathrm{NP}$).

**Proof:**

Assume first that $A \leq_p B$ and $B \in \mathrm{P}$.

Then there exist polynomials $p(n)$ and $q(n)$ as well as Turing machines $M$ and $N$ with the following properties:

- $M$ computes from an input $w \in \Sigma^*$ in time $p(|w|)$ a word $f(w)$ such that: $w \in A \iff f(w) \in B$.

  Note: Since machine $M$ can only produce an output of length at most $p(|w|)$ in $p(|w|)$ steps, it holds that $|f(w)| \leq p(|w|)$.

- $N$ accepts the language $B$ in time $q(n)$.

A Turing machine for the language $A$ operates on an input $w$ as follows:

1. Compute $f(w)$ (time requirement: $p(|w|)$).
2. Simulate machine $N$ on $f(w)$ (time requirement: $q(p(|w|))$).

Thus, the total time requirement is $p(|w|) + q(p(|w|))$, which is again a polynomial.

The statement for the class NP can be proven in the same way. □

## NP-Completeness

A language $A$ is NP-hard if for all $B \in \text{NP}$ it holds: $B \leq_p A$
($A$ is at least as hard as any problem in NP).

A language $A$ is NP-complete if it belongs to NP and is NP-hard.

Intuition: NP-complete languages are the hardest languages in NP.

We still do not know if there are any NP-complete languages at all. We will show this soon.

First, a simple result:

**Lemma 39**

If $A$ is NP-complete, then it holds that: $\mathrm{P} = \mathrm{NP} \iff A \in \mathrm{P}$.

**Proof:**

$\Rightarrow$: Assume $\mathrm{P} = \mathrm{NP}$.

Since $A$ is NP-complete, it follows that $A \in \mathrm{NP} = \mathrm{P}$.

$\Leftarrow$: Assume $A \in \mathrm{P}$ and let $B \in \mathrm{NP}$ be arbitrary.

Since $A$ is NP-complete, it follows that $B \leq_p A \in \mathrm{P}$.

Lemma 38 implies $B \in \mathrm{P}$.

Thus, it holds that $\mathrm{NP} \subseteq \mathrm{P}$ and consequently $\mathrm{NP} = \mathrm{P}$. □

Soon we will get to know a concrete NP-complete problem: the satisfiability problem for propositional logic formulas (SAT).

For many other problems $A$, NP-completeness can then be shown by a reduction SAT $\leq_p A$.

Here are some examples of NP-complete problems
(without proof; see Schöning for proofs).

**SUBSETSUM**

INPUT: Binary-encoded numbers $t, w_1, \ldots, w_n$
QUESTION: Is there a subset $U \subseteq \{w_1, \ldots, w_n\}$ such that $t = \sum_{w \in U} w$?

## CLIQUE

INPUT: An undirected graph $G = (V, E)$ (see the lecture on *Discrete Mathematics*) and a number $k$ (unary encoded)

QUESTION: Does $G$ have a clique of size $k$, i.e., is there a set $U \subseteq V$ with $|U| \geq k$ such that for all $u, v \in U$ with $u \neq v$: $\{u, v\} \in E$?

## VERTEX-COVER

INPUT: An undirected graph $G = (V, E)$ and a number $k$ (unary encoded)

QUESTION: Does $G$ have a vertex cover of size $k$, i.e., is there a set $U \subseteq V$ with $|U| \leq k$ such that for all $\{u, v\} \in E$ it holds that $U \cap \{u, v\} \neq \emptyset$?

## 3-COLORABILITY

INPUT: An undirected graph $G = (V, E)$

QUESTION: Is the chromatic number of $G$ at most 3?

## HAMILTON-CIRCUIT

INPUT: An undirected graph $G = (V, E)$

QUESTION: Does $G$ have a Hamiltonian circuit (see the lecture on *Discrete Mathematics*)?

# Logic

We will return to complexity theory later (and show that SAT is NP-complete).

First, however, we want to focus on logic.

Intuitively speaking, a logic is a language through which formal facts (e.g., statements from mathematics, correctness claims for programs, etc.) can be formulated.

Partly, we have already used such logical statements.

We will learn about two important (arguably the most important) logics:

- Propositional Logic
- Predicate Logic

# Logic

The approach to introducing a new logic is always the same:

- First, we define the syntax of the logic. In doing so, we define a language of syntactically correct formulas.
- Next, we define the semantics of the logic, i.e., we define when a formula is true or false.

# Syntax of Propositional Logic

An atomic formula has the form $A_i$ (where $i = 1, 2, 3, \ldots$).
Formulas are defined by the following inductive process:

1. All atomic formulas are formulas.
2. If $F$ and $G$ are formulas, then $(F \wedge G)$ and $(F \vee G)$ are also formulas.
3. If $F$ is a formula, then $\neg F$ is also a formula.

**Terminology:**

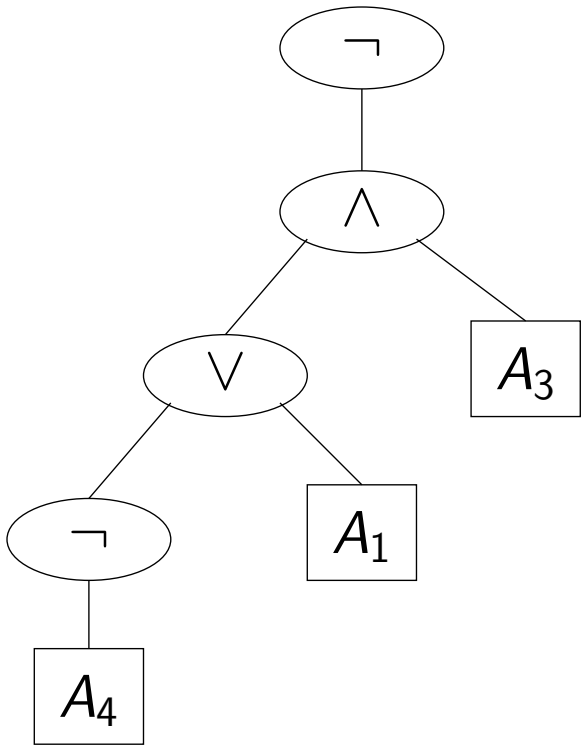
- $(F \wedge G)$: $F$ and $G$, the conjunction of $F$ and $G$
- $(F \vee G)$: $F$ or $G$, the disjunction of $F$ and $G$
- $\neg F$: not $F$, the negation of $F$

**Example:** $\neg((\neg A_4 \vee A_1) \wedge A_3)$ is a formula.

# Formula as Syntax Tree

Each formula can also be represented by an syntax tree.

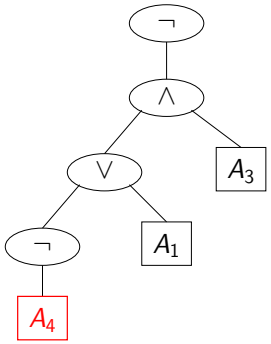
**Example:** $F = \neg((\neg A_4 \vee A_1) \wedge A_3)$



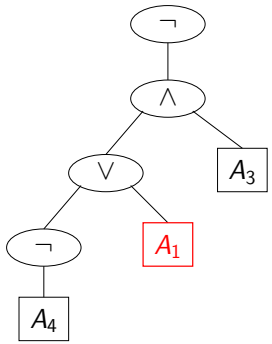
# Subformulas

The subformulas of a formula $F$ correspond to the subtrees.

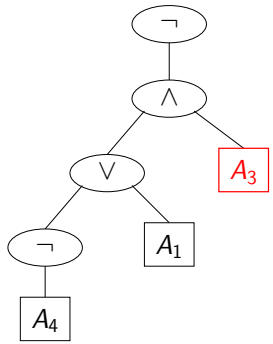
$A_4$



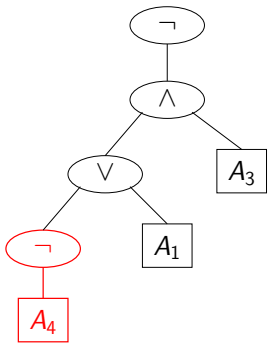

$A_1$



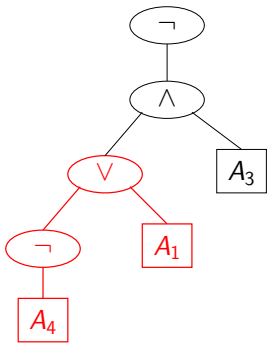

$A_3$



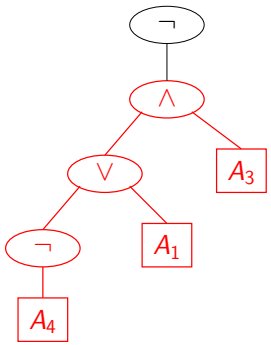

$\neg A_4$



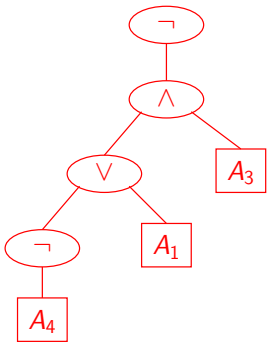

$(\neg A_4 \vee A_1)$




$((\neg A_4 \vee A_1) \wedge A_3)$




$\neg((\neg A_4 \vee A_1) \wedge A_3)$



# Semantics of Propositional Logic: Assignments

The elements of the set $\{0, 1\}$ are called truth values.

An assignment is a function $\mathcal{B}\colon D \to \{0, 1\}$, where $D \subseteq \{A_1, A_2, A_3, \ldots\}$ is a subset of the atomic formulas.

On the next slide, we will extend $\mathcal{B}$ to a function $\widehat{\mathcal{B}}\colon E \to \{0, 1\}$, where $E \supseteq D$ is the set of all formulas that are constructed only from the atomic formulas in $D$.

**Example:** Let $D = \{A_1, A_5, A_8\}$.

Then $F = \neg((\neg A_5 \vee A_1) \wedge A_8) \in E$ but $\neg((\neg A_4 \vee A_1) \wedge A_3) \notin E$.

A possible truth assignment could be defined by: $\mathcal{B}(A_1) = 1$, $\mathcal{B}(A_5) = 0$, $\mathcal{B}(A_8) = 1$.

Question: What is $\widehat{\mathcal{B}}(F)$?

# Semantics of Propositional Logic: Assignments

$$
\begin{aligned}
\widehat{\mathcal{B}}(A) &= \mathcal{B}(A) \quad \text{if } A \in D \text{ is an atomic formula} \\
\widehat{\mathcal{B}}((F \wedge G)) &= \begin{cases} 1 & \text{if } \widehat{\mathcal{B}}(F) = 1 \text{ and } \widehat{\mathcal{B}}(G) = 1 \\ 0 & \text{otherwise} \end{cases} \\
\widehat{\mathcal{B}}((F \vee G)) &= \begin{cases} 1 & \text{if } \widehat{\mathcal{B}}(F) = 1 \text{ or } \widehat{\mathcal{B}}(G) = 1 \\ 0 & \text{otherwise} \end{cases} \\
\widehat{\mathcal{B}}(\neg F) &= \begin{cases} 1 & \text{if } \widehat{\mathcal{B}}(F) = 0 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

From now on, we will write $\mathcal{B}$ instead of $\widehat{\mathcal{B}}$.

# Truth Tables for $\wedge$, $\vee$, and $\neg$

Calculation of $\mathcal{B}$ using truth tables.

**Observation:** The value of $\mathcal{B}(F)$ only depends on how $\mathcal{B}$ is defined on the atomic formulas occurring in $F$.

Tables for the operators $\vee$, $\wedge$, $\neg$:

| $A$ | $B$ | $A \vee B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $A$ | $B$ | $A \wedge B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $A$ | $\neg A$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

# Abbreviations

$A, B, C$ or
$P, Q, R$ or $\ldots$ instead of $A_1, A_2, A_3 \ldots$

$(F_1 \rightarrow F_2)$ instead of $(\neg F_1 \vee F_2)$

$(F_1 \leftrightarrow F_2)$ instead of $((F_1 \wedge F_2) \vee (\neg F_1 \wedge \neg F_2))$

$(\bigvee_{i=1}^{n} F_i)$ instead of $(\ldots((F_1 \vee F_2) \vee F_3) \vee \ldots \vee F_n)$

$(\bigwedge_{i=1}^{n} F_i)$ instead of $(\ldots((F_1 \wedge F_2) \wedge F_3) \wedge \ldots \wedge F_n)$

# Truth tables for $\rightarrow$ and $\leftrightarrow$

Truth tables for the operators $\rightarrow$, $\leftrightarrow$:

| $A$ | $B$ | $A \rightarrow B$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Name: *Implication*

Interpretation: If $A$ holds, then $B$ must also hold.

| $A$ | $B$ | $A \leftrightarrow B$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Name: *Equivalence*

Interpretation: $A$ holds if and only if $B$ holds.

# Attention!!!

$A \to B$ does not say that $A$ is a cause for $B$.

"Penguins swim $\to$ Dogs bark"
is true (in our world).

$A \to B$ says nothing about whether $A$ is true or false.

"$x = y \to 2x = 2y$"
is true for all numbers $x$ and $y$.

A false statement implies anything.

"Penguins fly $\to$ Cats bark"
is true (in our world).

# Formalization of Natural Language (I)

A device consists of a component $A$, a component $B$, and a red light. The following is known:

- Component $A$ or Component $B$ (or both) are broken.
- If Component $A$ is broken, then Component $B$ is also broken.
- If Component $B$ is broken and the red light is on, then Component $A$ is not broken.
- The red light is on.

Formalize this situation as a propositional logic formula and create the truth table for this formula. Use the following atomic formulas: $RL$ (red light is on), $AK$ (Component $A$ broken), $BK$ (Component $B$ broken).

# Formalization of Natural Language (II)

**Full Truth Table:**

| $RL$ | $AK$ | $BK$ | $(AK \vee BK) \wedge (AK \rightarrow BK) \wedge ((BK \wedge RL) \rightarrow \neg AK) \wedge RL$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

# Formalization of Sudoku

Formalize the Sudoku problem:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | | | | 9 | | | | 2 |
| | | 1 | | | | 5 | | |
| | 9 | | 3 | 4 | 5 | | 1 | |
| | | 8 | | | | 2 | 5 | |
| 7 | | 5 | | 3 | | 4 | 6 | 1 |
| | 4 | 6 | | | | 9 | | 8 |
| | 6 | | 1 | 5 | 9 | | 8 | |
| | | 9 | | | | 6 | | |
| 5 | | | | 7 | | | | 4 |

Use an atomic formula $A[n, x, y]$ for each triple $(n, x, y) \in \{1, \ldots, 9\}^3$:

$A[n, x, y] = 1$, if: In row $x$ and column $y$ the number $n$ is placed.

# Formalization of Sudoku

**Example**: In the first row, all numbers from 1 to 9 appear

$$\bigwedge_{n=1}^{9} \left( \bigvee_{y=1}^{9} A[n, 1, y] \right)$$

The truth table has

$$2^{729} = 28240139587082174969491088422046278633513539118515775246834019308626938303611984999058739209952299969708978654982839965781232968658783909476265530884869461064307960914827161205726320724927035277237573594788345303657349 12$$

rows. Why?

# Models

Let $F$ be a formula and $\mathcal{B}$ an assignment.
If $\mathcal{B}$ is defined for all atomic formulas occurring in $F$,
then $\mathcal{B}$ is called suitable for $F$.

Let $\mathcal{B}$ be suitable for $F$:

| | | |
|---|---|---|
| If $\mathcal{B}(F) = 1$ | we write | $\mathcal{B} \models F$ |
| | and say | $F$ holds under $\mathcal{B}$ |
| | or | $\mathcal{B}$ is a model for $F$ |

| | | |
|---|---|---|
| If $\mathcal{B}(F) = 0$ | we write | $\mathcal{B} \not\models F$ |
| | and say | $F$ does not hold under $\mathcal{B}$ |
| | or | $\mathcal{B}$ is not a model for $F$ |

# Validity and Satisfiability

Satisfiability: A formula $F$ is called satisfiable if $F$ has at least one model; otherwise, $F$ is called unsatisfiable.

A (finite or infinite!) set of formulas $M$ is called satisfiable if there is an assignment that is a model for each formula in $M$.

Validity: A formula $F$ is called valid (or universally valid or a tautology) if every assignment suitable for $F$ is a model for $F$. We write $\models F$ if $F$ is valid, and $\not\models F$ otherwise.

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | | | |
| $A \vee B$ | | | |
| $A \vee \neg A$ | | | |
| $A \wedge \neg A$ | | | |
| $A \rightarrow \neg A$ | | | |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | | |
| $A \vee B$ | | | |
| $A \vee \neg A$ | | | |
| $A \wedge \neg A$ | | | |
| $A \rightarrow \neg A$ | | | |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | |
| $A \vee B$ | | | |
| $A \vee \neg A$ | | | |
| $A \wedge \neg A$ | | | |
| $A \rightarrow \neg A$ | | | |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | | | |
| $A \vee \neg A$ | | | |
| $A \wedge \neg A$ | | | |
| $A \rightarrow \neg A$ | | | |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | | |
| $A \vee \neg A$ | | | |
| $A \wedge \neg A$ | | | |
| $A \rightarrow \neg A$ | | | |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | |
| $A \vee \neg A$ | | | |
| $A \wedge \neg A$ | | | |
| $A \rightarrow \neg A$ | | | |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | | | |
| $A \wedge \neg A$ | | | |
| $A \rightarrow \neg A$ | | | |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | | |
| $A \wedge \neg A$ | | | |
| $A \rightarrow \neg A$ | | | |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | |
| $A \wedge \neg A$ | | | |
| $A \rightarrow \neg A$ | | | |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | | | |
| $A \rightarrow \neg A$ | | | |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | | |
| $A \rightarrow \neg A$ | | | |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | |
| $A \rightarrow \neg A$ | | | |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | | | |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | N | | |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | N | Y | |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | N | Y | N |
| $A \rightarrow B$ | | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | N | Y | N |
| $A \rightarrow B$ | N | | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | N | Y | N |
| $A \rightarrow B$ | N | Y | |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | N | Y | N |
| $A \rightarrow B$ | N | Y | N |
| $A \rightarrow (B \rightarrow A)$ | | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | N | Y | N |
| $A \rightarrow B$ | N | Y | N |
| $A \rightarrow (B \rightarrow A)$ | Y | | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | N | Y | N |
| $A \rightarrow B$ | N | Y | N |
| $A \rightarrow (B \rightarrow A)$ | Y | Y | |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | N | Y | N |
| $A \rightarrow B$ | N | Y | N |
| $A \rightarrow (B \rightarrow A)$ | Y | Y | N |
| $A \rightarrow (A \rightarrow B)$ | | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | N | Y | N |
| $A \rightarrow B$ | N | Y | N |
| $A \rightarrow (B \rightarrow A)$ | Y | Y | N |
| $A \rightarrow (A \rightarrow B)$ | N | | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | N | Y | N |
| $A \rightarrow B$ | N | Y | N |
| $A \rightarrow (B \rightarrow A)$ | Y | Y | N |
| $A \rightarrow (A \rightarrow B)$ | N | Y | |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | N | Y | N |
| $A \rightarrow B$ | N | Y | N |
| $A \rightarrow (B \rightarrow A)$ | Y | Y | N |
| $A \rightarrow (A \rightarrow B)$ | N | Y | N |
| $A \leftrightarrow \neg A$ | | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | N | Y | N |
| $A \rightarrow B$ | N | Y | N |
| $A \rightarrow (B \rightarrow A)$ | Y | Y | N |
| $A \rightarrow (A \rightarrow B)$ | N | Y | N |
| $A \leftrightarrow \neg A$ | N | | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | N | Y | N |
| $A \rightarrow B$ | N | Y | N |
| $A \rightarrow (B \rightarrow A)$ | Y | Y | N |
| $A \rightarrow (A \rightarrow B)$ | N | Y | N |
| $A \leftrightarrow \neg A$ | N | N | |

# Aufgabe

| | Valid | Satisfiable | Unsatisfiable |
|---|---|---|---|
| $A$ | N | Y | N |
| $A \vee B$ | N | Y | N |
| $A \vee \neg A$ | Y | Y | N |
| $A \wedge \neg A$ | N | N | Y |
| $A \rightarrow \neg A$ | N | Y | N |
| $A \rightarrow B$ | N | Y | N |
| $A \rightarrow (B \rightarrow A)$ | Y | Y | N |
| $A \rightarrow (A \rightarrow B)$ | N | Y | N |
| $A \leftrightarrow \neg A$ | N | N | Y |

# Task

Do the following statements hold true?

| | | | | Y/N | Counterex. |
|---|---|---|---|---|---|
| If | $F$ is valid, | then | $F$ is satisfiable | | |
| If | $F$ is satisfiable, | then | $\neg F$ is unsatisfiable | | |
| If | $F$ is valid, | then | $\neg F$ is unsatisfiable | | |
| If | $F$ is unsatisfiable, | then | $\neg F$ is valid | | |

# Task

Do the following statements hold true?

| | | | | Y/N | Counterex. |
|---|---|---|---|---|---|
| If | $F$ is valid, | then | $F$ is satisfiable | Y | |
| If | $F$ is satisfiable, | then | $\neg F$ is unsatisfiable | | |
| If | $F$ is valid, | then | $\neg F$ is unsatisfiable | | |
| If | $F$ is unsatisfiable, | then | $\neg F$ is valid | | |

# Task

Do the following statements hold true?

| | | | | Y/N | Counterex. |
|---|---|---|---|---|---|
| If | $F$ is valid, | then | $F$ is satisfiable | Y | |
| If | $F$ is satisfiable, | then | $\neg F$ is unsatisfiable | N | $F = A$ |
| If | $F$ is valid, | then | $\neg F$ is unsatisfiable | | |
| If | $F$ is unsatisfiable, | then | $\neg F$ is valid | | |

# Task

Do the following statements hold true?

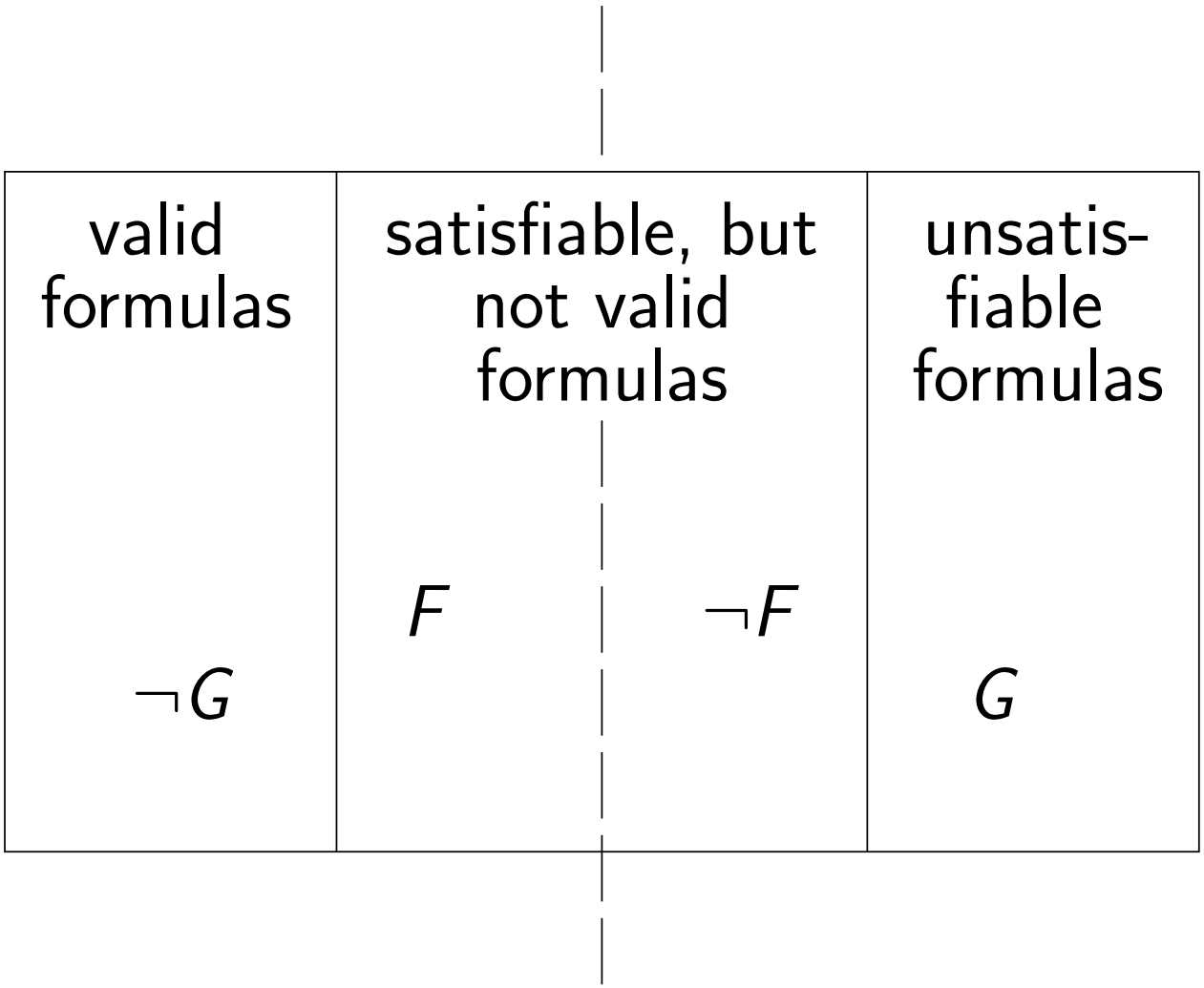
| | | | | Y/N | Counterex. |
|---|---|---|---|---|---|
| If | $F$ is valid, | then | $F$ is satisfiable | Y | |
| If | $F$ is satisfiable, | then | $\neg F$ is unsatisfiable | N | $F = A$ |
| If | $F$ is valid, | then | $\neg F$ is unsatisfiable | Y | |
| If | $F$ is unsatisfiable, | then | $\neg F$ is valid | | |

# Task

Do the following statements hold true?

| | | | | Y/N | Counterex. |
|---|---|---|---|---|---|
| If | $F$ is valid, | then | $F$ is satisfiable | Y | |
| If | $F$ is satisfiable, | then | $\neg F$ is unsatisfiable | N | $F = A$ |
| If | $F$ is valid, | then | $\neg F$ is unsatisfiable | Y | |
| If | $F$ is unsatisfiable, | then | $\neg F$ is valid | Y | |

# Reflection Principle



# A Validity Test

How can one check whether a formula $F$ is valid/satisfiable?

One possibility: create a truth table.

Suppose the formula $F$ contains $n$ different atomic formulas. How large is the truth table?

Number of Rows in the truth table: $2^n$

Is there a more efficient method?

Probably not: Satisfiability of propositional formulas is NP-complete and thus cannot be done in polynomial time unless P = NP.

**The SAT Problem**

INPUT: A propositional formula $F$

QUESTION: Is $F$ satisfiable?

Propositional formulas can be encoded, for example, using words over the alphabet $\{a, \vee, \wedge, \neg, ), (\}$ (atomic formulas are encoded using words of the form $a^n$).

**Theorem 40**

SAT $\in$ NP

**Proof:** Let $F$ be a propositional formula in which the atomic formulas $A_1, \ldots, A_n$ occur.

A nondeterministic Turing machine "guesses" an assignment $\mathcal{B} : \{A_1, \ldots, A_n\} \to \{0, 1\}$ in a first phase:

- In the first step, the Turing machine branches (i.e., there are two subsequent configurations).
  In the first branch, the Turing machine writes $A_1 0$ on the tape, and in the second branch, it writes $A_1 1$ on the tape.
- In the second step, the Turing machine branches again.
  In the first branch, it writes (after $A_1 b$ with $b \in \{0, 1\}$) $A_2 0$ on the tape, and in the second branch, it writes $A_2 1$ on the tape.
  $\vdots$

# Complexity Theory: Cook's Theorem

After $n$ steps, each of the $2^n$ computation branches contains a word of the form $A_1 b_1 A_2 b_2 \cdots A_n b_n$ on the tape, with $b_1, \ldots, b_n \in \{0, 1\}$.

This word encodes the assignment $\mathcal{B}$ with $\mathcal{B}(A_i) = b_i$ for $1 \leq i \leq n$.

In a second phase, the Turing machine can now deterministically compute the value $\mathcal{B}(F)$ by traversing the formula $F$ from left to right, determining the value $\mathcal{B}(A_i) = b_i$ from the stored "assignment word" every time an atomic formula $A_i$ is encountered.

This requires at most $|F|^2$ steps, so the Turing machine performs only $O(|F|^2)$ steps on each computation path.

The machine outputs 1 at the end if and only if $\mathcal{B}(F) = 1$.

Therefore, there exists a computation path on which the machine outputs 1 if and only if $F$ is satisfiable. □

**Theorem 41 (Cook's Theorem)**

SAT is NP-complete.

**Proof:**

We still need to show that SAT is NP-hard.

Let $L \in \mathrm{NP}$, $L \subseteq \Sigma^*$.

For $w \in \Sigma^*$, we construct a propositional formula $f(w)$ such that:
$w \in L \iff f(w)$ is satisfiable.

The mapping $f$ will be computable in polynomial time.

Let $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ be a nondeterministic Turing machine for $L$ that terminates on every computation path in at most $p(n)$ steps for an input of length $n$ (where $p(n)$ is a polynomial).

Let $w = w_1 w_2 \cdots w_n \in \Sigma^*$ be an input of length $n$.

# Complexity Theory: Cook's Theorem

We make the following assumptions about $M$ without loss of generality:

1. The head of $M$ never moves left of the position where it starts (this can be achieved through special marking).
2. $M$ terminates with output 1 if and only if $M$ enters the special state $z_1 \in E$. This means that the state $z_1$ signals acceptance of the input.
3. All tuples of the form $(z_1, a, z_1, a, N)$ (with $a \in \Gamma$) belong to $\delta$.
4. $(z, a, z', a', D), (z, b, z'', b', D') \in \delta \implies a = b, a' = b', D = D'$

   Thus, we only have a nondeterministic choice regarding the successor state $z'$ (see next slide).

# Complexity Theory: Cook's Theorem

Property (4) can be enforced as follows.

First, we can replace the language $L$ with $\$L$ for a new symbol \$, since $L \leq_p \$L$ holds (i.e., from $\$L \leq_p$ SAT it follows that $L \leq_p$ SAT again).

Thus, we know that all positive inputs must begin with a \$.

Now define the state set and transition relation as follows:

$$\begin{aligned} Z' =& \{z[a, a', D] \mid z \in Z, a, a' \in \Gamma, D \in \{L, R, N\}\} \cup \{z_0[\$, \$, R]\} \\ \delta' =& \{(z[a, a', D], a, z'[b, b', D'], a', D) \mid (z, a, z', a', D) \in \delta, \\ & \qquad b, b' \in \Gamma, D' \in \{L, R, N\}\} \cup \\ & \{(z_0[\$, \$, R], \$, z_0[a, a', D], \$, R) \mid a, a' \in \Gamma, D \in \{L, R, N\}\} \end{aligned}$$

The new starting state is $z_0[\$, \$, R]$.

# Complexity Theory: Cook's Theorem

Every configuration reachable from the start configuration can be described by a word from

$$\text{Conf} = \{\Box uzv\Box \mid z \in Z;\ u, v \in \Gamma^*;\ |uv| = p(n)\}$$

The start configuration is $\Box z_0 w \Box^{p(n)+1-n}$.

Due to points (2) and (3), the machine $M$ accepts the input $w$ on a specific computation path if and only if $M$ is in state $z_1$ after $p(n)$ steps.

**Notation:** For an $\alpha \in \text{Conf}$, we write

$$\alpha = \alpha[-1]\alpha[0]\cdots\alpha[p(n)]\alpha[p(n)+1]$$

where $\alpha[-1] = \Box$, $\alpha[0], \ldots, \alpha[p(n)] \in Z \cup \Gamma$, and $\alpha[p(n)+1] = \Box$.

# Complexity Theory: Cook's Theorem

Define the set of 4-tuples

$$\begin{aligned}\Delta = &\{(a,b,c,b) \mid a,b,c \in \Gamma\} \\ &\cup \{(c,b,z,z'),(b,z,a,b),(z,a,d,a') \mid (z,a,z',a',L) \in \delta, c,b,d \in \Gamma\} \\ &\cup \{(c,b,z,b),(b,z,a,z'),(z,a,d,a') \mid (z,a,z',a',N) \in \delta, c,b,d \in \Gamma\} \\ &\cup \{(c,b,z,b),(b,z,a,a'),(z,a,d,z') \mid (z,a,z',a',R) \in \delta, c,b,d \in \Gamma\}\end{aligned}$$

The idea of the $\Delta$-tuples: If three consecutive positions $i-1$, $i$, $i+1$ of a configuration $\alpha$ contain the symbols $x, y, z \in Z \cup \Gamma$, then for every subsequent configuration $\alpha'$ of $\alpha$, there must exist a tuple $(x, y, z, y')$ where the symbol $y'$ is present at position $i$.

Due to point (4), it holds for all $\alpha, \alpha' \in \square(Z \cup \Gamma)^*\square$ with $|\alpha| = |\alpha'|$:

$$\alpha, \alpha' \in \text{Conf and } \alpha \vdash_M \alpha'$$

$$\Longleftrightarrow$$

$$\alpha \in \text{Conf and } \forall i \in \{0, \ldots, p(n)\} : (\alpha[i-1], \alpha[i], \alpha[i+1], \alpha'[i]) \in \Delta.$$

**Example:**

If $(z, a, z', a', L) \in \delta$, the following local tape modification is possible for all $b \in \Gamma$:

| Position | | | $i-1$ | $i$ | $i+1$ | | |
|---|---|---|---|---|---|---|---|
| $\alpha$ | $=$ | $\cdots$ | $\cdots$ | $b$ | $z$ | $a$ | $\cdots$ | $\cdots$ |
| $\alpha'$ | $=$ | $\cdots$ | $\cdots$ | $z'$ | $b$ | $a'$ | $\cdots$ | $\cdots$ |

If $(z, a, z', a', R) \in \delta$, the following local tape modification is possible for all $b \in \Gamma$:

| Position | | | $i-1$ | $i$ | $i+1$ | | |
|---|---|---|---|---|---|---|---|
| $\alpha$ | $=$ | $\cdots$ | $\cdots$ | $b$ | $z$ | $a$ | $\cdots$ | $\cdots$ |
| $\alpha'$ | $=$ | $\cdots$ | $\cdots$ | $b$ | $a'$ | $z'$ | $\cdots$ | $\cdots$ |

# Complexity Theory: Cook's Theorem

We can now describe a computation of $M$ as a matrix:

$$\begin{array}{lcccccc}
\alpha_0 & = & \square & \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,p(n)} & \square \\
\alpha_1 & = & \square & \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,p(n)} & \square \\
 & & & & \vdots & & & \\
\alpha_{p(n)} & = & \square & \alpha_{p(n),0} & \alpha_{p(n),1} & \cdots & \alpha_{p(n),p(n)} & \square
\end{array}$$

For each triple $(a, i, t)$ ($a \in Z \cup \Gamma, -1 \leq i \leq p(n) + 1, 0 \leq t \leq p(n)$), let $x(a, i, t)$ be a propositional variable (atomic formula).

**Interpretation:** $x(a, i, t) =$ **true** if and only if, at time $t$, the $i$-th character of the current configuration is $a$.

# Complexity Theory: Cook's Theorem

At positions $-1$ and $p(n)+1$, $\square$ is always present:

$$G(n) \quad = \quad \bigwedge_{0 \leq t \leq p(n)} \Big( x(\square, -1, t) \wedge x(\square, p(n)+1, t) \Big)$$

For each pair $(i, t)$, exactly one variable $x(a, i, t)$ is true (at any time, only one symbol can be on a tape cell):

$$X(n) \quad = \quad \bigwedge_{\substack{0 \leq t \leq p(n) \\ -1 \leq i \leq p(n)+1}} \left( \bigvee_{a \in Z \cup \Gamma} \left( x(a, i, t) \wedge \bigwedge_{b \neq a} \neg x(b, i, t) \right) \right)$$

# Complexity Theory: Cook's Theorem

At time $t = 0$, the configuration is equal to $\square z_0 w \square^{p(n)+1-n}$:

$$S(w) \quad = \quad \left( x(z_0, 0, 0) \wedge \bigwedge_{i=1}^{n} x(w_i, i, 0) \wedge \bigwedge_{i=n+1}^{p(n)} x(\square, i, 0) \right)$$

The computation respects the local relation $\Delta$:

$$D(n) \quad = \quad \bigwedge_{\substack{0 \leq i \leq p(n) \\ 0 \leq t < p(n)}} \quad \bigvee_{(a,b,c,d) \in \Delta} \left( \begin{array}{lll} x(a, i-1, t) & \wedge & x(b, i, t) \wedge \\ x(c, i+1, t) & \wedge & x(d, i, t+1) \end{array} \right)$$

# Complexity Theory: Cook's Theorem

Finally, let

$$R(w) = G(n) \wedge X(n) \wedge S(w) \wedge D(n).$$

A natural bijection arises between the set of satisfying assignments for $R(w)$ and the set of computations of $M$ on the input $w$ that consist of $p(n)$ computation steps.

For $f(w) = R(w) \wedge \bigvee_{i=0}^{p(n)} x(z_1, i, p(n))$ it holds that:

$$f(w) \text{ is satisfiable} \quad \Longleftrightarrow \quad w \in L.$$

The number of variables of $f(w) \in \mathcal{O}(p(n)^2)$

Length of $f(w) \in \mathcal{O}(p(n)^2 \log p(n))$

The factor $\mathcal{O}(\log p(n))$ is necessary since writing down the indices requires $\log p(n)$ many bits. □

# Complexity Theory: Cook's Theorem

Cook's theorem can be strengthened as follows:

**3-CNF-SAT**

INPUT: A propositional formula $F$ in CNF, where each clause consists of at most 3 literals (atomic formulas or negated atomic formulas).
QUESTION: Is $F$ satisfiable?

**Theorem 42**

*3-CNF-SAT is NP-complete*

Note: 2-CNF-SAT $\in$ P

# Consequence

A formula $G$ is called a consequence of the formulas $F_1, \ldots, F_k$ if for every assignment $\mathcal{B}$ suitable for both $F_1, \ldots, F_k$ and $G$, the following holds:

> *If $\mathcal{B}$ is a model of $\{F_1, \ldots, F_k\}$ (i.e., a model of $F_1$ and a model of $F_2$ and ... and a model of $F_k$), then $\mathcal{B}$ is also a model of $G$.*

We write $F_1, \ldots, F_k \models G$ if $G$ is a consequence of $F_1, \ldots, F_k$.

# Consequence: Example

$$(A1 \vee B1), (A1 \rightarrow B1),$$
$$((B1 \wedge R1) \rightarrow \neg A1), R1 \quad \models \quad (R1 \wedge \neg A1) \wedge B1$$

If Component A or Component B is faulty *and* from the fact that Component A is faulty, it always follows that Component B is faulty *and* ...

... then one can conclude: the red light is on, Component A is not faulty and Component B is faulty.

# Exercise

| $M$ | $F$ | Does $M \models F$ hold? |
|---|---|---|
| $A$ | $A \vee B$ | |
| $A$ | $A \wedge B$ | |
| $A, B$ | $A \vee B$ | |
| $A, B$ | $A \wedge B$ | |
| $A \wedge B$ | $A$ | |
| $A \vee B$ | $A$ | |
| $A, A \rightarrow B$ | $B$ | |

# Exercise

| $M$ | $F$ | Does $M \models F$ hold? |
|---|---|---|
| $A$ | $A \vee B$ | Y |
| $A$ | $A \wedge B$ | |
| $A, B$ | $A \vee B$ | |
| $A, B$ | $A \wedge B$ | |
| $A \wedge B$ | $A$ | |
| $A \vee B$ | $A$ | |
| $A, A \rightarrow B$ | $B$ | |

# Exercise

| $M$ | $F$ | Does $M \models F$ hold? |
|---|---|---|
| $A$ | $A \vee B$ | Y |
| $A$ | $A \wedge B$ | N |
| $A, B$ | $A \vee B$ | |
| $A, B$ | $A \wedge B$ | |
| $A \wedge B$ | $A$ | |
| $A \vee B$ | $A$ | |
| $A, A \rightarrow B$ | $B$ | |

# Exercise

| $M$ | $F$ | Does $M \models F$ hold? |
|---|---|---|
| $A$ | $A \vee B$ | Y |
| $A$ | $A \wedge B$ | N |
| $A, B$ | $A \vee B$ | Y |
| $A, B$ | $A \wedge B$ | |
| $A \wedge B$ | $A$ | |
| $A \vee B$ | $A$ | |
| $A, A \rightarrow B$ | $B$ | |

# Exercise

| $M$ | $F$ | Does $M \models F$ hold? |
|---|---|---|
| $A$ | $A \vee B$ | Y |
| $A$ | $A \wedge B$ | N |
| $A, B$ | $A \vee B$ | Y |
| $A, B$ | $A \wedge B$ | Y |
| $A \wedge B$ | $A$ | |
| $A \vee B$ | $A$ | |
| $A, A \rightarrow B$ | $B$ | |

# Exercise

| $M$ | $F$ | Does $M \models F$ hold? |
|---|---|---|
| $A$ | $A \vee B$ | Y |
| $A$ | $A \wedge B$ | N |
| $A, B$ | $A \vee B$ | Y |
| $A, B$ | $A \wedge B$ | Y |
| $A \wedge B$ | $A$ | Y |
| $A \vee B$ | $A$ | |
| $A, A \rightarrow B$ | $B$ | |

# Exercise

| $M$ | $F$ | Does $M \models F$ hold? |
|---|---|---|
| $A$ | $A \vee B$ | Y |
| $A$ | $A \wedge B$ | N |
| $A, B$ | $A \vee B$ | Y |
| $A, B$ | $A \wedge B$ | Y |
| $A \wedge B$ | $A$ | Y |
| $A \vee B$ | $A$ | N |
| $A, A \rightarrow B$ | $B$ | |

# Exercise

| $M$ | $F$ | Does $M \models F$ hold? |
|---|---|---|
| $A$ | $A \vee B$ | Y |
| $A$ | $A \wedge B$ | N |
| $A, B$ | $A \vee B$ | Y |
| $A, B$ | $A \wedge B$ | Y |
| $A \wedge B$ | $A$ | Y |
| $A \vee B$ | $A$ | N |
| $A, A \rightarrow B$ | $B$ | Y |

# Inference, Validity, and Unsatisfiability

**Theorem 43**

*The following statements are equivalent:*

1. $F_1, \ldots, F_k \models G$, *i.e.,* $G$ *is an inference from* $F_1, \ldots, F_k$.
2. $((\bigwedge_{i=1}^{k} F_i) \rightarrow G)$ *is valid.*
3. $((\bigwedge_{i=1}^{k} F_i) \wedge \neg G)$ *is unsatisfiable.*

**Proof:**

**1 $\Rightarrow$ 2**: Assume $F_1, \ldots, F_k \models G$.

**Claim:** $((\bigwedge_{i=1}^{k} F_i) \rightarrow G)$ is valid.

Let $\mathcal{B}$ be any assignment that is suitable for $((\bigwedge_{i=1}^{k} F_i) \rightarrow G)$.

**Case 1:** There exists an $i \in \{1, \ldots, k\}$ such that $\mathcal{B}(F_i) = 0$:

Then $\mathcal{B}(\bigwedge_{i=1}^{k} F_i) = 0$ also holds, and thus $\mathcal{B}((\bigwedge_{i=1}^{k} F_i) \rightarrow G) = 1$.

**Case 2:** For all $i \in \{1, \ldots, k\}$, it holds that $\mathcal{B}(F_i) = 1$:

From $F_1, \ldots, F_k \models G$, it follows that $\mathcal{B}(G) = 1$, and thus $\mathcal{B}((\bigwedge_{i=1}^{k} F_i) \rightarrow G) = 1$ as well.

# Inference, Validity, and Unsatisfiability

**2 $\Rightarrow$ 3**: Let $((\bigwedge_{i=1}^{k} F_i) \rightarrow G)$ be valid.

**Claim:** $((\bigwedge_{i=1}^{k} F_i) \wedge \neg G)$ is unsatisfiable.

Let $\mathcal{B}$ be an arbitrary assignment.

**Case 1:** $\mathcal{B}(G) = 1$:

Then $\mathcal{B}((\bigwedge_{i=1}^{k} F_i) \wedge \neg G) = 0$ holds.

**Case 2:** $\mathcal{B}(\bigwedge_{i=1}^{k} F_i) = 0$:

Then again, $\mathcal{B}((\bigwedge_{i=1}^{k} F_i) \wedge \neg G) = 0$ holds.

**Case 3:** $\mathcal{B}(\bigwedge_{i=1}^{k} F_i) = 1$ and $\mathcal{B}(G) = 0$:

Then $\mathcal{B}((\bigwedge_{i=1}^{k} F_i) \rightarrow G) = 0$ holds, but this contradicts the fact that $((\bigwedge_{i=1}^{k} F_i) \rightarrow G)$ is valid.

Thus, Case 3 cannot occur.

# Inference, Validity, and Unsatisfiability

**3 $\Rightarrow$ 1**: Let $((\bigwedge_{i=1}^{k} F_i) \wedge \neg G)$ be unsatisfiable.

**Claim:** $F_1, \ldots, F_k \models G$

Let $\mathcal{B}$ be an arbitrary assignment with $\mathcal{B}(F_i) = 1$ for all $i \in \{1, \ldots, k\}$.

Since $((\bigwedge_{i=1}^{k} F_i) \wedge \neg G)$ is unsatisfiable, it must hold that $\mathcal{B}(G) = 1$ (otherwise, $\mathcal{B}((\bigwedge_{i=1}^{k} F_i) \wedge \neg G) = 1$ would be true).

# Equivalence

Two formulas $F$ and $G$ are called (semantically) equivalent, if for all assignments $\mathcal{B}$, which are suitable for both $F$ and $G$, it holds that $\mathcal{B}(F) = \mathcal{B}(G)$. We denote this as $F \equiv G$.

# Aufgabe

Do the following equivalences hold?

$$
\begin{aligned}
(A \wedge (A \vee B)) &\equiv A \\
\neg(A \vee B) &\equiv (\neg A \wedge \neg B) \\
(A \wedge (B \vee C)) &\equiv ((A \wedge B) \vee C) \\
(A \wedge (B \vee C)) &\equiv ((A \wedge B) \vee (A \wedge C)) \\
(A \to B) \to C &\equiv A \to (B \to C) \\
(A \to B) \to C &\equiv (A \wedge B) \to C \\
(A \leftrightarrow B) \leftrightarrow C &\equiv A \leftrightarrow (B \leftrightarrow C)
\end{aligned}
$$

# Aufgabe

Do the following equivalences hold?

$$
\begin{aligned}
(A \wedge (A \vee B)) &\equiv A && Y \\
\neg(A \vee B) &\equiv (\neg A \wedge \neg B) \\
(A \wedge (B \vee C)) &\equiv ((A \wedge B) \vee C) \\
(A \wedge (B \vee C)) &\equiv ((A \wedge B) \vee (A \wedge C)) \\
(A \to B) \to C &\equiv A \to (B \to C) \\
(A \to B) \to C &\equiv (A \wedge B) \to C \\
(A \leftrightarrow B) \leftrightarrow C &\equiv A \leftrightarrow (B \leftrightarrow C)
\end{aligned}
$$

# Aufgabe

Do the following equivalences hold?

$$
\begin{array}{rcll}
(A \wedge (A \vee B)) & \equiv & A & Y \\
\neg(A \vee B) & \equiv & (\neg A \wedge \neg B) & Y \\
(A \wedge (B \vee C)) & \equiv & ((A \wedge B) \vee C) & \\
(A \wedge (B \vee C)) & \equiv & ((A \wedge B) \vee (A \wedge C)) & \\
(A \to B) \to C & \equiv & A \to (B \to C) & \\
(A \to B) \to C & \equiv & (A \wedge B) \to C & \\
(A \leftrightarrow B) \leftrightarrow C & \equiv & A \leftrightarrow (B \leftrightarrow C) &
\end{array}
$$

# Aufgabe

Do the following equivalences hold?

$$
\begin{array}{rcll}
(A \wedge (A \vee B)) & \equiv & A & Y \\
\neg(A \vee B) & \equiv & (\neg A \wedge \neg B) & Y \\
(A \wedge (B \vee C)) & \equiv & ((A \wedge B) \vee C) & N : \mathcal{B}(A) = 0, \mathcal{B}(C) = 1 \\
(A \wedge (B \vee C)) & \equiv & ((A \wedge B) \vee (A \wedge C)) & \\
(A \to B) \to C & \equiv & A \to (B \to C) & \\
(A \to B) \to C & \equiv & (A \wedge B) \to C & \\
(A \leftrightarrow B) \leftrightarrow C & \equiv & A \leftrightarrow (B \leftrightarrow C) &
\end{array}
$$

# Aufgabe

Do the following equivalences hold?

$$
\begin{array}{rcll}
(A \wedge (A \vee B)) & \equiv & A & Y \\
\neg(A \vee B) & \equiv & (\neg A \wedge \neg B) & Y \\
(A \wedge (B \vee C)) & \equiv & ((A \wedge B) \vee C) & N : \mathcal{B}(A) = 0, \mathcal{B}(C) = 1 \\
(A \wedge (B \vee C)) & \equiv & ((A \wedge B) \vee (A \wedge C)) & Y \\
(A \to B) \to C & \equiv & A \to (B \to C) & \\
(A \to B) \to C & \equiv & (A \wedge B) \to C & \\
(A \leftrightarrow B) \leftrightarrow C & \equiv & A \leftrightarrow (B \leftrightarrow C) &
\end{array}
$$

# Aufgabe

Do the following equivalences hold?

$$
\begin{array}{rcll}
(A \wedge (A \vee B)) & \equiv & A & Y \\
\neg(A \vee B) & \equiv & (\neg A \wedge \neg B) & Y \\
(A \wedge (B \vee C)) & \equiv & ((A \wedge B) \vee C) & N : \mathcal{B}(A) = 0, \mathcal{B}(C) = 1 \\
(A \wedge (B \vee C)) & \equiv & ((A \wedge B) \vee (A \wedge C)) & Y \\
(A \to B) \to C & \equiv & A \to (B \to C) & N : \mathcal{B}(A) = \mathcal{B}(B) = \mathcal{B}(C) = 0 \\
(A \to B) \to C & \equiv & (A \wedge B) \to C & \\
(A \leftrightarrow B) \leftrightarrow C & \equiv & A \leftrightarrow (B \leftrightarrow C) &
\end{array}
$$

# Aufgabe

Do the following equivalences hold?

$$
\begin{array}{rcll}
(A \wedge (A \vee B)) & \equiv & A & Y \\
\neg(A \vee B) & \equiv & (\neg A \wedge \neg B) & Y \\
(A \wedge (B \vee C)) & \equiv & ((A \wedge B) \vee C) & N : \mathcal{B}(A) = 0, \mathcal{B}(C) = 1 \\
(A \wedge (B \vee C)) & \equiv & ((A \wedge B) \vee (A \wedge C)) & Y \\
(A \to B) \to C & \equiv & A \to (B \to C) & N : \mathcal{B}(A) = \mathcal{B}(B) = \mathcal{B}(C) = 0 \\
(A \to B) \to C & \equiv & (A \wedge B) \to C & N : \mathcal{B}(A) = \mathcal{B}(B) = \mathcal{B}(C) = 0 \\
(A \leftrightarrow B) \leftrightarrow C & \equiv & A \leftrightarrow (B \leftrightarrow C) &
\end{array}
$$

# Aufgabe

Do the following equivalences hold?

$$
\begin{array}{rcll}
(A \wedge (A \vee B)) & \equiv & A & Y \\
\neg(A \vee B) & \equiv & (\neg A \wedge \neg B) & Y \\
(A \wedge (B \vee C)) & \equiv & ((A \wedge B) \vee C) & N : \mathcal{B}(A) = 0, \mathcal{B}(C) = 1 \\
(A \wedge (B \vee C)) & \equiv & ((A \wedge B) \vee (A \wedge C)) & Y \\
(A \to B) \to C & \equiv & A \to (B \to C) & N : \mathcal{B}(A) = \mathcal{B}(B) = \mathcal{B}(C) = 0 \\
(A \to B) \to C & \equiv & (A \wedge B) \to C & N : \mathcal{B}(A) = \mathcal{B}(B) = \mathcal{B}(C) = 0 \\
(A \leftrightarrow B) \leftrightarrow C & \equiv & A \leftrightarrow (B \leftrightarrow C) & Y
\end{array}
$$

# Equivalence

Truth tables for $(A \leftrightarrow B) \leftrightarrow C$ und $A \leftrightarrow (B \leftrightarrow C)$

| $A$ | $B$ | $C$ | $(A \leftrightarrow B) \leftrightarrow C$ | $A \leftrightarrow (B \leftrightarrow C)$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# The Main Problems of Propositional Logic

In "computational" propositional logic, we seek methods that solve the following tasks (problems):

- Model Checking
  Let $F$ be a formula and $\mathcal{B}$ an suitable assignment. Does $\mathcal{B}(F) = 1$ hold?
- Satisfiability (SAT)
  Let $F$ be a formula. Is $F$ satisfiable?
- Validity
  Let $F$ be a formula. Is $F$ valid?
- Consequence
  Let $F$ and $G$ be formulas. Does $F \models G$ hold?
- Equivalence
  Let $F$ and $G$ be formulas. Does $F \equiv G$ hold?

# Exercise

Prove that the following statements hold:

| | | | |
|---|---|---|---|
| If | $(F \to G)$ is valid, | then | $F \models G$. |
| If | $F \models G$, | then | $(F \to G)$ is valid. |
| If | $(F \leftrightarrow G)$ is valid, | then | $F \equiv G$. |
| If | $F \equiv G$, | then | $(F \leftrightarrow G)$ is valid. |

# Reduction of Problems

Which problems can be reduced to which (polynomially)?
($A \equiv_p B$ stands for $A \leq_p B \leq_p A$.)

- Validity $\equiv_p$ (Unsatisfiability (complement of SAT)):
  $F$ is valid if and only if $\neg F$ is unsatisfiable.
  $F$ is satisfiable if and only if $\neg F$ is not valid.
- Validity $\leq_p$ Consequence:
  $F$ is valid if and only if $T \models F$ ($T$ is any valid formula).
- Consequence $\leq_p$ Validity:
  $F \models G$ if and only if $F \rightarrow G$ is valid.
- Validity $\leq_p$ Equivalence:
  $F$ is valid if and only if $F \equiv T$ ($T$ is any valid formula).
- Equivalence $\leq_p$ Validity:
  $F \equiv G$ if and only if $F \leftrightarrow G$ is valid.

# Interlude: Equivalence Relations

Let $R$ be a binary relation on the set $A$, i.e., $R \subseteq A \times A$.

- $R$ is reflexive if for all $a \in A$: $(a, a) \in R$.
- $R$ is symmetrical if for all $a, b \in A$:
  If $(a, b) \in R$, then also $(b, a) \in R$.
- $R$ is transitive if for all $a, b, c \in A$:
  If $(a, b) \in R$ and $(b, c) \in R$, then also $(a, c) \in R$.

A reflexive, symmetrical, and transitive relation is also called an equivalence relation.

For a binary relation $R$, we will also write $a\,R\,b$ instead of $(a, b) \in R$ (infix notation).

**Example:** For a natural number $k \geq 1$, we define the binary relation $\equiv_k$ on $\mathbb{Z}$: $n \equiv_k m$ if and only if $n - m$ is divisible by $k$.

Exercise: Prove that $\equiv_k$ is an equivalence relation for every $k \geq 1$.

# Interlude: Congruence Relations

Let $f$ be an $n$-ary operator on $A$, i.e., $f : A^n \to A$, where $A^n = \{(a_1, \ldots, a_n) \mid a_1, \ldots, a_n \in A\}$.

The binary relation $R \subseteq A \times A$ is closed under the operator $f$ if the following holds:

For all $(a_1, \ldots, a_n), (b_1, \ldots, b_n) \in A^n$:
If $a_1 \, R \, b_1$ and $\ldots$ $a_n \, R \, b_n$, then also $f(a_1, \ldots, a_n) \, R \, f(b_1, \ldots, b_n)$.

We also say that $R$ and $f$ are compatible.

Let $f_1, \ldots, f_n$ be operators on $A$ (of arbitrary arity).
$R$ is a congruence relation on $A$ (with respect to $f_1, \ldots, f_n$), if the following holds:

- $R$ is an equivalence relation.
- $R$ is closed under $f_1, \ldots, f_n$.

**Example:** $\equiv_k$ is a congruence relation on $\mathbb{Z}$ with respect to the 2-ary operators $+$ and $\cdot$ (multiplication).

# Equivalence is a Congruence Relation

The equivalence $\equiv$ of formulas is a binary relation on the set of all formulas: Let $\mathcal{F}$ be the set of all formulas. Then $\equiv \subseteq \mathcal{F} \times \mathcal{F}$.

$\wedge$ and $\vee$ are 2-ary operators on $\mathcal{F}$.
$\neg$ is a 1-ary operator on $\mathcal{F}$.

The equivalence $\equiv$ is a congruence relation on the set of all formulas (with respect to the operators $\wedge, \vee$, and $\neg$):

**reflexive:** It holds that $F \equiv F$ for every formula $F$ (every formula is equivalent to itself).

**symmetrical:** If $F \equiv G$ holds, then $G \equiv F$ also holds.

**transitive:** If $F \equiv G$ and $G \equiv H$ hold, then $F \equiv H$ also holds.

**closed under operators:** If $F_1 \equiv F_2$ and $G_1 \equiv G_2$ hold, then $(F_1 \wedge G_1) \equiv (F_2 \wedge G_2)$, $(F_1 \vee G_1) \equiv (F_2 \vee G_2)$, and $\neg F_1 \equiv \neg F_2$ also hold.

# Substitutability Theorem

The closure can also be formulated as follows:

**Substitutability Theorem**

Let $F$ and $G$ be equivalent formulas. Let $H$ be a formula with (at least) one occurrence of the subformula $F$. Then $H$ is equivalent to $H'$, where $H'$ is derived from $H$ by (somehow) replacing an occurrence of $F$ in $H$ with $G$.

# Proof of the Substitutability Theorem

**Proof** (by induction on the structure of the formula $H$):

**Base case:** If $H$ is an atomic formula, then it can only be $H = F$. It is clear that $H$ is equivalent to $H'$, since $H' = G$.

**Inductive step:** If $F$ is exactly $H$ itself, the same argument as in the base case applies.

So, let us assume that $F$ is a subformula of $H$ with $F \neq H$. We need to distinguish three cases.

# Proof of the Substitutability Theorem

**Case 1:** $H$ has the form $H = \neg H_1$.
By the inductive hypothesis, $H_1$ is equivalent to $H_1'$, where $H_1'$ is derived from $H_1$ by replacing $F$ with $G$.
Then, $H' = \neg H_1'$.
From the (semantic) definition of $\neg$, it follows that $H$ and $H'$ are equivalent.

**Case 2:** $H$ has the form $H = (H_1 \vee H_2)$.

Then $F$ appears in either $H_1$ or $H_2$. Let us assume the former case (the latter is entirely analogous).
By the inductive assumption, $H_1$ is again equivalent to $H_1'$, where $H_1'$ is derived from $H_1$ by replacing $F$ with $G$.
With the definition of $\vee$, it is clear that $H \equiv (H_1' \vee H_2) = H'$.

**Case 3:** $H$ has the form $H = (H_1 \wedge H_2)$.
This case can be proved entirely analogously to *Case 2*.

# Equivalences (I)

## Theorem

The following equivalences hold:

$$
\begin{array}{rcll}
(F \wedge F) & \equiv & F & \\
(F \vee F) & \equiv & F & \text{(Idempotency)} \\
 & & & \\
(F \wedge G) & \equiv & (G \wedge F) & \\
(F \vee G) & \equiv & (G \vee F) & \text{(Commutativity)} \\
 & & & \\
((F \wedge G) \wedge H) & \equiv & (F \wedge (G \wedge H)) & \\
((F \vee G) \vee H) & \equiv & (F \vee (G \vee H)) & \text{(Associativity)} \\
 & & & \\
(F \wedge (F \vee G)) & \equiv & F & \\
(F \vee (F \wedge G)) & \equiv & F & \text{(Absorption)} \\
 & & & \\
(F \wedge (G \vee H)) & \equiv & ((F \wedge G) \vee (F \wedge H)) & \\
(F \vee (G \wedge H)) & \equiv & ((F \vee G) \wedge (F \vee H)) & \text{(Distributivity)}
\end{array}
$$

# Equivalences (II)

**Theorem**

The following equivalences hold:

$$\neg\neg F \equiv F \qquad \text{(Double Negation)}$$

$$\begin{aligned} \neg(F \wedge G) &\equiv (\neg F \vee \neg G) \\ \neg(F \vee G) &\equiv (\neg F \wedge \neg G) \end{aligned} \qquad \text{(de Morgan's Laws)}$$

$$\begin{aligned} (F \vee G) &\equiv F, \text{ if } F \text{ is a tautology} \\ (F \wedge G) &\equiv G, \text{ if } F \text{ is a tautology} \end{aligned} \qquad \text{(Tautology Rules)}$$

$$\begin{aligned} (F \vee G) &\equiv G, \text{ if } F \text{ is unsatisfiable} \\ (F \wedge G) &\equiv F, \text{ if } F \text{ is unsatisfiable} \end{aligned} \qquad \text{(Unsatisfiability Rules)}$$

**Proof:** Exercise

# Normal Forms: **CNF**

**Definition** (Normal Forms)
A literal is an atomic formula or the negation of an atomic formula. In the first case, we speak of a positive literal, and in the second case, of a negative literal.

A formula $F$ is in conjunctive normal form (**CNF**) if it is a conjunction of disjunctions of literals:

$$F = (\bigwedge_{i=1}^{n} (\bigvee_{j=1}^{m_i} L_{i,j})),$$

where $L_{i,j} \in \{A_1, A_2, \ldots\} \cup \{\neg A_1, \neg A_2, \ldots\}$

**Example:** $A_1 \wedge \neg A_2 \wedge (\neg A_1 \vee A_2 \vee \neg A_3) \wedge (A_2 \vee A_4)$

# Normal Forms: **DNF**

A formula $F$ is in disjunctive normal form (**DNF**) if it is a disjunction of conjunctions of literals:

$$F = (\bigvee_{i=1}^{n} (\bigwedge_{j=1}^{m_i} L_{i,j})),$$

where $L_{i,j} \in \{A_1, A_2, \ldots\} \cup \{\neg A_1, \neg A_2, \ldots\}$

**Theorem**

For every formula $F$, there exists an equivalent formula in CNF, as well as an equivalent formula in DNF.

# Method 1: Reading from the Truth Table

For an atomic formula $A_i$, define

$$A_i^0 := \neg A_i \quad \text{and} \quad A_i^1 := A_i.$$

For a formula $F$, in which exactly the atomic formulas $A_1, \ldots, A_n$ occur, define

$$\mathrm{DNF}(F) \quad := \bigvee_{\substack{\mathcal{B}:\{A_1,\ldots,A_n\}\to\{0,1\},\\ \mathcal{B}(F)=1}} \bigwedge_{i=1}^{n} A_i^{\mathcal{B}(A_i)}$$

$$\mathrm{KNF}(F) \quad := \bigwedge_{\substack{\mathcal{B}:\{A_1,\ldots,A_n\}\to\{0,1\},\\ \mathcal{B}(F)=0}} \bigvee_{i=1}^{n} A_i^{1-\mathcal{B}(A_i)}$$

**Lemma 44**

*For every formula $F$, it holds that $F \equiv \mathrm{DNF}(F) \equiv \mathrm{KNF}(F)$.*

# Method 1: Reading from the Truth Table

**Proof:** We show that $F \equiv \mathrm{KNF}(F)$, $F \equiv \mathrm{DNF}(F)$ follows analogously.

Let $\mathcal{B}' : \{A_1, \ldots, A_n\} \to \{0, 1\}$ be an arbitrary assignment.

We show: $\mathcal{B}'(F) = 0$ if and only if $\mathcal{B}'(\mathrm{KNF}(F)) = 0$.

1. Assume $\mathcal{B}'(F) = 0$.

**Claim:** For all $i \in \{1, \ldots, n\}$, it holds that $\mathcal{B}'(A_i^{1-\mathcal{B}'(A_i)}) = 0$.
(This holds for every suitable assignment):

Case A: $\mathcal{B}'(A_i) = 0$.

Then it holds that $A_i^{1-\mathcal{B}'(A_i)} = A_i^1 = A_i$, and thus
$\mathcal{B}'(A_i^{1-\mathcal{B}'(A_i)}) = \mathcal{B}'(A_i) = 0$.

Case B: $\mathcal{B}'(A_i) = 1$.

Then it holds that $A_i^{1-\mathcal{B}'(A_i)} = A_i^0 = \neg A_i$, and thus
$\mathcal{B}'(A_i^{1-\mathcal{B}'(A_i)}) = \mathcal{B}'(\neg A_i) = 0$.

# Method 1: Reading from the Truth Table

From the claim, it follows that $\mathcal{B}'(\bigvee_{i=1}^{n} A_i^{1-\mathcal{B}'(A_i)}) = 0$.

Since $\mathcal{B}'(F) = 0$, $\mathcal{B}'$ is one of the assignments over which the large $\bigwedge$ in $\mathrm{KNF}(F)$ is formed.

Thus, there exists a formula $G$ such that

$$\mathrm{KNF}(F) \equiv G \wedge \bigvee_{i=1}^{n} A_i^{1-\mathcal{B}'(A_i)}$$

Due to $\mathcal{B}'(\bigvee_{i=1}^{n} A_i^{1-\mathcal{B}'(A_i)}) = 0$, it follows that $\mathcal{B}'(\mathrm{KNF}(F)) = 0$.

# Method 1: Reading from the Truth Table

2. Let $\mathcal{B}'(\mathrm{KNF}(F)) = 0$.

From

$$\mathrm{KNF}(F) = \bigwedge_{\substack{\mathcal{B}:\{A_1,\ldots,A_n\}\to\{0,1\},\\ \mathcal{B}(F)=0}} \bigvee_{i=1}^{n} A_i^{1-\mathcal{B}(A_i)}$$

it follows that one of the disjunctions in $\mathrm{KNF}(F)$ is equal to 0 under $\mathcal{B}'$.

Thus, there exists an assignment $\mathcal{B}$ with: $\mathcal{B}(F) = 0$ and $\mathcal{B}'(\bigvee_{i=1}^{n} A_i^{1-\mathcal{B}(A_i)}) = 0$.

Therefore: $\mathcal{B}'(A_i^{1-\mathcal{B}(A_i)}) = 0$ for all $i \in \{1, \ldots, n\}$.

This implies $\mathcal{B}'(A_i) = \mathcal{B}(A_i)$ for all $i \in \{1, \ldots, n\}$, and thus $\mathcal{B}'(F) = 0$ (because $\mathcal{B}(F) = 0$). □

# Method 1: Reading from the Truth Table

**Note:**

- If $F$ is unsatisfiable, i.e., $\mathcal{B}(F) = 0$ for all suitable assignments $\mathcal{B}$, then $\mathrm{DNF}(F)$ is the empty disjunction. This is to be considered an unsatisfiable formula.
- If $F$ is valid, i.e., $\mathcal{B}(F) = 1$ for all suitable assignments $\mathcal{B}$, then $\mathrm{KNF}(F)$ is the empty conjunction. This is to be considered a tautology.

# Method 1: Reading from Truth Table

| $A$ | $B$ | $C$ | $F$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

**DNF:** From each row with truth value 1, a conjunction is formed; from a 0 in the column $A$, we take $\neg A$, and from a 1, we take $A$.

$$(\neg A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge B \wedge C)$$
$$\vee (A \wedge \neg B \wedge \neg C) \vee (A \wedge B \wedge C)$$

**KNF:** From each row with truth value 0, a disjunction is formed; from a 0 in the column $A$, we take $A$, and from a 1, we take $\neg A$.

$$(A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C)$$
$$\wedge (\neg A \vee B \vee \neg C) \wedge (\neg A \vee \neg B \vee C)$$

# Method 2: Syntactic Transformation

*Given*: a formula $F$.
We form the KNF of $F$ as follows:

1. Replace each occurrence of a subformula of the form

$$\begin{array}{rcl} \neg\neg G & \text{by} & G \\ \neg(G \wedge H) & \text{by} & (\neg G \vee \neg H) \\ \neg(G \vee H) & \text{by} & (\neg G \wedge \neg H) \end{array}$$

   until no such subformulas remain.

2. Replace each occurrence of a subformula of the form

$$\begin{array}{rcl} (F \vee (G \wedge H)) & \text{by} & ((F \vee G) \wedge (F \vee H)) \\ ((F \wedge G) \vee H) & \text{by} & ((F \vee H) \wedge (G \vee H)) \end{array}$$

   until no such subformulas remain.

# Set Representation

A clause is a disjunction of literals.

The clause $L_1 \vee L_2 \vee \cdots \vee L_n$, where $L_1, \ldots, L_n$ are literals, is also identified with the set $\{L_1, \ldots, L_n\}$.

A formula in **KNF** (= conjunction of clauses) is identified with a set of clauses (i.e., a set of sets of literals):
$(\bigwedge_{i=1}^{n} (\bigvee_{j=1}^{m_i} L_{i,j}))$ is identified with $\{\{L_{i,j} \mid 1 \leq j \leq m_i\} \mid 1 \leq i \leq n\}$.

The empty clause (= empty disjunction) is equivalent to an unsatisfiable formula.

The empty **KNF** formula (= empty conjunction) is equivalent to a valid formula.

# Set Representation

**Example:** The set notation of the KNF

$$A_1 \wedge \neg A_2 \wedge (\neg A_1 \vee A_2 \vee \neg A_3) \wedge (A_2 \vee A_4)$$

is $\{\{A_1\}, \{\neg A_2\}, \{\neg A_1, A_2, \neg A_3\}, \{A_2, A_4\}\}$.

# Precedences

**Precedence of Operators:**

| | |
|---|---|
| $\leftrightarrow$ | has the weakest binding |
| $\rightarrow$ | ... |
| $\vee$ | ... |
| $\wedge$ | ... |
| $\neg$ | has the strongest binding |

Thus, it holds:

$$A \leftrightarrow B \vee \neg C \rightarrow D \wedge \neg E \equiv (A \leftrightarrow ((B \vee \neg C) \rightarrow (D \wedge \neg E)))$$

**Nevertheless:** Too many parentheses generally do not harm.

# Satisfiability and Validity in DNF and KNF

Satisfiability is easy (solvable in linear time) for formulas in DNF:

> *A formula in DNF is satisfiable if and only if there is a conjunction that does not simultaneously contain $A$ and $\neg A$ for an atomic formula $A$.*

Satisfiable: $(\neg B \wedge A \wedge B) \vee (\neg A \wedge C)$
Not satisfiable: $(A \wedge \neg A \wedge B) \vee (C \wedge \neg C)$

Validity is easy (solvable in linear time) for formulas in KNF:

> *A formula in KNF is valid if and only if every disjunction simultaneously contains $A$ and $\neg A$ for an atomic formula $A$ (or it is the empty conjunction).*

Valid: $(A \vee \neg A \vee B) \wedge (C \vee \neg C)$
Not valid: $(A \vee \neg A) \wedge (\neg A \vee C)$

# Efficient Satisfiability Tests

In the following:

- A very efficient satisfiability test for a special class of formulas, called Horn formulas
- A generally efficient unsatisfiability test for formulas in KNF (Resolution)

# Horn Formula

A formula $F$ is a Horn formula (named after Alfred Horn, 1918–2001) if $F$ is in **KNF** and each clause in $F$ contains at most one positive literal.

**Notation:**

$$\begin{array}{rll} (\neg A \vee \neg B \vee C) & \text{is rewritten as} & (A \wedge B \rightarrow C) \\ (\neg A \vee \neg B) & \text{is rewritten as} & (A \wedge B \rightarrow 0) \\ A & \text{is rewritten as} & (1 \rightarrow A) \end{array}$$

0: represents any unsatisfiable formula

1: represents any valid formula

**In general:**

$$\begin{array}{rcl} \neg A_1 \vee \cdots \vee \neg A_k \vee B & \equiv & \neg(A_1 \wedge \cdots \wedge A_k) \vee B \equiv (A_1 \wedge \cdots \wedge A_k) \rightarrow B \\ \neg A_1 \vee \cdots \vee \neg A_k & \equiv & \neg(A_1 \wedge \cdots \wedge A_k) \vee 0 \equiv (A_1 \wedge \cdots \wedge A_k) \rightarrow 0 \end{array}$$

$(A_1 \wedge \cdots \wedge A_k) \rightarrow B$ (or $(A_1 \wedge \cdots \wedge A_k) \rightarrow 0$) is the implicative form of the clause $\neg A_1 \vee \cdots \vee \neg A_k \vee B$ (or $\neg A_1 \vee \cdots \vee \neg A_k$).

# Satisfiability Test for Horn Formulas

**Marking Algorithm**

**Input:** a Horn formula $F$.

(1) Mark each occurrence of an atomic formula $A$ in $F$ with a mark if there is a subformula of the form $(1 \to A)$ in $F$;

(2) **while** there is in $F$ a subformula $G$ of the form $(A_1 \wedge \ldots \wedge A_k \to B)$ or $(A_1 \wedge \ldots \wedge A_k \to 0)$, $k \geq 1$, where $A_1, \ldots, A_k$ are already marked and $B$ is not yet marked **do**:

    **if** $G$ has the first form **then**

        mark each occurrence of $B$

    **else** output "unsatisfiable" and stop;

    **endwhile**

(3) Output "satisfiable" and stop.

# Examples for the Marking Algorithm

We apply the marking algorithm to the following Horn formula:

$$(1 \to A_1) \wedge (1 \to A_2) \wedge (A_1 \wedge A_2 \to A_3) \wedge (A_2 \wedge A_3 \to A_4) \wedge (A_1 \wedge A_3 \wedge A_5 \to 0)$$

# Examples for the Marking Algorithm

We apply the marking algorithm to the following Horn formula:

$$(1 \to A_1) \wedge (1 \to A_2) \wedge (A_1 \wedge A_2 \to A_3) \wedge (A_2 \wedge A_3 \to A_4) \wedge (A_1 \wedge A_3 \wedge A_5 \to 0)$$

# Examples for the Marking Algorithm

We apply the marking algorithm to the following Horn formula:

$$(1 \to A_1) \wedge (1 \to A_2) \wedge (A_1 \wedge A_2 \to A_3) \wedge (A_2 \wedge A_3 \to A_4) \wedge (A_1 \wedge A_3 \wedge A_5 \to 0)$$

# Examples for the Marking Algorithm

We apply the marking algorithm to the following Horn formula:

$$(1 \to A_1) \wedge (1 \to A_2) \wedge (A_1 \wedge A_2 \to A_3) \wedge (A_2 \wedge A_3 \to A_4) \wedge (A_1 \wedge A_3 \wedge A_5 \to 0)$$

# Examples for the Marking Algorithm

We apply the marking algorithm to the following Horn formula:

$$(1 \to A_1) \wedge (1 \to A_2) \wedge (A_1 \wedge A_2 \to A_3) \wedge (A_2 \wedge A_3 \to A_4) \wedge (A_1 \wedge A_3 \wedge A_5 \to 0)$$

The formula is satisfiable.

# Examples for the Marking Algorithm

We apply the marking algorithm to the following Horn formula:

$$(1 \to A_1) \wedge (1 \to A_2) \wedge (A_1 \wedge A_2 \to A_3) \wedge (A_2 \wedge A_3 \to A_4) \wedge (A_1 \wedge A_3 \wedge A_5 \to 0)$$

The formula is satisfiable.

We apply the marking algorithm to the following Horn formula:

$$(1 \to A_1) \wedge (1 \to A_2) \wedge (A_1 \wedge A_2 \to A_3) \wedge (A_2 \wedge A_3 \to A_5) \wedge (A_1 \wedge A_3 \wedge A_5 \to 0)$$

# Examples for the Marking Algorithm

We apply the marking algorithm to the following Horn formula:

$$(1 \to A_1) \wedge (1 \to A_2) \wedge (A_1 \wedge A_2 \to A_3) \wedge (A_2 \wedge A_3 \to A_4) \wedge (A_1 \wedge A_3 \wedge A_5 \to 0)$$

The formula is satisfiable.

We apply the marking algorithm to the following Horn formula:

$$(1 \to A_1) \wedge (1 \to A_2) \wedge (A_1 \wedge A_2 \to A_3) \wedge (A_2 \wedge A_3 \to A_5) \wedge (A_1 \wedge A_3 \wedge A_5 \to 0)$$

# Examples for the Marking Algorithm

We apply the marking algorithm to the following Horn formula:

$$(1 \to A_1) \wedge (1 \to A_2) \wedge (A_1 \wedge A_2 \to A_3) \wedge (A_2 \wedge A_3 \to A_4) \wedge (A_1 \wedge A_3 \wedge A_5 \to 0)$$

The formula is satisfiable.

We apply the marking algorithm to the following Horn formula:

$$(1 \to A_1) \wedge (1 \to A_2) \wedge (A_1 \wedge A_2 \to A_3) \wedge (A_2 \wedge A_3 \to A_5) \wedge (A_1 \wedge A_3 \wedge A_5 \to 0)$$

# Examples for the Marking Algorithm

We apply the marking algorithm to the following Horn formula:

$$(1 \to A_1) \wedge (1 \to A_2) \wedge (A_1 \wedge A_2 \to A_3) \wedge (A_2 \wedge A_3 \to A_4) \wedge (A_1 \wedge A_3 \wedge A_5 \to 0)$$

The formula is satisfiable.

We apply the marking algorithm to the following Horn formula:

$$(1 \to A_1) \wedge (1 \to A_2) \wedge (A_1 \wedge A_2 \to A_3) \wedge (A_2 \wedge A_3 \to A_5) \wedge (A_1 \wedge A_3 \wedge A_5 \to 0)$$

# Examples for the Marking Algorithm

We apply the marking algorithm to the following Horn formula:

$$(1 \to A_1) \wedge (1 \to A_2) \wedge (A_1 \wedge A_2 \to A_3) \wedge (A_2 \wedge A_3 \to A_4) \wedge (A_1 \wedge A_3 \wedge A_5 \to 0)$$

The formula is satisfiable.

We apply the marking algorithm to the following Horn formula:

$$(1 \to A_1) \wedge (1 \to A_2) \wedge (A_1 \wedge A_2 \to A_3) \wedge (A_2 \wedge A_3 \to A_5) \wedge (A_1 \wedge A_3 \wedge A_5 \to 0)$$

The formula is not satisfiable.

# Principle of Induction

To prove the statement

*For every* $n \in \{0, 1, 2, 3, \ldots\}$, $P(n)$ *holds.*

we generally proceed as follows:

- We show that $P(0)$ holds. (Base case)
- We show that for every $n$:
  If $P(n)$ holds, then $P(n+1)$ also holds. (Inductive step)

Then we can conclude that $P(n)$ holds for every arbitrary $n$.

Alternative principle of induction: We show that for every $n$:
If $P(k)$ holds for all $k < n$, then $P(n)$ also holds.

**Application:** Proof that a condition is always satisfied during the execution of an algorithm (invariant). To do this, one shows by induction that the condition is satisfied after $n$ steps of the algorithm.

# Correctness of the Marking Algorithm

**Theorem 45**

*The marking algorithm is correct and always terminates after at most $n$ marking steps.*
*Here, $n$ is the number of atomic formulas in the input formula $F$.*

**Proof:**

(A) Algorithm terminates:

After at most $n$ steps, all atomic formulas are marked.

(B) If the algorithm marks an atomic formula $A$, then $\mathcal{B}(A) = 1$ for every satisfying assignment $\mathcal{B}$ of $F$.

Proof of (B) by induction:

1st case: atomic formula $A$ is marked in step (1): clear

# Correctness of the Marking Algorithm

2nd case: atomic formula $A$ is marked in step (2):

Then there is a subformula $(A_1 \wedge \ldots \wedge A_k \rightarrow A)$, such that $A_1, \ldots, A_k$ were marked at earlier times.

Thus, it holds that $\mathcal{B}(A_1) = \cdots = \mathcal{B}(A_k) = 1$ for every satisfying assignment $\mathcal{B}$ of $F$.

However, it must also hold that $\mathcal{B}(A) = 1$ for every satisfying assignment $\mathcal{B}$ of $F$.

This proves (B).

# Correctness of the Marking Algorithm

(C) If the algorithm outputs "unsatisfiable", then $F$ is unsatisfiable.

Let $(A_1 \wedge \ldots \wedge A_k \rightarrow 0)$ be the subformula of $F$, which causes the output "unsatisfiable".

By (B), it holds that $\mathcal{B}(A_1) = \cdots = \mathcal{B}(A_k) = 1$ for every satisfying assignment $\mathcal{B}$ of $F$.

But for such assignments, it holds: $\mathcal{B}(A_1 \wedge \cdots \wedge A_k \rightarrow 0) = 0$ and thus $\mathcal{B}(F) = 0$.

So there cannot be a satisfying assignment for $F$.

(D) If the algorithm outputs "satisfiable", then $F$ is satisfiable.

Assume the algorithm outputs "satisfiable".

Define an assignment $\mathcal{B}$ as follows:

$$\mathcal{B}(A_i) = \begin{cases} 1 & \text{if the algorithm marks } A_i \\ 0 & \text{otherwise} \end{cases}$$

We claim that the assignment $\mathcal{B}$ satisfies the conjunction $F$:

- In $(A_1 \wedge \ldots \wedge A_k \rightarrow B)$, $B$ is marked or at least one $A_i$ is not marked.
- In $(A_1 \wedge \ldots \wedge A_k \rightarrow 0)$, at least one $A_i$ is not marked (otherwise the algorithm would have terminated with "unsatisfiable").

□

**Remark:** With a suitable implementation, the algorithm runs in linear time.

# Example: MYCIN

MYCIN: Expert system for the investigation of blood infections (developed in the 1970s)

**Example:**
IF the infection is primary-bacteremia AND the site of the culture is one of the sterile sites AND the suspected portal of entry is the gastrointestinal tract THEN there is suggestive evidence (0.7) that the infection is bacteroid.

# Resolution (Idea)

Resolution is a procedure used to determine whether a formula $F$ in **CNF** is unsatisfiable.

**Idea:**

$$(F \vee A) \wedge (F' \vee \neg A) \equiv (F \vee A) \wedge (F' \vee \neg A) \wedge (F \vee F')$$

From the derivation of the empty disjunction (i.e., empty clause) follows unsatisfiability.

Two questions:

- Can we always derive the empty clause from an unsatisfiable formula? (Completeness)
- Is there a way to write the derivation more compactly?

# Set Representation

To recall:

- Clause: Set of literals (disjunction).

$$\{A, B\} \text{ represents } (A \vee B).$$

- Formula in **CNF**: Set of clauses (conjunction of clauses).

$$\{\{A, B\}, \{\neg A, B\}\} \text{ represents } ((A \vee B) \wedge (\neg A \vee B)).$$

The empty clause (i.e., empty disjunction) is equivalent to an unsatisfiable formula.
This is also denoted by $\square$.

The empty CNF formula (i.e., empty conjunction) is equivalent to a valid formula.

Note: The CNF formula $\{\}$ (i.e., empty conjunction = tautology) is to be distinguished from the formula $\{\square\}$ (i.e., unsatisfiable formula).

# Advantages of Set Representation

One automatically obtains:

- Commutativity:
  $(A \vee B) \equiv (B \vee A)$,
  both represented by $\{A, B\}$
- Associativity:
  $((A \vee B) \vee C) \equiv (A \vee (B \vee C))$,
  both represented by $\{A, B, C\}$
- Idempotence:
  $(A \vee A) \equiv A$,
  both represented by $\{A\}$

# Resolvent

**Definition:** Let $K_1$, $K_2$, and $R$ be clauses. Then $R$ is called the resolvent of $K_1$ and $K_2$ if there exists a literal $L$ such that $L \in K_1$ and $\overline{L} \in K_2$, and $R$ has the following form:

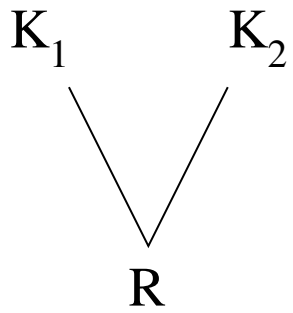
$$R = (K_1 \setminus \{L\}) \cup (K_2 \setminus \{\overline{L}\}).$$

Here, $\overline{L}$ is defined as

$$\overline{L} = \begin{cases} \neg A_i & \text{if } L = A_i \text{ for some } i \geq 1, \\ A_i & \text{if } L = \neg A_i \text{ for some } i \geq 1. \end{cases}$$

# Resolvent

We represent this fact with the following diagram:




Terminology: $R$ is resolved from $K_1$, $K_2$ by $L$.

Furthermore, if $K_1 = \{L\}$ and $K_2 = \{\overline{L}\}$, then the empty set arises as the resolvent. This is denoted by the special symbol $\square$, which represents an unsatisfiable formula.

**Example:** All resolvents of $\{A, \neg B, \neg C\}$ and $\{\neg A, B, D\}$:

$$\{\neg B, B, \neg C, D\} \text{ and } \{A, \neg A, \neg C, D\}$$

# Resolutionslemma

## Resolutionslemma

Let $F$ be a formula in **CNF**, represented as a set of clauses. Furthermore, let $R$ be a resolvent of two clauses $K_1$ and $K_2$ in $F$. Then $F$ and $F \cup \{R\}$ are equivalent.

**Proof:** Follows directly from

$$\underbrace{(F_1 \vee A)}_{K_1} \wedge \underbrace{(F_2 \vee \neg A)}_{K_2} \equiv \underbrace{(F_1 \vee A)}_{K_1} \wedge \underbrace{(F_2 \vee \neg A)}_{K_2} \wedge \underbrace{(F_1 \vee F_2)}_{R}$$

# Resolution Closure

**Definition:** Let $F$ be a set of clauses. Then $Res(F)$ is defined as

$$Res(F) = F \cup \{R \mid R \text{ is a resolvent of two clauses in } F\}.$$

Additionally, we set:

$$\begin{aligned} Res^0(F) &= F \\ Res^{n+1}(F) &= Res(Res^n(F)) \qquad \text{for } n \geq 0 \end{aligned}$$

and finally let

$$Res^*(F) = \bigcup_{n \geq 0} Res^n(F).$$

$Res^*(F)$ is also referred to as the resolution closure of $F$.

From the resolution lemma, it immediately follows that

$$F \equiv Res^*(F).$$

# Resolution Closure

Assume the formula $F$ (in **CNF**) contains $n$ atomic formulas.
What is the maximum possible size of $Res^*(F)$?

(A) $|Res^*(F)| \leq 2^n$

(B) $|Res^*(F)| \leq 4^n$

(C) $|Res^*(F)|$ can become infinite

Here, $|Res^*(F)|$ denotes the number of elements in $Res^*(F)$.

**Example:** The resolution closure of $\{\{A, \neg B, \neg C\}, \{\neg A, B, D\}\}$ consists of the following clauses:

$$\{A, \neg B, \neg C\}, \{\neg A, B, D\},$$
$$\{\neg B, B, \neg C, D\}, \{A, \neg A, \neg C, D\}$$
$$\{A, \neg B, \neg C, D\}, \{\neg A, B, \neg C, D\}$$

# Resolution Theorem

We now show the Correctness and Completeness of Resolution:

**Resolution Theorem of Propositional Logic**

A finite set $F$ of clauses is unsatisfiable if and only if $\square \in Res^*(F)$.

**Proof:**

**Correctness:** If $\square \in Res^*(F)$, then $F$ is unsatisfiable.

Assume $\square \in Res^*(F)$.

From the resolution lemma, we have $F \equiv Res^*(F)$.

Since $\square$ is unsatisfiable, $Res^*(F)$ is also unsatisfiable, and thus $F$ is unsatisfiable.

# Proof of the Resolution Theorem (Completeness)

**Completeness:** If $F$ is unsatisfiable, then $\square \in Res^*(F)$.

Let $F$ be unsatisfiable.

We prove completeness by induction on the number $n(F)$ of atomic formulas occurring in $F$.

Base case: $n(F) = 0$. Then it must hold that $F = \{\square\}$. Thus, $\square \in F \subseteq Res^*(F)$.

Inductive step: Assume $n(F) > 0$.

Choose any atomic formula $A$ that occurs in $F$.
We define the formula $F_0$ from $F$ as follows:

$$F_0 = \{K \setminus \{A\} \mid K \in F, \neg A \notin K\}.$$

Intuition: $F_0$ is derived from $F$ by replacing $A$ with 0 and performing the öbviousßimplifications.

# Proof of the Resolution Theorem (Completeness)

Analogously, we define the formula $F_1$ from $F$ as:

$$F_1 = \{K \setminus \{\neg A\} \mid K \in F, A \notin K\}.$$

Intuition: $F_1$ is derived from $F$ by replacing $A$ with 1 and performing the öbviousßimplifications.

Since $F$ is unsatisfiable, both $F_0$ and $F_1$ are also unsatisfiable:

If, for example, $F_0$ were to be satisfied by the assignment $\mathcal{B}$, then $F$ would be satisfied by the following assignment $\mathcal{B}'$:

$$\mathcal{B}'(A_i) = \begin{cases} 0 & \text{if } A_i = A \\ \mathcal{B}(A_i) & \text{if } A_i \neq A \end{cases}$$

Since $n(F_0) = n(F_1) = n(F) - 1$, we can conclude from the induction hypothesis that $\square \in Res^*(F_0)$ and $\square \in Res^*(F_1)$.

# Proof of the Resolution Theorem (Completeness)

Thus, there exists a sequence of clauses $K_1, K_2, \ldots, K_m$ such that

- $K_m = \square$
- $K_i \in F_0$ or $K_i$ is a resolvent of $K_j$ and $K_\ell$ with $j, \ell < i$.

We now define a sequence of clauses $K'_1, K'_2, \ldots, K'_m$, where $K'_i = K_i$ or $K'_i = K_i \cup \{A\}$, as follows:

1. Case $K_i \in F_0$ and $K_i \in F$: $K'_i := K_i$
2. Case $K_i \in F_0$ and $K_i \notin F$: $K'_i := K_i \cup \{A\} \in F$
3. Case $K_i \notin F_0$ and $K_i$ is resolved from $K_j$ and $K_\ell$ $(j, \ell < i)$ on literal $L$: $K'_i$ is formed from $K'_j$ and $K'_\ell$ by resolving on $L$.

Then either $K'_m = \square$ or $K'_m = \{A\}$, and thus

$$\square \in Res^*(F) \quad \text{or} \quad \{A\} \in Res^*(F)$$

Similarly, by considering $F_1$, we have:

$$\square \in Res^*(F) \quad \text{or} \quad \{\neg A\} \in Res^*(F)$$

From this, we conclude $\square \in Res^*(F)$. $\square$

# Illustration of the Induction Step

$F = \{\{A_1\}, \{\neg A_2,\ A_4\}, \{\neg A_1, A_2,\ A_4\}, \{A_3, \neg A_4\},\ \{\neg A_1, \neg A_3, \neg A_4\}\ \}$

# Illustration of the Induction Step

$F = \{\{A_1\}, \{\neg A_2, \cancel{A_4}\}, \{\neg A_1, A_2, \cancel{A_4}\}, \cancel{\{A_3, \neg A_4\}}, \cancel{\{\neg A_1, \neg A_3, \neg A_4\}}\}$

$F_0 = \{\{A_1\}, \{\neg A_2\}, \{\neg A_1, A_2\}\}$

# Illustration of the Induction Step

$F = \{\{A_1\}, \cancel{\{\neg A_2, A_4\}}, \cancel{\{\neg A_1, A_2, A_4\}}, \{A_3, \cancel{\neg A_4}\}, \{\neg A_1, \neg A_3, \cancel{\neg A_4}\}\}$

$F_0 = \{\{A_1\}, \{\neg A_2\}, \{\neg A_1, A_2\}\}$

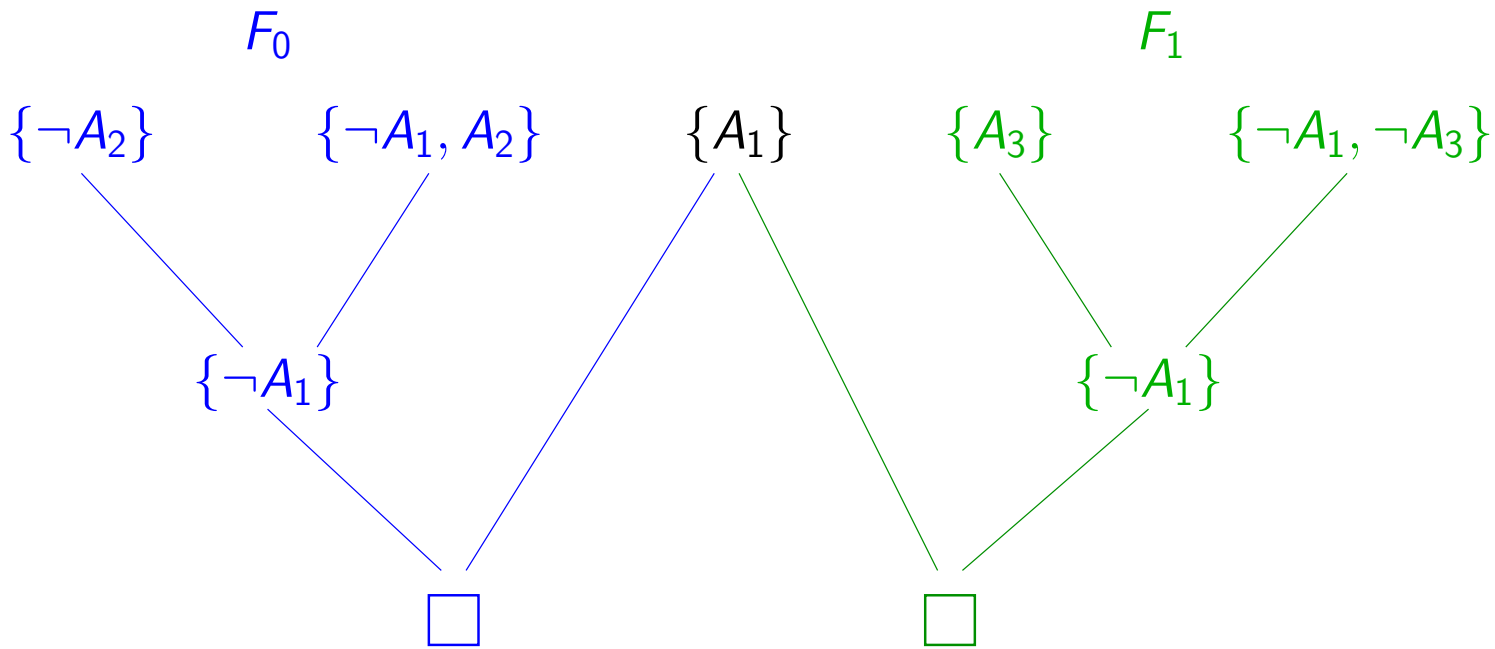
$F_1 = \{\{A_1\}, \{A_3\}, \{\neg A_1, \neg A_3\}\}$

# Illustration of the Induction Step

$F = \{\{A_1\}, \{\neg A_2, A_4\}, \{\neg A_1, A_2, A_4\}, \{A_3, \neg A_4\}, \{\neg A_1, \neg A_3, \neg A_4\}\}$

$F_0 = \{\{A_1\}, \{\neg A_2\}, \{\neg A_1, A_2\}\}$

$F_1 = \{\{A_1\}, \{A_3\}, \{\neg A_1, \neg A_3\}\}$

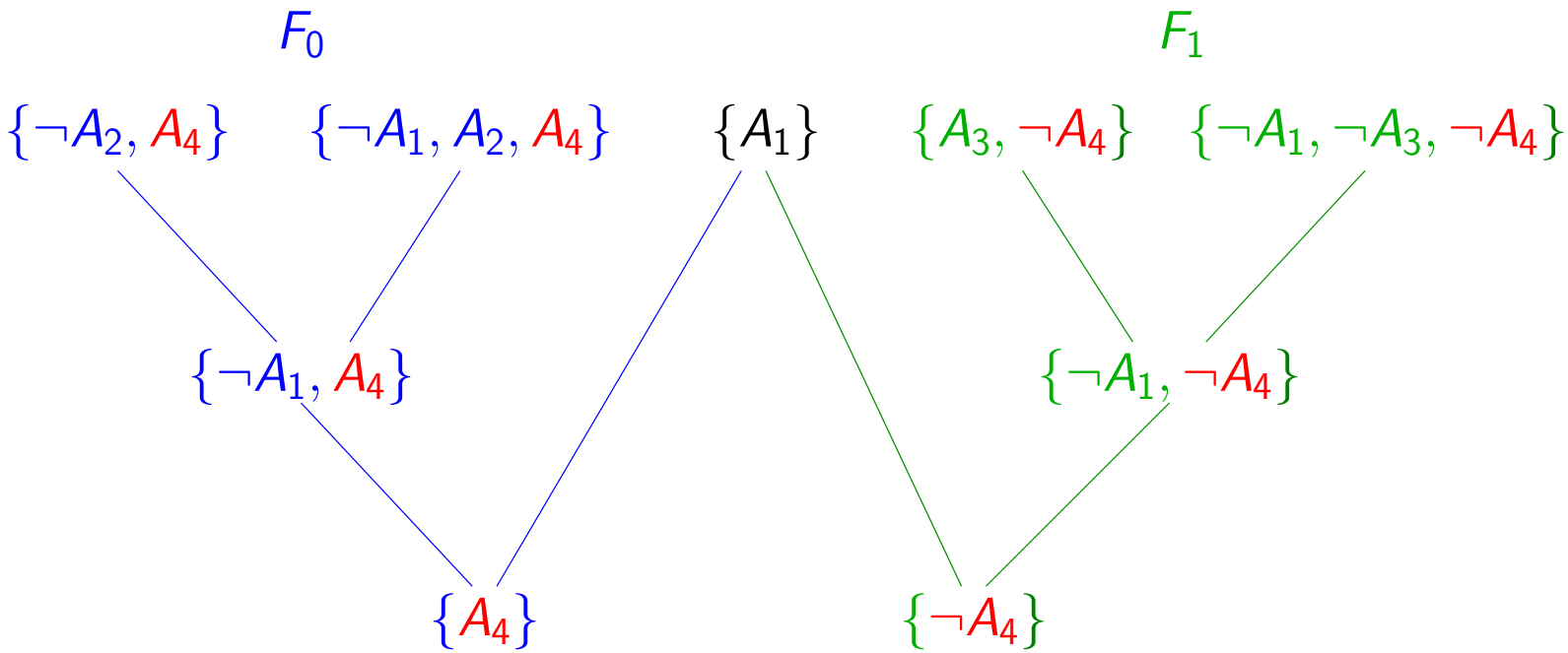


# Illustration of the Induction Step

$F = \{\{A_1\}, \{\neg A_2, A_4\}, \{\neg A_1, A_2, A_4\}, \{A_3, \neg A_4\}, \{\neg A_1, \neg A_3, \neg A_4\}\}$

$F_0 = \{\{A_1\}, \{\neg A_2\}, \{\neg A_1, A_2\}\}$

$F_1 = \{\{A_1\}, \{A_3\}, \{\neg A_1, \neg A_3\}\}$

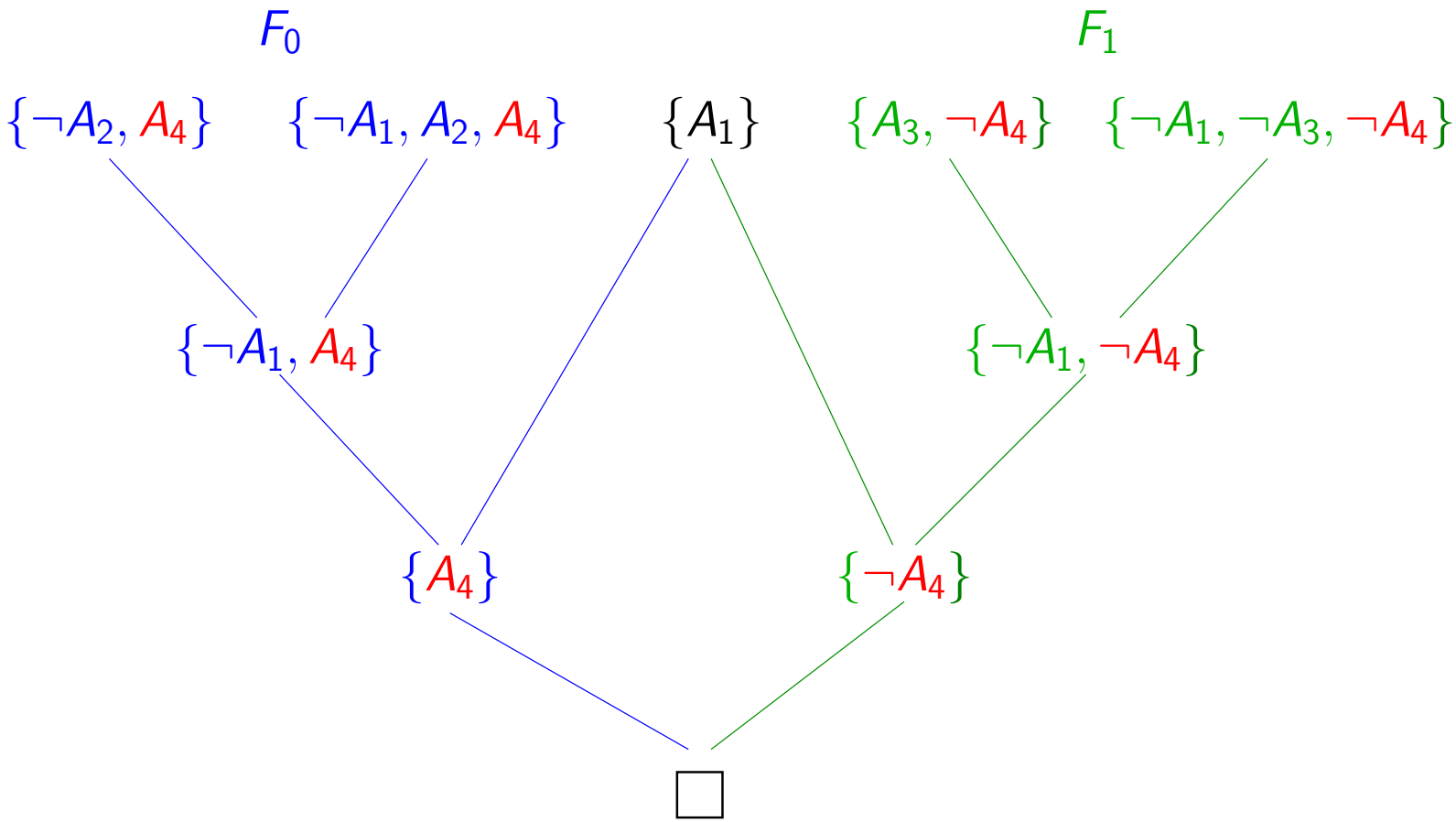


# Illustration of the Induction Step

$F = \{\{A_1\}, \{\neg A_2, A_4\}, \{\neg A_1, A_2, A_4\}, \{A_3, \neg A_4\}, \{\neg A_1, \neg A_3, \neg A_4\}\}$

$F_0 = \{\{A_1\}, \{\neg A_2\}, \{\neg A_1, A_2\}\}$

$F_1 = \{\{A_1\}, \{A_3\}, \{\neg A_1, \neg A_3\}\}$



# Deduction

A deduction (or derivation or proof) of the empty clause from a set of clauses $F$ is a sequence of clauses $K_1, K_2, \ldots, K_m$ with the following properties:

> *$K_m$ is the empty clause, and for every $i \in \{1, \ldots, m\}$, $K_i$ is an element of $F$ or can be resolved from certain clauses $K_j$, $K_\ell$ with $j, \ell < i$.*

From the resolution theorem, it follows:
A finite set of clauses is unsatisfiable if and only if a deduction of the empty clause exists.

**Note**: It may be the case that $Res^*(F)$ is very large, but a short deduction of the empty clause still exists.

# Resolution Calculus

The term calculus refers to a set of syntactic transformation rules that allow one to derive semantic properties of the input formula.

For the resolution calculus:

- Syntactic transformation rules: resolution, stopping upon reaching the empty clause
- Semantic property of the input formula: unsatisfiability

Desirable properties of a calculus:

- Correctness: If the empty clause can be derived from $F$, then $F$ is unsatisfiable.
- Completeness: If $F$ is unsatisfiable, then the empty clause can be derived from $F$.

# Example of Resolution

We want to show that

$$((AK \vee BK) \wedge (AK \rightarrow BK) \wedge (BK \wedge RL \rightarrow \neg AK) \wedge RL) \rightarrow (\neg AK \wedge BK)$$

is valid.

This is the case if and only if

$$(AK \vee BK) \wedge (\neg AK \vee BK) \wedge (\neg BK \vee \neg RL \vee \neg AK) \wedge RL \wedge (AK \vee \neg BK)$$

is unsatisfiable. (Because: $F \rightarrow G$ is valid iff $F \wedge \neg G$ is unsatisfiable.)

In set representation:

$$\{\{AK, BK\},\ \{\neg AK, BK\},\ \{\neg BK, \neg RL, \neg AK\},\ \{RL\},\ \{AK, \neg BK\}\}$$

# Example of Resolution

A possible deduction of the empty clause from

$$\{\{AK, BK\}, \{\neg AK, BK\}, \{\neg BK, \neg RL, \neg AK\}, \{RL\}, \{AK, \neg BK\}\} \quad (6)$$

is as follows:

# Example of Resolution

A possible deduction of the empty clause from

$$\{\{AK, BK\},\ \{\neg AK, BK\},\ \{\neg BK, \neg RL, \neg AK\},\ \{RL\},\ \{AK, \neg BK\}\} \quad (6)$$

is as follows:

| | | |
|---|---|---|
| $\{AK, BK\}$ | is from (6) | (7) |
| $\{\neg AK, BK\}$ | is from (6) | (8) |
| $\{BK\}$ | from (7) and (8) | (9) |
| $\{\neg BK, \neg RL, \neg AK\}$ | is from (6) | (10) |
| $\{AK, \neg BK\}$ | is from (6) | (11) |
| $\{\neg BK, \neg RL\}$ | from (10) and (11) | (12) |
| $\{RL\}$ | is from (6) | (13) |
| $\{\neg BK\}$ | from (12) and (13) | (14) |
| $\square$ | from (9) and (14) | (15) |

# Remarks on Resolution

Armin Haken provided in 1985 a set of clauses $F_n$ for each $n$ with:

- $F_n$ is unsatisfiable.
- $F_n$ contains $n \cdot (n+1)$ atomic subformulas.
- $F_n$ consists of $\frac{n^3+n^2}{2} + n + 1$ clauses.
- Any deduction of the empty clause from $F_n$ has a length of at least $c^n$ for some fixed constant $c > 1$.

Conclusion: Deductions can become very lengthy.

# The Compactness Theorem of Propositional Logic

**Satz 46 (Compactness Theorem)**

*Let $M$ be a (possibly infinite) set of formulas. Then $M$ is satisfiable if and only if every finite subset of $M$ is satisfiable.*

**Proof:**

**1.** If $M$ is satisfiable, then every finite subset of $M$ is satisfiable.

This statement is trivial because every model of $M$ is also a model for any subset of $M$.

# The Compactness Theorem of Propositional Logic: Proof

**2.** If every finite subset of $M$ is satisfiable, then $M$ is also satisfiable.

Recall: The set of atomic formulas is $\{A_1, A_2, A_3, \ldots\}$.

Let $M_n \subseteq M$ be the set of formulas in $M$ that contain only atomic subformulas from $\{A_1, \ldots, A_n\}$.

**Note:** $M_n$ may be infinite; for example, $M_1$ could contain the formulas $A_1, A_1 \wedge A_1, A_1 \wedge A_1 \wedge A_1, \ldots$.

**But:** There are only $2^{2^n}$ different truth tables with the atomic formulas $A_1, \ldots, A_n$.

Therefore, there exists a finite subset $M'_n \subseteq M_n$ with:

1. $|M'_n| \leq 2^{2^n}$
2. For every formula $F \in M_n$, there exists a formula $F' \in M'_n$ such that $F \equiv F'$

# The Compactness Theorem of Propositional Logic: Proof

According to the assumption, $M_n'$ has a model $\mathcal{B}_n$.

Thus, $\mathcal{B}_n$ is also a model of $M_n$.

We now construct a model $\mathcal{B}$ of $M$ in stages.

In stage $n$, the value $\mathcal{B}(A_n) \in \{0, 1\}$ is determined.

$I_0 := \{1, 2, 3, \ldots\}$
**for all** $n \geq 1$ **do**
    **if** there are infinitely many indices $i \in I_{n-1}$ with $\mathcal{B}_i(A_n) = 1$ **then**
        $\mathcal{B}(A_n) := 1$
        $I_n := \{i \in I_{n-1} \mid \mathcal{B}_i(A_n) = 1\}$
    **else** ($***$ there are infinitely many $i \in I_{n-1}$ with $\mathcal{B}_i(A_n) = 0$ $***$)
        $\mathcal{B}(A_n) := 0$
        $I_n := \{i \in I_{n-1} \mid \mathcal{B}_i(A_n) = 0\}$
**endfor**

# The Compactness Theorem of Propositional Logic: Proof

Although it holds that $I_0 \supseteq I_1 \supseteq I_2 \supseteq I_3 \cdots$, we have:

**Claim 1:** For every $n \geq 0$, $I_n$ is infinite.

This is shown by induction over $n$:

$I_0 = \{1, 2, 3, \ldots\}$ is obviously infinite.

If $I_{n-1}$ is infinite, then by construction, $I_n$ is also infinite, since it holds that

$$I_{n-1} = \{i \in I_{n-1} \mid \mathcal{B}_i(A_n) = 1\} \cup \{i \in I_{n-1} \mid \mathcal{B}_i(A_n) = 0\}.$$

**Claim 2:** For all $n \geq 1$ and all $i \in I_n$, it holds that:

$$\mathcal{B}_i(A_1) = \mathcal{B}(A_1), \ldots, \mathcal{B}_i(A_{n-1}) = \mathcal{B}(A_{n-1}),\ \mathcal{B}_i(A_n) = \mathcal{B}(A_n).$$

Let $i \in I_n$. Then, by the construction of $\mathcal{B}$, we have $\mathcal{B}_i(A_n) = \mathcal{B}(A_n)$.

Now, let $j \in \{1, \ldots, n-1\}$.

Since $I_n \subseteq I_j$, it follows that $i \in I_j$, and thus again $\mathcal{B}_i(A_j) = \mathcal{B}(A_j)$.

# The Compactness Theorem of Propositional Logic: Proof

**Claim 3:** The constructed assignment $\mathcal{B}$ is a model of $M$.

**Proof of Claim 3:** Let $F \in M$.

Then there exists an $n \geq 1$ with $F \in M_n$ (since $F$ contains only finitely many atomic formulas).

Thus, it holds that $F \in M_i$ for all $i \geq n$.

Therefore, each of the assignments $\mathcal{B}_i$ with $i \geq n$ is a model of $F$.

Since $I_n$ is infinite by Claim 1, there exists an $i \geq n$ with $i \in I_n$.

For this $i$, it holds by Claim 2:

$$\mathcal{B}_i(A_1) = \mathcal{B}(A_1), \quad \mathcal{B}_i(A_2) = \mathcal{B}(A_2), \ldots, \mathcal{B}_i(A_n) = \mathcal{B}(A_n)$$

Since $\mathcal{B}_i$ is a model of $F$ due to $i \geq n$, it follows that $\mathcal{B}$ is also a model of $F$. □

# Consequences of the Compactness Theorem

The following statement has only been proven for a finite set of clauses $F$.

**Resolution Theorem of Propositional Logic for Arbitrary Sets of Formulas**

A set $F$ of clauses is unsatisfiable if and only if $\square \in Res^*(F)$.

**Proof:**

**Correctness:** If $\square \in Res^*(F)$, then $F$ is unsatisfiable:
Proof as in the finite case.

**Completeness:** If $F$ is unsatisfiable, then $\square \in Res^*(F)$.

If $F$ is unsatisfiable, then by the Compactness Theorem, there must exist a finite subset $F' \subseteq F$ that is also unsatisfiable.

From the already proven Resolution Theorem for finite sets of formulas, it follows that $\square \in Res^*(F')$.

Thus, it also holds that $\square \in Res^*(F)$. $\square$

# Predicate Logic

Propositional logic is generally too weak in expressing mathematical statements.

**Example:** In analysis, one wants to formulate statements of the form

> *For all* $x \in \mathbb{R}$ *and for all* $\varepsilon > 0$, *there exists a* $\delta > 0$ *such that for all* $y \in \mathbb{R}$, *if* $\mathit{abs}(x - y) < \delta$, *then* $\mathit{abs}(f(x) - f(y)) < \varepsilon$.

In this context, we use:

- Relations such as $<$ or $>$.
- Functions such as $\mathrm{abs} : \mathbb{R} \to \mathbb{R}_+$ (absolute value) or $- : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ (subtraction).
- Quantifications such as "for all $x \in \mathbb{R}$" or "there exists a $\delta > 0$".

This leads to predicate logic.

# Syntax of Predicate Logic: Variables, Terms

The set of variables is $\{x_0, x_1, x_2, \ldots\}$.

A predicate symbol has the form $P_i^k$ with $i, k \in \{0, 1, 2, \ldots\}$.

A function symbol has the form $f_i^k$ with $i, k \in \{0, 1, 2, \ldots\}$.

In both cases, $i$ is called the distinguishing index and $k$ is the arity.

Variables are also denoted by $u, x, y, z$,
predicate symbols (function symbols) are also denoted by $P, Q, R$ ($f, g, h$).

We now define the set of terms inductively:

1. Every variable is a term.
2. If $f$ is a function symbol with arity $k$, and if $t_1, \ldots, t_k$ are terms, then $f(t_1, \ldots, t_k)$ is also a term.

Function symbols of arity 0 are also included, and in this case, the parentheses are omitted.

Zero-arity function symbols are called constants
(usually denoted by $a, b, c$).

# Formulas

We can now (again inductively) define what formulas (of predicate logic) are.

1. If $P$ is a predicate symbol of arity $k$, and if $t_1, \ldots, t_k$ are terms, then $P(t_1, \ldots, t_k)$ is a formula.
2. For every formula $F$, $\neg F$ is also a formula.
3. For all formulas $F$ and $G$, $(F \wedge G)$ and $(F \vee G)$ are also formulas.
4. If $x$ is a variable and $F$ is a formula, then $\exists x F$ and $\forall x F$ are also formulas.
   The symbol $\exists$ is called the existential quantifier and $\forall$ the universal quantifier.

We call atomic formulas exactly those that are constructed according to 1.

If $F$ is a formula and $F$ occurs as a part of a formula $G$, then $F$ is called a subformula of $G$.

# Free and Bound Variables, Statements

All occurrences of variables in a formula that are not directly behind a quantifier are divided into free and bound occurrences:

An occurrence of the variable $x$ in the formula $F$, which is not directly behind a quantifier, is bound if this occurrence appears in a subformula of $F$ of the form $\exists xG$ or $\forall xG$.
Otherwise, this occurrence of $x$ is free.

A formula without any occurrences of a free variable is called closed or a statement.

The matrix of a formula $F$ is the formula obtained from $F$ by deleting every occurrence of $\exists$ or $\forall$, along with the variables following them.

We denote the matrix of the formula $F$ by $F^*$.

# Example of a Formula in Predicate Logic

Let $F$ be the formula

$$\exists x P(x, f(y)) \ \vee \ \neg \forall y Q(y, g(a, h(z)))$$

The red marked occurrence of $y$ in $F$ is free, while the green marked occurrence of $y$ in $F$ is bound:

$$\exists x P(x, f(y)) \ \vee \ \neg \forall y Q(y, g(a, h(z)))$$

The matrix of $F$ is

$$P(x, f(y)) \ \vee \ \neg Q(y, g(a, h(z))).$$

# Exercise

NF: Non-formula  F: Formula, but not a statement  A: Statement

($x, y$ are variables, $a$ is a constant, $P, Q, R$ are predicate symbols, $f$ is a function symbol)

| | NF | F | A |
|---|---|---|---|
| $\forall x P(a)$ | | | |
| $\forall x \exists y (Q(x,y) \vee R(x,y))$ | | | |
| $\forall x Q(x,x) \rightarrow \exists x Q(x,y)$ | | | |
| $\forall x P(x) \vee \forall x Q(x,x)$ | | | |
| $\forall x (P(y) \wedge \forall y P(x))$ | | | |
| $P(x) \rightarrow \exists x Q(x, P(x))$ | | | |
| $\forall f \exists x P(f(x))$ | | | |

# Exercise

NF: Non-formula  F: Formula, but not a statement  A: Statement

($x, y$ are variables, $a$ is a constant, $P, Q, R$ are predicate symbols, $f$ is a function symbol)

| | NF | F | A |
|---|---|---|---|
| $\forall x P(a)$ | N | N | Y |
| $\forall x \exists y (Q(x,y) \vee R(x,y))$ | | | |
| $\forall x Q(x,x) \rightarrow \exists x Q(x,y)$ | | | |
| $\forall x P(x) \vee \forall x Q(x,x)$ | | | |
| $\forall x (P(y) \wedge \forall y P(x))$ | | | |
| $P(x) \rightarrow \exists x Q(x, P(x))$ | | | |
| $\forall f \exists x P(f(x))$ | | | |

# Exercise

NF: Non-formula F: Formula, but not a statement A: Statement

($x, y$ are variables, $a$ is a constant, $P, Q, R$ are predicate symbols, $f$ is a function symbol)

| | NF | F | A |
|---|---|---|---|
| $\forall x P(a)$ | N | N | Y |
| $\forall x \exists y (Q(x,y) \vee R(x,y))$ | N | N | Y |
| $\forall x Q(x,x) \rightarrow \exists x Q(x,y)$ | | | |
| $\forall x P(x) \vee \forall x Q(x,x)$ | | | |
| $\forall x (P(y) \wedge \forall y P(x))$ | | | |
| $P(x) \rightarrow \exists x Q(x, P(x))$ | | | |
| $\forall f \exists x P(f(x))$ | | | |

# Exercise

NF: Non-formula F: Formula, but not a statement A: Statement

($x, y$ are variables, $a$ is a constant, $P, Q, R$ are predicate symbols, $f$ is a function symbol)

| | NF | F | A |
|---|---|---|---|
| $\forall x P(a)$ | N | N | Y |
| $\forall x \exists y (Q(x,y) \vee R(x,y))$ | N | N | Y |
| $\forall x Q(x,x) \rightarrow \exists x Q(x,y)$ | N | Y | N |
| $\forall x P(x) \vee \forall x Q(x,x)$ | | | |
| $\forall x (P(y) \wedge \forall y P(x))$ | | | |
| $P(x) \rightarrow \exists x Q(x, P(x))$ | | | |
| $\forall f \exists x P(f(x))$ | | | |

# Exercise

NF: Non-formula  F: Formula, but not a statement  A: Statement

($x, y$ are variables, $a$ is a constant, $P, Q, R$ are predicate symbols, $f$ is a function symbol)

| | NF | F | A |
|---|---|---|---|
| $\forall x P(a)$ | N | N | Y |
| $\forall x \exists y (Q(x,y) \vee R(x,y))$ | N | N | Y |
| $\forall x Q(x,x) \rightarrow \exists x Q(x,y)$ | N | Y | N |
| $\forall x P(x) \vee \forall x Q(x,x)$ | N | N | Y |
| $\forall x (P(y) \wedge \forall y P(x))$ | | | |
| $P(x) \rightarrow \exists x Q(x, P(x))$ | | | |
| $\forall f \exists x P(f(x))$ | | | |

# Exercise

NF: Non-formula  F: Formula, but not a statement  A: Statement

($x, y$ are variables, $a$ is a constant, $P, Q, R$ are predicate symbols, $f$ is a function symbol)

| | NF | F | A |
|---|---|---|---|
| $\forall x P(a)$ | N | N | Y |
| $\forall x \exists y (Q(x,y) \vee R(x,y))$ | N | N | Y |
| $\forall x Q(x,x) \rightarrow \exists x Q(x,y)$ | N | Y | N |
| $\forall x P(x) \vee \forall x Q(x,x)$ | N | N | Y |
| $\forall x (P(y) \wedge \forall y P(x))$ | N | Y | N |
| $P(x) \rightarrow \exists x Q(x, P(x))$ | | | |
| $\forall f \exists x P(f(x))$ | | | |

# Exercise

NF: Non-formula F: Formula, but not a statement A: Statement

($x, y$ are variables, $a$ is a constant, $P, Q, R$ are predicate symbols, $f$ is a function symbol)

| | NF | F | A |
|---|---|---|---|
| $\forall x P(a)$ | N | N | Y |
| $\forall x \exists y (Q(x,y) \vee R(x,y))$ | N | N | Y |
| $\forall x Q(x,x) \rightarrow \exists x Q(x,y)$ | N | Y | N |
| $\forall x P(x) \vee \forall x Q(x,x)$ | N | N | Y |
| $\forall x (P(y) \wedge \forall y P(x))$ | N | Y | N |
| $P(x) \rightarrow \exists x Q(x, P(x))$ | Y | N | N |
| $\forall f \exists x P(f(x))$ | | | |

# Exercise

NF: Non-formula F: Formula, but not a statement A: Statement

($x, y$ are variables, $a$ is a constant, $P, Q, R$ are predicate symbols, $f$ is a function symbol)

| | NF | F | A |
|---|---|---|---|
| $\forall x P(a)$ | N | N | Y |
| $\forall x \exists y (Q(x,y) \vee R(x,y))$ | N | N | Y |
| $\forall x Q(x,x) \rightarrow \exists x Q(x,y)$ | N | Y | N |
| $\forall x P(x) \vee \forall x Q(x,x)$ | N | N | Y |
| $\forall x (P(y) \wedge \forall y P(x))$ | N | Y | N |
| $P(x) \rightarrow \exists x Q(x, P(x))$ | Y | N | N |
| $\forall f \exists x P(f(x))$ | Y | N | N |

NF: Non-formula F: Formula, but not a statement A: Statement

| | NF | F | A |
|---|---|---|---|
| $\forall x(\neg\forall y Q(x,y) \wedge R(x,y))$ | | | |
| $\exists z(Q(z,x) \vee R(y,z)) \rightarrow \exists y(R(x,y) \wedge Q(x,z))$ | | | |
| $\exists x(\neg P(x) \vee P(f(a)))$ | | | |
| $P(x) \rightarrow \exists x P(x)$ | | | |
| $\exists x\forall y((P(y) \rightarrow Q(x,y)) \vee \neg P(x))$ | | | |
| $\exists x\forall x Q(x,x)$ | | | |

NF: Non-formula  F: Formula, but not a statement  A: Statement

| | NF | F | A |
|---|---|---|---|
| $\forall x(\neg\forall y Q(x,y) \wedge R(x,y))$ | N | Y | N |
| $\exists z(Q(z,x) \vee R(y,z)) \rightarrow \exists y(R(x,y) \wedge Q(x,z))$ | | | |
| $\exists x(\neg P(x) \vee P(f(a)))$ | | | |
| $P(x) \rightarrow \exists x P(x)$ | | | |
| $\exists x \forall y((P(y) \rightarrow Q(x,y)) \vee \neg P(x))$ | | | |
| $\exists x \forall x Q(x,x)$ | | | |

NF: Non-formula  F: Formula, but not a statement  A: Statement

| | NF | F | A |
|---|---|---|---|
| $\forall x(\neg\forall y Q(x,y) \land R(x,y))$ | N | Y | N |
| $\exists z(Q(z,x) \lor R(y,z)) \to \exists y(R(x,y) \land Q(x,z))$ | N | Y | N |
| $\exists x(\neg P(x) \lor P(f(a)))$ | | | |
| $P(x) \to \exists x P(x)$ | | | |
| $\exists x \forall y((P(y) \to Q(x,y)) \lor \neg P(x))$ | | | |
| $\exists x \forall x Q(x,x)$ | | | |

NF: Non-formula F: Formula, but not a statement A: Statement

| | NF | F | A |
|---|---|---|---|
| $\forall x(\neg\forall y Q(x,y) \wedge R(x,y))$ | N | Y | N |
| $\exists z(Q(z,x) \vee R(y,z)) \rightarrow \exists y(R(x,y) \wedge Q(x,z))$ | N | Y | N |
| $\exists x(\neg P(x) \vee P(f(a)))$ | N | N | Y |
| $P(x) \rightarrow \exists x P(x)$ | | | |
| $\exists x\forall y((P(y) \rightarrow Q(x,y)) \vee \neg P(x))$ | | | |
| $\exists x\forall x Q(x,x)$ | | | |

NF: Non-formula F: Formula, but not a statement A: Statement

| | NF | F | A |
|---|---|---|---|
| $\forall x(\neg\forall y Q(x,y) \land R(x,y))$ | N | Y | N |
| $\exists z(Q(z,x) \lor R(y,z)) \to \exists y(R(x,y) \land Q(x,z))$ | N | Y | N |
| $\exists x(\neg P(x) \lor P(f(a)))$ | N | N | Y |
| $P(x) \to \exists x P(x)$ | N | Y | N |
| $\exists x\forall y((P(y) \to Q(x,y)) \lor \neg P(x))$ | | | |
| $\exists x\forall x Q(x,x)$ | | | |

NF: Non-formula  F: Formula, but not a statement  A: Statement

| | NF | F | A |
|---|---|---|---|
| $\forall x(\neg\forall y Q(x,y) \wedge R(x,y))$ | N | Y | N |
| $\exists z(Q(z,x) \vee R(y,z)) \rightarrow \exists y(R(x,y) \wedge Q(x,z))$ | N | Y | N |
| $\exists x(\neg P(x) \vee P(f(a)))$ | N | N | Y |
| $P(x) \rightarrow \exists x P(x)$ | N | Y | N |
| $\exists x\forall y((P(y) \rightarrow Q(x,y)) \vee \neg P(x))$ | N | N | Y |
| $\exists x\forall x Q(x,x)$ | | | |

NF: Non-formula  F: Formula, but not a statement  A: Statement

| | NF | F | A |
|---|---|---|---|
| $\forall x(\neg\forall y Q(x,y) \wedge R(x,y))$ | N | Y | N |
| $\exists z(Q(z,x) \vee R(y,z)) \rightarrow \exists y(R(x,y) \wedge Q(x,z))$ | N | Y | N |
| $\exists x(\neg P(x) \vee P(f(a)))$ | N | Y | Y |
| $P(x) \rightarrow \exists x P(x)$ | N | Y | N |
| $\exists x\forall y((P(y) \rightarrow Q(x,y)) \vee \neg P(x))$ | N | N | Y |
| $\exists x\forall x Q(x,x)$ | N | N | Y |

# Insert: Relations and Functions of Arbitrary Arity

Let $A$ be any set.

For $n \geq 0$, let $A^n = \{(a_1, a_2, \ldots, a_n) \mid a_1, a_2, \ldots, a_n \in A\}$ be the set of all *n*-tuples over the set $A$.

Note: $A^0 = \{()\}$ is a 1-element set and $A^1 = \{(a) \mid a \in A\}$ can be identified with $A$.

An *n*-ary relation $R$ over $A$ is a subset of $A^n$: $R \subseteq A^n$.

In particular, there are only two 0-ary relations: $\emptyset$ and $\{()\}$.

If $R$ is a 2-ary relation (also called a binary relation), it is often written as $a\,R\,b$ instead of $(a, b) \in R$ (for $a, b \in A$).

An *n*-ary function $f$ on $A$ is a function $f : A^n \to A$.

Note: A 0-ary function $f$ can be identified with an element of $A$, namely with the element $f(())$. We also write
for this $f()$ or simply $f$.

# Semantics of Predicate Logic: Structures

A structure is a pair $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$ with:

$U_\mathcal{A}$ is an arbitrary but non-empty set, called the domain of $\mathcal{A}$ (or the base set, the individuals set, the universe).

Furthermore, $I_\mathcal{A}$ is a partially defined mapping that

- assigns to each $k$-ary predicate symbol $P$ from the domain of $I_\mathcal{A}$ a $k$-ary relation $I_\mathcal{A}(P) \subseteq U_\mathcal{A}^k$,
- assigns to each $k$-ary function symbol $f$ from the domain of $I_\mathcal{A}$ a $k$-ary function $I_\mathcal{A}(f) : U_\mathcal{A}^k \to U_\mathcal{A}$, and
- assigns to each variable $x$ from the domain of $I_\mathcal{A}$ an element $I_\mathcal{A}(x) \in U_\mathcal{A}$.

Let $F$ be a formula and $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$ be a structure.

$\mathcal{A}$ is called suitable for $F$ if $I_\mathcal{A}$ is defined for all predicate symbols, function symbols, and free variables occurring in $F$.

In other words, the domain $\mathrm{Def}(I_\mathcal{A})$ of $I_\mathcal{A}$ is a subset of

$$\{P_i^k \mid i, k \geq 0\} \cup \{f_i^k \mid i, k \geq 0\} \cup \{x_i \mid i \geq 0\},$$

and the codomain of $I_\mathcal{A}$ is the set of all relations, functions, and elements of $U_\mathcal{A}$.

We abbreviate $I_\mathcal{A}(P)$ as $P^\mathcal{A}$, $I_\mathcal{A}(f)$ as $f^\mathcal{A}$, and $I_\mathcal{A}(x)$ as $x^\mathcal{A}$.

# Example of Structure

Let $F$ be the formula $\forall x P(x, f(x)) \ \wedge \ Q(g(a, z))$.

A suitable structure for $F$ is, for example, $\mathcal{A} = (\mathbb{N}, I_{\mathcal{A}})$, where

$$\begin{aligned} P^{\mathcal{A}} &= \{(n, m) \mid n, m \in \mathbb{N}, n < m\} \\ Q^{\mathcal{A}} &= \{n \mid n \text{ is prime}\} \\ f^{\mathcal{A}}(n) &= n + 1 \text{ for all } n \in \mathbb{N} \\ g^{\mathcal{A}}(n, m) &= n + m \text{ for all } n, m \in \mathbb{N} \\ a^{\mathcal{A}} &= 2 \\ z^{\mathcal{A}} &= 3 \end{aligned}$$

In an intuitive sense, the formula $F$ holds in the structure $\mathcal{A}$:
$\forall x \in \mathbb{N}\,(x < x + 1) \ \wedge \ (2 + 3 \text{ is prime})$ is a true statement.

# Evaluation in a Structure

Let $F$ be a formula and $\mathcal{A}$ a suitable structure for $F$.

For every term $t$ that can be formed from the components of $F$ (i.e., from the free variables and function symbols), we inductively define the value $\mathcal{A}(t) \in U_{\mathcal{A}}$ of $t$ in the structure $\mathcal{A}$:

1. If $t$ is a variable (i.e., $t = x$), then $\mathcal{A}(t) = x^{\mathcal{A}}$.
2. If $t$ has the form $t = f(t_1, \ldots, t_k)$ where $t_1, \ldots, t_k$ are terms and $f$ is a $k$-ary function symbol, then $\mathcal{A}(t) = f^{\mathcal{A}}(\mathcal{A}(t_1), \ldots, \mathcal{A}(t_k))$.

Case 2 also includes the possibility that $f$ is nullary, so that $t$ has the form $t = a$.

In this case, we have $\mathcal{A}(t) = a^{\mathcal{A}}$.

# Evaluation in a Structure

In a similar way, we inductively define the (truth–) value $\mathcal{A}(F)$ of the formulas $F$ under the structure $\mathcal{A}$:

- If $F$ has the form $F = P(t_1, \ldots, t_k)$ with the terms $t_1, \ldots, t_k$ and $k$-ary predicate symbol $P$, then

$$\mathcal{A}(F) = \begin{cases} 1, & \textit{if } (\mathcal{A}(t_1), \ldots, \mathcal{A}(t_k)) \in P^{\mathcal{A}} \\ 0, & \textit{otherwise} \end{cases}$$

- If $F$ has the form $F = \neg G$, then

$$\mathcal{A}(F) = \begin{cases} 1, & \textit{if } \mathcal{A}(G) = 0 \\ 0, & \textit{otherwise} \end{cases}$$

# Evaluation in a Structure

- If $F$ has the form $F = (G \wedge H)$, then

$$\mathcal{A}(F) = \begin{cases} 1, & \textit{if } \mathcal{A}(G) = 1 \textit{ and } \mathcal{A}(H) = 1 \\ 0, & \textit{otherwise} \end{cases}$$

- If $F$ has the form $F = (G \vee H)$, then

$$\mathcal{A}(F) = \begin{cases} 1, & \textit{if } \mathcal{A}(G) = 1 \textit{ or } \mathcal{A}(H) = 1 \\ 0, & \textit{otherwise} \end{cases}$$

# Evaluation in a Structure

- If $F$ has the form $F = \forall x G$, then

$$\mathcal{A}(F) = \begin{cases} 1, & \text{if for all } d \in U_{\mathcal{A}}\text{: } \mathcal{A}_{[x/d]}(G) = 1 \\ 0, & \text{otherwise} \end{cases}$$

- If $F$ has the form $F = \exists x G$, then

$$\mathcal{A}(F) = \begin{cases} 1, & \text{if there exists a } d \in U_{\mathcal{A}} \text{ such that: } \mathcal{A}_{[x/d]}(G) = 1 \\ 0, & \text{otherwise} \end{cases}$$

Here, $\mathcal{A}_{[x/d]}$ for a variable $x$ and $d \in U_{\mathcal{A}}$ is the structure that is identical to $\mathcal{A}$, except that $x$ is interpreted as $d$:

$$x^{\mathcal{A}_{[x/d]}} = d$$

# Evaluation in a Structure

More precisely, the structure $\mathcal{A}_{[x/d]}$ is defined as follows.

- $U_{\mathcal{A}} = U_{\mathcal{A}_{[x/d]}}$
- $\mathrm{Def}(I_{\mathcal{A}_{[x/d]}}) = \mathrm{Def}(I_{\mathcal{A}}) \cup \{x\}$
- $x^{\mathcal{A}_{[x/d]}} = d$ (regardless of whether $x^{\mathcal{A}}$ is defined)
- $y^{\mathcal{A}_{[x/d]}} = y^{\mathcal{A}}$ for all variables $y \in \mathrm{Def}(I_{\mathcal{A}}) \setminus \{x\}$
- $P^{\mathcal{A}_{[x/d]}} = P^{\mathcal{A}}$ for all predicate symbols $P \in \mathrm{Def}(I_{\mathcal{A}})$
- $f^{\mathcal{A}_{[x/d]}} = f^{\mathcal{A}}$ for all function symbols $f \in \mathrm{Def}(I_{\mathcal{A}})$

Note: $x^{\mathcal{A}}$ may be defined, but this has no influence on the truth values $\mathcal{A}(\forall x G)$ and $\mathcal{A}(\exists x G)$. When evaluating $\forall x G$ and $\exists x G$ in the structure $\mathcal{A}$, $x^{\mathcal{A}}$ is "overwritten".

# Convention on $=$

We adopt the following convention going forward:

Whenever we use the predicate symbol $=$ in formulas, $=$ is to be considered a binary predicate, and for every structure $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$, the following holds:

For all $a, b \in U_\mathcal{A}$ : $a =^\mathcal{A} b$ if and only if $a$ and $b$ are the same.

# Model, Validity, Satisfiability

If for a formula $F$ and a structure $\mathcal{A}$ corresponding to $F$, we have $\mathcal{A}(F) = 1$, we again write $\mathcal{A} \models F$.

Terminology: $F$ is said to hold in $\mathcal{A}$, or $\mathcal{A}$ is a model for $F$.

$\mathcal{A}$ is a model for a set $M$ of formulas if $\mathcal{A}$ is a model for each formula $F \in M$.

If every structure corresponding to $F$ is a model for $F$, we write $\models F$, otherwise $\not\models F$.

Terminology: $F$ is (universally) valid.

If there is at least one model for the formula $F$, then $F$ is called satisfiable, otherwise it is unsatisfiable.

# Examples

Models for the formula $\forall x \exists y P(x, y)$ would be $\mathcal{A} = (\mathbb{N}, I_{\mathcal{A}})$ as well as $\mathcal{B} = (\mathbb{N}, I_{\mathcal{B}})$ with

$$\begin{aligned} P^{\mathcal{A}} &= \{(n, m) \in \mathbb{N} \times \mathbb{N} \mid n < m\} \text{ and} \\ P^{\mathcal{B}} &= \{(n, m) \in \mathbb{N} \times \mathbb{N} \mid n \leq m\}. \end{aligned}$$

Another (finite) model would be $\mathcal{C} = (\{0\}, I_{\mathcal{C}})$ with $P^{\mathcal{C}} = \{(0, 0)\}$.

Now consider the formula $\exists x \forall y P(x, y)$.

$\mathcal{B}$ and $\mathcal{C}$ are also models of $\exists x \forall y P(x, y)$.

$\mathcal{A}$ is not a model of $\exists x \forall y P(x, y)$.

# Exercise

G: valid  E: satisfiable  U: unsatisfiable

| | G | E | U |
|---|---|---|---|
| $\forall x P(a)$ | | | |
| $\exists x(\neg P(x) \vee P(a))$ | | | |
| $P(a) \rightarrow \exists x P(x)$ | | | |
| $P(x) \rightarrow \exists x P(x)$ | | | |
| $\forall x P(x) \rightarrow \exists x P(x)$ | | | |
| $\forall x P(x) \wedge \neg \forall y P(y)$ | | | |

# Exercise

G: valid  E: satisfiable  U: unsatisfiable

| | G | E | U |
|---|---|---|---|
| $\forall x P(a)$ | N | Y | N |
| $\exists x(\neg P(x) \vee P(a))$ | | | |
| $P(a) \rightarrow \exists x P(x)$ | | | |
| $P(x) \rightarrow \exists x P(x)$ | | | |
| $\forall x P(x) \rightarrow \exists x P(x)$ | | | |
| $\forall x P(x) \wedge \neg \forall y P(y)$ | | | |

# Exercise

G: valid  E: satisfiable  U: unsatisfiable

| | G | E | U |
|---|---|---|---|
| $\forall x P(a)$ | N | Y | N |
| $\exists x(\neg P(x) \vee P(a))$ | Y | Y | N |
| $P(a) \rightarrow \exists x P(x)$ | | | |
| $P(x) \rightarrow \exists x P(x)$ | | | |
| $\forall x P(x) \rightarrow \exists x P(x)$ | | | |
| $\forall x P(x) \wedge \neg \forall y P(y)$ | | | |

# Exercise

G: valid  E: satisfiable  U: unsatisfiable

| | G | E | U |
|---|---|---|---|
| $\forall x P(a)$ | N | Y | N |
| $\exists x(\neg P(x) \vee P(a))$ | Y | Y | N |
| $P(a) \rightarrow \exists x P(x)$ | Y | Y | N |
| $P(x) \rightarrow \exists x P(x)$ | | | |
| $\forall x P(x) \rightarrow \exists x P(x)$ | | | |
| $\forall x P(x) \wedge \neg \forall y P(y)$ | | | |

# Exercise

G: valid  E: satisfiable  U: unsatisfiable

| | G | E | U |
|---|---|---|---|
| $\forall x P(a)$ | N | Y | N |
| $\exists x(\neg P(x) \vee P(a))$ | Y | Y | N |
| $P(a) \rightarrow \exists x P(x)$ | Y | Y | N |
| $P(x) \rightarrow \exists x P(x)$ | Y | Y | N |
| $\forall x P(x) \rightarrow \exists x P(x)$ | | | |
| $\forall x P(x) \wedge \neg \forall y P(y)$ | | | |

# Exercise

G: valid  E: satisfiable  U: unsatisfiable

| | G | E | U |
|---|---|---|---|
| $\forall x P(a)$ | N | Y | N |
| $\exists x(\neg P(x) \vee P(a))$ | Y | Y | N |
| $P(a) \rightarrow \exists x P(x)$ | Y | Y | N |
| $P(x) \rightarrow \exists x P(x)$ | Y | Y | N |
| $\forall x P(x) \rightarrow \exists x P(x)$ | Y | Y | N |
| $\forall x P(x) \wedge \neg \forall y P(y)$ | | | |

# Exercise

G: valid  E: satisfiable  U: unsatisfiable

| | G | E | U |
|---|---|---|---|
| $\forall x P(a)$ | N | Y | N |
| $\exists x(\neg P(x) \vee P(a))$ | Y | Y | N |
| $P(a) \rightarrow \exists x P(x)$ | Y | Y | N |
| $P(x) \rightarrow \exists x P(x)$ | Y | Y | N |
| $\forall x P(x) \rightarrow \exists x P(x)$ | Y | Y | N |
| $\forall x P(x) \wedge \neg \forall y P(y)$ | N | N | Y |

G: valid  E: satisfiable  U: unsatisfiable

| | G | E | U |
|---|---|---|---|
| $\forall x(P(x,x) \rightarrow \exists x \forall y P(x,y))$ | | | |
| $\forall x \forall y(x = y \rightarrow f(x) = f(y))$ | | | |
| $\forall x \forall y(f(x) = f(y) \rightarrow x = y)$ | | | |
| $\exists x \exists y \exists z(f(x) = y \wedge f(x) = z \wedge y \neq z)$ | | | |

G: valid  E: satisfiable  U: unsatisfiable

| | G | E | U |
|---|---|---|---|
| $\forall x(P(x,x) \rightarrow \exists x \forall y P(x,y))$ | N | Y | N |
| $\forall x \forall y(x = y \rightarrow f(x) = f(y))$ | | | |
| $\forall x \forall y(f(x) = f(y) \rightarrow x = y)$ | | | |
| $\exists x \exists y \exists z(f(x) = y \wedge f(x) = z \wedge y \neq z)$ | | | |

G: valid  E: satisfiable  U: unsatisfiable

| | G | E | U |
|---|---|---|---|
| $\forall x(P(x,x) \rightarrow \exists x \forall y P(x,y))$ | N | Y | N |
| $\forall x \forall y(x = y \rightarrow f(x) = f(y))$ | Y | Y | N |
| $\forall x \forall y(f(x) = f(y) \rightarrow x = y)$ | | | |
| $\exists x \exists y \exists z(f(x) = y \wedge f(x) = z \wedge y \neq z)$ | | | |

G: valid  E: satisfiable  U: unsatisfiable

| | G | E | U |
|---|---|---|---|
| $\forall x(P(x,x) \rightarrow \exists x \forall y P(x,y))$ | N | Y | N |
| $\forall x \forall y(x = y \rightarrow f(x) = f(y))$ | Y | Y | N |
| $\forall x \forall y(f(x) = f(y) \rightarrow x = y)$ | N | Y | N |
| $\exists x \exists y \exists z(f(x) = y \wedge f(x) = z \wedge y \neq z)$ | | | |

G: valid  E: satisfiable  U: unsatisfiable

| | G | E | U |
|---|---|---|---|
| $\forall x(P(x,x) \rightarrow \exists x \forall y P(x,y))$ | N | Y | N |
| $\forall x \forall y(x = y \rightarrow f(x) = f(y))$ | Y | Y | N |
| $\forall x \forall y(f(x) = f(y) \rightarrow x = y)$ | N | Y | N |
| $\exists x \exists y \exists z(f(x) = y \wedge f(x) = z \wedge y \neq z)$ | N | N | Y |

# Implication and Equivalence

A formula $G$ is called a consequence of the formulas $F_1, \ldots, F_k$, if for every structure $\mathcal{A}$ that is suitable for both $F_1, \ldots, F_k$ and $G$, the following holds:

*If $\mathcal{A}$ is a model of $\{F_1, \ldots, F_k\}$, then $\mathcal{A}$ is also a model of $G$.*

We write $F_1, \ldots, F_k \models G$, if $G$ is a consequence of $F_1, \ldots, F_k$.

Two formulas $F$ and $G$ are called (semantically) equivalent, if for all structures $\mathcal{A}$, which are suitable for both $F$ and $G$, it holds that $\mathcal{A}(F) = \mathcal{A}(G)$.

We write $F \equiv G$ for this.

# Exercise

1. $\forall x P(x) \vee \forall x Q(x, x)$
2. $\forall x (P(x) \vee Q(x, x))$
3. $\forall x (\forall z P(z) \vee \forall y Q(x, y))$

| | Y | N |
|---|---|---|
| 1. $\models$ 2. | | |
| 2. $\models$ 3. | | |
| 3. $\models$ 1. | | |

# Exercise

1. $\forall x P(x) \vee \forall x Q(x, x)$
2. $\forall x (P(x) \vee Q(x, x))$
3. $\forall x (\forall z P(z) \vee \forall y Q(x, y))$

| | Y | N |
|---|---|---|
| 1. $\models$ 2. | ✓ | |
| 2. $\models$ 3. | | |
| 3. $\models$ 1. | | |

# Exercise

1. $\forall x P(x) \vee \forall x Q(x, x)$
2. $\forall x (P(x) \vee Q(x, x))$
3. $\forall x (\forall z P(z) \vee \forall y Q(x, y))$

| | Y | N |
|---|---|---|
| 1. $\models$ 2. | ✓ | |
| 2. $\models$ 3. | | ✓ |
| 3. $\models$ 1. | | |

Beachte: $\forall x (\forall z P(z) \vee \forall y Q(x, y)) \equiv \forall z P(z) \vee \forall x \forall y Q(x, y)$

# Exercise

1. $\forall x P(x) \vee \forall x Q(x, x)$
2. $\forall x (P(x) \vee Q(x, x))$
3. $\forall x (\forall z P(z) \vee \forall y Q(x, y))$

| | Y | N |
|---|---|---|
| 1. $\models$ 2. | ✓ | |
| 2. $\models$ 3. | | ✓ |
| 3. $\models$ 1. | ✓ | |

Beachte: $\forall x (\forall z P(z) \vee \forall y Q(x, y)) \equiv \forall z P(z) \vee \forall x \forall y Q(x, y)$

# Exercise

1. $\exists y \forall x \; P(x, y)$
2. $\forall x \exists y \; P(x, y)$

| | Y | N |
|---|---|---|
| 1. $\models$ 2. | | |
| 2. $\models$ 1. | | |

# Exercise

1. $\exists y \forall x\ P(x, y)$
2. $\forall x \exists y\ P(x, y)$

| | Y | N |
|---|---|---|
| 1. $\models$ 2. | ✓ | |
| 2. $\models$ 1. | | |

# Exercise

1. $\exists y \forall x\ P(x, y)$
2. $\forall x \exists y\ P(x, y)$

| | Y | N |
|---|---|---|
| 1. $\models$ 2. | ✓ | |
| 2. $\models$ 1. | | ✓ |

# Exercise

1. $\exists y \forall x\ P(x, y)$
2. $\forall x \exists y\ P(x, y)$

| | Y | N |
|---|---|---|
| 1. $\models$ 2. | ✓ | |
| 2. $\models$ 1. | | ✓ |

Example for 2. $\not\models$ 1.

Let $\mathcal{A} = (\{0, 1\}, \mathcal{I})$ with $P^{\mathcal{I}} = \{(0, 0), (1, 1)\}$.

Dann gilt $\mathcal{A} \models \forall x \exists y\ P(x, y)$ und $\mathcal{A} \not\models \exists y \forall x\ P(x, y)$.

# Exercise

| | Y | N |
|---|---|---|
| $\forall x \forall y F \equiv \forall y \forall x F$ | | |
| $\forall x \exists y F \equiv \exists x \forall y F$ | | |
| $\exists x \exists y F \equiv \exists y \exists x F$ | | |
| $\forall x F \vee \forall x G \equiv \forall x (F \vee G)$ | | |
| $\forall x F \wedge \forall x G \equiv \forall x (F \wedge G)$ | | |
| $\exists x F \vee \exists x G \equiv \exists x (F \vee G)$ | | |
| $\exists x F \wedge \exists x G \equiv \exists x (F \wedge G)$ | | |

# Exercise

| | Y | N |
|---|---|---|
| $\forall x \forall y F \equiv \forall y \forall x F$ | ✓ | |
| $\forall x \exists y F \equiv \exists x \forall y F$ | | |
| $\exists x \exists y F \equiv \exists y \exists x F$ | | |
| $\forall x F \vee \forall x G \equiv \forall x (F \vee G)$ | | |
| $\forall x F \wedge \forall x G \equiv \forall x (F \wedge G)$ | | |
| $\exists x F \vee \exists x G \equiv \exists x (F \vee G)$ | | |
| $\exists x F \wedge \exists x G \equiv \exists x (F \wedge G)$ | | |

# Exercise

| | Y | N |
|---|---|---|
| $\forall x \forall y F \equiv \forall y \forall x F$ | ✓ | |
| $\forall x \exists y F \equiv \exists x \forall y F$ | | ✓ |
| $\exists x \exists y F \equiv \exists y \exists x F$ | | |
| $\forall x F \vee \forall x G \equiv \forall x (F \vee G)$ | | |
| $\forall x F \wedge \forall x G \equiv \forall x (F \wedge G)$ | | |
| $\exists x F \vee \exists x G \equiv \exists x (F \vee G)$ | | |
| $\exists x F \wedge \exists x G \equiv \exists x (F \wedge G)$ | | |

# Exercise

| | Y | N |
|---|---|---|
| $\forall x \forall y F \equiv \forall y \forall x F$ | ✓ | |
| $\forall x \exists y F \equiv \exists x \forall y F$ | | ✓ |
| $\exists x \exists y F \equiv \exists y \exists x F$ | ✓ | |
| $\forall x F \vee \forall x G \equiv \forall x (F \vee G)$ | | |
| $\forall x F \wedge \forall x G \equiv \forall x (F \wedge G)$ | | |
| $\exists x F \vee \exists x G \equiv \exists x (F \vee G)$ | | |
| $\exists x F \wedge \exists x G \equiv \exists x (F \wedge G)$ | | |

# Exercise

| | Y | N |
|---|---|---|
| $\forall x \forall y F \equiv \forall y \forall x F$ | ✓ | |
| $\forall x \exists y F \equiv \exists x \forall y F$ | | ✓ |
| $\exists x \exists y F \equiv \exists y \exists x F$ | ✓ | |
| $\forall x F \vee \forall x G \equiv \forall x (F \vee G)$ | | ✓ |
| $\forall x F \wedge \forall x G \equiv \forall x (F \wedge G)$ | | |
| $\exists x F \vee \exists x G \equiv \exists x (F \vee G)$ | | |
| $\exists x F \wedge \exists x G \equiv \exists x (F \wedge G)$ | | |

# Exercise

| | Y | N |
|---|---|---|
| $\forall x \forall y F \equiv \forall y \forall x F$ | ✓ | |
| $\forall x \exists y F \equiv \exists x \forall y F$ | | ✓ |
| $\exists x \exists y F \equiv \exists y \exists x F$ | ✓ | |
| $\forall x F \vee \forall x G \equiv \forall x (F \vee G)$ | | ✓ |
| $\forall x F \wedge \forall x G \equiv \forall x (F \wedge G)$ | ✓ | |
| $\exists x F \vee \exists x G \equiv \exists x (F \vee G)$ | | |
| $\exists x F \wedge \exists x G \equiv \exists x (F \wedge G)$ | | |

# Exercise

| | Y | N |
|---|---|---|
| $\forall x \forall y F \equiv \forall y \forall x F$ | ✓ | |
| $\forall x \exists y F \equiv \exists x \forall y F$ | | ✓ |
| $\exists x \exists y F \equiv \exists y \exists x F$ | ✓ | |
| $\forall x F \vee \forall x G \equiv \forall x (F \vee G)$ | | ✓ |
| $\forall x F \wedge \forall x G \equiv \forall x (F \wedge G)$ | ✓ | |
| $\exists x F \vee \exists x G \equiv \exists x (F \vee G)$ | ✓ | |
| $\exists x F \wedge \exists x G \equiv \exists x (F \wedge G)$ | | |

# Exercise

| | Y | N |
|---|---|---|
| $\forall x \forall y F \equiv \forall y \forall x F$ | ✓ | |
| $\forall x \exists y F \equiv \exists x \forall y F$ | | ✓ |
| $\exists x \exists y F \equiv \exists y \exists x F$ | ✓ | |
| $\forall x F \vee \forall x G \equiv \forall x (F \vee G)$ | | ✓ |
| $\forall x F \wedge \forall x G \equiv \forall x (F \wedge G)$ | ✓ | |
| $\exists x F \vee \exists x G \equiv \exists x (F \vee G)$ | ✓ | |
| $\exists x F \wedge \exists x G \equiv \exists x (F \wedge G)$ | | ✓ |

# Equivalences

## Theorem (Equivalences of Predicate Logic)

Let $F$ and $G$ be arbitrary formulas:

1. $\neg\forall xF \equiv \exists x\neg F$
   $\neg\exists xF \equiv \forall x\neg F$
2. If $x$ does not occur freely in $G$, then:
   $(\forall xF \wedge G) \equiv \forall x(F \wedge G)$
   $(\forall xF \vee G) \equiv \forall x(F \vee G)$
   $(\exists xF \wedge G) \equiv \exists x(F \wedge G)$
   $(\exists xF \vee G) \equiv \exists x(F \vee G)$
3. $(\forall xF \wedge \forall xG) \equiv \forall x(F \wedge G)$
   $(\exists xF \vee \exists xG) \equiv \exists x(F \vee G)$
4. $\forall x\forall yF \equiv \forall y\forall xF$
   $\exists x\exists yF \equiv \exists y\exists xF$

# Equivalences

We prove the equivalence

$$(\forall x F \wedge G) \equiv \forall x (F \wedge G)$$

where $x$ does not occur freely in $G$.

Let $\mathcal{A}$ be any structure suitable for $(\forall x F \wedge G)$ and $\forall x (F \wedge G)$. Then we have:

$\mathcal{A}(\forall x F \wedge G) = 1$

iff $\mathcal{A}(\forall x F) = 1$ and $\mathcal{A}(G) = 1$

iff for all $d \in U_{\mathcal{A}}$ it holds $\mathcal{A}_{[x/d]}(F) = 1$ and $\mathcal{A}(G) = 1$

iff for all $d \in U_{\mathcal{A}}$ it holds $\mathcal{A}_{[x/d]}(F) = 1$ and $\mathcal{A}_{[x/d]}(G) = 1$

(note: since $x$ does not occur freely in $G$, it holds that $\mathcal{A}(G) = \mathcal{A}_{[x/d]}(G)$)

iff for all $d \in U_{\mathcal{A}}$ it holds $\mathcal{A}_{[x/d]}(F \wedge G) = 1$

iff $\mathcal{A}(\forall x (F \wedge G)) = 1$

# Renaming Bound Variables

For a formula $G$, a variable $x$, and a term $t$, let $G[x/t]$ be the formula that is obtained from $G$ by replacing every free occurrence of $x$ in $G$ with the term $t$.

**Example:**

$$\Big(\exists x P\Big(x, f(y)\Big) \ \vee \ \neg\forall y Q\Big(y, g(a, h(z))\Big)\Big)[y/f(u)] =$$
$$\Big(\exists x P\Big(x, f(f(u))\Big) \ \vee \ \neg\forall y Q\Big(y, g(a, h(z))\Big)\Big)$$

**Exercise:** Define $G[x/t]$ formally by induction on the structure of $G$.

# Renaming Bound Variables

**Lemma (Renaming of Variables)**

Let $F = QxG$ be a formula with $Q \in \{\forall, \exists\}$. Let $y$ be a variable that does not occur in $G$. Then it holds that $F \equiv QyG[x/y]$.

The condition that $y$ does not occur in $G$ is important here.

For example, let $G = P(x, y)$.

Then we have $G[x/y] = P(y, y)$ and thus

$$\exists xG = \exists xP(x, y) \not\equiv \exists yP(y, y) = \exists yG[x/y].$$

# Cleaned Formulas

A formula is called cleaned if there are

- no variable that occurs both bound and free in the formula, and
- different variables behind all occurring quantifiers.

**Examples:**

- $P(x) \wedge \forall x Q(x)$ is not cleaned.
- $\exists x P(x) \wedge \forall x Q(x)$ is not cleaned.
- $P(x) \wedge \forall y Q(y)$ is cleaned.
- $\exists x P(x) \wedge \forall y Q(y)$ is cleaned.

Repeated application of the lemma "Renaming of Variables" yields:

**Lemma 47**

*For every formula $F$, there exists an equivalent cleaned formula.*

# Prenex Form

A formula is called prenex or in prenex form if it has the structure

$$Q_1 y_1 Q_2 y_2 \cdots Q_n y_n F$$

where

- $n \geq 0$, $Q_1, \ldots, Q_n \in \{\exists, \forall\}$, $y_1, \ldots, y_n$ are variables, and
- no quantifier occurs in $F$.

A cleaned formula in prenex form is in **BPF**.

**Theorem**

For every formula, there exists an equivalent formula in **BPF**.

**Proof:** Repeated application of the equivalences (1) and (2) from the theorem "Equivalences of Predicate Logic" (Slide 322).

# Formation of Prenex Form

Let $F$ be any formula.

We define a formula $F'$ equivalent to $F$ in **BPF** by induction on the structure of $F$:

- $F$ is atomic:
  Then $F$ is already in **BPF**, and we can set $F' = F$.
- $F = \neg G$:
  By induction, there exists a formula $G'$ equivalent to $G$ in **BPF**

$$G' = Q_1 y_1 Q_2 y_2 \cdots Q_n y_n H,$$

  where $Q_1, \ldots, Q_n \in \{\exists, \forall\}$ and $H$ contains no quantifiers.
  From point (1) in the theorem "Equivalences of Predicate Logic" it follows:

$$\begin{aligned} F = \neg G \equiv \neg G' &= \neg Q_1 y_1 Q_2 y_2 \cdots Q_n y_n H \\ &\equiv \overline{Q}_1 y_1 \overline{Q}_2 y_2 \cdots \overline{Q}_n y_n \neg H \end{aligned}$$

  where $\overline{\exists} = \forall$ and $\overline{\forall} = \exists$. The latter formula is in **BPF**.

# Formation of Prenex Form

- $F = F_1 \wedge F_2$:
  By induction, there exist formulas $F_1'$ and $F_2'$ equivalent to $F_1$ and $F_2$ in **BPF** $F_1' = Q_1 y_1 Q_2 y_2 \cdots Q_m y_m G_1$ and $F_2' = P_1 z_1 P_2 z_2 \cdots P_n z_n G_2$, where $Q_1, \ldots, Q_m, P_1, \ldots, P_n \in \{\exists, \forall\}$ and $G_1, G_2$ contain no quantifiers.
  Due to the renaming lemma, we can assume that $y_1, \ldots, y_m$ do not occur in $F_2'$ and $z_1, \ldots, z_n$ do not occur in $F_1'$.
  From point (2) in the theorem "Equivalences of Predicate Logic" it follows that

$$
\begin{aligned}
F &= F_1 \wedge F_2 \equiv F_1' \wedge F_2' \\
&= (Q_1 y_1 Q_2 y_2 \cdots Q_m y_m G_1) \wedge F_2' \\
&\equiv Q_1 y_1 Q_2 y_2 \cdots Q_m y_m (G_1 \wedge F_2') \\
&= Q_1 y_1 Q_2 y_2 \cdots Q_m y_m (G_1 \wedge P_1 z_1 P_2 z_2 \cdots P_n z_n G_2) \\
&\equiv Q_1 y_1 Q_2 y_2 \cdots Q_m y_m P_1 z_1 P_2 z_2 \cdots P_n z_n (G_1 \wedge G_2)
\end{aligned}
$$

# Formation of Prenex Form

- $F = F_1 \vee F_2$: the same argument as with $\wedge$
- $F = QxG$ with $Q \in \{\exists, \forall\}$:

  By induction, there exists a formula $G'$ equivalent to $G$ in **BPF**

  $$G' = Q_1 y_1 Q_2 y_2 \cdots Q_n y_n H,$$

  where $H$ contains no quantifiers.

  By the renaming lemma, we can assume that $x \notin \{y_1, \ldots, y_n\}$.

  Thus we have

  $$F = QxG \equiv QxQ_1 y_1 Q_2 y_2 \cdots Q_n y_n H$$

  and the latter formula is in **BPF**. □

# Formation of Prenex Form: Example

$$\Big(\forall x \exists y\ P(x, g(y, f(x)))\ \vee\ \neg Q(z)\Big)\ \vee\ \neg \forall x\ R(x, y)$$

$$\equiv \Big(\forall x \exists y\ P(x, g(y, f(x)))\ \vee\ \neg Q(z)\Big)\ \vee\ \exists x\ \neg R(x, y)$$

$$\equiv \forall x \exists y \Big(P(x, g(y, f(x)))\ \vee\ \neg Q(z)\Big)\ \vee\ \exists x\ \neg R(x, y)$$

$$\equiv \forall x \exists y \Big(P(x, g(y, f(x)))\ \vee\ \neg Q(z)\Big)\ \vee\ \exists w\ \neg R(w, y)$$

$$\equiv \forall x \Big(\exists y \Big(P(x, g(y, f(x)))\ \vee\ \neg Q(z)\Big)\ \vee\ \exists w\ \neg R(w, y)\Big)$$

$$\equiv \forall x \Big(\exists v \Big(P(x, g(v, f(x)))\ \vee\ \neg Q(z)\Big)\ \vee\ \exists w\ \neg R(w, y)\Big)$$

$$\equiv \forall x \exists v \Big(\Big(P(x, g(v, f(x)))\ \vee\ \neg Q(z)\Big)\ \vee\ \exists w\ \neg R(w, y)\Big)$$

$$\equiv \forall x \exists v \exists w \Big(P(x, g(v, f(x)))\ \vee\ \neg Q(z)\ \vee\ \neg R(w, y)\Big)$$

# Skolem Form

For every formula $F$ in **BPF**, we define its Skolem form as the result of applying the following algorithm to $F$:

**while** $F$ contains an existential quantifier **do**
**begin**

$F$ has the form $F = \forall y_1 \forall y_2 \cdots \forall y_n \exists z G$ for some formula $G$ in **BPF** and $n \geq 0$ (the universal quantifier block can also be empty);

Let $f$ be a new $n$-ary function symbol not occurring in $F$;

$F := \forall y_1 \forall y_2 \cdots \forall y_n G[z/f(y_1, y_2, \ldots, y_n)]$;
(i.e., the existential quantifier in $F$ is removed and each occurrence of the variable $z$ in $G$ is replaced by $f(y_1, y_2, \ldots, y_n)$)

**end**

# Skolem Form: Example

**Example 1:** We want to form the Skolem form of

$$\forall x \exists v \exists w \Big( P(x, g(v, f(x))) \ \vee \ \neg Q(z) \ \vee \ \neg R(w, y) \Big)$$

After the 1st iteration through the **while** loop:

$$\forall x \exists w \Big( P(x, g(f_1(x), f(x))) \ \vee \ \neg Q(z) \ \vee \ \neg R(w, y) \Big)$$

After the 2nd iteration through the **while** loop:

$$\forall x \Big( P(x, g(f_1(x), f(x))) \ \vee \ \neg Q(z) \ \vee \ \neg R(f_2(x), y) \Big)$$

**Example 2:** The Skolem form of $\exists x P(x)$ is $P(a)$, where $a$ is a constant.

# Skolem Form($F$) is satisfiable iff $F$ is satisfiable.

**Theorem**

For every formula $F$ in **BPF**, it holds that: $F$ is satisfiable if and only if the Skolem form of $F$ is satisfiable.

For the proof, we need the following simple transfer lemma:

**Transfer Lemma**

Let $F$ be a formula, $x$ a variable, and $t$ a term that does not contain any variable bound in $F$. Then for every structure $\mathcal{A}$ suitable for $F$ and $F[x/t]$:

$$\mathcal{A}(F[x/t]) = \mathcal{A}_{[x/\mathcal{A}(t)]}(F)$$

# Proof of the Transfer Lemma

We first show by induction on the structure of terms:

For every term $t'$ it holds: $\mathcal{A}(t'[x/t]) = \mathcal{A}_{[x/\mathcal{A}(t)]}(t')$

- $t' = y$ for a variable $y$.
  - If $y = x$, i.e., $t' = x$:

    Then it holds that $t'[x/t] = t$.

    Thus: $\mathcal{A}_{[x/\mathcal{A}(t)]}(t') = \mathcal{A}_{[x/\mathcal{A}(t)]}(x) = \mathcal{A}(t) = \mathcal{A}(t'[x/t])$
  - If $y \neq x$.

    Then it holds that $t'[x/t] = t' = y$.

    Thus: $\mathcal{A}_{[x/\mathcal{A}(t)]}(t') = \mathcal{A}(t') = \mathcal{A}(t'[x/t])$.

# Proof of the Transfer Lemma

- $t' = f(t_1, \ldots, t_n)$.

  Then it holds that:

$$\begin{aligned}
\mathcal{A}(t'[x/t]) &= \mathcal{A}(f(t_1[x/t], \ldots, t_n[x/t])) \\
&= f^{\mathcal{A}}(\mathcal{A}(t_1[x/t]), \ldots, \mathcal{A}(t_n[x/t])) \\
&= f^{\mathcal{A}_{[x/\mathcal{A}(t)]}}(\mathcal{A}_{[x/\mathcal{A}(t)]}(t_1), \ldots, \mathcal{A}_{[x/\mathcal{A}(t)]}(t_n)) \\
&= \mathcal{A}_{[x/\mathcal{A}(t)]}(f(t_1, \ldots, t_n)) \\
&= \mathcal{A}_{[x/\mathcal{A}(t)]}(t')
\end{aligned}$$

# Proof of the Transfer Lemma

Now we can prove $\mathcal{A}(F[x/t]) = \mathcal{A}_{[x/\mathcal{A}(t)]}(F)$ for a formula $F$ by induction on the structure of $F$:

- $F$ is atomic, i.e., $F = P(t_1, \ldots, t_n)$ for a predicate symbol $P$ and terms $t_1, \ldots, t_n$.

  Then it holds:

$$\begin{aligned}
\mathcal{A}(F[x/t]) = 1 \quad &\text{iff} \quad \mathcal{A}(P(t_1[x/t], \ldots, t_n[x/t])) = 1 \\
&\text{iff} \quad (\mathcal{A}(t_1[x/t]), \ldots, \mathcal{A}(t_n[x/t])) \in P^{\mathcal{A}} \\
&\text{iff} \quad (\mathcal{A}_{[x/\mathcal{A}(t)]}(t_1), \ldots, \mathcal{A}_{[x/\mathcal{A}(t)]}(t_n)) \in P^{\mathcal{A}_{[x/\mathcal{A}(t)]}} \\
&\text{iff} \quad \mathcal{A}_{[x/\mathcal{A}(t)]}(P(t_1, \ldots, t_n)) = 1 \\
&\text{iff} \quad \mathcal{A}_{[x/\mathcal{A}(t)]}(F) = 1
\end{aligned}$$

# Proof of the Transfer Lemma

- $F = \neg G$.

  Then it holds:

$$\begin{aligned} \mathcal{A}(F[x/t]) = 1 \quad & \text{iff} \quad \mathcal{A}(\neg G[x/t]) = 1 \\ & \text{iff} \quad \mathcal{A}(G[x/t]) = 0 \\ & \text{iff} \quad \mathcal{A}_{[x/\mathcal{A}(t)]}(G) = 0 \\ & \text{iff} \quad \mathcal{A}_{[x/\mathcal{A}(t)]}(\neg G) = 1 \\ & \text{iff} \quad \mathcal{A}_{[x/\mathcal{A}(t)]}(F) = 1 \end{aligned}$$

- $F = F_1 \wedge F_2$ or $F = F_1 \vee F_2$:

  Analogous argument as in the previous case.

# Proof of the Transfer Lemma

- $F = \exists y G$, where $y$ does not occur in $t$ (since $t$ should not contain any bound variable in $F$).

  If $y = x$, then $\mathcal{A}(F[x/t]) = \mathcal{A}(F) = \mathcal{A}_{[x/\mathcal{A}(t)]}(F)$.

  The last equality holds because $x$ does not occur free in $F$.

  Now, let $y \neq x$. Then it holds:

$$
\begin{aligned}
\mathcal{A}(F[x/t]) = 1 \quad & \text{iff} \quad \mathcal{A}(\exists y G[x/t]) = 1 \\
& \text{iff} \quad \exists d \in U_{\mathcal{A}} \text{ such that } \mathcal{A}_{[y/d]}(G[x/t]) = 1 \\
& \text{iff} \quad \exists d \in U_{\mathcal{A}} \text{ such that } \mathcal{A}_{[y/d][x/\mathcal{A}_{[y/d]}(t)]}(G) = 1 \\
& \text{iff} \quad \exists d \in U_{\mathcal{A}_{[x/\mathcal{A}(t)]}} \text{ such that } \mathcal{A}_{[x/\mathcal{A}(t)][y/d]}(G) = 1 \\
& \qquad (\mathcal{A}_{[y/d]}(t) = \mathcal{A}(t), \text{ since } y \text{ does not occur in } t) \\
& \text{iff} \quad \mathcal{A}_{[x/\mathcal{A}(t)]}(\exists y G) = 1 \\
& \text{iff} \quad \mathcal{A}_{[x/\mathcal{A}(t)]}(F) = 1
\end{aligned}
$$

# Proof that Skolemform($F$) is satisfiable iff $F$ is satisfiable

**Proof of the Theorem from Slide 334:**

We show: After each iteration of the **while** loop, the resulting formula is satisfiable iff the formula before the iteration is satisfiable.

Formula before the iteration of the **while** loop:

$$F = \forall y_1 \forall y_2 \cdots \forall y_n \exists z G$$

Formula after the iteration of the **while** loop:

$$F' = \forall y_1 \forall y_2 \cdots \forall y_n G[z/f(y_1, y_2, \ldots, y_n)]$$

Here, $f$ is a function symbol that does not occur in $F$.

To show: $F$ is satisfiable iff $F'$ is satisfiable.

# Proof that Skolemform($F$) is satisfiable iff $F$ is satisfiable

(1) Assume $F' = \forall y_1 \forall y_2 \cdots \forall y_n G[z/f(y_1, y_2, \ldots, y_n)]$ is satisfiable.

Then there exists a structure $\mathcal{A}$ (suitable for $F'$) such that $\mathcal{A}(F') = 1$.

Then $\mathcal{A}$ is also suitable for $F$ and it holds:

For all $d_1, \ldots, d_n \in U_{\mathcal{A}}$, it holds:

$$\mathcal{A}_{[y_1/d_1][y_2/d_2]\cdots[y_n/d_n]}(G[z/f(y_1, y_2, \ldots, y_n)]) = 1$$

By the Transfer Lemma (replace $\mathcal{A} \to \mathcal{A}_{[y_1/d_1][y_2/d_2]\cdots[y_n/d_n]}$, $t \to f(y_1, y_2, \ldots, y_n)$, $F \to G$, $x \to z$), it follows:

For all $d_1, \ldots, d_n \in U_{\mathcal{A}}$, it holds:

$$\mathcal{A}_{[y_1/d_1][y_2/d_2]\cdots[y_n/d_n][z/d]}(G) = 1$$

where $d = \mathcal{A}_{[y_1/d_1][y_2/d_2]\cdots[y_n/d_n]}(f(y_1, y_2, \ldots, y_n)) = f^{\mathcal{A}}(d_1, \ldots, d_n)$.

# Proof that Skolemform($F$) is satisfiable iff $F$ is satisfiable

This implies

$$\text{For all } d_1, \ldots, d_n \in U_{\mathcal{A}}, \text{ there exists a } d \in U_{\mathcal{A}} \text{ such that:}$$
$$\mathcal{A}_{[y_1/d_1][y_2/d_2]\cdots[y_n/d_n][z/d]}(G) = 1$$

i.e., $\mathcal{A}(\forall y_1 \forall y_2 \cdots \forall y_n \exists z G) = 1$.

(2) Assume $F = \forall y_1 \forall y_2 \cdots \forall y_n \exists z G$ is satisfiable.

Then there exists a structure $\mathcal{A}$ such that

$$\text{For all } d_1, \ldots, d_n \in U_{\mathcal{A}}, \text{ there exists a } d \in U_{\mathcal{A}} \text{ such that:}$$
$$\mathcal{A}_{[y_1/d_1][y_2/d_2]\cdots[y_n/d_n][z/d]}(G) = 1$$

Thus, for all $d_1, \ldots, d_n \in U_{\mathcal{A}}$, we can choose a $u(d_1, \ldots, d_n) \in U_{\mathcal{A}}$ such that

$$\mathcal{A}_{[y_1/d_1][y_2/d_2]\cdots[y_n/d_n][z/u(d_1,\ldots,d_n)]}(G) = 1$$

# Proof that Skolemform($F$) is satisfiable iff $F$ is satisfiable

We now define a structure $\mathcal{A}'$ as follows:

- $\mathcal{A}'$ is identical to $\mathcal{A}$ except
- $f^{\mathcal{A}'}(d_1, \ldots, d_n) = u(d_1, \ldots, d_n)$ for all $d_1, \ldots, d_n \in U_{\mathcal{A}}$.

Then it holds that:

For all $d_1, \ldots, d_n \in U_{\mathcal{A}} = U_{\mathcal{A}'}$ it holds:

$$\mathcal{A}'_{[y_1/d_1][y_2/d_2]\cdots[y_n/d_n][z/f^{\mathcal{A}'}(d_1,\ldots,d_n)]}(G) = 1$$

Using the transfer lemma (replace $\mathcal{A} \to \mathcal{A}'_{[y_1/d_1][y_2/d_2]\cdots[y_n/d_n]}$, $t \to f(y_1, y_2, \ldots, y_n)$, $F \to G$, $x \to z$), it follows that:

For all $d_1, \ldots, d_n \in U_{\mathcal{A}'}$ it holds:

$$\mathcal{A}'_{[y_1/d_1][y_2/d_2]\cdots[y_n/d_n]}\left(G\left[z/f(y_1, y_2, \ldots, y_n)\right]\right) = 1$$

and thus $\mathcal{A}'\left(\forall y_1 \forall y_2 \cdots \forall y_n G\left[z/f(y_1, y_2, \ldots, y_n)\right]\right) = 1$. □

# Remarks on Skolem Form

- The Skolem form of a formula $F$ is generally not equivalent to $F$.

  Example: The Skolem form of $\exists x P(x)$ is $P(a)$ for a constant $a$.

  However: $\exists x P(x) \not\equiv P(a)$
- The above proof (Point (1)) even shows that every model of the Skolem form of $F$ is also a model of $F$.
- Furthermore, we obtain a model of the Skolem form of $F$ by extending a model of $F$ with an interpretation of the new function symbols (which are introduced during the formation of the Skolem form).

# Skolem Form of a Set of Formulas

Let $M$ be a (generally infinite) set of predicate logic formulas, all of which are in **BPF** without loss of generality.

We define the Skolem form of $M$ as the set $M'$ of Skolem formulas obtained as follows:

We replace each formula $F$ in $M$ with its Skolem form. We ensure that we introduce disjoint sets of new function symbols for different formulas $F, G \in M$.

We then obtain:

**Theorem 48**

*Let $M$ be a (generally infinite) set of predicate logic formulas in* **BPF**. *Then $M$ is satisfiable if and only if the Skolem form of $M$ is satisfiable.*

# Skolem Form of a Set of Formulas

**Proof:** Let $M' = \{F' \mid F \in M\}$ be the Skolem form of $M$, where $F'$ is a Skolem form of $F$.

(1) Let $\mathcal{A}$ be a model of $M'$, and let $F \in M$ be arbitrary.

Then, $\mathcal{A}$ is a model of $F'$, and thus, it is also a model of $F$.

(2) Let $\mathcal{A}$ be a model of $M$.

For a formula $F \in M$, let $\mathrm{fun}(F)$ be the set of new function symbols introduced in the formation of the Skolem form $F'$ of $F$.

Thus, we have $\mathrm{fun}(F) \cap \mathrm{fun}(G) = \emptyset$ for $F \neq G$ from $M$.

We obtain a model of $F' \in M'$ by expanding $\mathcal{A}$ with an interpretation of the symbols from $\mathrm{fun}(F)$.

These expansions then provide a model of $M'$. □

# Clause Form

A statement (= closed formula) is in clause form if it has the structure

$$\forall y_1 \forall y_2 \cdots \forall y_n \, F$$

where $F$ contains no quantifiers and is in **CNF**, with $y_i \neq y_j$ for $i \neq j$.

A statement in clause form can be represented as a set of clauses.

**Example:** The following statement is in clause form:

$$\forall y_1 \forall y_2 \forall y_3 \forall y_4 ((P(y_1) \vee \neg Q(y_2, y_4)) \wedge (Q(y_1, y_3) \vee \neg P(y_3)) \wedge (Q(y_2, y_2) \vee P(y_4)))$$

# Transformation of Any Formula into a Statement in Clause Form

Given: a predicate logic formula $F$ (possibly containing free variables).

1. Simplify $F$ by systematically renaming the bound variables. This produces a formula $F_1$ that is equivalent to $F$.
2. Let $y_1, y_2, \ldots, y_n$ be the free variables occurring in $F$ or $F_1$. Replace $F_1$ with $F_2 = F_1[y_1/a_1, y_2/a_2, \ldots, y_n/a_n]$, where $a_1, \ldots, a_n$ are distinct constants that do not occur in $F_1$. Then, $F_2$ is a statement and is satisfiable if and only if $F_1$ is satisfiable.
3. Create an equivalent statement $F_3$ in prenex form that is also satisfiability-equivalent to $F_2$ (and thus to $F$).
4. Eliminate the existential quantifiers by converting to the Skolem form of $F_3$. Let this be $F_4$, which is satisfiability-equivalent to $F_3$ and hence also to $F$.
5. Transform the matrix of $F_4$ into **CNF** (and write this formula $F_5$ as a set of clauses).

# Exercise on Normal Forms

Which of these formulas are cleaned, in prenex form, in Skolem form, and in clause form?

| | Cln. | P | S | Cls. |
|---|---|---|---|---|
| $\forall x(Tet(x) \vee Cube(x) \vee Dodec(x))$ | | | | |
| $\exists x \exists y(Cube(y) \vee BackOf(x, y))$ | | | | |
| $\forall x(\neg FrontOf(x, x) \wedge \neg BackOf(x, x))$ | | | | |
| $\neg \exists x Cube(x) \leftrightarrow \forall x \neg Cube(x)$ | | | | |
| $\forall x(Cube(x) \rightarrow Small(x)) \rightarrow \forall y(\neg Cube(y) \rightarrow \neg Small(y))$ | | | | |
| $(Cube(a) \wedge \forall x Small(x)) \rightarrow Small(a)$ | | | | |
| $\exists x(Larger(a, x) \wedge Larger(x, b)) \rightarrow Larger(a, b)$ | | | | |

# Exercise on Normal Forms

Which of these formulas are cleaned, in prenex form, in Skolem form, and in clause form?

| | Cln. | P | S | Cls. |
|---|---|---|---|---|
| $\forall x(Tet(x) \vee Cube(x) \vee Dodec(x))$ | Y | Y | Y | Y |
| $\exists x \exists y(Cube(y) \vee BackOf(x, y))$ | | | | |
| $\forall x(\neg FrontOf(x, x) \wedge \neg BackOf(x, x))$ | | | | |
| $\neg \exists x Cube(x) \leftrightarrow \forall x \neg Cube(x)$ | | | | |
| $\forall x(Cube(x) \rightarrow Small(x)) \rightarrow \forall y(\neg Cube(y) \rightarrow \neg Small(y))$ | | | | |
| $(Cube(a) \wedge \forall x Small(x)) \rightarrow Small(a)$ | | | | |
| $\exists x(Larger(a, x) \wedge Larger(x, b)) \rightarrow Larger(a, b)$ | | | | |

# Exercise on Normal Forms

Which of these formulas are cleaned, in prenex form, in Skolem form, and in clause form?

| | Cln. | P | S | Cls. |
|---|---|---|---|---|
| $\forall x(Tet(x) \vee Cube(x) \vee Dodec(x))$ | Y | Y | Y | Y |
| $\exists x \exists y(Cube(y) \vee BackOf(x, y))$ | Y | Y | N | N |
| $\forall x(\neg FrontOf(x, x) \wedge \neg BackOf(x, x))$ | | | | |
| $\neg \exists x Cube(x) \leftrightarrow \forall x \neg Cube(x)$ | | | | |
| $\forall x(Cube(x) \rightarrow Small(x)) \rightarrow \forall y(\neg Cube(y) \rightarrow \neg Small(y))$ | | | | |
| $(Cube(a) \wedge \forall x Small(x)) \rightarrow Small(a)$ | | | | |
| $\exists x(Larger(a, x) \wedge Larger(x, b)) \rightarrow Larger(a, b)$ | | | | |

# Exercise on Normal Forms

Which of these formulas are cleaned, in prenex form, in Skolem form, and in clause form?

| | Cln. | P | S | Cls. |
|---|---|---|---|---|
| $\forall x(Tet(x) \vee Cube(x) \vee Dodec(x))$ | Y | Y | Y | Y |
| $\exists x \exists y(Cube(y) \vee BackOf(x, y))$ | Y | Y | N | N |
| $\forall x(\neg FrontOf(x, x) \wedge \neg BackOf(x, x))$ | Y | Y | Y | Y |
| $\neg \exists x Cube(x) \leftrightarrow \forall x \neg Cube(x)$ | | | | |
| $\forall x(Cube(x) \rightarrow Small(x)) \rightarrow \forall y(\neg Cube(y) \rightarrow \neg Small(y))$ | | | | |
| $(Cube(a) \wedge \forall x Small(x)) \rightarrow Small(a)$ | | | | |
| $\exists x(Larger(a, x) \wedge Larger(x, b)) \rightarrow Larger(a, b)$ | | | | |

# Exercise on Normal Forms

Which of these formulas are cleaned, in prenex form, in Skolem form, and in clause form?

| | Cln. | P | S | Cls. |
|---|---|---|---|---|
| $\forall x(Tet(x) \vee Cube(x) \vee Dodec(x))$ | Y | Y | Y | Y |
| $\exists x \exists y(Cube(y) \vee BackOf(x, y))$ | Y | Y | N | N |
| $\forall x(\neg FrontOf(x, x) \wedge \neg BackOf(x, x))$ | Y | Y | Y | Y |
| $\neg \exists x Cube(x) \leftrightarrow \forall x \neg Cube(x)$ | N | N | N | N |
| $\forall x(Cube(x) \rightarrow Small(x)) \rightarrow \forall y(\neg Cube(y) \rightarrow \neg Small(y))$ | | | | |
| $(Cube(a) \wedge \forall x Small(x)) \rightarrow Small(a)$ | | | | |
| $\exists x(Larger(a, x) \wedge Larger(x, b)) \rightarrow Larger(a, b)$ | | | | |

# Exercise on Normal Forms

Which of these formulas are cleaned, in prenex form, in Skolem form, and in clause form?

| | Cln. | P | S | Cls. |
|---|---|---|---|---|
| $\forall x(Tet(x) \vee Cube(x) \vee Dodec(x))$ | Y | Y | Y | Y |
| $\exists x \exists y(Cube(y) \vee BackOf(x, y))$ | Y | Y | N | N |
| $\forall x(\neg FrontOf(x, x) \wedge \neg BackOf(x, x))$ | Y | Y | Y | Y |
| $\neg \exists x Cube(x) \leftrightarrow \forall x \neg Cube(x)$ | N | N | N | N |
| $\forall x(Cube(x) \rightarrow Small(x)) \rightarrow \forall y(\neg Cube(y) \rightarrow \neg Small(y))$ | Y | N | N | N |
| $(Cube(a) \wedge \forall x Small(x)) \rightarrow Small(a)$ | | | | |
| $\exists x(Larger(a, x) \wedge Larger(x, b)) \rightarrow Larger(a, b)$ | | | | |

# Exercise on Normal Forms

Which of these formulas are cleaned, in prenex form, in Skolem form, and in clause form?

| | Cln. | P | S | Cls. |
|---|---|---|---|---|
| $\forall x(Tet(x) \vee Cube(x) \vee Dodec(x))$ | Y | Y | Y | Y |
| $\exists x \exists y(Cube(y) \vee BackOf(x, y))$ | Y | Y | N | N |
| $\forall x(\neg FrontOf(x, x) \wedge \neg BackOf(x, x))$ | Y | Y | Y | Y |
| $\neg \exists x Cube(x) \leftrightarrow \forall x \neg Cube(x)$ | N | N | N | N |
| $\forall x(Cube(x) \rightarrow Small(x)) \rightarrow \forall y(\neg Cube(y) \rightarrow \neg Small(y))$ | Y | N | N | N |
| $(Cube(a) \wedge \forall x Small(x)) \rightarrow Small(a)$ | Y | N | N | N |
| $\exists x(Larger(a, x) \wedge Larger(x, b)) \rightarrow Larger(a, b)$ | | | | |

# Exercise on Normal Forms

Which of these formulas are cleaned, in prenex form, in Skolem form, and in clause form?

| | Cln. | P | S | Cls. |
|---|---|---|---|---|
| $\forall x(Tet(x) \vee Cube(x) \vee Dodec(x))$ | Y | Y | Y | Y |
| $\exists x \exists y(Cube(y) \vee BackOf(x, y))$ | Y | Y | N | N |
| $\forall x(\neg FrontOf(x, x) \wedge \neg BackOf(x, x))$ | Y | Y | Y | Y |
| $\neg \exists x Cube(x) \leftrightarrow \forall x \neg Cube(x)$ | N | N | N | N |
| $\forall x(Cube(x) \rightarrow Small(x)) \rightarrow \forall y(\neg Cube(y) \rightarrow \neg Small(y))$ | Y | N | N | N |
| $(Cube(a) \wedge \forall x Small(x)) \rightarrow Small(a)$ | Y | N | N | N |
| $\exists x(Larger(a, x) \wedge Larger(x, b)) \rightarrow Larger(a, b)$ | Y | N | N | N |

# Herbrand Universe

Let $\mathcal{F} \subseteq \{f_i^k \mid i, k \geq 0\}$ be a set of function symbols that contains at least one constant.

The Herbrand Universe $D(\mathcal{F})$ is the set of all variable-free terms that can be formed from the symbols in $\mathcal{F}$.

More formally, $D(\mathcal{F})$ is defined inductively as follows:

For every $n$-ary function symbol $f \in \mathcal{F}$ (with $n = 0$ possible) and terms $t_1, t_2, \ldots, t_n \in D(\mathcal{F})$, the term $f(t_1, t_2, \ldots, t_n)$ also belongs to $D(\mathcal{F})$.

**Example:** $D(\{f, a\}) = \{a, f(a), f(f(a)), f(f(f(a))), \ldots\}$

Note: $D(\mathcal{F})$ is finite if and only if $\mathcal{F}$ contains only constants.

# Herbrand Structure

A Herbrand Structure is a structure $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$ such that there exists a set $\mathcal{F} \subseteq \{f_i^k \mid i, k \geq 0\}$ of function symbols with:

- $\mathcal{F}$ contains a constant.
- $U_\mathcal{A} = D(\mathcal{F})$.
- A function symbol $f$ belongs to $\mathrm{Def}(I_\mathcal{A})$ if and only if $f \in \mathcal{F}$.
- For all $f \in \mathcal{F}$ (with $n$-arity) and $t_1, t_2, \ldots, t_n \in D(\mathcal{F})$, we have

$$f^\mathcal{A}(t_1, t_2, \ldots, t_n) = f(t_1, t_2, \ldots, t_n).$$

For every Herbrand structure $\mathcal{A}$ as above and all $t \in D(\mathcal{F})$, it holds that $\mathcal{A}(t) = t$.

Let $M$ be a set of formulas. A Herbrand Model of $M$ is a model of $M$ that is also a Herbrand structure. The fundamental theorem of predicate logic:

**Theorem 49**

Let $M$ be a set of statements in Skolem form. Then it holds:

$M$ is satisfiable $\iff$ $M$ has a Herbrand model.

# $M$ satisfiable if and only if $M$ has a Herbrand model

**Proof:**

If $M$ has a Herbrand model, then $M$ is naturally satisfiable.

Now assume that $M$ is satisfiable and let $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$ be a model of $M$.

By possibly replacing $M$ with $M \cup \{P(a) \vee \neg P(a)\}$ (where $a$ is a constant and $P$ is a unary predicate symbol), we can assume that there is a constant in $M$.

Let $\mathcal{F}$ (and $\mathcal{P}$) be the set of all function symbols (and predicate symbols) that occur in $M$.

We now define a Herbrand structure $\mathcal{B} = (D(\mathcal{F}), I_\mathcal{B})$:

We still need to define $I_\mathcal{B}$ for the predicate symbols in $\mathcal{P}$.

For all $n$-ary predicates $P \in \mathcal{P}$ and all terms $t_1, \ldots, t_n \in D(\mathcal{F})$, we set:

$$(t_1, \ldots, t_n) \in P^\mathcal{B} \text{ if and only if } (\mathcal{A}(t_1), \ldots, \mathcal{A}(t_n)) \in P^\mathcal{A}.$$

# $M$ satisfiable if and only if $M$ has a Herbrand model

**Claim:** For every statement $F$ in Skolem form, built from the symbols in $\mathcal{F} \cup \mathcal{P}$, it holds that: If $\mathcal{A}(F) = 1$, then also $\mathcal{B}(F) = 1$.

If $F$ contains no quantifiers, we even show $\mathcal{A}(F) = \mathcal{B}(F)$ by induction on the structure of $F$:

- $F$ is atomic, i.e., $F = P(t_1, \ldots, t_n)$ for a predicate symbol $P \in \mathcal{P}$ and variable-free terms $t_1, \ldots, t_n \in D(\mathcal{F})$.
  (Note: $F$ is a statement, meaning $F$ contains no free variables.)

$$\begin{aligned} \mathcal{A}(F) = 1 \quad & \textbf{if and only if} \quad (\mathcal{A}(t_1), \ldots, \mathcal{A}(t_n)) \in P^{\mathcal{A}} \\ & \textbf{if and only if} \quad (t_1, \ldots, t_n) \in P^{\mathcal{B}} \\ & \textbf{if and only if} \quad (\mathcal{B}(t_1), \ldots, \mathcal{B}(t_n)) \in P^{\mathcal{B}} \\ & \textbf{if and only if} \quad \mathcal{B}(F) = 1 \end{aligned}$$

- $F = \neg G$: $\mathcal{A}(F) = 1$ **if and only if** $\mathcal{A}(G) = 0$ **if and only if** $\mathcal{B}(G) = 0$ **if and only if** $\mathcal{B}(F) = 1$
- $F = F_1 \wedge F_2$ or $F = F_1 \vee F_2$: An analogous argument as for $F = \neg G$.

# $M$ satisfiable if and only if $M$ has a Herbrand model

Thus, the case where $F$ contains no quantifiers has been handled.

We handle the general case by induction on the number $n$ of quantifiers in $F$.

**Note:** Since $F$ is in Skolem form, $F$ is of the form $\forall y_1 \cdots \forall y_n H$, where $H$ contains no quantifiers.

**Induction Base:** $n = 0$:

From $\mathcal{A}(F) = 1$, it follows that $\mathcal{B}(F) = 1$, see the previous slide.

**Induction Step:** Let $F = \forall x G$.

From $\mathcal{A}(\forall x G) = 1$, it follows:

$$\text{For all } d \in U_{\mathcal{A}},\ \mathcal{A}_{[x/d]}(G) = 1$$

Due to $\{\mathcal{A}(t) \mid t \in D(\mathcal{F})\} \subseteq U_{\mathcal{A}}$, it follows:

$$\text{For all } t \in D(\mathcal{F}),\ \mathcal{A}_{[x/\mathcal{A}(t)]}(G) = 1$$

# $M$ satisfiable if and only if $M$ has a Herbrand model

With the transfer lemma, it follows that:

$$\text{For all } t \in D(\mathcal{F}),\ \mathcal{A}(G[x/t]) = 1.$$

The statement $G[x/t]$ is again in Skolem form and has only $n-1$ quantifiers.

By the induction hypothesis, it holds that:

$$\text{For all } t \in D(\mathcal{F}),\ \mathcal{B}(G[x/t]) = 1.$$

Again, by the transfer lemma, it follows that:

$$\text{For all } t \in D(\mathcal{F}),\ \mathcal{B}_{[x/\mathcal{B}(t)]}(G) = \mathcal{B}_{[x/t]}(G) = 1$$

and thus $\mathcal{B}(F) = \mathcal{B}(\forall x G) = 1$. □

**Remark:** In the proof just conducted, it is important that all $F \in M$ are in Skolem form, and thus contain no existential quantifiers.

# The Löwenheim-Skolem Theorem

A set $A$ is countable if $A$ is finite or if there exists a bijection $f : \mathbb{N} \to A$.

Note: The universe $D(\mathcal{F})$ of a Herbrand structure is countable.

**Löwenheim-Skolem Theorem**

Every satisfiable set of statements in predicate logic has a model with a countable domain (a countable model).

**Proof:** Let $M$ be a satisfiable set of statements in predicate logic.

Let $M'$ be the Skolem form of $M$.

Theorem 48 (Slide 345) $\implies$ $M'$ is satisfiable.

Theorem 49 (Slide 351) $\implies$ $M'$ has a Herbrand model $\mathcal{B}$.

Remark on Slide 344 $\implies$ $\mathcal{B}$ is also a model of $M$.

Furthermore, $\mathcal{B}$ is countable. □

# Herbrand Expansion

Let $M$ be a set of statements in Skolem form.

Let $\mathcal{F}$ be the set of function symbols occurring in formulas from $M$, and let $a$ be a fixed constant.

We define:

$$D(M) = \begin{cases} D(\mathcal{F}) & \text{if } \mathcal{F} \text{ contains a constant} \\ D(\mathcal{F} \cup \{a\}) & \text{otherwise} \end{cases}$$

In the following, $F^*$ will always denote a quantifier-free formula.

The set of statements

$$E(M) = \{F^*[y_1/t_1][y_2/t_2]\ldots[y_n/t_n] \mid \forall y_1 \forall y_2 \cdots \forall y_n F^* \in M, \\ t_1, t_2, \ldots, t_n \in D(M)\}$$

is the Herbrand Expansion of $M$.

# Herbrand Expansion

The formulas in $E(M)$ are formed by substituting terms from $D(M)$ for the variables in $F$ ($F \in M$) in every possible way.

**Note:** If $M$ is satisfiable, there exists a Herbrand model of $M$ with universe $D(M)$.

**Example:** For $M = \{\forall x \forall y (P(a, x) \wedge \neg R(f(y)))\}$, we have

$$D(M) = \{a, f(a), f(f(a)), \ldots\}.$$

The Herbrand expansion of $M$ is therefore

$$\begin{aligned} E(M) = \{ & P(a, a) \wedge \neg R(f(a)), \\ & P(a, f(a)) \wedge \neg R(f(a)), \\ & P(a, a) \wedge \neg R(f(f(a))), \\ & P(a, f(a)) \wedge \neg R(f(f(a))), \ldots\} \end{aligned}$$

# Herbrand Expansion

We consider the Herbrand expansion of $M$ in the following as a set of propositional formulas.

The atomic formulas here are of the form $P(t_1, \ldots, t_n)$, where $P$ is a predicate symbol occurring in $M$ and $t_1, \ldots, t_n \in D(M)$.

In the example from the previous slide, the Herbrand expansion

$$E(M) = \{P(a, f^n(a)) \wedge \neg R(f^m(a)) \mid n \geq 0, m \geq 1\}$$

(where $f^n(a)$ is an abbreviation for the term $f(f(\cdots f(a) \cdots))$, with $f$ occurring exactly $n$ times) contains exactly the atomic formulas from the set

$$\{P(a, f^n(a)) \mid n \geq 0\} \cup \{R(f^m(a)) \mid m \geq 1\}.$$

# Herbrand Expansion

The assignment $\mathcal{B}$ with

$$\mathcal{B}(P(a, f^n(a))) = 1 \text{ for } n \geq 0 \quad \text{and} \quad \mathcal{B}(R(f^m(a))) = 0 \text{ for } m \geq 1$$

clearly also satisfies $E(M)$ in the propositional sense: $\mathcal{B}(G) = 1$ for all $G \in E(M)$.

The (only) formula $\forall x \forall y (P(a, x) \wedge \neg R(f(y))) \in M$ is also satisfiable. As the following theorem shows, this is no coincidence.

# Gödel-Herbrand-Skolem Theorem

## Gödel-Herbrand-Skolem Theorem

Let $M$ be a set of statements in Skolem form. Then $M$ is satisfiable if and only if the formula set $E(M)$ (in the propositional sense) is satisfiable.

**Proof:** It suffices to show that $M$ has a Herbrand model with universe $D(M)$ if and only if $E(M)$ is satisfiable:

$\mathcal{A}$ *is a Herbrand model for* $M$ *with universe* $D(M)$

**iff** *for all* $\forall y_1 \forall y_2 \cdots \forall y_n F^* \in M$, $t_1, t_2, \ldots, t_n \in D(M)$, *we have*

$$\mathcal{A}_{[y_1/t_1][y_2/t_2]\ldots[y_n/t_n]}(F^*) = 1$$

**iff** *for all* $\forall y_1 \forall y_2 \cdots \forall y_n F^* \in M$, $t_1, t_2, \ldots, t_n \in D(M)$, *we have*

$$\mathcal{A}(F^*[y_1/t_1][y_2/t_2]\ldots[y_n/t_n]) = 1$$

**iff** *for all* $G \in E(M)$, *we have* $\mathcal{A}(G) = 1$

**iff** $\mathcal{A}$ *is a model for* $E(M)$

# Gödel-Herbrand-Skolem Theorem

In this chain of equivalences, $\mathcal{A}$ is a model of $E(M)$ in the sense of predicate logic.

From this, we can easily obtain a model for $E(M)$ in the sense of propositional logic:
For all atomic formulas $P(t_1, \ldots, t_n)$ with $t_1, \ldots, t_n \in D(M)$, we have

$$\mathcal{B}(P(t_1, \ldots, t_n)) = \mathcal{A}(P(t_1, \ldots, t_n)). \tag{16}$$

Thus, it follows that:

$$\mathcal{A} \text{ is a model for } E(M) \textbf{ iff } \mathcal{B} \text{ is a model for } E(M)$$

Conversely, a model $\mathcal{B}$ for $E(M)$ in the sense of propositional logic also yields (via the rule (16)) a (Herbrand) model $\mathcal{A}$ for $E(M)$ in the sense of predicate logic. □

# Theorem of Predicate Logic on Finiteness

**Theorem of Predicate Logic on Finiteness (Gödel 1930)**

A set $M$ of predicate logic formulas is satisfiable if and only if every finite subset of $M$ is satisfiable.

**Proof:** It suffices to show the "if" direction.

First, we replace each free variable in a formula $F \in M$ with a new constant $a$ that does not occur in $M$ (as shown on slide 348).

Important: If $x$ occurs freely in both $F \in M$ and $G \in M$, then $x$ must be replaced by the same constant $a$ in both formulas.

Let $M'$ be the new set of formulas.

Then $M$ is satisfiable if and only if $M'$ is satisfiable.

# Theorem of Predicate Logic on Finiteness

Now form the Skolem form $M''$ of $M'$ (see slide 345).

According to Theorem 48 (slide 345), $M''$ is satisfiable if and only if $M'$ (or $M$) is satisfiable.

We can therefore assume without loss of generality that $M$ is a set of statements in Skolem form.

Let each finite subset $N$ of $M$ be satisfiable.

Gödel-Herbrand-Skolem Theorem $\Rightarrow$ For every finite subset $N \subseteq M$, the Herbrand expansion $E(N)$ is satisfiable in the propositional sense.

In particular, every finite subset of $E(M)$ is satisfiable in the propositional sense (for every finite set $A \subseteq E(M)$, there exists a finite set $B \subseteq M$ with $A \subseteq E(B)$).

The Compactness Theorem of Propositional Logic (slide 291) $\Rightarrow$ $E(M)$ is satisfiable.

Gödel-Herbrand-Skolem Theorem $\Rightarrow$ $M$ is satisfiable. □

# Herbrand's Theorem

**Herbrand's Theorem**

A statement $F$ in Skolem form is unsatisfiable if and only if there is a finite subset of $E(F)$ that is unsatisfiable (in the propositional sense).

**Proof:** Immediate consequence of the Gödel-Herbrand-Skolem Theorem and the Compactness Theorem of Propositional Logic (slide 291). □

# Gilmore's Algorithm

Let $F$ be a predicate logic statement in Skolem form, and let $\{F_1, F_2, F_3, \ldots,\}$ be an enumeration of the Herbrand expansion $E(F)$.

**Gilmore's Algorithm**

Input: $F$
$n := 0$;
**repeat** $n := n + 1$;
**until** $(F_1 \wedge F_2 \wedge \ldots \wedge F_n)$ is unsatisfiable;
Output "unsatisfiable" and stop.

If $F$ is unsatisfiable, then by Herbrand's Theorem, there exists a finite set $M \subseteq E(F)$ that is (in the sense of propositional logic) unsatisfiable.

Then there exists an $n$ such that $M \subseteq \{F_1, \ldots, F_n\}$.

Thus, $\{F_1, \ldots, F_n\}$ is unsatisfiable, and consequently $F_1 \wedge F_2 \wedge \ldots \wedge F_n$ is unsatisfiable.

# Valid Formulas are Semi-Decidable

Conversely, if $F$ is satisfiable, then $E(F)$ is satisfiable, and thus every formula $F_1 \wedge F_2 \wedge \ldots \wedge F_n$ is satisfiable.

We thus obtain:

**Theorem**

Let $F$ be a predicate logic statement in Skolem form. Then the following holds:

- If the input formula $F$ is unsatisfiable, then Gilmore's algorithm terminates after a finite time with the output "unsatisfiable".
- If the input formula $F$ is satisfiable, then Gilmore's algorithm does not terminate, i.e., it runs indefinitely.

The set of unsatisfiable predicate logic statements is thus semi-decidable (recursively enumerable).

# The Satisfiability Problem is Undecidable

**Corollary**

The set of valid predicate logic statements is semi-decidable.

**Proof:** $F$ is valid if and only if $\neg F$ is unsatisfiable.

In the Advanced Logic course (every summer semester), it will be shown that the set of (un)satisfiable predicate logic statements is undecidable.

# Resolution in Predicate Logic

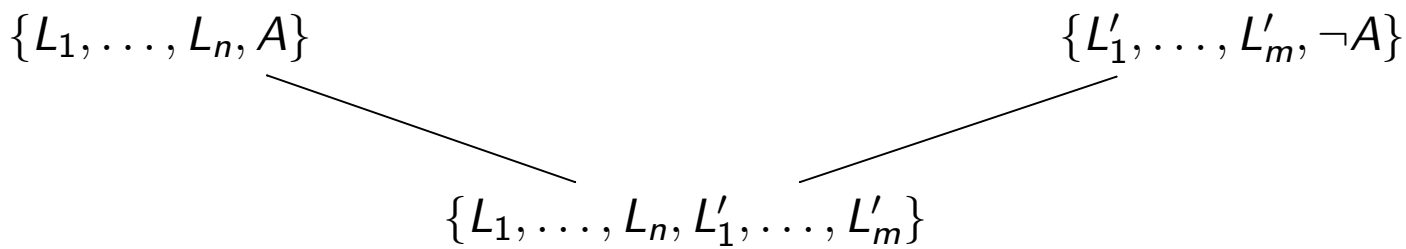
Although Gilmore's algorithm works, it is impractical in practice.

Therefore, our program for the next hours is:

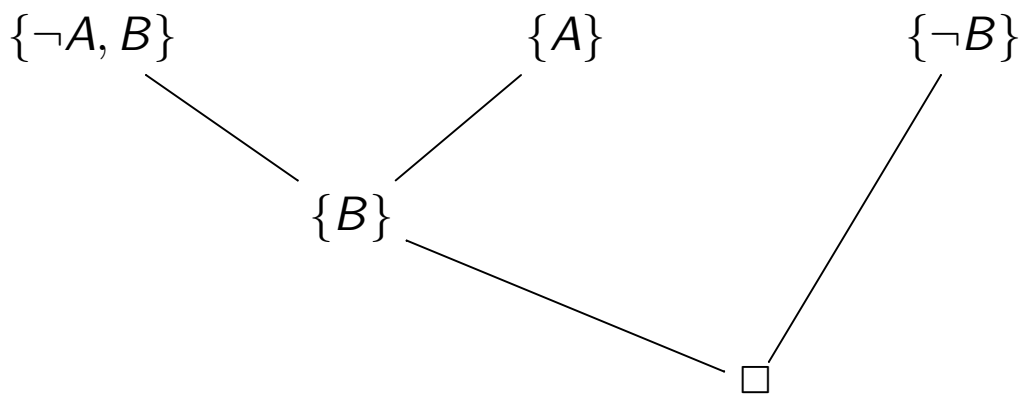
*What does Resolution look like in predicate logic?*

# Review: Resolution in Propositional Logic

**Resolution Step:**




**Mini-Example:**




A set of clauses is unsatisfiable if and only if the empty clause can be derived.

# Adjustment of Gilmore's Algorithm

**Gilmore's Algorithm:**
Let $F$ be a predicate logic statement in Skolem form, and let $\{F_1, F_2, F_3, \ldots\}$ be a enumeration of $E(F)$.

**Input:** $F$
$n := 0$;
**repeat** $n := n + 1$;
**until** $(F_1 \wedge F_2 \wedge \ldots \wedge F_n)$ is unsatisfiable;
(this can be tested using methods from propositional logic, e.g., truth tables,
or other techniques)
Output "unsatisfiable" and stop.

"Methods of propositional logic" $\leadsto$ we use Resolution for the unsatisfiability test.

# Definition of $Res(F)$ (Review)

**Definition:** Let $F$ be a set of clauses. Then $Res(F)$ is defined as

$$Res(F) = F \cup \{R \mid R \text{ is a resolvent of two clauses in } F\}.$$

Additionally, we set:

$$\begin{aligned} Res^0(F) &= F \\ Res^{n+1}(F) &= Res(Res^n(F)) \qquad \text{for } n \geq 0 \end{aligned}$$

Finally, let

$$Res^*(F) = \bigcup_{n \geq 0} Res^n(F).$$

# Basic Resolution Algorithm

Let $F_1, F_2, F_3, \ldots$ be an enumeration of the Herbrand expansion of $F$.

Let $F$ be in clause form, i.e., $F = \forall y_1 \forall y_2 \cdots \forall y_n F^*$, where $F^*$ is in **CNF**.

We consider $F^*$ as a set of clauses, then each $F_i$ is also a set of clauses.

We already know: $F$ is unsatisfiable if and only if there exists an $n$ such that $F_1 \wedge F_2 \wedge \cdots \wedge F_n$ is unsatisfiable.

Note: $F_1 \wedge F_2 \wedge \cdots \wedge F_n$ is again in **CNF** (in terms of propositional logic) and can be identified with the set of clauses $\bigcup_{i=1}^{n} F_i$.

From propositional logic, we know:

$$F_1 \wedge F_2 \wedge \cdots \wedge F_n \text{ is unsatisfiable} \iff Res^*\left(\bigcup_{i=1}^{n} F_i\right) \text{ is unsatisfiable}$$

$$\iff \square \in Res^*\left(\bigcup_{i=1}^{n} F_i\right)$$

# Basic Resolution Algorithm

This leads to the Basic Resolution Algorithm:

**Input:** a formula $F$ in Skolem form with the matrix $F^*$ in **CNF**
$i := 0$;
$M := \emptyset$;
**repeat**
$i := i + 1$; $M := M \cup F_i$; $M := Res^*(M)$
**until** $\square \in M$
Output "unsatisfiable" and stop.

Why the name Basic Resolution?

In contrast to later methods, terms without variables (also known as basic terms) are substituted to obtain the formulas of the Herbrand expansion.

# Basic Resolution Algorithm

**Example:** Consider the following formula in clause form:

$$F = \forall x \forall y((P(x) \vee \neg Q(y, a)) \wedge (Q(f(x), y) \vee \neg P(y)))$$

The first three formulas of the Herbrand expansion of $F$ are:

$$\begin{aligned} F_1 &= (P(a) \vee \neg Q(a, a)) \wedge (Q(f(a), a) \vee \neg P(a)), \\ F_2 &= (P(f(a)) \vee \neg Q(a, a)) \wedge (Q(f(f(a)), a) \vee \neg P(a)), \\ F_3 &= (P(a) \vee \neg Q(f(a), a)) \wedge (Q(f(a), f(a)) \vee \neg P(f(a))) \end{aligned}$$

In set notation:

$$\begin{aligned} F_1 &= \{\{P(a), \neg Q(a, a)\}, \{Q(f(a), a), \neg P(a)\}\}, \\ F_2 &= \{\{P(f(a)), \neg Q(a, a)\}, \{Q(f(f(a)), a), \neg P(a)\}\}, \\ F_3 &= \{\{P(a), \neg Q(f(a), a)\}, \{Q(f(a), f(a)), \neg P(f(a))\}\} \end{aligned}$$

# Basic Resolution Algorithm

Then, after the first three iterations through the **repeat** loop, we obtain the following values for the set variable $M$:

After 1st iteration:

$$\{P(a), \neg Q(a,a)\},\ \{Q(f(a),a), \neg P(a)\},$$
$$\{\neg Q(a,a), Q(f(a),a)\}$$

After 2nd iteration:

$$\{P(a), \neg Q(a,a)\},\ \{Q(f(a),a), \neg P(a)\},\ \{\neg Q(a,a), Q(f(a),a)\},$$
$$\{P(f(a)), \neg Q(a,a)\},\ \{Q(f(f(a)),a), \neg P(a)\}$$
$$\{\neg Q(a,a), Q(f(f(a)),a)\}$$

# Basic Resolution Algorithm

After 3rd iteration:

$$\{P(a), \neg Q(a,a)\},\ \{Q(f(a),a), \neg P(a)\},\ \{\neg Q(a,a), Q(f(a),a)\},$$
$$\{P(f(a)), \neg Q(a,a)\},\ \{Q(f(f(a)),a), \neg P(a)\},$$
$$\{\neg Q(a,a), Q(f(f(a)),a)\},$$
$$\{P(a), \neg Q(f(a),a)\},\ \{Q(f(a),f(a)), \neg P(f(a))\},$$
$$\{Q(f(a),a), \neg Q(f(a),a)\},\ \{Q(f(f(a)),a), \neg Q(f(a),a)\},$$
$$\{P(a), \neg P(a)\},\ \{\neg Q(a,a), P(a)\}, \{Q(f(a),f(a)), \neg Q(a,a)\}$$

# Basic Resolution Theorem

The basic resolution algorithm can also be reformulated into the following basic resolution theorem:

## Basic Resolution Theorem

A formula in Skolem form $F = \forall y_1 \ldots \forall y_k F^*$ with the matrix $F^*$ in **CNF** is unsatisfiable if and only if there exists a sequence of clauses $K_1, \ldots, K_n$ with the following properties:

- $K_n$ is the empty clause
- For all $i \in \{1, \ldots, n\}$ holds:
  - either $K_i$ is a basic instance of a clause $K \in F^*$, i.e. $K_i = K[y_1/t_1] \ldots [y_k/t_k]$ with $t_i \in D(F)$
  - or $K_i$ is a (propositional) resolvent of two clauses $K_a, K_b$ with $a < i$ and $b < i$

Omitting clauses and resolution steps that do not contribute to deriving the empty clause.

# Substitutions

A substitution $\mathrm{sub}$ is a mapping from a finite set of variables to the set of all terms.

Let $\mathrm{Def}(\mathrm{sub})$ denote the domain of the substitution $\mathrm{sub}$.

For a term $t$, we define the term $t\,\mathrm{sub}$ (application of the substitution $\mathrm{sub}$ to the term $t$) inductively as follows:

- $x\,\mathrm{sub} = \mathrm{sub}(x)$, if $x \in \mathrm{Def}(\mathrm{sub})$.
- $y\,\mathrm{sub} = y$, if $y \notin \mathrm{Def}(\mathrm{sub})$.
- $f(t_1, \ldots, t_n)\,\mathrm{sub} = f(t_1\,\mathrm{sub}, \ldots, t_n\,\mathrm{sub})$ for terms $t_1, \ldots, t_n$ and an $n$-ary function symbol $f$
  (this implies $a\,\mathrm{sub} = a$, if $a$ is a constant)

For a literal $F$ (= possibly negated atomic formula), we define $F\,\mathrm{sub}$ as follows, where $P$ is an $n$-ary predicate symbol and $t_1, \ldots, t_n$ are terms:

$$\begin{aligned} P(t_1, \ldots, t_n)\,\mathrm{sub} &= P(t_1\,\mathrm{sub}, \ldots, t_n\,\mathrm{sub}) \\ \neg P(t_1, \ldots, t_n)\,\mathrm{sub} &= \neg P(t_1\mathrm{sub}, \ldots, t_n\mathrm{sub}) \end{aligned}$$

# Substitutions and Replacements

A replacement $[x/t]$ ($x$ is a variable, $t$ is a term) can be identified with the substitution $\mathrm{sub}$ with $\mathrm{Def}(\mathrm{sub}) = \{x\}$ and $\mathrm{sub}(x) = t$.

A substitution $\mathrm{sub}$ with $\mathrm{Def}(\mathrm{sub}) = \{y_1, \ldots, y_n\}$ (each $y_i$ is a variable) can also be written as a sequence of replacements $[y_1/t_1][y_2/t_2]\cdots[y_n/t_n]$.

**Note:** Replacements are performed from left to right!

**Example:** The substitution $\mathrm{sub}$ with $\mathrm{Def}(\mathrm{sub}) = \{x, y, z\}$ and

$$\mathrm{sub}(x) = f(h(w)), \quad \mathrm{sub}(y) = g(a, h(w)), \quad \mathrm{sub}(z) = h(w)$$

is equal to the substitution

$$[x/f(z)]\,[y/g(a,z)]\,[z/h(w)].$$

Composition of substitutions: For $\mathrm{sub}_1\mathrm{sub}_2$ the substitution $\mathrm{sub}_1$ is applied first, followed by $\mathrm{sub}_2$.

# Commuting Substitutions

**Lemma (Rule for Commuting Substitutions)**

If (i) $x \notin \mathrm{Def}(\mathrm{sub})$ and (ii) $x$ does not occur in any of the terms $y\,\mathrm{sub}$ with $y \in \mathrm{Def}(\mathrm{sub})$, then

$$[x/t]\mathrm{sub} = \mathrm{sub}[x/t\,\mathrm{sub}].$$

**Examples:**

- $[x/f(y)]\underbrace{[y/g(z)]}_{\mathrm{sub}} = \underbrace{[y/g(z)]}_{\mathrm{sub}}[x/f(g(z))]$
- but $[x/f(y)]\underbrace{[x/g(z)]}_{\mathrm{sub}} \neq \underbrace{[x/g(z)]}_{\mathrm{sub}}[x/f(y)]$

# Commuting Substitutions

**Proof of the Lemma:**

We show $t'[x/t]\mathrm{sub} = t'\mathrm{sub}[x/t\,\mathrm{sub}]$ for all terms $t'$ by induction on the structure of $t'$.

- $t' = x$:
  Then we have $x[x/t]\mathrm{sub} = t\,\mathrm{sub}$ and likewise $x\,\mathrm{sub}[x/t\,\mathrm{sub}] = x[x/t\,\mathrm{sub}] = t\,\mathrm{sub}$, since $x\,\mathrm{sub} = x$ because $x \notin \mathrm{Def}(\mathrm{sub})$.
- $t' = y$ for a variable $y \neq x$:
  Then we have $y[x/t]\mathrm{sub} = y\,\mathrm{sub}$ and likewise $y\,\mathrm{sub}[x/t\,\mathrm{sub}] = y\,\mathrm{sub}$, since $x$ does not occur in $y\,\mathrm{sub}$.
- $t' = f(t_1, \ldots, t_n)$:
  This case can be immediately handled with the induction hypothesis for $t_1, \ldots, t_n$. □

# Unifier/Most General Unifier

Let $\mathbf{L} = \{L_1, \ldots, L_k\}$ ($k \geq 1$) be a set of literals (i.e., possibly negated atomic predicate logic formulas).

A substitution $\mathrm{sub}$ is called a unifier of $\mathbf{L}$ if

$$L_1\mathrm{sub} = L_2\mathrm{sub} = \cdots = L_k\mathrm{sub}$$

This is equivalent to $|\mathbf{L}\,\mathrm{sub}| = 1$, where $\mathbf{L}\,\mathrm{sub} = \{L_1\mathrm{sub}, \ldots, L_k\mathrm{sub}\}$.

A unifier $\mathrm{sub}$ of $\mathbf{L}$ is called the most general unifier of $\mathbf{L}$ if for any unifier $\mathrm{sub}'$ of $\mathbf{L}$ there exists a substitution $s$ such that $\mathrm{sub}' = \mathrm{sub}\,s$.

# Exercise

| unifiable? | | | Yes | No |
|---|---|---|---|---|
| | $P(f(x))$ | $P(g(y))$ | | |
| | $P(x)$ | $P(f(y))$ | | |
| | $P(x, f(y))$ | $P(f(u), z)$ | | |
| | $P(x, f(y))$ | $P(f(u), f(z))$ | | |
| | $P(x, f(x))$ | $P(f(y), y)$ | | |
| | $P(x, g(x), g^2(x))$ | $P(f(z), w, g(w))$ | | |
| $P(x, f(y))$ | $P(g(y), f(a))$ | $P(g(a), z)$ | | |

# Exercise

| unifiable? | | | Yes | No |
|---|---|---|---|---|
| | $P(f(x))$ | $P(g(y))$ | | ✓ |
| | $P(x)$ | $P(f(y))$ | | |
| | $P(x, f(y))$ | $P(f(u), z)$ | | |
| | $P(x, f(y))$ | $P(f(u), f(z))$ | | |
| | $P(x, f(x))$ | $P(f(y), y)$ | | |
| | $P(x, g(x), g^2(x))$ | $P(f(z), w, g(w))$ | | |
| $P(x, f(y))$ | $P(g(y), f(a))$ | $P(g(a), z)$ | | |

# Exercise

| unifiable? | | | Yes | No |
|---|---|---|---|---|
| | $P(f(x))$ | $P(g(y))$ | | ✓ |
| | $P(x)$ | $P(f(y))$ | ✓ | |
| | $P(x, f(y))$ | $P(f(u), z)$ | | |
| | $P(x, f(y))$ | $P(f(u), f(z))$ | | |
| | $P(x, f(x))$ | $P(f(y), y)$ | | |
| | $P(x, g(x), g^2(x))$ | $P(f(z), w, g(w))$ | | |
| $P(x, f(y))$ | $P(g(y), f(a))$ | $P(g(a), z)$ | | |

# Exercise

| unifiable? | | | Yes | No |
|---|---|---|---|---|
| | $P(f(x))$ | $P(g(y))$ | | ✓ |
| | $P(x)$ | $P(f(y))$ | ✓ | |
| | $P(x, f(y))$ | $P(f(u), z)$ | ✓ | |
| | $P(x, f(y))$ | $P(f(u), f(z))$ | | |
| | $P(x, f(x))$ | $P(f(y), y)$ | | |
| | $P(x, g(x), g^2(x))$ | $P(f(z), w, g(w))$ | | |
| $P(x, f(y))$ | $P(g(y), f(a))$ | $P(g(a), z)$ | | |

# Exercise

| unifiable? | | | Yes | No |
|---|---|---|---|---|
| | $P(f(x))$ | $P(g(y))$ | | ✓ |
| | $P(x)$ | $P(f(y))$ | ✓ | |
| | $P(x, f(y))$ | $P(f(u), z)$ | ✓ | |
| | $P(x, f(y))$ | $P(f(u), f(z))$ | ✓ | |
| | $P(x, f(x))$ | $P(f(y), y)$ | | |
| | $P(x, g(x), g^2(x))$ | $P(f(z), w, g(w))$ | | |
| $P(x, f(y))$ | $P(g(y), f(a))$ | $P(g(a), z)$ | | |

# Exercise

| unifiable? | | | Yes | No |
|---|---|---|---|---|
| | $P(f(x))$ | $P(g(y))$ | | ✓ |
| | $P(x)$ | $P(f(y))$ | ✓ | |
| | $P(x, f(y))$ | $P(f(u), z)$ | ✓ | |
| | $P(x, f(y))$ | $P(f(u), f(z))$ | ✓ | |
| | $P(x, f(x))$ | $P(f(y), y)$ | | ✓ |
| | $P(x, g(x), g^2(x))$ | $P(f(z), w, g(w))$ | | |
| $P(x, f(y))$ | $P(g(y), f(a))$ | $P(g(a), z)$ | | |

# Exercise

| unifiable? | | | Yes | No |
|---|---|---|---|---|
| | $P(f(x))$ | $P(g(y))$ | | ✓ |
| | $P(x)$ | $P(f(y))$ | ✓ | |
| | $P(x, f(y))$ | $P(f(u), z)$ | ✓ | |
| | $P(x, f(y))$ | $P(f(u), f(z))$ | ✓ | |
| | $P(x, f(x))$ | $P(f(y), y)$ | | ✓ |
| | $P(x, g(x), g^2(x))$ | $P(f(z), w, g(w))$ | ✓ | |
| $P(x, f(y))$ | $P(g(y), f(a))$ | $P(g(a), z)$ | | |

# Exercise

| unifiable? | | | Yes | No |
|---|---|---|---|---|
| | $P(f(x))$ | $P(g(y))$ | | ✓ |
| | $P(x)$ | $P(f(y))$ | ✓ | |
| | $P(x, f(y))$ | $P(f(u), z)$ | ✓ | |
| | $P(x, f(y))$ | $P(f(u), f(z))$ | ✓ | |
| | $P(x, f(x))$ | $P(f(y), y)$ | | ✓ |
| | $P(x, g(x), g^2(x))$ | $P(f(z), w, g(w))$ | ✓ | |
| $P(x, f(y))$ | $P(g(y), f(a))$ | $P(g(a), z)$ | ✓ | |

# Exercise

- A unifier for $\{P(x, g(x), g^2(x)), P(f(z), w, g(w))\}$:

$$x \mapsto f(z),\ \ w \mapsto g(f(z))$$

- A unifier for $\{P(x, f(y)), P(g(y), f(a)), P(g(a), z)\}$:

$$x \mapsto g(a),\ \ y \mapsto a,\ \ z \mapsto f(a)$$

# Unification Algorithm

**Input:** a non-empty finite set of literals $\mathbf{L} \neq \emptyset$

$\mathrm{sub} := []$; (empty substitution, i.e., $\mathrm{Def}([]) = \emptyset$)
**while** $|\mathbf{L}\,\mathrm{sub}| > 1$ **do**
    Take two different literals $L_1, L_2 \in \mathbf{L}\,\mathrm{sub}$.
    Search for the first position $p$ where $L_1$ and $L_2$ differ.
    **if** neither of the symbols at position $p$ is a variable **then**
        stop with "not unifiable"
    **else** let $x$ be the variable and $t$ the term in the other literal that starts at position $p$ (possibly also a variable)
        **if** $x$ occurs in $t$ **then**
            stop with "not unifiable"
        **else** $\mathrm{sub} := \mathrm{sub}\,[x/t]$
**endwhile**

**Output:** $\mathrm{sub}$

# Unification Algorithm

**Examples:** (the position $p$ is marked in red)

$$\begin{aligned} L_1 &= P(f(a,x), f(a,y), g(z)) \\ L_2 &= P(f(a,x), g(x), g(z)) \end{aligned}$$

Here the algorithm stops with "not unifiable".

$$\begin{aligned} L_1 &= P(f(a,x), y, g(z)) \\ L_2 &= P(f(a,x), g(x), g(z)) \end{aligned}$$

Here the term $t = g(x)$ and we set $\mathrm{sub} := \mathrm{sub}\,[y/g(x)]$.

$$\begin{aligned} L_1 &= P(f(a,x), x, g(z)) \\ L_2 &= P(f(a,x), g(x), g(z)) \end{aligned}$$

Here the algorithm stops with "not unifiable".

# Correctness of the Unification Algorithm

**Theorem**

It holds:

A. The unification algorithm terminates for every input **L**.

B. If the input **L** is not unifiable, then the unification algorithm terminates with the output "not unifiable".

C. If the input **L** is unifiable, then the unification algorithm always finds a most general unifier of **L**.

(C) particularly implies that every unifiable set of literals has a most general unifier.

# Proof of the Correctness of the Unification Algorithm

**Proof:**

(A) The unification algorithm terminates for every input **L**.

This holds because the number of variables present in $\mathbf{L}\,\mathrm{sub}$ decreases in each step.

Consider a single iteration of the **while** loop.

If the algorithm does not terminate in this iteration, then $\mathrm{sub}$ is set to $\mathrm{sub}\,[x/t]$.

In this case, $x$ is present in $\mathbf{L}\,\mathrm{sub}$ and the term $t$ does not contain $x$.

Thus, $x$ no longer appears in $\mathbf{L}\,\mathrm{sub}\,[x/t]$.

# Proof of the Correctness of the Unification Algorithm

(B) If the input **L** is not unifiable, then the unification algorithm terminates with the output "not unifiable".

Assume that the input **L** is not unifiable.

If the condition $|\mathbf{L}\,\mathrm{sub}| > 1$ in the **while** loop were ever violated, then **L** would indeed be unifiable.

Since, according to (A), the algorithm terminates for input **L**, it must eventually output "not unifiable".

# Proof of the Correctness of the Unification Algorithm

(C) If the input **L** is unifiable, then the unification algorithm always finds a most general unifier of **L**.

Assume **L** is unifiable, and let $\mathrm{sub}_i$ ($i \geq 0$) be the substitution computed after the $i$-th iteration of the **while** loop.

Suppose the algorithm completes $N$ iterations of the **while** loop.

Note: $\mathrm{sub}_0 := []$ and $\mathrm{sub}_N$ is the output of the algorithm (if it exists).

**Claim:**

1. For every unifier $\mathrm{sub}'$ of **L** and for all $0 \leq i \leq N$, there exists a substitution $s_i$ such that $\mathrm{sub}' = \mathrm{sub}_i \; s_i$.
2. In the $i$-th iteration of the **while** loop ($1 \leq i \leq N$), the algorithm either terminates successfully (and outputs the substitution $\mathrm{sub}_N$) or enters both **else** branches.

# Proof of the Correctness of the Unification Algorithm

**Proof of the Claim:**

Let $\mathrm{sub}'$ be a unifier of **L**.

We first consider by induction over $i$ the case where **L** and $\{y\,\mathrm{sub}' \mid y \in \mathrm{Def}(\mathrm{sub}')\}$ have no common variables.

We will choose $s_i$ as a restriction of $\mathrm{sub}'$, i.e., $\mathrm{Def}(s_i) \subseteq \mathrm{Def}(\mathrm{sub}')$ and $s_i(x) = \mathrm{sub}'(x)$ for all $x \in \mathrm{Def}(s_i)$.

Then, **L** and $\{y\,s_i \mid y \in \mathrm{Def}(s_i)\}$ also have no common variables.

**Base Case:** $i = 0$.

It holds that $\mathrm{sub}' = []\,\mathrm{sub}' = \mathrm{sub}_i\,\mathrm{sub}'$. Thus, we can set $s_0 = \mathrm{sub}'$.

# Proof of the Correctness of the Unification Algorithm

**Inductive Step:** Let $i > 0$ and assume that (1) and (2) have already been proven for $i - 1$.

By the induction hypothesis, there exists a restriction $s_{i-1}$ of $\mathrm{sub}'$ such that $\mathrm{sub}' = \mathrm{sub}_{i-1} s_{i-1}$.

If $|\mathbf{L}\,\mathrm{sub}_{i-1}| = 1$, then the algorithm terminates in the $i$-th iteration.

Now, let $|\mathbf{L}\,\mathrm{sub}_{i-1}| > 1$.

Consider the first position $p$ where two literals $L_1$ and $L_2$ from $\mathbf{L}\,\mathrm{sub}_{i-1}$ differ.

Since $|\mathbf{L}\,\mathrm{sub}_{i-1} s_{i-1}| = |\mathbf{L}\,\mathrm{sub}'| = 1$, it holds that $L_1 s_{i-1} = L_2 s_{i-1}$.

Thus, at position $p$ in $L_1$ and $L_2$, there cannot be two different function symbols.

Suppose that in $L_1$ at position $p$ there is a variable $x$, and in $L_2$ at position $p$, a term $t \neq x$ begins.

# Proof of the Correctness of the Unification Algorithm

Then $x\,s_{i-1} = t\,s_{i-1}$ holds.

The variable $x$ cannot appear in $t$:

This is clear if $t$ is a variable (since $t \neq x$) or a constant.

If $t$ has the form $f(t_1, \ldots, t_n)$ with $n \geq 1$, then it must hold that $x\,s_{i-1} = t\,s_{i-1} = f(t_1 s_{i-1}, \ldots, t_n s_{i-1})$.

If $x$ appeared in one of the terms $t_i$, then $f(t_1 s_{i-1}, \ldots, t_n s_{i-1})$ would contain more symbols than $x\,s_{i-1}$.

Thus, the two **else** branches in the body of the **while** loop are entered (this proves (2)).

It holds that $\mathrm{sub}_i = \mathrm{sub}_{i-1}[x/t]$.

Let $s_i$ be the restriction of $s_{i-1}$ to all variables different from $x$.

# Proof of the Correctness of the Unification Algorithm

Then the following holds:

$$
\begin{aligned}
\mathrm{sub}_i\, s_i &= \mathrm{sub}_{i-1}\,[x/t]\, s_i \\
&= \mathrm{sub}_{i-1}\, s_i\,[x/t s_i] && \text{(since } x \notin \mathrm{Def}(s_i) \text{ and } x \text{ does not appear in any} \\
& && \text{terms } y\, s_i \text{ for } y \in \mathrm{Def}(s_i)) \\
&= \mathrm{sub}_{i-1}\, s_i\,[x/t\, s_{i-1}] && \text{(since } x \text{ does not appear in } t) \\
&= \mathrm{sub}_{i-1}\, s_i\,[x/x\, s_{i-1}] && \text{(since } x\, s_{i-1} = t\, s_{i-1}) \\
&= \mathrm{sub}_{i-1}\, s_{i-1} && \text{(definition of } s_i \text{ and } x \text{ does not appear in any of} \\
& && \text{terms } y\, s_i \text{ for } y \in \mathrm{Def}(s_i)) \\
&= \mathrm{sub}' && \text{(inductive hypothesis)}
\end{aligned}
$$

This proves (1).

Thus, the case where **L** and $\{y\,\mathrm{sub}' \mid y \in \mathrm{Def}(\mathrm{sub}')\}$ have no common variables is concluded.

# Proof of the Correctness of the Unification Algorithm

In the general case ($\mathrm{sub}'$ is any unifier of **L**), let $X$ be the set of all variables that appear in $\{y\,\mathrm{sub}' \mid y \in \mathrm{Def}(\mathrm{sub}')\}$.

Let $Y$ be a set of variables with $|X| = |Y|$, such that $Y$ and **L** have no common variables.

Let $u : X \to Y$ be any bijection between $X$ and $Y$.
We call $u$ a variable renaming.

Then $\mathrm{sub}'u$ is also a unifier of **L**, such that **L** and $\{y\,\mathrm{sub}'u \mid y \in \mathrm{Def}(\mathrm{sub}')\}$ have no common variables.

Thus, for all $0 \leq i \leq N$, there exists a substitution $s_i$ such that $\mathrm{sub}'u = \mathrm{sub}_i\ s_i$.

Therefore, $\mathrm{sub}' = \mathrm{sub}_i(s_i u^{-1})$ holds.

# Proof of the Correctness of the Unification Algorithm

From (1) and (2), it now follows:

The algorithm terminates after $N$ iterations through the **while** loop with a unifier $\mathrm{sub} = \mathrm{sub}_N$.

If $\mathrm{sub}'$ is any unifier of **L**, then, due to (1), there exists a substitution $s$ such that $\mathrm{sub}' = \mathrm{sub}\, s$.

Thus, $\mathrm{sub}$ is a most general unifier of **L**. □

# Example of the Unification Algorithm

Consider $\mathbf{L} = \{P(f(z, g(a, y)), h(z)),\ P(f(f(u, v), w), h(f(a, b)))\}$

# Example of the Unification Algorithm

Consider $\mathbf{L} = \{P(f(z, g(a, y)), h(z)),\ P(f(f(u, v), w), h(f(a, b)))\}$

$P(f(z, g(a, y)), h(z))$
$P(f(f(u, v), w), h(f(a, b)))$ $\quad\quad \text{sub} = []$

$P(f(f(u, v), g(a, y)), h(f(u, v)))$
$P(f(f(u, v), w), h(f(a, b)))$ $\quad\quad \text{sub} = [z/f(u, v)]$

$P(f(f(u, v), g(a, y)), h(f(u, v)))$
$P(f(f(u, v), w), h(f(a, b)))$ $\quad\quad \text{sub} = [z/f(u, v)]$

$P(f(f(u, v), g(a, y)), h(f(u, v)))$
$P(f(f(u, v), g(a, y)), h(f(a, b)))$ $\quad\quad \text{sub} = [z/f(u, v)][w/g(a, y)]$

# Example of the Unification Algorithm

$P(f(f(u,v),g(a,y)),h(f(u,v)))$

$P(f(f(u,v),g(a,y)),h(f(a,b)))$ $\quad \text{sub} = [z/f(u,v)][w/g(a,y)]$

$P(f(f(a,v),g(a,y)),h(f(a,v)))$

$P(f(f(a,v),g(a,y)),h(f(a,b)))$ $\quad \text{sub} = [z/f(u,v)][w/g(a,y)][u/a]$

$P(f(f(a,v),g(a,y)),h(f(a,v)))$

$P(f(f(a,v),g(a,y)),h(f(a,b)))$ $\quad \text{sub} = [z/f(u,v)][w/g(a,y)][u/a]$

$P(f(f(a,b),g(a,y)),h(f(a,b)))$

$P(f(f(a,b),g(a,y)),h(f(a,b)))$ $\quad \text{sub} = [z/f(u,v)][w/g(a,y)][u/a][v/b]$

# Complexity of the Unification Algorithm

Although the number of iterations through the while-loop in the unification algorithm is bounded by the input length, the following applies:

Repeated substitution of terms can lead to the creation of very large terms.

In fact, the runtime of our unification algorithm is generally exponential in the input length.

On the other hand, the following result holds:

**Paterson, Wegman 1976**

There exists a unification algorithm whose runtime is bounded linearly in the input length.

# Predicate Logic Resolution

A clause $R$ is called a predicate logic resolvent of two clauses $K_1$ and $K_2$ if the following holds:

- There exist variable renamings $s_1$ and $s_2$ such that $K_1 s_1$ and $K_2 s_2$ have no common variables.
- There exist $m, n \geq 1$ and literals $L_1, \ldots, L_m$ from $K_1 s_1$ and literals $L'_1, \ldots, L'_n$ from $K_2 s_2$ such that

$$\mathbf{L} = \{\overline{L_1}, \ldots, \overline{L_m}, L'_1, \ldots, L'_n\}$$

  is unifiable. Let $\mathrm{sub}$ be the most general unifier of $\mathbf{L}$.
  ($\overline{L}$ denotes the negation of the literal $L$)
- It holds that

$$R = ((K_1 s_1 \setminus \{L_1, \ldots, L_m\}) \cup (K_2 s_2 \setminus \{L'_1, \ldots, L'_n\}))\mathrm{sub}.$$

# Example of a Predicate Logic Resolvent

Let

$$\begin{aligned} K_1 &= \{P(f(x)), \neg Q(z), P(z)\} \\ K_2 &= \{\neg P(x), R(g(x), a)\} \end{aligned}$$

For the variable renamings $s_1 = []$ and $s_2 = [x/u]$, we have:

$$\begin{aligned} K_1 s_1 &= \{P(f(x)), \neg Q(z), P(z)\} \\ K_2 s_2 &= \{\neg P(u), R(g(u), a)\} \end{aligned}$$

These clauses have no common variables.

Let $L_1 = P(f(x)) \in K_1 s_1$, $L_2 = P(z) \in K_1 s_1$, and $L_1' = \neg P(u) \in K_2 s_2$.

The set $\mathbf{L} = \{\overline{L_1}, \overline{L_2}, L_1'\} = \{\neg P(f(x)), \neg P(z), \neg P(u)\}$ is unifiable.

A most general unifier is $\mathrm{sub} = [z/f(x)][u/f(x)]$.

Thus,

$$((K_1 s_1 \setminus \{L_1, L_2\}) \cup (K_2 s_2 \setminus \{L_1'\}))\mathrm{sub} = \{\neg Q(f(x)), R(g(f(x)), a)\}$$

is a resolvent of $K_1$ and $K_2$.

# Correctness and Completeness

Two Questions:

- If one can derive the empty clause $\square$ from a formula $F$ using predicate logic resolution, is $F$ then unsatisfiable? (Correctness)
- Can one always derive the empty clause from an unsatisfiable formula $F$ using predicate logic resolution? (Completeness)

# Exercise

Are these clauses resolvable?

How many possible resolvents are there?

| $K_1$ | $K_2$ | Possibilities |
|---|---|---|
| $\{P(x), Q(x,y)\}$ | $\{\neg P(f(x))\}$ | |
| $\{Q(g(x)), R(f(x))\}$ | $\{\neg Q(f(x))\}$ | |
| $\{P(x), P(f(x))\}$ | $\{\neg P(y), Q(y,z)\}$ | |

# Exercise

Are these clauses resolvable?

How many possible resolvents are there?

| $K_1$ | $K_2$ | Possibilities |
|---|---|---|
| $\{P(x), Q(x,y)\}$ | $\{\neg P(f(x))\}$ | 1 |
| $\{Q(g(x)), R(f(x))\}$ | $\{\neg Q(f(x))\}$ | |
| $\{P(x), P(f(x))\}$ | $\{\neg P(y), Q(y,z)\}$ | |

# Exercise

Are these clauses resolvable?

How many possible resolvents are there?

| $K_1$ | $K_2$ | Possibilities |
|---|---|---|
| $\{P(x), Q(x,y)\}$ | $\{\neg P(f(x))\}$ | 1 |
| $\{Q(g(x)), R(f(x))\}$ | $\{\neg Q(f(x))\}$ | 0 |
| $\{P(x), P(f(x))\}$ | $\{\neg P(y), Q(y,z)\}$ | |

# Exercise

Are these clauses resolvable?

How many possible resolvents are there?

| $K_1$ | $K_2$ | Possibilities |
|---|---|---|
| $\{P(x), Q(x,y)\}$ | $\{\neg P(f(x))\}$ | 1 |
| $\{Q(g(x)), R(f(x))\}$ | $\{\neg Q(f(x))\}$ | 0 |
| $\{P(x), P(f(x))\}$ | $\{\neg P(y), Q(y,z)\}$ | 2 |

# Lifting-Lemma

A ground instance of a literal $L$ is a literal $L\mathrm{sub}$ that contains no variables.

A ground instance of a clause $K = \{L_1, \ldots, L_n\}$ is a clause $K\mathrm{sub} = \{L_1\mathrm{sub}, \ldots, L_n\mathrm{sub}\}$, which contains no variables.

Example: $P(f(a), f(f(a)), g(a, b))$ is a ground instance of the literal $P(x, f(x), g(a, y))$.

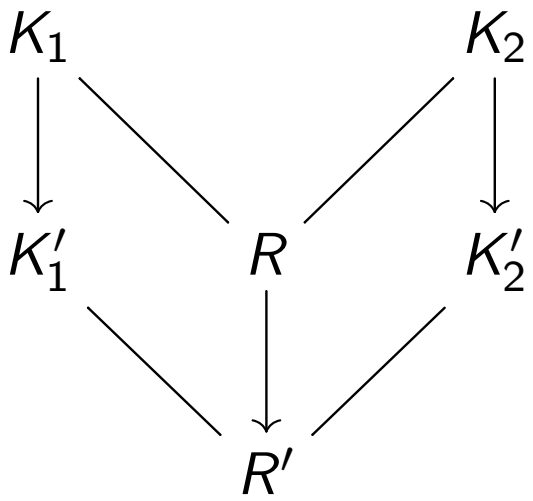
## Lifting-Lemma

Let $K_1, K_2$ be two predicate logic clauses and let $K_1', K_2'$ be two ground instances of these clauses that are satisfiably resolvable and yield the resolvent $R'$.
Then there exists a predicate logic resolvent $R$ of $K_1, K_2$, such that $R'$ is a ground instance of $R$.

# Lifting-Lemma

Visualization of the

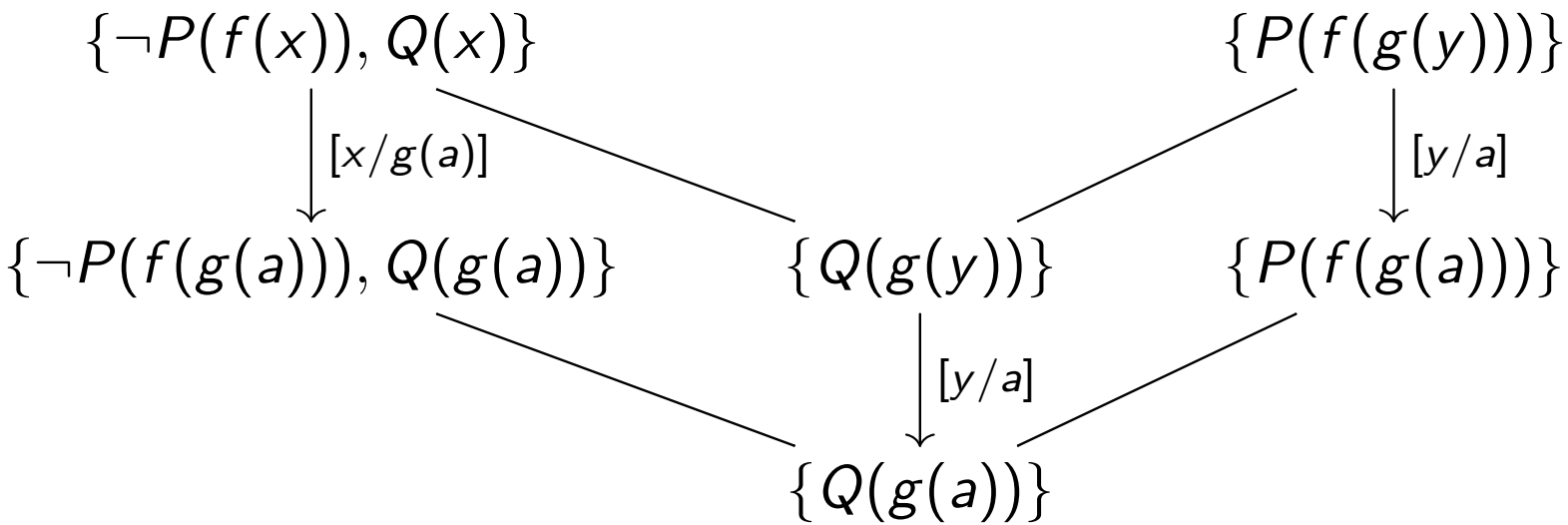
Lifting-Lemma:




—: Resolution

→: Substitution

**Example:**



# Proof of the Lifting-Lemma

**Proof:**

Let $s_1$ and $s_2$ be variable renamings such that $K_1 s_1$ and $K_2 s_2$ have no common variables.

$K_i'$ is a ground instance of $K_i$. $\Longrightarrow$ $K_i'$ is a ground instance of $K_i s_i$.

Let $\mathrm{sub}_i$ be a substitution such that $K_i' = K_i s_i \, \mathrm{sub}_i$.

Assuming without loss of generality for $i \in \{1, 2\}$:

1. $\mathrm{Def}(\mathrm{sub}_i)$ is the set of variables occurring in $K_i s_i$.
2. For all $x \in \mathrm{Def}(\mathrm{sub}_i)$, the term $\mathrm{sub}_i(x)$ contains no variables (i.e., $\mathrm{sub}_i$ is a ground substitution).

From (1), it follows in particular that $\mathrm{Def}(\mathrm{sub}_1) \cap \mathrm{Def}(\mathrm{sub}_2) = \emptyset$.

Let $\mathrm{sub} = \mathrm{sub}_1 \mathrm{sub}_2 = \mathrm{sub}_2 \mathrm{sub}_1$.

It holds that $K_i' = K_i s_i \, \mathrm{sub}_i = K_i s_i \, \mathrm{sub}$.

By assumption, $R'$ is a propositional resolvent of $K_1'$ and $K_2'$.

# Proof of the Lifting-Lemma

Thus, there exist $L \in K_1' = K_1 s_1 \,\mathrm{sub}$ and $\overline{L} \in K_2' = K_2 s_2 \,\mathrm{sub}$ such that

$$R' = (K_1' \setminus \{L\}) \cup (K_2' \setminus \{\overline{L}\}).$$

Let $L_1, \ldots, L_m \in K_1 s_1$ (with $m \geq 1$) be all literals from $K_1 s_1$ satisfying

$$L = L_1 \mathrm{sub} = \cdots = L_m \mathrm{sub}.$$

Let $L_1', \ldots, L_n' \in K_2 s_2$ (with $n \geq 1$) be all literals from $K_2 s_2$ satisfying

$$\overline{L} = L_1' \mathrm{sub} = \cdots = L_n' \mathrm{sub}.$$

Thus, $\mathrm{sub}$ is a unifier of the literal set

$$\mathbf{L} = \{\overline{L_1}, \ldots, \overline{L_m}, L_1', \ldots, L_n'\}$$

and the clauses $K_1$ and $K_2$ are predicate-logically resolvable.

Let $\mathrm{sub}_0$ be a most general unifier of $\mathbf{L}$.

# Proof of the Lifting-Lemma

Then,

$$R = ((K_1 s_1 \setminus \{L_1, \ldots, L_m\}) \cup (K_2 s_2 \setminus \{L'_1, \ldots, L'_n\}))\mathrm{sub}_0$$

is a predicate-logical resolvent of $K_1$ and $K_2$.

Since $\mathrm{sub}_0$ is the most general unifier of $\mathbf{L}$ and $\mathrm{sub}$ is a unifier of $\mathbf{L}$, there exists a substitution $s$ such that $\mathrm{sub}_0\, s = \mathrm{sub}$. It follows that

$$\begin{aligned}
R' &= (K'_1 \setminus \{L\}) \cup (K'_2 \setminus \{\overline{L}\}) \\
&= (K_1 s_1\, \mathrm{sub} \setminus \{L\}) \cup (K_2 s_2\, \mathrm{sub} \setminus \{\overline{L}\}) \\
&= (K_1 s_1\, \mathrm{sub} \setminus \{L_1\mathrm{sub}, \ldots, L_m\mathrm{sub}\}) \cup (K_2 s_2\, \mathrm{sub} \setminus \{L'_1\mathrm{sub}, \ldots, L'_n\mathrm{sub}\}) \\
&= \Big((K_1 s_1 \setminus \{L_1, \ldots, L_m\}) \cup (K_2 s_2 \setminus \{L'_1, \ldots, L'_n\})\Big)\,\mathrm{sub} \\
&= \Big((K_1 s_1 \setminus \{L_1, \ldots, L_m\}) \cup (K_2 s_2 \setminus \{L'_1, \ldots, L'_n\})\Big)\,\mathrm{sub}_0 s \\
&= R\, s
\end{aligned}$$

Thus, it has been shown that $R'$ is a ground instance of $R$. □

# Resolution Theorem

## Resolution Theorem of Predicate Logic

Let $F$ be a formula in Skolem form with a matrix $F^*$ in **CNF**. Then: $F$ is unsatisfiable if and only if $\square \in Res^*(F^*)$.

For the proof of the resolution theorem, we need the following concept:

For a formula $H$ with free variables $x_1, \ldots, x_n$, we denote

$$\forall H = \forall x_1 \forall x_2 \cdots \forall x_n H$$

as its universal closure.

# Lemmas on Universal Closure

## Lemma

Let $F$ be a formula in Skolem form, whose matrix $F^*$ is in **CNF**. Then:

$$F \equiv \forall F^* \equiv \bigwedge_{K \in F^*} \forall K$$

**Proof:** Since $F$ is a formula (i.e., has no free variables), we have $F = \forall F^*$.

Since $F^*$ is in **CNF**, it holds that

$$F^* \equiv \bigwedge_{K \in F^*} K.$$

The lemma thus follows from the equivalence
$\forall y(G \wedge H) \equiv \forall y G \wedge \forall y H$. □

**Example:**

$$\begin{aligned} F^* &= P(x,y) \wedge \neg Q(y,x) \\ F &\equiv \forall x \forall y (P(x,y) \wedge \neg Q(y,x)) \equiv \forall x \forall y P(x,y) \wedge \forall x \forall y (\neg Q(y,x)) \end{aligned}$$

# Lemmas on Universal Closure

**Lemma**

Let $R$ be the resolvent of two clauses $K_1$ and $K_2$. Then $\forall R$ is a consequence of $\forall K_1 \ \wedge \ \forall K_2$.

**Proof:**

Let $\mathcal{A}$ be a model of $\forall K_1$ and $\forall K_2$: $\mathcal{A}(\forall K_1) = \mathcal{A}(\forall K_2) = 1$.

Let

$$R = ((K_1 s_1 \setminus \{L_1, \ldots, L_m\}) \cup (K_2 s_2 \setminus \{L'_1, \ldots, L'_n\}))\mathrm{sub}$$

where $L_1, \ldots, L_m \in K_1 s_1$, $L'_1, \ldots, L'_n \in K_2 s_2$, and $\mathrm{sub}$ is most general unifier of

$$\mathbf{L} = \{\overline{L_1}, \ldots, \overline{L_m}, L'_1, \ldots, L'_n\}.$$

Let $L = \overline{L_1}\,\mathrm{sub} = \cdots = \overline{L_m}\,\mathrm{sub} = L'_1\,\mathrm{sub} = \cdots = L'_n\,\mathrm{sub}$. Then we have

$$(K_1 s_1\,\mathrm{sub} \setminus \{\overline{L}\}) \cup (K_2 s_2\,\mathrm{sub} \setminus \{L\}) \subseteq R.$$

# Lemmas on Universal Closure

Assume that $\mathcal{A}(\forall R) = 0$.

Then there exists a structure $\mathcal{A}'$ such that:

- $\mathcal{A}'$ is identical to $\mathcal{A}$ except for the values $I_{\mathcal{A}'}(x)$ for the variables $x$ appearing in $R$.
- $\mathcal{A}'(R) = 0$.

Thus, it follows that

$$\mathcal{A}'(K_1 s_1 \,\mathrm{sub} \setminus \{\overline{L}\}) = \mathcal{A}'(K_2 s_2 \,\mathrm{sub} \setminus \{L\}) = 0. \tag{17}$$

From $\mathcal{A}(\forall K_1) = \mathcal{A}(\forall K_2) = 1$, we conclude

$$\mathcal{A}'(K_1 s_1 \,\mathrm{sub}) = \mathcal{A}'(K_2 s_2 \,\mathrm{sub}) = 1. \tag{18}$$

Equations (17) and (18) together imply: $\mathcal{A}'(L) = \mathcal{A}'(\overline{L}) = 1$.
Contradiction! □

# Proof of the Resolution Theorem

**Proof of the Resolution Theorem:**

**(A) Correctness:** If $\square \in Res^*(F^*)$, then $F$ is unsatisfiable.

Assume $\square \in Res^*(F^*)$.

From the lemmas just proven, it follows that $\square = \forall\square$ is a consequence of $\bigwedge_{K \in F^*} \forall K \equiv F$.

Since $\square$ has no model, $F$ cannot have a model either.

**(B) Completeness:** If $F$ is unsatisfiable, then $\square \in Res^*(F^*)$.

Assume $F$ is unsatisfiable.

# Proof of the Resolution Theorem

From the basic resolution theorem, it follows that there is a sequence of clauses $K'_1, \ldots, K'_n$ with the following properties:

- $K'_n$ is the empty clause.
- For $i = 1, \ldots, n$ it holds:
  - $K'_i$ is a ground instance of a clause $K \in F^*$, i.e., $K'_i = K[y_1/t_1] \ldots [y_k/t_k]$ with $t_i \in D(F)$.
  - **or** $K'_i$ is (propositional) resolvent of two clauses $K'_a, K'_b$ with $a < i$ and $b < i$.

For all $i \in \{1, \ldots, n\}$, we provide a clause $K_i$ such that $K'_i$ is a ground instance of $K_i$, and $(K_1, \ldots, K_n)$ is a predicate logical resolution derivation of the empty clause $K_n = \square$ from the clauses in $F^*$.

Consider an $i \in \{1, \ldots, n\}$ and let $K_1, \ldots, K_{i-1}$ be already constructed.

# Proof of the Resolution Theorem

**Case 1:** $K'_i$ is a ground instance of a clause $K \in F^*$.

Define $K_i = K$.

**Case 2:** $K'_i$ is a propositional resolvent of two clauses $K'_a, K'_b$ with $a < i$ and $b < i$.

From the lifting lemma, we obtain a predicate logical resolvent $R$ of $K_a$ and $K_b$, such that $K'_i$ is a ground instance of $R$.

Define $K_i = R$. □

# Example I

Is the set of clauses

$$\{\{P(f(x))\},\ \{\neg P(x), Q(x, f(x))\},\ \{\neg Q(f(a), f(f(a)))\}\}$$

i.e., the statement

$$\forall x\big(P(f(x)) \wedge (\neg P(x) \vee Q(x, f(x))) \wedge \neg Q(f(a), f(f(a)))\big)$$

unsatisfiable?

# Example I

Is the set of clauses

$$\{\{P(f(x))\},\ \{\neg P(x), Q(x, f(x))\},\ \{\neg Q(f(a), f(f(a)))\}\}$$

i.e., the statement

$$\forall x\big(P(f(x)) \wedge (\neg P(x) \vee Q(x, f(x))) \wedge \neg Q(f(a), f(f(a)))\big)$$

unsatisfiable?

Yes, here is a resolution derivation of the empty clause:

$\{P(f(x))\}$ and $\{\neg P(x), Q(x, f(x))\}$ yield the resolvent $\{Q(f(x), f(f(x)))\}$

$\{Q(f(x), f(f(x)))\}$ and $\{\neg Q(f(a), f(f(a)))\}$ yield the resolvent $\{\} = \square$.

# Example II

We consider the following set of clauses
(example from Schöning's book):

$$\begin{aligned} F \quad = \quad & \{\{\neg P(x), Q(x), R(x, f(x))\}, \{\neg P(x), Q(x), S(f(x))\}, \{T(a)\}, \\ & \{P(a)\}, \{\neg R(a, x), T(x)\}, \{\neg T(x), \neg Q(x)\}, \{\neg T(x), \neg S(x)\}\} \end{aligned}$$

# Refinement of Resolution (Outlook)

**Problems** with predicate logical resolution:

- Too many choices
- Still too many dead ends
- Combinatorial explosion of the search space

**Solutions:**

Strategies and Heuristics: Prohibiting certain resolution steps, thereby restricting the search space

**Caution:** Completeness must not be lost in the process!